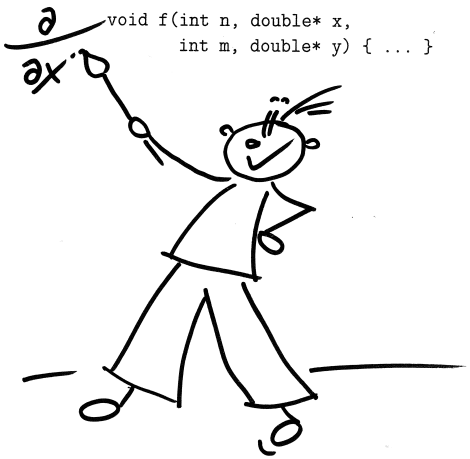
Algorithmic Differentiation Primer

Part II: Overloading with dco/c++



Uwe Naumann

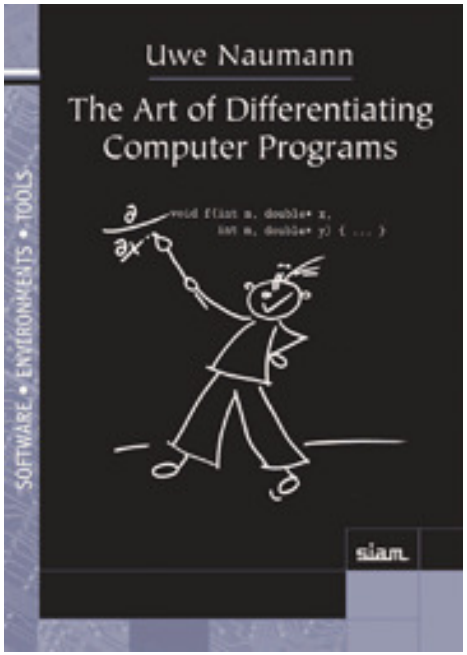
Software and Tools for Computational Engineering
Computer Science, RWTH Aachen University
Aachen, Germany
naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de

and

The Numerical Algorithms Group Ltd.
Oxford, United Kingdom
Uwe.Naumann@nag.co.uk
www.nag.co.uk

Contents

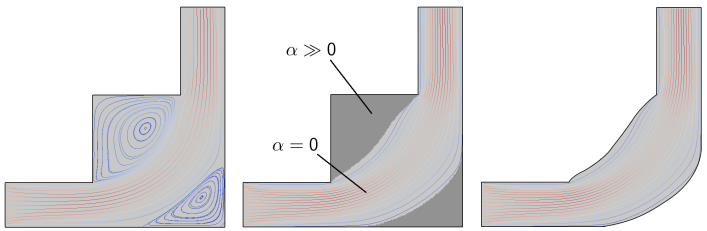
Course is based on ...

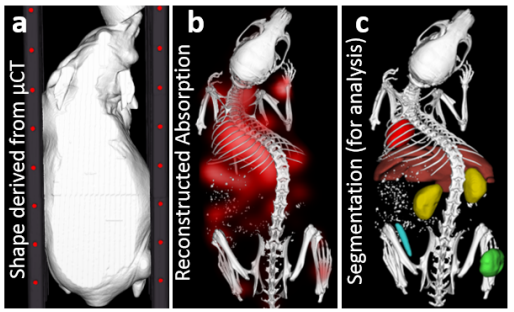


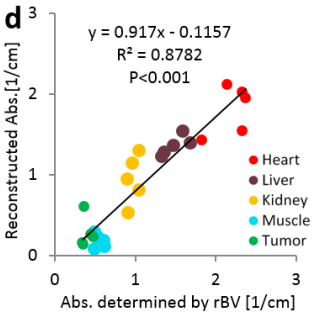
Motivation

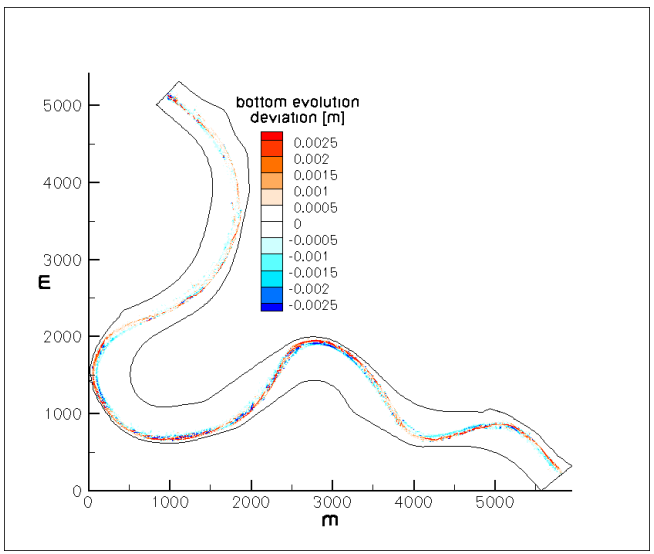
Motivation: Parameter Sensitivities

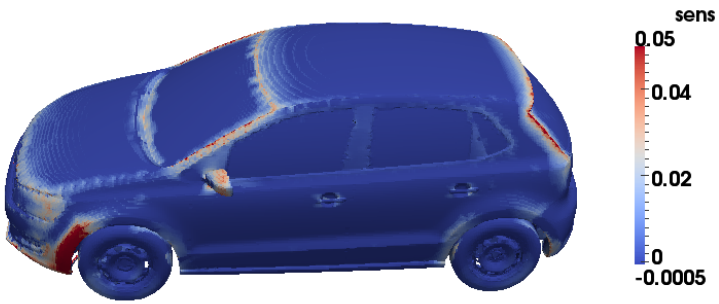
for Large-Scale Nonlinear Numerical Simulations









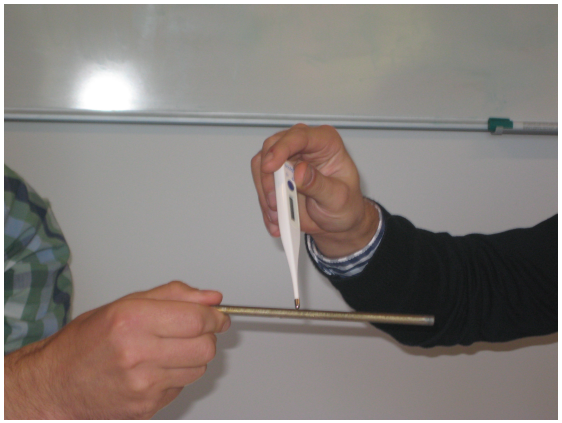


Some recently published work ...

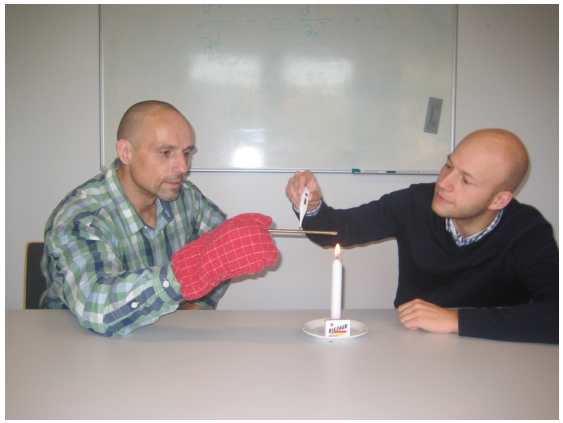
- F. Gremse, U.N., et al.: *Absorption Reconstruction Improves Biodistribution Assessment of Fluorescent Nanoprobes Using Hybrid Fluorescence-mediated Tomography*. Theranostics 4(10):960–971, Ivyspring International Publisher 2014.
- A. Vlasenko, U.N., et al.: *Estimation of Data Assimilation Error: A Shallow-Water Model Study*. Monthly Weather Review 142:2502–2520, 2014.
- M. Towara and U.N.: *A Discrete Adjoint Model for OpenFOAM.* Procedia Computer Science 18:429-438, Elsevier 2013.
- U. Merkel, U.N., et al.: *Reverse engineering of initial and boundary conditions with Telemac and algorithmic differentiation*. WASSERWIRTSCHAFT 103(12):22–27, Springer 2013.
- R. Hannemann-Tamás, U.N., et al.: *First- and Second-Order Parameter Sensitivities of a Metabolically and Isotopically Non-Stationary Biochemical Network Model*, Electronic Proceedings of the 9th International Modelica Conference, Modelica Association 2012.

Motivation: Cheap Gradients

Diffusion





$T = T(t, x, c(x)) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is given as the solution of the 1D diffusion equation

$$\frac{\partial T}{\partial t} = c(x) \cdot \frac{\partial^2 T}{\partial x^2}$$

over the domain $\Omega = [0, 1]$ and with initial condition $T(0, x) = i(x)$ for $x \in \Omega$ and Dirichlet boundary $T(t, 0) = b_0(t)$ and $T(t, 1) = b_1(t)$ for $t \in [0, 1]$.

The numerical solution is based on a central finite difference / implicit (backward) Euler integration scheme that exploits linearity of the residual r in T (single evaluation of constant Jacobian $\frac{\partial r}{\partial T}$ and factorization).

We aim to analyze sensitivities of the predicted $T(x, c(x))$ with respect to $c(x)$ or even calibrate the uncertain $c(x)$ to given observations $O(x)$ for $T(x, c(x))$ at time $t = 1$ by solving the (possibly constrained) least squares problem

$$\min_{c(x)} f(c(x), x) \quad \text{where } f \equiv \int_\Omega \left(T(1, x, c(x)) - O(x)\right)^2 dx.$$

Assuming a spatial discretization of Ω with n grid points we require

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial T(1)} \cdot \frac{\partial T(1)}{\partial c} \in \mathbb{R}^n \quad (\rightarrow \text{scalar adjoint integration at } O(1))$$

and possibly

$$\frac{\partial^2 f}{\partial c^2} \in \mathbb{R}^{n \times n} \quad (\rightarrow \text{2nd-order vector adjoint integrations at } O(n))$$

...
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

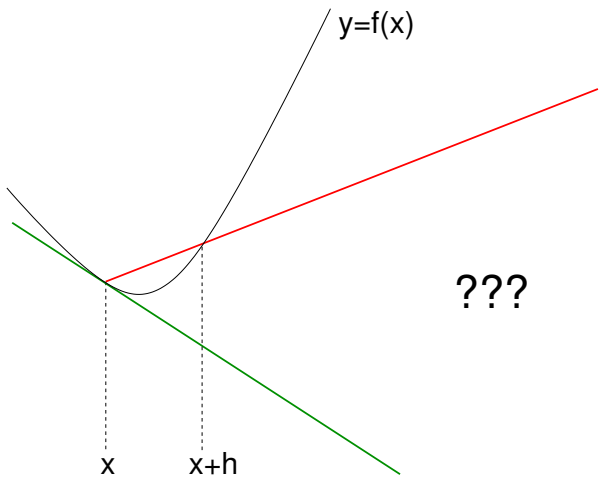
Naive application of Algorithmic Differentiation (AD) tool `dco/c++` yields:

$\frac{\partial f}{\partial c}$ (n=300, m=50)

	central FD	adjoint AD
run time (s)	15.2	0.7

$\frac{\partial^2 f}{\partial c^2}$ (n=100, m=50)

	central FD	adjoint AD
run time (s)	63.6	3.9



???

... while ignoring accuracy for the time being ...

We aim to price a simple European Call option written on an underlying $S = S(t) : \mathbb{R}^+ \to \mathbb{R}^+$, described by the SDE

$$dS(t) = S(t) \cdot r \cdot dt + S(t) \cdot \sigma(t, S(t)) \cdot dW(t)$$

with time $t \geq 0$, maturity $T > 0$, strike $K > 0$, constant interest rate $r > 0$, volatility $\sigma = \sigma(t, S) : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$, and Brownian motion $W = W(t) : \mathbb{R}^+ \to \mathbb{R}^+$.
The value of a European call option driven by S is given by the expectation

$$V = \mathbb{E} e^{-r \cdot T} \cdot (S(T) - K)^+$$

for given interest r, strike K, and maturity T. It can be evaluated using Monte Carlo simulation.

Alternatively, by the Feynman-Kac theorem, V can be computed by solving the PDE

$$\frac{\partial V}{\partial t} + r \cdot S \cdot \frac{\partial V}{\partial S} + \frac{1}{2} \cdot S^2 \cdot \sigma \cdot \frac{\partial^2 V}{\partial S^2} - r \cdot V = 0$$

for $t < T$, $S > 0$, and initial (actually, terminal) condition

$$V(T, S) = (S - K)^+$$

and asymptotic boundary conditions

$$\lim_{S \to \infty} V = e^{-r \cdot (T-t)} \cdot (S(t) - K)$$
$$\lim_{S \to 0} V = 0.$$

We compare the computation of the gradient of the option price $V \in \mathbb{R}^+$ wrt. the problem parameters $(K, S(0), r, T$, and the variable number of parameters of our "made up" local volatility surface) by finite differences with adjoint code based on dco/c++ ...

PDE Scenario:

- size of gradient: $n = 222$
- asset price grid points: 1000
- time grid points: 360

	primal	central FD	adjoint AD
Runtime (sec.)	0.5	175	6

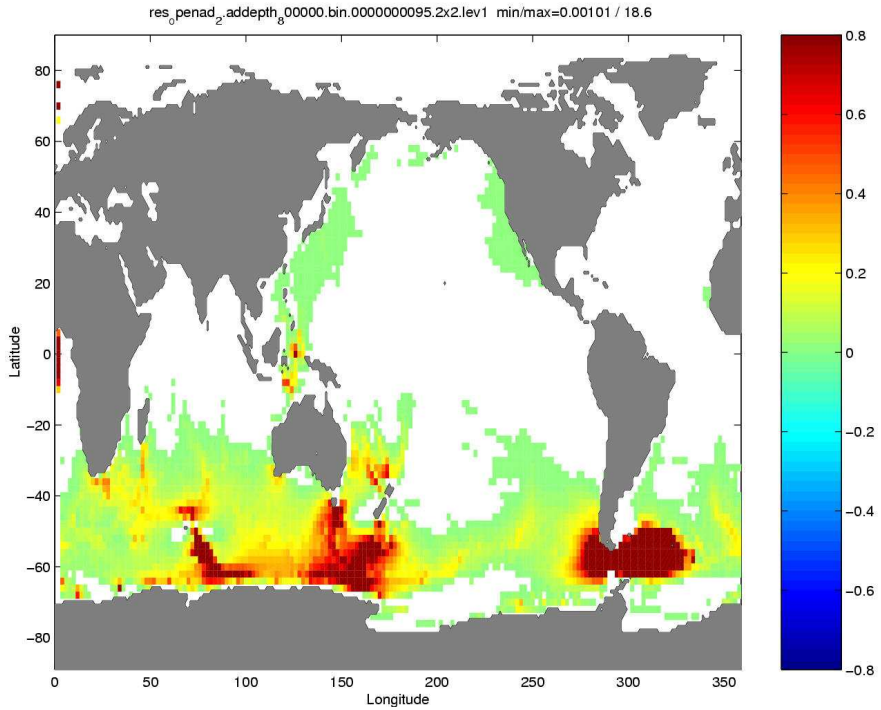
(: −() prohibitive run time of FD

(: −)) gradient by "white-box" [:-)] dco/c++ adjoint at cost of approx. $12 \sim O(1)$ function evaluations (as opposed to $350 \sim O(n)$)

(: −() low-quality sensitivity information by FD (nan ...)

(: −)) sensitivity information with machine accuracy by dco/c++ adjoint

Nice to have?



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 tangent computation / 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 :-)

We consider implementations of multivariate vector functions

$$F : D_F \subseteq \mathbb{R}^n \to I_F \subseteq \mathbb{R}^m : \mathbf{y} = F(\mathbf{x})$$

as computer programs. F assumed to be (k times) continuously differentiable over their entire (open) domain[1] implying the existence of the Jacobian (Hessian etc.)

$$\nabla F(\mathbf{x}) \equiv \frac{\partial F}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$$

the individual columns of which can be approximated at all points $\mathbf{x}^* \in D_F$ by (forward, backward, central) finite difference quotients as follows:

$$\nabla F(\mathbf{x}^*) \approx_1 \left(\frac{F(\mathbf{x}^* + h \cdot \mathbf{e}_i) - F(\mathbf{x}^*)}{h} \right)_{i=0}^{n-1} \approx_1 \left(\frac{F(\mathbf{x}^*) - F(\mathbf{x}^* - h \cdot \mathbf{e}_i)}{h} \right)_{i=0}^{n-1}$$
$$\approx_2 \left(\frac{F(\mathbf{x}^* + h \cdot \mathbf{e}_i) - F(\mathbf{x}^* - h \cdot \mathbf{e}_i)}{2 \cdot h} \right)_{i=0}^{n-1}$$

[1] to be relaxed later

Obviously,

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

with element-wise multiplication and division and where $\mathbf{h} \in \mathbb{R}^n$ contains suitable perturbations[2] for the x_i, $i = 0, \ldots, n-1$.

Alternatively,

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

or

$$\nabla F(\mathbf{x}^*) \cdot \mathbf{x}^{(1)} \approx \frac{F(\mathbf{x}^* + \mathbf{h} \cdot \mathbf{x}^{(1)}) - F(\mathbf{x}^* - \mathbf{h} \cdot \mathbf{x}^{(1)})}{2 \cdot \mathbf{h}}$$

[2] e.g. $h_i = \sqrt{\epsilon} \cdot |x_i|$ with machine epsilon ϵ

Accuracy of Forward/Backward Finite Differences

W.l.o.g., let $m = 1$. For $x = x^0 + h$ we get

$$f(x^0 + h) = f(x^0) + \frac{\partial f}{\partial x}(x^0) \cdot h + \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x^0) \cdot h^2 + \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x^0) \cdot h^3 + \ldots$$

and similarly for $x = x^0 - h$

$$f(x^0 - h) = f(x^0) - \frac{\partial f}{\partial x}(x^0) \cdot h + \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x^0) \cdot h^2 - \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x^0) \cdot h^3 + \ldots \quad .$$

Truncation after the respective first derivative terms yields scalar univariate versions of forward and backward finite difference quotients, respectively. For $0 < h << 1$ the truncation error is dominated by the value of the h^2 term which implies that only accuracy up to the order of h ($= h^1$ and hence first-order accuracy) can be expected.

Accuracy of Central Finite Differences

Second-order accuracy ($\approx_2$) follows immediately from the previous Taylor expansions. Their subtraction yields

$$
\begin{aligned}
& f(x^0 + h) - f(x^0 - h) = \\
& \quad f(x^0) + \frac{\partial f}{\partial x}(x^0) \cdot h + \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x^0) \cdot h^2 + \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x^0) \cdot h^3 + \ldots - \\
& \quad (f(x^0) - \frac{\partial f}{\partial x}(x^0) \cdot h + \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x^0) \cdot h^2 - \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x^0) \cdot h^3 + \ldots) \\
& \quad = 2 \cdot \frac{\partial f}{\partial x}(x^0) \cdot h + \frac{2}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x^0) \cdot h^3 + \ldots \quad .
\end{aligned}
$$

Truncation after the first derivative term yields the scalar univariate version of the central finite difference quotient. For small values of h the truncation error is dominated by the value of the h^3 term which implies that only accuracy up to the order of h^2 (second-order accuracy) can be expected.

For example, let

$$y = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

be implemented in C++ as

```
void f(int n, double *x, double& y) {
  int i=0;
  while (i<n) {
    x[i]=x[i]*x[i];
    i=i+1;
  }
  i=0;
  while (i<n) {
    if (i==0)
      y=x[i];
    else
      y=y+x[i];
    i=i+1;
  }
  y=y*y;
}
```

We are looking for a routine `fg(...)` returning for a given vector `x` of length `n` the value `y` of `f` and its gradient `g`.

```
int main() {
  const int n=5;
  double x[n],y,g[n];
  cout.precision(15);
  for (int i=0;i<n;i++) x[i]=cos(i);
  fg(n,x,y,g);
  cout << y << endl;
  for (int i=0;i<n;i++) cout << g[i] << endl;
  return 0;
}
```

Live:

- implementation of `fg` for given

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

- run for increasing values of `n`

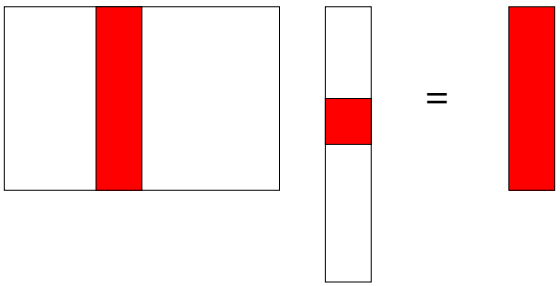
A forward finite difference approximation of the gradient can be computed alongside with the function value as follows:

```
void fg(int n, double *x, double &y, double *g) {
  double *x_ph = new double[n];
  double y_ph=0;
  for (int j=0;j<n;j++) x_ph[j]=x[j];
  f(n,x_ph,y);
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) x_ph[j]=x[j];
    double h=(x[i]==0) ? sqrt(DBL_EPSILON) : sqrt(DBL_EPSILON)*abs(x[i]);
    x_ph[i]+=h;
    f(n,x_ph,y_ph);
    g[i]=(y_ph-y)/h;
  }
  delete [] x_ph;
}
```

Challenger: Algorithmic Differentiation [2]

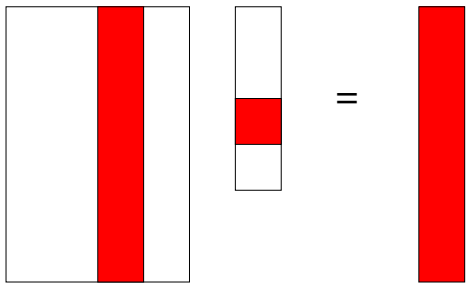
Accuracy and Speed

tangent mode: $\nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} =: \mathbf{y}^{(1)}$



∇F at $O(n) \cdot Cost(F)$

adjoint mode:[3] $\nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} =: \mathbf{x}_{(1)}$



∇F at $O(m) \cdot Cost(F)$

- ... directional derivatives with machine accuracy, thus, avoiding issues due to truncation;
- ... products of the transposed Jacobian with a vector, thus, eliminating the size of the input vector as a parameter of the relative computational cost

$$\mathcal{R} = \frac{Cost(\nabla F^T \cdot \mathbf{v})}{Cost(F)}, \quad v \in \mathbb{R}^m$$

[3] to be refined

The Jacobian is a linear operator $\nabla F : \mathbb{R}^n \to \mathbb{R}^m$.

Its adjoint is defined as $(\nabla F)^* : \mathbb{R}^m \to \mathbb{R}^n$ where

$$< (\nabla F)^* \cdot \mathbf{y}_{(1)}, \mathbf{x}^{(1)} >_{\mathbf{R}^n} = < \mathbf{y}_{(1)}, \nabla F \cdot \mathbf{x}^{(1)} >_{\mathbf{R}^m} \quad ,$$

and where $< ., . >_{\mathbf{R}^n}$ and $< ., . >_{\mathbf{R}^m}$ denote appropriate scalar products in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively.

Theorem

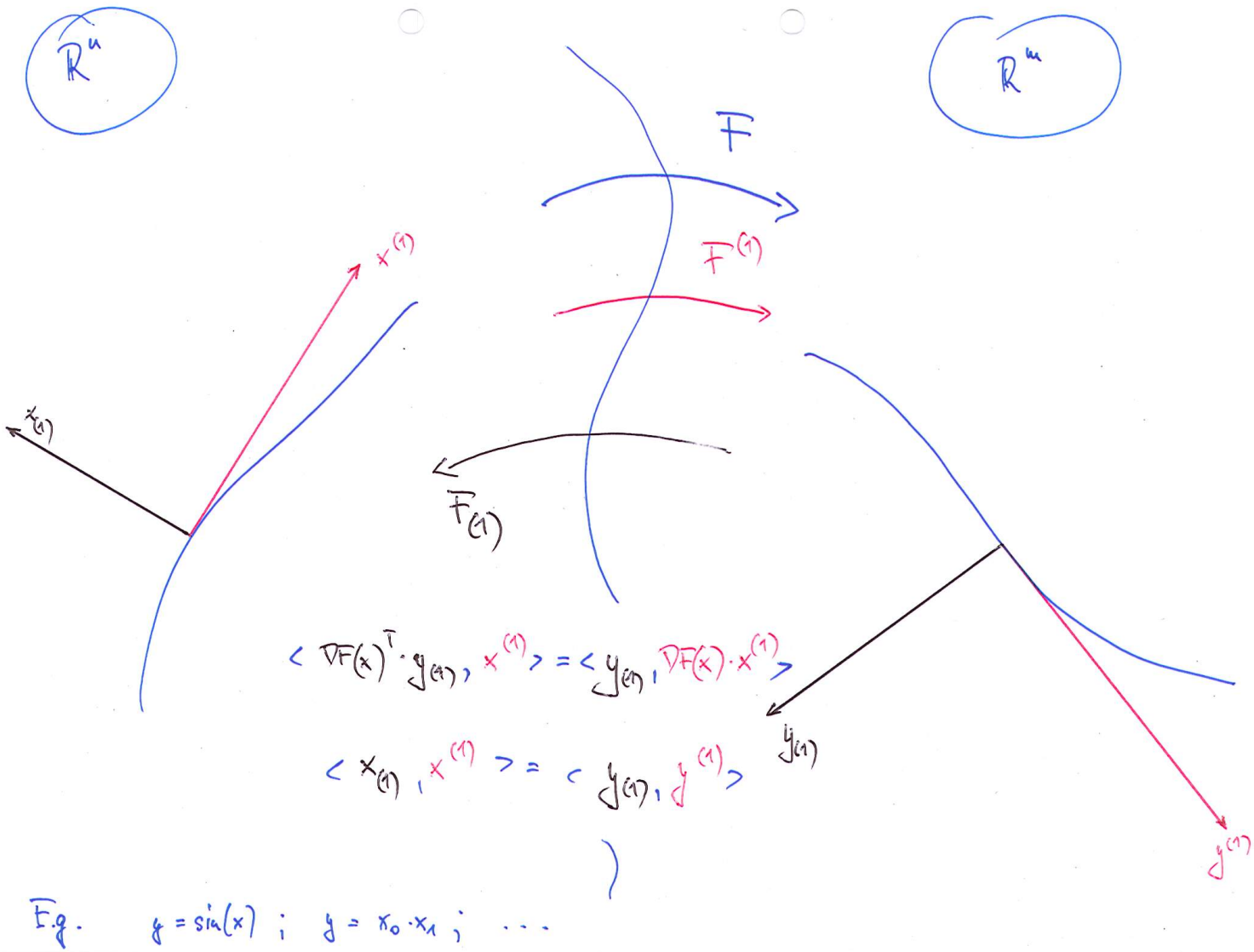
$(\nabla F)^* = (\nabla F)^T$.

$$< \underbrace{(\nabla F)^T}_{[=:\mathbf{x}_{(1)}]} \cdot \mathbf{y}_{(1)}, \mathbf{x}^{(1)} >_{\mathbf{R}^n} = < \mathbf{y}_{(1)}, \underbrace{\nabla F \cdot \mathbf{x}^{(1)}}_{[=:\mathbf{y}^{(1)}]} >_{\mathbf{R}^m}$$

Note invariant at each point in the program execution $\to$ validation of derivatives.

Challenger: Algorithmic Differentiation

Intuition



Comment on Notation

1. $v^{(i)}$ denotes a directional derivative of v resulting from the i-th application of AD (this application in tangent mode) to the code containing v

2. $v_{(i)}$ denotes an adjoint (derivative) of v resulting from the i-th application of AD (this application in adjoint mode) to the code containing v

3. $v^{(1,\ldots,k)}$ denotes a k-th directional derivative of v

4. $v^{(k_1,\ldots,k_i)(p)} \equiv v^{(k_1,\ldots,k_i,p)}$

5. $v^{(k_1,\ldots,k_i)}_{(l_1,\ldots,l_j)}$, $j > 0$, denotes a $(i+j)$-th adjoint derivative of v

6. $v^{(k_1,\ldots,k_i)}_{(l_1,\ldots,l_j)(p)} \equiv v^{(k_1,\ldots,k_i)}_{(l_1,\ldots,l_j,p)}$

Chain Rule of Differential Calculus

... without proof

Let $\mathbf{y} = F(\mathbf{x}) : D_F \subseteq \mathbb{R}^n \to I_F \subseteq \mathbb{R}^m$ be defined over D_F and let

$$\mathbf{y} = F(\mathbf{x}) = G(H(\mathbf{x}), \mathbf{x}) = G(\mathbf{v}, \mathbf{x})$$

be such that both G and H are continuously differentiable over their respective domains $D_G = I_H \times D_F$ and $D_H \subseteq D_F$. Then F is continuously differentiable over D_F and

$$\frac{dF}{d\mathbf{x}}(\mathbf{x}^*) = \frac{\partial G}{\partial \mathbf{v}}(\mathbf{v}^*) \cdot \frac{dH}{d\mathbf{x}}(\mathbf{x}^*) + \frac{\partial G}{\partial \mathbf{x}}(\mathbf{x}^*)$$

for all $\mathbf{x}^* \in D_F$ and $\mathbf{v}^* = H(\mathbf{x}^*)$.

