The Art of Differentiating Computer Programs[1]

Algorithmic Differentiation – Why and How?

Uwe Naumann

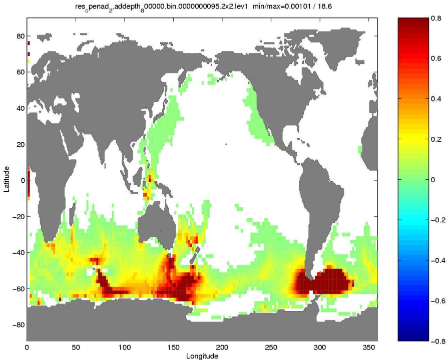
Software and Tools for Computational Engineering
RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de

and

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

[1] See also upcoming SIAM book

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 :-)

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

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

Approximate Tangent-Linear Code (Finite Differences)

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

Adjoint Code

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

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

$$\mathbf{y}^{(1)} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \quad \Rightarrow \text{ cheap directional derivatives}$$

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

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

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

$$\mathbf{x}_{(1)} = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \quad \Rightarrow \text{ cheap gradients}$$

Consider an implementation [2] of the pde-constrained optimization problem $\min_{u(x,0)} J(u, u^{\text{obs}})$ where

$$J(u, u^{\text{obs}}) \equiv \int_\Omega \Big(u(x,T) - u^{\text{obs}}(x) \Big)^2 dx$$

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 $x \in \Gamma$.

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

Solution requires the gradient of the Lagrangian

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

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

- Lagrangian in `f.c` → `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 ...

Disclaimer

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

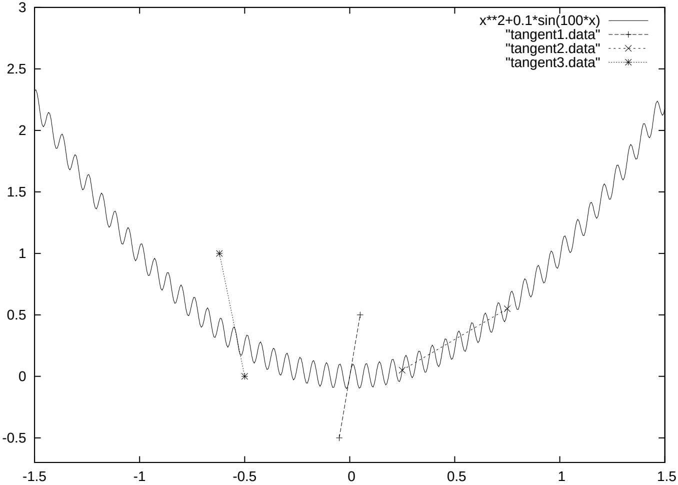
We differentiate what you implemented – not what you possibly intended to implement.

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

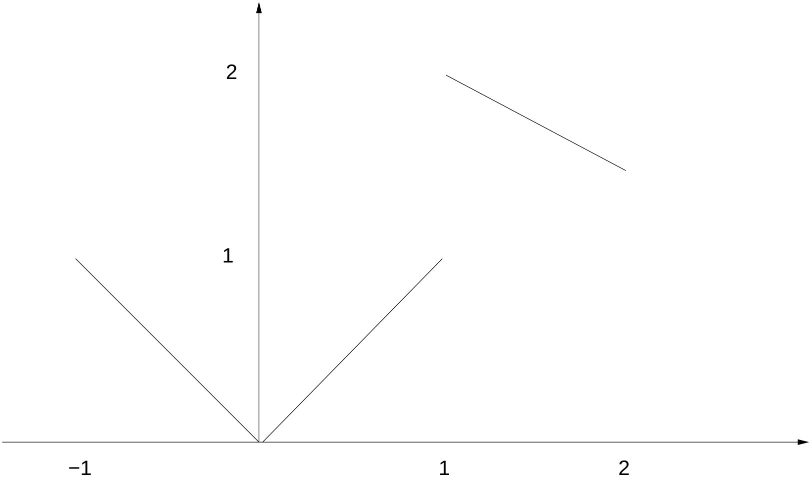
Fact: AD (also know as Automatic Differentiation) is not fully automatic and never will be except for simple cases.

Is it derivatives you want?

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



Do derivatives exist?



