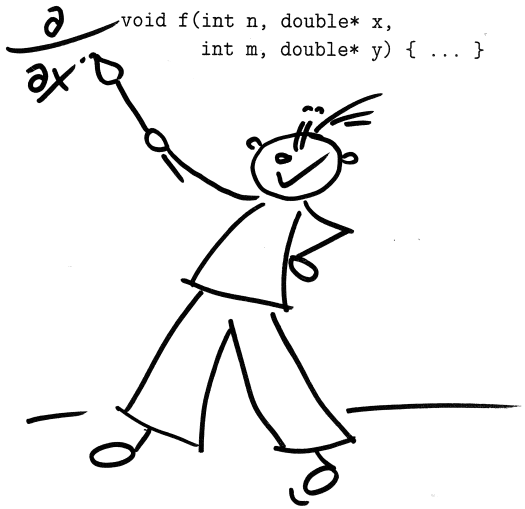


Algorithmic Differentiation

The Art of Differentiating Computer Programms

Uwe Naumann

Software and Tools for Computational Engineering
RWTH Aachen University, Germany

Oxford University, Trinity Term 2017

Motivation

Who knows how to differentiate ...

```
void implicit_euler(const int n, double& x,
      const vector<double>& p, const double& T) {
   double dt=T/n;
   for (int i=0;i<n;i++) {
      double x_prev=x;
      double f=-dt*p[i]*sin(x*i*dt);
      while (abs(f)>1e-15) {
         x=x-f/(1-dt*p[i]*i*dt*cos(x*i*dt));
         f=x-x_prev-dt*p[i]*sin(x*i*dt);
      }
   }
}
```

???

... you will!

Aim of the Course

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.

Assuming differentiability of F and of its implementation[1] we aim to study methods for transforming the given computer program into one for computing first or higher derivatives efficiently.

[1] counter example: if(x) y=sin(x); else y=0;

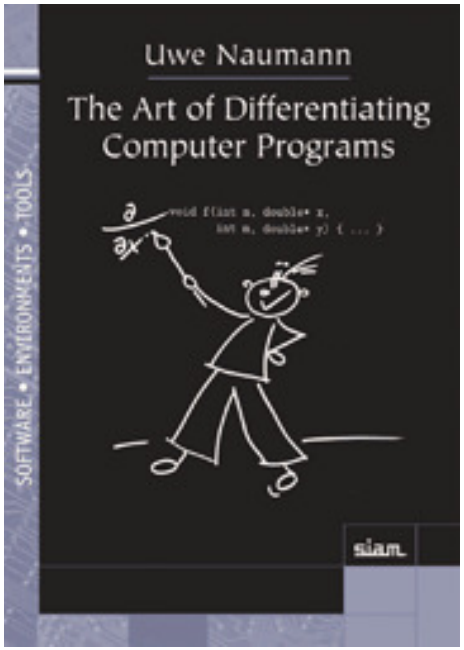
Outline

Logistics

- intro: `https://www.stce.rwth-aachen.de/people/uwe-naumann`
- website: `http://www.stce.rwth-aachen.de/teaching/lectures/oxford2017/`
- registration: email to `naumann@stce.rwth-aachen.de`
- prerequisites: basic numerics and programming
- material: slides, code (C++), own notes
- software: dco/c++ from NAG (`http://nag.co.uk/content/downloads-dco-c-versions`)
  - download and unpack
  - request 3-month trial license key using webform (use Oxford Uni email and reference "OU/AD/61914")
- practice: weekly assignments
- project: implement an adjoint simulation code

Course on Algorithmic Differentiation (AD)

based on ...



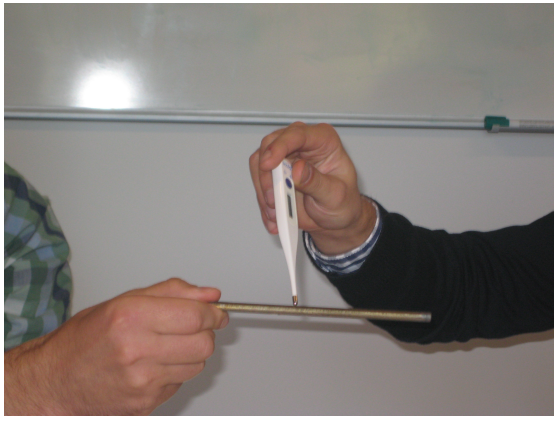
U.N.: The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation. Number 24 in Software , Environments, and Tools, SIAM, 2012.

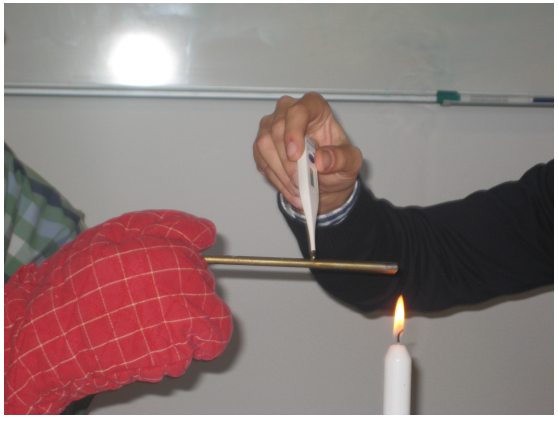
`https://www.stce.rwth-aachen.de/research/the-art`

Outline

Motivation: Cheap Gradients

E.g, 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

$$\frac{dT}{dt} = c(x) \cdot \frac{d^2T}{dx^2}$$

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 $\frac{dr}{dT}$ and factorization).

Motivation: Cheap Gradients

E.g, Diffusion

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

$$\frac{df}{dc} = \frac{df}{dT(1)} \cdot \frac{dT(1)}{dc} \in \mathbb{R}^n \quad (\rightarrow \text{scalar adjoint integration at } O(1))$$

and possibly

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

...
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

Motivation: Cheap Gradients

e.g, Diffusion

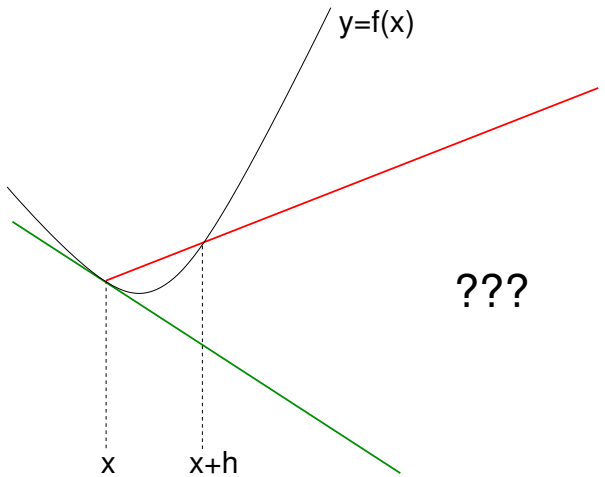
Naive application of Algorithmic Differentiation (AD) tool `dco/c++` yields:

$\frac{df}{dc}$ (n=300, m=50)

	central FD	adjoint AD
run time (s)	15.2	0.7

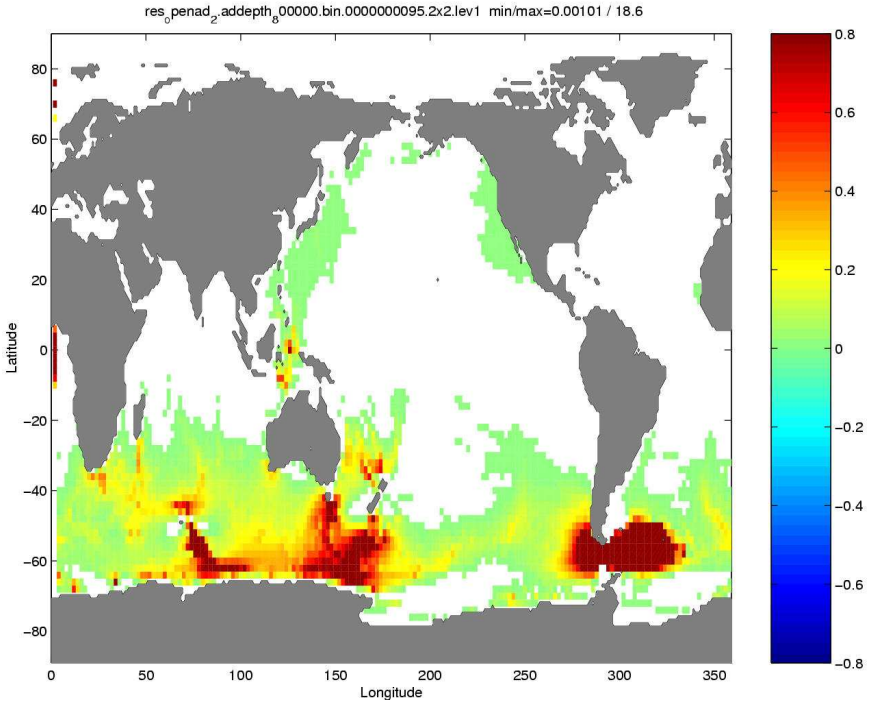
$\frac{d^2 f}{dc^2}$ (n=100, m=50)

	central FD	adjoint AD
run time (s)	63.6	3.9



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

Nice to have?



MITgcm, (EAPS, MIT)

in collaboration with ANL, MIT, Rice, UColorado

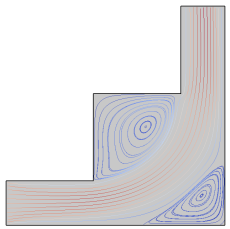
J. Utke, U.N. et al: *OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes* . ACM TOMS 34(4), 2008.

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 :-)

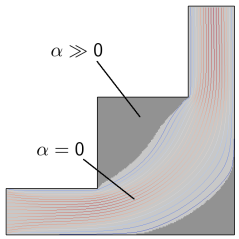
Adjoints Are Everywhere ...

Topology Optimization with Adjoint OpenFOAM [AboutFlow]

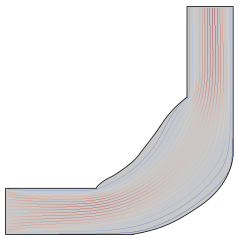
design space



optimization



result



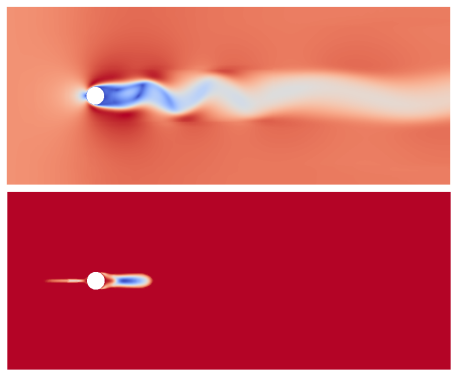
primal (penalized Navier-Stokes [Othmer, 2008])

$$(\mathbf{v} \cdot \nabla)\,\mathbf{v} = \nu \nabla^2 \mathbf{v} - \nabla p - \alpha \mathbf{v}, \quad \alpha > 0$$

objective (total pressure loss between in- and outlet)

$$J = \int_\Gamma p + \frac{1}{2} v_n^2 \, \mathrm{d}\Gamma$$

solver (gradient descent) $\alpha^{n+1} = \alpha^n - \lambda \cdot \frac{dJ^n}{d\alpha^n}$
requires $\frac{dJ^n}{d\alpha^n}$ computed by dco/c++ / AMPI



aiming for unsteady, coupled (e.g, fluid/heat, fluid/structure)

[10]: *MPI-Parallel Discrete Adjoint OpenFOAM*, ICCS Conf., 2015.

Adjoints Are Everywhere ...

DAEs with Optimality Conditions (DAEO) [FZJ-IBG, AVT]

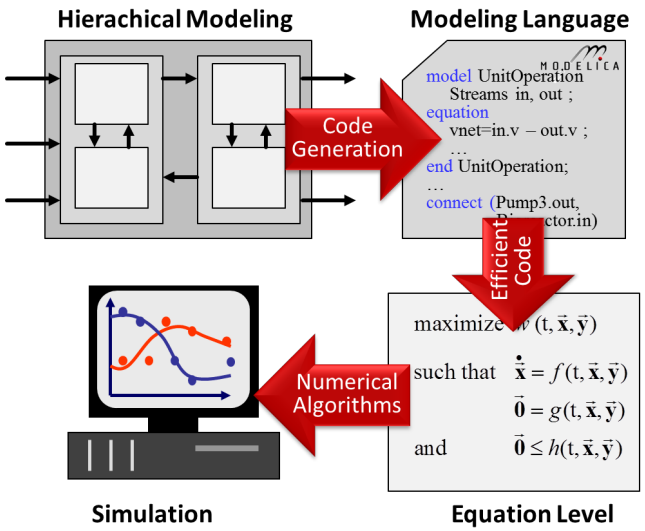
The Jülich-Aachen Dynamic Optimization Environment (JADE) targets DAEO

$$
\begin{aligned}
\dot{y}_d(t) &= f(y_d(t), y_a(t), t, p), \text{ for given } y_d(0) \\
y_a(t) &= \text{argmin}_{\hat{y}_a \in \mathbf{R}^{n_a}} h(y_d(t), \hat{y}_a, t, p) \\
\text{s.t.} \quad & 0 = g_i(y_d(t), \hat{y}_a, t, p), \ i = 1, \ldots, n_e \\
& 0 \leq g_j(y_d(t), \hat{y}_a, t, p), \ j = n_e + 1, \ldots, n_g.
\end{aligned}
$$

Embedding KKT condition

$$
\begin{bmatrix} \nabla_{\hat{y}_a} h - \nabla^T_{\hat{y}_a} g \cdot \lambda \\ g \end{bmatrix} = 0
$$

for Langrangian $L = h - < \lambda, g >$ yields need for $\nabla_{\hat{y}_a} h$, $\nabla^2_{\hat{y}_a} h \cdot u$, $\nabla_{\hat{y}_a} g \cdot v$, $\nabla^T_{\hat{y}_a} g \cdot w$, and $< w, \nabla^2_{\hat{y}_a} g, v >$ generated by dcc / dco/c++ .

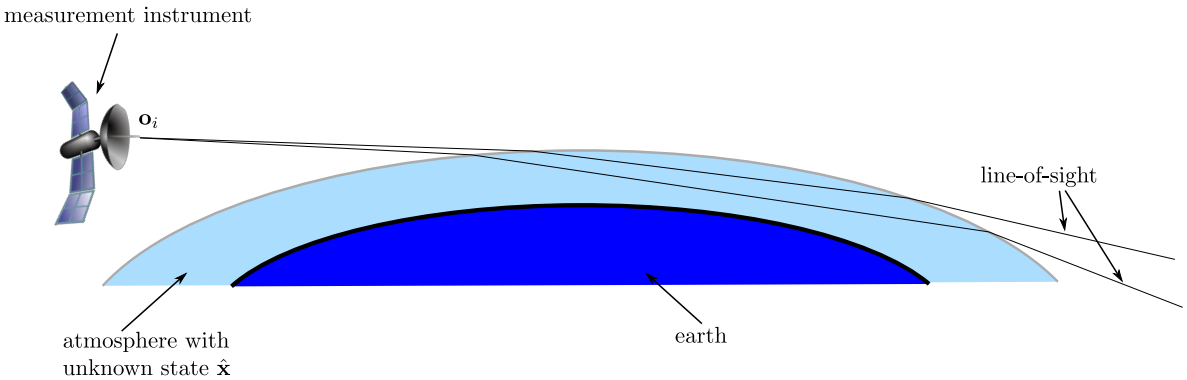


[3]: *First- and Second-Order Parameter Sensitivities of a Metabolically and Isotopically Non-Stationary Biochemical Network Model*, Modelica Conf., 2012.

Adjoints Are Everywhere ...

Parameter Estimation with JURASSIC2 [FZJ-IEK]

recovery of atmospheric state in upper troposphere / lower stratosphere for given radiance measurements along line-of-sight



primal model $\mathbf{y}_i = g(\mathbf{x}, \lambda_i)$ with simulated radiances $\mathbf{y} \in \mathbb{R}^m$, atmospheric parameters $\mathbf{x} \in \mathbb{R}^n$, and line-of-sight elevations $\lambda \in \mathbb{R}^m$

residual $\mathbb{R}^m \ni F = (\mathbf{o}_i - g(\mathbf{x}, \lambda_i))_{i=1\ldots m}$ for measured radiances $\mathbf{o} \in \mathbb{R}^m$

objective $G(\mathbf{x}, \lambda) = F^T S_\epsilon^{-1} F + (\mathbf{x} - \mathbf{x}_a)^T S_a^{-1} (\mathbf{x} - \mathbf{x}_a)$ with measurement error correlation matrix $S_\epsilon \in \mathbb{R}^{m \times m}$, typical atmospheric state values from historical data $\mathbf{x}_a \in \mathbb{R}^n$, and (Tikhonov) regularization matrix $S_a \in \mathbb{R}^{n \times n}$

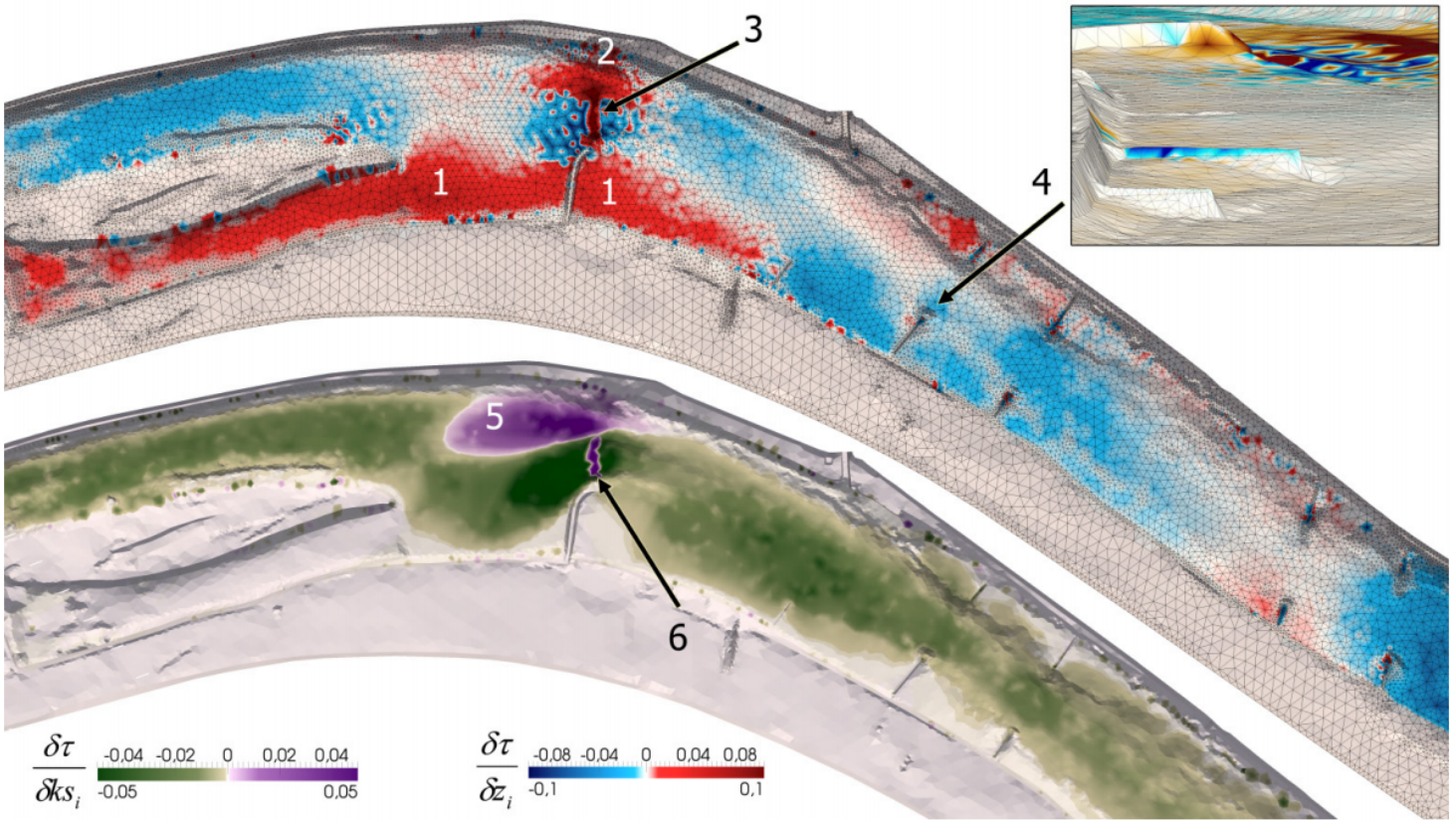
Gauss-Newton solver requires $\nabla_{\mathbf{x}} F$ computed by dco/c++

[11]: *A 3-D tomographic retrieval approach with advection compensation for the air-borne limb-imager GLORIA*, Atmos. Meas. Tech., 2011.

[4]: *A Case Study in Adjoint Sensitivity Analysis of Parameter Calibration*, Submitted, 2016.

Adjoints Are Everywhere ...

Waterways Engineering [BAW, EDF, HRW]



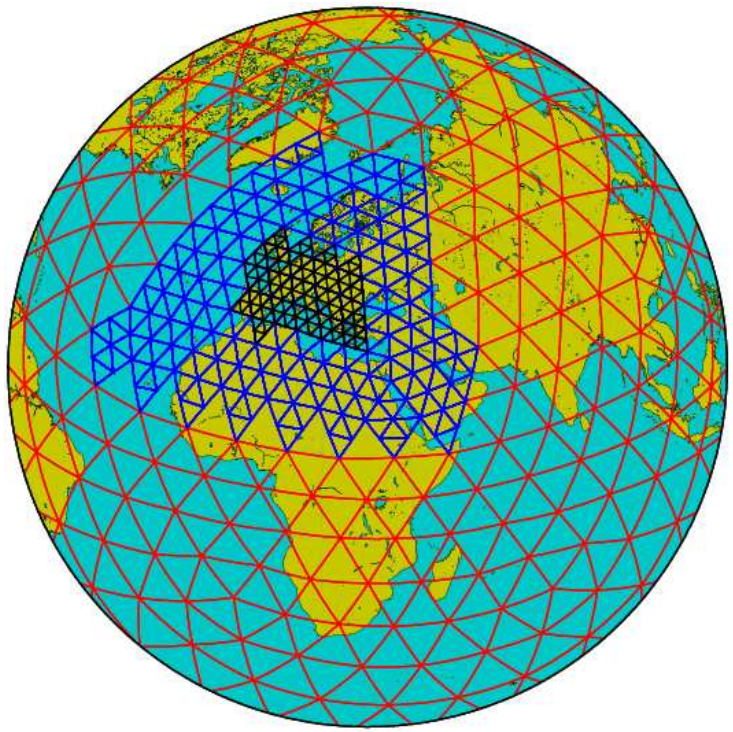
Sensitivity of local shear stress with respect to geometry (top) and roughness of sediment (bottom) generated by dco/fortran.

[5]: *Reverse engineering of initial and boundary conditions with Telemac and algorithmic differentiation*, Wasserwirtschaft, 2013

Adjoints Are Everywhere ...

Physical Oceanography [MPI-M]

Adjoint version of ICON (Icosahedral Non-hydrostatic General Circulation Model generated by dco/fortran for use in 4DVAR data assimilation and error estimation.



[13]: *Estimation of Data Assimilation Error: A Shallow-Water Model Study*, Monthly Weather Review, 2014

Adjoints Are Everywhere ...

Option Pricing / Risk Management [NAG]

Let's 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 is usually evaluated using Monte Carlo simulation.

Scenario

- 10^4 paths
- 360 Euler steps
- 62 uncertain parameters
- pricer takes 1s

Greeks

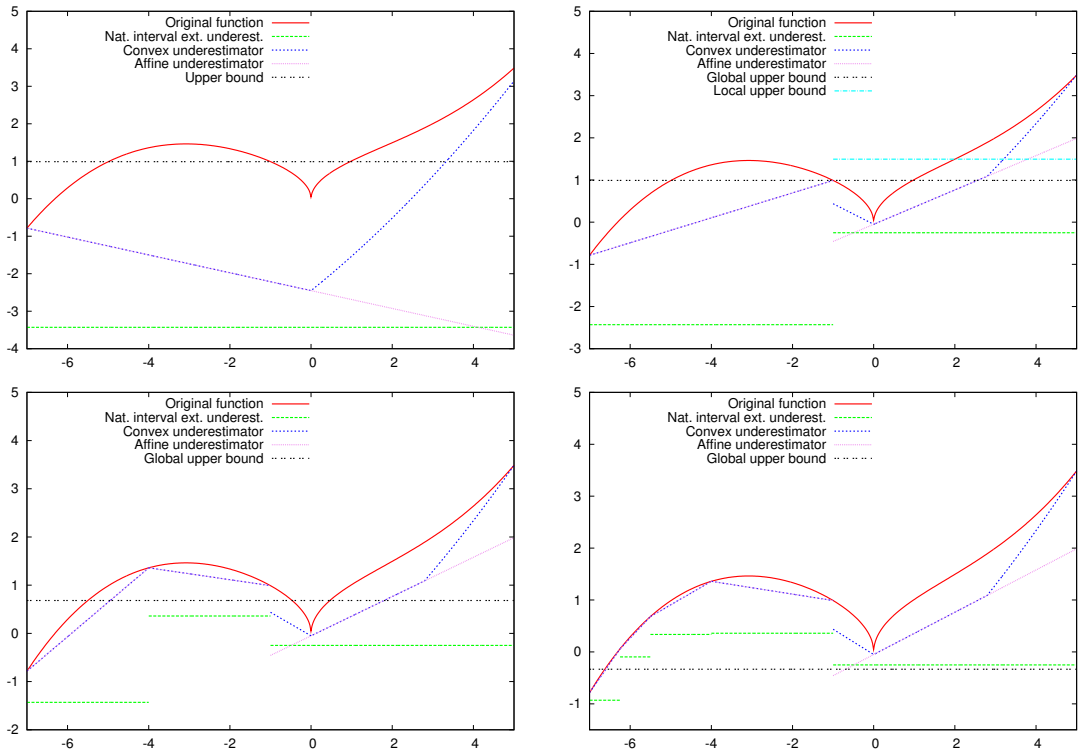
- first order (dco/c++)
  - finite diffs: 81s
  - adjoint: 5s
- second order (dco/c++)
  - finite diffs: 480s
  - adjoint: 53s

[8]: *Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance*, NAG, 2014.

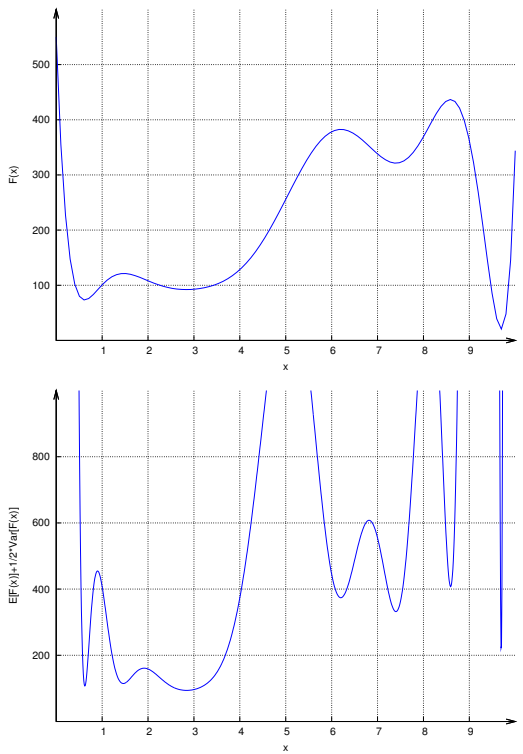
Adjoints Are Everywhere ...

Robust Non-Convex Optimization [AVT, Imperial College]

Branch and (Improved) Bounds through Subgradients of McCormick Relaxations



Robust Objective through Variance Penalty



[1]: *Adjoint Mode Computations of Subgradients for McCormick Relaxations*, AD Conf., LNCSE, 2012.

M. Beckers: *Toward Global Robust Optimization*, Ph.D. Thesis, RWTH Aachen, 2014.

Adjoints Are Everywhere ...

Adjoint Numerical Methods / Libraries [NAG]

For example, nag_zero_cont_func_brent locates a simple zero x of a continuous function f in a given interval [a,b] using Brent's method. The adjoint version enables computation of sensitivities of the solution x wrt. all relevant input parameters (potentially passed through comm).

```
void
nag_zero_cont_func_brent_dco_a1s (
  dco::a1s::type a, // in: lower bound
  dco::a1s::type b, // in: upper bound
  dco::a1s::type eps, // in: termination tolerance on x
  dco::a1s::type eta, // in: acceptance tolerance for vanishing f(x)
  dco::a1s::type (*f)(dco::a1s::type x, Nag_Comm_dco_a1s *comm), // in: f
  dco::a1s::type *x, // out: x
  Nag_Comm_dco_a1s *comm, // inout: parameters
  NagError *fail // out: error code
);
```

We develop first- and higher order tangent and adjoint versions of a growing number of numerical methods using dco/c++, dco/fortran, and hand-coding.

Adjoints Are Everywhere ...

Significance-Based Approximate Computing [SCoRPiO]

For illustration consider

```
y=log(exp(sin(x1))*(x1*x2))/10+x2/100;
```

Interval evaluation for $x1, x2 \in [1, 36000]$ yields

```
v5=log(exp(sin(x1))*(x1*x2)) -> [-1,22]
```

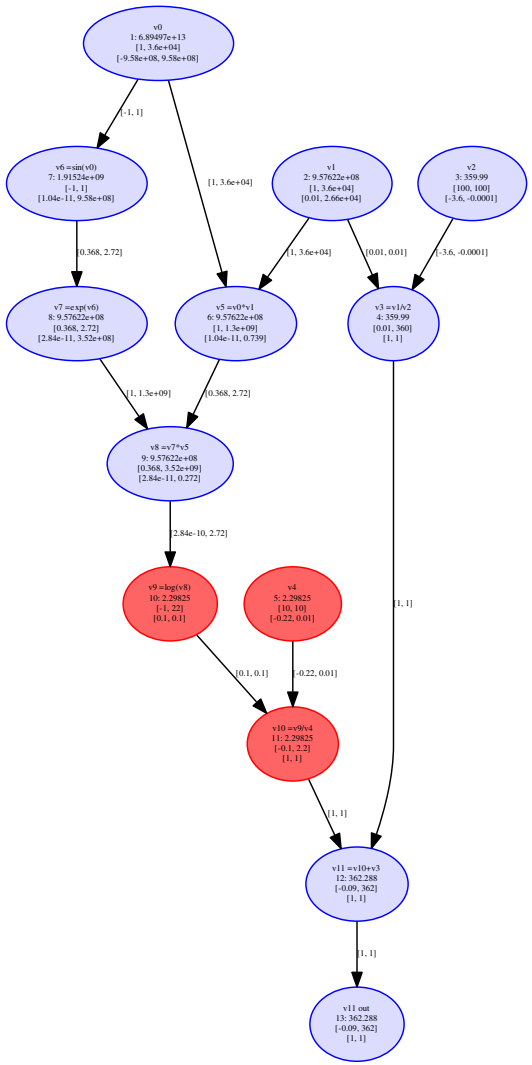
Significance criterion, e.g,

$$w([v5] \cdot \nabla_{[v5]}[y]) \leq 2.5$$

selects v5=10.5; and hence y=1.05+x2/100;

Significance analysis requires gradient of objective y with respect to all intermediates computed by dco/scorpio ($\rightarrow$ scenarios through interval splitting and exploratory spawning).

[12]: *Towards Automatic Significance Analysis for Approximate Computing*, IEEE/ACM CGO, 2016.



- continuity
- differentiability
- gradient, Jacobian, Hessian, higher derivative tensors
- Taylor expansion
- chain rule

Let $y = f(x) : D_f \subseteq \mathbb{R} \to I_f \subseteq \mathbb{R}$ be defined over D_f and let

$$y = f(x) = g(h(x)) = g(z)$$

be such that both g and h are continuously differentiable over their respective domains $D_g = I_h$ and $D_h = D_f$. Then f is continuously differentiable over D_f and

$$\frac{df}{dx}(x^*) = \frac{dg}{dx}(z^*) = \frac{dg}{dz}(z^*) \cdot \frac{dh}{dx}(x^*)$$

for all $x^* \in D_f$ and $z^* = h(x^*)$.

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

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

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

$$\frac{dF}{d\mathbf{x}}(\mathbf{x}^*) = \frac{dG}{d\mathbf{z}}(\mathbf{z}^*) \cdot \frac{dH}{d\mathbf{x}}(\mathbf{x}^*)$$

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

Proof follows immediately from product of the two Jacobians.

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

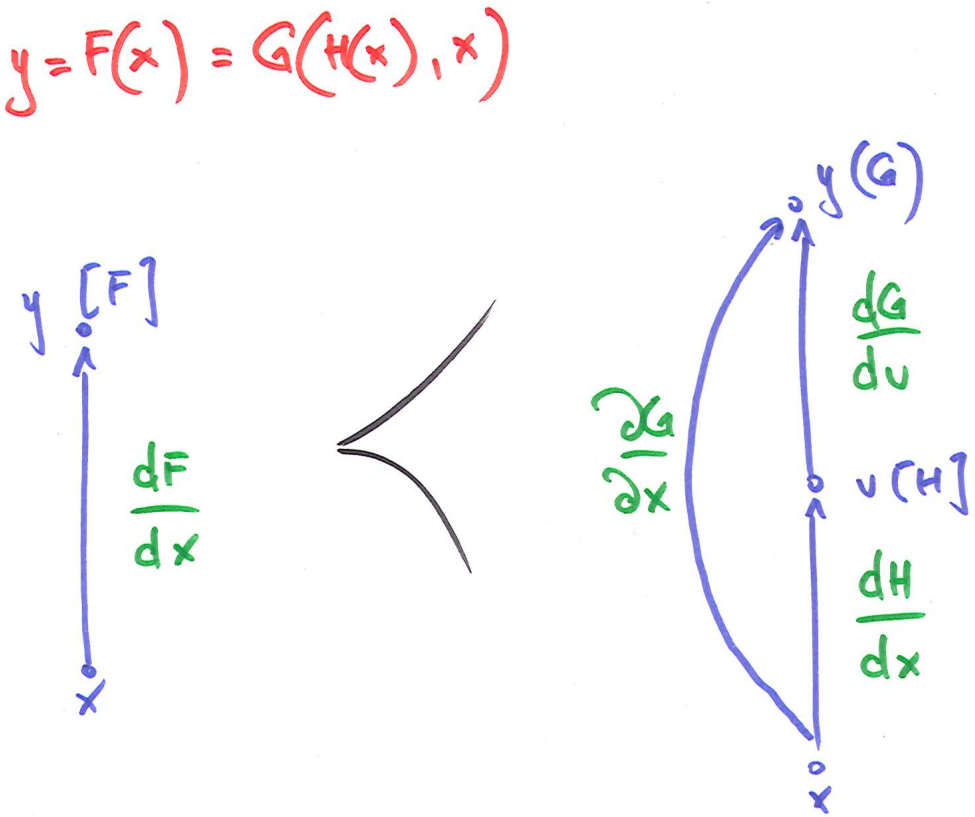
$$\mathbf{y} = F(\mathbf{x}) = G(H(\mathbf{x}), \mathbf{x}) = G(\mathbf{z}, \mathbf{x})$$

be such that both $G : D_G \subseteq \mathbb{R}^p \times \mathbb{R}^n \to I_G \subseteq \mathbb{R}^m$ and $H : D_H \subseteq \mathbb{R}^n \to I_F \subseteq \mathbb{R}^p$ 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

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

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

Notation: $\frac{\partial G}{\partial \mathbf{x}}$ incomplete derivative; $\frac{dG}{d\mathbf{x}}$ [complete] derivative


y = F(x) = G(H(x), x)
y [F]
dF/dx
x
y (G)
dG/dv
∂G/∂x
v [H]
dH/dx
x

Let $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^{n+p+m}$ such that $\boldsymbol{u} = (\mathbf{x}\ 0\ 0)^T$, $\boldsymbol{v} = \tilde{H}(\boldsymbol{u}) = (\mathbf{x}\ H(\mathbf{x})\ 0)^T$, and $\boldsymbol{w} = \tilde{G}(\boldsymbol{v}) = (\mathbf{x}\ H(\mathbf{x})\ G(H(\mathbf{x}), \mathbf{x}))^T$.