1. The given implementation of $F : \mathbb{R}^n \to \mathbb{R}^m : \mathbf{y} = F(\mathbf{x})$, can be decomposed into a single assignment code (sac)

$$\begin{aligned} v_i &= x_i & i &= 0, \ldots, n-1 \\ v_j &= \varphi_j\left((v_k)_{k \prec j}\right) & j &= n, \ldots, n+q-1 \\ y_k &= v_{n+p+k} & k &= 0, \ldots, m-1 \end{aligned}$$

   where $q = p + m$ and $k \prec j$ denotes a direct dependence of v_j on v_k as an argument of φ_j.

2. All elemental functions φ_j possess continuous partial derivatives

$$\partial_{j,i} \equiv \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j}$$

   with respect to their arguments $(v_k)_{k \prec j}$ at all points of interest.

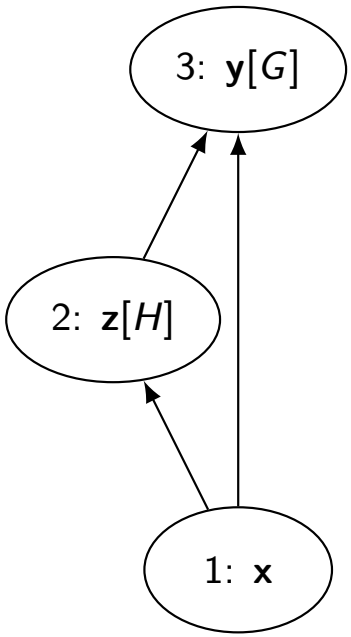
3. The sac induces a directed acyclic graph (DAG) (V, E) with integer vertices $V = \{0, \ldots, n+q\}$ and edges $V \times V \supseteq E = \{(i,j) : i \prec j\}$.

4. A linearized DAG (IDAG) is obtained by attaching the $\partial_{j,i}$ to the corresponding edges (i,j) in the DAG.

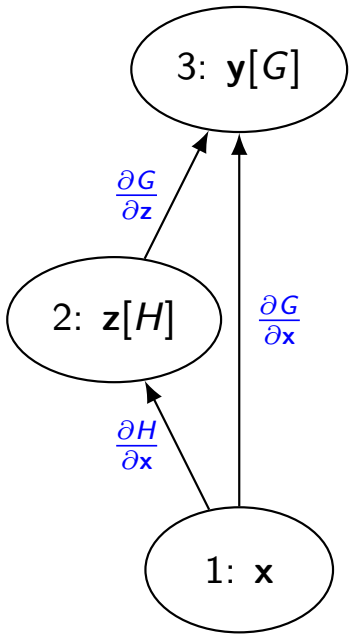
sac:

$$\begin{aligned} \mathbf{z} &:= H(\mathbf{x}) \\ \mathbf{y} &:= G(\mathbf{z}, \mathbf{x}) \end{aligned}$$

DAG:



IDAG:



$$\nabla F(\mathbf{x}) \equiv \frac{d\mathbf{y}}{d\mathbf{x}} = \sum_{\text{path} \in \text{IDAG}} \prod_{(i,j) \in \text{path}} \partial_{j,i}$$

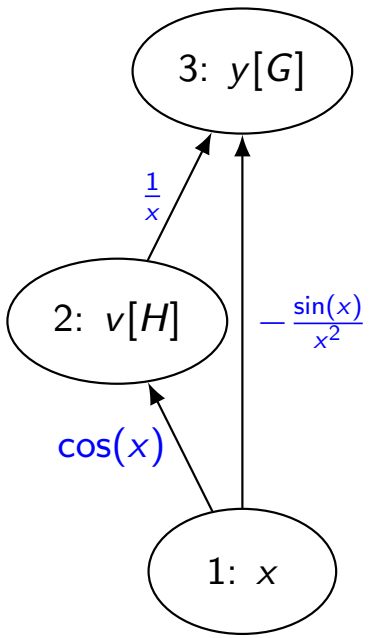
Let $y = F(x)$ be defined over $D_F = \mathbb{R} \setminus \{0\}$ as

$$y = \frac{\sin(x)}{x}.$$

Both $G \equiv /$ and $H \equiv \sin$ are continuously differentiable over their respective domains $D_G = \mathbb{R} \times (\mathbb{R} \setminus \{0\})$ and $D_H = \mathbb{R}$. Hence F is continuously differentiable over its domain D_F and

$$\frac{dF}{dx}(x^*) = \frac{\partial G}{\partial v}(v^*) \cdot \frac{dH}{dx}(x^*) + \frac{\partial G}{\partial x}(x^*) = \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2}$$

for all $x^* \in D_F$ and $v^* = \sin(x^*)$.



Chain Rule $\Rightarrow$ Chained Jacobian Matrix Products

Consider

$$A_3 \cdot A_2 \cdot A_1 \cdot A_0 \in \mathbb{R}^{k_1 \times k_0}$$

for $A_0 \equiv \nabla F_0 \in \mathbb{R}^{l \times k_0}$, $A_1 \equiv \nabla F_1 \in \mathbb{R}^{l \times l}$, $A_2 \equiv \nabla F_2 \in \mathbb{R}^{l \times l}$, and $A_3 \equiv \nabla F_3 \in \mathbb{R}^{k_1 \times l}$, and where $l < k_1 < k_0$.

Compare

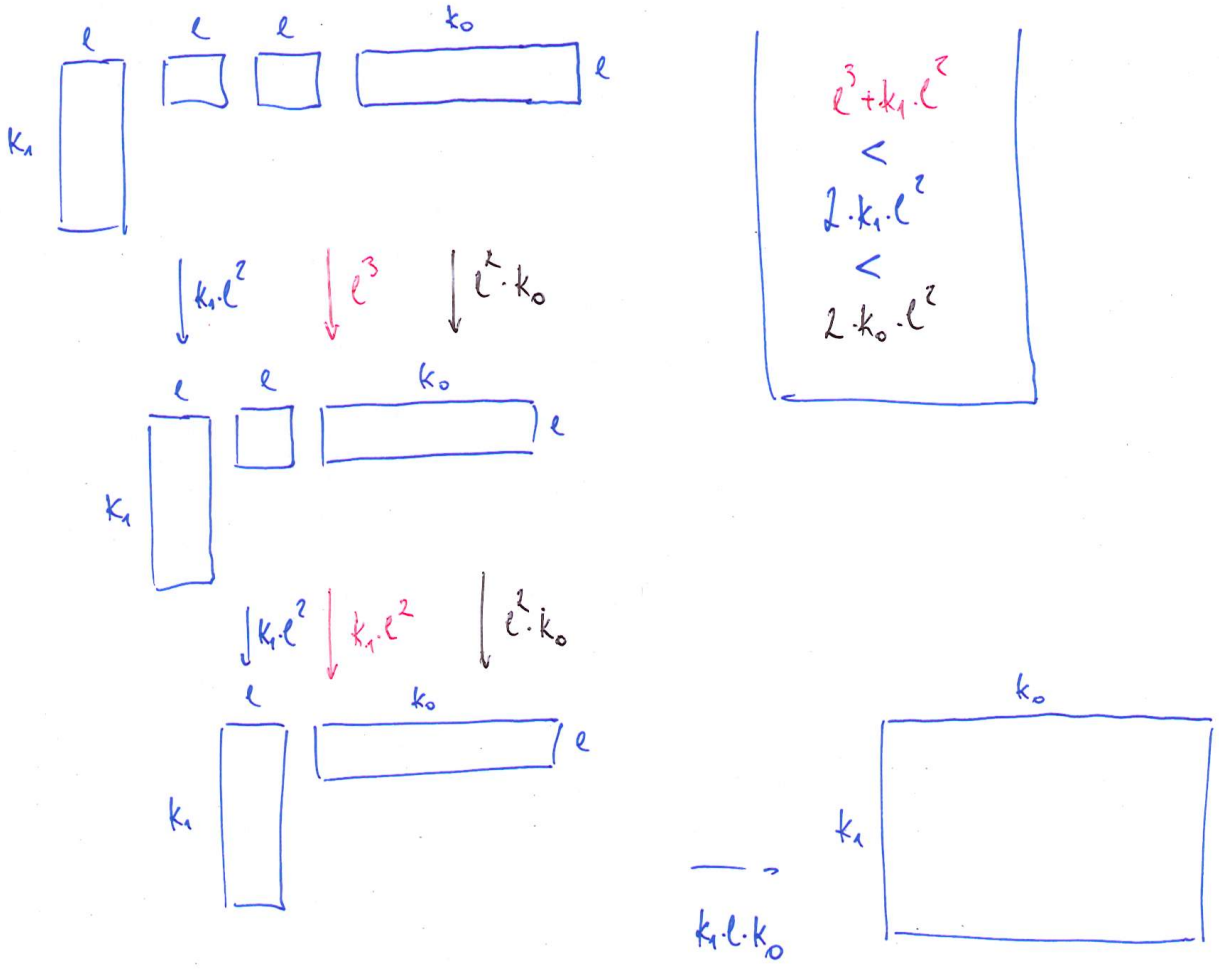
$$OPS(A_3 \cdot (A_2 \cdot (A_1 \cdot A_0))) = k_0 \cdot l \cdot l + k_0 \cdot l \cdot l + k_0 \cdot l \cdot k_1 \quad \text{(forward)}$$

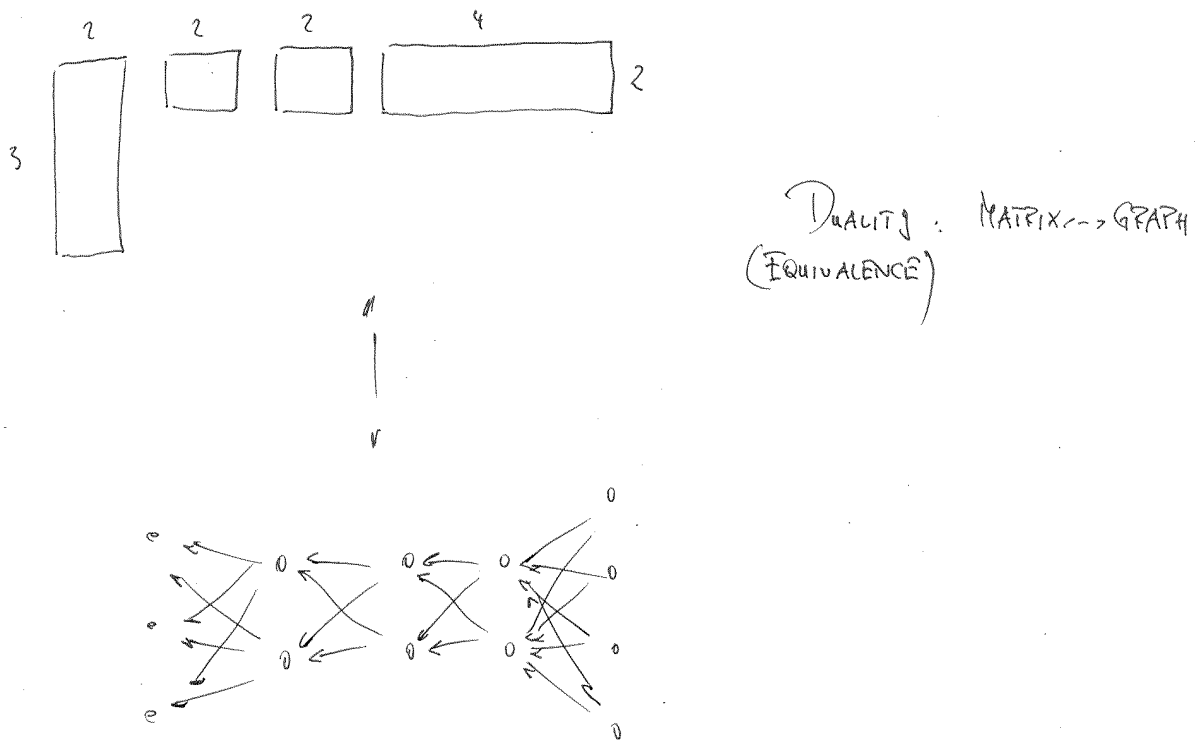
vs.

$$OPS(((A_3 \cdot A_2) \cdot A_1) \cdot A_0) = k_1 \cdot l \cdot l + k_1 \cdot l \cdot l + k_0 \cdot l \cdot k_1 \quad \text{(reverse)}$$

vs.

$$OPS((A_3 \cdot (A_2 \cdot A_1)) \cdot A_0) = l \cdot l \cdot l + k_0 \cdot l \cdot l + k_0 \cdot l \cdot k_1 \quad \text{(mix)}$$


l
l
l
k0
l
k1
k1·l²
l³
l²·k0
l
l
k0
l
k1
k1·l²
k1·l²
l²·k0
l
k0
l
k1
k1·l·k0
k0
k1
l³+k1·l²
<
2·k1·l²
<
2·k0·l²

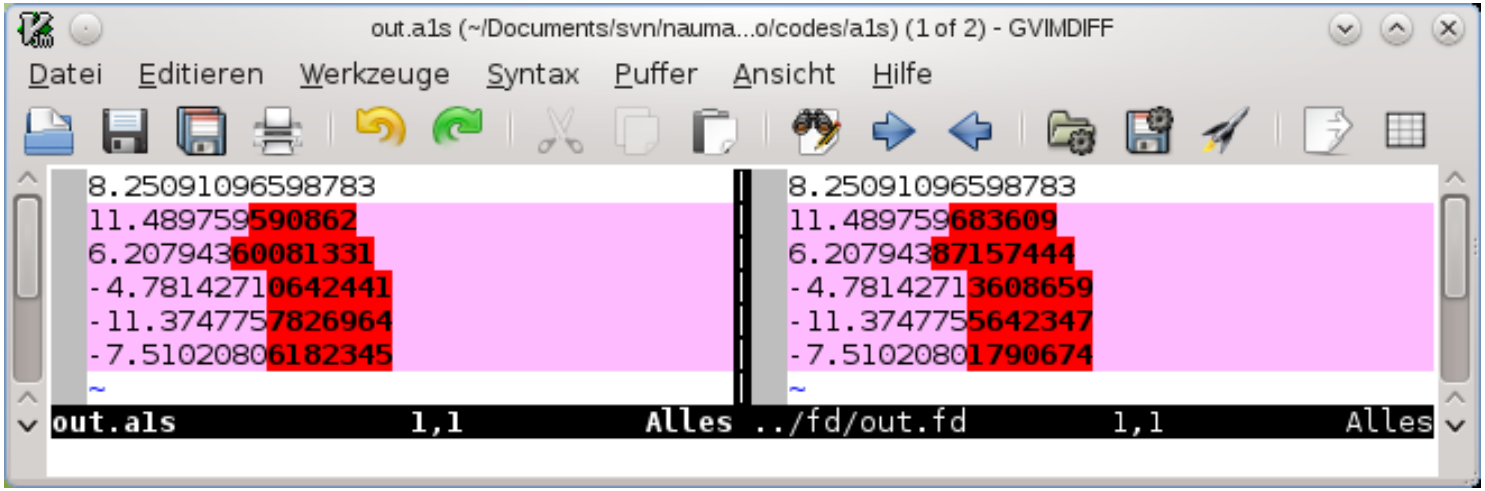


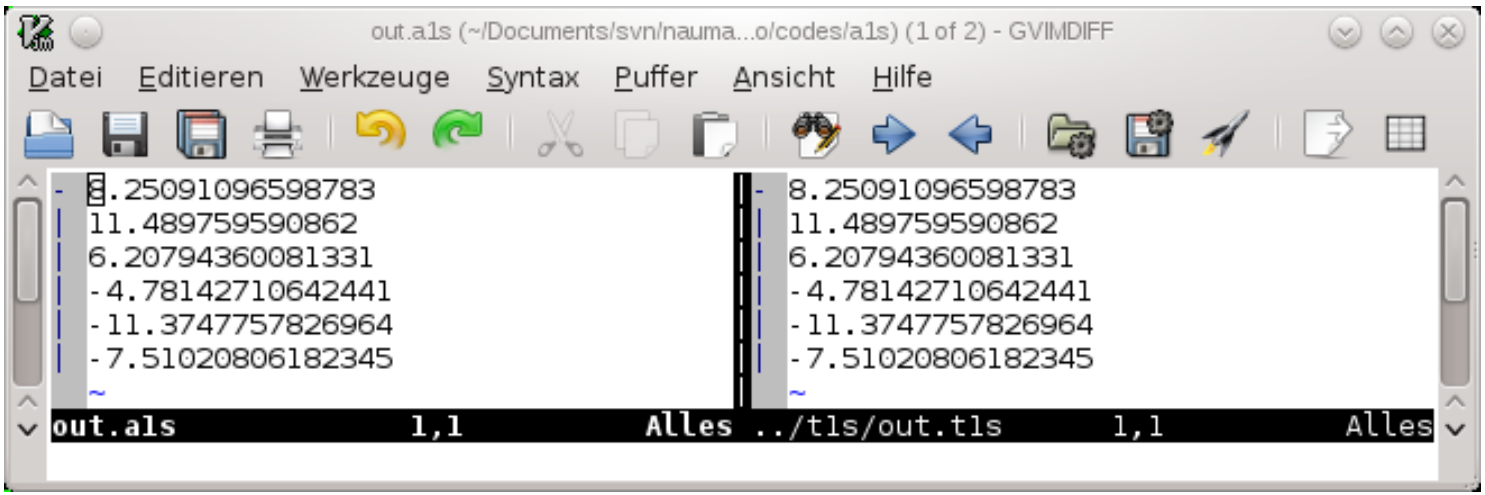
Note: Matrix product by vertex elimination on DAG.

Live for given implementation of

$$y = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

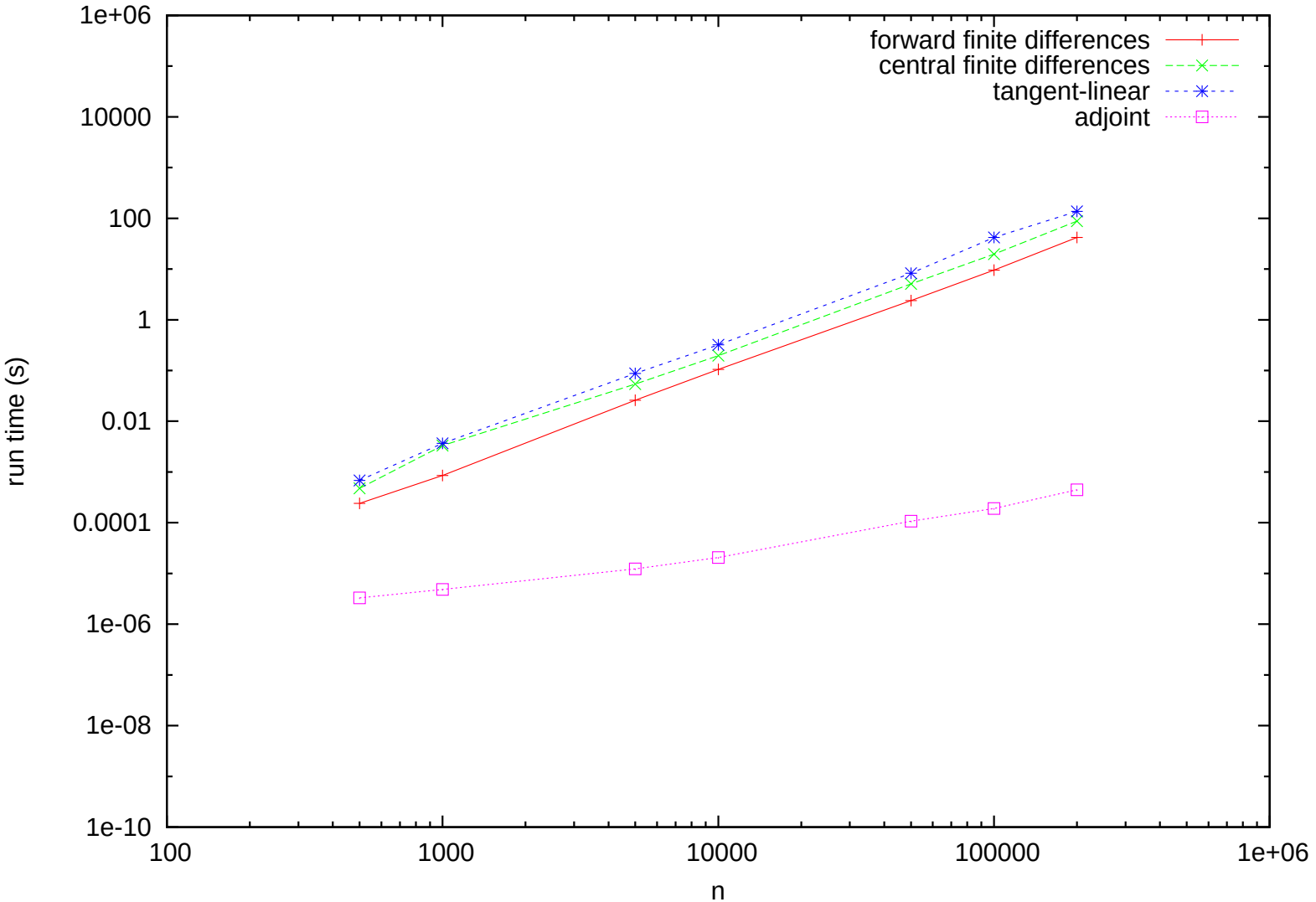
- run and time tangent code for increasing n
- compare output with finite differences
- run and time adjoint code for increasing n
- compare output with tangent


out.a1s (~/Documents/svn/nauma...o/codes/a1s) (1 of 2) - GVIMDIFF
Datei Editieren Werkzeuge Syntax Puffer Ansicht Hilfe
8.25091096598783
11.489759590862
6.2079436008133 1
-4.7814271 0642441
-11.374775 7826964
-7.5102080 6182345
8.25091096598783
11.489759683609
6.2079438 7157444
-4.7814271 3608659
-11.374775 5642347
-7.5102080 1790674
out.a1s 1,1 Alles
../fd/out.fd 1,1 Alles


out.a1s (~/Documents/svn/nauma...o/codes/a1s) (1 of 2) - GVIMDIFF
Datei Editieren Werkzeuge Syntax Puffer Ansicht Hilfe
8.25091096598783
11.489759590862
6.20794360081331
-4.78142710642441
-11.3747757826964
-7.51020806182345
8.25091096598783
11.489759590862
6.20794360081331
-4.78142710642441
-11.3747757826964
-7.51020806182345
out.a1s 1,1 Alles
../tls/out.tls 1,1 Alles

Challenger: Algorithmic Differentiation

Speed



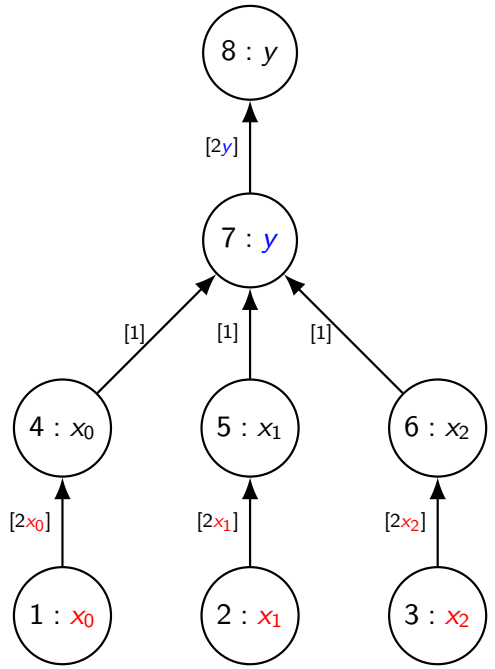
Consider

$$y = f(x) = f_3(f_2(f_1(x))) = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$$

where

$$\mathbb{R}^n \ni x = f_1(x) : x_i = x_i^2$$
$$\mathbb{R}^1 \ni y = f_2(x) = \sum_{i=0}^{n-1} x_i$$
$$\mathbb{R}^1 \ni y = f_3(y) = y^2$$

yielding the IDAG on the right.



The f_i are continuously differentiable wrt. their arguments. Hence, by the chain rule

$$\underset{\in \mathbf{R}^n}{\nabla f(x)} = \underset{\in \mathbf{R}}{\nabla f_3(y)} \underset{\in \mathbf{R}^{1\times n}}{\nabla f_2(x)} \underset{\in \mathbf{R}^{n\times n}}{\nabla f_1(x)} = 2y \cdot (1 \ldots 1) \cdot \begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} = \begin{pmatrix} 4yx_0 \\ \vdots \\ 4yx_{n-1} \end{pmatrix}$$

Note overwriting! ($\rightarrow$ "Maths $\overset{!}{\neq}$ CS")

A first-order central finite difference quotient

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

yields a second-order accurate[4] approximation of the first directional derivative

$$y^{(1)} = \nabla f(x) \cdot x^{(1)} = \nabla f_3(f_2(f_1(x))) \cdot \nabla f_2(f_1(x)) \cdot \nabla f_1(x) \cdot x^{(1)}$$

$$= 2 \sum_{i=0}^{n-1} x_i^2 \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \cdot \begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0^{(1)} \\ \vdots \\ x_{n-1}^{(1)} \end{pmatrix}.$$

The whole gradient $\nabla f(x)$ can be approximated by letting $x^{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ at a computational cost of $O(n) \cdot \text{Cost}(f)$.

[4] cubic remainder term in Taylor expansion of nominator

Can we avoid truncation?

YES, WE CAN!

$\rightarrow$ For $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, implement the tangent model $y^{(1)} = \nabla F \cdot x^{(1)}$. At most n runs of the tangent code are required to obtain the whole Jacobian with machine accuracy.

A first-order tangent code computes the directional derivative

$$
\begin{aligned}
y^{(1)} &= \nabla f(x) \cdot x^{(1)} \\
&= \nabla f_3(f_2(f_1(x))) \cdot \nabla f_2(f_1(x)) \cdot \nabla f_1(x) \cdot x^{(1)} \\
&= 2\sum_{i=0}^{n-1} x_i^2 \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \cdot \begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0^{(1)} \\ \vdots \\ x_{n-1}^{(1)} \end{pmatrix} \\
&= 2\sum_{i=0}^{n-1} x_i^2 \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \cdot \begin{pmatrix} 2x_0 x_0^{(1)} \\ \vdots \\ 2x_{n-1} x_{n-1}^{(1)} \end{pmatrix} \\
&= 2\sum_{i=0}^{n-1} x_i^2 \cdot 2\sum_{i=0}^{n-1} x_i x_i^{(1)} = 4\sum_{i=0}^{n-1} x_i^2 \cdot \sum_{i=0}^{n-1} x_i x_i^{(1)}
\end{aligned}
$$

with machine accuracy.

To compute the whole gradient with machine accuracy an implementation of

$$y^{(1)} = 4 \sum_{i=0}^{n-1} x_i^2 \cdot \sum_{i=0}^{n-1} x_i x_i^{(1)}$$

such as

```
void f_t1s(const vector<double>& x, const vector<double>& x_t1s,
           double& y_t1s) {
  double t=0;
  for (int i=0;i<x.size();i++) t=t+x[i]*x[i];
  y_t1s=0;
  for (int i=0;i<x.size();i++) y_t1s=y_t1s+x[i]*x_t1s[i];
  y_t1s=4*t*y_t1s;
}
```

needs to be called n times with `x_t1s` ranging over the Cartesian basis vectors in $\mathbb{R}^n$.

Note: Solution tailored toward the given problem.