First Derivatives in Newton's Algorithm

Given: Implementation $\mathbf{y} = F(\mathbf{x})$ of the residual $\mathbf{y} \in \mathbb{R}^n$ of a system of nonlinear equations and a starting point $\mathbf{x}^0 \in \mathbb{R}^n$

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

Solution: Newton algorithm

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

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

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

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 $\nabla F(\mathbf{x}^k)$) are used (matrix-free preconditioners?).

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

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

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

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

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

from some suitable start value $\mathbf{x}^0$ 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 $\nabla F(\mathbf{x}^k)$. Improved quasi-Newton methods, such as BFGS, are also based on $\nabla F(\mathbf{x}^k)$.

First Derivatives in the NAG Library

- systems of nonlinear equations (c05ubc); user provides

```
void j_f(Integer n, const double x[],
         double f[], double j[], ...);
```

- unconstrained nonlinear optimization (e04dgc); user provides

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

- unconstrained nonlinear least squares (e04gbc); user provides

```
void j_f(Integer m, Integer n, const double x[],
         double f[], double j[], ...);
```

Back To The Race

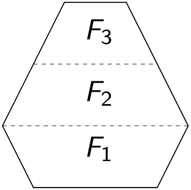
- Gradient by tangent-linear Lagrangian took several minutes.
- Gradient by adjoint Lagrangian takes a few seconds
  - `g++ a1_main.cpp -o a1_main`
  - `time ./a1_main`
- `diff t1.out a1.out`
- Adjoint for more complex problems / codes ... nontrivial :-)

AD in Action

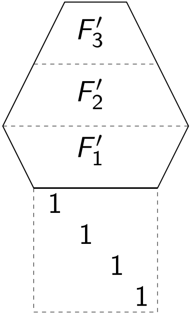
Algorithmic Differentiation of $F = \circ_{i=1}^{k} F_i$ where $F_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{m_i}$

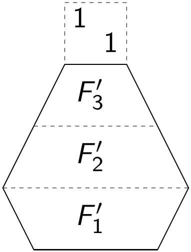
	Forward Mode	Reverse Mode
	$F' = \prod \ldots$	

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



$\mathbf{x}$

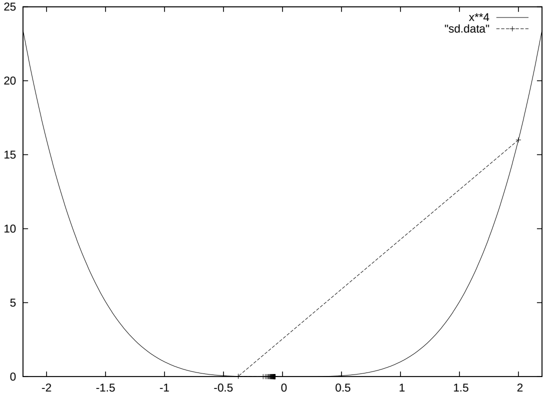




$F' = \prod \ldots$

$(F_3 \circ F_2 \circ F_1)(\mathbf{x})$	$F_3'(F_2'(F_1' \cdot \mathbf{x}^{(1)}))$	$(F_1')^T((F_2')^T((F_3')^T \cdot \mathbf{y}_{(1)}))$
F_i' at $Cost(F)$	F' at $n \cdot O(Cost(F))$	F' at $m \cdot O(Cost(F))$

E.g., minimization of $y = f(\mathbf{x}) = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ by Steepest Descent:



$|\nabla f| < 10^{-4}$ after 9 iterations; $|\nabla f| < 10^{-10}$ after 1461997 iterations

... implemented as

```
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 $\rightarrow O(n) \cdot Cost(F)$
- tangent-linear code $\rightarrow O(n) \cdot Cost(F)$
- adjoint code $\rightarrow O(1) \cdot Cost(F)$

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((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

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

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

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{y}^{(1)} &= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \quad . \end{aligned}$$

The signature of the resulting tangent-linear subroutine becomes

```
void t1_f(int n, double* x, double* t1_x,
                 double& y, double& t1_y)     .
```

1st-Order Adjoint Code

We transform the given implementation

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

of the function $\mathbf{y} = F(\mathbf{x})$ into adjoint code computing

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{x}_{(1)} &= (\nabla F(\mathbf{x}))^T \cdot \mathbf{y}_{(1)} \quad . \end{aligned}$$

The signature of the resulting adjoint subroutine becomes