By the standard chain rule

$$\frac{d\tilde{F}}{d\boldsymbol{u}} = \frac{d\tilde{G}}{d\boldsymbol{v}} \cdot \frac{d\tilde{H}}{d\boldsymbol{u}} = \begin{pmatrix} I_n & 0 & 0 \\ 0 & I_k & 0 \\ \frac{\partial G}{\partial \mathbf{x}} & \frac{dG}{d\mathbf{z}} & 0 \end{pmatrix} \cdot \begin{pmatrix} I_n & 0 & 0 \\ \frac{dH}{d\mathbf{x}} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} I_n & 0 & 0 \\ \frac{dH}{d\mathbf{x}} & 0 & 0 \\ \frac{\partial G}{\partial \mathbf{x}} + \frac{dG}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}} & 0 & 0 \end{pmatrix}.$$

Hence,

$$\frac{dF}{d\mathbf{x}} = \frac{d\mathbf{y}}{d\boldsymbol{w}} \cdot \frac{d\boldsymbol{w}}{d\boldsymbol{u}} \cdot \frac{d\boldsymbol{u}}{d\mathbf{x}} = Q_m \cdot \frac{d\tilde{F}}{d\boldsymbol{u}} \cdot P_n^T = \frac{\partial G}{\partial \mathbf{x}} + \frac{dG}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}}$$

where $P_n = (I_n\ 0) \in \mathbb{R}^{n \times (n+p+m)}$, and $Q_m = (0\ I_m) \in \mathbb{R}^{m \times (n+p+m)}$ with I_k denoting the identity matrix in $\mathbb{R}^{k \times k}$, $k \in \{n, m\}$, and using appropriate numbers of zero padding columns. ■

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

$$\mathbf{y} = F(\mathbf{x}) = G(H_1(\mathbf{x}), \ldots, H_k(\mathbf{x}))$$

be such that G and all H_i, $i = 1, \ldots, k$ are continuously differentiable over their respective domains.

Then F is continuously differentiable over D_F and its [complete] derivative (Jacobian) $\frac{dF}{d\mathbf{x}}$ is uniquely defined. Moreover, there are 2^k incomplete derivatives $\frac{\partial F}{\partial \mathbf{x}}$ due to considering subsets of the H_i as constant (or passive).

For notational simplicity we avoid precise annotation of incomplete derivatives. The incomplete derivatives referred to will be well defined by the context.

Let

$$y = f(x) = g(h_1(x), h_2(x)) = \sin(x) \cdot \cos(x).$$

By the chain rule the complete derivative becomes

$$\frac{df}{dx} = \cos(x)^2 - \sin(x)^2.$$

The following $2^2 = 4$ incomplete derivatives exist:

$$\frac{\partial f}{\partial x} \in \left\{0, \cos(x)^2, -\sin(x)^2, \cos(x)^2 - \sin(x)^2\right\},$$

where, admittedly, the vanishing derivative due to assuming no dependence on x could be considered as obsolete.

For $F : \mathbb{R}^n \to \mathbb{R}^m : \mathbf{y} = F(\mathbf{x})$ we use the following equivalent notations for total derivatives

$$F'(\mathbf{x}) \equiv \frac{dF(\mathbf{x})}{d\mathbf{x}} \equiv \nabla F(\mathbf{x})$$
$$F''(\mathbf{x}) \equiv \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \equiv \nabla^2 F(\mathbf{x})$$
$$F'''(\mathbf{x}) \equiv \dots$$

and partial derivatives

$$F_{x_i}(\mathbf{x}) \equiv \frac{dF(\mathbf{x})}{dx_i} \equiv \nabla_{x_i} F(\mathbf{x})$$
$$F_{x_i,x_j}(\mathbf{x}) \equiv \frac{d^2F(\mathbf{x})}{dx_i dx_j} \left(\overset{i=j}{\equiv} \frac{d^2F(\mathbf{x})}{dx_i^2} \right) \equiv \nabla_{x_i,x_j} F(\mathbf{x})$$
$$F_{x_i,x_j,x_k}(\mathbf{x}) \equiv \dots$$

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[2] implying the existence of the Jacobian (Hessian etc.)

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

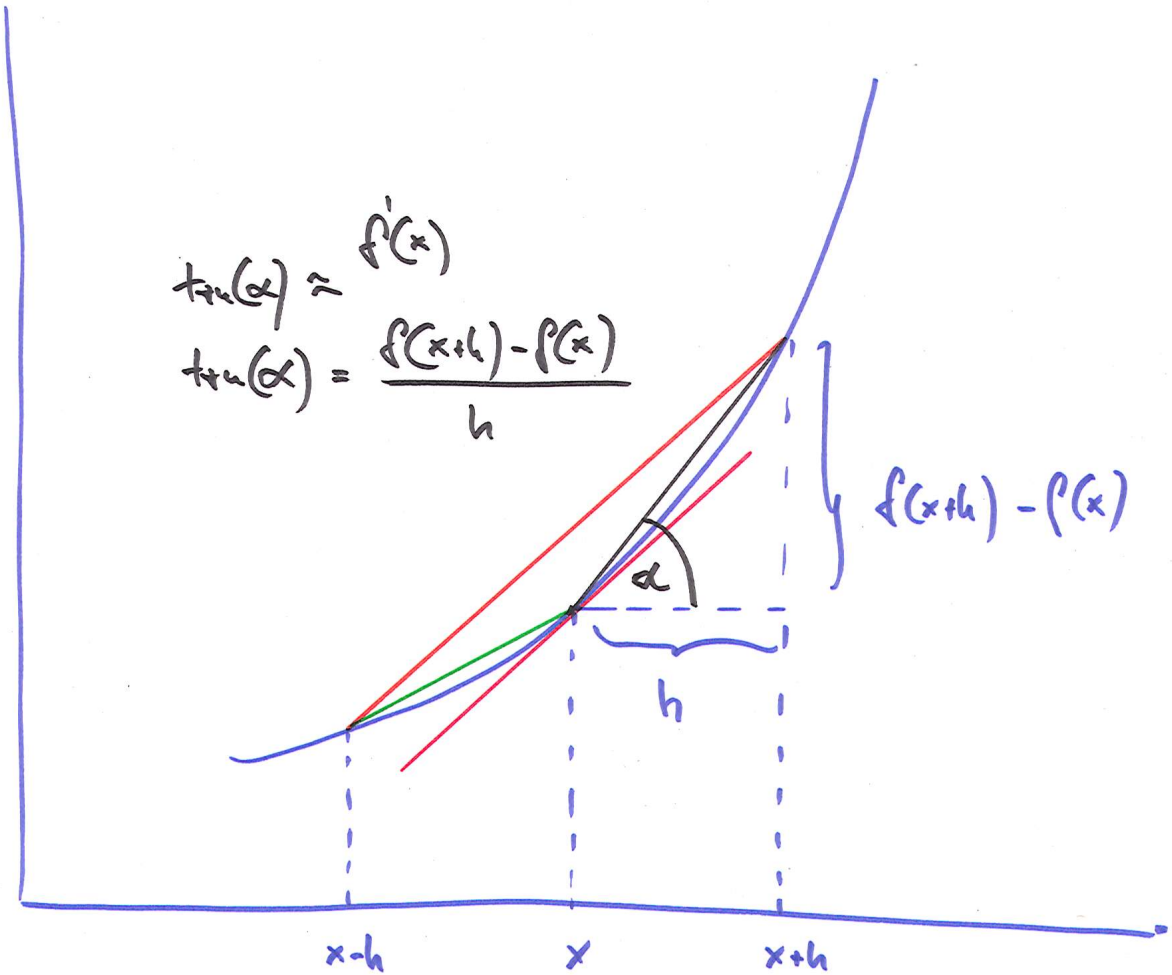
the individual columns of which can be approximated at all points $\mathbf{x}^* \in D_F$ by (forward, backward, central) finite difference quotients as follows:

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

[2] to be relaxed later

Numerical Approximation of First Derivatives

Finite Difference Quotients



Accuracy of Forward/Backward Finite Differences

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

$$f(x^0 + h) = f(x^0) + \frac{df}{dx}(x^0) \cdot h + \frac{1}{2!} \cdot \frac{d^2 f}{dx^2}(x^0) \cdot h^2 + \frac{1}{3!} \cdot \frac{d^3 f}{dx^3}(x^0) \cdot h^3 + \dots$$

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

$$f(x^0 - h) = f(x^0) - \frac{df}{dx}(x^0) \cdot h + \frac{1}{2!} \cdot \frac{d^2 f}{dx^2}(x^0) \cdot h^2 - \frac{1}{3!} \cdot \frac{d^3 f}{dx^3}(x^0) \cdot h^3 + \dots \quad .$$

Truncation after the respective first derivative terms yields scalar univariate versions of forward and backward finite difference quotients, respectively, e.g, from

$$f(x^0 + h) = f(x^0) + h\frac{df}{dx}(x^0) + O(h^2) \quad .$$

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, e.g,

$$\frac{df}{dx}(x^0) = \frac{f(x^0 + h) - f(x^0) + O(h^2)}{h} = \frac{f(x^0 + h) - f(x^0)}{h} + O(h)$$

Accuracy of Central Finite Differences

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

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

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, i.e,

$$
\frac{df}{dx}(x^0) == \frac{f(x^0 + h) - f(x^0 - h) + O(h^3)}{2h} = \frac{f(x^0 + h) - f(x^0 - h)}{2h} + O(h^2).
$$

Accuracy of Finite Differences

Case Study (sin.cpp)

```
1 double f(double x) { return sin(x); }
2
3 int main() {
4   double x=1;
5   for (double h=1e-1;h>=1e-15;h=h/10)
6     cout << h << "\t" << (f(x+h)-f(x))/h << "\t"
7          << (f(x+h/2)-f(x-h/2))/h << "\t" << cos(x) << endl;
8 }
```

h	FFD	CFD	EXACT
0.1	0.497363752535389	0.540077208046432	0.54030230586814
0.01	0.536085981011869	0.540300054611342	0.54030230586814
0.001	0.539881480360327	0.540302283355554	0.54030230586814
0.0001	0.540260231418621	0.540302305643836	0.54030230586814
1e-05	0.540298098505865	0.540302305873652	0.54030230586814
1e-06	0.54030188512133	0.540302305895857	0.54030230586814
1e-07	0.540302264040449	0.54030230628924	0.54030230586814
1e-08	0.540302302898254	0.540302291796024	0.54030230586814
1e-09	0.540302358409406	0.540302358409406	0.54030230586814
1e-10	0.540302247387103	0.540303357610128	0.54030230586814
1e-11	0.540301137164079	0.540301137164079	0.54030230586814
1e-12	0.540345546085064	0.540345546085064	0.54030230586814
1e-13	0.539568389967826	0.539568389967826	0.54030230586814
1e-14	0.544009282066327	0.544009282066327	0.54030230586814
1e-15	0.555111512312578	0.555111512312578	0.54030230586814

We are looking for the solution $x(p, T),\ T > 0$ of the scalar initial value problem

$$\frac{dx}{dt} = g(x(p(t), t), p(t), t), \quad x(p(0), 0) = x^0.$$

Forward finite differences in time with time step $0 < \delta t \ll 1$ yield

$$\frac{x(p(t + \delta t), t + \delta t) - x(p(t), t)}{\delta t} \approx g(x(p(t), t), p(t), t)$$

and hence the explicit Euler evolution

$$x^{i+1} := x^i + \delta t \cdot g(x^i, p^i, i \cdot \delta t)$$

for $i = 0, \ldots, n-1$, $T = n \cdot \delta t$, and $p^i = p(i \cdot \delta t)$.

We are interested in sensitivities of the final state $x^n = x^n(\mathbf{p}, T)$ wrt. intial state x^0, target time T and parameter vector $\mathbf{p} = (p^0, \ldots, p^{n-1})^T$.

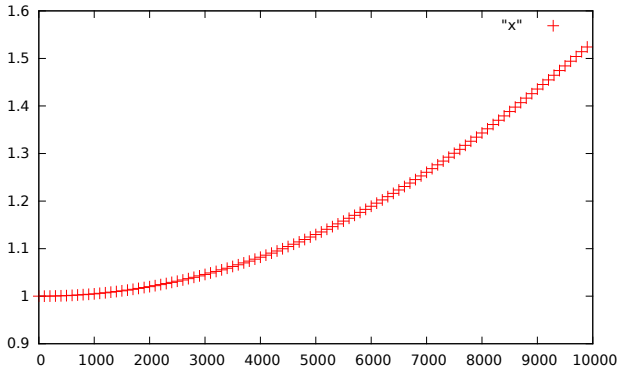
Numerical Approximation of First Derivatives

Example: Implementation

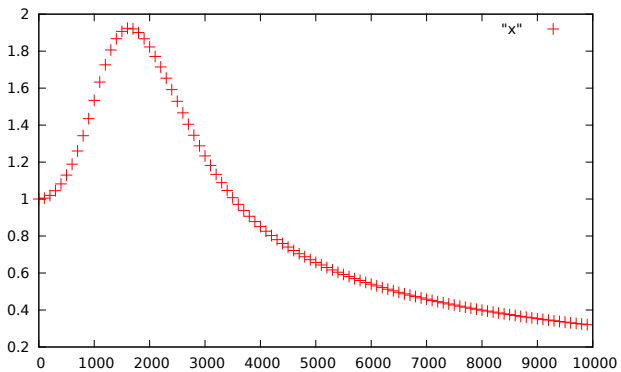
E.g,

$$\frac{dx}{dt} = p(t) \cdot \sin(x(p(t), t) \cdot t)$$

```
void explicit_euler(const int n, double& x,
      const vector<double>& p, const double& T) {
   double dt=T/n, t;
   t=0;
   for (int i=0;i<n;i++) {
      x+=dt*p[i]*sin(x*t);
      t+=dt;
   }
}
```



T=1



T=10

Numerical Approximation of First Derivatives

Example: Forward Finite Differences

```
int main(int c, char* v[]) {
  int n=atoi(v[1]);
  const double x0=1, T=1;
  vector<double> p(n,1);
  double x=x0;
  explicit_euler(n,x,p,T);
  double h=sqrt(DBL_EPSILON);
  double xp=x0+h;
  explicit_euler(n,xp,p,T);
  cout << "dx/dx0=" << (xp-x)/h << endl;
  for (int i=0;i<n;i++) {
    xp=x0;
    p[i]+=h;
    explicit_euler(n,xp,p,T);
    cout << "dx/dp[" << i << "]=" << (xp-x)/h << endl;
    p[i]-=h;
  }
  return 0;
}
```

Race against adjoint ...

n	primal	ffd	adjoint
10^3	~ 0	0.1	~ 0
10^4	~ 0	10	0.02
10^5	~ 0	597	0.15

Again, we are looking for $x(T),\ T > 0$ as the solution to the scalar initial value problem

$$\frac{dx}{dt} = g(x(p(t),t),p(t),t), \quad x(p(0),0) = x^0.$$

Backward finite differences in time with time step $0 < \delta t \ll 1$ yield

$$\frac{x(p(t),t) - x(p(t-\delta t),t-\delta t)}{\delta t} \approx g(x(p(t),t),p(t),t)$$

and hence $x = x(t) = x(p(t),t)$ as the solution of the nonlinear equation

$$f(x) = x - x(t-\delta t) - \delta t \cdot g(x,p(t),t) = 0,$$

e.g. by the Newton-Raphson algorithm.

E.g,

$$\frac{dx}{dt} = p(t) \cdot \sin(x(p(t), t) \cdot t)$$

```
void implicit_euler(const int n, double& x,
     const vector<double>& p, const double& T) {
  double dt=T/n;
  for (int i=0;i<n;i++) {
    double x_prev=x;
    double f=-dt*p[i]*sin(x*i*dt);
    while (abs(f)>1e-15) {
      x=x-f/(1-dt*p[i]*i*dt*cos(x*i*dt));
      f=x-x_prev-dt*p[i]*sin(x*i*dt);
    }
  }
}
```

We are looking for the expected value $\mathbb{E}(x)$ of the solution $x(\mathbf{p}, T), T > 0$ of the scalar stochastic initial value problem

$$dx = f(x(\mathbf{p}, t), \mathbf{p}, t))dt + g(x(\mathbf{p}, t), \mathbf{p}, t)dW$$

with Brownian Motion dW and for $x(\mathbf{p}, 0) = x^0$.

Forward finite differences in time with time step $0 < \delta t \ll 1$ yield the explicit Euler-Maruyama evolution

$$x^{i+1} := x^i + \delta t \cdot f(x^i, \mathbf{p}, i \cdot \delta t) + \sqrt{\delta t} \cdot g(x^i, \mathbf{p}, i \cdot \delta t) \cdot dW^i$$

for $i = 0, \ldots, n-1$, target time $T = n \cdot \delta t$, parameter vector $\mathbf{p} \in \mathbb{R}^l$, and with random numbers dW^i drawn from the standard normal distribution $N(0, 1)$.

The solution $\mathbb{E}(x)$ is approximated using Monte Carlo simulation over (a sufficiently large number of) Euler-Maruyama paths.

We are interested in sensitivities of the final state $\mathbb{E}(x)$ wrt. $\mathbf{p}$.

E.g,

$$dx = p(t) \cdot \sin(x(p(t), t) \cdot t)dt + p(t) \cdot \cos(x(p(t), t) \cdot t)dW$$

```
void euler_maruyama(const int m, const int n,
    double& x, const vector<double>& p,
    const double& T, const vector<vector<double>>& dW) {
  double s=0, x0=x; double dt=T/n, t;
  for (int j=0;j<m;j++) {
    t=0;
    for (int i=0;i<n;i++) {
      x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];
      t+=dt;
    }
    s+=x; x=x0;
  }
  x=s/m;
}
```

Practice

Using

```
explicit_euler
    adjoint.cpp fd.cpp primal.cpp
euler_maruyama
   adjoint.cpp primal.cpp
implicit_euler
   adjoint.cpp primal.cpp
```

- apply forward and/or central finite differences to Implicit Euler and Euler-Maruyama,
- compare run times and numerical results with given adjoint,
- investigate impact of different perturbation sizes on quality of the numerical results.

Outline

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)

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

   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

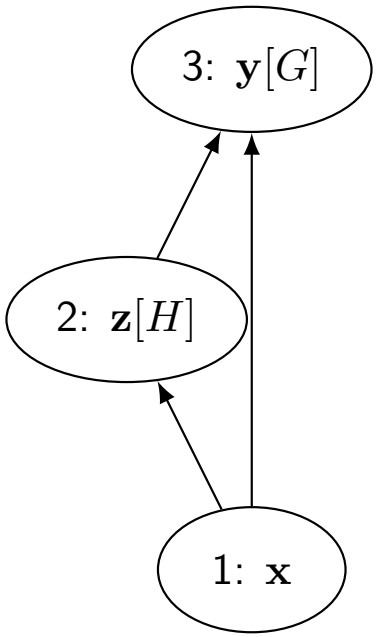
$$d_{j,i} \equiv \frac{d\varphi_j}{dv_i}(v_k)_{k \prec j}$$

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

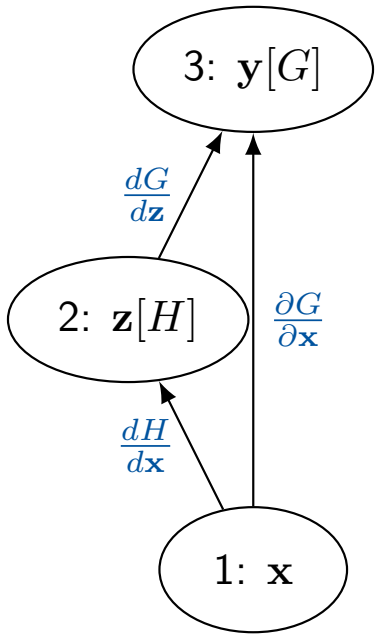
3. A linearized SAC (ISAC) is obtained by augmenting the elemental assignments with computations of the local partial derivatives $d_{j,i}$.
4. The SAC induces a directed acyclic graph (DAG) $G = G(F) = (V, E)$ with integer vertices $V = \{0, \ldots, n+q\}$ and edges $V \times V \supseteq E = \{(i,j) : i \prec j\}$.
5. The set of vertices representing the n inputs is denoted as $X \subseteq V$. The m outputs are collected in $Y \subseteq V$. All remaining intermediate vertices belong to $Z \subsetneq V$.
6. A linearized DAG (IDAG) is obtained by attaching the $d_{j,i}$ to the corresponding edges (i,j) in the DAG.
7. Example: explicit Euler step

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

DAG:



IDAG:



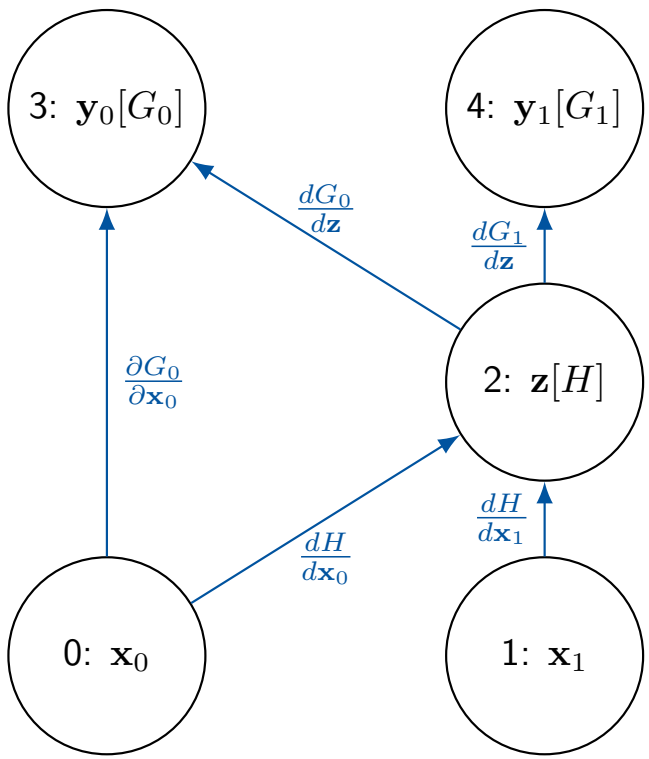
$$\nabla F(\mathbf{x}) \equiv \frac{d\mathbf{y}}{d\mathbf{x}} = \sum_{\text{path} \in \text{IDAG}} \prod_{(i,j) \in \text{path}} d_{j,i}$$

Example: explicit Euler step

Chain Rule on IDAG

... the Essence of AD

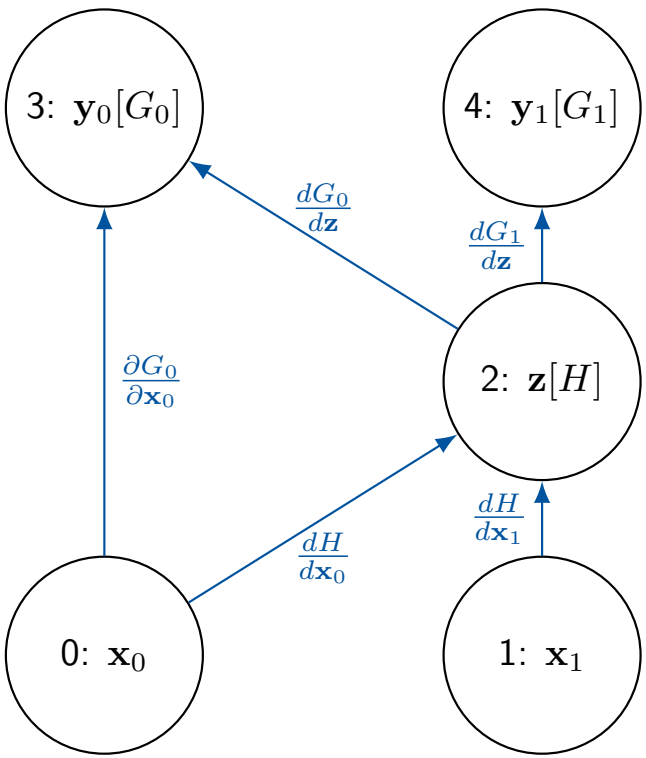
Consider $\mathbf{y} = F(\mathbf{x}) = \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} F_0(\mathbf{x}_0, \mathbf{x}_1) \\ F_1(\mathbf{x}_0, \mathbf{x}_1) \end{pmatrix} = \begin{pmatrix} G_0(\mathbf{x}_0, H(\mathbf{x}_0, \mathbf{x}_1)) \\ G_1(H(\mathbf{x}_0, \mathbf{x}_1)) \end{pmatrix}$

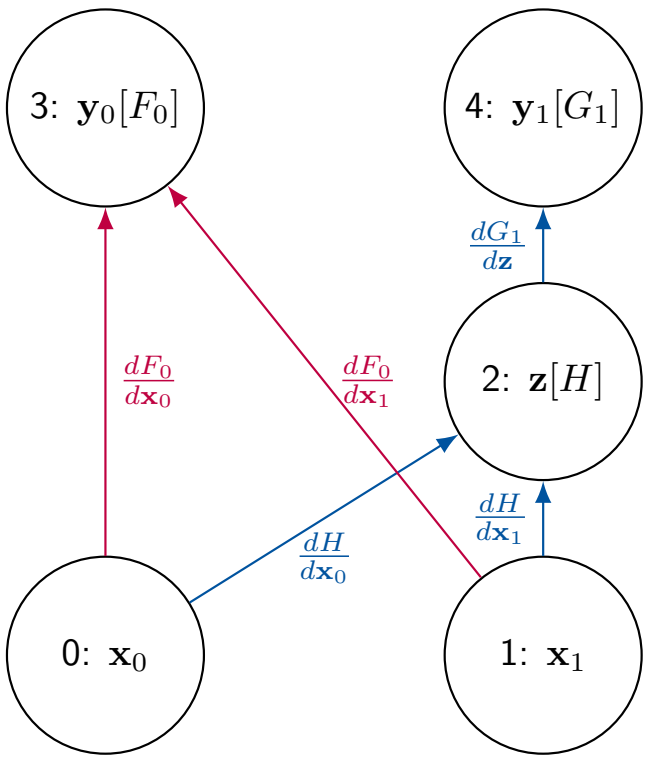


Chain Rule on IDAG

... Edge Back-Elimination (to be used for tangents)

$$\frac{dF_0}{d\mathbf{x}} = \begin{pmatrix} \frac{dF_0}{d\mathbf{x}_0} \\ \frac{dF_0}{d\mathbf{x}_1} \end{pmatrix}^T = \begin{pmatrix} \frac{\partial G_0}{\partial \mathbf{x}_0} + \frac{dG_0}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}_0} \\ \frac{dG_0}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}_1} \end{pmatrix}^T$$

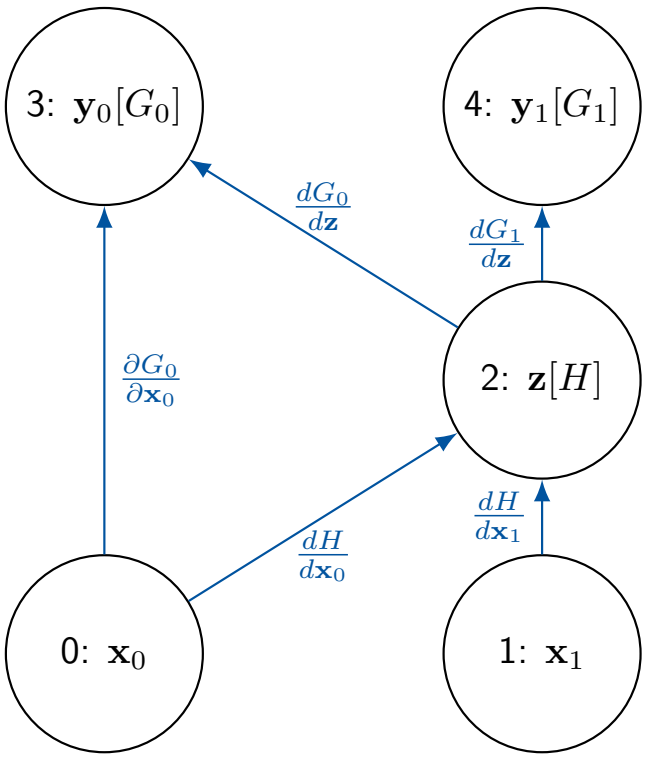


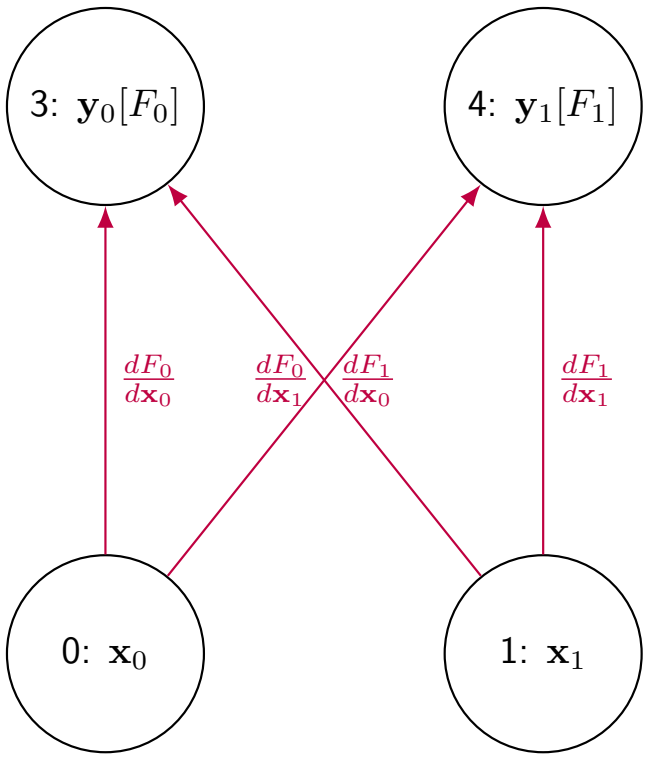


Chain Rule on IDAG

... Vertex Elimination (to be used for adjoints)

$$\frac{dF}{d\mathbf{x}} = \begin{pmatrix} \frac{dF_0}{d\mathbf{x}_0} & \frac{dF_0}{d\mathbf{x}_1} \\ \frac{dF_1}{d\mathbf{x}_0} & \frac{dF_1}{d\mathbf{x}_1} \end{pmatrix} = \begin{pmatrix} \frac{\partial G_0}{\partial \mathbf{x}_0} + \frac{dG_0}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}_0} & \frac{dG_0}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}_1} \\ \frac{dG_1}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}_0} & \frac{dG_1}{d\mathbf{z}} \cdot \frac{dH}{d\mathbf{x}_1} \end{pmatrix}$$





- [7] U. N.: *Optimal Jacobian accumulation is NP-complete.* Math. Prog. 112(2):427–441, Springer, 2008.

  Proof by reduction from Ensemble Computation

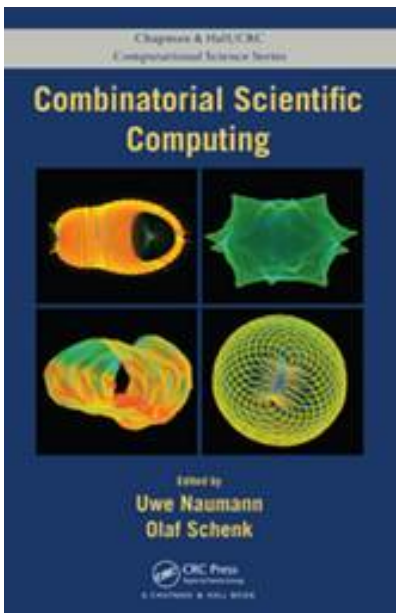
- [6] U. N.: *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph.* Math. Prog. 99(3):399–421, Springer, 2004.

  Example: bat graph in STCE logo

- [2] A. Griewank and U. N.: *Accumulating Jacobians as chained sparse matrix products.* Math. Prog. 95(3):555–571, Springer, 2003.

  Example: $\mathbb{R}^4 \to \mathbb{R}^2 \to \mathbb{R}^2 \to \mathbb{R}^2 \to \mathbb{R}^4$

See also [9]:

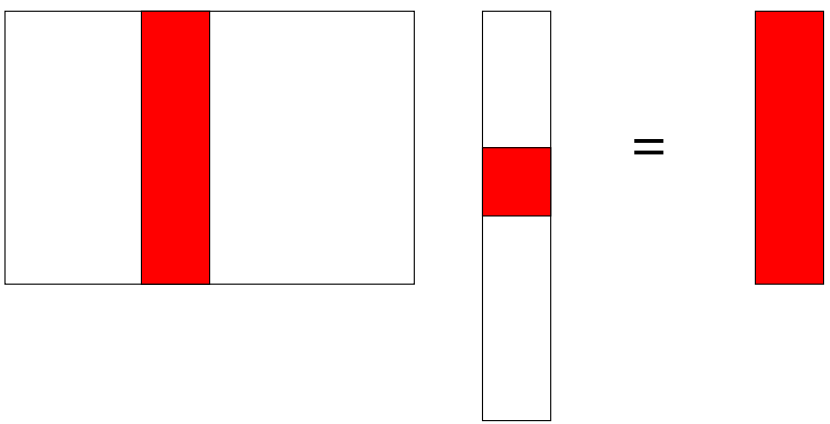


A first-order tangent model $F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m$,

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

defines a directional derivative alongside with the function value:

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



... definition of the whole Jacobian column-wise by input directions $\mathbf{x}^{(1)} \in \mathbb{R}^n$ equal to the Cartesian basis vectors in $\mathbb{R}^n$.

In

$$\frac{dF}{d\mathbf{x}} \cdot \mathbf{x}^{(1)}$$

the superscript on $\mathbf{x}$ denotes first directional differentiation of F performed in tangent mode in direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$.

Subscripts will be used later to denote adjoints.

Larger values for superscripts will become relevant in the context of higher derivatives.

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

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

computes a Jacobian $\times$ vector product alongside with the function value:

$$\mathbb{R}^m \times \mathbb{R}^{\tilde{m}} \ni \begin{pmatrix} \mathbf{z} \\ \tilde{\mathbf{z}} \\ \mathbf{y} \\ \tilde{\mathbf{y}} \end{pmatrix} := F(\mathbf{x}, \tilde{\mathbf{x}}, \mathbf{z}, \tilde{\mathbf{z}})$$

$$\mathbb{R}^m \ni \begin{pmatrix} \mathbf{z}^{(1)} \\ \mathbf{y}^{(1)} \end{pmatrix} := \nabla F(\mathbf{x}, \tilde{\mathbf{x}}, \mathbf{z}, \tilde{\mathbf{z}}) \cdot \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix}$$

Variables for which derivatives are computed are referred to as active; $\mathbf{x}$ and $\mathbf{z}$ are active inputs; $\mathbf{z}$ and $\mathbf{y}$ are active outputs.

Variables which depend on active inputs are referred to as varied.

Variables for which no derivatives are computed are referred to as passive; $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{z}}$ are passive inputs; $\tilde{\mathbf{z}}$ and $\tilde{\mathbf{y}}$ are passive outputs.

Variables which active outputs depend on are referred to as useful.

Active variables are both varied and useful.

The whole (dense) Jacobian can be *harvested* column-wise from the active output directions $(\mathbf{z}^{(1)}, \mathbf{y}^{(1)})^T \in \mathbb{R}^m$ by *seeding* active input directions $(\mathbf{x}^{(1)}, \mathbf{z}^{(1)})^T \in \mathbb{R}^n$ with the Cartesian basis vectors in $\mathbb{R}^n$.

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 $\times$ vector product alongside with the function value:

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

Define

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

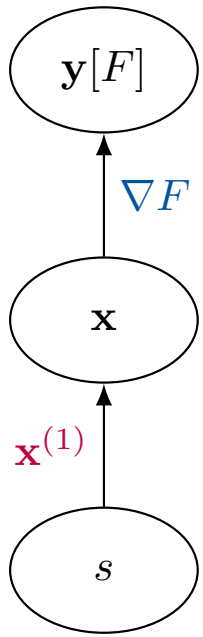
for $\boldsymbol{v} \in \{\mathbf{x}, \mathbf{y}\}$ and some auxiliary $s \in \mathbb{R}$ assuming that $F(\mathbf{x}(s))$ is continuously differentiable over its domain.