Wanted: Generic approach to tangent code generation!

Can we reduce the computational cost?

YES, WE CAN!

$\rightarrow$ For $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, implement the adjoint model $x_{(1)} = \nabla F^T \cdot y_{(1)}$. At most m runs of the adjoint code are required to obtain the whole Jacobian with machine accuracy. Hence, less (matrix-free) matrix vector products are required if $n > \mathcal{R} \cdot m$.[5]

[5] $\mathcal{R}$ denotes the constant computational overhead induced by the *reversal of the data flow* in adjoint mode.

A first-order adjoint code computes the transposed Jacobian vector product

$$
\begin{aligned}
x_{(1)} &= \nabla f(x)^T \cdot y_{(1)} \\
&= \nabla f_1(x)^T \cdot \nabla f_2(f_1(x))^T \cdot \nabla f_3(f_2(f_1(x)))^T \cdot y_{(1)} \\
&= \begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \cdot 2 \sum_{i=0}^{n-1} x_i^2 \cdot y_{(1)} \\
&= \begin{pmatrix} 4x_0 y_{(1)} \sum_{i=0}^{n-1} x_i^2 \\ \vdots \\ 4x_{n-1} y_{(1)} \sum_{i=0}^{n-1} x_i^2 \end{pmatrix}
\end{aligned}
$$

with machine accuracy.

To compute the whole gradient with machine accuracy an implementation of

$$x_{(1)} = \begin{pmatrix} 4x_0 y_{(1)} \sum_{i=0}^{n-1} x_i^2 \\ \vdots \\ 4x_{n-1} y_{(1)} \sum_{i=0}^{n-1} x_i^2 \end{pmatrix}$$

such as

```
void f_a1s(const vector<double>& x, vector<double>& x_a1s,
           double y_a1s) {
  double t=0;
  for (int i=0;i<x.size();i++) t=t+x[i]*x[i];
  for (int i=0;i<x.size();i++) x_a1s[i]=4*x[i]*y_a1s*t;
}
```

needs to called once with y_a1s=1.

Again: Solution tailored toward the given problem.

Wanted: Generic approach to adjoint code generation!

Motivational Example

... as Chained (Jacobian) Matrix Product

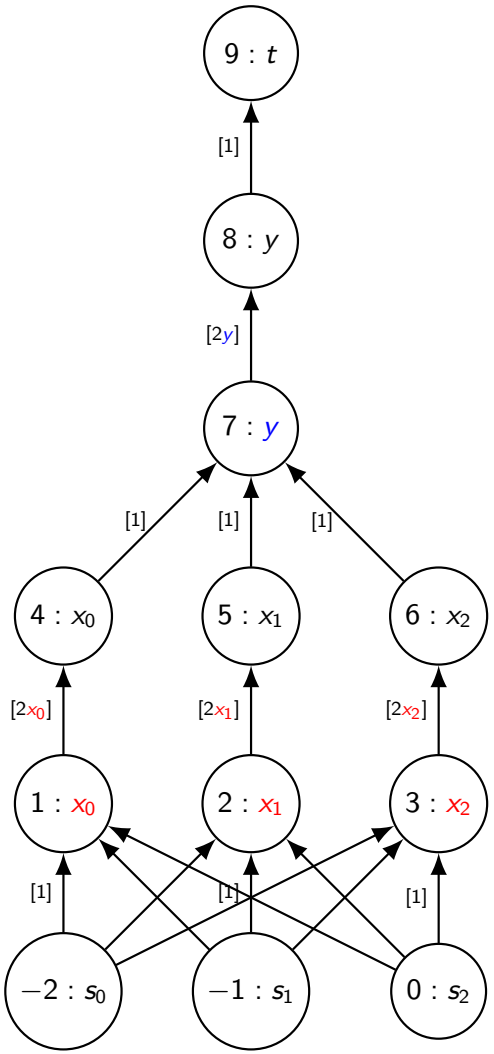
$$
\begin{aligned}
\nabla f(x) &= I_m \cdot \nabla f(x) = \nabla f(x) \cdot I_n = I_m \cdot \nabla f(x) \cdot I_n \\
&= I_m \cdot \nabla f_3(f_2(f_1(x))) \cdot \nabla f_2(f_1(x)) \cdot \nabla f_1(x) \cdot I_n \\
&= (1) \cdot (2 \sum_{i=0}^{n-1} x_i^2) \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \cdot \begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & \vdots & \\ & & 1 \end{pmatrix} \\
&= \left[\left[(1) \cdot (2 \sum_{i=0}^{n-1} x_i^2) \right] \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \right] \cdot \begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} \\
&= (2 \sum_{i=0}^{n-1} x_i^2) \cdot \left[\begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \cdot \left[\begin{pmatrix} 2x_0 & & \\ & \vdots & \\ & & 2x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & \vdots & \\ & & 1 \end{pmatrix} \right] \right]
\end{aligned}
$$

Note:

- matrix-free local Jacobian chain
- exploitation of sparsity comes at a cost

Motivational Example

... as Chained (Jacobian) Matrix Product (Graphically)



$$\frac{\partial t}{\partial \mathbf{s}} = \frac{\partial t}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{s}} = \frac{\partial y}{\partial \mathbf{x}}$$

Note:

- wanted: $\frac{\partial y}{\partial \mathbf{x}}$ (matrix-free Jacobian chain)
- $\frac{\partial y}{\partial \mathbf{s}}$ by tangent code
- $\frac{\partial t}{\partial \mathbf{x}}$ by adjoint code

Example: Product Reduction

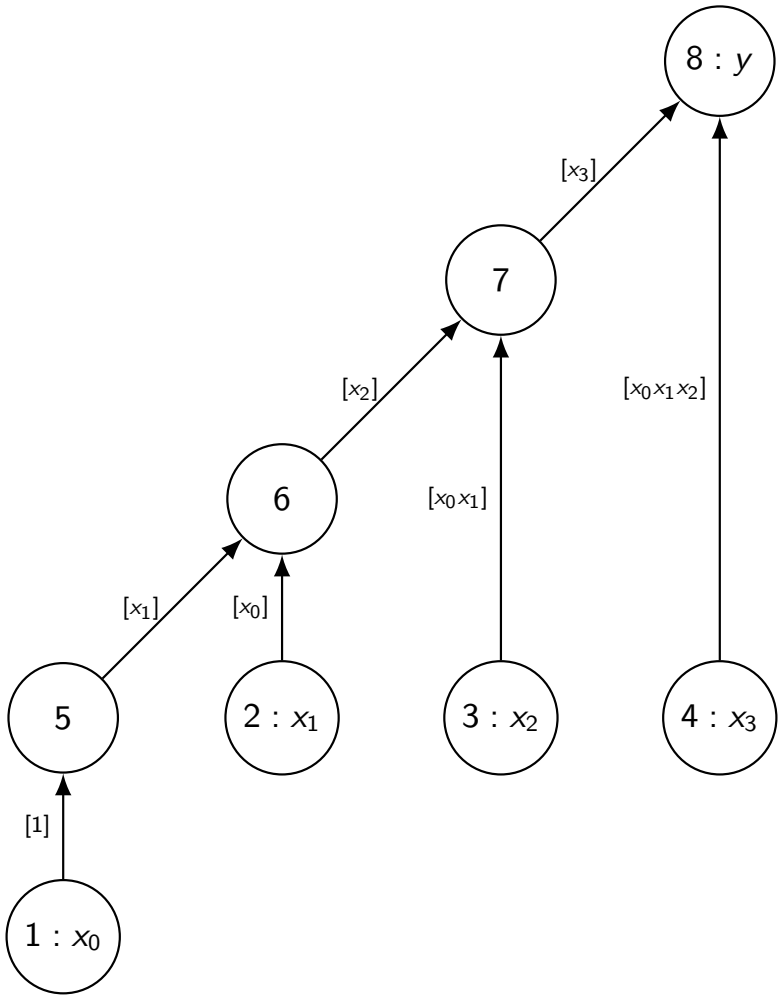
For the previously considered implementation of the product reduction

$$\mathbb{R} \ni y = \prod_{i=0}^{n-1} x_i$$

we obtain for $n = 4$

$$y = f(x) = f_4(f_3(f_2(f_1(x_0), x_1), x_2), x_3),$$

where $f_1 = I$ and $f_i = f_{i-1} \cdot x_i$ for $i = 2, 3, 4$.



By the chain rule,

$$
\begin{aligned}
y^{(1)} = \nabla f \cdot x^{(1)} &= \begin{pmatrix} x_3 & x_0 \cdot x_1 \cdot x_2 \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} x_2 & x_0 \cdot x_1 \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} x_1 & x_0 \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} 1 \cdot x_0^{(1)} \end{pmatrix} \\ x_1^{(1)} \end{pmatrix} \\ x_2^{(1)} \end{pmatrix} \\ x_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} x_3 & x_0 \cdot x_1 \cdot x_2 \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} x_2 & x_0 \cdot x_1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \cdot x_0^{(1)} + x_0 \cdot x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} \\ x_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} x_3 & x_0 \cdot x_1 \cdot x_2 \end{pmatrix} \cdot \begin{pmatrix} x_2 \cdot (x_1 \cdot x_0^{(1)} + x_0 \cdot x_1^{(1)}) + x_0 \cdot x_1 \cdot x_2^{(1)} \\ x_3^{(1)} \end{pmatrix} \\
&= x_3 \cdot (x_2 \cdot (x_1 \cdot x_0^{(1)} + x_0 \cdot x_1^{(1)}) + x_0 \cdot x_1 \cdot x_2^{(1)}) + x_0 \cdot x_1 \cdot x_2 \cdot x_3^{(1)}
\end{aligned}
$$

By the chain rule,

$$x_{(1)} = \nabla f^T \cdot y_{(1)} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} (1) \cdot x_1 \\ x_0 \end{pmatrix} \cdot x_2 \\ x_0 \cdot x_1 \end{pmatrix} \cdot x_3 \\ x_0 \cdot x_1 \cdot x_2 \end{pmatrix} \cdot y_{(1)} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} x_1 \\ x_0 \end{pmatrix} \cdot x_2 \\ x_0 \cdot x_1 \end{pmatrix} \cdot x_3 \cdot y_{(1)} \\ x_0 \cdot x_1 \cdot x_2 \cdot y_{(1)} \end{pmatrix}$$

$$= \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} x_1 \\ x_0 \end{pmatrix} \cdot x_2 \cdot x_3 \cdot y_{(1)} \\ x_0 \cdot x_1 \cdot x_3 \cdot y_{(1)} \end{pmatrix} \\ x_0 \cdot x_1 \cdot x_2 \cdot y_{(1)} \end{pmatrix} = \begin{pmatrix} x_1 \cdot x_2 \cdot x_3 \cdot y_{(1)} \\ x_0 \cdot x_2 \cdot x_3 \cdot y_{(1)} \\ x_0 \cdot x_1 \cdot x_3 \cdot y_{(1)} \\ x_0 \cdot x_1 \cdot x_2 \cdot y_{(1)} \end{pmatrix}$$

Example: Product Reduction

Race

... built under Linux with `g++ -O3` ... and run on an Intel CORE i7 with 4gb RAM; run time in seconds ...

n	cfd	tangent	adjoint (recompute)	adjoint (store)	primal
10^4	7	6	< 1	< 1	< 1
$2 \cdot 10^4$	27	26	1.5	< 1	< 1
$3 \cdot 10^4$	60	58	3.5	< 1	< 1
$4 \cdot 10^4$	106	103	6	< 1	< 1
$\vdots$					
10^7	$\to \infty$	$\to \infty$	:-(	1	1
10^8	$\to \infty$	$\to \infty$	:-(	10	9
$2 \cdot 10^8$	$\to \infty$	$\to \infty$	:-(	∞	18

$\mathcal{R} \in (1, \infty] \ldots$

Exercise

Use central finite differences for the approximation of the gradient of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

Objective

You will learn about

1. the principles of generating
   - tangent
   - adjoint
   - second-(and higher-)order tangent
   - second-(and higher-)order adjoint

   versions of numerical simulation code by overloading using the dco/c++ (derivative code by overloading in C++) library
2. the use of such derivative code, for example, in numerical methods.

You will understand that

1. further (course) work is necessary in order ot obtain robust, efficient, and scalable derivative code.

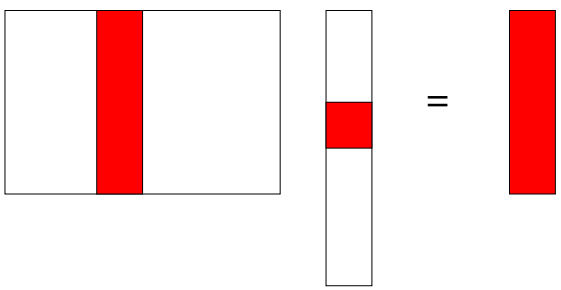
First-Order Tangent and Adjoint Code by Overloading

A first-order tangent code $F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m$,

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \end{pmatrix} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}),$$

computes a Jacobian vector product (a directional derivative) alongside with the function value as follows:

$$\mathbf{y} := F(\mathbf{x})$$
$$\mathbf{y}^{(1)} := \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$$



Define

$$\mathbf{v}^{(1)} \equiv \frac{d\mathbf{v}}{ds}$$

for $\mathbf{v} \in \{\mathbf{x}, \mathbf{y}\}$ and some auxiliary $s \in \mathbb{R}$ assuming that $F(\mathbf{x}(s))$ is continuously differentiable over $\mathbb{R}$.

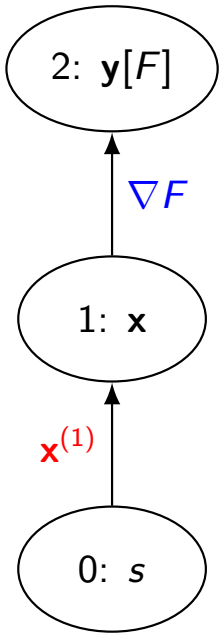
By the chain rule of differential calculus

$$\frac{d\mathbf{y}}{ds} = \frac{d\mathbf{y}}{d\mathbf{x}} \cdot \frac{d\mathbf{x}}{ds} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$$

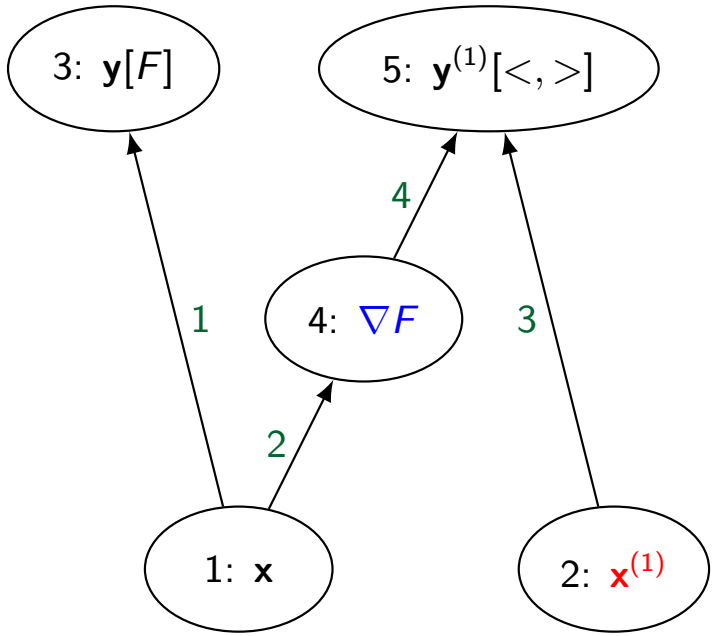
and hence

$$\mathbf{y}^{(1)} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}.$$

tangent-augmented lDAG



tangent DAG



- Vertices sorted (indexed) topologically wrt. data dependence.
- Edges sorted topologically wrt. index of target with index of source as tie-breaker.
- Note inner product notation $< \nabla F(\mathbf{x}), \mathbf{x}^{(1)} > \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$; see also [2].

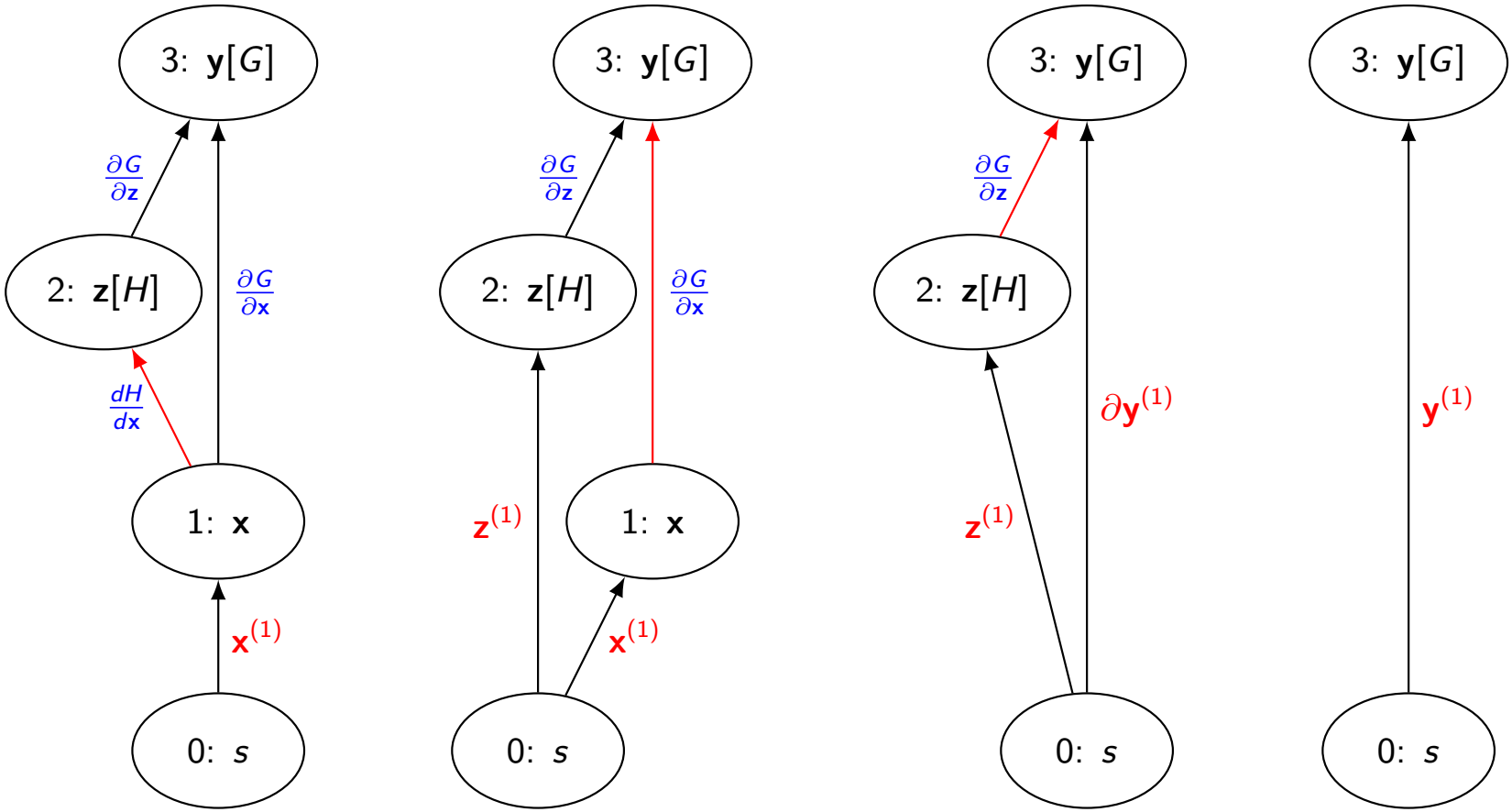
Let

$$\mathbf{y} = F(\mathbf{x}) = G(H(\mathbf{x}), \mathbf{x})$$

with $F : \mathbb{R}^n \to \mathbb{R}^m$, $H : \mathbb{R}^n \to \mathbb{R}^k$, $\mathbf{z} = H(\mathbf{x})$, and $G : \mathbb{R}^{n+k} \to \mathbb{R}^m$ continuously differentiable over their respective domains. By the chain rule

$$\begin{aligned}
\mathbf{y}^{(1)} = \frac{dF}{ds} = \frac{dF}{d\mathbf{x}} \cdot \mathbf{x}^{(1)} &= \left(\frac{\partial G}{\partial \mathbf{z}} \cdot \frac{dH}{d\mathbf{x}} + \frac{\partial G}{\partial \mathbf{x}} \right) \cdot \mathbf{x}^{(1)} \\
&= \frac{\partial G}{\partial \mathbf{z}} \cdot \left(\frac{dH}{d\mathbf{x}} \cdot \mathbf{x}^{(1)} \right) + \frac{\partial G}{\partial \mathbf{x}} \cdot \mathbf{x}^{(1)} \\
&= \frac{\partial G}{\partial \mathbf{z}} \cdot \mathbf{z}^{(1)} + \frac{\partial G}{\partial \mathbf{x}} \cdot \mathbf{x}^{(1)} \\
&= \frac{\partial G}{\partial \mathbf{z}} \cdot \mathbf{z}^{(1)} + \partial \mathbf{y}^{(1)}
\end{aligned}$$

Graphically, this sequence of operations can be represented by a sequence of *edge back eliminations* on the corresponding tangent-augmented IDAG as follows:



An edge is back eliminated by multiplying its label with the label(s) of the incoming edge(s) of its source followed by its removal. If the source has no further emanating edges, then it is also removed.

First-order tangent code back eliminates all back eliminatable edges in topological order (no storage of IDAG).

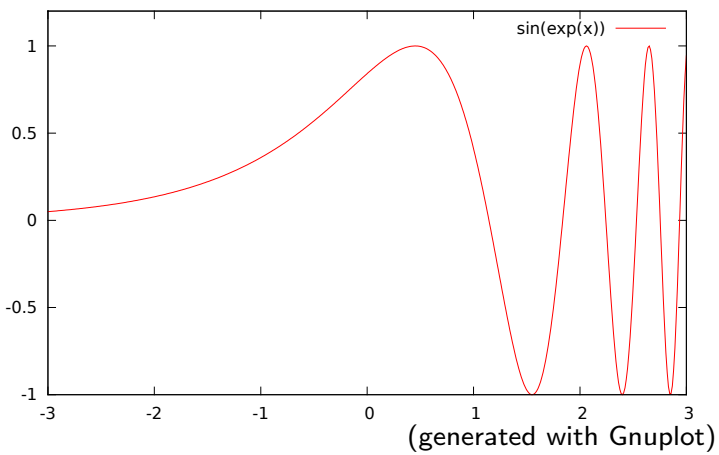
Bottom-up: Tangents of elemental functions + chain rule, e.g.

1. $y = \sin(x)$;
$$\frac{\partial y}{\partial s} = \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial s} = \cos(x) \cdot x^{(1)}$$
2. $y = x_0 \cdot x_1$;
$$y^{(1)} \equiv \frac{\partial y}{\partial s} = \frac{\partial y}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s} = \begin{pmatrix} x_1 & x_0 \end{pmatrix} \cdot \begin{pmatrix} x_0^{(1)} \\ x_1^{(1)} \end{pmatrix} = x_0^{(1)} \cdot x_1 + x_0 \cdot x_1^{(1)}$$
3. $y = \sin(x_0 \cdot x_1)$; Let $v = x_0 \cdot x_1$. Then
$$y^{(1)} \equiv \frac{\partial y}{\partial s} = \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s}$$
$$= \cos(v) \cdot \begin{pmatrix} x_1 & x_0 \end{pmatrix} \cdot \begin{pmatrix} x_0^{(1)} \\ x_1^{(1)} \end{pmatrix} = \cos(v) \cdot \left(x_0^{(1)} \cdot x_1 + x_0 \cdot x_1^{(1)} \right)$$

First-Order Tangent Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$


```
void f(double &z) {
  z=exp(z);
  z=sin(z);
}
```

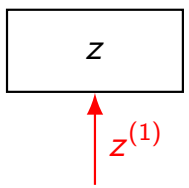


(generated with Gnuplot)

First-Order Tangent Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

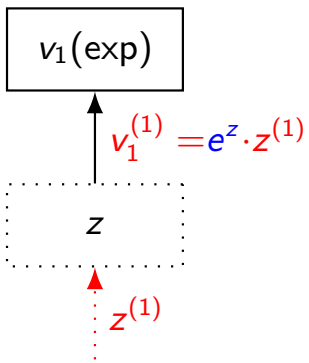

```
void t1_f(t1s::type &[z,t1_z]) {
  ...
}
```



First-Order Tangent Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

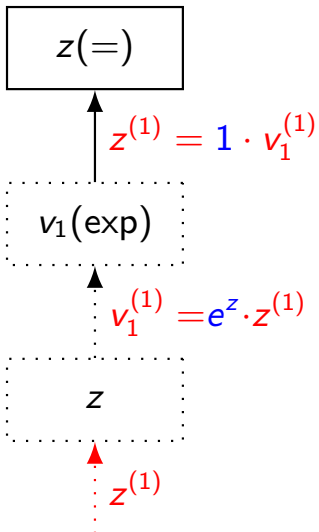
```
void t1_f(t1s::type &[z,t1_z]) {
  ... [exp(z),exp(z)*t1_z];
}
```



First-Order Tangent Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

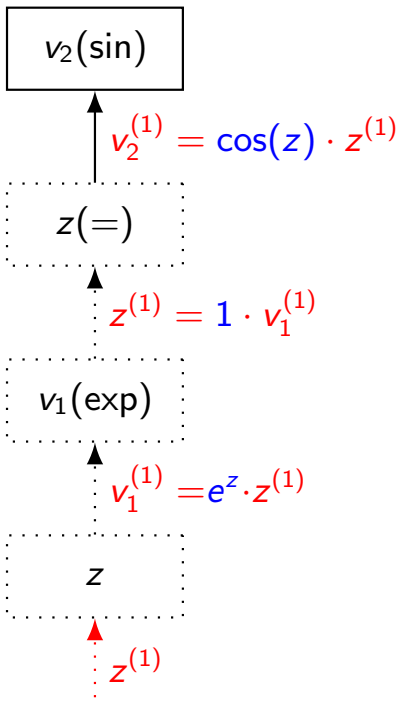
```
void t1_f(t1s::type &[z,t1_z]) {
  [z,t1_z]=[exp(z),exp(z)*t1_z];
}
```



First-Order Tangent Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$


```
void t1_f(t1s::type &[z,t1_z]) {
  [z,t1_z]=[exp(z),exp(z)*t1_z];
  ...  [sin(z),cos(z)*t1_z];
}
```



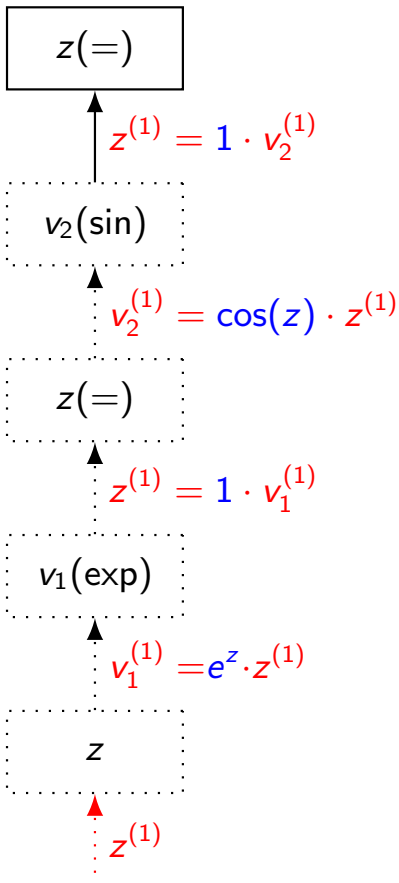
First-Order Tangent Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

```
void t1_f(t1s::type &[z,t1_z]) {
  [z,t1_z]=[exp(z),exp(z)*t1_z];
  [z,t1_z]=[sin(z),cos(z)*t1_z];
}
```

Driver:

```
...
t1s::type [z,t1_z]=[...,1];
t1_f([z,t1_z]);
cout << t1_z << endl;
...
```