```
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;
}
```

Driver for 1st-Order Tangent-Linear Code

```
...
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;
}
...
```

1st-Order Adjoint Code

```
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();
}
```

```
...
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
$n = 5 \cdot 10^4$, $h = 10^{-8}$

fd

```
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
...
```

t1/a1

```
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
...
```

Higher Derivative Codes

Second Derivative Codes for $f : \mathbb{R}^n \rightarrow \mathbb{R} : y = f(\mathbf{x})$

Second-Order Tangent-Linear Code

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

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

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

Second-Order Adjoint Code

$$\mathbf{x}_{(1)}^{(2)} = y_{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

Accumulation of Hessian $\nabla^2 f \in \mathbb{R}^{n \times n}$ ...

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

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

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

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

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

$$\mathbf{x}_{(1)}^{(2)} = y_{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

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

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

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

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

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

from some suitable start value $\mathbf{x}^0$ and with step length $\alpha_k > 0$ for $B_k = \nabla^2 F(\mathbf{x}^k) \in \mathbb{R}^{n \times n}$. 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

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

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 $\mathbf{x}^* \in \mathbb{R}^n$.

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

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

using Newton algorithm.

The iteration proceeds as

$$(\mathbf{x}_{k+1}, \lambda_{k+1}) = (\mathbf{x}_k, \lambda_k) + \alpha_k \cdot (\Delta_k^{\mathbf{x}}, \Delta_k^{\lambda})$$

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

$$\begin{bmatrix} \nabla_{\mathbf{xx}}\mathcal{L}(\mathbf{x}_k, \lambda_k) & -(\nabla c(\mathbf{x}_k))^T \\ \nabla c(\mathbf{x}_k) & 0 \end{bmatrix} \cdot \begin{bmatrix} \Delta_k^{\mathbf{x}} \\ \Delta_k^{\lambda} \end{bmatrix} = \begin{bmatrix} (\nabla c(\mathbf{x}_k))^T \cdot \lambda_k - \nabla F(\mathbf{x}_k) \\ -c(\mathbf{x}_k) \end{bmatrix}$$

Matrix-free implementations of Krylov-subspace methods compute the residual of the constraints ($c(\mathbf{x}_k)$), tangent projections of the Hessian of the Lagrangian ($< \nabla_{\mathbf{xx}}\mathcal{L}(\mathbf{x}_k, \lambda_k), \mathbf{v} >$), the gradient of the objective ($\nabla F(\mathbf{x}_k)$), and tangent and adjoint projections of the Jacobian of the constraints ($< \nabla c(\mathbf{x}_k), \mathbf{v} >$ and $< \mathbf{w}, \nabla c(\mathbf{x}_k) >$).

- unconstrained or bound-constrained minima of twice continuously differentiable nonlinear functions (e04lbc); user provides