By the chain rule

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

and hence

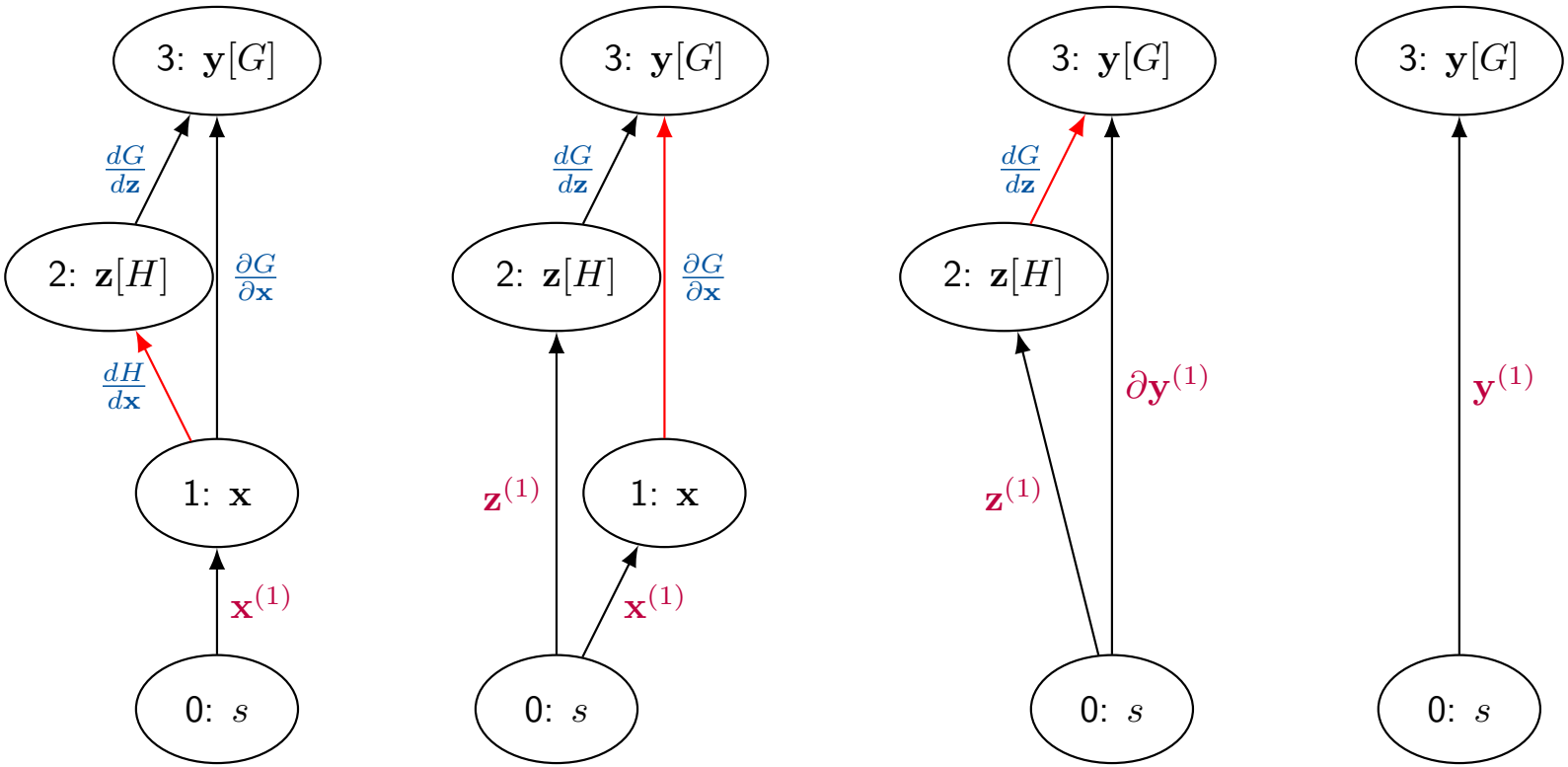
$$\mathbf{y}^{(1)} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}.$$



Example: explicit Euler step

First-Order Tangent Code

Implementation as Edge Back Elimination on IDAG



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).

For $i = 0, \ldots, n-1$

$$\begin{pmatrix} v_i \\ v_i^{(1)} \end{pmatrix} := \begin{pmatrix} \{x, z\}_i \\ \{x, z\}_i^{(1)} \end{pmatrix} \qquad \text{(seed)}$$

For $i = n, \ldots, q-1$

$$\begin{pmatrix} v_i \\ v_i^{(1)} \end{pmatrix} := \begin{pmatrix} \varphi_i(v_k)_{k \prec i} \\ \sum_{j \prec i} \frac{d\varphi_i(v_k)_{k \prec i}}{dv_j} \cdot v_j^{(1)} \end{pmatrix} \qquad \text{(propagate)}$$

For $i = 0, \ldots, m-1$

$$\begin{pmatrix} \{z, y\}_i \\ \{z, y\}_i^{(1)} \end{pmatrix} := \begin{pmatrix} v_{n+p+i} \\ v_{n+p+i}^{(1)} \end{pmatrix} \qquad \text{(harvest)}$$

1. duplication of active data segment

   Each variable is augmented with its tangent version.

2. assignment-level tangent models

   Construction of assignment-level single assignment code by assigning the results of all elemental functions to unique intermediate variables makes implementation of the local gradients and their concatenation according to the chain rule straight forward.

3. flow of control remains unchanged

   In an imperative program control flow statements (loops, branches, gotos, ...) determine the order in which the assignments are executed, which must be preserved in the tangent program.

4. tangent subprogram calls

   Replacing subprogram calls with calls of their tangent versions ensures correct propagation of the directional derivatives through the calling hierarchy.

First-Order Tangents by Hand-Coding

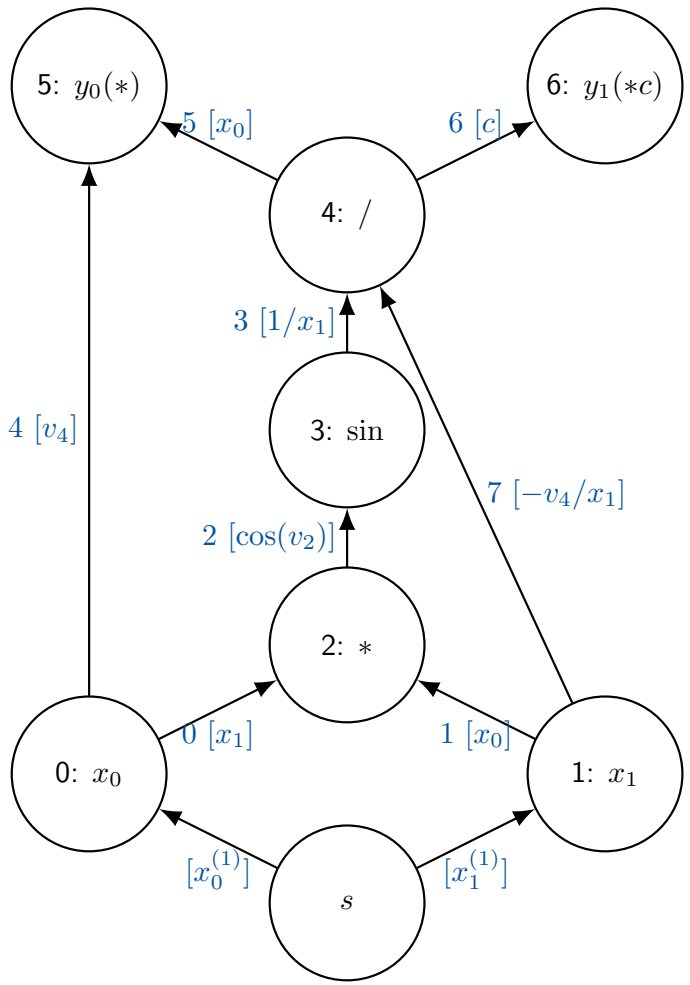
Example: Explicit Euler (Driver)

```
int main(int c, char* v[]) {
  int n=atoi(v[1]);
  const double x0=1, T=1;
  vector<double> p(n,1),pt(n,0);
  double x=x0, xt=1;
  explicit_euler(n,x,xt,p,pt,T);
  cout << "x=" << x << endl;
  cout << "dx/dx0=" << xt << endl;
  for (int i=0;i<n;i++) {
    x=x0; xt=0; pt[i]=1;
    explicit_euler(n,x,xt,p,pt,T);
    cout << "dx/dp[" << i << "]=" << xt << endl;
    pt[i]=0;
  }
  return 0;
}
```

First-Order Tangents by Hand-Coding

Example: Explicit Euler (Tangent)

```
void explicit_euler(const int n, double& x, double &xt,
      const vector<double>& p, const vector<double>& pt,
      const double& T
) {
   double dt=T/n, t;
   t=0;
   for (int i=0;i<n;i++) {
      xt+=dt*pt[i]*sin(x*t)+dt*p[i]*t*cos(x*t)*xt;
      x+=dt*p[i]*sin(x*t);
      t+=dt;
   }
}
```



Tangent IDAG

We consider

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 * \sin(x_0 * x_1)/x_1 \\ \sin(x_0 * x_1)/x_1 * c \end{pmatrix}$$

implemented as

$$t := \sin(x_0 * x_1)/x_1$$
$$y_0 := x_0 * t;\ y_1 := t * c$$

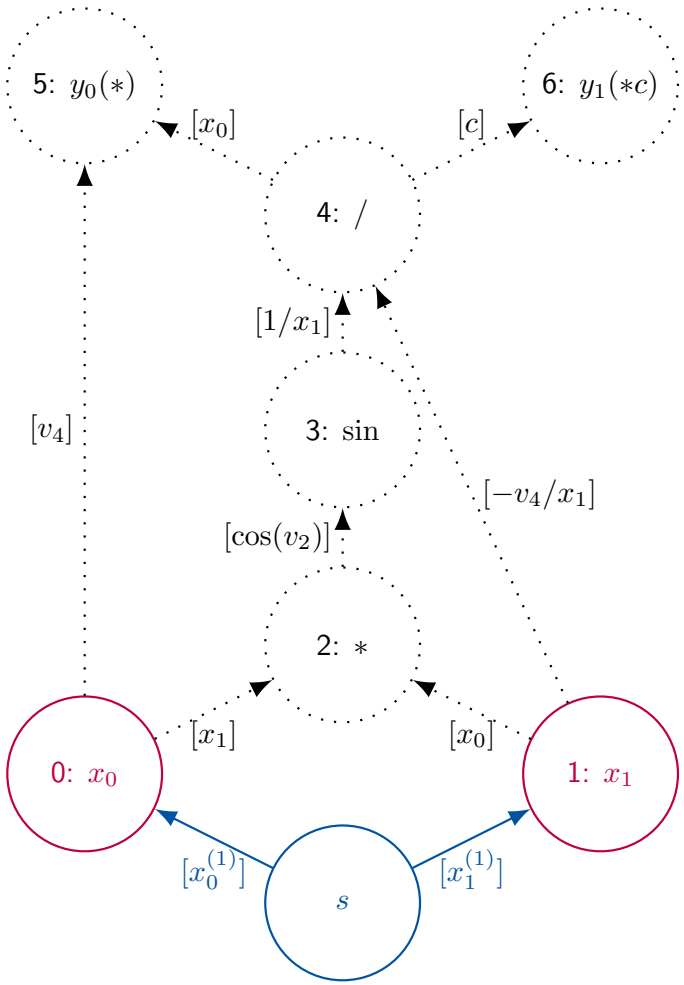
yielding SAC

$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3/x_1$$
$$y_0 := x_0 * v_4;\ y_1 := v_4 * c$$

for some *passive* value c, i.e, no derivatives of or with respect to required; $\mathbf{x}, \mathbf{y}$, and t are *active*.

First-Order Tangents by Overloading

Seed

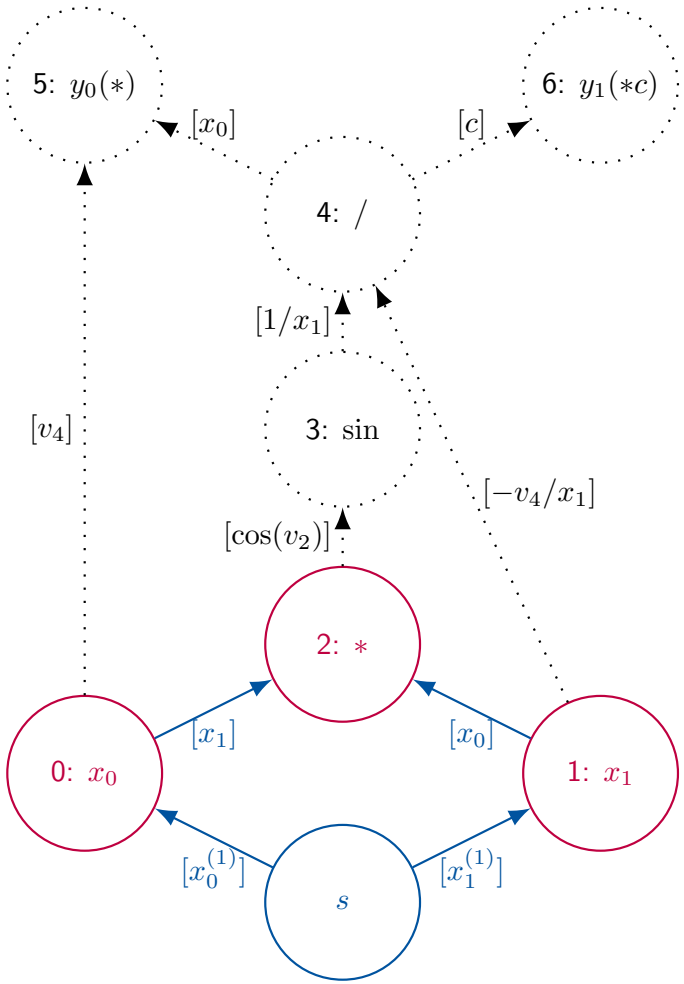


$$x_0 := ?$$
$$x_1 := ?$$

$$x_0^{(1)} := ?$$
$$x_1^{(1)} := ?$$

First-Order Tangents by Overloading

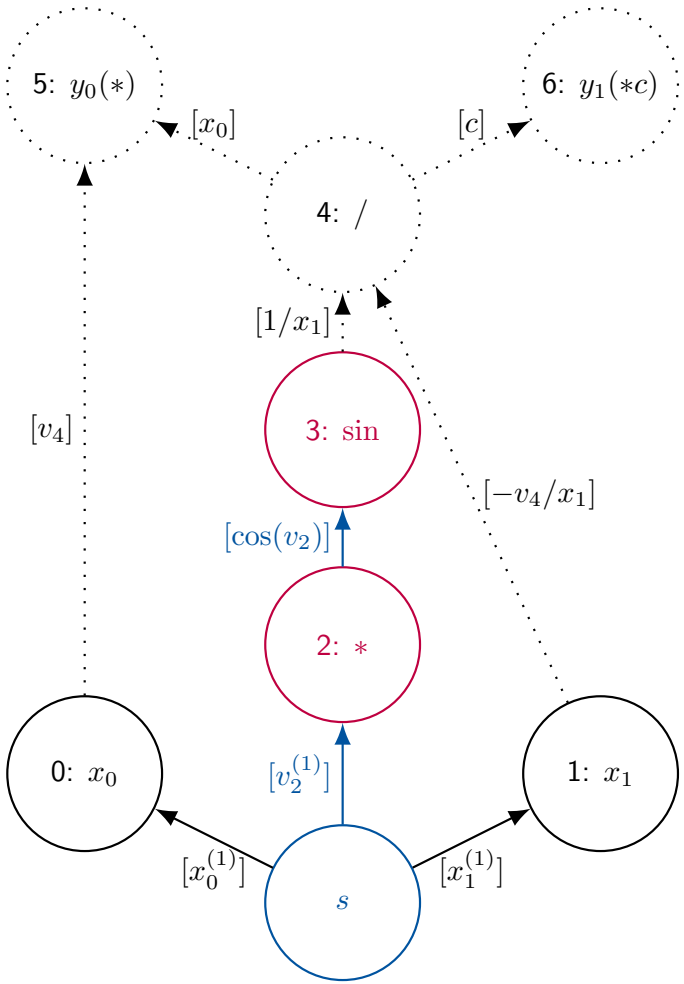
Propagate (Local Directional Derivatives)



$$v_2 := x_0 * x_1$$
$$v_2^{(1)} := x_1 * x_0^{(1)} + x_0 * x_1^{(1)}$$

First-Order Tangents by Overloading

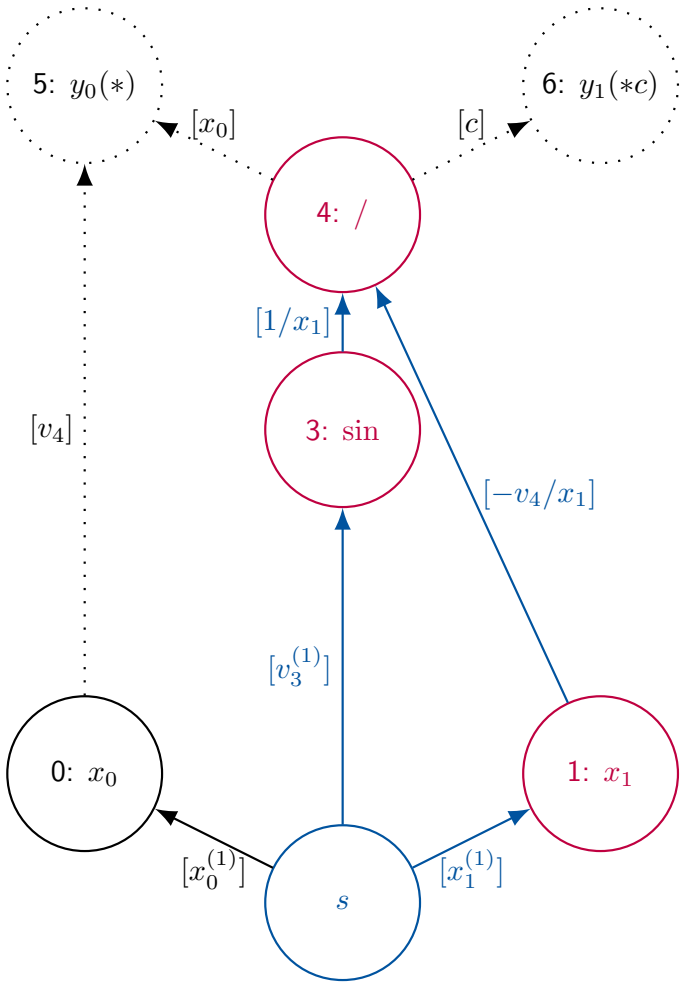
Propagate



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)}
\end{aligned}
$$

First-Order Tangents by Overloading

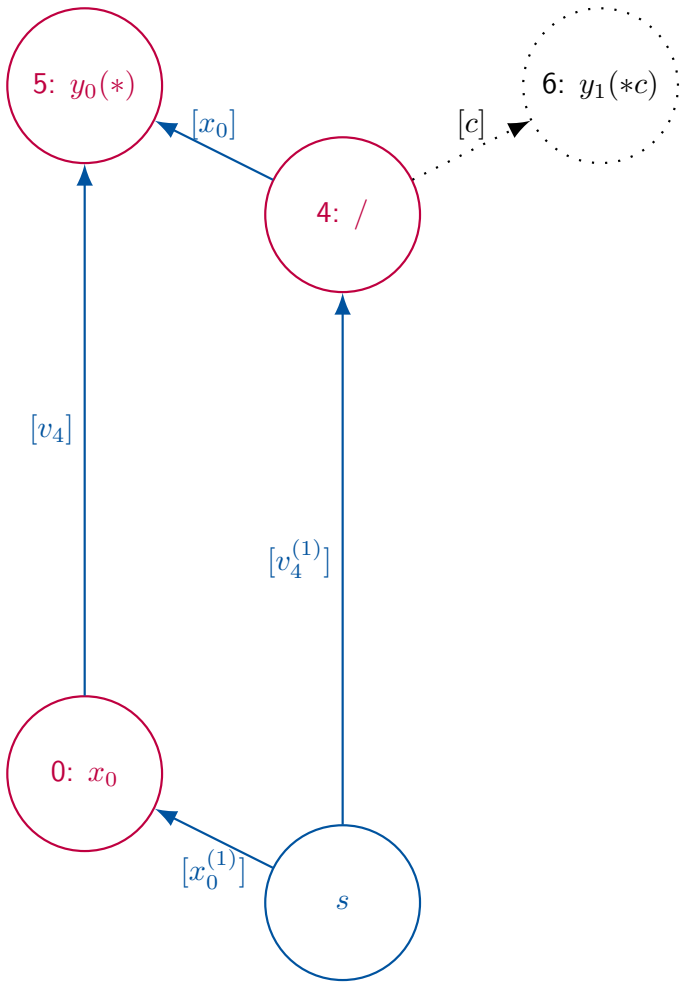
Propagate



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1
\end{aligned}
$$

First-Order Tangents by Overloading

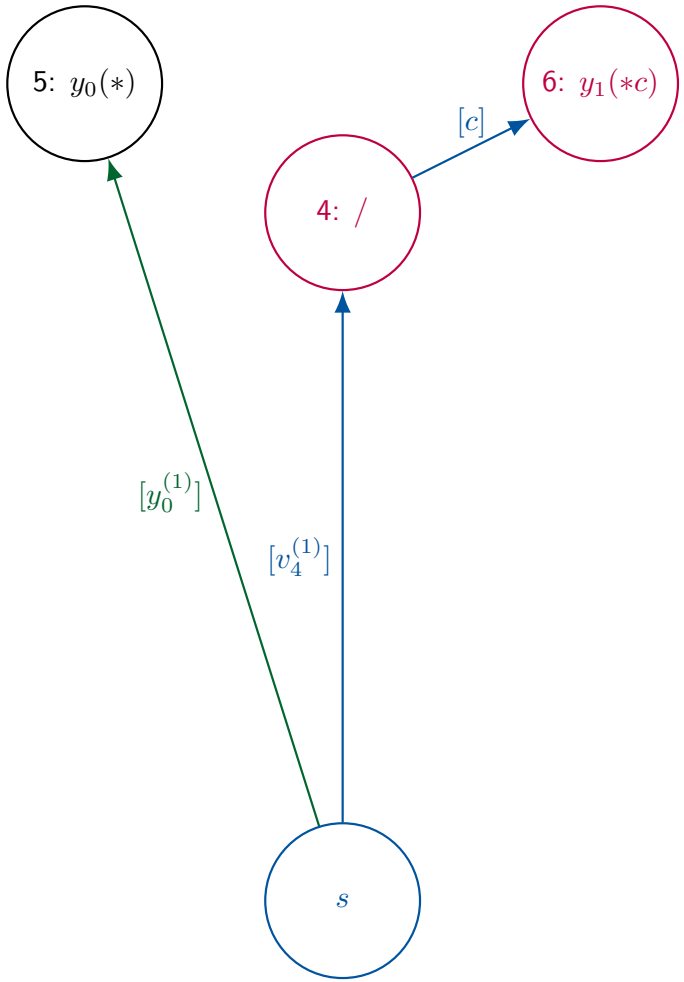
Propagate



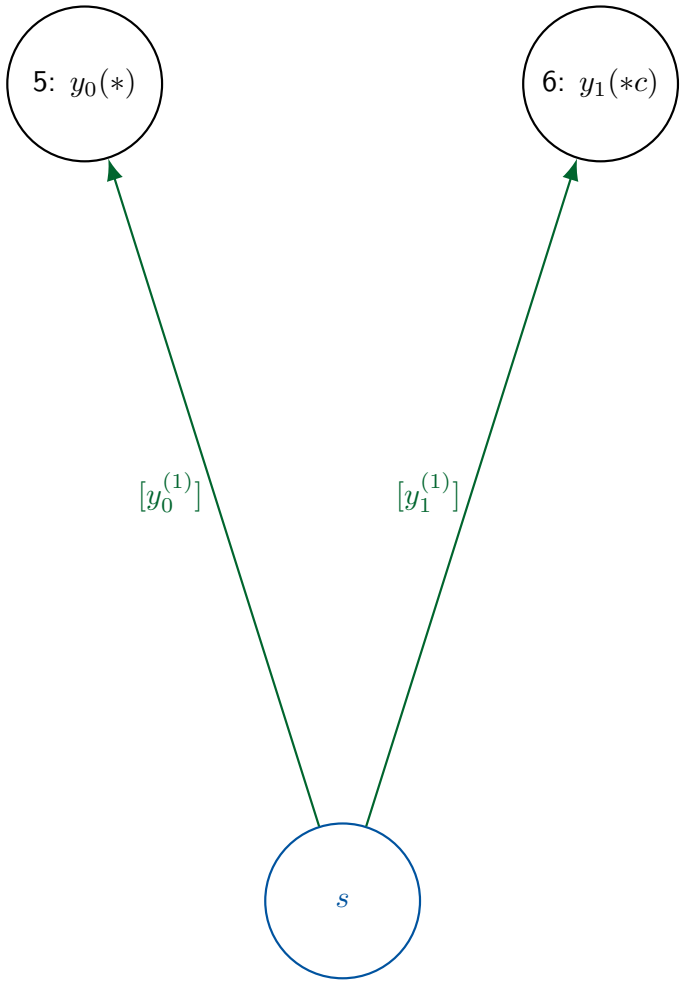
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1 \\
y_0 &:= x_0 * v_4 \\
y_0^{(1)} &:= v_4 * x_0^{(1)} + x_0 * v_4^{(1)}
\end{aligned}
$$

First-Order Tangents by Overloading

Propagate



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1 \\
y_0 &:= x_0 * v_4 \\
y_0^{(1)} &:= v_4 * x_0^{(1)} + x_0 * v_4^{(1)} \\
y_1 &:= v_4 * c \\
y_1^{(1)} &:= c * v_4^{(1)}
\end{aligned}
$$



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1 \\
y_0 &:= x_0 * v_4 \\
y_0^{(1)} &:= v_4 * x_0^{(1)} + x_0 * v_4^{(1)} \\
y_1 &:= v_4 * c \\
y_1^{(1)} &:= c * v_4^{(1)}
\end{aligned}
$$

RWTH AACHEN UNIVERSITY STCE

dco/c++ features

- tangents and adjoints of arbitrary order through recursive template instantiation for numerical simulation code implemented in C++
- front-ends for Fortran, C#, Matlab, Python (3x alpha)
- optimized assignment-level gradient code through expression templates
- cache-optimized internal representation in various incarnations
- vector modes / detection and exploitation of sparsity
- external adjoint / Jacobian interfaces
- user-defined intrinsics
- intrinsic NAG Library functions (e.g. Linear Algebra, Interpolation, Root Finding, Nearest Correlation Matrix)
- support for parallelism: thread-safe data structures, adjoint MPI library, GPU coupling, meta adjoint programming (map)

Type-generic primal

```
template<typename AT, typename PT>
void explicit_euler(const int n, AT& x,
     const vector<AT>& p, const PT& T) {
  PT dt=T/n, t;
  t=0;
  for (int i=0;i<n;i++) {
    x+=dt*p[i]*sin(x*t);
    t+=dt;
  }
}
```

Tangents by Overloading with dco/c++

Example: Explicit Euler

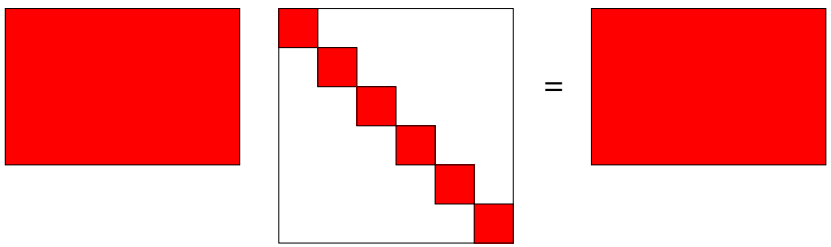
```
#include "dco.hpp"
typedef dco::gt1s<double>::type DCO_T;

#include "explicit_euler.h"

int main(int c, char* v[]) {
  int n=atoi(v[1]);
  const double x0=1, T=1;
  vector<DCO_T> p(n,1); DCO_T x=x0;
  dco::derivative(x)=1.0;
  explicit_euler(n,x,p,T);
  cout << "x=" << dco::value(x) << endl;
  cout << "dx/dx0=" << dco::derivative(x) << endl;
  for (int i=0;i<n;i++) {
    x=x0;
    dco::derivative(p[i])=1.0;
    explicit_euler(n,x,p,T);
    cout << "dx/dp[" << i << "]=" << dco::derivative(x) << endl;
    dco::derivative(p[i])=0.0;
  }
  return 0;
}
```

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 $\times$ matrix product alongside with the function value:

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



... harvesting 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!

Vector Tangents by Overloading with dco/c++

Example: Explicit Euler

```
#include "dco.hpp"
const size_t vs=10;
typedef dco::gt1v<double,vs>::type DCO_T;

#include "explicit_euler.h"

int main(int c, char* v[]) {
  int n=atoi(v[1]);
  const double x0=1, T=1;
  vector<DCO_T> p(n,1); DCO_T x=x0;
  dco::derivative(x)[0]=1.0;
  for (int i=0;i<n;i++) dco::derivative(p[i])[i+1]=1.0;
  explicit_euler(n,x,p,T);
  cout << "x=" << dco::value(x) << endl;
  cout << "dx/dx0=" << dco::derivative(x)[0] << endl;
  for (int i=0;i<n;i++)
    cout << "dx/dp[" << i << "]=" << dco::derivative(x)[i+1] << endl;
  return 0;
}
```

Note: typically n%vs calls of vector tangent code

Practice

Using

```
explicit_euler
    dco
       explicit_euler.h gt1s.cpp gt1v.cpp
    hand
      tangent.cpp
```

and sample code from lecture 1

- write tangent versions of Implicit Euler and Euler-Maruyama codes,
- use dco/c++ to derive tangent versions of Implicit Euler and Euler-Maruyama codes,
- compare numerical results and run times with finite differences

First-Order Adjoint Model

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.

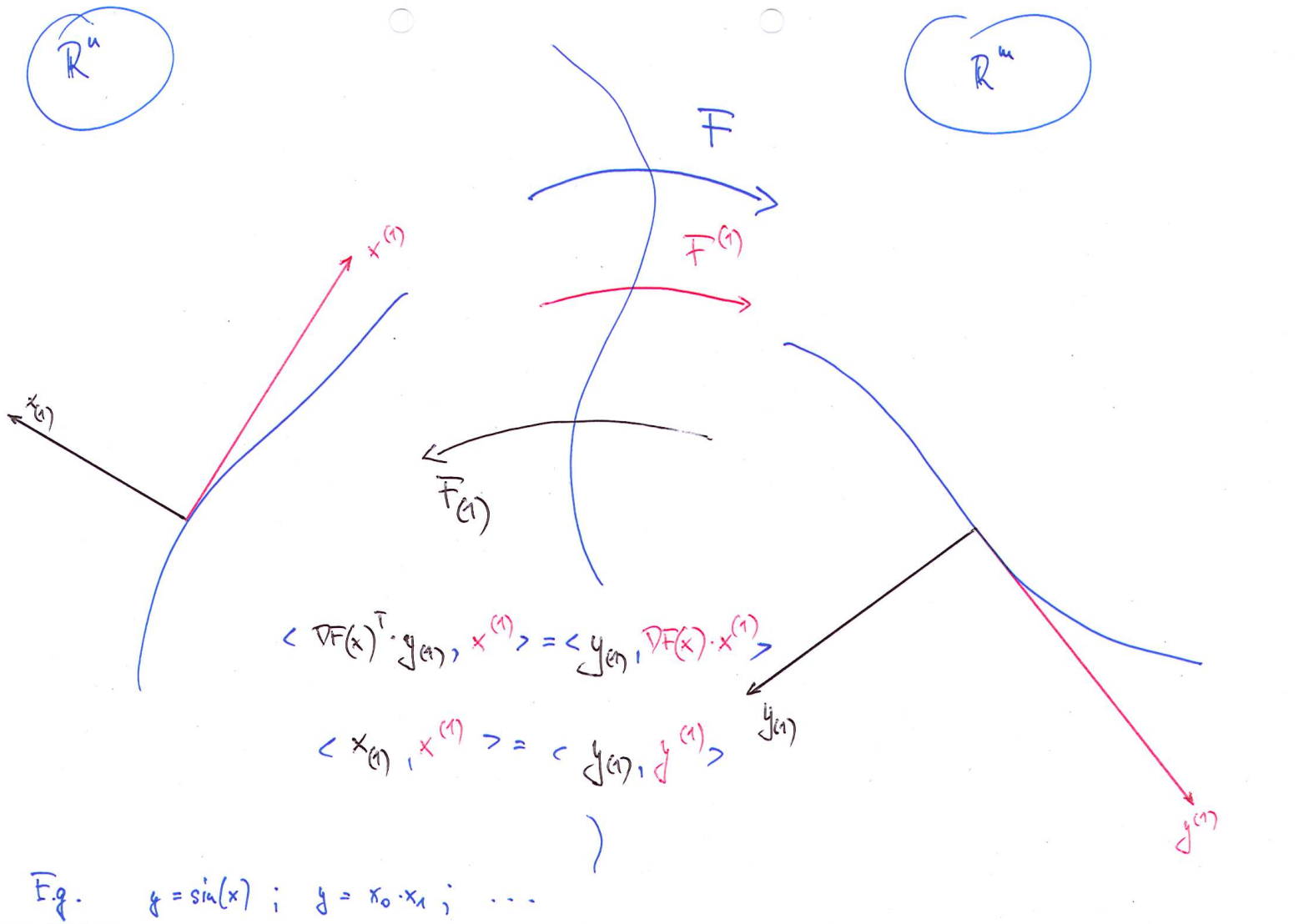
Theorem

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

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

Note invariant at each point in the program execution → validation of derivatives.

First-Order Adjoint Model

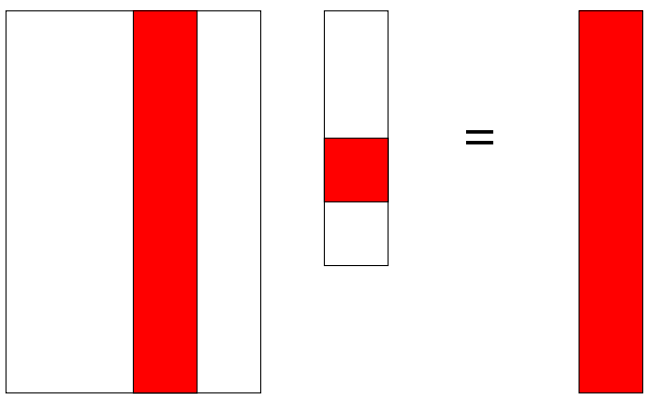


A first-order adjoint model $F_{(1)} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^n$,

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

defines an adjoint directional derivative alongside with the function value:

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



... definition of the whole Jacobian row-wise through input directions $\mathbf{y}_{(1)} \in \mathbb{R}^m$ equal to the Cartesian basis vectors in $\mathbb{R}^m$.

In

$$\left(\frac{dF}{d\mathbf{x}}\right)^T \cdot \mathbf{y}_{(1)}$$

the subscript on $\mathbf{y}$ denotes the first directional differentiation of F performed in adjoint mode in direction $\mathbf{y}_{(1)} \in \mathbb{R}^m$.

Enumeration of derivatives and distinction of super- and subscripts will become relevant in the discussion of higher derivatives computed by combinations of tangent and adjoint modes.

$$F_{(1)} : \mathbb{R}^n \times \mathbb{R}^{n_{\mathbf{x}}} \times \mathbb{R}^m \times \mathbb{R}^{\tilde{n}} \to \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^{m_{\mathbf{y}}} \times \mathbb{R}^{\tilde{m}},$$

$$\begin{pmatrix} \mathbf{z} & \tilde{\mathbf{z}} & \mathbf{y} & \tilde{\mathbf{y}} & \mathbf{x}_{(1)} & \mathbf{z}_{(1)} & \mathbf{y}_{(1)} \end{pmatrix}^T := F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \tilde{\mathbf{x}}, \mathbf{z}, \mathbf{z}_{(1)}, \tilde{\mathbf{z}}, \mathbf{y}_{(1)}),$$

computes a shifted transposed Jacobian $\times$ vector product alongside with the function value:

$$\mathbb{R}^m \times \mathbb{R}^m \ni \begin{pmatrix} \mathbf{z} \\ \tilde{\mathbf{z}} \\ \mathbf{y} \\ \tilde{\mathbf{y}} \end{pmatrix} := F(\mathbf{x}, \tilde{\mathbf{x}}, \mathbf{z}, \tilde{\mathbf{z}})$$

$$\begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} := \begin{pmatrix} \mathbf{x}_{(1)} \\ 0 \end{pmatrix} + \nabla F(\mathbf{x}, \tilde{\mathbf{x}}, \mathbf{z}, \tilde{\mathbf{z}})^T \cdot \begin{pmatrix} \mathbf{z}_{(1)} \\ \mathbf{y}_{(1)} \end{pmatrix}$$

$$\mathbf{y}_{(1)} := 0$$

The whole (dense) Jacobian can be harvested from the active input adjoints

$$\begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} \in \mathbb{R}^m$$

row-wise by seeding active output adjoints

$$\begin{pmatrix} \mathbf{z}_{(1)} \\ \mathbf{y}_{(1)} \end{pmatrix} \in \mathbb{R}^m$$

with the Cartesian basis vectors in $\mathbb{R}^m$ and for $\mathbf{x}_{(1)} := 0$ on input.