For scalar tangent mode AD, a class dco_t1s_type (dco's tangent 1st-order scalar type) is defined with **double** precision members v (value) and t (tangent).

```
class dco_t1s_type {
  public :
    double v;
    double t;
    dco_t1s_type(const double&);
    dco_t1s_type();
    dco_t1s_type& operator=(const dco_t1s_type&);
};
```

```
dco_t1s_type&
dco_t1s_type::operator=(const dco_t1s_type& x) {
  if (this==&x) return *this;
  v=x.v; t=x.t;
  return *this;
}
```

```
dco_t1s_type operator*(const dco_t1s_type& x1,
                       const dco_t1s_type& x2) {
  dco_t1s_type tmp;
  tmp.v=x1.v*x2.v;
  tmp.t=x1.t*x2.v+x1.v*x2.t;
  return tmp;
}
```

```
dco_t1s_type sin(const dco_t1s_type& x) {
  dco_t1s_type tmp;
  tmp.v=sin(x.v);
  tmp.t=cos(x.v)*x.t;
  return tmp;
}
```

```
...
#include "dco_t1s_type.hpp"
const int n=4;
void f(dco_t1s_type *x, dco_t1s_type &y) { ... }

int main() {
  dco_t1s_type x[n], y;
  for (int i=0;i<n;i++) x[i]=1
  for (int i=0;i<n;i++) {
    x[i].t=1; f(x,y); x[i].t=0;
    cout << y.t << endl;
  }
  return 0;
}
```

First-Order Tangent Code

Toy Problem with dco/c++

```
1  #include<iostream>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef gt1s<double> AD_MODE;
8  typedef AD_MODE::type AD_TYPE;
9
10 template<typename T>
11 void f(int n, const T *x, T& y ) { y=x[0]/x[1]; }
12
13 void driver(int n, const double* xv, double& yv, double* g) {
14   AD_TYPE *x=new AD_TYPE[n], y;
15   for (int i=0;i<n;i++) {
16     for (int j=0;j<n;j++) x[j]=xv[j];
17     derivative(x[i])=1;
18     f(n,x,y);
19     g[i]=derivative(y);
20   }
21   yv=value(y);
22   delete [] x;
23 }
24
25 int main() {
26   const int n=2;
```

First-Order Tangent Code

Toy Problem with dco/c++ (cont.)


```
27    double x[n]={2.71,3.14}, y, g[n];
28    driver(n,x,y,g);
29    cout << "y=" << y << endl;
30    cout << "g=[" << endl;
31    for (int i=0;i<n;i++) cout << g[i] << endl;
32    cout << "]" << endl;
33    return 0;
34 }
```

- ~/gt1s/gt1s_simple.cpp

Live:

- implementation of driver `fg` for given instantiation of

```
template<typename T>
void f(const int n, T *x, T& y) {
...
}
```

  with `dco::t1s<double>::type`

- build and run

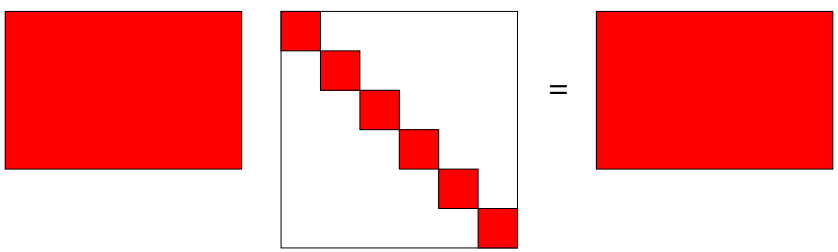
```
void fg(const double x[n], double& y, double g[n]) {
  T1S_TYPE t1s_x[n], t1s_y;
  for (int i=0;i<n;i++) t1s_x[i]=x[i];
  for (int i=0;i<n;i++) {
    dco::set_derivative(t1s_x[i],1);
    f(t1s_x,t1s_y);
    g[i]=dco::get_derivative(t1s_y);
    dco::set_derivative(t1s_x[i],0);
  }
  y=dco::get_value(t1s_y);
}
```

A first-order vector tangent code $F^{(1)} : \mathbb{R}^n \times \mathbb{R}^{n \times l} \to \mathbb{R}^m \times \mathbb{R}^{m \times l}$,

$$\begin{pmatrix} \mathbf{y} \\ Y^{(1)} \end{pmatrix} = F^{(1)}(\mathbf{x}, X^{(1)}),$$

computes a Jacobian matrix product (a collection of directional derivative) alongside with the function value as follows:

$$\mathbf{y} := F(\mathbf{x})$$
$$Y^{(1)} := \nabla F(\mathbf{x}) \cdot X^{(1)}$$



... accumulation of the whole Jacobian by *seeding* input directions $X^{(1)}[i] \in \mathbb{R}^n$, $i = 0, \ldots, n-1$, with the Cartesian basis vectors in $\mathbb{R}^n$. Note concurrency!

Use dco/c++ to generate a first-order tangent version of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

and use it for the computation of the gradient.

Further Examples

1. Weird C++ ...

```
template<class T>
struct C { T a,b; };

template<class T>
C<T> operator+(const C<T>& x, const C<T>& y) {
        C<T> r;
        r.a=x.a/y.b; r.b=sin(x.b)*y.a;
        return r;
}

template<class T>
void f(T &x) { x=x+x; }
```

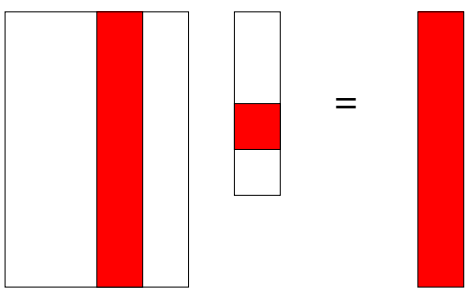
2. Heat Conduction
3. Option Pricing

A first-order adjoint code $F_{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^n$,

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{x}_{(1)} \\ \mathbf{y}_{(1)} \end{pmatrix} := F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}),$$

computes a shifted transposed Jacobian vector product (an adjoint directional derivative) alongside with the function value as follows:

$$\begin{aligned} \mathbf{y} &:= F(\mathbf{x}) \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \\ \mathbf{y}_{(1)} &:= 0 \end{aligned}$$



Define

$$\mathbf{v}_{(1)} \equiv \frac{dt}{d\mathbf{v}}^T$$

for $\mathbf{v} \in \{\mathbf{x}, \mathbf{y}\}$ and some auxiliary $t \in \mathbb{R}$ assuming that $t(F(\mathbf{x}))$ is continuously differentiable over D_F.

By the chain rule of differential calculus

$$\frac{dt}{d\mathbf{x}} = \frac{dt}{d\mathbf{y}} \cdot \frac{dF}{d\mathbf{x}} = \mathbf{y}_{(1)}^T \cdot \nabla F(\mathbf{x})$$

and hence

$$\mathbf{x}_{(1)} = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}.$$

Let

$$\mathbf{y} = F(\mathbf{x}) = \begin{pmatrix} F_0(\mathbf{x}) \\ F_1(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} \mathbf{y}_0(\mathbf{x}) \\ \mathbf{y}_1(\mathbf{x}) \end{pmatrix}$$

Clearly,

$$\begin{aligned} \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} &= \begin{pmatrix} \nabla F_0(\mathbf{x})^T & \nabla F_1(\mathbf{x})^T \end{pmatrix} \cdot \begin{pmatrix} \mathbf{y}_{0(1)}(\mathbf{x}) \\ \mathbf{y}_{1(1)}(\mathbf{x}) \end{pmatrix} \\ &= \nabla F_0(\mathbf{x})^T \cdot \mathbf{y}_{0(1)}(\mathbf{x}) + \nabla F_1(\mathbf{x})^T \cdot \mathbf{y}_{1(1)}(\mathbf{x}) \end{aligned}$$

Local (out of context) generation of adjoints of F_i requires incrementation of adjoint inputs due to possible use of inputs by other functions.

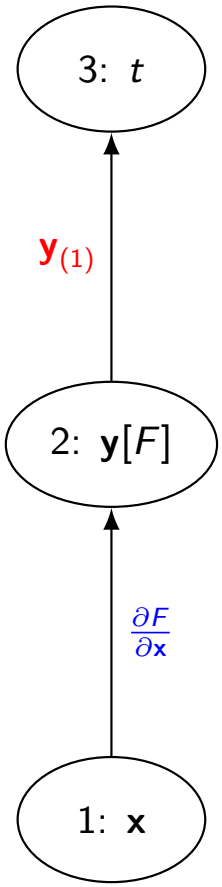
Let $\mathbf{y}$ be used as an input to a function evaluation which preceeds F. Its adjoint should be equal to zero prior to getting incremented. At the same time the memory location addressed by $\mathbf{y}_{(1)}$ potentially holds a nonzero adjoint prior to the evaulation of $F_{(1)}$. Correct incrementation can be achieved by resetting the adjoints of the results of a function equal to zero after the evlautaion of the adjoint version of this function. For example,

$$\mathbf{v} := G(\mathbf{y})$$
$$\mathbf{y} := F(\mathbf{v})$$

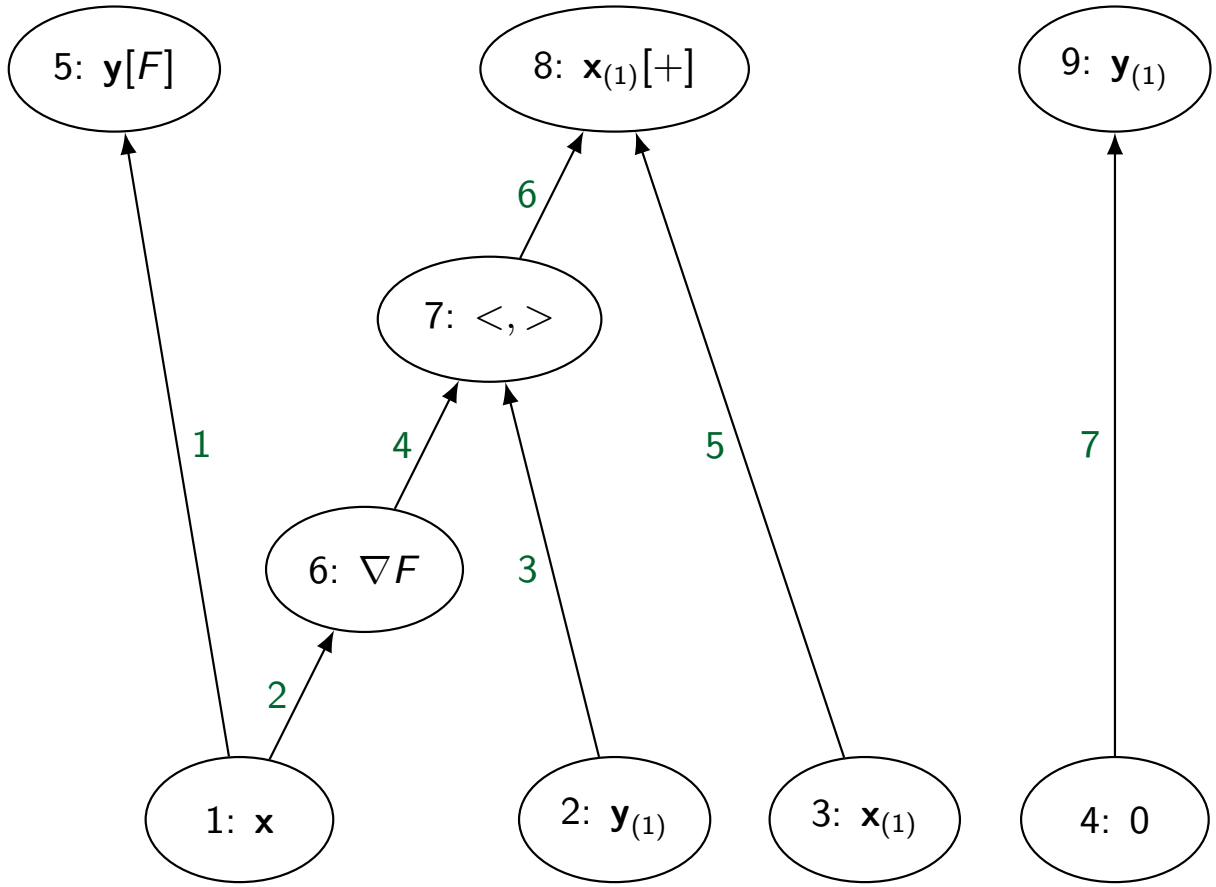
$$\mathbf{v}_{(1)} := \mathbf{v}_{(1)} + \frac{\partial F}{\partial \mathbf{v}}^T \cdot \mathbf{y}_{(1)}$$
$$\mathbf{y}_{(1)} := 0$$
$$\mathbf{y}_{(1)} := \mathbf{y}_{(1)} + \frac{\partial G}{\partial \mathbf{x}}^T \cdot \mathbf{v}_{(1)}$$

To miss the reset of $\mathbf{y}_{(1)}$ means to increment an incorrect initial adjoint of the input $\mathbf{y}$ to F.

adjoint-augmented IDAG



adjoint DAG



Note inner product notation $< \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) > \equiv \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$; see also [2].

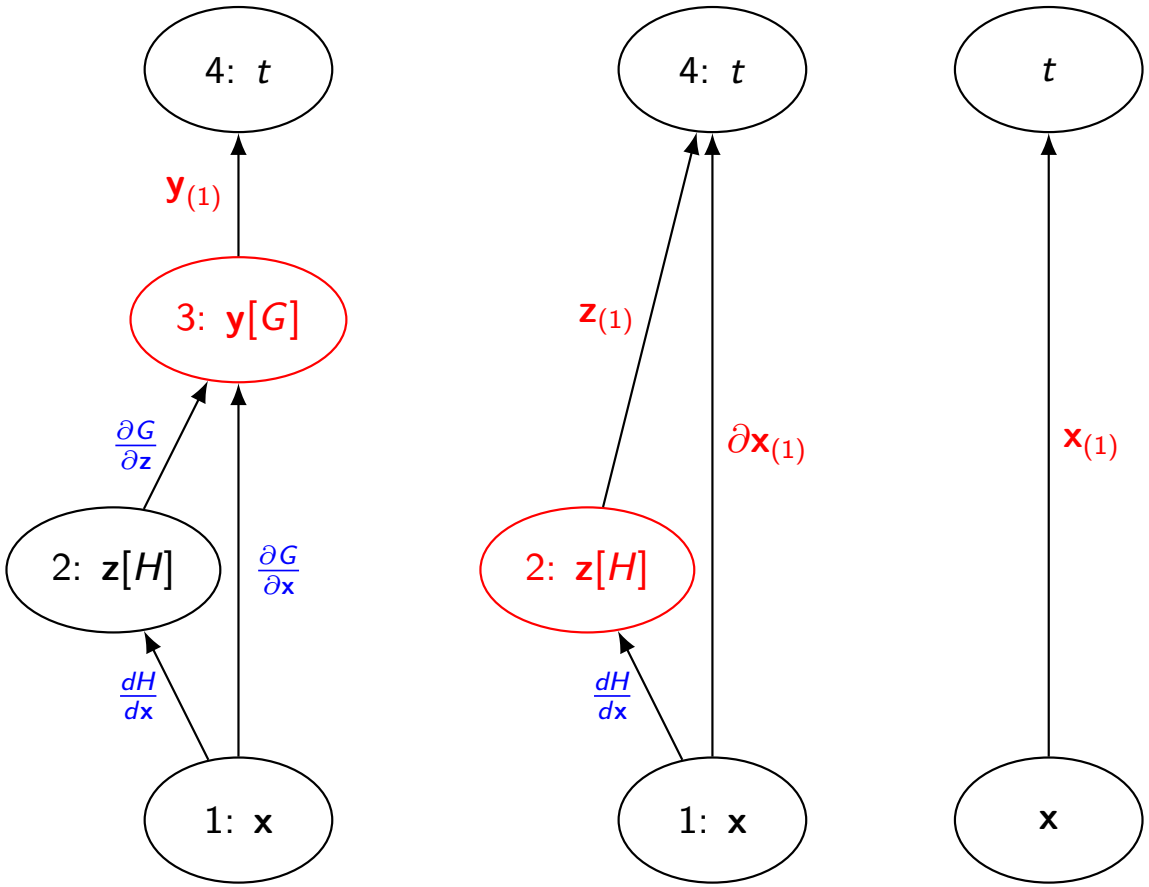
Let

$$\mathbf{y} = F(\mathbf{x}) = G(H(\mathbf{x}), \mathbf{x})$$

with $F : \mathbb{R}^n \to \mathbb{R}^m$, $H : \mathbb{R}^n \to \mathbb{R}^k$, $\mathbf{z} = H(\mathbf{x})$, and $G : \mathbb{R}^{n+k} \to \mathbb{R}^m$ continuously differentiable over their respective domains. By the chain rule

$$\begin{aligned}
\mathbf{x}_{(1)} \equiv \frac{dt}{d\mathbf{x}}^T = \frac{dF}{d\mathbf{x}}^T \cdot \mathbf{y}_{(1)} &= \left(\frac{dH}{d\mathbf{x}}^T \cdot \frac{\partial G}{\partial \mathbf{z}}^T + \frac{\partial G}{\partial \mathbf{x}}^T \right) \cdot \mathbf{y}_{(1)} \\
&= \frac{dH}{d\mathbf{x}}^T \cdot \left(\frac{\partial G}{\partial \mathbf{z}}^T \cdot \mathbf{y}_{(1)} \right) + \frac{\partial G}{\partial \mathbf{x}}^T \cdot \mathbf{y}_{(1)} \\
&= \frac{dH}{d\mathbf{x}}^T \cdot \left(\frac{\partial G}{\partial \mathbf{z}}^T \cdot \mathbf{y}_{(1)} \right) + \partial \mathbf{x}_{(1)} \\
&= \frac{dH}{d\mathbf{x}}^T \cdot \mathbf{z}_{(1)} + \partial \mathbf{x}_{(1)}
\end{aligned}$$

Graphically, this sequence of operations can be represented by a sequence of *vertex eliminations* on the corresponding adjoint-augmented IDAG as follows:



A vertex is eliminated by multiplying the labels of its incoming edges with the label(s) of the edge(s) emanating from its target (resulting in new edges or incrementation of existing edge labels) followed by its removal.
First-order adjoint code eliminates all eliminatable vertices in reverse topological order (storage of IDAG).

First-Order Adjoint Straight-Line Code

Bottom-up: Adjoints of elemental functions + chain rule, e.g.

- $y := \sin(x)$;

$$x_{(1)} \equiv \frac{\partial t}{\partial x} := x_{(1)} + \frac{\partial t}{\partial y} \cdot \frac{\partial y}{\partial x} = x_{(1)} + y_{(1)} \cdot \cos(x)$$

- $y := x_0 \cdot x_1$;

$$\mathbf{x}_{(1)} = \begin{pmatrix} x_{0(1)} \\ x_{1(1)} \end{pmatrix} \equiv \begin{pmatrix} \frac{\partial t}{\partial x_0} \\ \frac{\partial t}{\partial x_1} \end{pmatrix} := \mathbf{x}_{(1)} + \left(\frac{\partial t}{\partial y} \cdot \begin{pmatrix} \frac{\partial y}{\partial x_0} & \frac{\partial y}{\partial x_1} \end{pmatrix} \right)^T$$
$$= \mathbf{x}_{(1)} + \begin{pmatrix} x_1 \\ x_0 \end{pmatrix} \cdot y_{(1)} = \mathbf{x}_{(1)} + \begin{pmatrix} x_1 \cdot y_{(1)} \\ x_0 \cdot y_{(1)} \end{pmatrix}$$

Note accumulation gradient with 1×2 fmas[6] as opposed to 2×2 fmas when using tangent code $y^{(1)} = x_0^{(1)} \cdot x_1 + x_0 \cdot x_1^{(1)}$.

[6] fused multiply-add – the flop of the chain rule

First-Order Adjoint Straight-Line Code

- $y := \sin(x_0 \cdot x_1)$; Let $v := x_0 \cdot x_1$. Then

$$\mathbf{x}_{(1)} \equiv \begin{pmatrix} \frac{\partial t}{\partial x_0} \\ \frac{\partial t}{\partial x_1} \end{pmatrix} := \mathbf{x}_{(1)} + \left(\left(\frac{\partial t}{\partial y} \cdot \frac{\partial y}{\partial v} \right) \cdot \begin{pmatrix} \frac{\partial v}{\partial x_0} & \frac{\partial v}{\partial x_1} \end{pmatrix} \right)^T$$

$$= \mathbf{x}_{(1)} + (y_{(1)} \cdot \cos(v)) \cdot \begin{pmatrix} x_1 \\ x_0 \end{pmatrix}$$

Note reverse order of adjoint operations $\Rightarrow$ *data flow reversal*.

Compare accumulation of gradient using tangent

$$y^{(1)} := \cos(v) \cdot \left(x_0^{(1)} \cdot x_1 + x_0 \cdot x_1^{(1)} \right)$$

(requires 2×3 muls) and adjoint

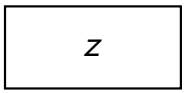
$$\mathbf{x}_{(1)} := (y_{(1)} \cdot \cos(v)) \cdot \begin{pmatrix} x_1 \\ x_0 \end{pmatrix}$$

(requires 1×3 muls).

First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$


```
void a1_f(a1s::type &[z,&TAPE]) {
  ...
}
...
```

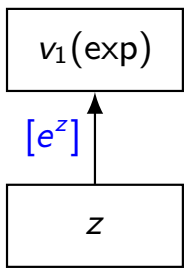


Taping is triggered by presence of entries representing the independent variables (also: sources in DAG; $\rightarrow$ registration).

First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

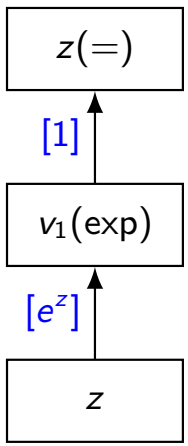

```
void a1_f(a1s::type &[z,&TAPE]) {
   ... exp([z,&TAPE]);
   ...
}
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

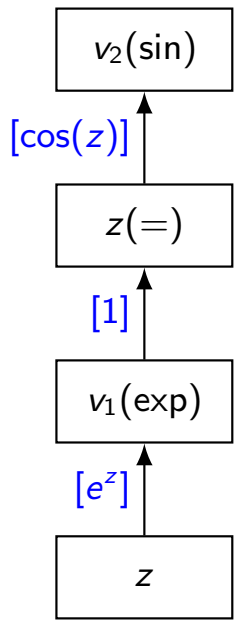

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  ...
}
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

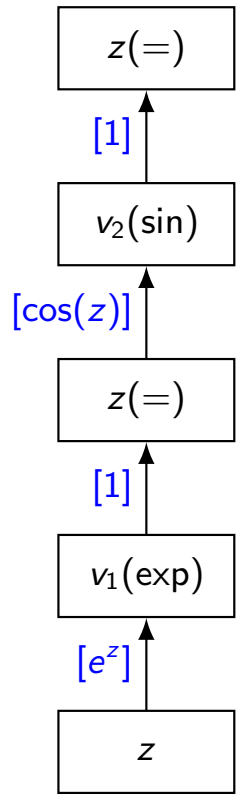

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  ... sin([z,&TAPE]);
}
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

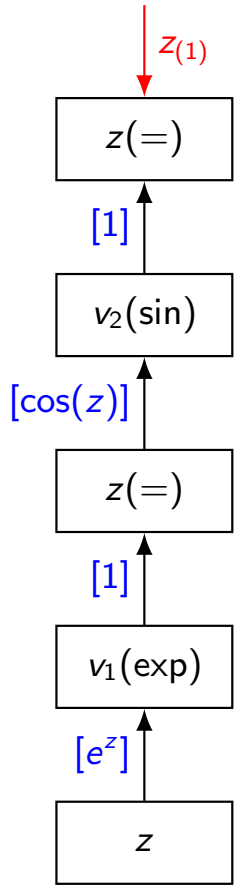

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  [z,&TAPE]=sin([z,&TAPE]);
}
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

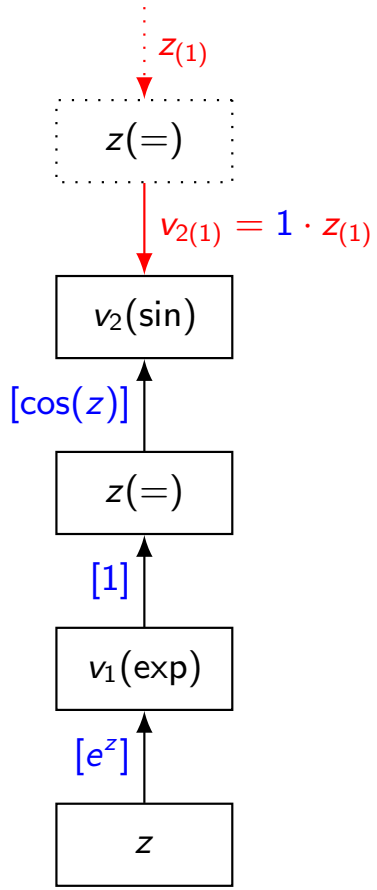

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  [z,&TAPE]=sin([z,&TAPE]);
}
...
a1_z=1;
TAPE.interpret(a1_z);
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

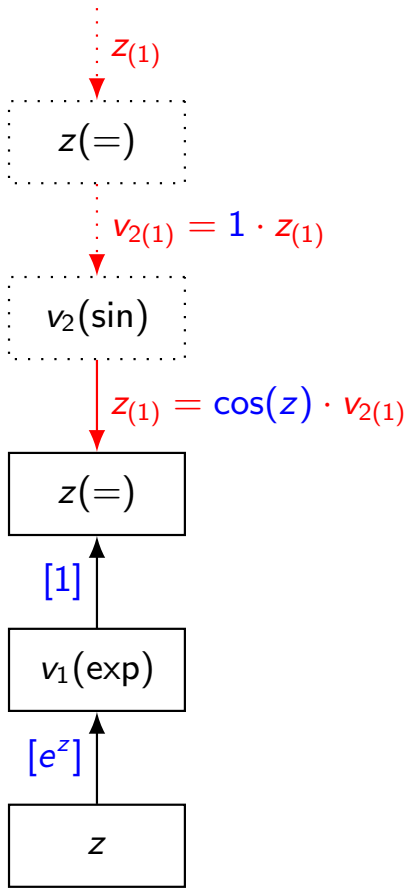

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  [z,&TAPE]=sin([z,&TAPE]);
}
...
a1_z=1;
TAPE.interpret(a1_z);
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

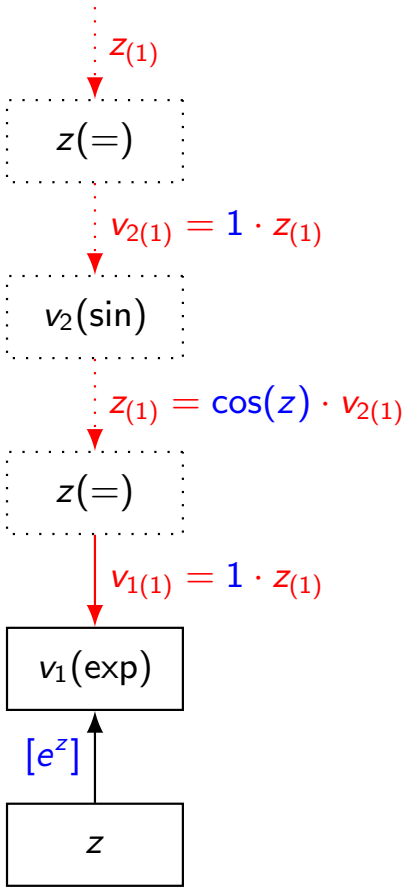

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  [z,&TAPE]=sin([z,&TAPE]);
}
...
a1_z=1;
TAPE.interpret(a1_z);
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$

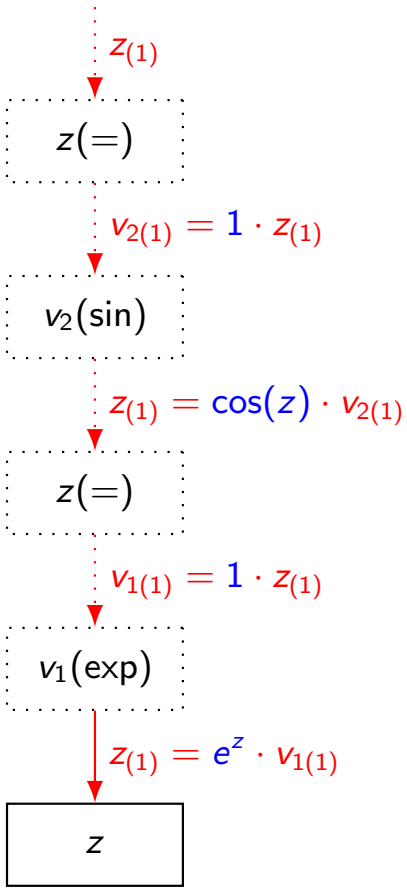

```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  [z,&TAPE]=sin([z,&TAPE]);
}
...
a1_z=1;
TAPE.interpret(a1_z);
...
```



First-Order Adjoint Code

Overloading: Conceptually for $y = f(x) = \sin(e^x)$


```
void a1_f(a1s::type &[z,&TAPE]) {
  [z,&TAPE]=exp([z,&TAPE]);
  [z,&TAPE]=sin([z,&TAPE]);
}
...
a1_z=1;
TAPE.interpret(a1_z);
cout << a1_z << endl;
...
```



The favored approach to a run-time version of the adjoint model is to build a *tape* (an augmented representation of the DAG) by overloading, followed by an interpretative reverse propagation of adjoints through the tape. In our case the tape is a statically allocated array of tape entries addressed by their position in the array.