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

and

```
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

$$F^{(1,2,3)}(\mathbf{x},\mathbf{x}^{(1)},\mathbf{x}^{(2)},\mathbf{x}^{(3)}) = < \nabla^3 F(\mathbf{x}),\mathbf{x}^{(1)},\mathbf{x}^{(2)},\mathbf{x}^{(3)} >, \quad \mathbf{x}^{(i)} \in \mathbb{R}^n$$

- Fourth-order adjoint model

$$F_{(1)}^{(2,3,4)}(\mathbf{x},\mathbf{y}_{(1)},\mathbf{x}^{(2)},\mathbf{x}^{(3)},\mathbf{x}^{(4)}) = < \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}),\mathbf{x}^{(2)},\mathbf{x}^{(3)},\mathbf{x}^{(4)} >$$

$$\mathbf{x}^{(i)} \in \mathbb{R}^n, \;\; \mathbf{y}_{(1)} \in \mathbb{R}^m$$

Given: $y = F(x)$ with $F : \mathbb{R} \to \mathbb{R}$ (for notational simplicity) and expected value μ_x and variance σ_x of x.

Wanted: Estimates for expected value μ_y and variance σ_y of y.

Solution: Method of Moments gives

$$\mu_y = F(\mu_x) + \frac{F''(\mu_x)}{2} \cdot \sigma_x^2 \qquad \text{(approximate mean)}$$

$$\begin{aligned}\sigma_y^2 = {} & F'(\mu_x)^2\, \sigma_x^2 + F'(\mu_x)\, F''(\mu_x)\, S_x\, \sigma_x^3 \\ & + \frac{1}{4} \left(F''(\mu_x)\right)^2 (K_x - 1)\, \sigma_x^4\end{aligned} \qquad \text{(approximate variance)}$$

for given initial mean μ_x, variance σ_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 $\min_{\mathbf{x}(s,t),\ s\in\Gamma} F(\mathbf{x})$ subject to $c(\mathbf{x}) = 0$ with objective

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

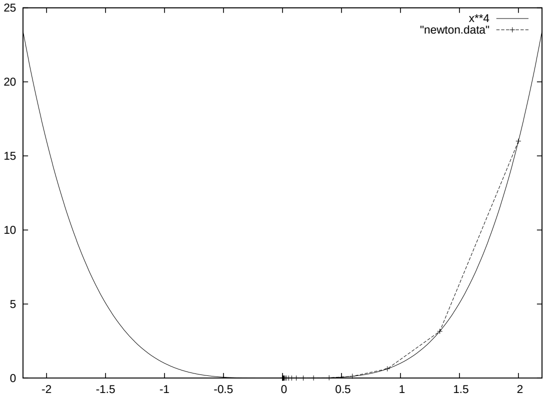
(measured) initial condition $\mathbf{x}(s,0)$ and boundary condition $\mathbf{x}(s,t)$ for $s \in \Gamma$.

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.

Example

E.g., minimization of $y = f(\mathbf{x}) = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$ by Newton's method



$|\nabla f| < 10^{-4}$ after 7 iterations; $|\nabla f| < 10^{-10}$ after 22 iterations

... implemented as

```
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 t2_t1_f from t1_f and computation of Hessian
   1.3 t2_a1_f from a1_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;
}
```

2nd-Order Tangent-Linear Code

We transform t1_f into second-order tangent-linear code computing

$$
\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{y}^{(2)} &=< \nabla F(\mathbf{x}), \mathbf{x}^{(2)} > \\
\mathbf{y}^{(1)} &=< \nabla F(\mathbf{x}), \mathbf{x}^{(1)} > \\
\mathbf{y}^{(1,2)} &=< \nabla F(\mathbf{x}), \mathbf{x}^{(1,2)} > + < \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} > \quad .
\end{aligned}
$$

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

```
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;
}
```

```
...
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

$$
\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{y}^{(2)} &= < \nabla F(\mathbf{x}), \mathbf{x}^{(2)} > \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + < \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) > \\
\mathbf{x}_{(1)}^{(2)} &= \mathbf{x}_{(1)}^{(2)} + < \mathbf{y}_{(1)}^{(2)}, \nabla F(\mathbf{x}) > + < \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} > \quad .
\end{aligned}
$$

The signature of the second-order adjoint subroutine becomes

```
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);
```

```
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;
```

2nd-Order Adjoint Code
Reverse Section

```
  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` :-(
- $n = 10^3$
  - sofd: 4.1s; t2_t1: 3.5s; t2_a1: 1.4s
  - `gvimdiff t2_t1_1000.out t2_a1_1000.out` :-)
  - `gvimdiff sofd_1000.out t2_t1_1000.out` :-((((
- $n = 2 \cdot 10^3$
  - sofd: 26.9s; t2_t1: 22.1s; t2_a1: 5.7s
- $n = 3 \cdot 10^3$
  - sofd: 85.2s; t2_t1: 69.7s; t2_a1: 12.9s

Quality of 2nd-order FD $n = 2000,\ h = 10^{-6}$

sofd

```
h[0][0]=3958.12
h[1][0]=0
h[1][1]=3958.12
h[2][0]=116.415
h[2][1]=0
h[2][2]=4190.95
h[3][0]=0
h[3][1]=116.415
h[3][2]=232.831
h[3][3]=4074.54
...
```

t2t1/t2a1

```
h[0][0]=4009.29
h[1][0]=4.32242
h[1][1]=4003.63
h[2][0]=-3.32917
h[2][1]=-1.79876
h[2][2]=4002.68
h[3][0]=-7.91994
h[3][1]=-4.27916
h[3][2]=3.29586
h[3][3]=4009.13
...
```

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

$$\frac{\text{runtime of adjoint}}{\text{runtime of original simulation}}$$

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!