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

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

computes a shifted transposed Jacobian $\times$ vector product alongside with the function value:

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

... harvesting of the whole Jacobian row-wise by seeding input directions $\mathbf{y}_{(1)} \in \mathbb{R}^m$ with the Cartesian basis vectors in $\mathbb{R}^m$ and for $\mathbf{x}_{(1)} = 0$ on input.

Define

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

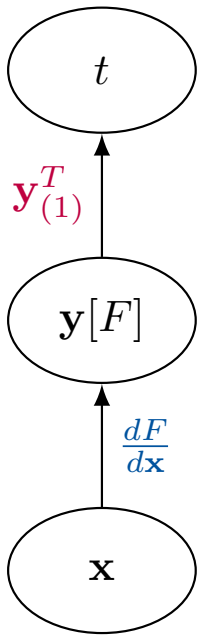
for $\boldsymbol{v} \in \{\mathbf{x}, \mathbf{y}\}$ and some auxiliary $t \in \mathbb{R}$ assuming that $t(F(\mathbf{x}))$ is continuously differentiable over its domain.

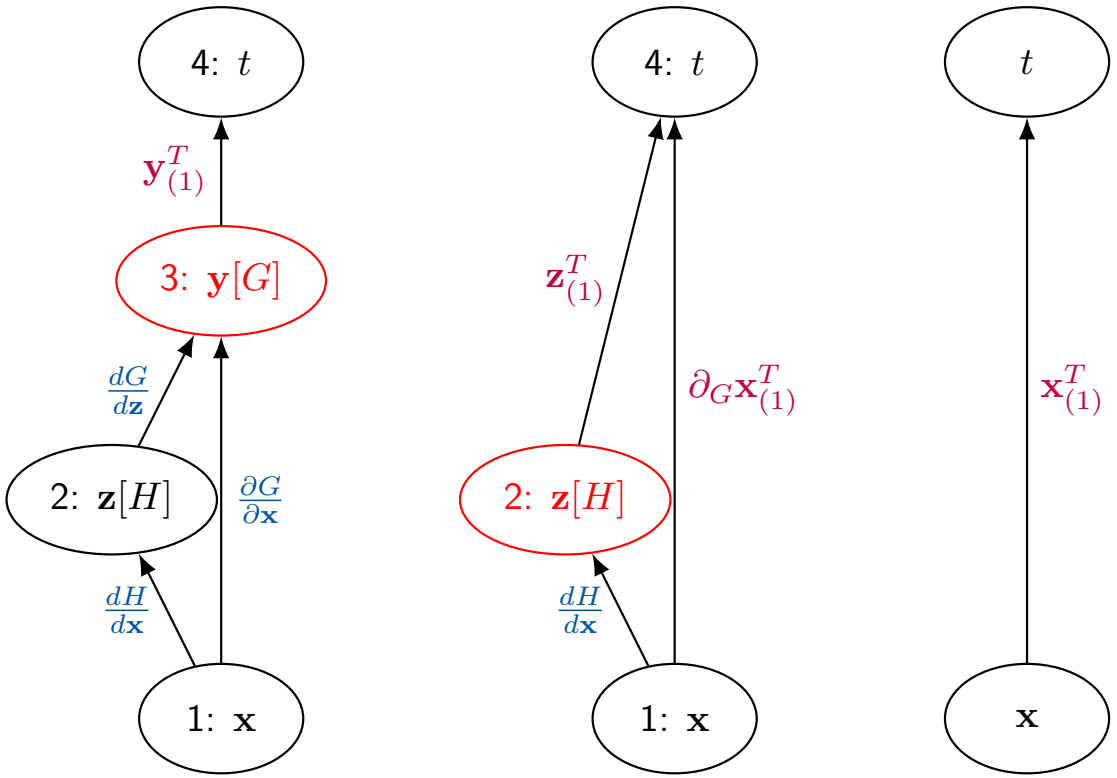
By the chain rule

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

and hence

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





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 (reverses primal data flow).

For $i = 0, \ldots, n-1$

$$v_i := \{x, z\}_i$$

record independent variables (for *harvesting*)

For $i = n, \ldots, q-1$

$$v_i := \varphi_i(v_k)_{k \prec i}$$

record intermediate variables and $d_{j,i} := \dfrac{d\varphi_i(v_k)_{k \prec i}}{dv_j}$ for $j \prec i$

For $i = 0, \ldots, m-1$

$$\{z, y\}_i := v_{n+p+i}$$

record dependent variables (for *seeding*)

For $i = 0, \ldots, m-1$

$$v_{n+p+i_{(1)}} := \{z, y\}_{i_{(1)}} \quad \text{(seed)}$$

For $i = q-1, \ldots, n$

$$\text{for } j \prec i: \; v_{j_{(1)}} := v_{j_{(1)}} + v_{i_{(1)}} \cdot d_{i,j}$$

For $i = 0, \ldots, n-1$

$$\{x, z\}_{i_{(1)}} := \{x_{i_{(1)}}, 0\} + v_{i_{(1)}} \quad \text{(harvest)}$$

→ Note data flow reversal!

1. augmented primal section
   - 1.1 duplicate active data segment
   - 1.2 enable recovery of lost required primal values (e.g, x=sin(x);)
   - 1.3 enable reversal of primal flow of control (e.g, loops and branches)
   - 1.4 run augmented primal sections of subprogram calls
   - 1.5 enable recovery of primal results
2. adjoint section
   - 2.1 recover of lost required primal values
   - 2.2 reverse primal flow of control
   - 2.3 increment adjoints (e.g, y=sin(x); ... z=cos(x);)
   - 2.4 reset adjoints of overwritten primals to zero after use (e.g, y=sin(x); ... z=cos(y);)
   - 2.5 run adjoint sections of subprogram calls
   - 2.6 recover primal results

First-Order Adjoints by Hand

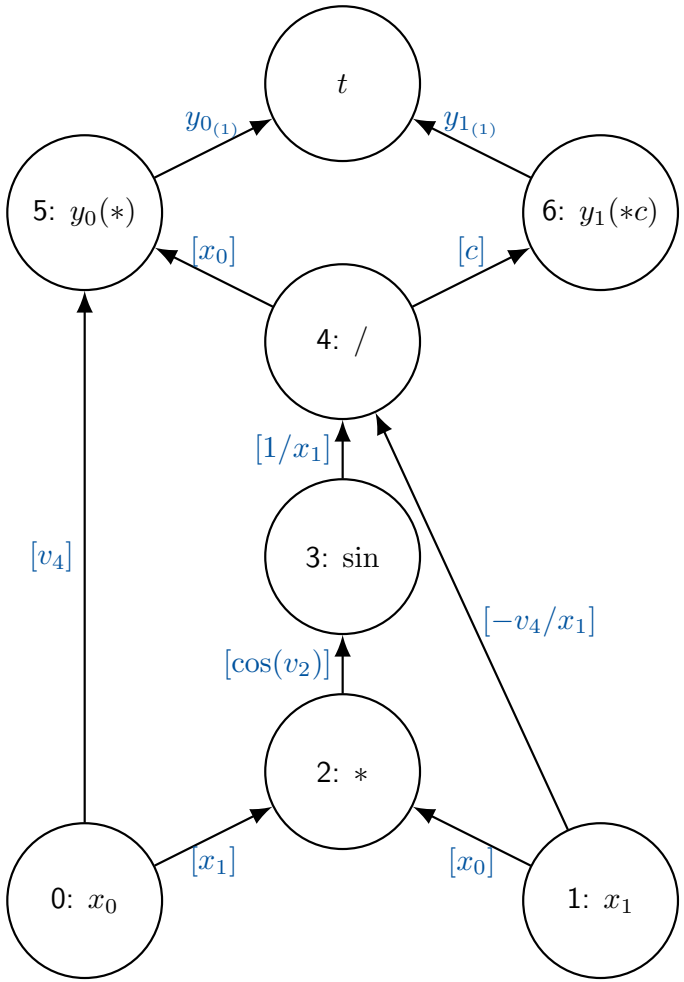
Example: Explicit Euler (Driver)

```
int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);

  const double x0=1, T=1;
  vector<double> p(n,1),pa(n,0);

  double x=x0; double xa=1;
  explicit_euler(n,x,xa,p,pa,T);
  cout << "x=" << x << endl;
  cout << "dx/dx0=" << xa << endl;
  for (int i=0;i<n;i++)
    cout << "dx/dp[" << i << "]=" << pa[i] << endl;
  return 0;
}
```

First-Order Adjoints by Hand

Example: Explicit Euler (Adjoint)

```
void explicit_euler(
  const int n, double& x, double &xa,
  const vector<double>& p, vector<double>& pa,
  const double& T
) {
  stack<double> rd;
  double dt=T/n, t;
  t=0;
  for (int i=0;i<n;i++) {
    rd.push(x);
    x+=dt*p[i]*sin(x*t);
    t+=dt;
  }
  double y=x;
  t=T;
  for (int i=n-1;i>=0;i--) {
    t-=dt;
    x=rd.top(); rd.pop();
    pa[i]+=dt*sin(x*t)*xa;
    xa+=dt*p[i]*t*cos(x*t)*xa;
  }
  x=y;
}
```



Adjoint IDAG

We consider

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 * \sin(x_0 * x_1)/x_1 \\ \sin(x_0 * x_1)/x_1 * c \end{pmatrix}$$

implemented as

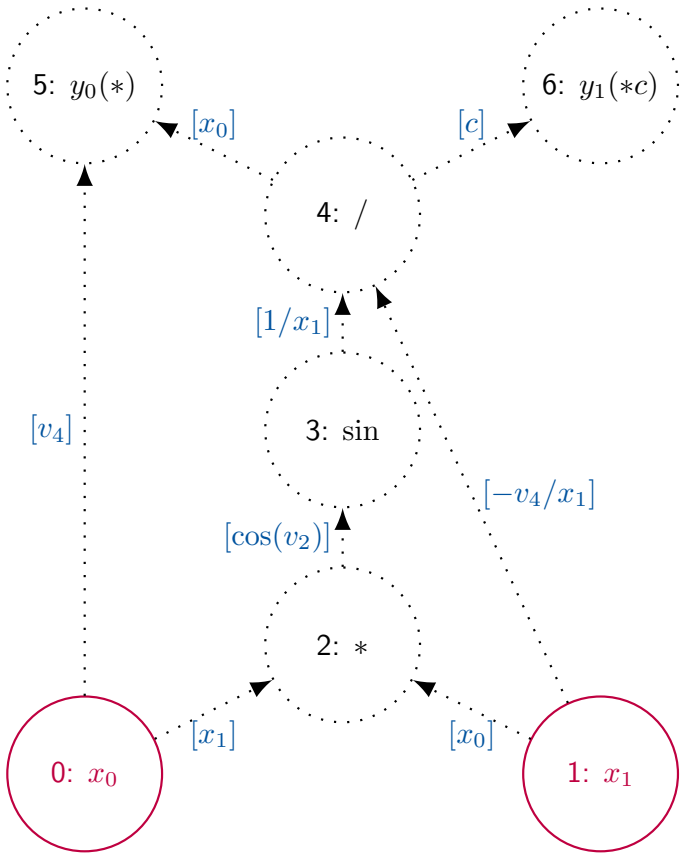
$$t := \sin(x_0 * x_1)/x_1$$
$$y_0 := x_0 * t$$
$$y_1 := t * c$$

yielding SAC

$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3/x_1$$
$$y_0 := x_0 * v_4$$
$$y_1 := v_4 * c$$

for some passive value c.

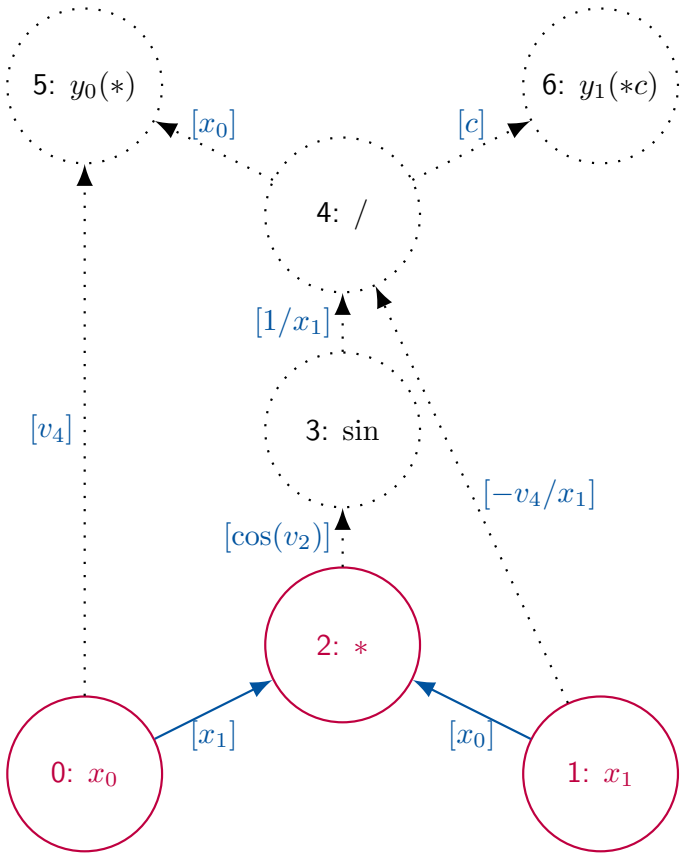
Register (Independent Inputs with Tape)



$$x_0 := ?$$
$$x_1 := ?$$

First-Order Adjoints by Overloading

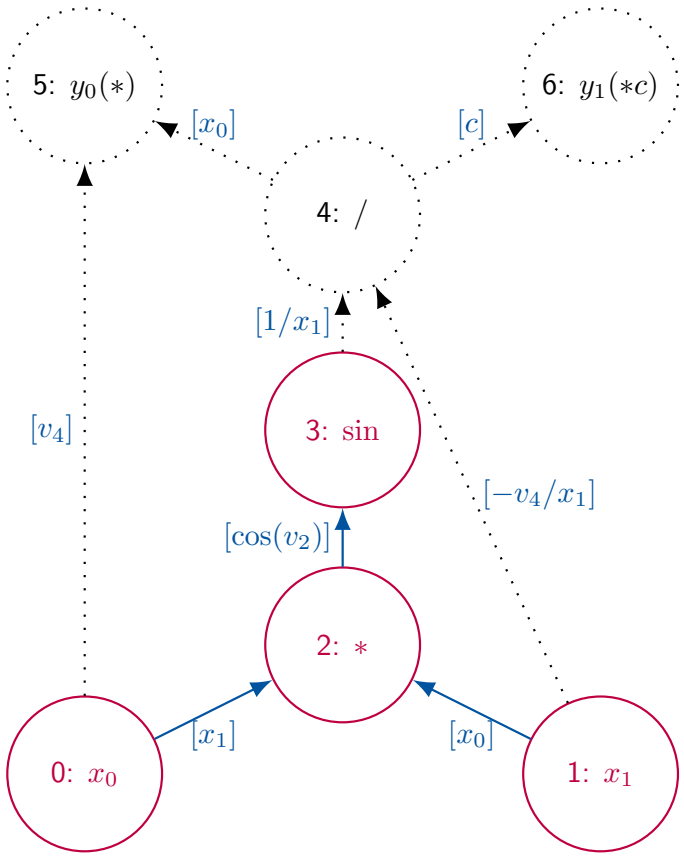
Record (Tape)



$$v_2 := x_0 * x_1$$

First-Order Adjoints by Overloading

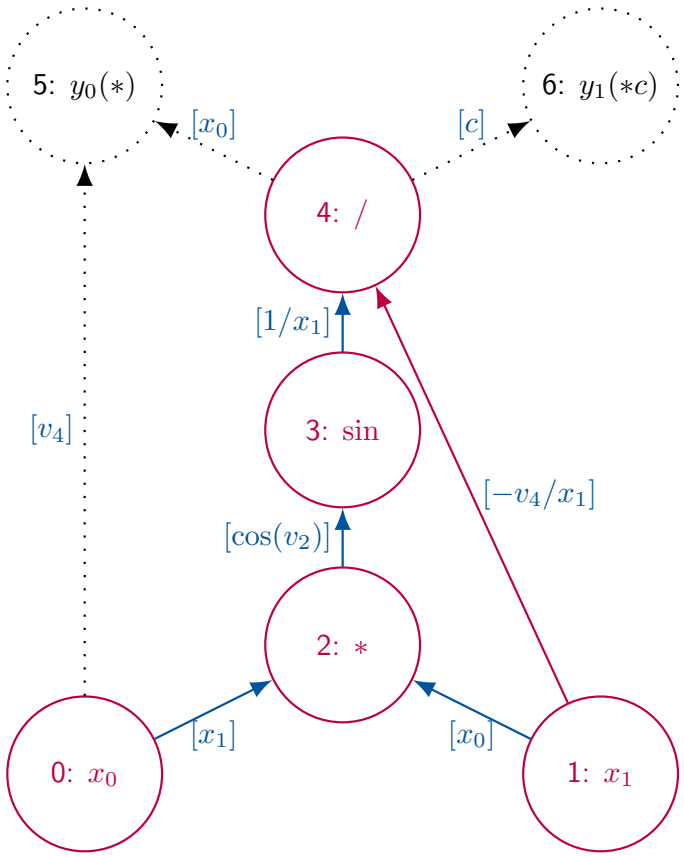
Record (Tape)



$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$

First-Order Adjoints by Overloading

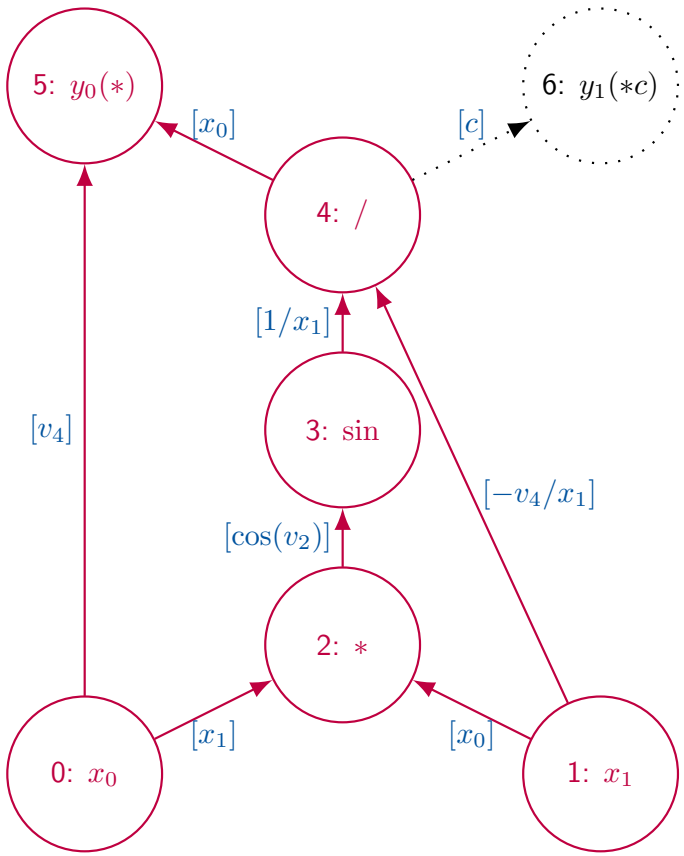
Record (Tape)



$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3/x_1$$

First-Order Adjoints by Overloading

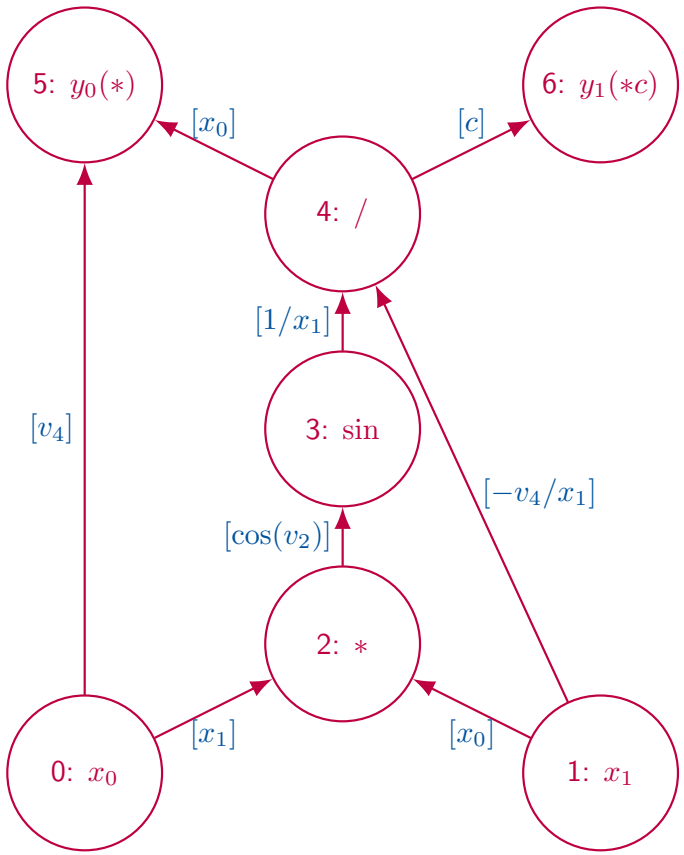
Record (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4
\end{aligned}
$$

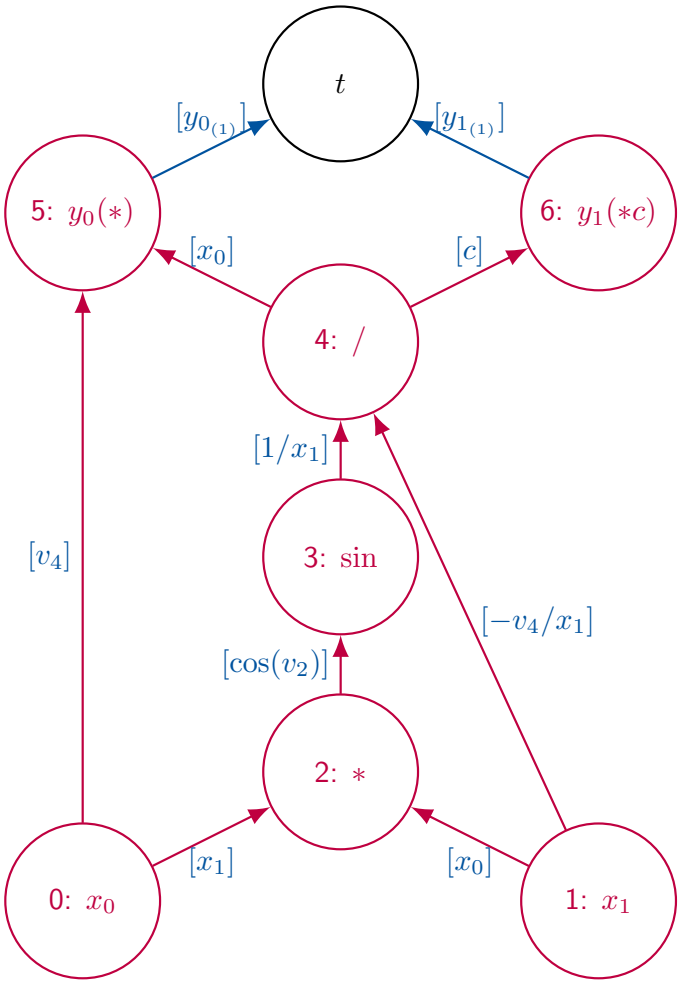
First-Order Adjoints by Overloading

Record (Tape)



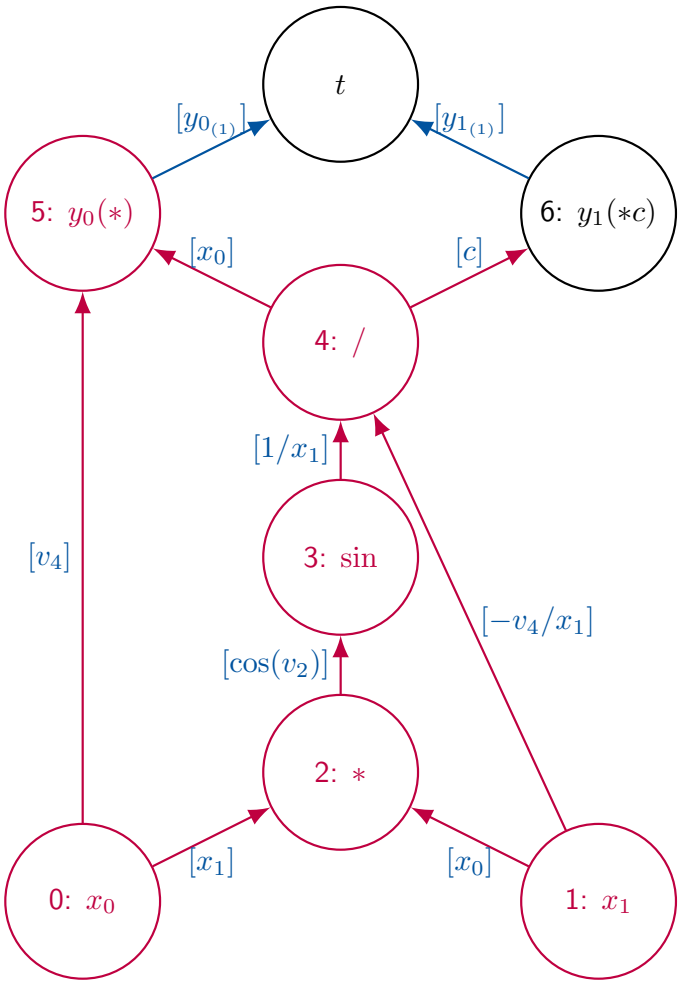
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c
\end{aligned}
$$

Seed



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
y_{0_{(1)}} &:= ? \\
y_{1_{(1)}} &:= ? \\
x_{0_{(1)}} &:= ? \\
x_{1_{(1)}} &:= ? \\
v_{2_{(1)}} &:= 0 \\
v_{3_{(1)}} &:= 0 \\
v_{4_{(1)}} &:= 0
\end{aligned}
$$

Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}}
\end{aligned}
$$

Note C++ Syntax:

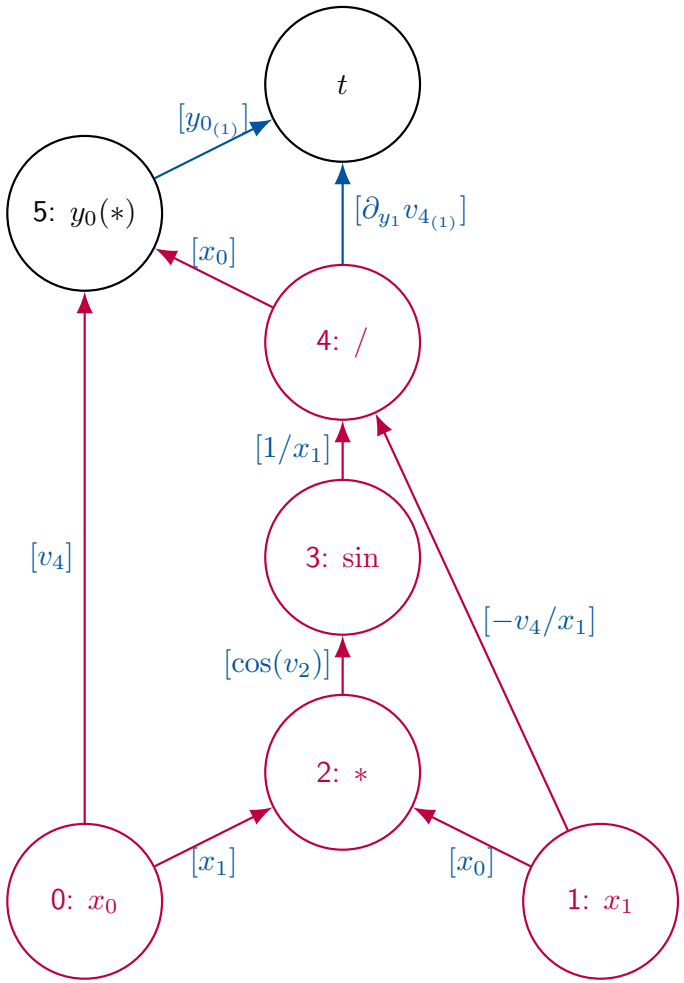
$$v_{4_{(1)}} += c * y_{1_{(1)}}$$

$$\Leftrightarrow$$

$$v_{4_{(1)}} := v_{4_{(1)}} + c * y_{1_{(1)}}.$$

First-Order Adjoints by Overloading

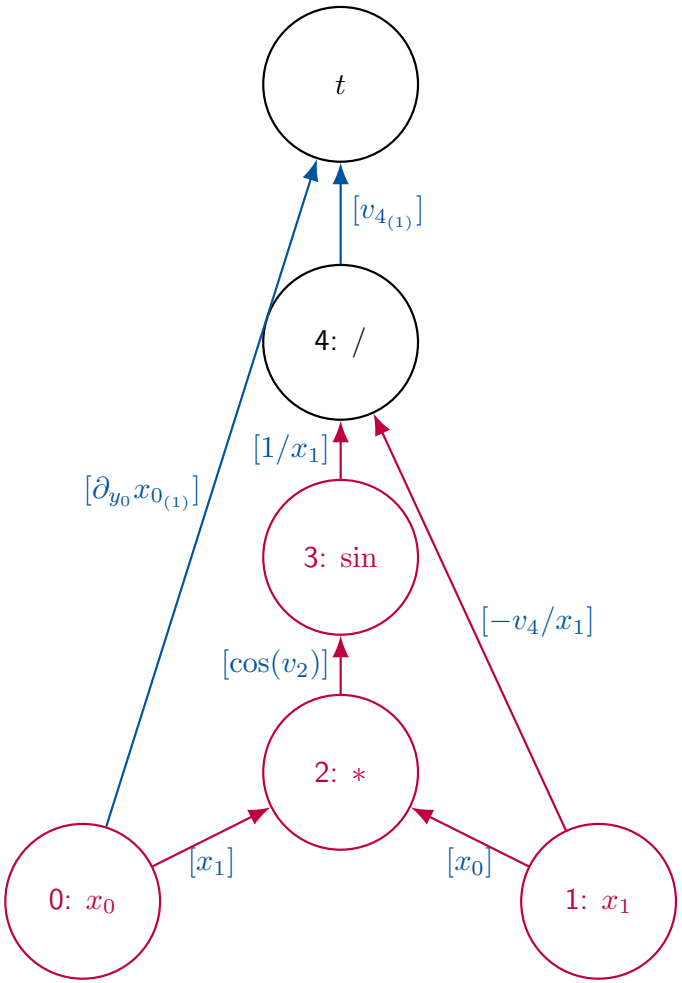
Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}}
\end{aligned}
$$

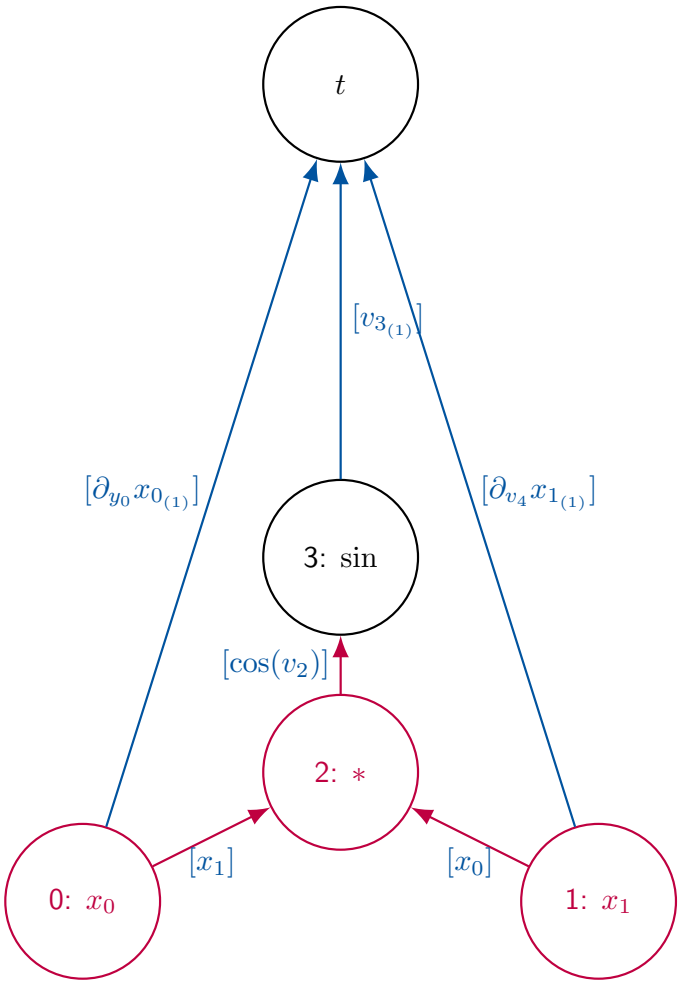
First-Order Adjoints by Overloading

Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
u &:= 1/x_1 \\
v_{3_{(1)}} &+= u * v_{4_{(1)}} \\
x_{1_{(1)}} &-= v_4 * u * v_{4_{(1)}}
\end{aligned}
$$

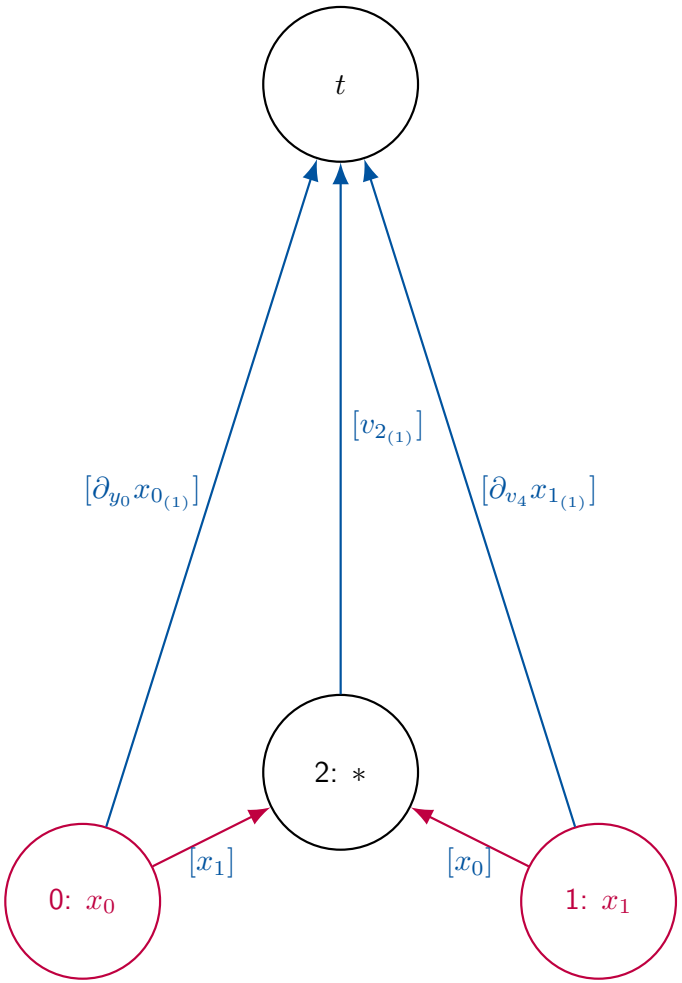
Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
u &:= 1 / x_1 \\
v_{3_{(1)}} &+= u * v_{4_{(1)}} \\
x_{1_{(1)}} &-= v_4 * u * v_{4_{(1)}} \\
v_{2_{(1)}} &+= \cos(x_2) * v_{3_{(1)}}
\end{aligned}
$$