```
class dco_a1s_tape_entry {
  public:
    int oc, arg1, arg2;
    double v,a;
    dco_a1s_tape_entry() :
      oc(DCO_A1S_UNDEF), arg1(DCO_A1S_UNDEF),
      arg2(DCO_A1S_UNDEF), v(0), a(0)
    {};
};
```

As in forward mode, an augmented data type is defined to replace the type of every active floating-point variable. The corresponding **class** dco_a1s_type (dco's adjoint 1st-order scalar type) contains the virtual address va (position in tape) of the current variable in addition to its value v.

```
class dco_a1s_type {
  public:
    int va;
    double v;
    dco_a1s_type() : va(DCO_A1S_UNDEF), v(0) {};
    dco_a1s_type(const double&);
    dco_a1s_type& operator=(const dco_a1s_type&);
};
```

```
dco_a1s_type&
dco_a1s_type::operator=(const dco_a1s_type& x) {
  if (this==&x) return *this;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ASG;
  dco_a1s_tape[dco_a1s_vac].v=v=x.v;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  va=dco_a1s_vac++;
  return *this;
}
```

```
dco_a1s_type
operator*(const dco_a1s_type& x1,
          const dco_a1s_type& x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_MUL;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v*x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}
```

```
dco_a1s_type sin(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SIN;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=sin(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}   .
```

```
void dco_a1s_interpret_tape () {
  for (int i=dco_a1s_vac;i>=0;i--) {
    switch (dco_a1s_tape[i].oc) {
      case DCO_A1S_ASG : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=
          dco_a1s_tape[i].a; break;
      }
      case DCO_A1S_SIN : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=
          cos(dco_a1s_tape[dco_a1s_tape[i].arg1].v)
            *dco_a1s_tape[i].a; break;
      }
      ...
    }
  }
}
```

```
#include <iostream>
#include "dco_a1s_type.hpp"
using namespace std;

const int n=4;

extern dco_a1s_tape_entry
  dco_a1s_tape[DCO_A1S_TAPE_SIZE];

void f(dco_a1s_type *x, dco_a1s_type &y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  y=y*y;
}
...
```

```
int main() {

  dco_a1s_type x[n], y;
  for (int j=0;j<n;j++) x[j]=1
  f(x,y);
  dco_a1s_tape[y.va].a=1;
  dco_a1s_interpret_tape();
  cout << i << "\t"
       << dco_a1s_tape[x[i].va].a << endl;
  dco_a1s_reset_tape();
  return 0;
}
```

First-Order Adjoint Code

Toy Problem with dco/c++

```
1  #include<iostream>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef ga1s<double> AD_MODE;
8  typedef AD_MODE::type AD_TYPE;
9  typedef AD_MODE::tape_t AD_TAPE_TYPE;
10 AD_TAPE_TYPE*& AD_TAPE_POINTER=AD_MODE::global_tape;
11
12 template<typename T>
13 void f(int n, const T *x, T& y ) { y=x[0]/x[1]; }
14
15 void driver(int n, const double* xv, double& yv, double* g) {
16   AD_TYPE *x=new AD_TYPE[n], y;
17   AD_TAPE_POINTER=AD_TAPE_TYPE::create();
18   for (int j=0;j<n;j++) {
19     x[j]=xv[j];
20     AD_TAPE_POINTER->register_variable(x[j]);
21   }
22   f(n,x,y);
23   AD_TAPE_POINTER->register_output_variable(y);
24   yv=value(y);
25   derivative(y)=1;
26   AD_TAPE_POINTER->interpret_adjoint();
```

```
27    for (int j=0;j<n;j++) g[j]=derivative(x[j]);
28    AD_TAPE_TYPE::remove(AD_TAPE_POINTER);
29    delete [] x;
30  }
31
32  int main() {
33    const int n=2;
34    double x[n]={2.71,3.14}, y, g[n];
35    driver(n,x,y,g);
36    cout << "y=" << y << endl;
37    cout << "g=[" << endl;
38    for (int i=0;i<n;i++) cout << g[i] << endl;
39    cout << "]" << endl;
40    return 0;
41  }
```

- ~/ga1s/ga1s_simple.cpp

Live:

- implementation of driver `fg` for given instantiation of

```
template<typename T>
void f(int n, T *x, T& y) {
    ...
}
```

  with `dco::ga1s<double>::type`.

- build and run

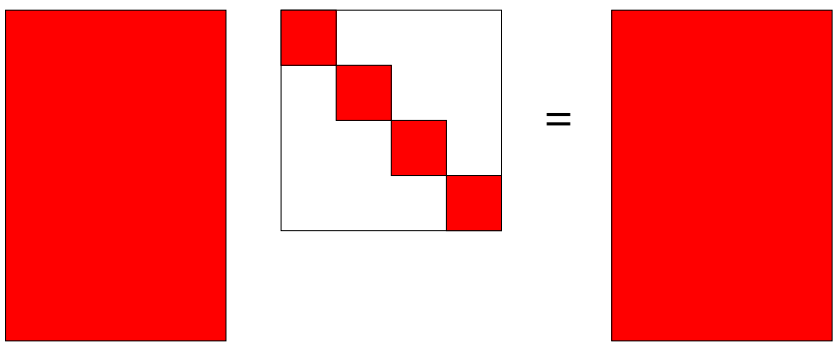
```
void fg(const double x[n], double& y, double g[n]) {
  A1S_TAPE_POINTER=A1S_MODE::tape::create();
  A1S_TYPE a1s_x[n], a1s_y;
  for (int i=0;i<n;i++) {
    a1s_x[i]=x[i];
    A1S_TAPE_POINTER->register_variable(a1s_x[i]);
  }
  f(a1s_x,a1s_y);
  A1S_TAPE_POINTER->register_output_variable(a1s_y);

  y=dco::get_value(a1s_y);
  dco::set_derivative(a1s_y,1);
  A1S_TAPE_POINTER->interpret_adjoint();
  for (int i=0;i<n;i++)
    g[i]=dco::get_derivative(a1s_x[i]);
}
```

A first-order vector adjoint code $F_{(1)} : \mathbb{R}^n \times \mathbb{R}^{n \times l} \times \mathbb{R}^{m \times l} \to \mathbb{R}^m \times \mathbb{R}^{n \times l} \times \mathbb{R}^{m \times l}$,

$$\begin{pmatrix} \mathbf{y} \\ X_{(1)} \\ Y_{(1)} \end{pmatrix} := F_{(1)}(\mathbf{x}, X_{(1)}, Y_{(1)}),$$

computes a shifted transposed Jacobian matrix product (an collection of adjoint directional derivatives) alongside with the function value as follows:

$$\begin{aligned} \mathbf{y} &:= F(\mathbf{x}) \\ X_{(1)} &:= X_{(1)} + \nabla F(\mathbf{x})^T \cdot Y_{(1)} \\ Y_{(1)} &:= 0 \end{aligned}$$



... accumulation of the whole Jacobian by *seeding* input directions $Y_{(1)}[i] \in \mathbb{R}^m$, $i = 0, \ldots, m-1$, with the Cartesian basis vectors in $\mathbb{R}^m$ and for $X_{(1)} = 0$ on input. Note concurrency!

Exercise

Use dco/c++ to generate a first-order adjoint version of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

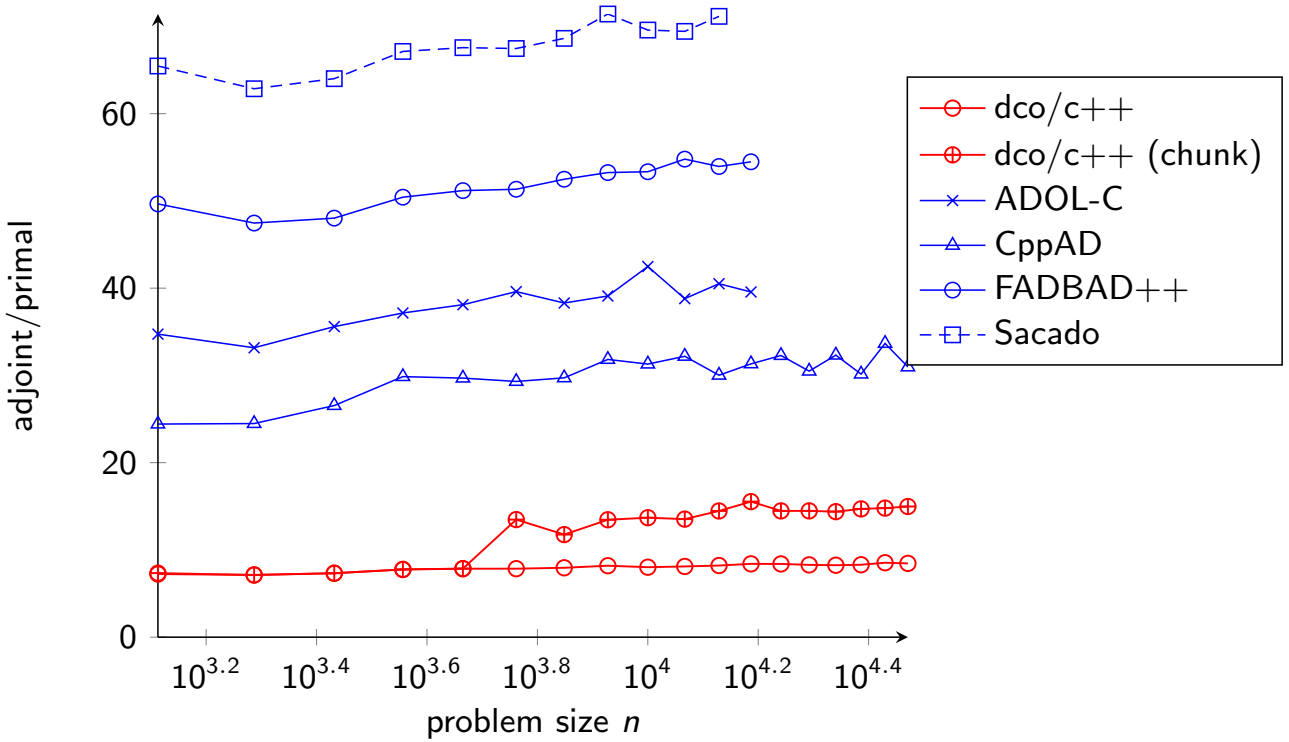
and use it for the computation of the gradient.

Further Examples

1. Weird C++ ...

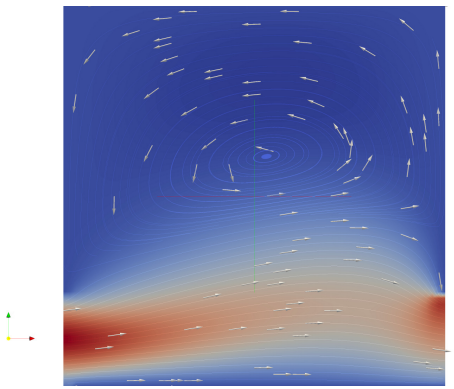
```
template<class T>
struct C { T a,b; };

template<class T>
C<T> operator+(const C<T>& x, const C<T>& y) {
        C<T> r;
        r.a=x.a/y.b; r.b=sin(x.b)*y.a;
        return r;
}

template<class T>
void f(T &x) { x=x+x; }
```

2. Heat Conduction
3. Option Pricing



text book reversal (no tricks!) within available memory of 10GB (RAM); max. problem size at $n \approx 16K$ vs. $n \approx 32K$

- in-house 3D unsteady incompressible flow simulation code by J. Lotz
- scenario
  - unit cube
  - zero initial condition
  - fixed inflow (Dirichlet boundary for velocity) at the lower left
  - outflow (Dirichlet boundary for pressure) at the lower right
  - cyclic z-boundary conditions



flow field after 1.5s colored by x-velocity

- equations are Boussinesq-approximation coupling momentum, energy, and mass conservation
- rectangular domain with equidistant Arakawa-C staggered grid
- SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm; explicit Euler time stepping scheme; Piascek-Williams scheme for the spacial discretization of the advection term; central finite differences for spacial discretization of the diffusion term
- linear system solved by matrix-free SOR (successive over-relaxation)

Second-(and Higher-)Order Tangent and Adjoint Code by Overloading

Without loss of generality, we consider multivariate scalar functions $y = F(\mathbf{x}) : D_F \subseteq \mathbb{R}^n \to I_F \subseteq \mathbb{R}$ in order to simplify the notation.

We assume F to be twice continuously differentiable over its domain D_F implying the existence of the Hessian

$$\nabla^2 F(\mathbf{x}) \equiv \frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}).$$

The Hessian is a three-tensor for multivariate vector functions complicating the notation slightly due to the need for tensor arithmetic; see [2].

A second-order *central finite difference* quotient

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}^0) \approx \Big[f(\mathbf{x}^0 + (\mathbf{e}_j + \mathbf{e}_i) \cdot h) - f(\mathbf{x}^0 + (\mathbf{e}_j - \mathbf{e}_i) \cdot h) - f(\mathbf{x}^0 + (\mathbf{e}_i - \mathbf{e}_j) \cdot h) + f(\mathbf{x}^0 - (\mathbf{e}_j + \mathbf{e}_i) \cdot h) \Big] / (4 \cdot h^2) \tag{1}$$

yields an approximation of the second directional derivative

$$y^{(1,2)} = \mathbf{x}^{(1)\,T} \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(2)} \quad \text{(w.l.o.g. } m = 1\text{)}$$

as

$$\begin{aligned} \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}^0) &\approx \frac{\frac{\partial f}{\partial x_i}(\mathbf{x}^0 + \mathbf{e}_j \cdot h) - \frac{\partial f}{\partial x_i}(\mathbf{x}^0 - \mathbf{e}_j \cdot h)}{2 \cdot h} \\ &= \left[\frac{f(\mathbf{x}^0 + \mathbf{e}_j \cdot h + \mathbf{e}_i \cdot h) - f(\mathbf{x}^0 + \mathbf{e}_j \cdot h - \mathbf{e}_i \cdot h)}{2 \cdot h} \right. \\ &\quad \left. - \frac{f(\mathbf{x}^0 - \mathbf{e}_j \cdot h + \mathbf{e}_i \cdot h) - f(\mathbf{x}^0 - \mathbf{e}_j \cdot h - \mathbf{e}_i \cdot h)}{2 \cdot h} \right] / (2 \cdot h). \end{aligned}$$

Second-Order Finite Differences

Example

Use second-order finite differences to approximate the Hessian for

```
void f(int n, double *x, double& y) {
  int i=0;
  while (i<n) {
    x[i]=x[i]*x[i];
    i=i+1;
  }
  i=0;
  while (i<n) {
    if (i==0)
      y=x[i];
    else
      y=y+x[i];
    i=i+1;
  }
  y=y*y;
}
```

- `~/fd/sofd.cpp`

Can we avoid truncation?

YES, WE CAN!

$\rightarrow$ For the given tangent model of $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, implement its tangent model

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

At most n^2 runs of the second-order tangent code are required to obtain the whole Hessian with machine accuracy.

- `~/gt2s_gt1s/gt2s_gt1s.cpp`

Can we reduce the computational cost?

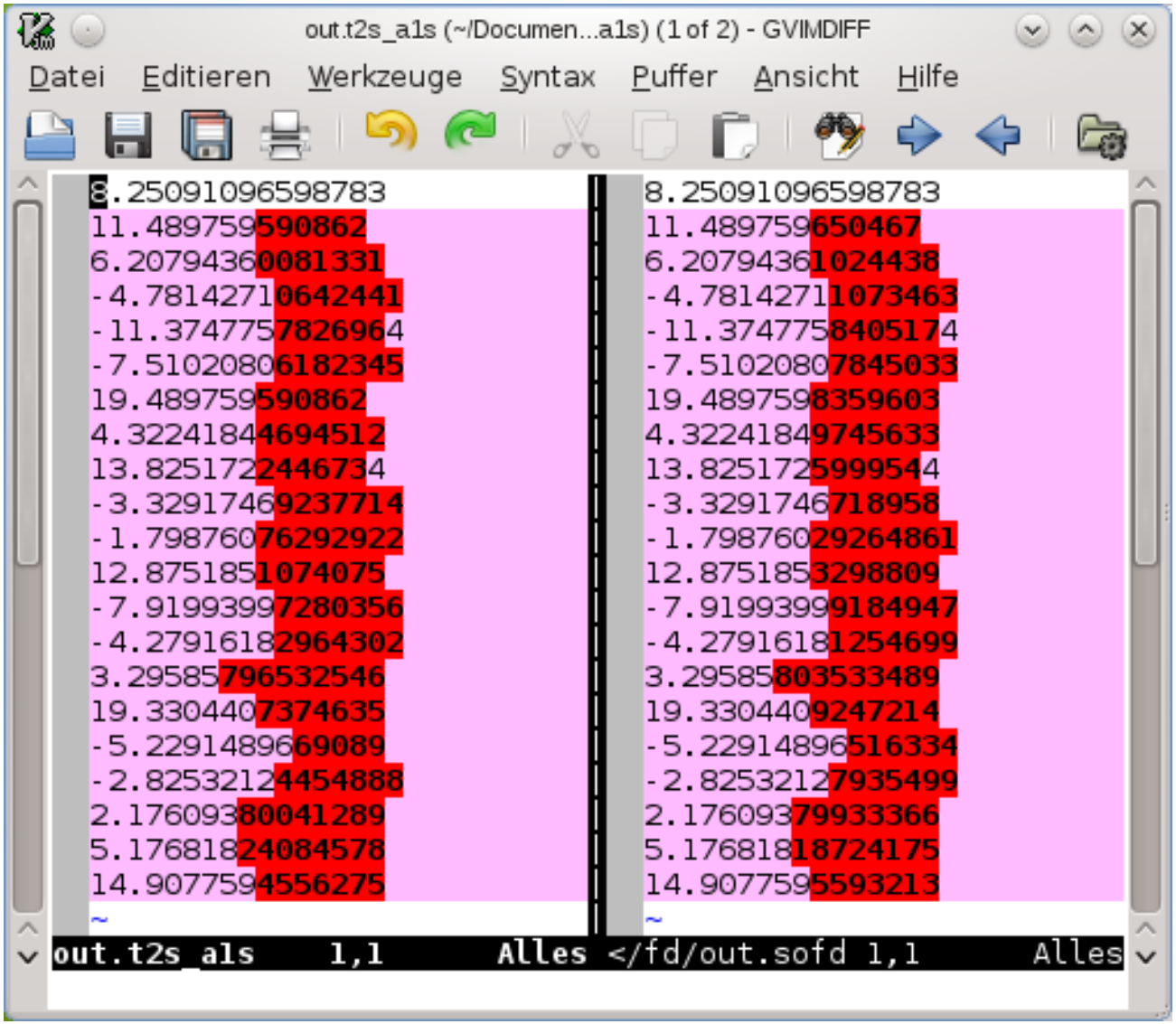
YES, WE CAN!

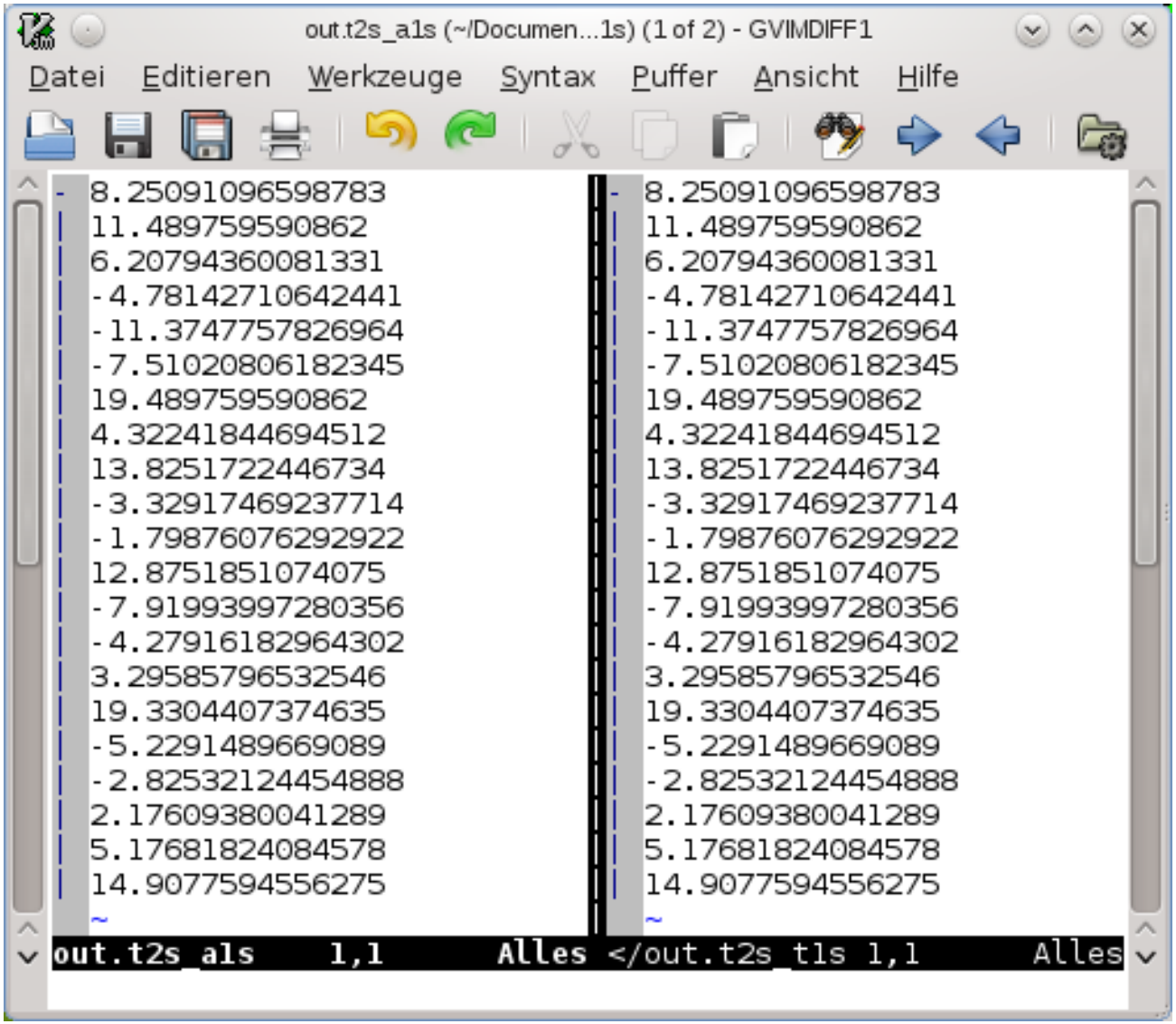
$\rightarrow$ For a given adjoint model of $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, implement its tangent model

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

At most $m \cdot n$ runs of the second-order adjoint code are required to obtain the whole Hessian with machine accuracy.

- `~/gt2s_ga1s/gt2s_ga1s.cpp`


out.t2s_a1s (~/Documen...a1s) (1 of 2) - GVIMDIFF
Datei Editieren Werkzeuge Syntax Puffer Ansicht Hilfe
8.25091096598783
11.489759590862
6.2079436008133 1
-4.781427106424 41
-11.37477578269 64
-7.5102080618234 5
19.489759590862
4.3224184469451 2
13.825172244673 4
-3.3291746923771 4
-1.7987607629292 2
12.875185107407 5
-7.9199399728035 6
-4.2791618296430 2
3.2958579653254 6
19.330440737463 5
-5.2291489669089
-2.8253212445488 8
2.1760938004128 9
5.1768182408457 8
14.907759455627 5
~
out.t2s_a1s 1,1 Alles
8.25091096598783
11.489759650467
6.2079436102443 8
-4.781427107346 3
-11.37477584051 74
-7.5102080784503 3
19.489759835960 3
4.3224184974563 3
13.825172599954 4
-3.3291746718958
-1.7987602926486 1
12.875185329880 9
-7.9199399918494 7
-4.2791618125469 9
3.2958580353348 9
19.330440924721 4
-5.2291489651633 4
-2.8253212793549 9
2.1760937993336 6
5.1768181872417 5
14.907759559321 3
~
</fd/out.sofd 1,1 Alles


out.t2s_a1s (~/Documen...1s) (1 of 2) - GVIMDIFF1
Datei Editieren Werkzeuge Syntax Puffer Ansicht Hilfe
8.25091096598783
11.489759590862
6.20794360081331
-4.78142710642441
-11.3747757826964
-7.51020806182345
19.489759590862
4.32241844694512
13.8251722446734
-3.32917469237714
-1.79876076292922
12.8751851074075
-7.91993997280356
-4.27916182964302
3.29585796532546
19.3304407374635
-5.2291489669089
-2.82532124454888
2.17609380041289
5.17681824084578
14.9077594556275
~
8.25091096598783
11.489759590862
6.20794360081331
-4.78142710642441
-11.3747757826964
-7.51020806182345
19.489759590862
4.32241844694512
13.8251722446734
-3.32917469237714
-1.79876076292922
12.8751851074075
-7.91993997280356
-4.27916182964302
3.29585796532546
19.3304407374635
-5.2291489669089
-2.82532124454888
2.17609380041289
5.17681824084578
14.9077594556275
~
out.t2s_a1s 1,1 Alles
</out.t2s_t1s 1,1 Alles

Use second-order central finite differences for the approximation of the Hessian of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

A second-order tangent code $F^{(1,2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m$,

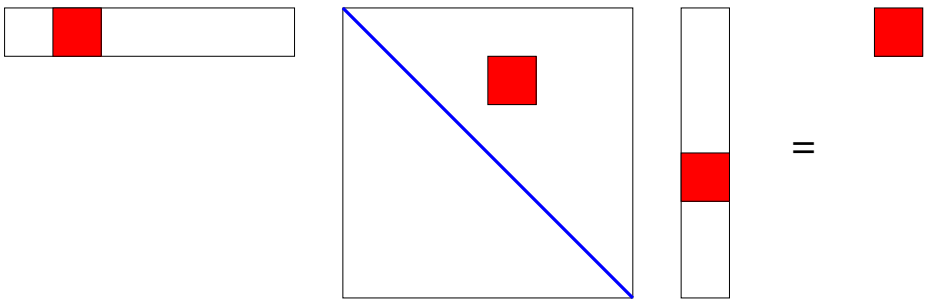
$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(1,2)} \end{pmatrix} = F^{(1,2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}, \mathbf{x}^{(1,2)}),$$

computes a mixture of first and second derivative information alongside with the function value as follows:

$$\begin{aligned} y &:= F(\mathbf{x}) \\ y^{(2)} &:= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(2)} \\ y^{(1)} &:= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ y^{(1,2)} &:= \mathbf{x}^{(1)\,T} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)} + \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1,2)} \end{aligned}$$

Second-Order Tangent

Accumulation of Hessian (Complexity)



$$\mathbf{x}^{(1)\,T} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

Define

$$\mathbf{v}^{(2)} \equiv \frac{\partial \mathbf{v}}{\partial s}$$

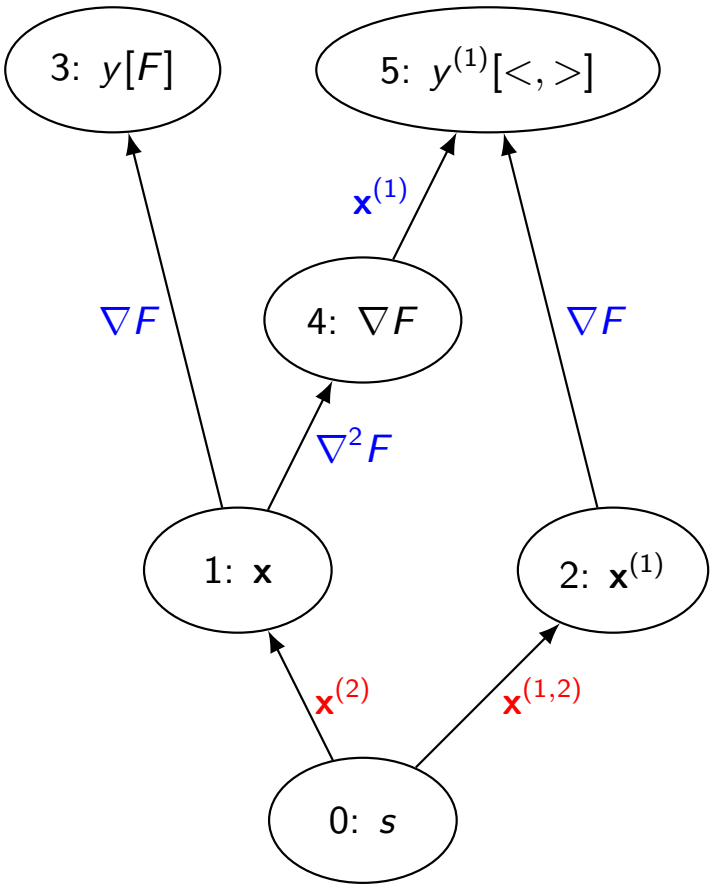
for $\mathbf{v} \in \{\mathbf{x}, \mathbf{x}^{(1)}, y, y^{(1)}\}$ and some auxiliary $s \in D_s \subseteq \mathbb{R}$ assuming that $F^{(1)}(\mathbf{x}(s), \mathbf{x}^{(1)}(s))$ is continuously differentiable over D_s. Set $\mathbf{v}^{(1)(2)} = \mathbf{v}^{(1,2)}$.

By the chain rule of differential calculus

$$\begin{aligned}
\frac{\partial(\nabla F(\mathbf{x}) \cdot \overset{\in\mathbb{R}}{\mathbf{x}^{(1)}})}{\partial s} &= \frac{\partial({\mathbf{x}^{(1)}}^T \cdot \nabla F(\mathbf{x})^T)}{\partial s} \\
&= \frac{\partial {\mathbf{x}^{(1)}}^T}{\partial s} \cdot \nabla F(\mathbf{x})^T + {\mathbf{x}^{(1)}}^T \cdot \frac{\partial \nabla F(\mathbf{x})}{\partial s} \\
&= {\mathbf{x}^{(1,2)}}^T \cdot \nabla F(\mathbf{x})^T + {\mathbf{x}^{(1)}}^T \cdot \frac{\partial \nabla F(\mathbf{x})^T}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s} \\
&= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1,2)} + {\mathbf{x}^{(1)}}^T \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}
\end{aligned}$$

Recall:

$$\frac{\partial A(\mathbf{x}) \cdot B(\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial A(\mathbf{x})}{\partial \mathbf{x}} \cdot B(\mathbf{x}) + A(\mathbf{x}) \cdot \frac{\partial B(\mathbf{x})}{\partial \mathbf{x}}$$



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

Second-Order Tangent Code

Toy Problem with dco/c++

```
1  #include<iostream>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef gt1s<double> AD_BASE_MODE;
8  typedef AD_BASE_MODE::type AD_BASE_TYPE;
9  typedef gt1s<AD_BASE_TYPE> AD_MODE;
10 typedef AD_MODE::type AD_TYPE;
11
12 template<typename T>
13 void f(int n, const T *x, T& y) { y=x[0]/x[1]; }
14
15 void driver(int n, const double* xv, double& yv, double* g, double* h)  {
16   AD_TYPE *x=new AD_TYPE[n], y;
17   for (int i=0,c=0;i<n;i++) {
18     for (int j=0;j<=i;j++) {
19       for (int k=0;k<n;k++) x[k]=xv[k];
20       derivative(value(x[i]))=1;
21       value(derivative(x[j]))=1;
22       f(n,x,y);
23       h[c++]=derivative(derivative(y));
24     }
25     g[i]=derivative(value(y));
26   }
```

```
27    yv=passive_value(y);
28    delete [] x;
29  }
30
31  int main() {
32    const int n=2;
33    double x[n]={2.71,3.14}, y, h[n*(n+1)/2], g[n];
34    cout.precision(15);
35    driver(n,x,y,g,h);
36    cout << "y=" << y << endl;
37    for (int i=0;i<n;i++)
38      cout << "g[" << i << "]=" << g[i] << endl;
39    cout << "h=[" << endl;
40    for (int i=0,c=0;i<n;i++) {
41      for (int j=0;j<=i;j++)
42        cout << h[c++] <<"\t";
43      cout << endl;
44    }
45    cout << "]" << endl;
46    return 0;
47  }
```

- ~/gt2s_gt1s/gt2s_gt1s_simple.cpp

Live:

- implementation of driver fgh for given instantiation of

```
template<class T>
void f(int n, T *x, T& y) {
    ...
}
```

  with dco::gt1s<dco::gt1s<**double**>::type>::type.

- build and run

Second-Order Tangent Code

Motivational Example with dco/c++ (Driver)

```
void fgh(int n, const double* x, double& y, double* g, double* h) {
  T2S_T1S_TYPE *t1s_x=new T2S_T1S_TYPE[n], t1s_y;
  T1S_TYPE v;

  int l=0;
  for (int i=0;i<n;i++) {
    for (int j=0;j<=i;j++) {
      for (int k=0;k<n;k++) t1s_x[k]=x[k];

      dco::set_derivative(dco::get_value(t1s_x[i]),1);
      dco::set_value(dco::get_derivative(t1s_x[j]),1);
      f(n,t1s_x,t1s_y);
      h[l++]=dco::get_derivative(dco::get_derivative(t1s_y));
    }
    g[i]=dco::get_value(dco::get_derivative(t1s_y));
  }
  y=dco::get_value(dco::get_value(t1s_y));
  delete [] t1s_x;
}
```

Use dco/c++ to generate a second-order tangent version of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

and use it for the computation of the Hessian.

A second-order adjoint code generated in tangent over adjoint mode

$$F_{(1)}^{(2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R},$$

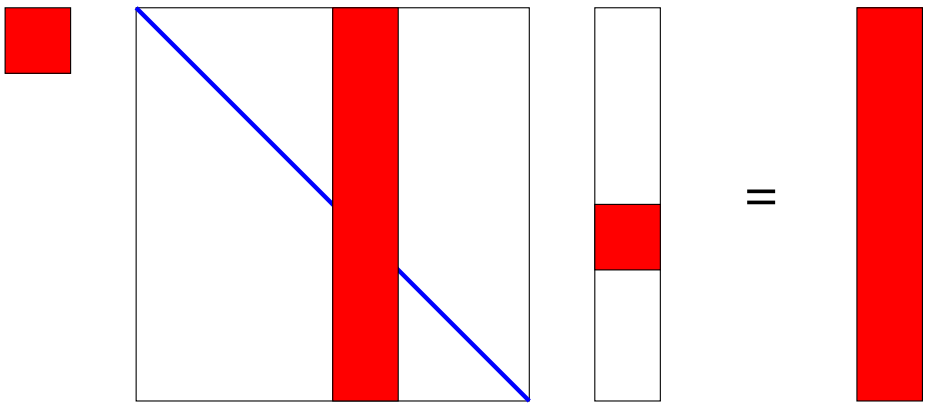
$$\begin{pmatrix} y \\ y^{(2)} \\ \mathbf{x}_{(1)} \\ \mathbf{x}_{(1)}^{(2)} \\ y_{(1)} \\ y_{(1)}^{(2)} \end{pmatrix} = F_{(1)}^{(2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(1)}^{(2)}, y_{(1)}, y_{(1)}^{(2)}),$$

computes

$$\begin{aligned}
y &:= F(\mathbf{x}) \\
y^{(2)} &:= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(2)} \\
\mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot y_{(1)} \\
\mathbf{x}_{(1)}^{(2)} &:= \mathbf{x}_{(1)}^{(2)} + y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(1)}^{(2)} \\
y_{(1)} &:= 0 \\
y_{(1)}^{(2)} &:= 0
\end{aligned}$$

Second-Order Adjoint (Tangent over Adjoint)

Accumulation of Hessian (Complexity)



$$y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

Consider

$$\mathbf{v}^{(2)} \equiv \frac{\partial \mathbf{v}}{\partial s}$$

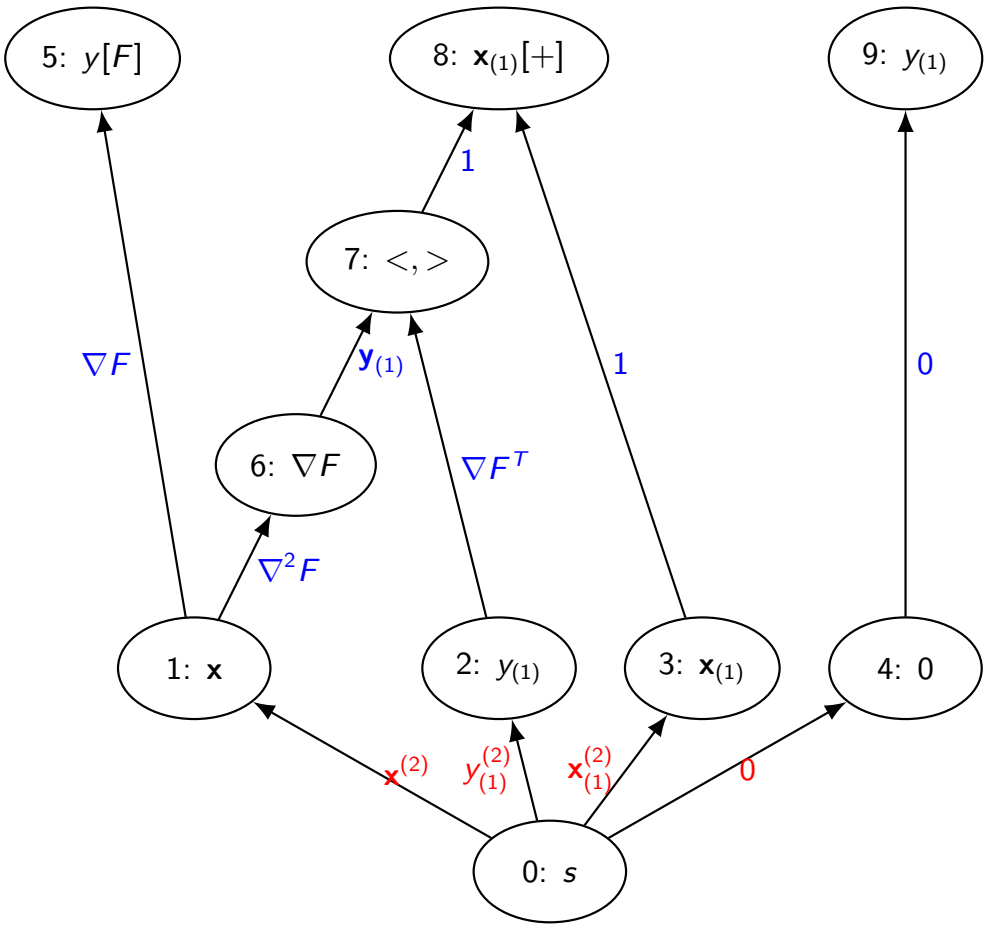
for $\mathbf{v} \in \{\mathbf{x}, \mathbf{x}_{(1)}, y, y_{(1)}\}$ and assuming that $F_{(1)}(\mathbf{x}(s), \mathbf{x}_{(1)}(s), y_{(1)}(s)$ is continuously differentiable over D_s.

By the chain rule of differential calculus

$$\begin{aligned}
\frac{\partial(\nabla F(\mathbf{x})^T \cdot y_{(1)})}{\partial s} &= \frac{\partial \nabla F(\mathbf{x})^T}{\partial s} \cdot y_{(1)} + \nabla F(\mathbf{x})^T \cdot \frac{\partial y_{(1)}}{\partial s} \\
&= \frac{\partial \nabla F(\mathbf{x})^T}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s} \cdot y_{(1)} + \nabla F(\mathbf{x})^T \cdot y_{(1)}^{(2)} \\
&= y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(1)}^{(2)}
\end{aligned}$$

Second-Order Adjoint (Tangent over Adjoint)

Tangent-Augmented Adjoint IDAG



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

$$y^{(2)} := \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

$$\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot y_{(1)}$$

$$\mathbf{x}^{(2)}_{(1)} := \mathbf{x}^{(2)}_{(1)} + y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)} + \nabla F(\mathbf{x})^T \cdot y^{(2)}_{(1)}$$

Second-Order Adjoint Code (Tangent over Adjoint)

Toy Problem with dco/c++

```
1  #include<iostream>
2  using namespace std;
3
4  #include "dco.hpp"
5  using namespace dco;
6
7  typedef gt1s<double> AD_BASE_MODE;
8  typedef AD_BASE_MODE::type AD_BASE_TYPE;
9  typedef ga1s<AD_BASE_TYPE> AD_MODE;
10 typedef AD_MODE::type AD_TYPE;
11 typedef AD_MODE::tape_t AD_TAPE_TYPE;
12 AD_TAPE_TYPE*& AD_TAPE_POINTER=AD_MODE::global_tape;
13
14 template<typename T>
15 void f(int n, const T *x, T& y ) { y=x[0]/x[1]; }
16
17 void driver(int n, const double* xv, double& yv, double* g, double* h) {
18   AD_TAPE_POINTER=AD_TAPE_TYPE::create();
19   AD_TYPE *x=new AD_TYPE[n], *x_in=new AD_TYPE[n], y;
20   for (int i=0,c=0;i<n;i++) {
21     for (int j=0;j<n;j++) {
22       x[j]=xv[j];
23       AD_TAPE_POINTER->register_variable(x[j]);
24       x_in[j]=x[j];
25     }
26     derivative(value(x[i]))=1;
```

```
27      f(n,x,y);
28      AD_TAPE_POINTER->register_output_variable(y);
29      value(derivative(y))=1;
30      AD_TAPE_POINTER->interpret_adjoint();
31      for (int j=0;j<=i;j++)
32        h[c++]=derivative(derivative(x_in[j]));
33      AD_TAPE_POINTER->reset();
34      g[i]=derivative(value(y));
35    }
36    yv=value(value(y));
37    AD_TAPE_TYPE::remove(AD_TAPE_POINTER);
38  }
39
40  int main() {
41    const int n=2;
42    double x[n]={2.71,3.14},y,g[n],h[n*(n+1)/2];
43    cout.precision(15);
44    driver(n,x,y,g,h);
45    cout << "y=" << y << endl;
46    for (int i=0;i<n;i++)
47      cout << "g[" << i << "]=" << g[i] << endl;
48    cout << "h=[" << endl;
49    for (int i=0,c=0;i<n;i++) {
50      for (int j=0;j<=i;j++)
51        cout << h[c++] <<"\t";
52      cout << endl;
```

Second-Order Adjoint Code (Tangent over Adjoint)

Toy Problem with dco/c++ (cont.)

```
53     }
54     cout << "]" << endl;
55     return 0;
56 }
```

- ~/gt2s_ga1s/gt2s_ga1s_simple.cpp

Live:

- implementation of driver fgh for given

```
template<class T>
void f(int n, T *x, T& y) {
    ...
}
```

- build and run

```
void driver(const double x[n], double& y, double h[][2]) {
  T2S_A1S_TAPE_POINTER=T2S_A1S_MODE::tape::create();
  T2S_A1S_TYPE t2s_a1s_x[n], t2s_a1s_y;
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) {
      t2s_a1s_x[j]=x[j];
      T2S_A1S_TAPE_POINTER->register_variable(t2s_a1s_x[j]);
    }
    dco::set_derivative(dco::get_value(t2s_a1s_x[i]),1);
    f(t2s_a1s_x,t2s_a1s_y);
    T2S_A1S_TAPE_POINTER->register_output_variable(t2s_a1s_y);

    dco::set_value(dco::get_derivative(t2s_a1s_y),1);
    T2S_A1S_TAPE_POINTER->interpret_adjoint();
    for (int j=0;j<n;j++)
      h[i][j]=dco::get_derivative(dco::get_derivative(t2s_a1s_x[j]));
    T2S_A1S_TAPE_POINTER->reset();
  }
  y=dco::get_value(dco::get_value(t2s_a1s_y));
  T2S_A1S_MODE::tape::remove(T2S_A1S_TAPE_POINTER);
}
```

Use dco/c++ to generate a second-order adjoint (tangent over adjoint) version of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

and use it for the computation of the Hessian.

A second-order adjoint code generated in adjoint over tangent mode

$$F_{(2)}^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R},$$

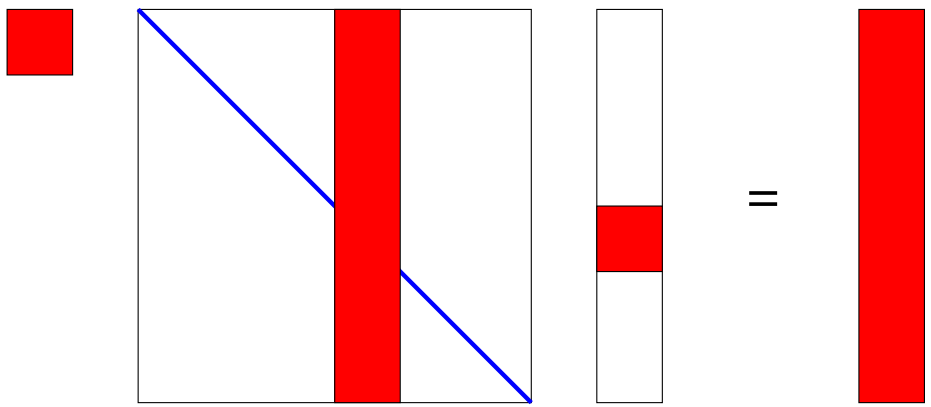
$$\begin{pmatrix} y \\ y^{(1)} \\ \mathbf{x}_{(2)} \\ \mathbf{x}_{(2)}^{(1)} \\ y_{(2)} \\ y_{(2)}^{(1)} \end{pmatrix} = F_{(2)}^{(1)}(\mathbf{x}, \mathbf{x}_{(2)}, \mathbf{x}^{(1)}, \mathbf{x}_{(2)}^{(1)}, y_{(2)}, y_{(2)}^{(1)}),$$

computes

$$\begin{aligned} y &:= F(\mathbf{x}) \\ y^{(1)} &:= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ \mathbf{x}_{(2)} &:= \mathbf{x}_{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(2)} + y_{(2)}^{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ \mathbf{x}_{(2)}^{(1)} &:= \mathbf{x}_{(2)}^{(1)} + \nabla F(\mathbf{x})^T \cdot y_{(2)}^{(1)} \\ y_{(2)} &:= 0 \\ y_{(2)}^{(1)} &:= 0 \end{aligned}$$

Second-Order Adjoint (Adjoint over Tangent)

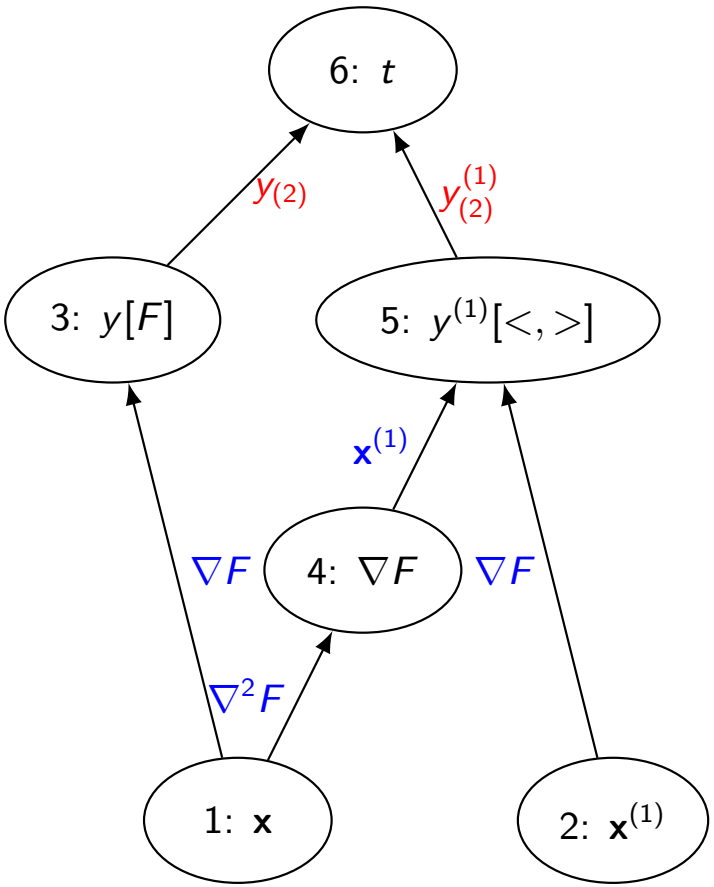
Accumulation of Hessian (Complexity)



$$y_{(1)}^{(2)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$$

Second-Order Adjoint (Adjoint over Tangent)

Adjoint-Augmented Tangent IDAG



$$
\begin{aligned}
y &:= F(\mathbf{x}) \\
y^{(1)} &:= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\
\mathbf{x}_{(2)} &:= \mathbf{x}_{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(2)} + y^{(2)}_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\
\mathbf{x}^{(2)}_{(1)} &:= \mathbf{x}^{(2)}_{(1)} + \nabla F(\mathbf{x})^T \cdot y^{(2)}_{(1)}
\end{aligned}
$$

Second-Order Adjoint Code (Adjoint over Tangent)

Toy Problem with dco/c++


```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef ga1s<double> AD_BASE_MODE;
9  typedef AD_BASE_MODE::type AD_BASE_TYPE;
10 typedef gt1s<AD_BASE_TYPE> AD_MODE;
11 typedef AD_MODE::type AD_TYPE;
12 typedef AD_BASE_MODE::tape_t AD_TAPE_TYPE;
13 AD_TAPE_TYPE*& AD_TAPE_POINTER=AD_BASE_MODE::global_tape;
14
15 template<typename T>
16 void f(int n, const T *x, T& y ) { y=x[0]/x[1]; }
17
18 void driver(int n, const double* xv, double& yv, double* g, double* h) {
19   AD_TAPE_POINTER=AD_TAPE_TYPE::create();
20   AD_TYPE *x=new AD_TYPE[n]; AD_TYPE *x_in=new AD_TYPE[n];
21   AD_TYPE y;
22   for (int i=0,c=0;i<n;i++) {
23     for (int j=0;j<n;j++) {
24       x[j]=xv[j];
25       AD_TAPE_POINTER->register_variable(value(x[j]));
26       AD_TAPE_POINTER->register_variable(derivative(x[j]));
```

```
27          x_in[j]=x[j];
28        }
29        value(derivative(x[i]))=1;
30        f(n,x,y);
31        AD_TAPE_POINTER->register_output_variable(value(y));
32        AD_TAPE_POINTER->register_output_variable(derivative(y));
33        derivative(derivative(y))=1;
34        AD_TAPE_POINTER->interpret_adjoint();
35        for (int j=0;j<=i;j++)
36          h[c++] = derivative(value(x_in[j]));
37        AD_TAPE_POINTER->reset();
38        g[i]=value(derivative(y));
39      }
40      yv=passive_value(y);
41      AD_TAPE_TYPE::remove(AD_TAPE_POINTER);
42      delete [] x; delete [] x_in;
43    }
44
45    int main() {
46      const int n=2;
47      double x[n]={2.71,3.14}, y, g[n], h[n*(n+1)/2];
48      cout.precision(15);
49      driver(n,x,y,g,h);
50      cout << "y=" << y << endl;
51      for (int i=0;i<n;i++)
52        cout << "g[" << i << "]=" << g[i] << endl;
```

Second-Order Adjoint Code (Adjoint over Tangent)

Toy Problem with dco/c++ (cont.)