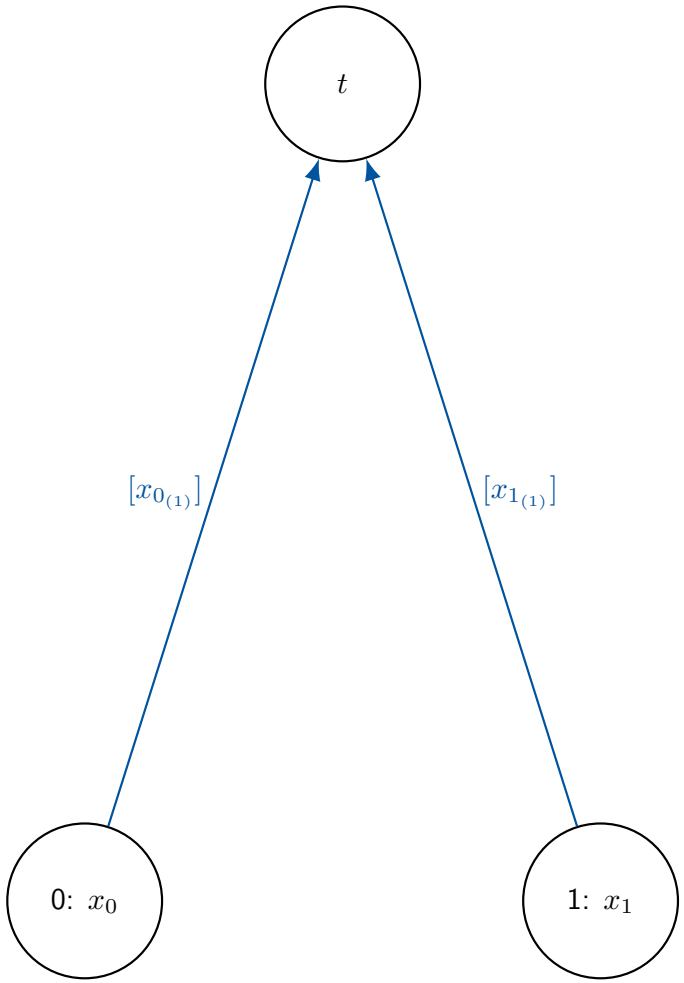
First-Order Adjoints by Overloading

Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
u &:= 1 / x_1 \\
v_{3_{(1)}} &+= u * v_{4_{(1)}} \\
x_{1_{(1)}} &-= v_4 * u * v_{4_{(1)}} \\
v_{2_{(1)}} &+= \cos(x_2) * v_{3_{(1)}} \\
x_{0_{(1)}} &+= x_1 * v_{2_{(1)}} \\
x_{1_{(1)}} &+= x_0 * v_{2_{(1)}}
\end{aligned}
$$

Harvest



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
u &:= 1 / x_1 \\
v_{3_{(1)}} &+= u * v_{4_{(1)}} \\
x_{1_{(1)}} &-= v_4 * u * v_{4_{(1)}} \\
v_{2_{(1)}} &+= \cos(x_2) * v_{3_{(1)}} \\
x_{0_{(1)}} &+= x_1 * v_{2_{(1)}} \\
x_{1_{(1)}} &+= x_0 * v_{2_{(1)}}
\end{aligned}
$$

Type-generic primal

```
template<typename AT, typename PT>
void explicit_euler(const int n, AT& x,
     const vector<AT>& p, const PT& T) {
  PT dt=T/n, t;
  t=0;
  for (int i=0;i<n;i++) {
    x+=dt*p[i]*sin(x*t);
    t+=dt;
  }
}
```

Adjoints by Overloading with dco/c++

Example: Explicit Euler (Augmented Primal Section)

```
#include "dco.hpp"
typedef dco::ga1s<double> DCO_M;
typedef DCO_M::type DCO_T;
typedef DCO_M::tape_t DCO_TAPE_T;

int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);
  const double x0=1, T=1;
  vector<DCO_T> p(n,1);

  DCO_T x=x0;
  DCO_M::global_tape=DCO_TAPE_T::create();
  DCO_M::global_tape->register_variable(x);
  DCO_M::global_tape->register_variable(p);
  DCO_T x_in=x;
  explicit_euler(n,x,p,T);
  size_t PMR=dco::size_of(DCO_M::global_tape);
  cout << "x=" << dco::value(x) << endl;
  ...
```

```
  ...
  dco::derivative(x)=1;
  DCO_M::global_tape->interpret_adjoint();
  cout << "dx/dx0=" << dco::derivative(x_in) << endl;
  for (int i=0;i<n;i++)
    cout << "dx/dp[" << i << "]=" << dco::derivative(p[i]) << endl;
  cout << "PMR=" << PMR/(1024*1024) << "mb" << endl;
  DCO_TAPE_T::remove(DCO_M::global_tape);
  return 0;
}
```

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

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

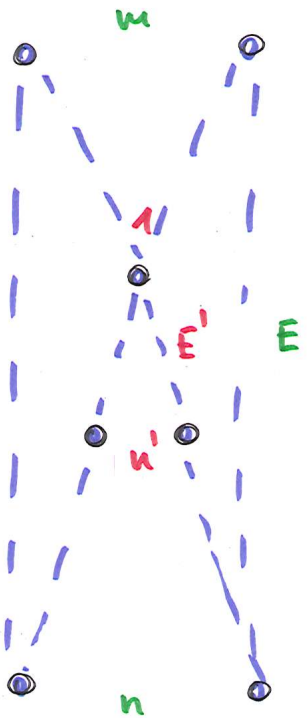
computes a shifted transposed Jacobian $\times$ matrix product alongside with the function value:

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

... harvesting of the whole Jacobian by seeding input matrix $Y_{(1)} \in \mathbb{R}^{m \times m}$ with the identity in $\mathbb{R}^m$ and for $X_{(1)} = 0$ on input.

Optimized First Derivative Code

Statement-Level Preaccumulation of Gradients



A) TANGENT

OPS: $u \cdot |E|$

MEM: $\emptyset$

B) ADJOINT

OPS: $m \cdot |E|$

MEM: $|E|$

C) TANGENT + ADJOINT

OPS: $|E'| + u \cdot (|E| - |E'| + u')$

MEM: $|E'|$

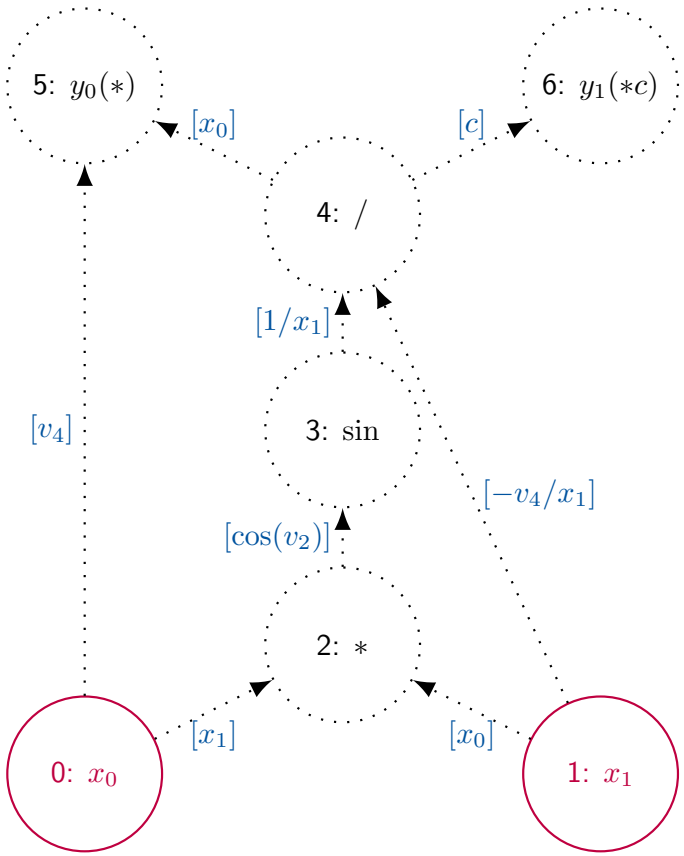
D) ADJOINT + ADJOINT

OPS: $|E'| + m\,(|E| - |E'| + u')$

MEM: $|E| - |E'| + u'$

First-Order Adjoints with dco/c++

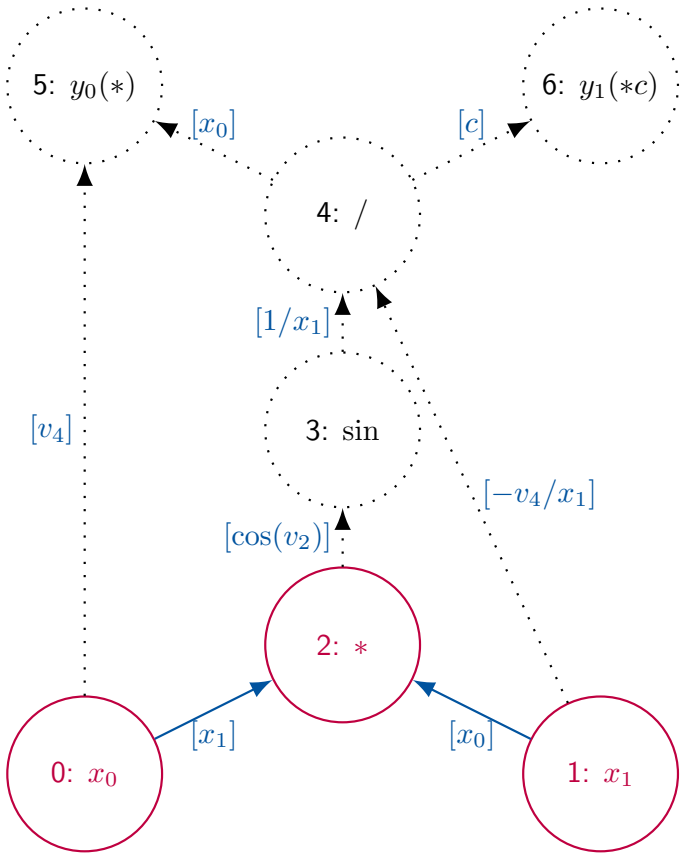
Register (Independent Inputs with Tape)



$$x_0 :=?$$
$$x_1 :=?$$

First-Order Adjoints with dco/c++

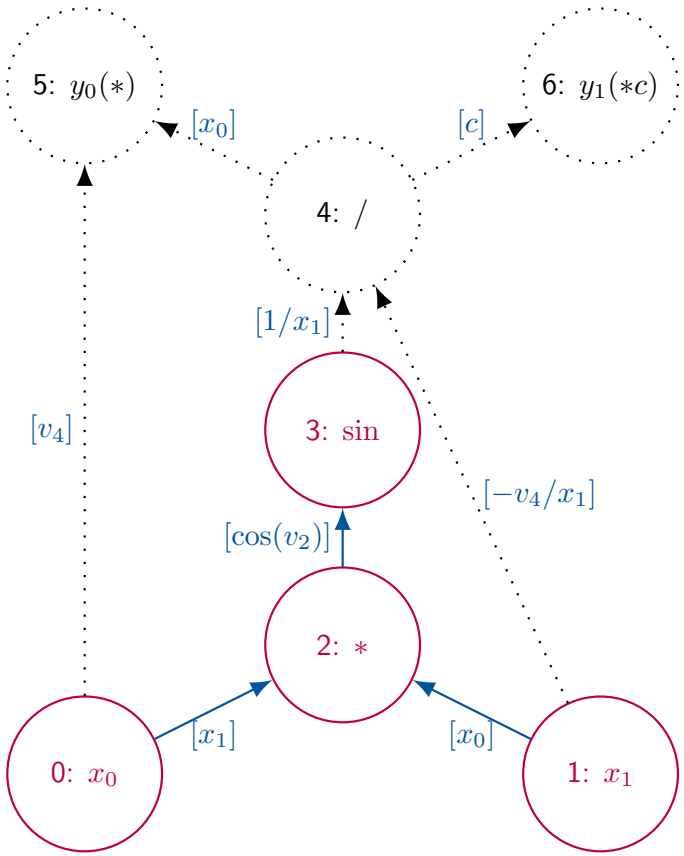
Record (Local Tape)



$$v_2 := x_0 * x_1$$

First-Order Adjoints with dco/c++

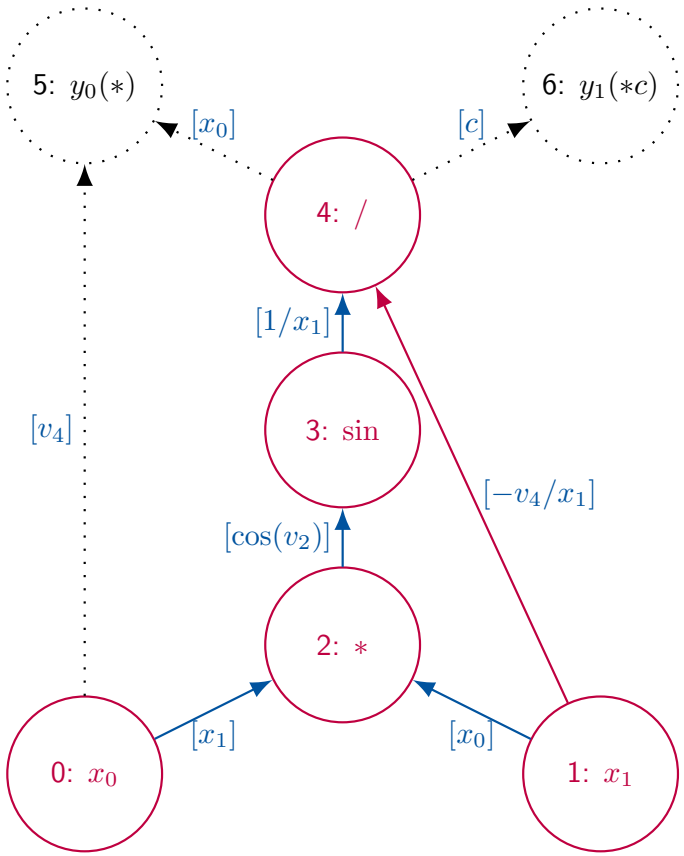
Record (Local Tape)



$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$

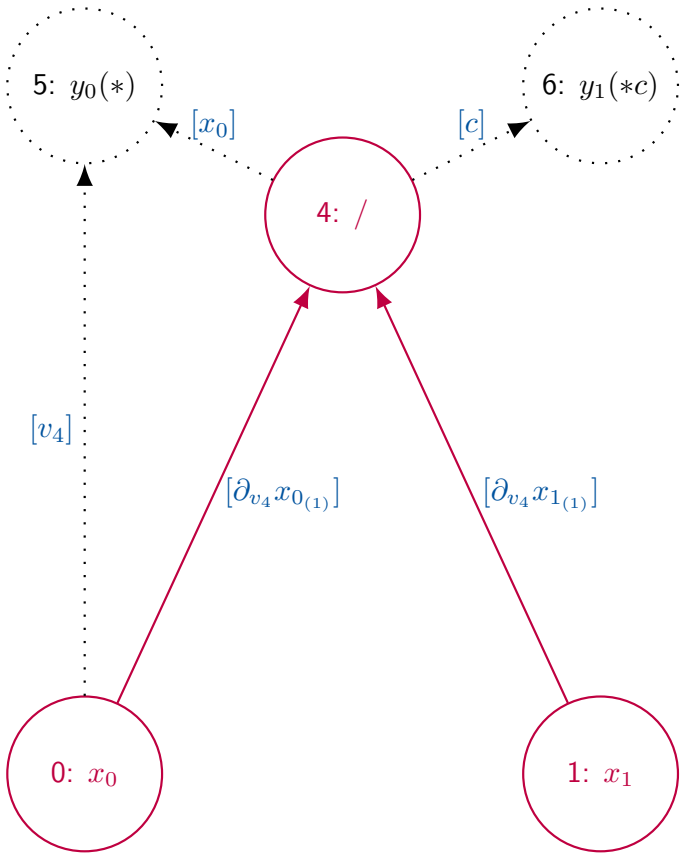
First-Order Adjoints with dco/c++

Record (Local Tape)



$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3/x_1$$

Preaccumulate (Local Gradient)

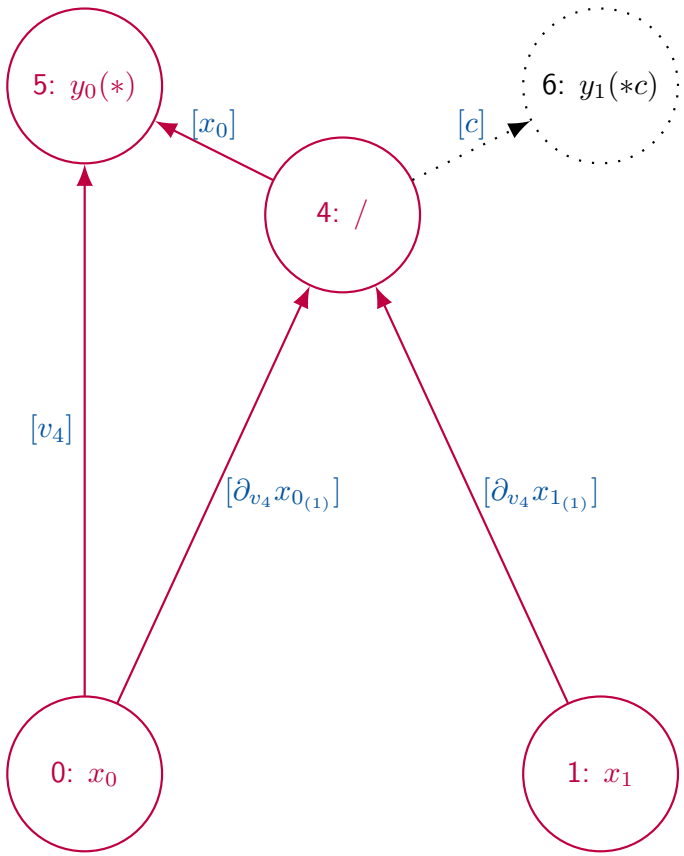


$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0
\end{aligned}
$$

... local gradient code exposed to compiler (optimization)

First-Order Adjoints with dco/c++

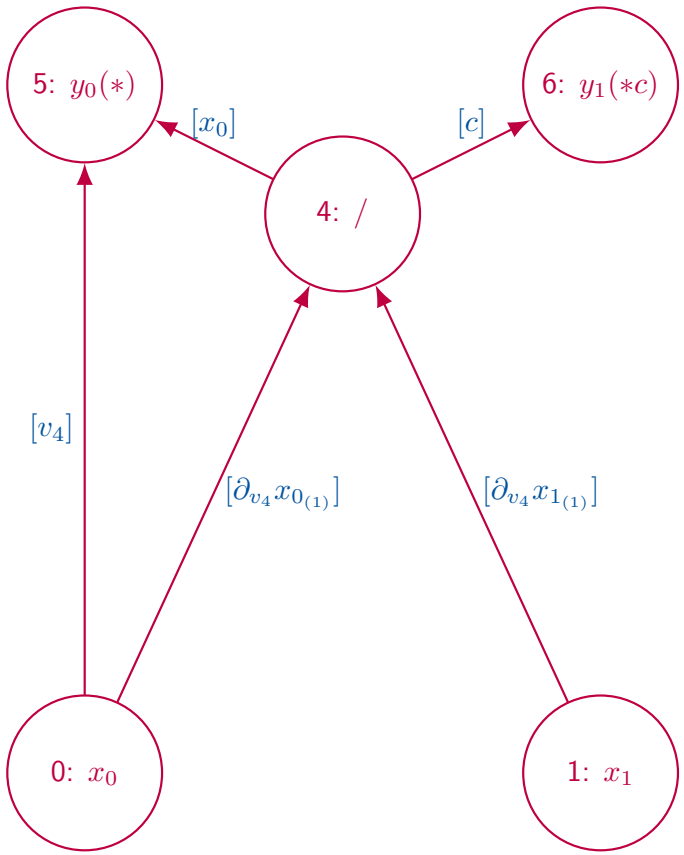
Record (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0 \\
y_0 &:= x_0 * v_4
\end{aligned}
$$

First-Order Adjoints with dco/c++

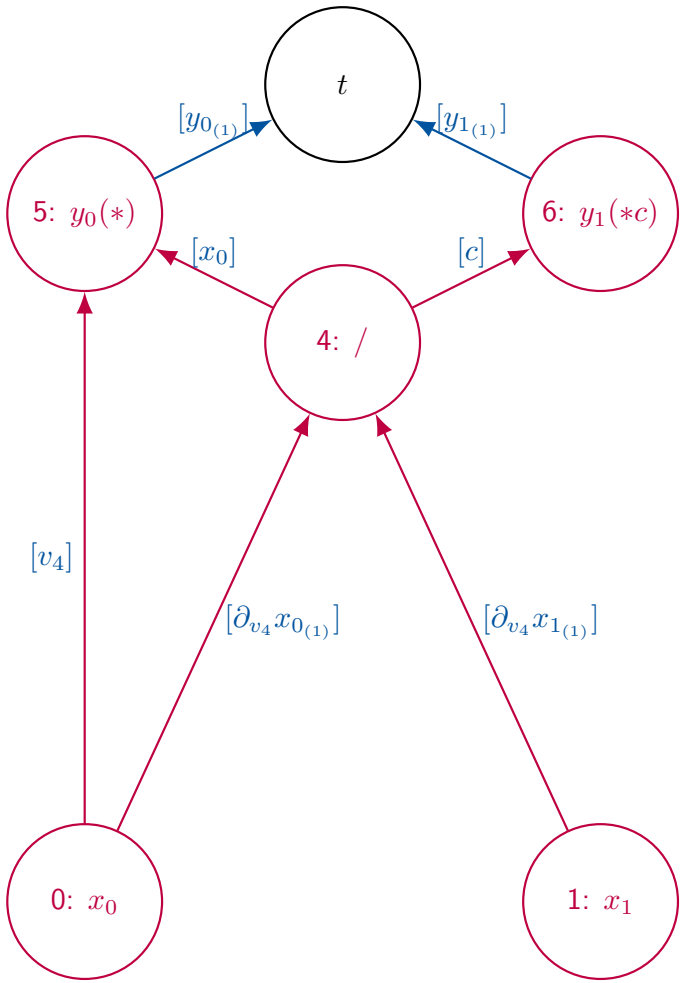
Record (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c
\end{aligned}
$$

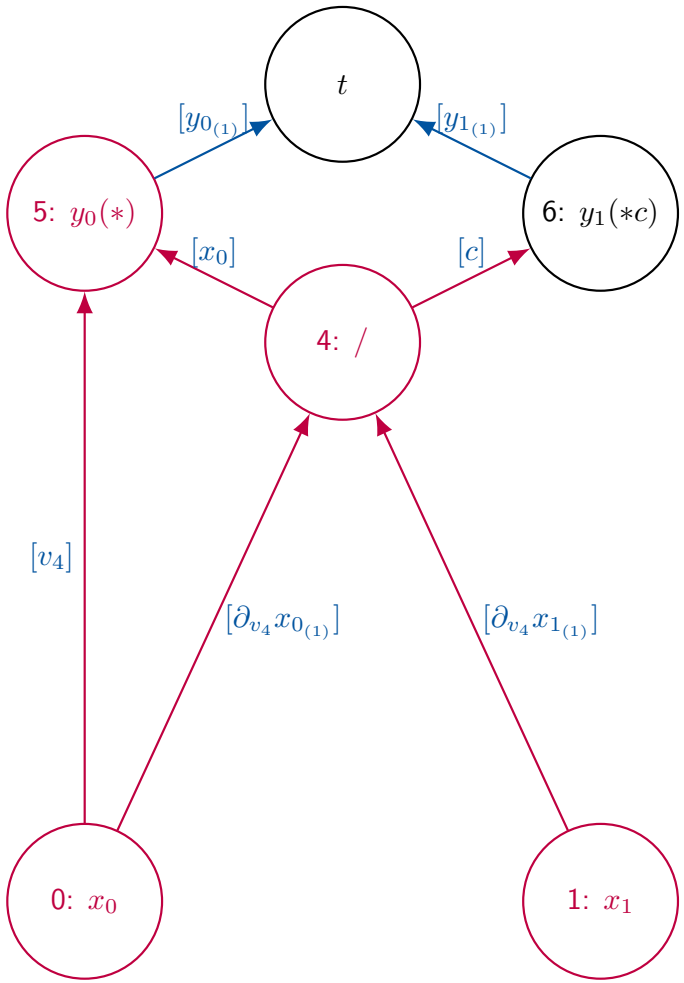
First-Order Adjoints with dco/c++

Seed



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
y_{0_{(1)}} &:=? \\
y_{1_{(1)}} &:=? \\
x_{0_{(1)}} &:=? \\
x_{1_{(1)}} &:=? \\
v_{2_{(1)}} &:= 0 \\
v_{3_{(1)}} &:= 0 \\
v_{4_{(1)}} &:= 0
\end{aligned}
$$

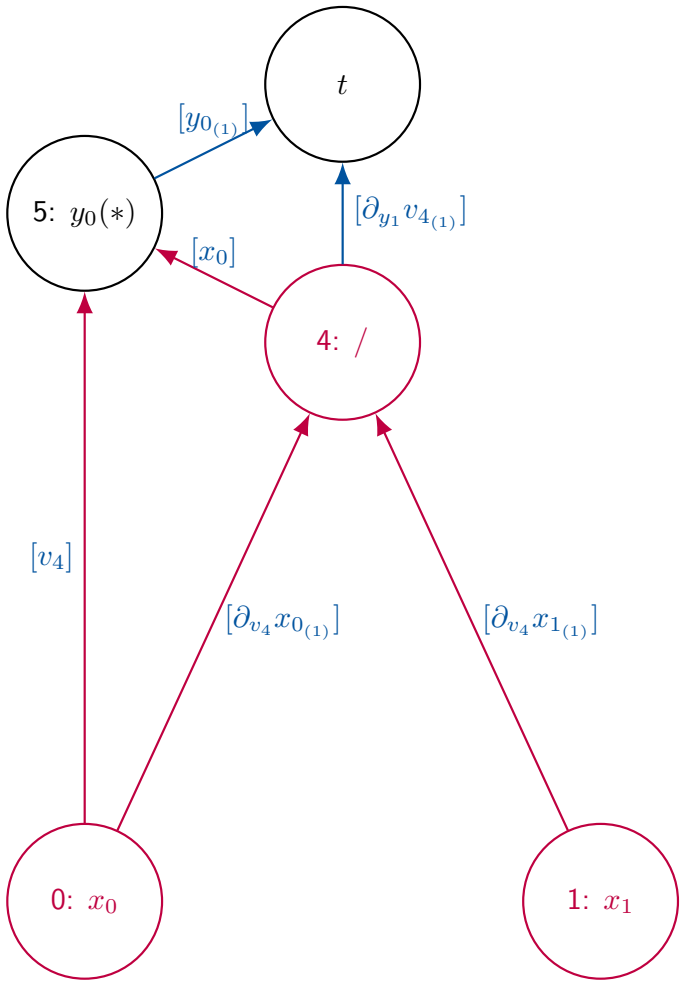
Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v3_{(1)} &:= 1/x_1 * 1 \\
\partial_{v_4} v2_{(1)} &:= \cos(v_2) * \partial_{v_4} v3_{(1)} \\
\partial_{v_4} x1_{(1)} &:= \partial_{v_4} v2_{(1)} * x_1 \\
\partial_{v_4} x0_{(1)} &:= -v_4/x_1 + \partial_{v_4} v2_{(1)} * x_0 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v4_{(1)} &+= c * y1_{(1)}
\end{aligned}
$$

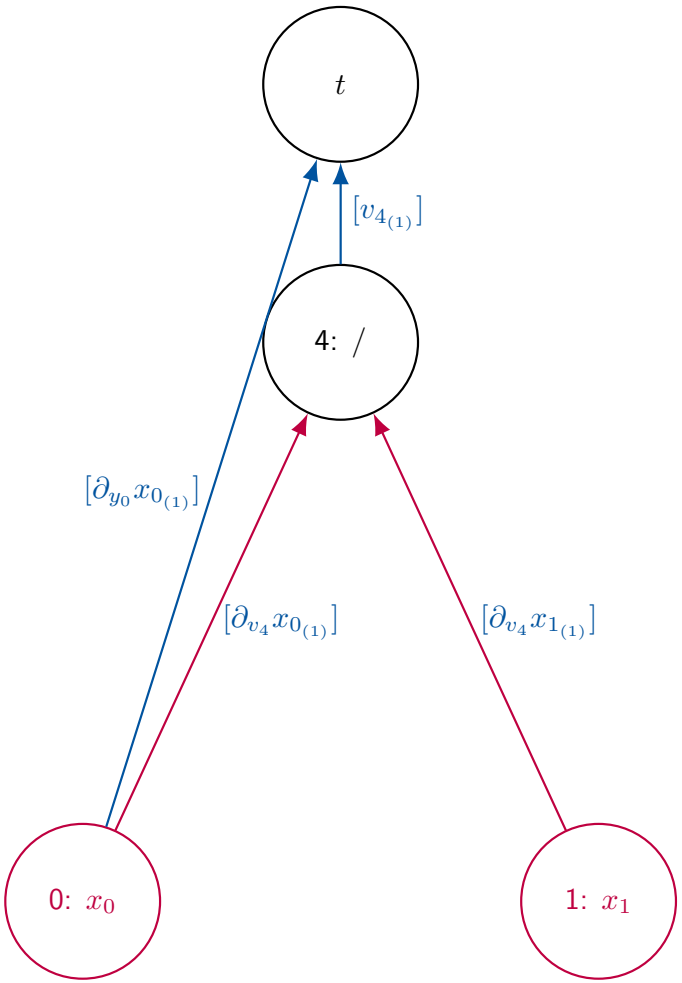
First-Order Adjoints with dco/c++

Interpret (Tape)

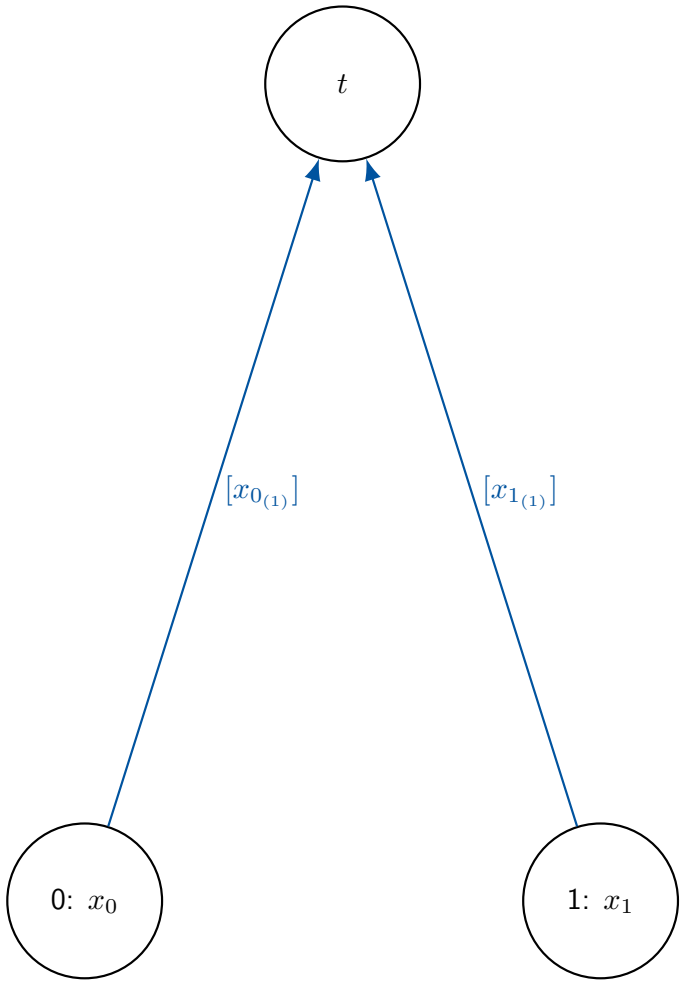


$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}}
\end{aligned}
$$

Interpret (Tape)



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
x_{1_{(1)}} &+= v_{4_{(1)}} * \partial_{v_4} x_{1_{(1)}} \\
x_{0_{(1)}} &+= v_{4_{(1)}} * \partial_{v_4} x_{0_{(1)}}
\end{aligned}
$$



$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
\partial_{v_4} v_{3_{(1)}} &:= 1/x_1 * 1 \\
\partial_{v_4} v_{2_{(1)}} &:= \cos(v_2) * \partial_{v_4} v_{3_{(1)}} \\
\partial_{v_4} x_{1_{(1)}} &:= \partial_{v_4} v_{2_{(1)}} * x_1 \\
\partial_{v_4} x_{0_{(1)}} &:= -v_4/x_1 + \partial_{v_4} v_{2_{(1)}} * x_0 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
x_{1_{(1)}} &+= v_{4_{(1)}} * \partial_{v_4} x_{1_{(1)}} \\
x_{0_{(1)}} &+= v_{4_{(1)}} * \partial_{v_4} x_{0_{(1)}}
\end{aligned}
$$

Practice

Using

```
explicit_euler
   dco
      explicit_euler.h ga1s.cpp
   hand
     adjoint.cpp
```

and sample code from previous lectures

- write adjoint versions of Implicit Euler and Euler-Maruyama codes,
- use dco/c++ to derive adjoint versions of Implicit Euler and Euler-Maruyama codes,
- compare numerical results and run times with finite differences and tangent versions.

Outline

Initially 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

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

For multivariate vector functions the Hessian is a three-tensor complicating the notation slightly due to the need for tensor arithmetic; see later.

Numerical Approximation of Second Derivatives

A second-order *central finite difference* quotient

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

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

$$\frac{d^2 f}{dx_i dx_j}(\mathbf{x}^0) \approx \frac{\frac{df}{dx_i}(\mathbf{x}^0 + \mathbf{e}_j \cdot h) - \frac{df}{dx_i}(\mathbf{x}^0 - \mathbf{e}_j \cdot h)}{2 \cdot h}$$

$$= \left[\frac{f(\mathbf{x}^0 + \mathbf{e}_j \cdot h + \mathbf{e}_i \cdot h) - f(\mathbf{x}^0 + \mathbf{e}_j \cdot h - \mathbf{e}_i \cdot h)}{2 \cdot h} - \frac{f(\mathbf{x}^0 - \mathbf{e}_j \cdot h + \mathbf{e}_i \cdot h) - f(\mathbf{x}^0 - \mathbf{e}_j \cdot h - \mathbf{e}_i \cdot h)}{2 \cdot h} \right] / (2 \cdot h).$$

Numerical Approximation of Second Derivatives

Example: Central Finite Differences I

```
int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);

  const double x0=1, T=1;
  vector<double> p(n,1);

  double x=x0;

  explicit_euler(n,x,p,T);
  cout << "x=" << x << endl;
  double h=sqrt(sqrt(DBL_EPSILON));
  for (int i=0;i<n;i++) {
    for (int j=0;j<=i;j++) {
      double xp2=x0, xp1=x0;
      p[i]+=h; p[j]+=h;
      explicit_euler(n,xp2,p,T);
      p[i]-=2*h;
      explicit_euler(n,xp1,p,T);
      xp2-=xp1; xp1=x0;
```

```
        p[i]+=2*h; p[j]-=2*h;
        explicit_euler(n,xp1,p,T);
        xp2-=xp1; xp1=x0;
        p[i]-=2*h;
        explicit_euler(n,xp1,p,T);
        xp2+=xp1;
        cout << "ddx/dpp[" << i << "][" <<j << "]=" << xp2/(4*h*h) << endl;
      }
    }
    return 0;
}
```

Race against second-order adjoint

n	primal	fd	adjoint
400	~ 0	3.5	0.7
600	~ 0	11.8	1.5
800	~ 0	48.8	2.6

A second derivative code $F^{(1,2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$, generated in Tangent-over-Tangent (ToT) mode computes

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

as follows:

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

Second directional differentiation of

$$\begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(1)} \end{pmatrix}$$

in tangent mode ...