```
53    cout << "h=[" << endl;
54    for (int i=0,c=0;i<n;i++) {
55      for (int j=0;j<=i;j++)
56        cout << h[c++] <<"\t";
57      cout << endl;
58    }
59    cout << "]" << endl;
60    return 0;
61  }
```

- ~/ga2s_gt1s/ga2s_gt1s_simple.cpp

Live:

- implementation of driver fgh for given

```
template<class T>
void f(vector<T> &x, T& y) {
    ...
}
```

- build and run

Second-Order Adjoint (Adjoint over Tangent)

Motivational Example with dco/c++ (Driver)

```
void fgh(int n, const double* x, double& y, double* g, double* h) {
    A1S_TAPE_POINTER=A1S_MODE::tape::create();
    vector<A2S_T1S_TYPE> t2s_a1s_x(n), t2s_a1s_x_in(n);
    A2S_T1S_TYPE t2s_a1s_y;
    A1S_TYPE v;
    int l=0;
    for (int i=0;i<n;i++) {
        for (int j=0;j<n;j++) {
            t2s_a1s_x[j]=x[j];
            A1S_TAPE_POINTER->register_variable(dco::get_value(t2s_a1s_x[j
                ]));
            A1S_TAPE_POINTER->register_variable(dco::get_derivative(
                t2s_a1s_x[j]));
            t2s_a1s_x_in[j]=t2s_a1s_x[j];
        }
        dco::set_value(dco::get_derivative(t2s_a1s_x[i]),1);
        f(t2s_a1s_x,t2s_a1s_y);

        A1S_TAPE_POINTER->register_output_variable(dco::get_value(t2s_a1s_y
            ));
        A1S_TAPE_POINTER->register_output_variable(dco::get_derivative(
            t2s_a1s_y));

        dco::set_derivative(dco::get_derivative(t2s_a1s_y),1);

        A1S_TAPE_POINTER->interpret_adjoint();
```

```
        for (int j=0;j<=i;j++)
            h[l++] = dco::get_derivative(dco::get_value(t2s_a1s_x_in[j]));

        A1S_TAPE_POINTER->reset();
        g[i]=dco::get_value(dco::get_derivative(t2s_a1s_y));
    }
    y = dco::get_passive_value(t2s_a1s_y);

    A1S_MODE::tape::remove(A1S_TAPE_POINTER);
}
```

Exercise

Use dco/c++ to generate a second-order adjoint (adjoint over tangent) version of the product reduction

$$y = \prod_{i=0}^{n-1} x_i$$

implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

and use it for the computation of the Hessian.

A second-order adjoint code generated in adjoint over adjoint mode

$$F_{(1,2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R},$$

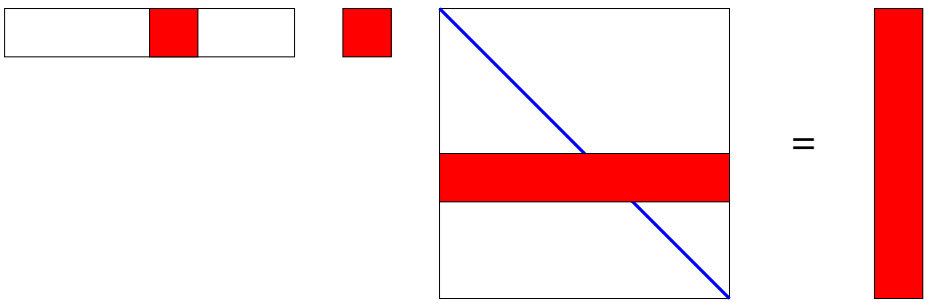
$$\begin{pmatrix} y \\ \mathbf{x}_{(1)} \\ y_{(1)} \\ \mathbf{x}_{(2)} \\ y_{(1,2)} \\ y_{(2)} \end{pmatrix} = F_{(1,2)}(\mathbf{x}, \mathbf{x}_{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(1,2)}, y_{(1)}, y_{(2)}),$$

computes

$$\begin{aligned} y &:= F(\mathbf{x}) \\ \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot y_{(1)} \\ y_{(1)} &:= 0 \\ \mathbf{x}_{(2)} &:= \mathbf{x}_{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(2)} + y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}_{(1,2)} \\ y_{(1,2)} &:= \nabla F(\mathbf{x})^T \cdot \mathbf{x}_{(1,2)} \\ y_{(2)} &:= 0 \end{aligned}$$

Second-Order Adjoint (Adjoint over Adjoint)

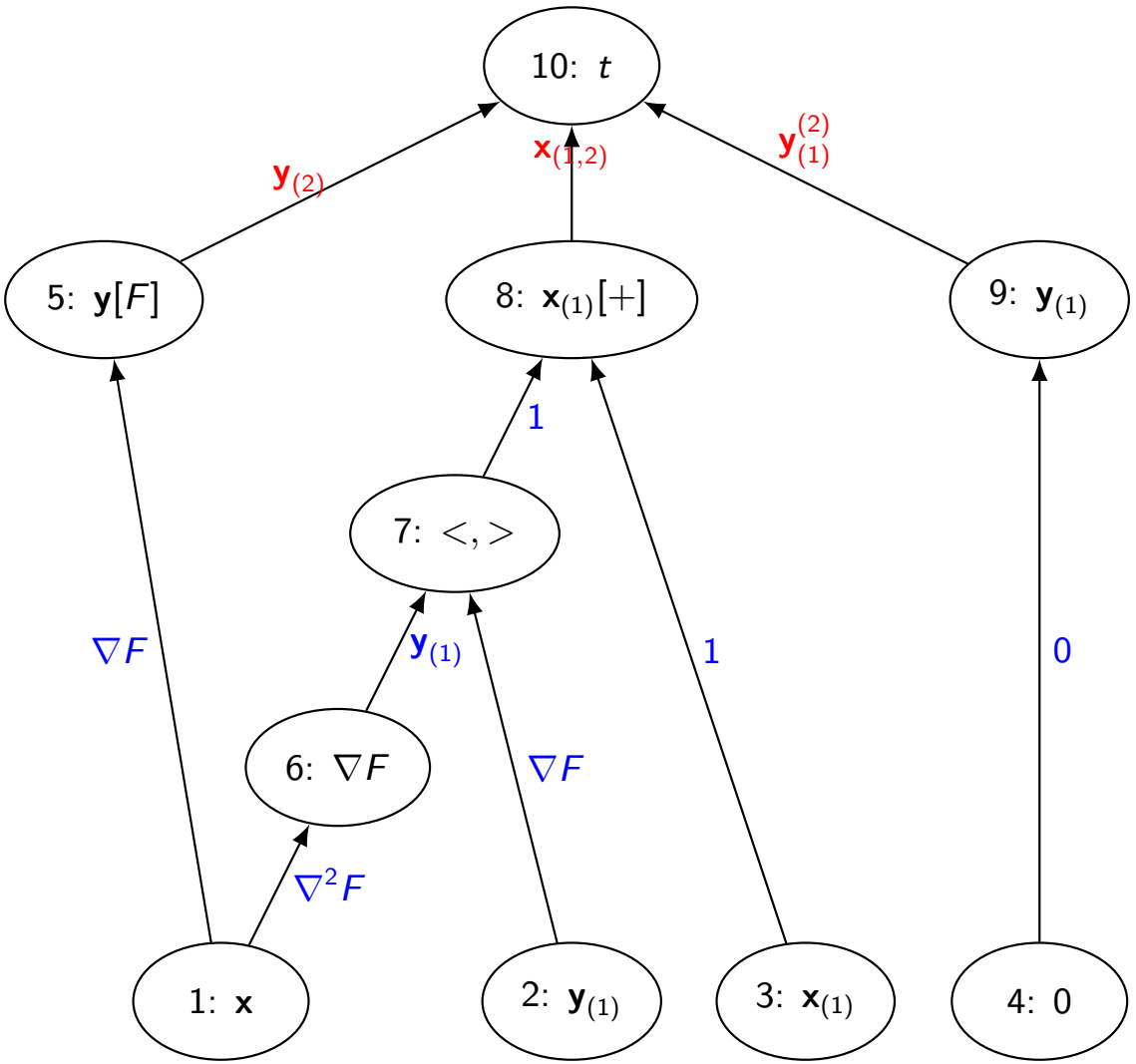
Accumulation of Hessian (Complexity)



$$\mathbf{x}_{(1,2)} \cdot y_{(1)} \cdot \nabla^2 F(\mathbf{x})$$

Second-Order Adjoint (Adjoint over Adjoint)

Adjoint-Augmented Adjoint IDAG



$$
\begin{aligned}
y &:= F(\mathbf{x}) \\
\mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot y_{(1)} \\
y_{(1)} &:= 0 \\
\mathbf{x}_{(2)} &:= \mathbf{x}_{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(2)} \\
&\quad + y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}_{(1,2)} \\
y_{(1,2)} &:= \nabla F(\mathbf{x})^T \cdot \mathbf{x}_{(1,2)} \\
y_{(2)} &:= 0
\end{aligned}
$$

Second-Order Adjoint Code (Adjoint over Adjoint)

Toy Problem with dco/c++


```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  #include "dco.hpp"
6  using namespace dco;
7
8  typedef ga1s<double> AD_BASE_MODE;
9  typedef AD_BASE_MODE::type AD_BASE_TYPE;
10 typedef AD_BASE_MODE::tape_t AD_BASE_TAPE_TYPE;
11 typedef ga1s<AD_BASE_TYPE> AD_MODE;
12 typedef AD_MODE::type AD_TYPE;
13 typedef AD_MODE::tape_t AD_TAPE_TYPE;
14 AD_BASE_TAPE_TYPE*& AD_BASE_TAPE_POINTER=AD_BASE_MODE::global_tape;
15 AD_TAPE_TYPE*& AD_TAPE_POINTER=AD_MODE::global_tape;
16
17 template<typename T>
18 void f(int n, const T *x, T& y ) { y=x[0]/x[1]; }
19
20 void driver(int n, const double* xv, double& yv, double* g, double* h) {
21     AD_TYPE* x=new AD_TYPE[n], *x_in=new AD_TYPE[n];
22     AD_TYPE y;
23     AD_BASE_TAPE_POINTER=AD_BASE_TAPE_TYPE::create();
24     AD_TAPE_POINTER=AD_TAPE_TYPE::create();
25     for (int j=0;j<n;j++) {
26       x[j]=xv[j];
```

```
27        AD_BASE_TAPE_POINTER->register_variable(value(x[j]));
28        AD_TAPE_POINTER->register_variable(x[j]);
29        x_in[j]=x[j];
30      }
31      f(n,x,y);
32      AD_BASE_TYPE &ya=derivative(y);
33      ya=1.0;
34      AD_BASE_TAPE_POINTER->register_variable(ya);
35      AD_TAPE_POINTER->interpret_adjoint();
36      for (int j=0;j<n;j++)
37        g[j]=value(derivative(x_in[j]));
38      for (int i=0,c=0;i<n;i++) {
39        derivative(derivative(x_in[i]))=1;
40        AD_BASE_TAPE_POINTER->interpret_adjoint();
41        for (int j=0;j<=i;j++)
42          h[c++]=derivative(value(x_in[j]));
43        AD_BASE_TAPE_POINTER->zero_adjoints();
44      }
45      yv=passive_value(y);
46      AD_BASE_TAPE_TYPE::remove(AD_BASE_TAPE_POINTER);
47      AD_TAPE_TYPE::remove(AD_TAPE_POINTER);
48      delete [] x; delete [] x_in;
49  }
50
51  int main() {
52    const int n=2;
```

Second-Order Adjoint Code (Adjoint over Adjoint)

Toy Problem with dco/c++ (cont.)

```
53    double x[n]={2.71,3.14},y,g[n],h[n*(n+1)/2];
54    cout.precision(15);
55    driver(n,x,y,g,h);
56    cout << "y=" << y << endl;
57    for (int i=0;i<n;i++)
58      cout << "g[" << i << "]=" << g[i] << endl;
59    cout << "h=[" << endl;
60    for (int i=0,c=0;i<n;i++) {
61      for (int j=0;j<=i;j++)
62        cout << h[c++] <<"\t";
63      cout << endl;
64    }
65    cout << "]" << endl;
66    return 0;
67  }
```

- ~/ga2s_ga1s/ga2s_ga1s_simple.cpp

Live:

- implementation of driver fgh for given

```
template<class T>
    void f(vector<T> &x, T& y) {
    ...
}
```

- build and run

Second-Order Adjoint (Adjoint over Adjoint)

Motivational Example with dco/c++


```
void fgh(int n, const double* x, double& y, double* g, double* h) {
    vector<A2S_A1S_TYPE> t2s_a1s_x(n), t2s_a1s_x_in(n);
    A2S_A1S_TYPE t2s_a1s_y;

    A1S_TAPE_POINTER=A1S_MODE::tape::create();
    A2S_A1S_TAPE_POINTER=A2S_A1S_MODE::tape::create();

    int l=0;

    for (int j=0;j<n;j++) {
      t2s_a1s_x[j]=x[j];
      A1S_TAPE_POINTER->register_variable(dco::get_value(t2s_a1s_x[j]));
      A2S_A1S_TAPE_POINTER->register_variable(t2s_a1s_x[j]);
      t2s_a1s_x_in[j]=t2s_a1s_x[j];
    }

    f(t2s_a1s_x,t2s_a1s_y);

    A1S_TYPE &ya = dco::get_derivative(t2s_a1s_y);
    ya = 1.0;
    A1S_TAPE_POINTER->register_variable(ya);

    A2S_A1S_TAPE_POINTER->interpret_adjoint();

    for (int j=0;j<n;j++)
      g[j]=dco::get_value(dco::get_derivative(t2s_a1s_x_in[j]));
```

```
    for (int i=0;i<n;i++) {
      dco::set_derivative(dco::get_derivative(t2s_a1s_x_in[i]),1.0);
      A1S_TAPE_POINTER->interpret_adjoint();
      for (int j=0;j<=i;j++)
          h[l++] = dco::get_derivative(dco::get_value(t2s_a1s_x_in[j]));
      A1S_TAPE_POINTER->zero_adjoints();
    }

    y = dco::get_passive_value(t2s_a1s_y);

    A1S_MODE::tape::remove(A1S_TAPE_POINTER);
    A2S_A1S_MODE::tape::remove(A2S_A1S_TAPE_POINTER);
}
```

Exercise

Use dco/c++ to generate a second-order adjoint (adjoint over adjoint) version of the product reduction

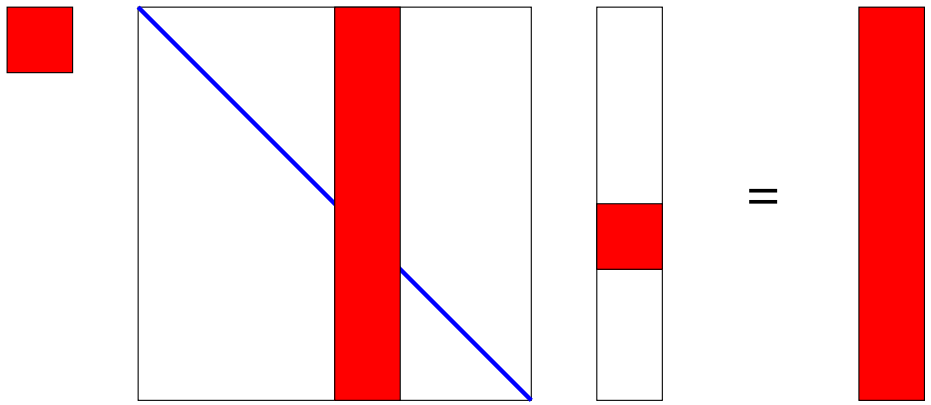
$$y = \prod_{i=0}^{n-1} x_i$$

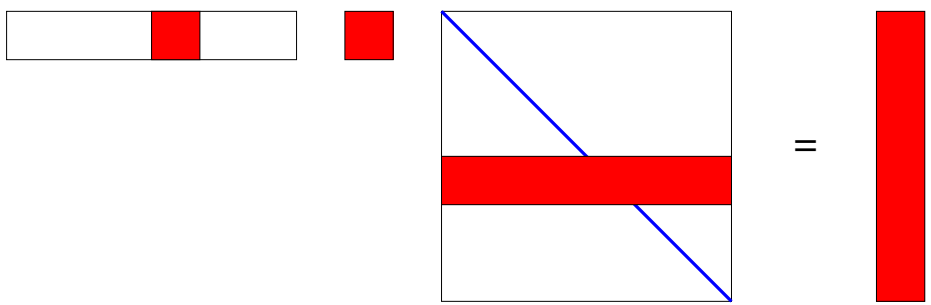
implemented as

```
void f(int n, double *x, double& y) {
  int i=1;
  y=x[0];
  while (i<n) {
    y=y*x[i];
    i=i+1;
  }
}
```

and use it for the computation of the Hessian.

Equivalence of Second-Order Adjoints





Outlook

Wanted: Solution to system of n nonlinear equations $F(\mathbf{x}) = 0$

In:

- $\rightarrow$ implementation of the residual $\mathbf{y}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $\mathbf{y} = F(\mathbf{x})$
- $\rightarrow$ implementation of the Jacobian $A \equiv \nabla F(\mathbf{x})$ of the residual at the current point $\mathbf{x}$:
  $F' : \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$, $A = F'(\mathbf{x})$
- $\rightarrow$ solver for computing the Newton step $d\mathbf{x} \in \mathbb{R}^n$ as the solution of the linear Newton system $A \cdot d\mathbf{x} = -\mathbf{y}$:
  $s : \mathbb{R}^n \times \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^n$, $d\mathbf{x} = s(\mathbf{y}, A)$
- $\rightarrow$ starting point: $\mathbf{x} \in \mathbb{R}^n$
- $\rightarrow$ upper bound on the norm of the residual $\|F(\mathbf{x})\|$ at the approximate solution:
  $\epsilon \in \mathbb{R}$

Out:

- $\leftarrow$ approximate solution of the nonlinear system $F(\mathbf{x}) = 0$: $\mathbf{x} \in \mathbb{R}^n$

Algorithm:

1: $\mathbf{y} = F(\mathbf{x})$
2: **while** $\|\mathbf{y}\| > \epsilon$ **do**
3: $\quad A = F'(\mathbf{x})$
4: $\quad d\mathbf{x} = s(\mathbf{y}, A)$
5: $\quad \mathbf{x} \leftarrow \mathbf{x} + d\mathbf{x}$
6: $\quad \mathbf{y} = F(\mathbf{x})$
7: **end while**

Wanted: Solution to $\mathbf{argmin}_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$

In:

- $\rightarrow$ implementation of the objective $y \in \mathbb{R}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
  $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $y = f(\mathbf{x})$
- $\rightarrow$ implementation of f' for computing the objective $y \equiv f(\mathbf{x})$ and its gradient $\mathbf{g} \equiv \nabla f(\mathbf{x})$ at the current point $\mathbf{x}$:
  $f' : \mathbb{R}^n \rightarrow \mathbb{R} \times \mathbb{R}^n$, $(y, \mathbf{g}) = f'(\mathbf{x})$
- $\rightarrow$ starting point: $\mathbf{x} \in \mathbb{R}^n$
- $\rightarrow$ upper bound on gradient norm $\|\mathbf{g}\|$ at the approximate minimal point: $\epsilon \in \mathbb{R}$

Out:

- $\leftarrow$ approximate minimal value of the objective: $y \in \mathbb{R}$
- $\leftarrow$ approximate minimal point: $\mathbf{x} \in \mathbb{R}^n$

Algorithm:

1: **repeat**
2: $(y, \mathbf{g}) = f'(\mathbf{x})$
3: **if** $\|\mathbf{g}\| > \epsilon$ **then**
4: $\alpha \leftarrow 1$
5: $\tilde{y} \leftarrow y$
6: **while** $\tilde{y} \geq y$ **do**
7: $\tilde{\mathbf{x}} \leftarrow \mathbf{x} - \alpha \cdot \mathbf{g}$
8: $\tilde{y} = f(\tilde{\mathbf{x}})$
9: $\alpha \leftarrow \alpha/2$
10: **end while**
11: $\mathbf{x} \leftarrow \tilde{\mathbf{x}}$
12: **end if**
13: **until** $\|\mathbf{g}\| \leq \epsilon$

Considering

$$\mathbf{argmin}_{\mathbf{x}\in\mathbf{R}^n} \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

n	$f(\mathbf{x} + \mathbf{e}_i \cdot h)$	$f^{(1)}$	$f_{(1)}$
100	13	8	< 1
200	47	28	1
300	104	63	2
400	184	113	2.5
500	284	173	3
1000	1129	689	6

Wanted: Solution to $\mathbf{argmin}_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$

In:

- → implementation of the objective $y \in \mathbb{R}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
  $f : \mathbb{R}^n \to \mathbb{R},\ y = f(\mathbf{x})$
- → implementation of the differentiated objective function f' for computing the objective $y \equiv f(\mathbf{x})$ and its gradient $\mathbf{g} \equiv \nabla f(\mathbf{x})$ at the current point $\mathbf{x}$:
  $f' : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}^n,\ (y, \mathbf{g}) = f'(\mathbf{x})$
- → implementation of the differentiated objective function f'' for computing the objective y, its gradient $\mathbf{g}$, and its Hessian $H \equiv \nabla^2 f(\mathbf{x})$ at the current point $\mathbf{x}$:
  $f'' : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n\times n},\ (y, \mathbf{g}, H) = f''(\mathbf{x})$
- → solver to determine the Newton step $d\mathbf{x} \in \mathbb{R}^n$ as the solution of linear Newton system $H \cdot d\mathbf{x} = -\mathbf{g}$:
  $s : \mathbb{R}^n \times \mathbb{R}^{n\times n} \to \mathbb{R}^n,\ d\mathbf{x} = s(\mathbf{g}, H)$
- → starting point: $\mathbf{x} \in \mathbb{R}^n$
- → upper bound on the gradient norm $\|\mathbf{g}\|$ at the approximate solution: $\epsilon \in \mathbb{R}$

Out:

- ← approximate minimal value: $y \in \mathbb{R}$

$\leftarrow$ approximate minimal point: $\mathbf{x} \in \mathbb{R}^n$

Algorithm:

1: $(y, \mathbf{g}) = f'(\mathbf{x})$
2: **while** $\|\mathbf{g}\| > \epsilon$ **do**
3: $\quad (y, \mathbf{g}, H) = f''(\mathbf{x})$
4: $\quad d\mathbf{x} = s(\mathbf{g}, H)$
5: $\quad \alpha \leftarrow 1$
6: $\quad \tilde{y} \leftarrow y$
7: $\quad \tilde{\mathbf{x}} \leftarrow \mathbf{x}$
8: $\quad$ **while** $\tilde{y} \geq y$ **do**
9: $\quad\quad \tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} - \alpha \cdot d\mathbf{x}$
10: $\quad\quad \tilde{y} = f(\tilde{\mathbf{x}})$
11: $\quad\quad \alpha \leftarrow \alpha/2$
12: $\quad$ **end while**
13: $\quad \mathbf{x} \leftarrow \tilde{\mathbf{x}}$
14: $\quad (y, \mathbf{g}) = f'(\mathbf{x})$
15: **end while**

Considering

$$\mathbf{argmin}_{\mathbf{x}\in\mathrm{R}^n} \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

n	$f(\mathbf{x} + \mathbf{e}_i \cdot h)$	$f^{(1,2)}$	$f^{(2)}_{(1)}$	$f^{(2)}_{(1)} \cdot v$
100	< 1	< 1	< 1	< 1
200	2	1	< 1	< 1
300	7	3	1	< 1
400	17	9	4	< 1
500	36	21	10	< 1
1000	365	231	138	< 1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
10^5	$> 10^4$	$> 10^4$	$> 10^4$	1

Live: Derivative of 10^9 iterations of

$$x = \sin(x \cdot p)$$

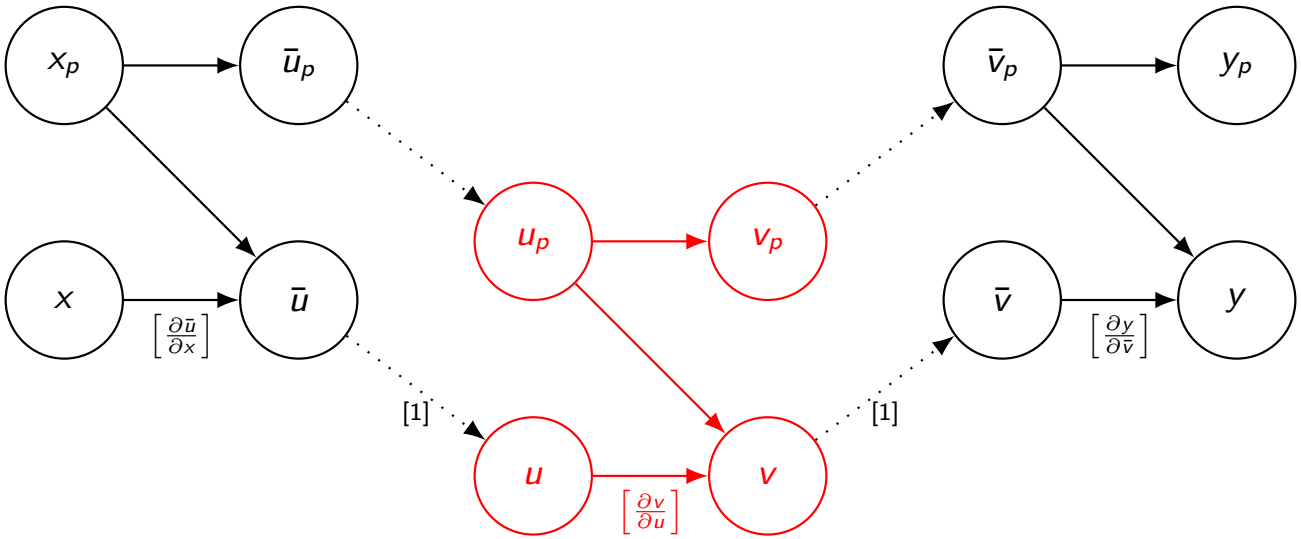
by

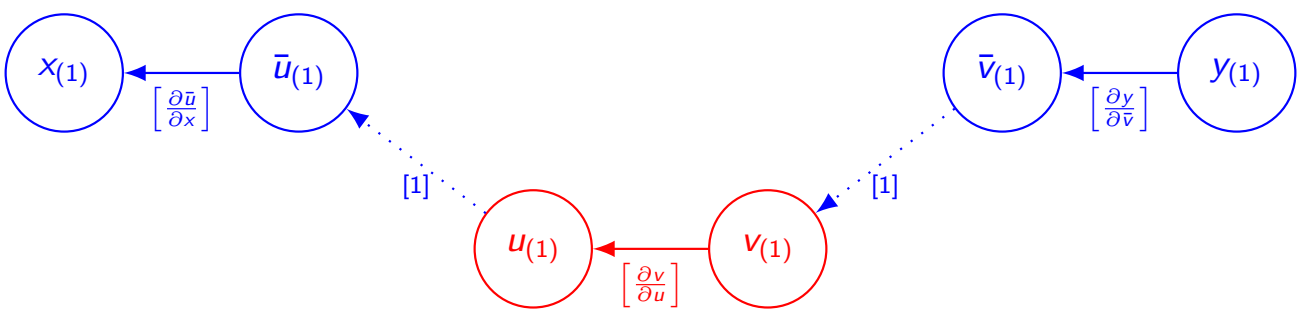
- `~/outlook/checkpointing/ga1s_split.cpp` (TODO)

$\Rightarrow$ `std::bad_alloc`

Outlook: Checkpointing

Mind the Gap



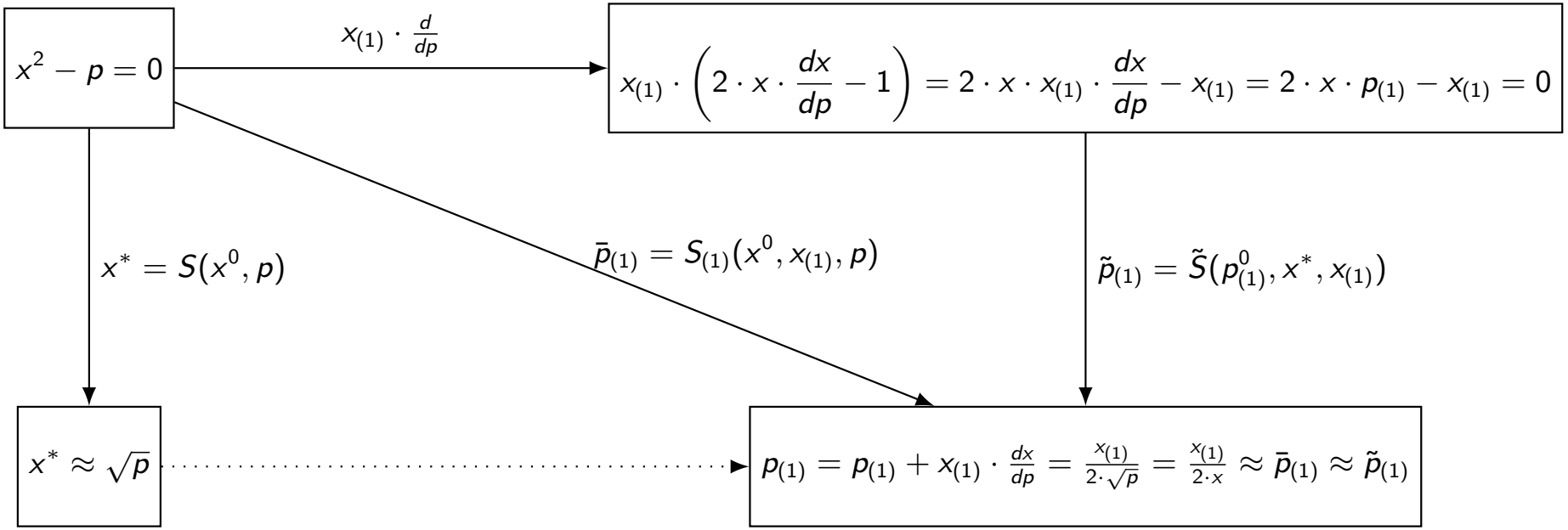


Outlook: User-Defined Tangents and Adjoints

- symbolic tangents and adjoints of numerical methods
- integration of differentiated numerical libraries
- preaccumulation
- hybrid overloading / source transformation
- local finite differencing

Outlook: Symbolic Adjoints of Numerical Methods

Example: Adjoint Root Finder



Conclusion

Conclusion

- AD is conceptually simple.
- Application to large codes requires in-depth understanding of their structure; can be tedious and error-prone.
- AD tool support is highly desirable and use is convenient.
- AD should become part of your software engineering strategy.
- Further (course) work is necessary in order to obtain robust, efficient, and scalable derivative code when using tool support.

What Next?

- Further Modules
  - AD manually – Advanced Topics
  - AD by OL with dco – Advanced Topics
- In-house courses


A. Griewank and A. Walther.
Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.
Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.


U. Naumann.
The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.
Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

Multivariate Vector Function (Example)

Lighthouse Example [1]

Generic Primal $F : \mathbb{R}^4 \to \mathbb{R}^2$


```
template<typename T>
void f(const vector<T>& x, vector<T>& y) {
  T v=tan(x[2]*x[3]); T w=x[1]-v;
  y[0]=x[0]*v/w;
  y[1]=y[0]*x[1];
}
```

```
#include <vector>
using namespace std;

#include "dco.hpp"
using namespace dco;

typedef double AD_BASE_TYPE;
typedef gt1s<AD_BASE_TYPE> AD_MODE;
typedef AD_MODE::type AD_TYPE;

#include "f.hpp"

void driver(
    const vector<double>& xv, const vector<double>& xt,
    vector<double>& yv, vector<double>& yt
) {
  const int n=xv.size(), m=yv.size();
  vector<AD_TYPE> x(n), y(m);
  for (int i=0;i<n;i++) { value(x[i])=xv[i]; derivative(x[i])=xt[i]; }
  f(x,y);
  for (int i=0;i<m;i++) { yv[i]=value(y[i]); yt[i]=derivative(y[i]); }
}
```

The Jacobian of F at point $\mathbf{x} = (1, 1, 1, 1)^T$ is equal to

$$\nabla F(\mathbf{x}) = \begin{pmatrix} -2.79402 & -5.01252 & 11.025 & 11.025 \\ -2.79402 & -7.80654 & 11.025 & 11.025 \end{pmatrix}$$

Consequently, the sum of its columns is computed as the first directional derivative of F with respect to $\mathbf{x}$ in direction $\mathbf{x}^{(1)} = (1, 1, 1, 1)$, that is,

$$\begin{aligned} \mathbf{y}^{(1)} &:= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ &= \begin{pmatrix} -2.79402 & -5.01252 & 11.025 & 11.025 \\ -2.79402 & -7.80654 & 11.025 & 11.025 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 14.2435 \\ 11.4495 \end{pmatrix}. \end{aligned}$$

```
#include <vector>
using namespace std;

#include "dco.hpp"
using namespace dco;

typedef double AD_BASE_TYPE;
typedef ga1s<AD_BASE_TYPE> AD_MODE;
typedef AD_MODE::type AD_TYPE;
typedef AD_MODE::tape_t AD_TAPE_TYPE;

#include "../gt1s/f.hpp"

void driver(
    const vector<double>& xv, vector<double>& xa,
    vector<double>& yv, vector<double>& ya
) {
  AD_MODE::global_tape=AD_TAPE_TYPE::create();
  int n=xv.size(), m=yv.size();
  vector<AD_TYPE> x(n), y(m);
  for (int i=0;i<n;i++) {
    x[i]=xv[i];
    AD_MODE::global_tape->register_variable(x[i]);
  }
  f(x,y);
  for (int i=0;i<m;i++) {
```

```
    AD_MODE::global_tape->register_output_variable(y[i]);
    yv[i]=value(y[i]); derivative(y[i])=ya[i];
  }
  for (int i=0;i<n;i++) derivative(x[i])=xa[i];
  AD_MODE::global_tape->interpret_adjoint();
  for (int i=0;i<n;i++) xa[i]=derivative(x[i]);
  for (int i=0;i<m;i++) ya[i]=derivative(y[i]);
  AD_TAPE_TYPE::remove(AD_MODE::global_tape);
}
```

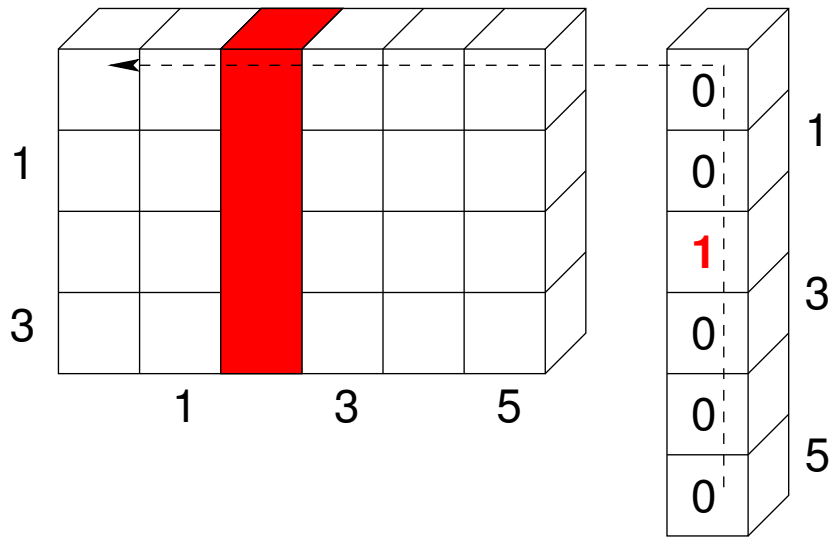
The Jacobian of F at point $\mathbf{x} = (1, 1, 1, 1)^T$ is equal to

$$\nabla F(\mathbf{x}) = \begin{pmatrix} -2.79402 & -5.01252 & 11.025 & 11.025 \\ -2.79402 & -7.80654 & 11.025 & 11.025 \end{pmatrix}.$$

Consequently, the shifted sum of its rows is computed as the first-order adjoint of F with respect to $\mathbf{x}$ in direction $\mathbf{y}_{(1)} = (1, 1)^T$ added to $\mathbf{x}_{(1)} = (1, 1)^T$, that is,

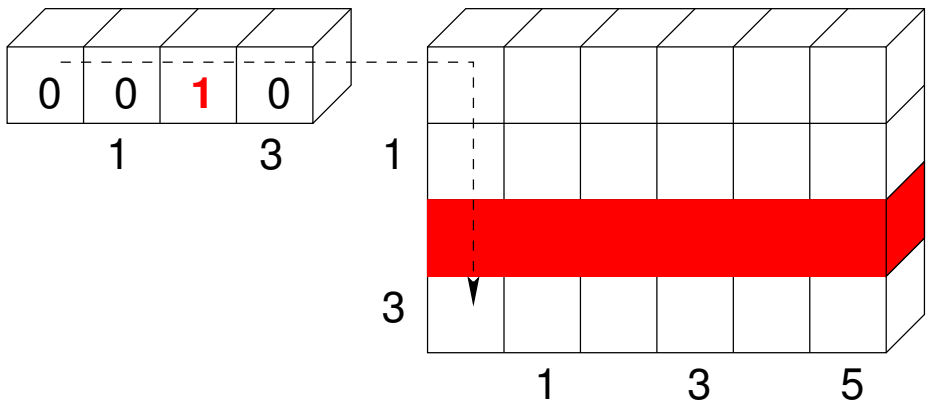
$$\begin{aligned} \mathbf{x}_{(1)} &:= \mathbf{x}_{(1)} + (\nabla F)^T \cdot \mathbf{y}_{(1)} \\ &= \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} -2.79402 & -2.79402 \\ -5.01252 & -7.80654 \\ 11.025 & 11.025 \\ 11.025 & 11.025 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -4.58804 \\ -11.8191 \\ 23.0501 \\ 23.0501 \end{pmatrix}. \end{aligned}$$

Tangent Projection



$$< A, v >\equiv A \cdot v$$

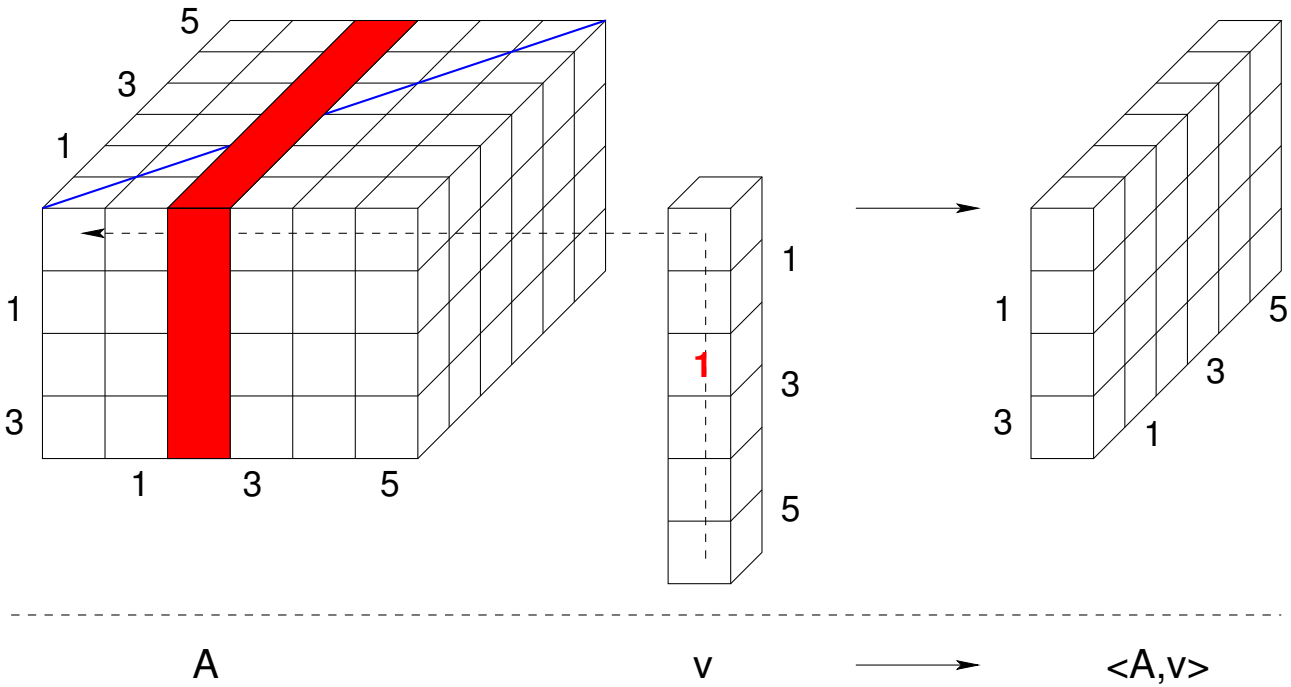
Adjoint Projection



$$< w, A >\equiv A^T \cdot w = (w^T \cdot A)^T$$

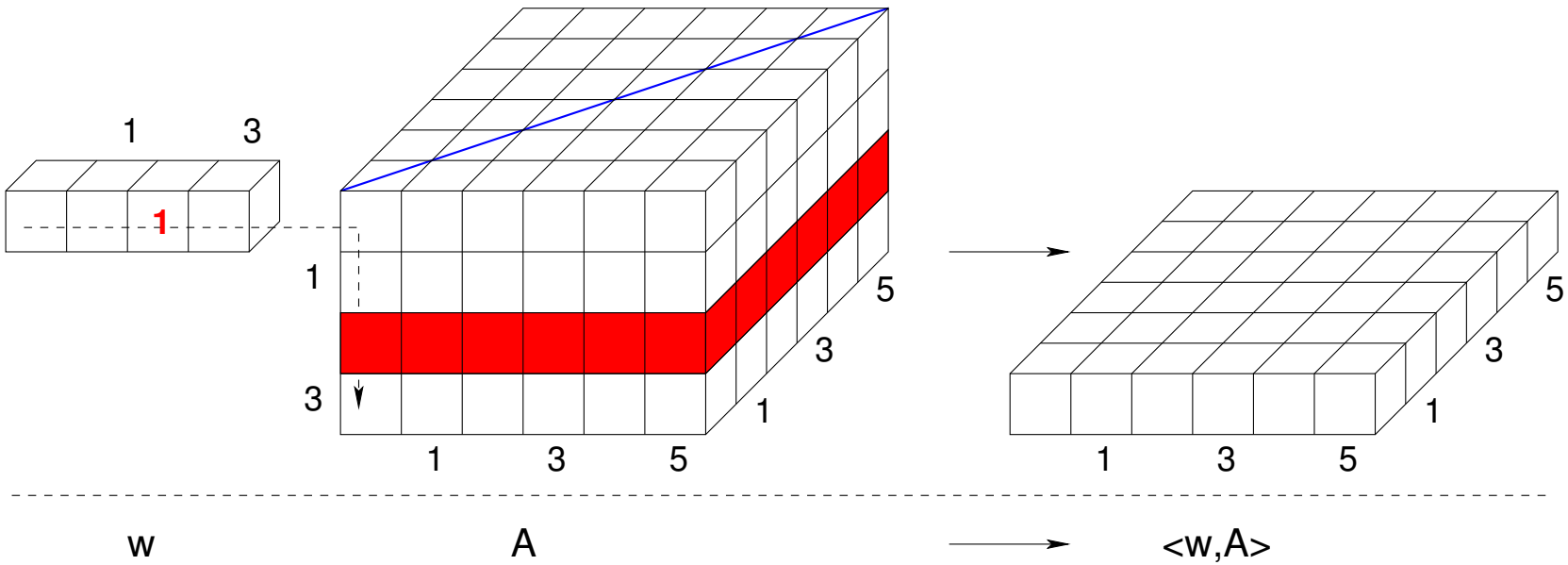
Tensor Arithmetic

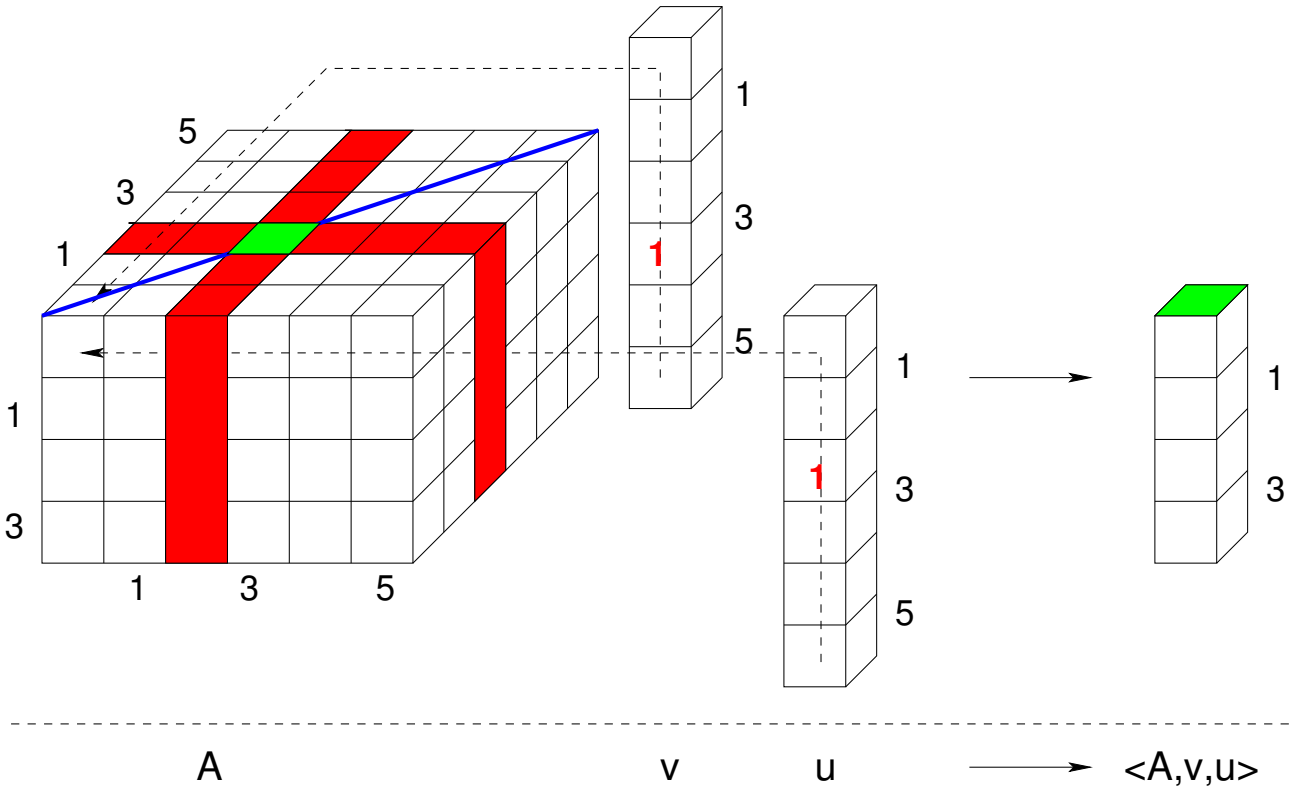
First-Order Tangent Projection of Hessians (e.g. $A \equiv \nabla^2 F$, $F : \mathbb{R}^6 \to \mathbb{R}^4$)



Tensor Arithmetic

First-Order Adjoint Projection of Hessians (e.g. $A \equiv \nabla^2 F$, $F : \mathbb{R}^6 \to \mathbb{R}^4$)

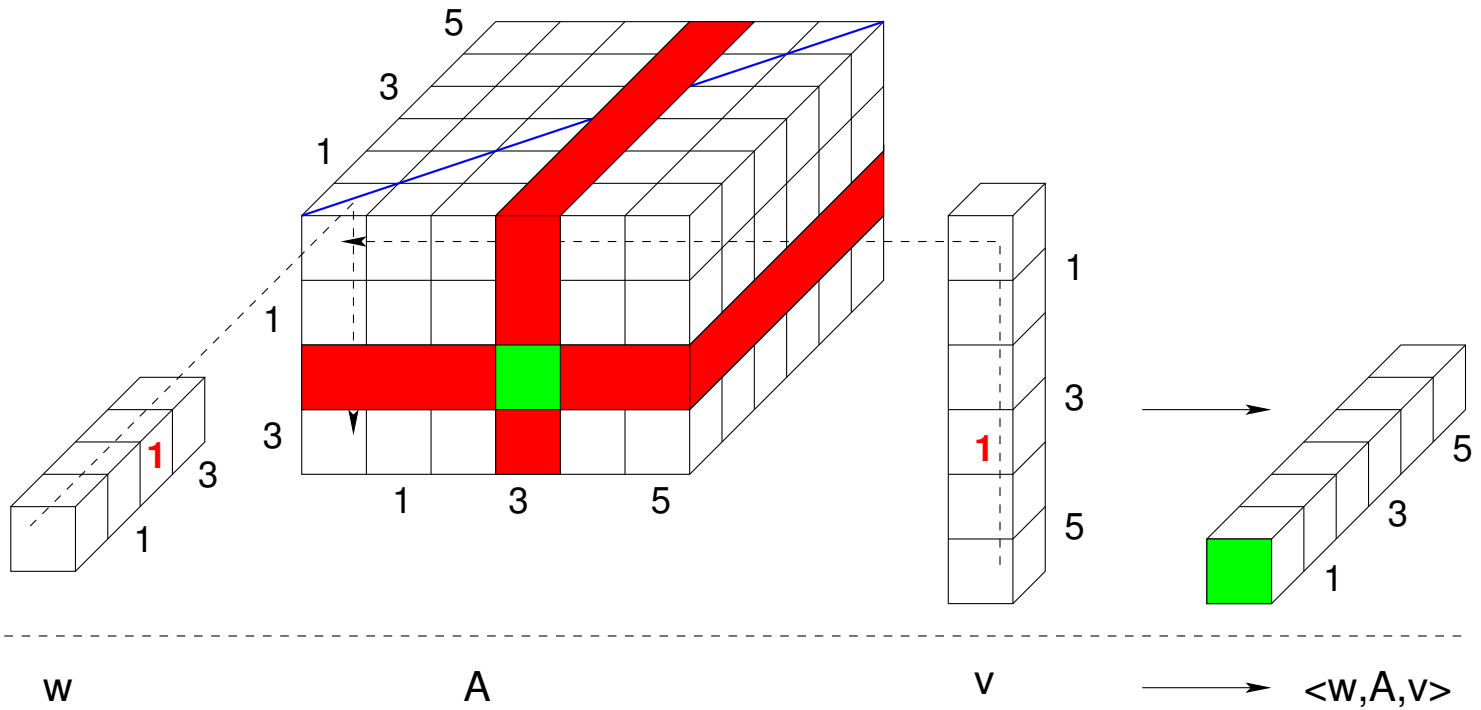




Note:

$$< A, v, u >=<< A, v >, u >=<< A, u >, v >=< A, u, v >$$

due to symmetry.



Note:

$$< w, A, v >=< w <, A, v >>=<< w, A >, v >=< v, < w, A >>=< v, w, A >$$

due to symmetry.

```
#include<iostream>
using namespace std;

#include "dco.hpp"
using namespace dco;
typedef gt1s<double> AD_BASE_MODE;
typedef AD_BASE_MODE::type AD_BASE_TYPE;
typedef gt1s<AD_BASE_TYPE> AD_MODE;
typedef AD_MODE::type AD_TYPE;

#include "../gt1s/f.hpp"

void driver(
    const vector<double>& xv,
    const vector<double>& xt1,
    const vector<double>& xt2,
    const vector<double>& xt1t2,
    vector<double>& yv,
    vector<double>& yt1,
    vector<double>& yt2,
    vector<double>& yt1t2
) {
  const int n=xv.size(), m=yv.size();
  vector<AD_TYPE> x(n),y(m);
  for (int i=0;i<n;i++) {
    value(value(x[i]))=xv[i];
```

```
    derivative(value(x[i]))=xt1[i];
    value(derivative(x[i]))=xt2[i];
    derivative(derivative(x[i]))=xt1t2[i];
  }
  f(x,y);
  for (int i=0;i<m;i++) {
    yv[i]=passive_value(y[i]);
    yt1[i]=derivative(value(y[i]));
    yt2[i]=value(derivative(y[i]));
    yt1t2[i]=derivative(derivative(y[i]));
  }
}
```

The Jacobian of F at point $\mathbf{x} = (1, 1, 1, 1)^T$ is equal to

$$\nabla F(\mathbf{x}) = \begin{pmatrix} -2.79402 & -5.01252 & 11.025 & 11.025 \\ -2.79402 & -7.80654 & 11.025 & 11.025 \end{pmatrix}$$

yielding for $\mathbf{x}^{(1)} = \mathbf{x}^{(2)} = (1, 1, 1, 1)^T$

$$< \nabla F(\mathbf{x}), \mathbf{x}^{(1)} >=< \nabla F(\mathbf{x}), \mathbf{x}^{(2)} >= \begin{pmatrix} 14.2435 \\ 11.4495 \end{pmatrix}.$$

A serialization of the trailing dimension of the corresponding Hessian $\nabla^2 F(\mathbf{x}) \in \mathbb{R}^{2\times 4\times 4}$ is equal to

$$\nabla^2 F(\mathbf{x}) = \begin{pmatrix} \begin{pmatrix} 0 & -5.01252 & 11.025 & 11.025 \\ 0 & -7.80654 & 11.025 & 11.025 \end{pmatrix} \\ \begin{pmatrix} -5.01252 & -17.9851 & 50.5833 & 50.5833 \\ -7.80654 & -28.0102 & 61.6084 & 61.6084 \end{pmatrix} \\ \begin{pmatrix} 11.025 & 50.5833 & -101.167 & -90.1416 \\ 11.025 & 61.6084 & -101.167 & -90.1416 \end{pmatrix} \\ \begin{pmatrix} 11.025 & 50.5833 & -90.1416 & -101.167 \\ 11.025 & 61.6084 & -90.1416 & -101.167 \end{pmatrix} \end{pmatrix}.$$

The sum of the above four (2×4)-matrices can be computed for $\mathbf{x}^{(1)} = (1,1,1,1)^T$ as

$$< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} >= \begin{pmatrix} 17.0376 & 78.169 & -129.7 & -129.7 \\ 14.2435 & 87.4 & -118.675 & -118.675 \end{pmatrix}.$$

The following product with $\mathbf{x}^{(2)} = (1,1,1,1)^T$ yields

$$< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} >\equiv<< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} >, \mathbf{x}^{(2)} >$$

$$= \begin{pmatrix} 17.0376 & 78.169 & -129.7 & -129.7 \\ 14.2435 & 87.4 & -118.675 & -118.675 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -164.193 \\ -135.706 \end{pmatrix}.$$

Consequently,

$$\mathbf{y}^{(1,2)} :=< \nabla F(\mathbf{x}), \mathbf{x}^{(1,2)} > + < \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} >$$

$$= \begin{pmatrix} 14.2435 \\ 11.4495 \end{pmatrix} + \begin{pmatrix} -164.193 \\ -135.706 \end{pmatrix} = \begin{pmatrix} -149.95 \\ -124.257 \end{pmatrix}$$

for $\mathbf{x}^{(1,2)} = (1,1,1,1)^T$.

```
#include<vector>
using namespace std;

#include "dco.hpp"
using namespace dco;

typedef gt1s<double> AD_BASE_MODE;
typedef AD_BASE_MODE::type AD_BASE_TYPE;
typedef ga1s<AD_BASE_TYPE> AD_MODE;
typedef AD_MODE::type AD_TYPE;
typedef AD_MODE::tape_t AD_TAPE_TYPE;

#include "../gt1s/f.hpp"

void driver(
    const vector<double>& xv,
    const vector<double>& xt2,
    vector<double>& xa1,
    vector<double>& xa1t2,
    vector<double>& yv,
    vector<double>& yt2,
    vector<double>& ya1,
    vector<double>& ya1t2
) {
  AD_MODE::global_tape=AD_TAPE_TYPE::create();
  const int n=xv.size(), m=yv.size();
```

```
vector<AD_TYPE> x(n), y(m);
for (int i=0;i<n;i++) {
  x[i]=xv[i];
  AD_MODE::global_tape->register_variable(x[i]);
  derivative(value(x[i]))=xt2[i];
}
f(x,y);
for (int i=0;i<n;i++) {
  value(derivative(x[i]))=xa1[i];
  derivative(derivative(x[i]))=xa1t2[i];
}
for (int i=0;i<m;i++) {
  yv[i]=passive_value(y[i]);
  yt2[i]=derivative(value(y[i]));
  AD_MODE::global_tape->register_output_variable(y[i]);
  value(derivative(y[i]))=ya1[i];
  derivative(derivative(y[i]))=ya1t2[i];
}
AD_MODE::global_tape->interpret_adjoint();
for (int i=0;i<n;i++) {
  xa1t2[i]=derivative(derivative(x[i]));
  xa1[i]=value(derivative(x[i]));
}
for (int i=0;i<m;i++) {
  ya1t2[i]=derivative(derivative(y[i]));
  ya1[i]=value(derivative(y[i]));
```

```
    }
    AD_TAPE_TYPE::remove(AD_MODE::global_tape);
}
```

The Jacobian of F at point $\mathbf{x} = (1,1,1,1)^T$ is equal to

$$\nabla F(\mathbf{x}) = \begin{pmatrix} -2.79402 & -5.01252 & 11.025 & 11.025 \\ -2.79402 & -7.80654 & 11.025 & 11.025 \end{pmatrix}$$

yielding for $\mathbf{x}_{(1)} = \mathbf{x}^{(2)} = (1,1,1,1)^T$ and $\mathbf{y}_{(1)} = (1,1)^T$

$$\mathbf{y}^{(2)} := < \nabla F(\mathbf{x}), \mathbf{x}^{(2)} > = \begin{pmatrix} 14.2435 \\ 11.4495 \end{pmatrix}$$

and

$$\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + < \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) > = \begin{pmatrix} -4.58804 \\ -11.8191 \\ 23.0501 \\ 23.0501 \end{pmatrix}.$$

A serialization of the trailing dimension of the corresponding Hessian $\nabla^2 F(\mathbf{x}) \in \mathbb{R}^{2\times 4\times 4}$ is equal to

$$\nabla^2 F(\mathbf{x}) = \begin{pmatrix} \begin{pmatrix} 0 & -5.01252 & 11.025 & 11.025 \\ 0 & -7.80654 & 11.025 & 11.025 \end{pmatrix} \\ \begin{pmatrix} -5.01252 & -17.9851 & 50.5833 & 50.5833 \\ -7.80654 & -28.0102 & 61.6084 & 61.6084 \end{pmatrix} \\ \begin{pmatrix} 11.025 & 50.5833 & -101.167 & -90.1416 \\ 11.025 & 61.6084 & -101.167 & -90.1416 \end{pmatrix} \\ \begin{pmatrix} 11.025 & 50.5833 & -90.1416 & -101.167 \\ 11.025 & 61.6084 & -90.1416 & -101.167 \end{pmatrix} \end{pmatrix}.$$

The sum of the above four (2×4)-matrices can be computed for $\mathbf{x}^{(2)} = (1, 1, 1, 1)^T$ as

$$< \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} >= \begin{pmatrix} 17.0376 & 78.169 & -129.7 & -129.7 \\ 14.2435 & 87.4 & -118.675 & -118.675 \end{pmatrix}.$$

The following product of its transpose with $\mathbf{y}_{(1)} = (1,1)^T$ yields

$$< \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} > \equiv < \mathbf{y}_{(1)}, < \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} >>$$
$$= \begin{pmatrix} 17.0376 & 14.2435 \\ 78.169 & 87.4 \\ -129.7 & -118.675 \\ -129.7 & -118.675 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 31.2811 \\ 165.569 \\ -248.375 \\ -248.375 \end{pmatrix}.$$

Consequently,

$$\mathbf{x}^{(2)}_{(1)} := \mathbf{x}^{(2)}_{(1)} + < \mathbf{y}^{(2)}_{(1)}, \nabla F(\mathbf{x}) > + < \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} >$$
$$= \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} -5.58804 \\ -12.8191 \\ 22.0501 \\ 22.0501 \end{pmatrix} + \begin{pmatrix} 31.2811 \\ 165.569 \\ -248.375 \\ -248.375 \end{pmatrix} = \begin{pmatrix} 26.6931 \\ 153.75 \\ -225.325 \\ -225.325 \end{pmatrix}$$

for $\mathbf{y}^{(2)}_{(1)} = (1,1,1,1)^T$.