- ... yields for $\frac{dy}{d\mathbf{x}}$ in direction $\mathbf{x}^{(2)}$ the (potentially vanishing; see *essential activity / usefulness*) result $y^{(2)}$;
- ... yields for $\frac{dy}{d\mathbf{x}^{(1)}}$ in direction $\mathbf{x}^{(1)(2)} \equiv \mathbf{x}^{(1,2)}$ a vanishing contribution to $y^{(2)}$; (no dependence of y on $x^{(1)}$)
- ... yields for $\frac{dy^{(1)}}{d\mathbf{x}}$ in direction $\mathbf{x}^{(2)}$ a contribution to $y^{(1)(2)} \equiv y^{(1,2)}$;
- ... yields for $\frac{dy^{(1)}}{d\mathbf{x}^{(1)}}$ in direction $\mathbf{x}^{(1,2)}$ a (potentially vanishing; see *essential activity / variedness*) contribution to $y^{(1,2)}$.

Directional differentiation in tangent mode of the first-order tangent model

$$\begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(1)} \end{pmatrix}$$

in direction $(\mathbf{x}^{(2)}\ \mathbf{x}^{(1,2)})^T$ yields

$$\begin{pmatrix} y^{(2)} \\ y^{(1,2)} \end{pmatrix} \equiv \frac{d\begin{pmatrix} y \\ y^{(1)} \end{pmatrix}}{d(\mathbf{x}\ \mathbf{x}^{(1)})} \cdot \begin{pmatrix} \mathbf{x}^{(2)} \\ \mathbf{x}^{(1,2)} \end{pmatrix} = \begin{pmatrix} \frac{dy}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} + \overbrace{\frac{dy}{d\mathbf{x}^{(1)}} \cdot \mathbf{x}^{(1,2)}}^{=0} \\ \frac{dy^{(1)}}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} + \frac{dy^{(1)}}{d\mathbf{x}^{(1)}} \cdot \mathbf{x}^{(1,2)} \end{pmatrix}$$

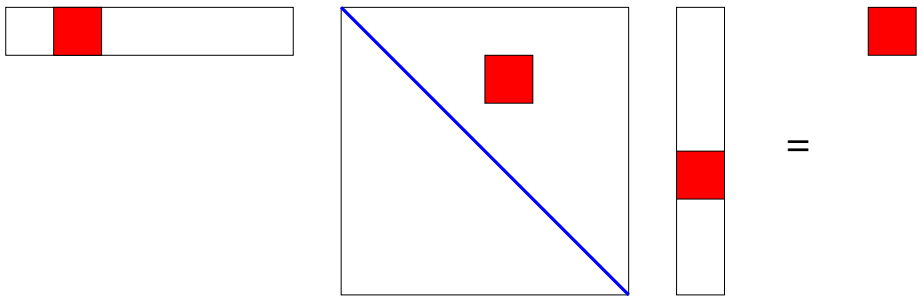
$$\left[y^{(1)} = \mathbf{x}^{(1)\,T} \cdot \frac{dF(\mathbf{x})}{d\mathbf{x}}^T;\ \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2}^T = \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \right] = \begin{pmatrix} \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} \\ \mathbf{x}^{(1)\,T} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(2)} + \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(1,2)} \end{pmatrix}$$

Aiming for second-order derivatives with respect to $\mathbf{x}$ we may consider y as not useful and $\mathbf{x}^{(1)}$ as not varied; i.e, both y and $\mathbf{x}^{(1)}$ are passive with respect to the application of tangent mode AD to the first-order tangent code.

Consequently,

$$y^{(1,2)} \equiv \frac{dy^{(1)}}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} = \mathbf{x}^{(1)^T} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(2)}$$

Graphical illustration is provided by a corresponding reduction of the tangent-augmented tangent IDAG; see below.

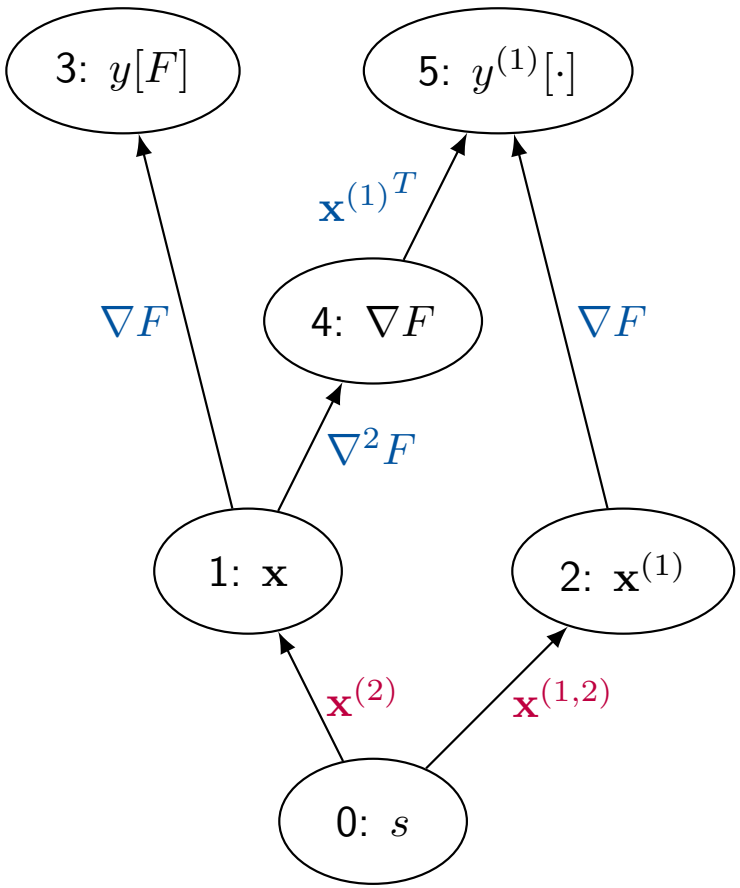


$$\mathbf{x}^{(1)^T} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

... accumulation of the whole Hessian element-wise by *seeding* input directions $\mathbf{x}^{(1)} \in \mathbb{R}^n$ $\mathbf{x}^{(2)} \in \mathbb{R}^n$ independently with the Cartesian basis vectors in $\mathbb{R}^n$ for $\mathbf{x}^{(1,2)} = 0$; harvesting from $y^{(1,2)}$.

Second-Order ToT Model

Graphically on Tangent-Augmented Tangent IDAG



$$
\begin{aligned}
y^{(2)} &\equiv \frac{dy}{ds} \\
&= \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} \\
y^{(1,2)} &\equiv \frac{dy^{(1)}}{ds} \\
&= \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(1,2)} + \mathbf{x}^{(1)^T} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(2)}
\end{aligned}
$$

Comments:

- second-order term requires proof since actually $\frac{dy^{(1)}}{d\nabla F} \in \mathbb{R}^{1\times(1\times n)}$, $\frac{d\nabla F}{d\mathbf{x}} \in \mathbb{R}^{(1\times n)\times n}$, and $\frac{d\mathbf{x}}{ds} \in \mathbb{R}^n$.
- $\mathbf{x}, \mathbf{x}^{(2)} \Rightarrow y^{(2)}$
- $\mathbf{x}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(1,2)} \Rightarrow y^{(1,2)}$
- $\mathbf{x}^{(1)}$ varied? y useful?

Second-Order ToT Code by Hand

Example: Explicit Euler (Driver)

```
int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);

  const double x0=1, T=1;
  vector<double> p(n,1),pt1(n,0),pt2(n,0);

  double x=x0;
  explicit_euler(n,x,p,T);
  cout << "x=" << x << endl;
  for (int i=0;i<n;i++) {
    for (int j=0;j<=i;j++) {
      x=x0; double xt1=0, xt2=0, xtt=0;
      pt1[i]=1; pt2[j]=1;
      explicit_euler(n,x,xt2,xt1,xtt,p,pt2,pt1,T);
      cout << "ddx/dpp[" << i << "][" << j << "]=" << xtt << endl;
      pt1[i]=0; pt2[j]=0;
    }
  }
  return 0;
}
```

Second-Order ToT Code by Hand

Example: Explicit Euler (Second-Order Tangent)

```
void explicit_euler(const int n,
   double& x, double &xt2, double& xt1, double &xtt,
   const vector<double>& p, const vector<double>& pt2,
   const vector<double>& pt1,
   const double& T
) {
   double dt=T/n, t;
   t=0;
   for (int i=0;i<n;i++) {
      xtt+=dt*pt1[i]*t*cos(x*t)*xt2+dt*pt2[i]*t*cos(x*t)*xt1
          -dt*p[i]*t*t*sin(x*t)*xt1*xt2+dt*p[i]*t*cos(x*t)*xtt;
      xt1+=dt*pt1[i]*sin(x*t)+dt*p[i]*t*cos(x*t)*xt1;
      xt2+=dt*pt2[i]*sin(x*t)+dt*p[i]*t*cos(x*t)*xt2;
      x+=dt*p[i]*sin(x*t);
      t+=dt;
   }
}
```

Type-generic primal

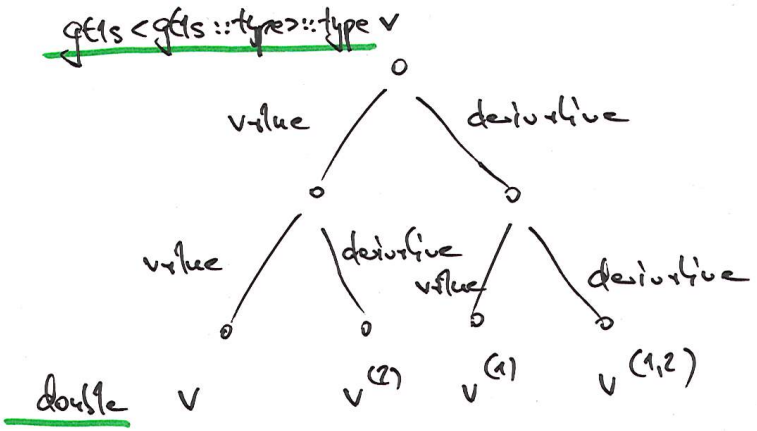
```
template<typename AT, typename PT>
void explicit_euler(const int n, AT& x,
     const vector<AT>& p, const PT& T) {
  PT dt=T/n, t;
  t=0;
  for (int i=0;i<n;i++) {
    x+=dt*p[i]*sin(x*t);
    t+=dt;
  }
}
```

TANGENT OVER TANGENT

$$\begin{pmatrix} y \\ \nabla y \\ \nabla y^{(2)} \\ \nabla y^{(1)} \\ \nabla y^{(1,2)} \end{pmatrix} = F^{(1,2)}\left(x, x^{(1)}, x^{(2)}, x^{(1,2)}\right); \quad y = F(x) : \mathbb{R}^n \to \mathbb{R}$$

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

$$= \langle \nabla F(x), x^{(2)} \rangle + \langle \nabla^2 F(x), x^{(1)} x^{(2)} \rangle$$



Second-Order ToT Code with dco/c++

Example: Explicit Euler (Second-Order Tangent)

```
int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);

  const double x0=1, T=1;
  vector<DCO_T> p(n,1);

  for (int i=0;i<n;i++) {
    for (int j=0;j<=i;j++) {
      dco::derivative(dco::value(p[i]))=1;
      dco::value(dco::derivative(p[j]))=1;
      DCO_T x=x0;
      explicit_euler(n,x,p,T);
      if (i+j==0)
        cout << "x=" << dco::passive_value(x) << endl;
      cout << "ddx/dpp[" << i << "][" << j << "]="
           << dco::derivative(dco::derivative(x)) << endl;
      dco::value(dco::derivative(p[j]))=0;
      dco::derivative(dco::value(p[i]))=0;
    }
  }
  return 0;
}
```

A second derivative code

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

generated in Tangent-over-Adjoint (ToA) mode computes

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

as follows:

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

Second directional differentiation of

$$\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(1)} \end{pmatrix}$$

in tangent mode ...

- ... yields for $\frac{dy}{d\mathbf{x}}$ in direction $\mathbf{x}^{(2)}$ the (potentially vanishing; see *essential activity / usefulness*) result $y^{(2)}$;
- ... yields for $\frac{dy}{dy_{(1)}}$ in direction $y_{(1)}^{(2)}$ a vanishing contribution to $y^{(2)}$; (no dependence of y on $y_{(1)}$)
- ... yields for $\frac{d\mathbf{x}_{(1)}}{d\mathbf{x}}$ in direction $\mathbf{x}^{(2)}$ a contribution to $\mathbf{x}_{(1)}^{(2)}$;
- ... yields for $\frac{d\mathbf{x}_{(1)}}{dy_{(1)}}$ in direction $y_{(1)}^{(2)}$ a (potentially vanishing; see *essential activity / variedness*) contribution to $\mathbf{x}_{(1)}^{(2)}$.

Directional differentiation in tangent mode of the first-order adjoint model

$$\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(1)} \end{pmatrix}$$

in direction $(\mathbf{x}^{(2)} \; y_{(1)}^{(2)})^T$ yields

$$\begin{pmatrix} y^{(2)} \\ \mathbf{x}_{(1)}^{(2)} \end{pmatrix} \equiv \frac{d\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix}}{d(\mathbf{x} \; y_{(1)})} \cdot \begin{pmatrix} \mathbf{x}^{(2)} \\ y_{(1)}^{(2)} \end{pmatrix} = \begin{pmatrix} \frac{dy}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} + \overbrace{\frac{dy}{dy_{(1)}}}^{=0} \cdot y_{(1)}^{(2)} \\ \frac{d\mathbf{x}_{(1)}}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} + \frac{d\mathbf{x}_{(1)}}{dy_{(1)}} \cdot y_{(1)}^{(2)} \end{pmatrix}$$

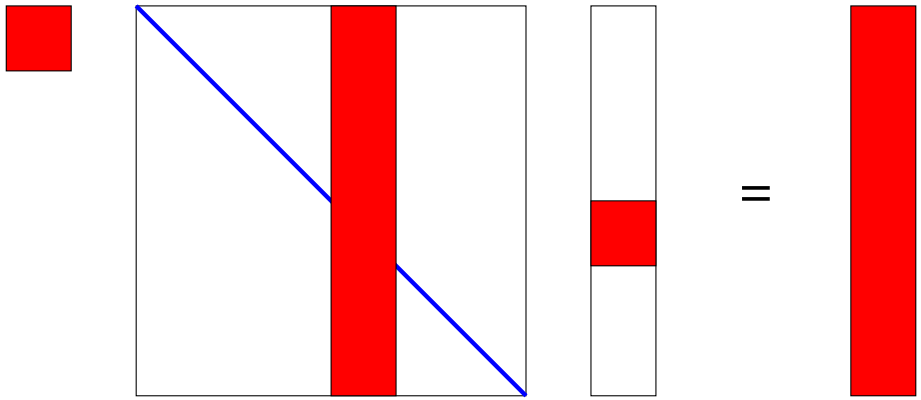
$$\left[\mathbf{x}_{(1)} = y_{(1)} \cdot \frac{dF(\mathbf{x})}{d\mathbf{x}}^T ; \; \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2}^T = \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2}\right] = \begin{pmatrix} \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} \\ y_{(1)} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(2)} + \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(1)}^{(2)} \end{pmatrix}$$

Aiming for second-order derivatives with respect to $\mathbf{x}$ we may consider y as not useful and $y_{(1)}$ as not varied; i.e, both y and $y_{(1)}$ are passive with respect to the application of tangent mode AD to the first-order adjoint code.

Consequently,

$$\mathbf{x}_{(1)}^{(2)} \equiv \frac{d\mathbf{x}_{(1)}}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} = y_{(1)} \cdot \frac{d^2 F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(2)}$$

Graphical illustration is provided by a corresponding reduction of the tangent-augmented adjoint IDAG; see below.

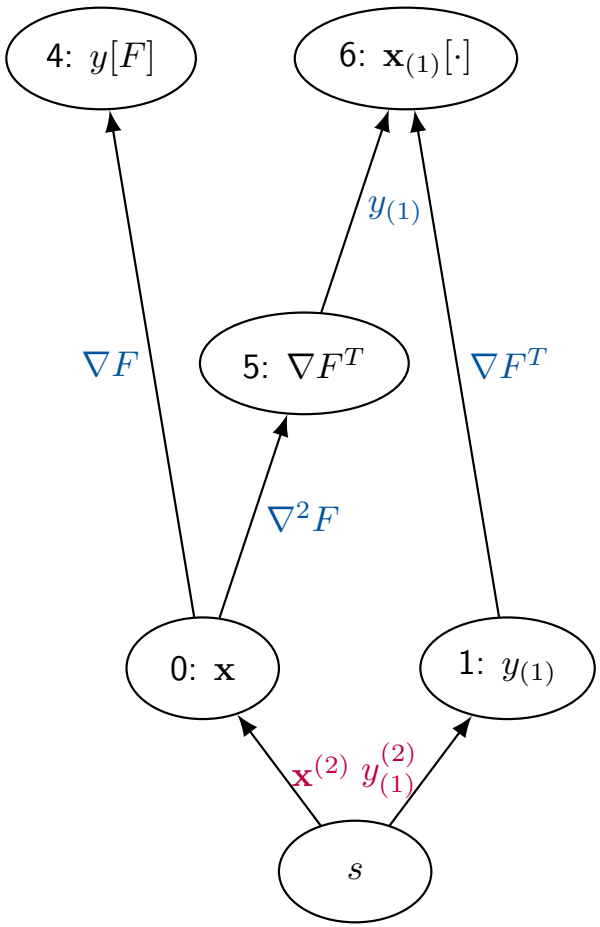


$$y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

... accumulation of the whole Hessian column-wise by seeding input directions $\mathbf{x}^{(2)} \in \mathbb{R}^n$ with the Cartesian basis vectors in $\mathbb{R}^n$ for $y_{(1)} = 1$ and $y_{(1)}^{(2)} = 0$; harvesting from $\mathbf{x}_{(1)}^{(2)}$.

Second-Order ToA Model

Graphically on Tangent-Augmented Adjoint IDAG



$$y^{(2)} \equiv \frac{dy}{ds}$$
$$= \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(2)}$$
$$\mathbf{x}^{(2)}_{(1)} \equiv \frac{d\mathbf{x}_{(1)}}{ds}$$
$$= y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)} + \nabla F(\mathbf{x})^T \cdot y^{(2)}_{(1)}$$

Comments:

- second-order term requires proof since actually $\frac{d\mathbf{x}_{(1)}}{d\nabla F} \in \mathbb{R}^{n\times(1\times n)}$, $\frac{d\nabla F}{d\mathbf{x}} \in \mathbb{R}^{(1\times n)\times n}$, and $\frac{d\mathbf{x}}{ds} \in \mathbb{R}^n$.
- $\mathbf{x}, \mathbf{x}^{(2)} \Rightarrow y^{(2)}$
- $\mathbf{x}, y_{(1)}, \mathbf{x}^{(2)}, y^{(2)}_{(1)} \Rightarrow \mathbf{x}^{(2)}_{(1)}$
- $y_{(1)}$ varied? y useful?

Second-Order ToA Code by Hand

Example: Explicit Euler (Driver)

```
int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);
  const double x0=1, T=1;
  for (int j=0;j<n;j++) {
    vector<double> p(n,1),pt(n,0),pa(n,0),pat(n,0);
    double x=x0, xt=0, xa=1, xat=0;
    pt[j]=1;
    explicit_euler(n,x,xt,xa,xat,p,pt,pa,pat,T);
    if (j==0) cout << "x=" << x << endl;
    for (int i=0;i<=j;i++)
      cout << "ddx/dpp[" << j << "][" << i << "]=" << pat[i] << endl;
  }
  return 0;
}
```

Second-Order ToA Code by Hand

Example: Explicit Euler (Second-Order Adjoint) I

```
void explicit_euler(
  const int n, double& x, double& xt, double &xa, double &xat,
  const vector<double>& p, const vector<double>& pt,
  vector<double>& pa, vector<double>& pat,
  const double& T
) {
  stack<double> rd,rdt;
  double dt=T/n, t;
  t=0;
  for (int i=0;i<n;i++) {
    rdt.push(xt); rd.push(x);
    xt+=dt*pt[i]*sin(x*t)+dt*p[i]*t*cos(x*t)*xt;
    x+=dt*p[i]*sin(x*t);
    t+=dt;
  }
  double yt=xt, y=x;
  t=T;
  for (int i=n-1;i>=0;i--) {
    t-=dt;
    xt=rdt.top(); rdt.pop();
```

Second-Order ToA Code by Hand

Example: Explicit Euler (Second-Order Adjoint) II

```
      x=rd.top(); rd.pop();
      pat[i]+=dt*sin(x*t)*xat+dt*t*cos(x*t)*xa*xt;
      pa[i]+=dt*sin(x*t)*xa;
      xat+=dt*pt[i]*t*cos(x*t)*xa-dt*p[i]*t*t*sin(x*t)*xa*xt+dt*p[i]*t*cos(x*t)*xat;
      xa+=dt*p[i]*t*cos(x*t)*xa;
    }
    xt=yt; x=y;
}
```

Type-generic primal

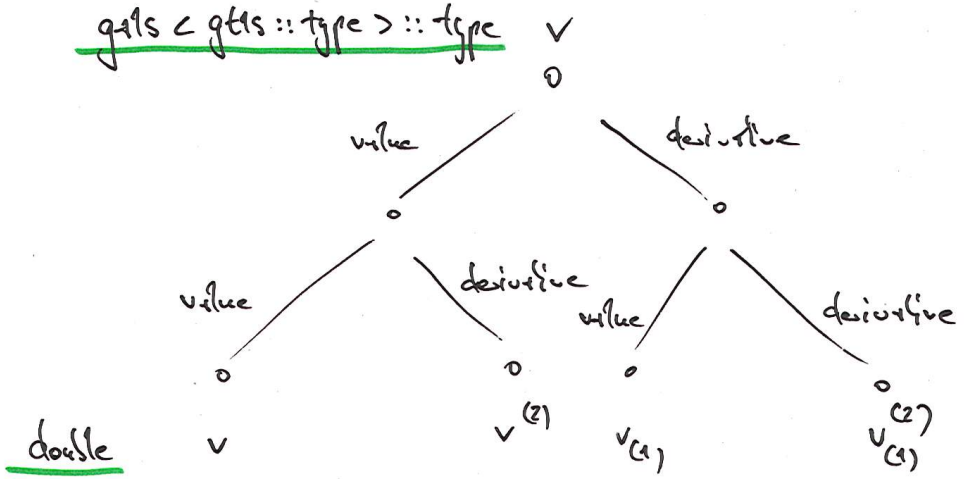
```
template<typename AT, typename PT>
void explicit_euler(const int n, AT& x,
     const vector<AT>& p, const PT& T) {
  PT dt=T/n, t;
  t=0;
  for (int i=0;i<n;i++) {
    x+=dt*p[i]*sin(x*t);
    t+=dt;
  }
}
```

$y = F(x) : \mathbb{R}^4 \to \mathbb{R}$

TANGENT OVER ADJOINT

$$\begin{pmatrix} \text{dco}^{(2)} \\ x_{(1)} \\ x^{(2)}_{(1)} \end{pmatrix} = F^{(2)}_{(1)}\left(x, x^{(2)}, y_{(1)}, y^{(2)}_{(1)}\right) \quad ; \quad x^{(2)}_{(1)} = y_{(1)} \cdot \nabla^2 F(x) \cdot x^{(2)} + \nabla F(x)^T \cdot y^{(2)}_{(1)}$$

$$= \langle y_{(1)}, \nabla^2 F(x), x^{(2)} \rangle + \langle y^{(2)}_{(1)}, \nabla F(x) \rangle$$



Second-Order ToA Code with dco/c++

Example: Explicit Euler (Second-Order Adjoint) I

```
int main(int c, char* v[]) {
  assert(c==2);
  int n=atoi(v[1]);

  const double x0=1, T=1;
  vector<DCO_T> p(n,1);

  DCO_M::global_tape=DCO_TAPE_T::create();
  DCO_M::global_tape->register_variable(p);
  DCO_TAPE_POSITION_T tpos=DCO_M::global_tape->get_position();
  for (int i=0;i<n;i++) {
    dco::derivative(dco::value(p[i]))=1;
    DCO_T x=x0;
    explicit_euler(n,x,p,T);
    if (i==0)
      cout << "x=" << dco::passive_value(x) << endl;
    dco::value(dco::derivative(x))=1;
    DCO_M::global_tape->interpret_adjoint_and_reset_to(tpos);
    for (int j=0;j<=i;j++)
      cout << "ddx/dpp[" << i << "][" << j << "]="
```

```
            << dco::derivative(dco::derivative(p[j])) << endl;
      for (int j=0;j<n;j++) {
        dco::derivative(dco::derivative(p[j]))=0;
        dco::value(dco::derivative(p[j]))=0;
      }
      dco::derivative(dco::value(p[i]))=0;
    }
    DCO_TAPE_T::remove(DCO_M::global_tape);
    return 0;
}
```

A second derivative code

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

generated in Adjoint-over-Tangent (AoT) mode computes

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

as follows:

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

Second directional differentiation of

$$\begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(1)} \end{pmatrix}$$

in adjoint mode ...

- ... yields for $\frac{dy}{d\mathbf{x}^{(1)}}$ in direction $y_{(2)}$ a vanishing contribution to $\mathbf{x}^{(1)}_{(2)}$; (no dependence of y on $\mathbf{x}^{(1)}$)
- ... yields for $\frac{dy}{d\mathbf{x}}$ in direction $y_{(2)}$ the (potentially vanishing; see *essential activity / usefulness*) result $\mathbf{x}_{(2)}$;
- ... yields for $\frac{dy^{(1)}}{d\mathbf{x}}$ in direction $y_{(2)}$ a contribution to $\mathbf{x}_{(2)}$;
- ... yields for $\frac{dy^{(1)}}{d\mathbf{x}^{(1)}}$ in direction $y^{(1)}_{(2)}$ a (potentially vanishing; see *essential activity / variedness*) contribution to $\mathbf{x}^{(1)}_{(2)}$.

Directional differentiation in adjoint mode of the first-order tangent model

$$\begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(1)} \end{pmatrix}$$

in direction $(y_{(2)}\ y_{(2)}^{(1)})^T$ yields

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{x}_{(2)}^{(1)} \end{pmatrix} \equiv \frac{d\begin{pmatrix} y \\ y^{(1)} \end{pmatrix}}{d(\mathbf{x}\ \mathbf{x}^{(1)})}^T \cdot \begin{pmatrix} y_{(2)} \\ y_{(2)}^{(1)} \end{pmatrix} = \begin{pmatrix} \frac{dy}{d\mathbf{x}}^T \cdot y_{(2)} + \frac{dy^{(1)}}{d\mathbf{x}}^T \cdot y_{(2)}^{(1)} \\ \underbrace{\frac{dy}{d\mathbf{x}^{(1)}}^T \cdot y_{(2)}}_{=0} + \frac{dy^{(1)}}{d\mathbf{x}^{(1)}}^T \cdot y_{(2)}^{(1)} \end{pmatrix}$$

$$\left[\frac{dy^{(1)}}{d\mathbf{x}} = \mathbf{x}^{(1)T} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2};\ \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2}^T = \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2}\right] \begin{pmatrix} \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(2)} + y_{(2)}^{(1)} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(1)} \\ \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(2)}^{(1)} \end{pmatrix}$$
$$=$$

Aiming for second-order derivatives with respect to $\mathbf{x}$ we may consider y as not useful and $\mathbf{x}^{(1)}$ as not varied; i.e, both y and $\mathbf{x}^{(1)}$ are passive with respect to the application of adjoint mode AD to the first-order tangent code.

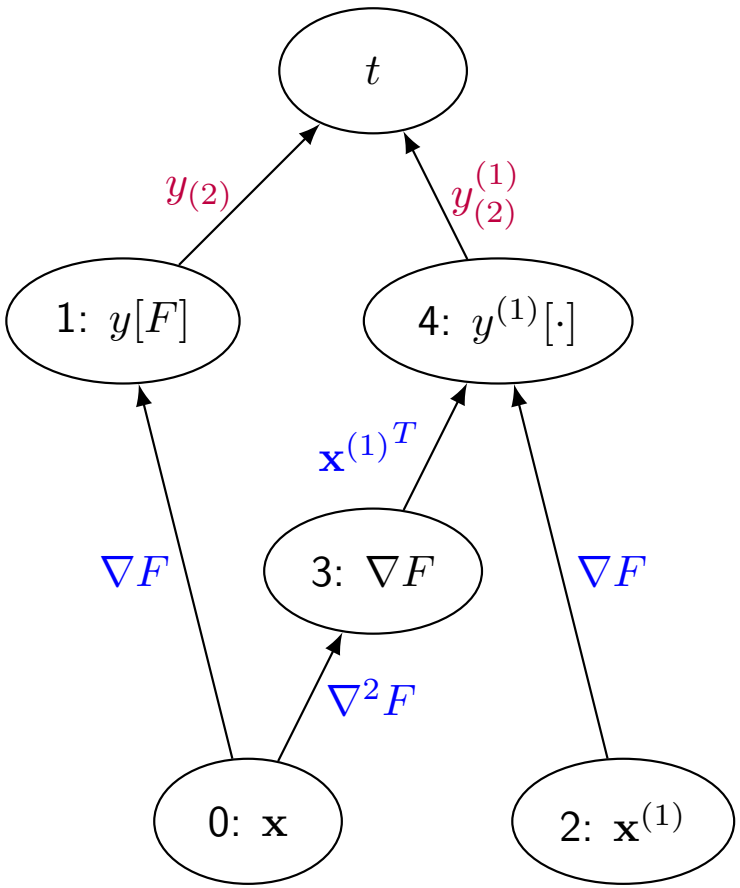
Consequently,

$$\mathbf{x}_{(2)} \equiv \frac{dy^{(1)}}{d\mathbf{x}}^T \cdot y_{(2)}^{(1)} = y_{(2)}^{(1)} \cdot \frac{d^2 F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}^{(1)}$$

Graphical illustration is provided by a corresponding reduction of the adjoint-augmented tangent IDAG; see below.

Second-Order AoT Model

Graphically on Adjoint-Augmented Tangent IDAG

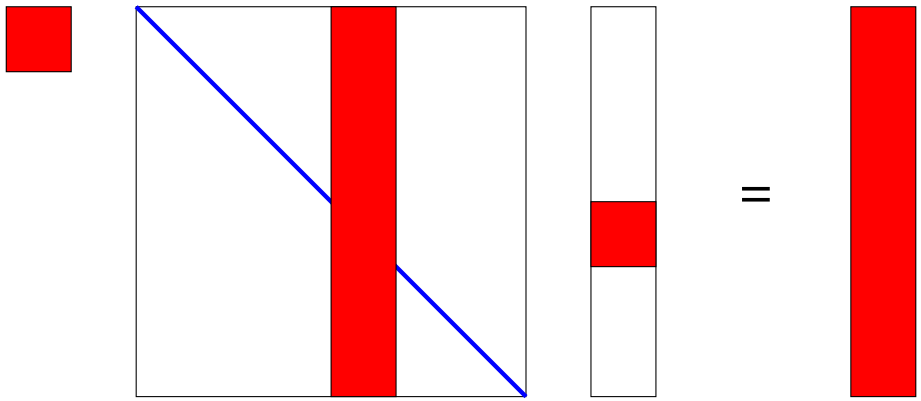


$$\mathbf{x}^{(1)}_{(2)} \equiv \frac{dt}{d\mathbf{x}^{(1)}}^T = \nabla F(\mathbf{x}))^T \cdot y^{(1)}_{(2)}$$

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

Comments:

- second-order term requires proof since actually $\frac{dt}{dy^{(1)}} \in \mathbb{R}$, $\frac{dy^{(1)}}{d\nabla F} \in \mathbb{R}^{1\times(1\times n)}$, and $\frac{d\nabla F}{d\mathbf{x}} \in \mathbb{R}^{(1\times n)\times n}$.
- $\mathbf{x}, \mathbf{x}^{(1)}, y_{(2)}, y^{(1)}_{(2)} \Rightarrow \mathbf{x}^{(2)}$
- $\mathbf{x}, y^{(1)}_{(2)} \Rightarrow \mathbf{x}^{(1)}_{(2)}$
- $\mathbf{x}^{(1)}$ varied? y useful?



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

... harvesting of the whole Hessian column-wise by seeding input directions $\mathbf{x}^{(1)} \in \mathbb{R}^n$ with the Cartesian basis vectors in $\mathbb{R}^n$ for $y_{(1)}^{(2)} = 1$, $\mathbf{x}_{(2)} = 0$, and $y_{(2)}^{(1)} = 0$; harvesting from $\mathbf{x}_{(2)}$.

A second derivative code

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

generated in Adjoint-over-Adjoint (AoA) mode computes

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

as follows:

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

Second directional differentiation of

$$\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(1)} \end{pmatrix}$$

in adjoint mode ...

- ... yields for $\frac{dy}{d\mathbf{x}}$ in direction $y_{(2)}$ a (potentially vanishing; see *essential activity / usefulness*) contribution to result $\mathbf{x}_{(2)}$;
- ... yields for $\frac{dy}{dy_{(1)}}$ in direction $y_{(2)}$ a vanishing contribution to $y_{(1,2)}$; (no dependence of y on $y_{(1)}$)
- ... yields for $\frac{d\mathbf{x}_{(1)}}{d\mathbf{x}}$ in direction $\mathbf{x}_{(1,2)}$ a contribution to $\mathbf{x}_{(2)}$;
- ... yields for $\frac{d\mathbf{x}_{(1)}}{dy_{(1)}}$ in direction $\mathbf{x}_{(1,2)}$ a (potentially vanishing; see *essential activity / variedness*) contribution to $y_{(1,2)}$.

Directional differentiation in adjoint mode of the first-order adjoint model

$$\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix} = \begin{pmatrix} F(\mathbf{x}) \\ \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(1)} \end{pmatrix}$$

in direction $(y_{(2)} \ \mathbf{x}_{(1,2)})^T$ yields

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ y_{(1,2)} \end{pmatrix} \equiv \frac{d\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix}}{d(\mathbf{x} \ y_{(1)})}^T \cdot \begin{pmatrix} y_{(2)} \\ \mathbf{x}_{(1,2)} \end{pmatrix} = \begin{pmatrix} \frac{dy}{d\mathbf{x}}^T \cdot y_{(2)} + \frac{d\mathbf{x}_{(1)}}{d\mathbf{x}}^T \cdot \mathbf{x}_{(1,2)} \\ \underbrace{\frac{dy}{dy_{(1)}}^T \cdot y_{(2)}}_{=0} + \frac{d\mathbf{x}_{(1)}}{dy_{(1)}}^T \cdot \mathbf{x}_{(1,2)} \end{pmatrix}$$

$$\underset{=}{\left[\mathbf{x}_{(1)}=y_{(1)}\cdot\nabla F(\mathbf{x})^T\right]} \begin{pmatrix} \frac{dF(\mathbf{x})}{d\mathbf{x}}^T \cdot y_{(2)} + y_{(1)} \cdot \frac{d^2F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}_{(1,2)} \\ \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}_{(1,2)} \end{pmatrix}$$

Aiming for second-order derivatives with respect to $\mathbf{x}$ we may consider y as not useful and $y_{(1)}$ as not varied; i.e, both y and $y_{(1)}$ are passive with respect to the application of adjoint mode AD to the first-order adjoint code.

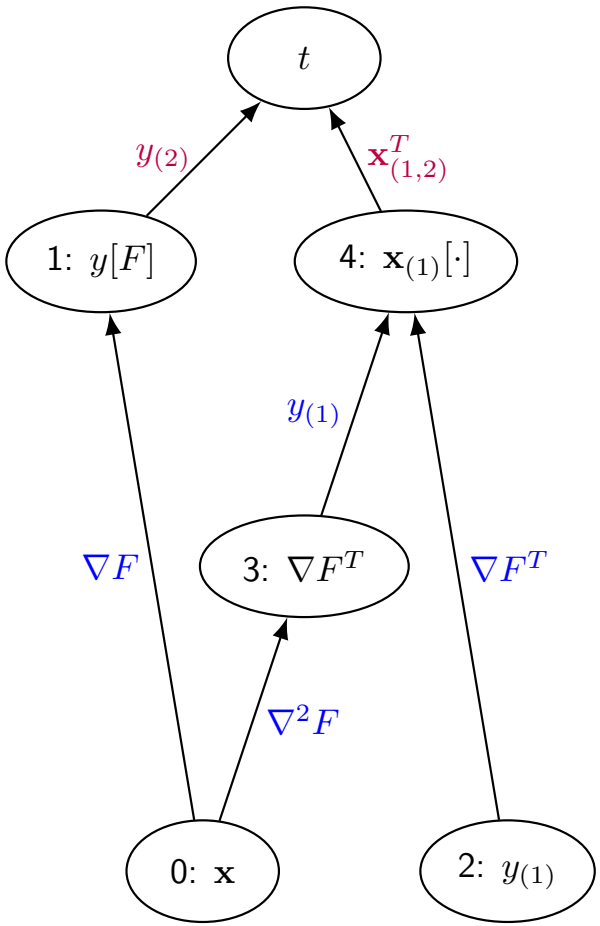
Consequently,

$$\mathbf{x}_{(1)} \equiv \frac{d\mathbf{x}_{(1)}}{d\mathbf{x}} \cdot \mathbf{x}_{(1,2)} = y_{(1)} \cdot \frac{d^2 F(\mathbf{x})}{d\mathbf{x}^2} \cdot \mathbf{x}_{(1,2)}$$

Graphical illustration is provided by a corresponding reduction of the adjoint-augmented adjoint IDAG; see below.

Second-Order AoA Model

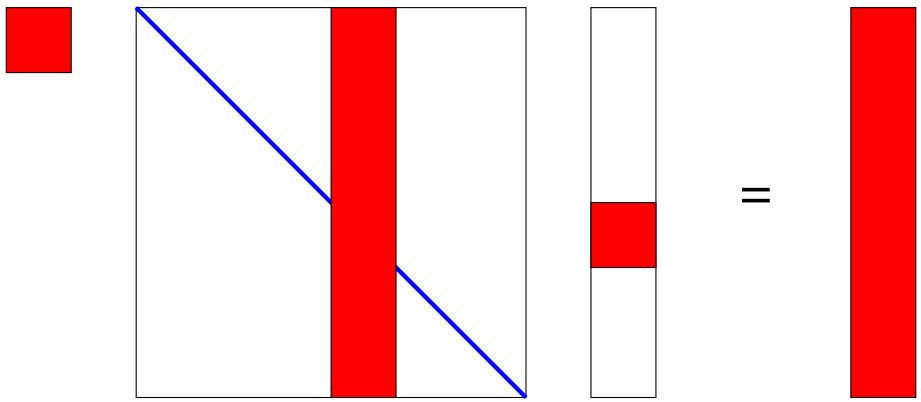
Graphically on Adjoint-Augmented Adjoint lDAG



$$\mathbf{x}_{(2)} \equiv \frac{dt}{d\mathbf{x}}^T$$
$$= \nabla F(\mathbf{x})^T \cdot y_{(2)} + y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}_{(1,2)}$$
$$y_{(1,2)} \equiv \frac{dt}{dy_{(1)}}^T$$
$$= \nabla F(\mathbf{x})^T \cdot \mathbf{x}_{(1,2)}$$

Comments:

- second-order term requires proof since actually $\frac{dt}{d\mathbf{x}_{(1)}} \in \mathbb{R}^{1\times n}$, $\frac{d\mathbf{x}_{(1)}}{d\nabla F} \in \mathbb{R}^{n\times(1\times n)}$, and $\frac{d\nabla F}{d\mathbf{x}} \in \mathbb{R}^{(1\times n)\times n}$.
- $\mathbf{x}, y_{(1)}, y_{(2)}, \mathbf{x}_{(1,2)} \Rightarrow \mathbf{x}_{(2)}$
- $\mathbf{x}, \mathbf{x}_{(1,2)} \Rightarrow y_{(1,2)}$
- $y_{(1)}$ varied? y useful?



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

... harvesting of the whole Hessian row-wise by seeding input directions $\mathbf{x}^{(1,2)} \in \mathbb{R}^n$ with the Cartesian basis vectors in $\mathbb{R}^n$ for $y_{(1)} = 1$, $\mathbf{x}_{(2)} = 0$, and $y_{(2)} = 0$; harvesting from $\mathbf{x}_{(2)} = 0$.

We consider multivariate vector functions

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

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

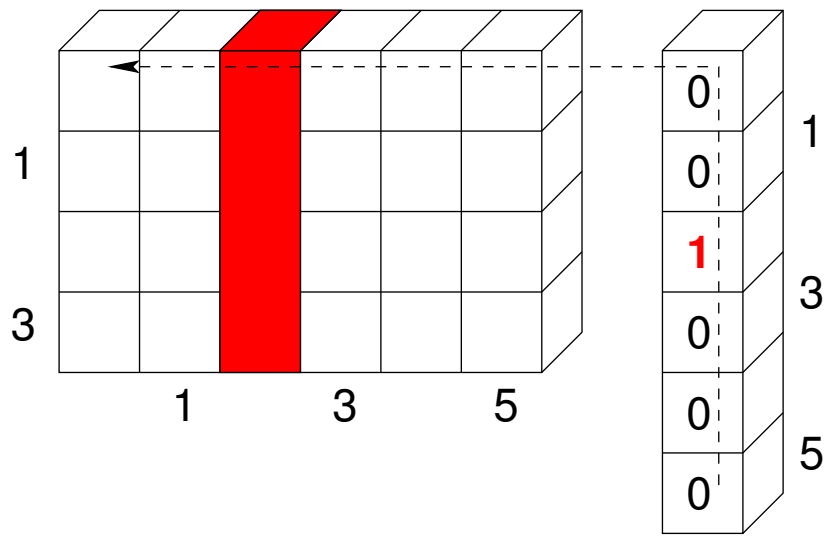
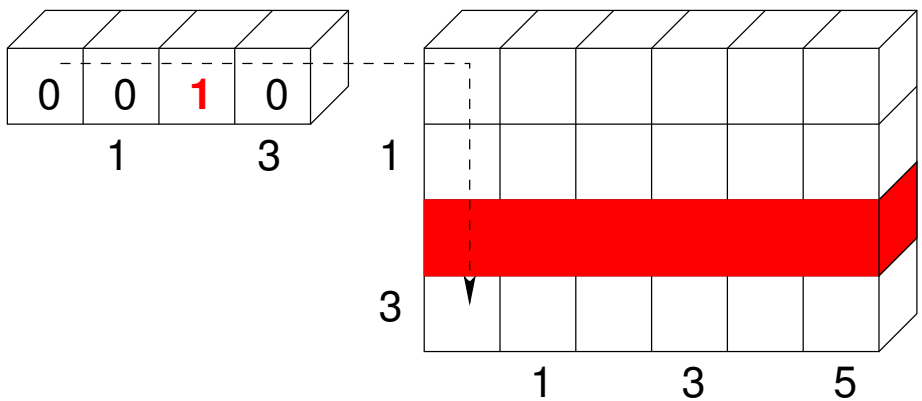
$$\nabla^2 F(\mathbf{x}) \equiv \frac{d^2 F}{d\mathbf{x}^2}(\mathbf{x}).$$

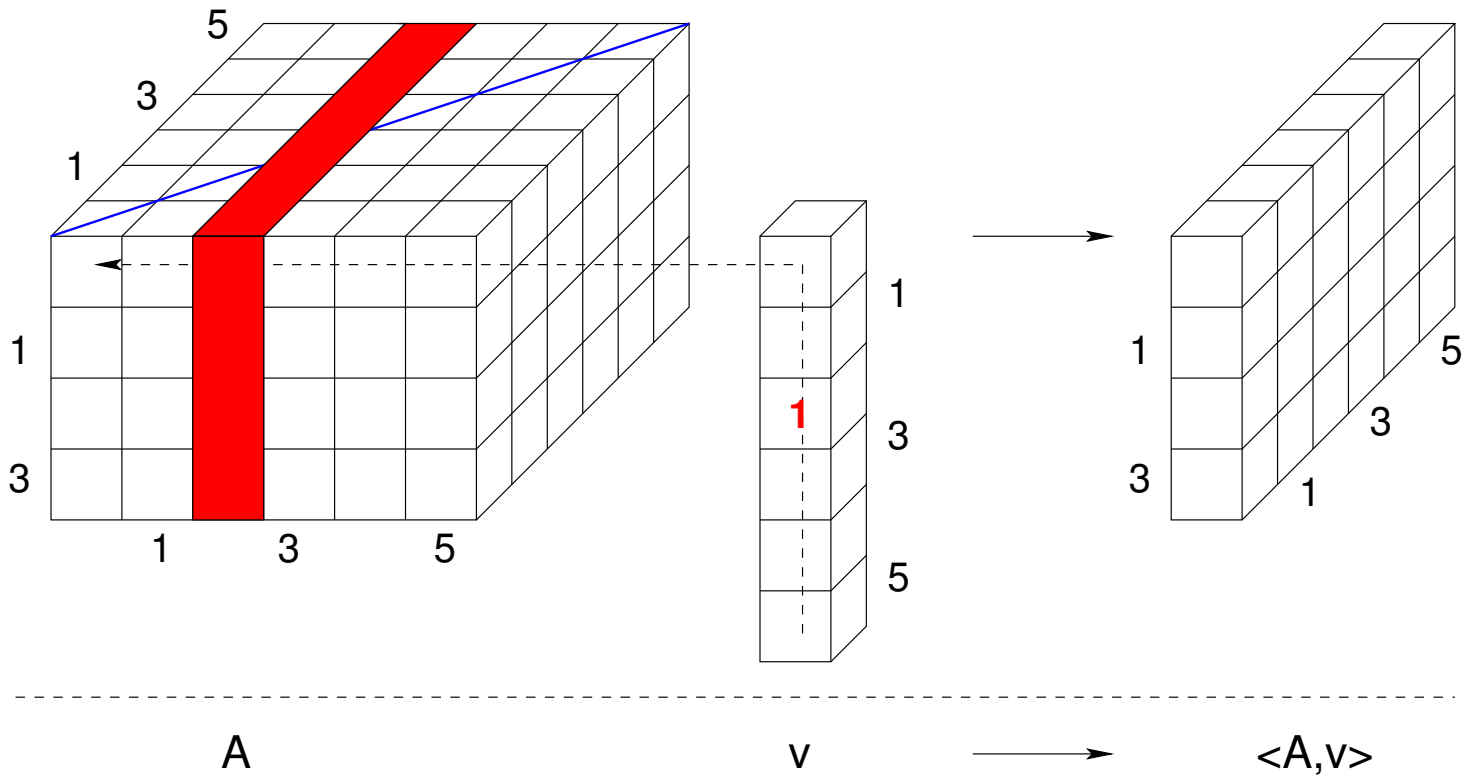
The Hessian is a three-tensor, that is

$$\nabla^2 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n}.$$

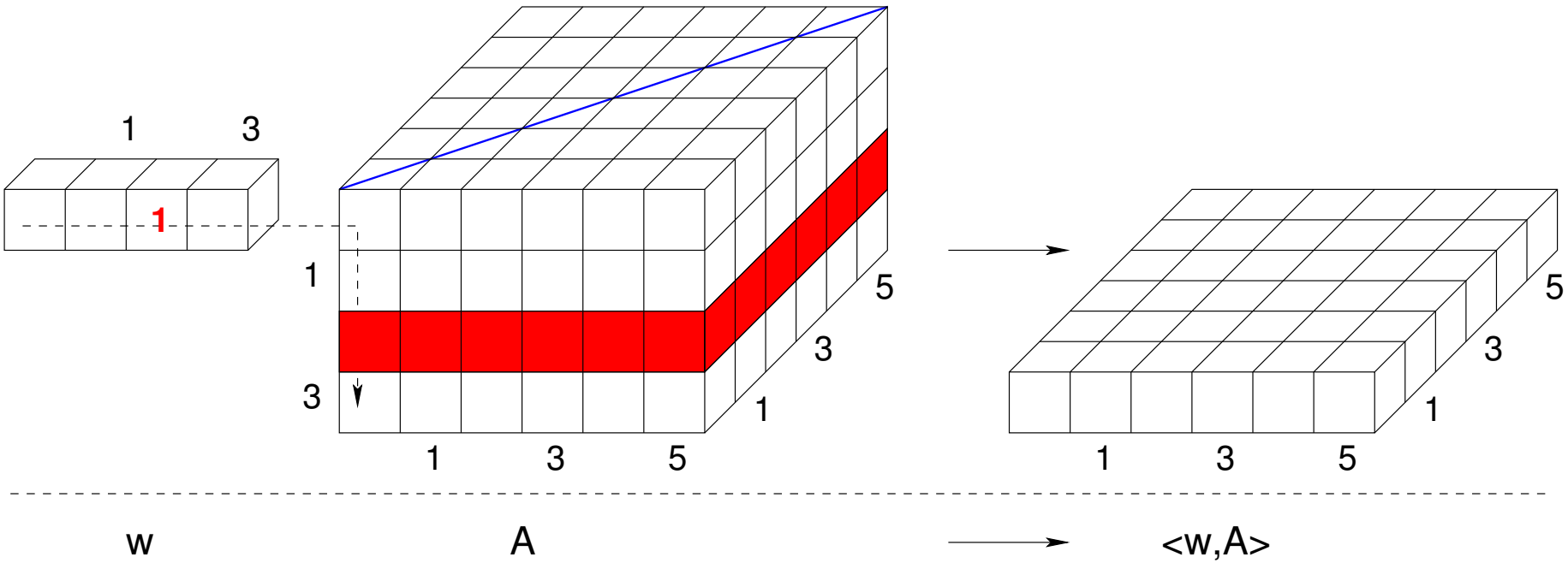
The notation needs to be extended to accommodate projections of Hessian tensors.

e.g. $A \equiv \nabla^2 F,\ F : \mathbb{R}^6 \to \mathbb{R}^4$

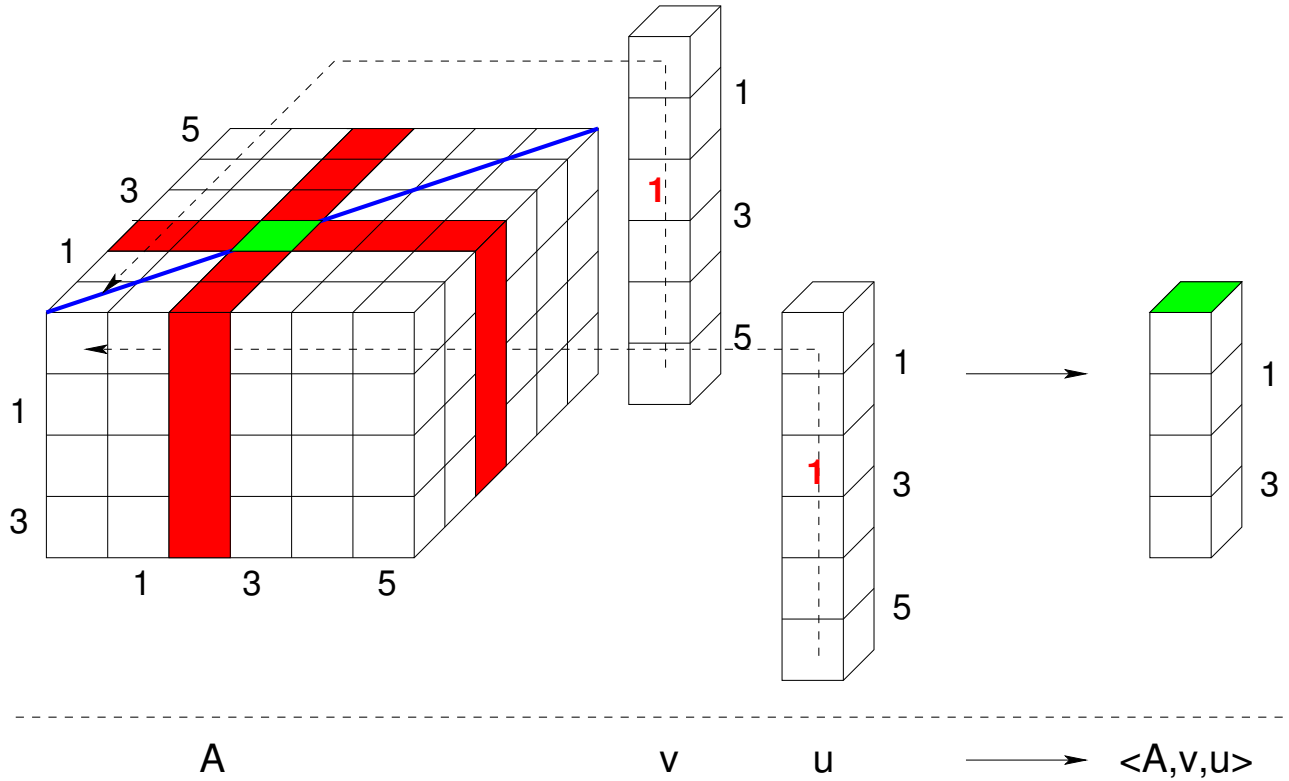
Tangent Projection	Adjoint Projection
	
$< A, v >\equiv A \cdot v$	$< w, A >\equiv A^T \cdot w = (w^T \cdot A)^T$



$$[< A, v >]_{*,j} = [A]_{*,*,j} \cdot v$$



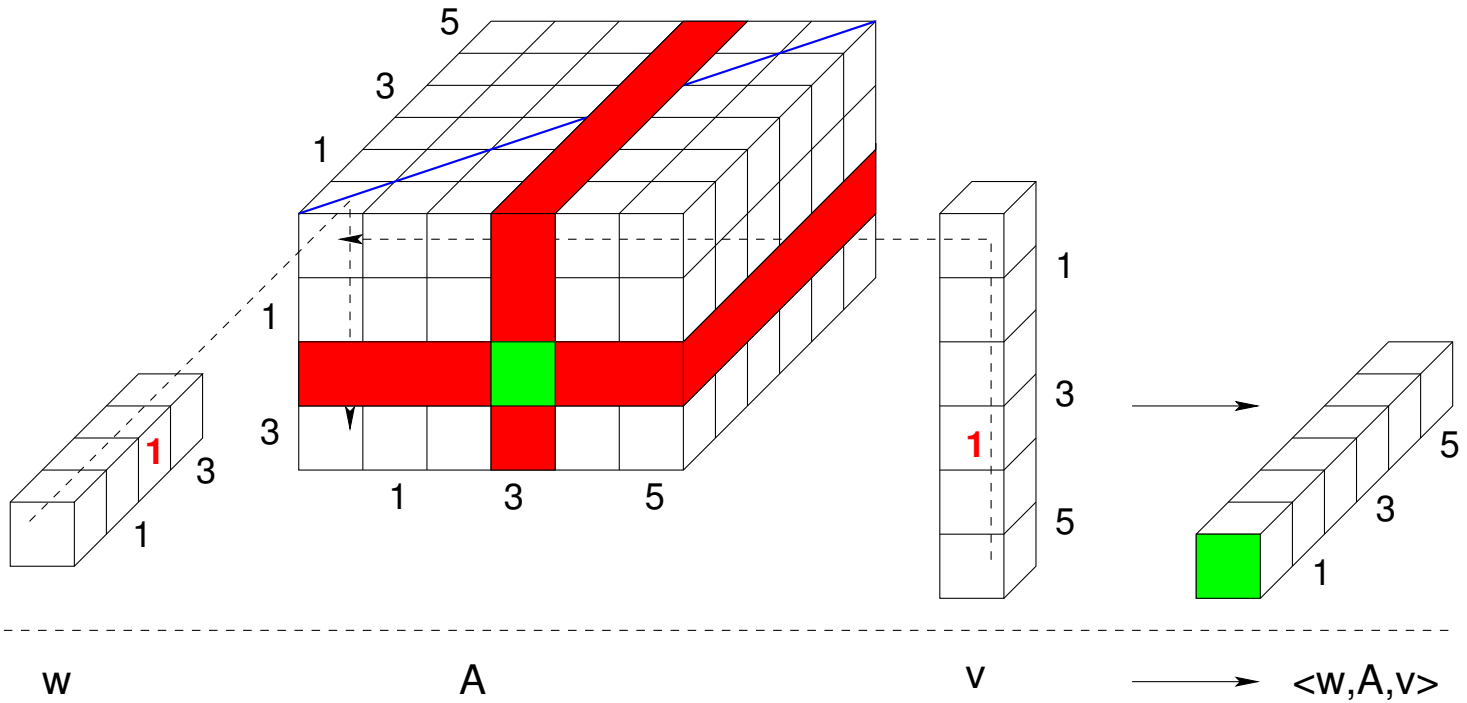
$$[< w, A >]_{*,j} = w^T \cdot [A]_{*,*,j}$$



Note:

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

due to symmetry; see, e.g., [Nau12] for proof.



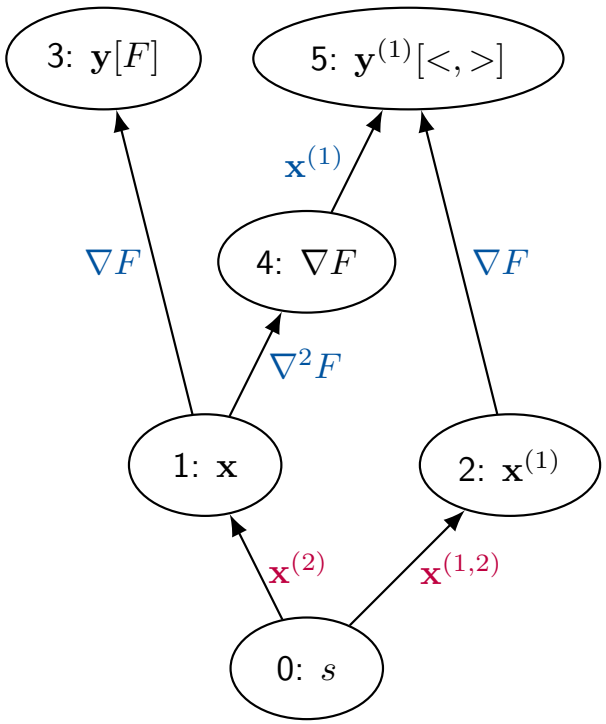
Note:

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

due to symmetry; see, e.g., [Nau12] for proof.

Second-Order ToT Model

Graphically on Tangent-Augmented Tangent IDAG

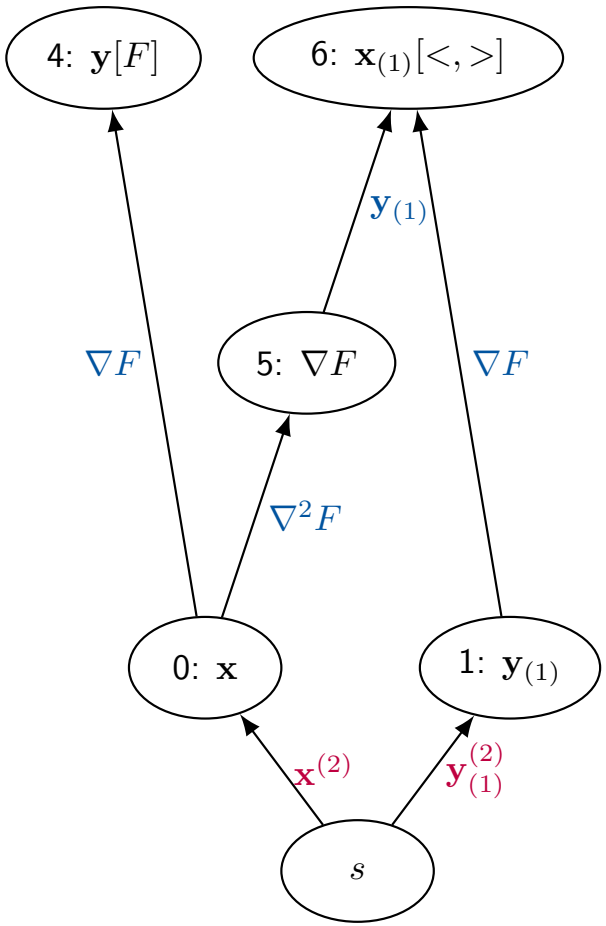


$$
\begin{aligned}
\mathbf{y}^{(1,2)} &= \frac{d\mathbf{y}^{(1)}}{ds} = \frac{d\nabla F \cdot \mathbf{x}^{(1)}}{ds} \\
&= \frac{d < \nabla F, \mathbf{x}^{(1)} >}{ds} \\
&=< \frac{d\nabla F}{ds}, \mathbf{x}^{(1)} > \\
&=<< \frac{d\nabla F}{d\mathbf{x}}, \frac{d\mathbf{x}}{ds} >, \mathbf{x}^{(1)} > \\
&=<< \nabla^2 F, \mathbf{x}^{(2)} >, \mathbf{x}^{(1)} >
\end{aligned}
$$

for passive $\mathbf{x}^{(1)}$.

Second-Order ToA Model

Graphically on Tangent-Augmented Adjoint IDAG

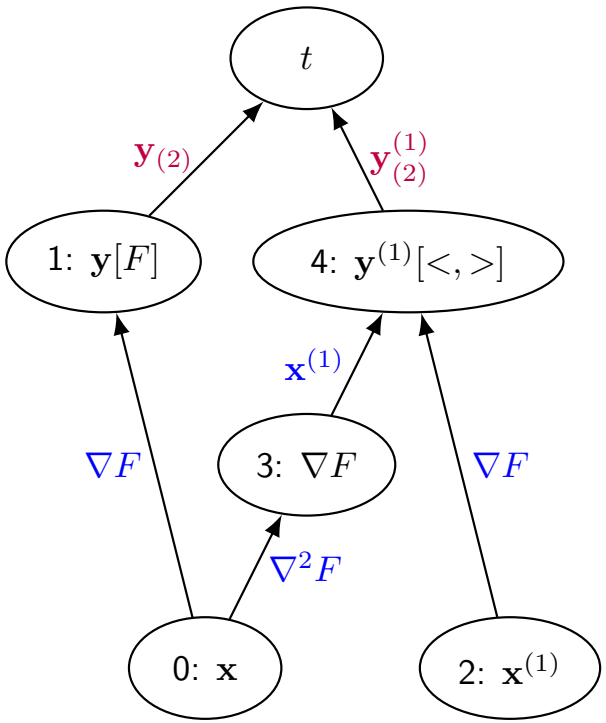


$$
\begin{aligned}
\mathbf{x}^{(2)}_{(1)} &= \frac{d\mathbf{x}_{(1)}}{ds} = \frac{d\nabla F^T \cdot \mathbf{y}_{(1)}}{ds} \\
&= \frac{d < \mathbf{y}_{(1)}, \nabla F >}{ds} \\
&=< \mathbf{y}_{(1)}, \frac{d\nabla F}{ds} > \\
&=< \mathbf{y}_{(1)}, < \frac{d\nabla F}{d\mathbf{x}}, \frac{d\mathbf{x}}{ds} >> \\
&=< \mathbf{y}_{(1)}, < \nabla^2 F, \mathbf{x}^{(2)} >>
\end{aligned}
$$

for passive $\mathbf{y}_{(1)}$.

Second-Order AoT Model

Graphically on Adjoint-Augmented Tangent IDAG

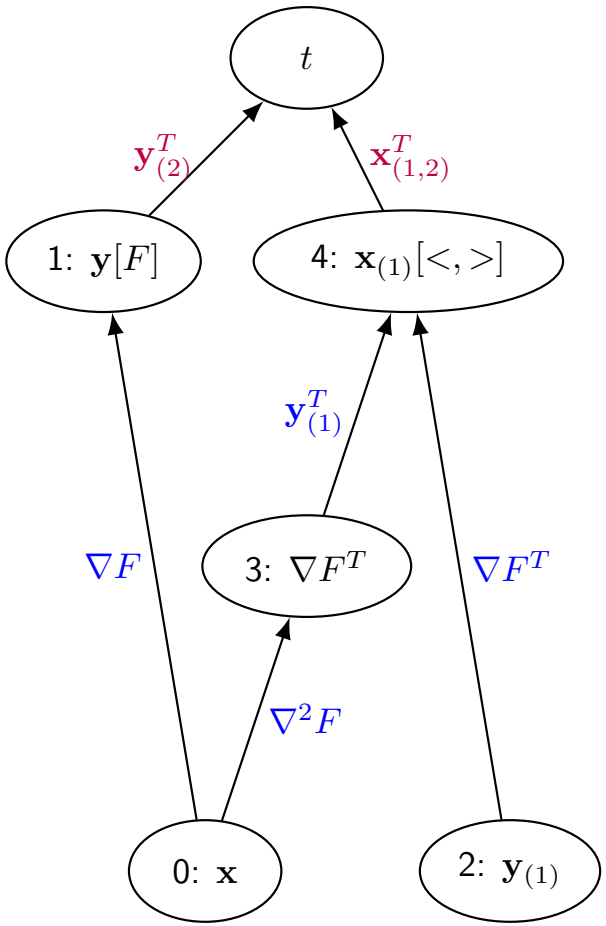


$$\begin{aligned}
\mathbf{x}^T_{(2)} = \frac{dt}{d\mathbf{x}} &=< \frac{dt}{d\mathbf{y}^{(1)}}, \frac{d\nabla F \cdot \mathbf{x}^{(1)}}{d\mathbf{x}} > \\
&=< \frac{dt}{d\mathbf{y}^{(1)}}, \frac{d < \nabla F, \mathbf{x}^{(1)} >}{d\mathbf{x}} > \\
&=< \frac{dt}{d\mathbf{y}^{(1)}}, < \frac{d\nabla F}{d\mathbf{x}}, \mathbf{x}^{(1)} >> \\
&=< \mathbf{y}^{(1)}_{(2)}, < \nabla^2 F, \mathbf{x}^{(1)} >>
\end{aligned}$$

for passive $\mathbf{x}^{(1)}$.

Second-Order AoA Model

Graphically on Adjoint-Augmented Adjoint lDAG



$$\begin{aligned}
\mathbf{x}_{(2)}^T = \frac{dt}{d\mathbf{x}} &=< \frac{dt}{d\mathbf{x}_{(1)}}, \frac{d\nabla F^T \cdot \mathbf{y}_{(1)}}{d\mathbf{x}} > \\
&=< \frac{dt}{d\mathbf{x}_{(1)}}, \frac{d < \mathbf{y}_{(1)}, \nabla F >}{d\mathbf{x}} > \\
&=< \frac{dt}{d\mathbf{x}_{(1)}}, < \mathbf{y}_{(1)}, \frac{d\nabla F}{d\mathbf{x}} > \\
&=< \mathbf{x}_{(1,2)}, < \mathbf{y}_{(1)}, \nabla^2 F >
\end{aligned}$$

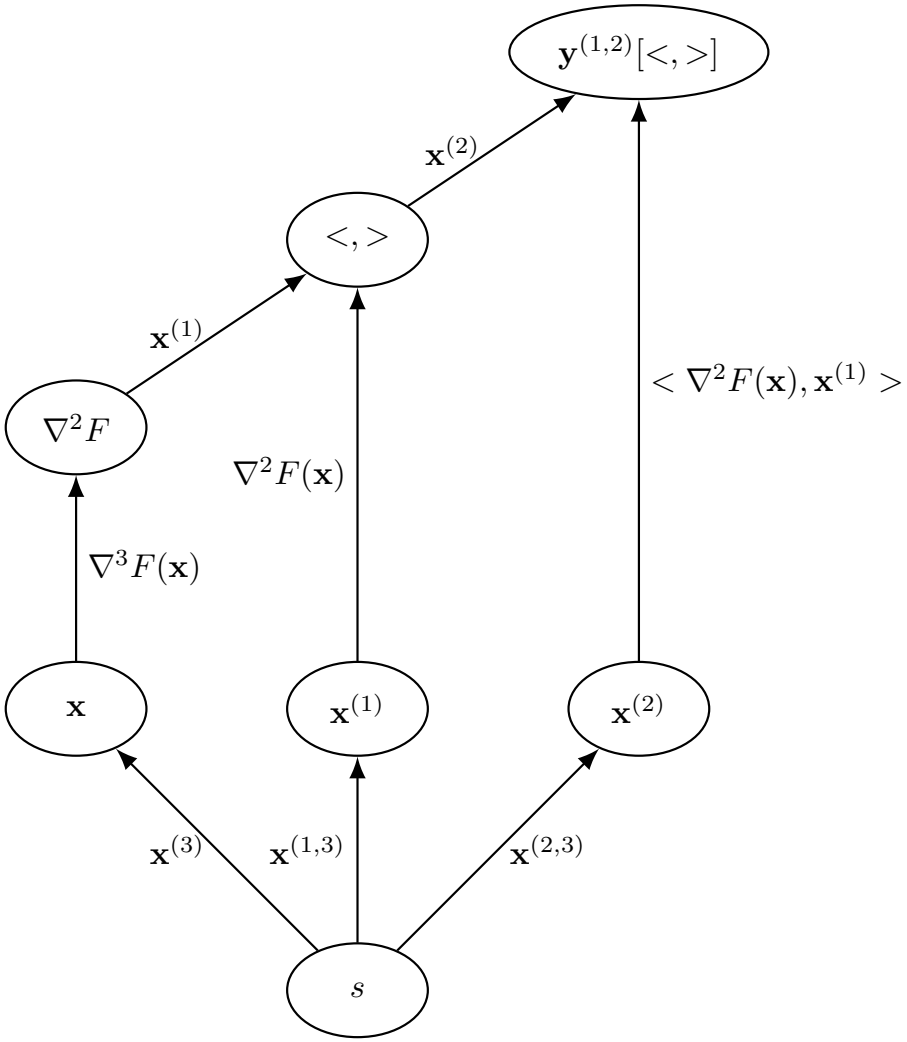
for passive $\mathbf{y}_{(1)}$.

Outline

$$
\begin{aligned}
\mathbf{y} &:= F(\mathbf{x}) \\
\mathbf{y}^{(3)} &:= < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(3)} > \\
\mathbf{y}^{(2)} &:= < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(2)} > \\
\mathbf{y}^{(2,3)} &:= < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} > + < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(2,3)} > \\
\mathbf{y}^{(1)} &:= < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(1)} > \\
\mathbf{y}^{(1,3)} &:= < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(3)} > + < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(1,3)} > \\
\mathbf{y}^{(1,2)} &:= < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} > + < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(1,2)} > \\
\mathbf{y}^{(1,2,3)} &:= < \frac{d^3F}{d\mathbf{x}^3}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} > + < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(1,3)}, \mathbf{x}^{(2)} > + < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(1)}, \mathbf{x}^{(2,3)} > \\
&\quad + < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(1,2)}, \mathbf{x}^{(3)} > + < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(1,2,3)} > .
\end{aligned}
$$

Third-Order ToToT Model

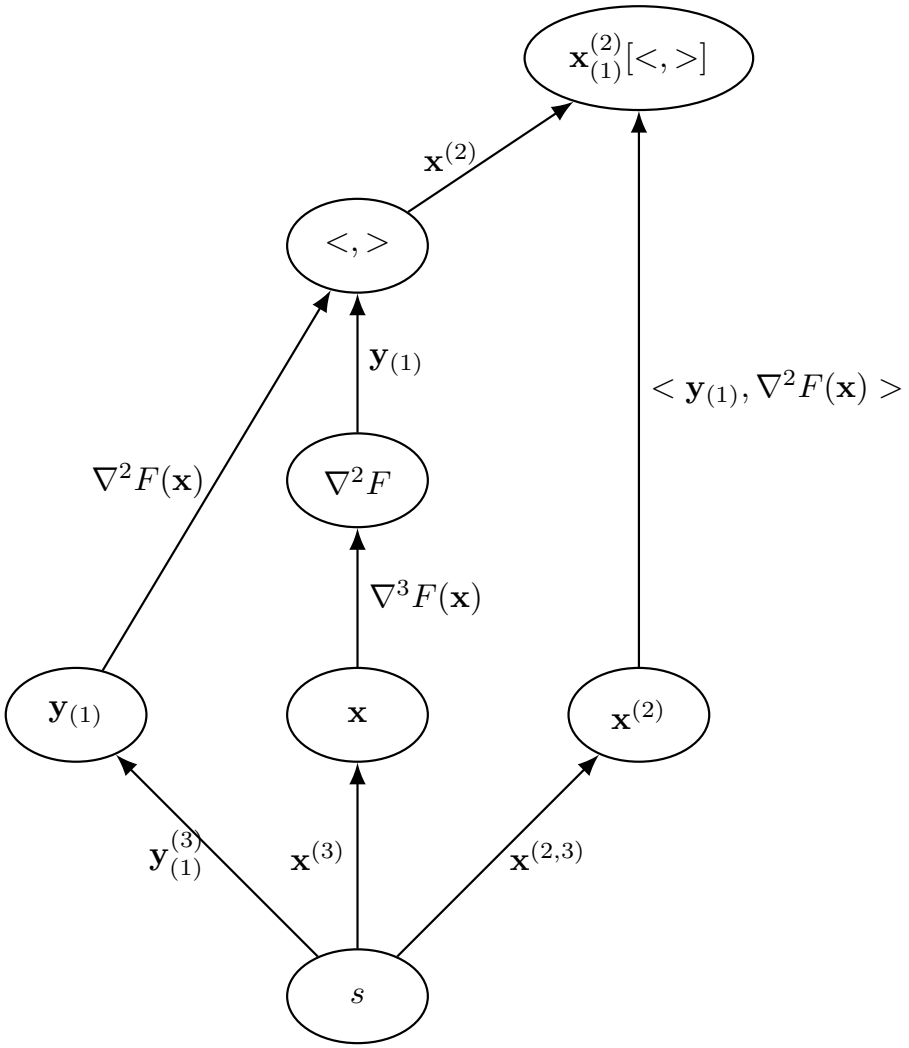
Graphically on Tangent-Augmented ToT-IDAG



$$
\begin{aligned}
\mathbf{y} &:= F(\mathbf{x}) \\
\mathbf{y}^{(3)} &:= < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(3)} > \\
\mathbf{y}^{(2)} &:= < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(2)} > \\
\mathbf{y}^{(2,3)} &:= < \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} > + < \frac{dF}{d\mathbf{x}}, \mathbf{x}^{(2,3)} > \\
\mathbf{x}_{(1)} &:= < \mathbf{y}_{(1)}, \frac{dF}{d\mathbf{x}} > \\
\mathbf{x}_{(1)}^{(3)} &:= < \mathbf{y}_{(1)}^{(3)}, \frac{dF}{d\mathbf{x}} > + < \mathbf{y}_{(1)}, \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(3)} > \\
\mathbf{x}_{(1)}^{(2)} &:= < \mathbf{y}_{(1)}^{(2)}, \frac{dF}{d\mathbf{x}} > + < \mathbf{y}_{(1)}, \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(2)} > \\
\mathbf{x}_{(1)}^{(2,3)} &:= < \mathbf{y}_{(1)}^{(2,3)}, \frac{dF}{d\mathbf{x}} > + < \mathbf{y}_{(1)}^{(2)}, \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(3)} > + < \mathbf{y}_{(1,2)}, \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(2)} > \\
&\quad + < \mathbf{y}_{(1)}, \frac{d^3F}{d\mathbf{x}^3}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} > + < \mathbf{y}_{(1)}, \frac{d^2F}{d\mathbf{x}^2}, \mathbf{x}^{(2,3)} > .
\end{aligned}
$$

Third-Order ToToA Model

Graphically on Tangent-Augmented ToA-IDAG



Live:

- Tangent over Adjoint over Tangent over Adjoint (ToAoToA)
- Adjoint over Tangent over Adjoint over Tangent (AoToAoT)

Outline

Adjoint AD computes

$$X_{0(1)} = \underset{\in \mathbf{R}^{n \times l}}{X_{(1)}} = \nabla F(\mathbf{x})^T \cdot \underset{\in \mathbf{R}^{m \times l}}{Y_{(1)}} = \nabla F_1^T \cdot (\ldots (\nabla F_q^T \cdot X_{q_{(1)}}) \ldots)$$

assuming availability of adjoint elementals

$$X_{i-1_{(1)}} = \nabla F_i(\mathbf{x}_{i-1})^T \cdot X_{i(1)}$$

for $i = q, \ldots, 1$ ($\rightarrow$ reversal of data flow).

The minimum requirement for AAD tools is the implementation of adjoint elementals for the intrinsic operations $(+, *, \ldots)$ and functions $(\sin, \exp, \ldots)$ of the given programming language.

Overloading tools (e.g, dco/c++) generate a tape entry per adjoint elemental for interpretation during propagation of the adjoints ($\rightarrow$ basic AAD). Varying levels of granularity (e.g, right-hand sides of assignments) can be considered.

An adjoint object $F_{i(1)}$ comprises both data and instructions necessary for evaluating $X_{i-1(1)} = \nabla F_i(\mathbf{x}_{i-1})^T \cdot X_{i(1)}$.

An adjoint program $F_{(1)}$ is an abstract data structure defined as a partially ordered sequence of adjoint objects.

An appropriately augmented version of the given implementation of F (the forward section $\overrightarrow{F}_{(1)}$ of the adjoint program) is executed to record data required for the evaluation of

$$X_{i-1(1)} = F_{i(1)}(\mathbf{x}_{i-1}, X_{i(1)}) \equiv \nabla F_i(\mathbf{x}_{i-1})^T \cdot X_{i(1)} \text{ for } i = q, \ldots, 1$$

by the reverse section $\overleftarrow{F}_{(1)}$ of the adjoint program.

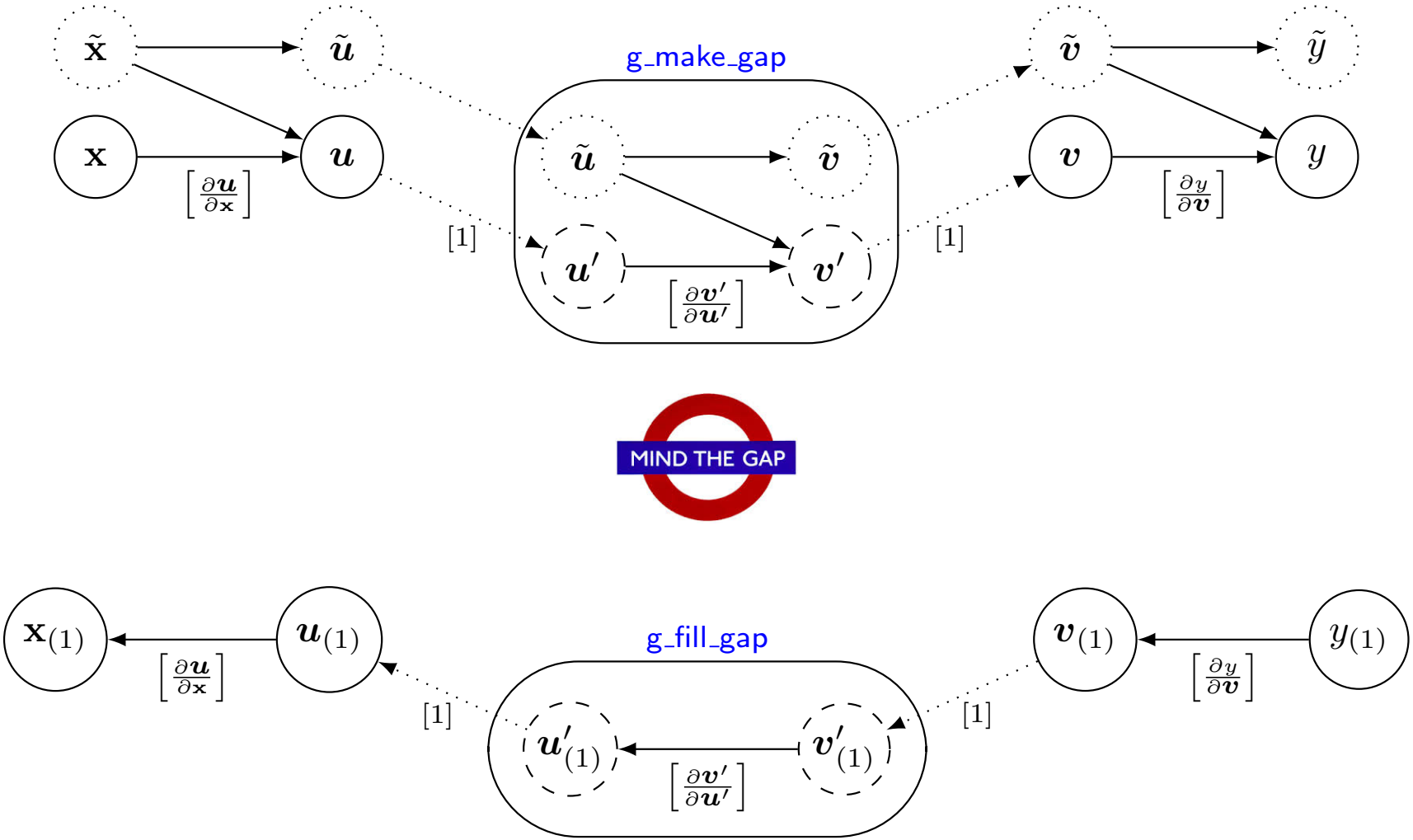
Let $F_{k\,(1)}$ not be implemented by basic AAD.

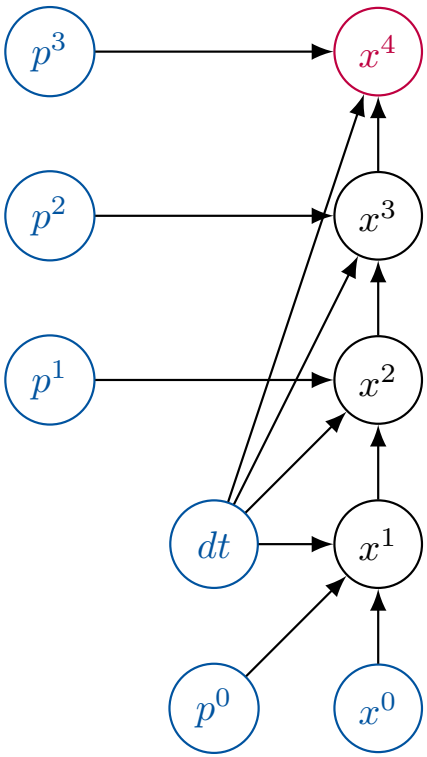
A gap is induced in the tape of the adjoint program

$$X_{(1)} = X_{0\,(1)} \nabla F_1^T \cdot \ldots \cdot \overbrace{\nabla F_k^T(\mathbf{x}_{k-1}) \cdot \underbrace{(\nabla F_{k+1}^T \cdot \ldots \cdot (\nabla F_q^T \cdot X_{q\,(1)}) \ldots)}_{X_{k\,(1)}}}^{X_{k-1\,(1)}}$$

to be filled by a custom version of $F_{k\,(1)}$.

For example, checkpointing methods decrease the maximum tape size by storing $\mathbf{x}_{k-1}$ and postponing the generation of the tape for F_k to the reverse section of $F_{(1)}$.


g_make_gap
[1]
[1]
g_fill_gap
[1]
[1]

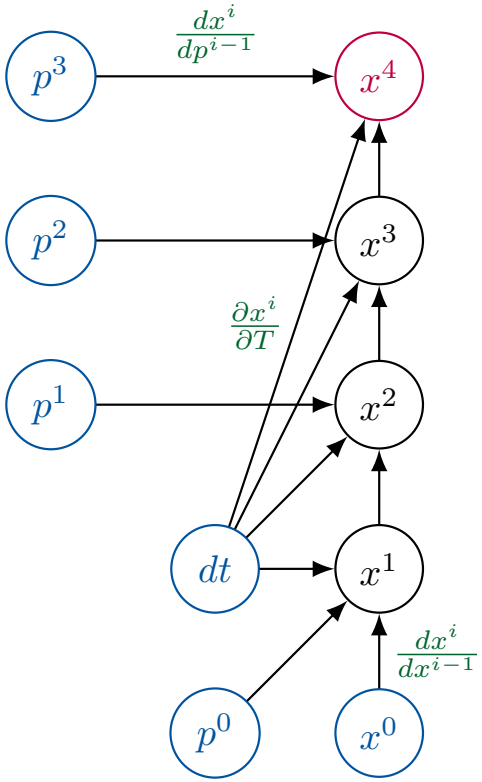



```
void newton(double& x, const double& p,
    const int i, const double& dt) {
  double x_prev=x;
  double f=-dt*p*sin(x*i*dt);
  while (abs(f)>1e-15) {
    x=x-f/(1-dt*p*i*dt*cos(x*i*dt));
    f=x-x_prev-dt*p*sin(x*i*dt);
  }
}
```

Unit memory per vertex / edge in $G = (V, E)$.

Beyond Black-Box Adjoint AD

Implicit Euler: Store All



(PMR,POC)

$(6, 0) \leftarrow$ register $x^0, \mathbf{p}, dt$

$(10, 1) \leftarrow$ compute x^1 and record

$(14, 2) \leftarrow$ compute x^2 and record

$(18, 3) \leftarrow$ compute x^3 and record

$(\mathbf{22}, 4) \leftarrow$ compute x^4 and record; return x^4; set $x^4_{(1)}$

$(18, 4) \leftarrow$ compute $x^3_{(1)}, p^3_{(1)}, dt_{(1)}$; release x_4

$(14, 4) \leftarrow$ compute $x^2_{(1)}, p^2_{(1)}$; increment $dt_{(1)}$; release x_3

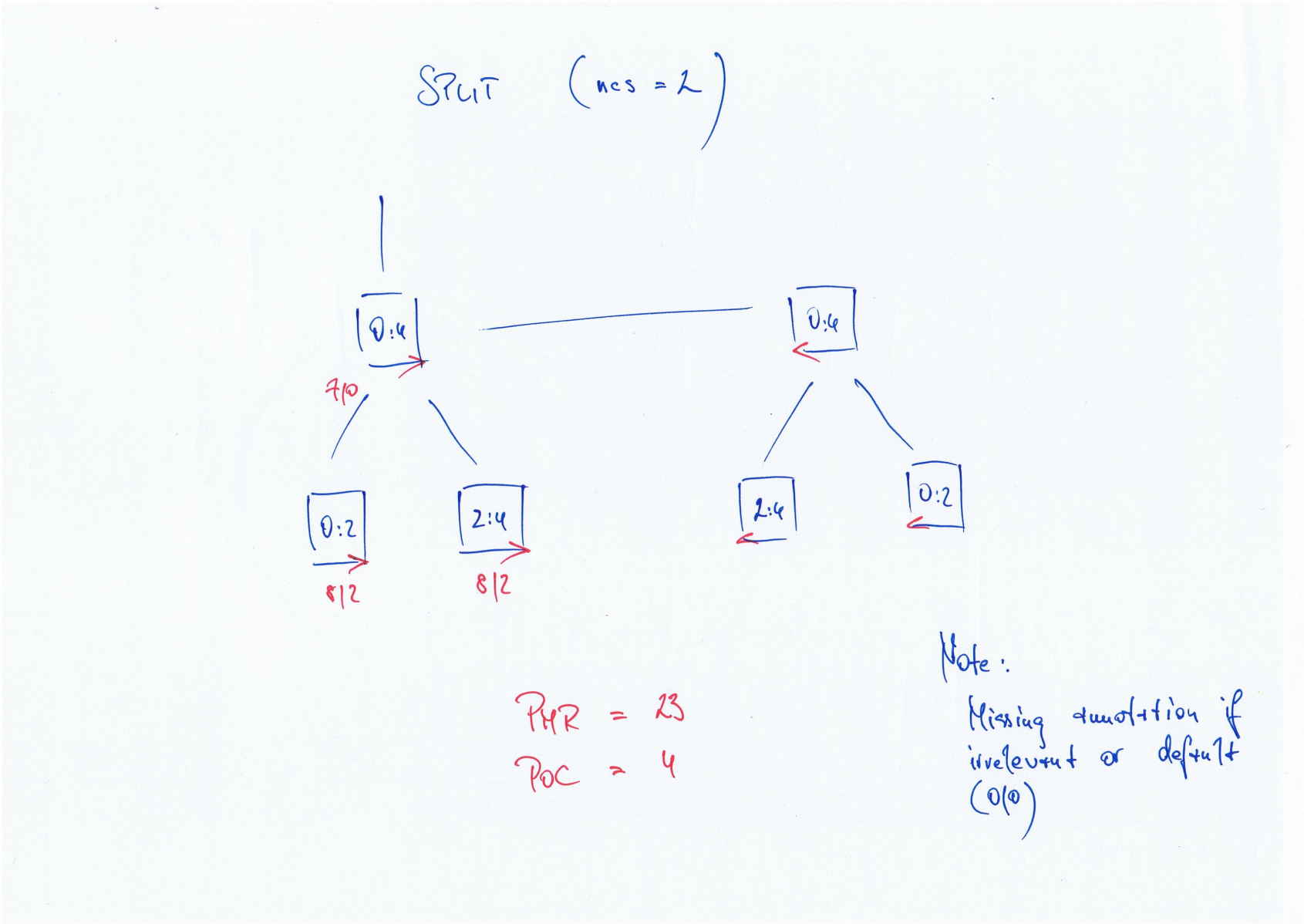
$(10, 4) \leftarrow$ compute $x^1_{(1)}, p^1_{(1)}$; increment $dt_{(1)}$; release x_2

$(8, 4) \leftarrow$ compute $x^0_{(1)}, p^0_{(1)}$; increment $dt_{(1)}$; release x_1

$(0, \mathbf{4}) \leftarrow$ return $x^0_{(1)}, \mathbf{p}_{(1)}, dt_{(1)}$; release $x_0, \mathbf{p}, dt$

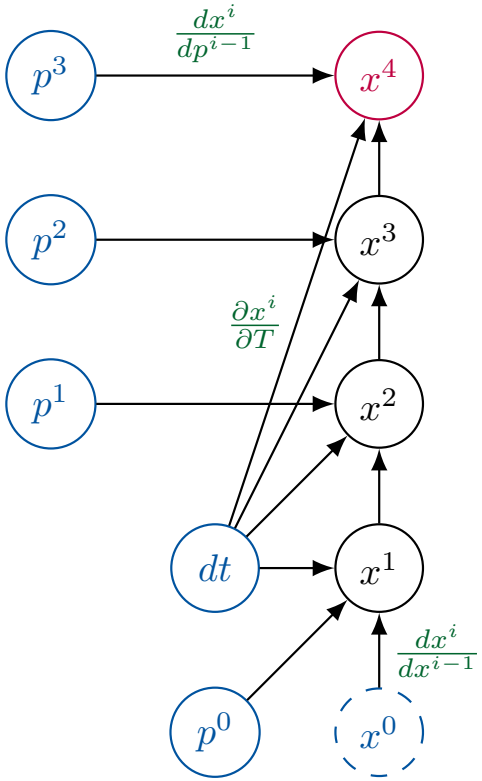
Note: $PMR \sim |E|$, $POC \sim |V|$

Implicit Euler: Store All



Beyond Black-Box Adjoint AD

Implicit Euler: Recompute All



(PMR,POC)

$(7, 0) \leftarrow$ register $x^0, \mathbf{p}, dt$; store x^0

$(11, 4) \leftarrow$ compute x^3; compute x^4 and record; return x^4

$(7, 4) \leftarrow$ set $x^4_{(1)}$; compute $x^3_{(1)}, p^3_{(1)}, dt_{(1)}$; release x^4

$(11, 7) \leftarrow$ restore x^0; compute x^2; compute x^3 and record

$(7, 7) \leftarrow$ compute $x^2_{(1)}, p^2_{(1)}$; increment $dt_{(1)}$; release x^3

$(\mathbf{11}, 9) \leftarrow$ restore x^0; compute x^1; compute x^2 and record

$(7, 9) \leftarrow$ compute $x^1_{(1)}, p^1_{(1)}$; increment $dt_{(1)}$; release x^2

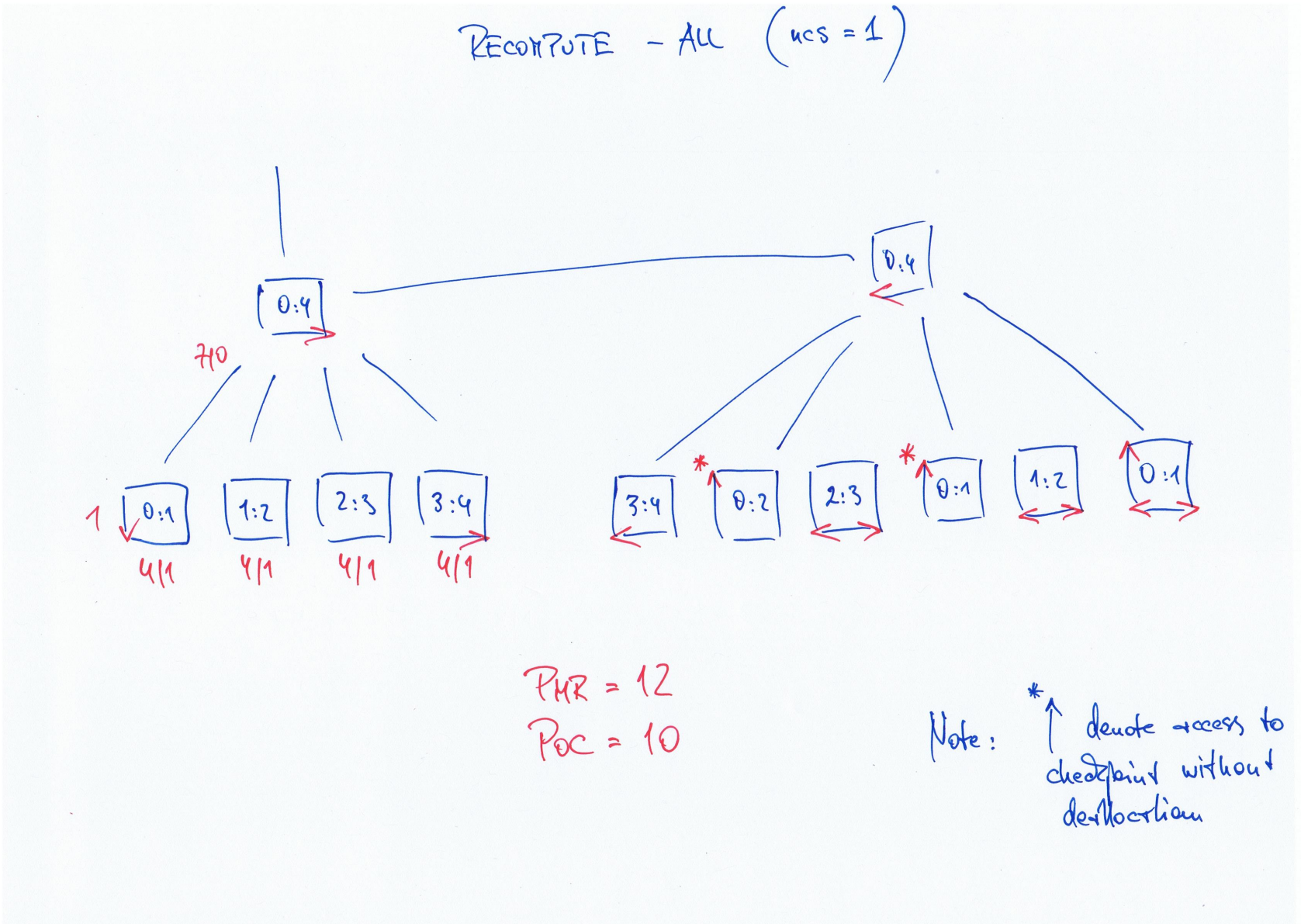
$(10, 10) \leftarrow$ restore and free x^0; compute x^1 and record

$(6, 10) \leftarrow$ compute $x^0_{(1)}, p^0_{(1)}$; increment $dt_{(1)}$; release x^1

$(0, \mathbf{10}) \leftarrow$ return $x^0_{(1)}, \mathbf{p}_{(1)}, dt_{(1)}$; release $x^0, \mathbf{p}, dt$

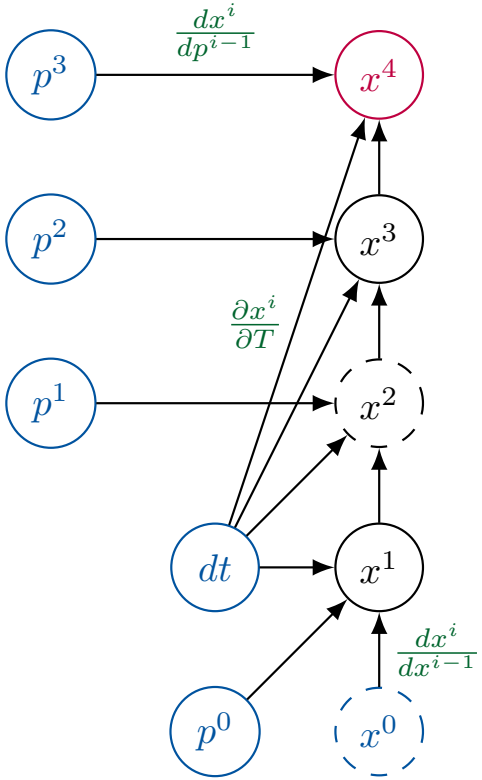
Note: Recording single steps

Implicit Euler: Recompute All



Beyond Black-Box Adjoint AD

Implicit Euler: Joint Reversal / Checkpointing



(PMR,POC)

$(7, 0) \leftarrow$ register $x^0, \mathbf{p}, dt$; store x^0

$(8, 2) \leftarrow$ compute x^2 and store

$(8, 4) \leftarrow$ compute x^4 and return

$(\mathbf{15}, 6) \leftarrow$ restore and free x^2; compute x^3 and record; compute x^4 and record;

$(11, 6) \leftarrow$ set $x^4_{(1)}$; compute $x^3_{(1)}, p^3_{(1)}, dt_{(1)}$; release x^4

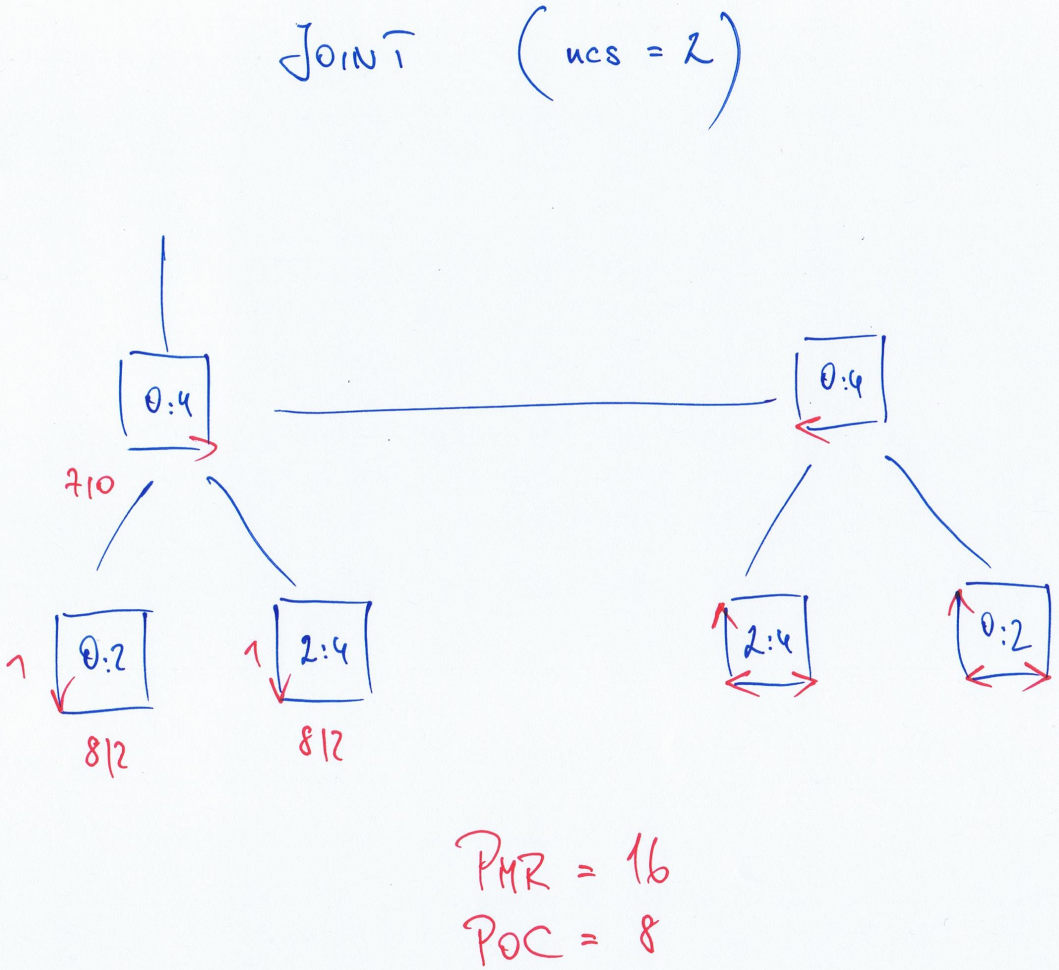
$(7, 6) \leftarrow$ compute $x^2_{(1)}, p^2_{(1)}$; increment $dt_{(1)}$; release x^3

$(14, 8) \leftarrow$ restore and free x^2; compute x^1 and record; compute x^2 and record;

$(10, 8) \leftarrow$ compute $x^1_{(1)}, p^1_{(1)}$; increment $dt_{(1)}$; release x^2

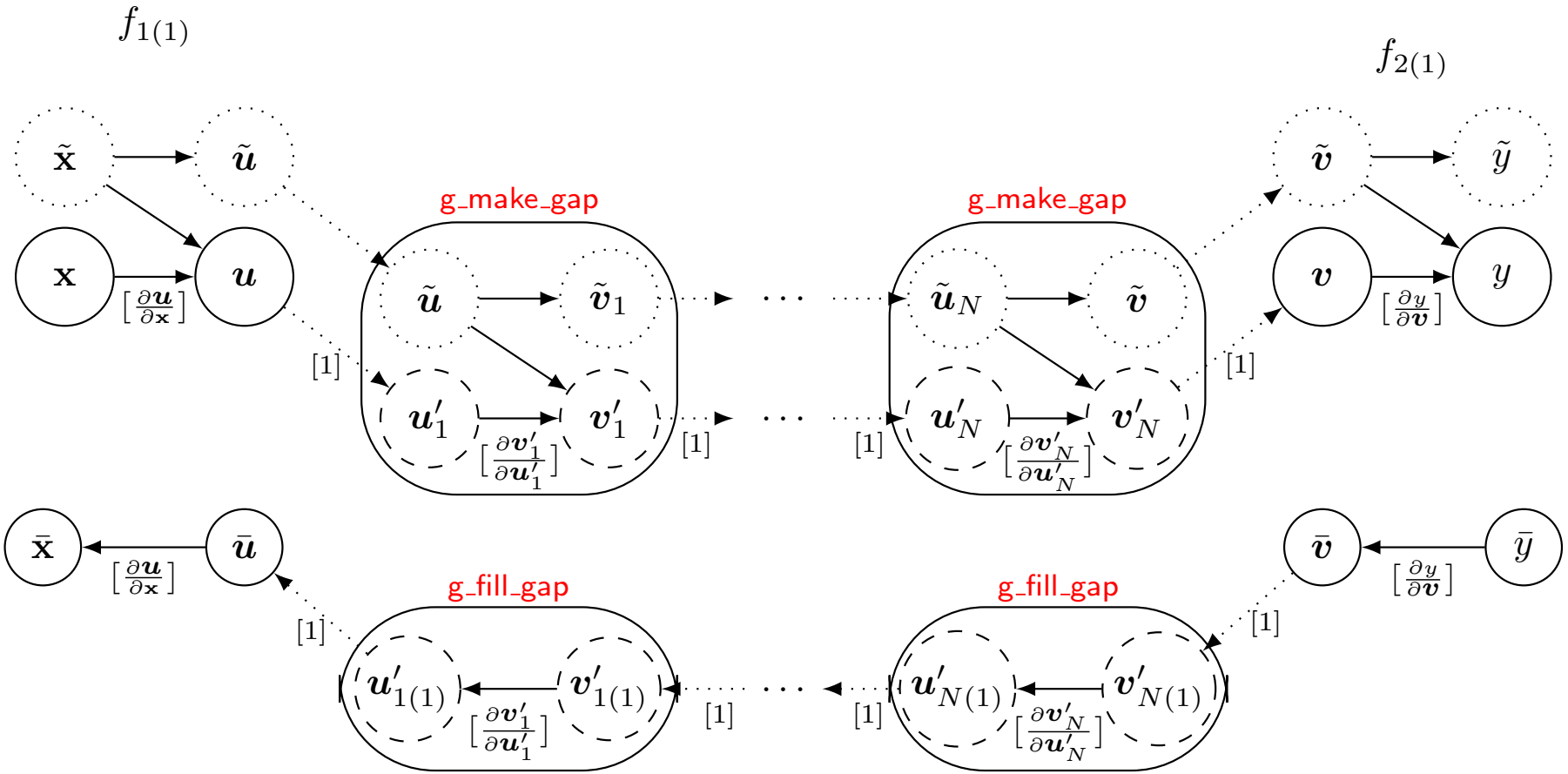
$(6, 8) \leftarrow$ compute $x^0_{(1)}, p^0_{(1)}$; increment $dt_{(1)}$; release x^1

$(0, \mathbf{8}) \leftarrow$ return $x^0_{(1)}, \mathbf{p}_{(1)}, dt_{(1)}$; release $x^0, \mathbf{p}, dt$


JOINT (ncs = 2)
0:4
0:4
7|0
0:2
2:4
2:4
0:2
8|2
8|2
PMR = 16
POC = 8

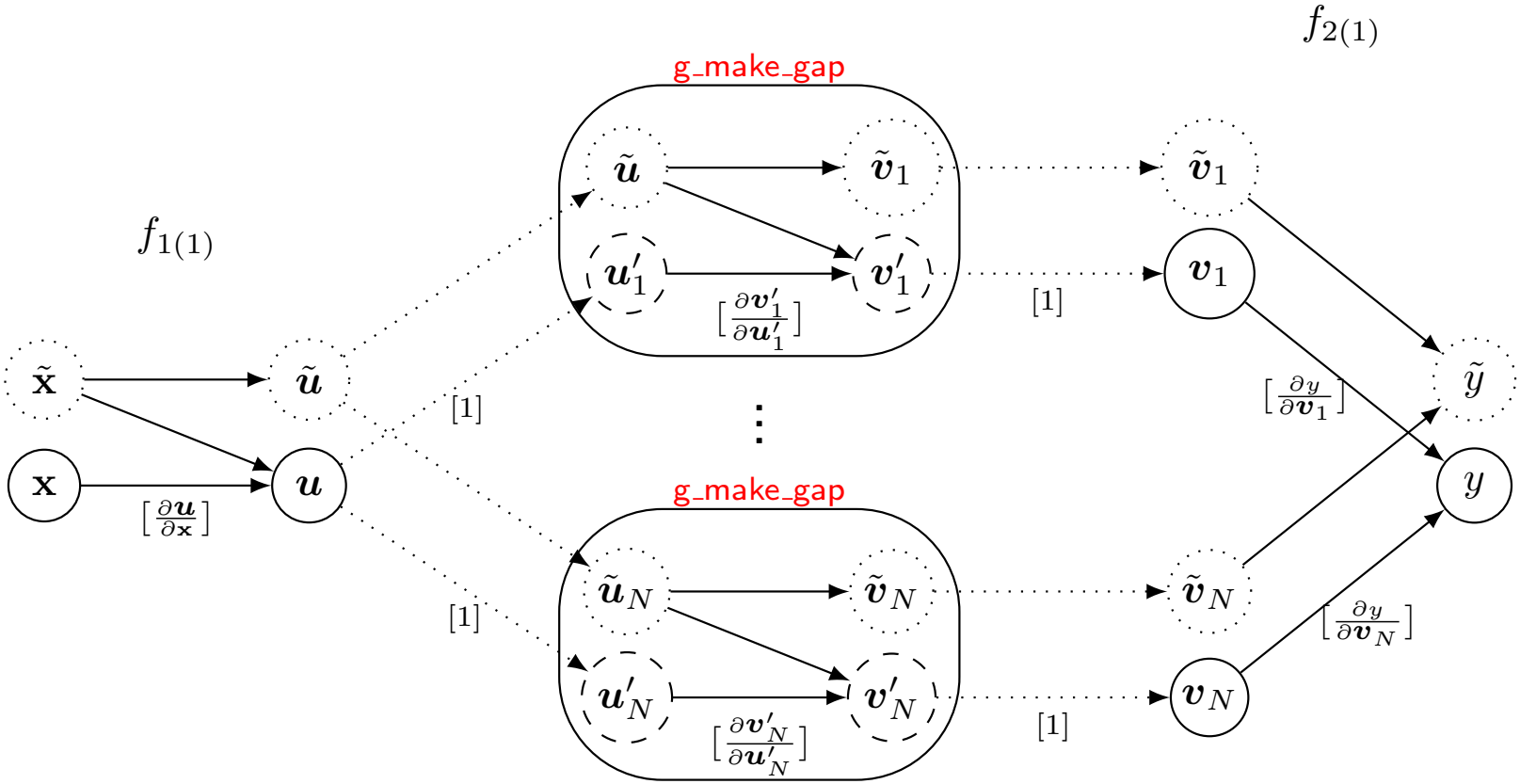
Beyond Black-Box Adjoint AD

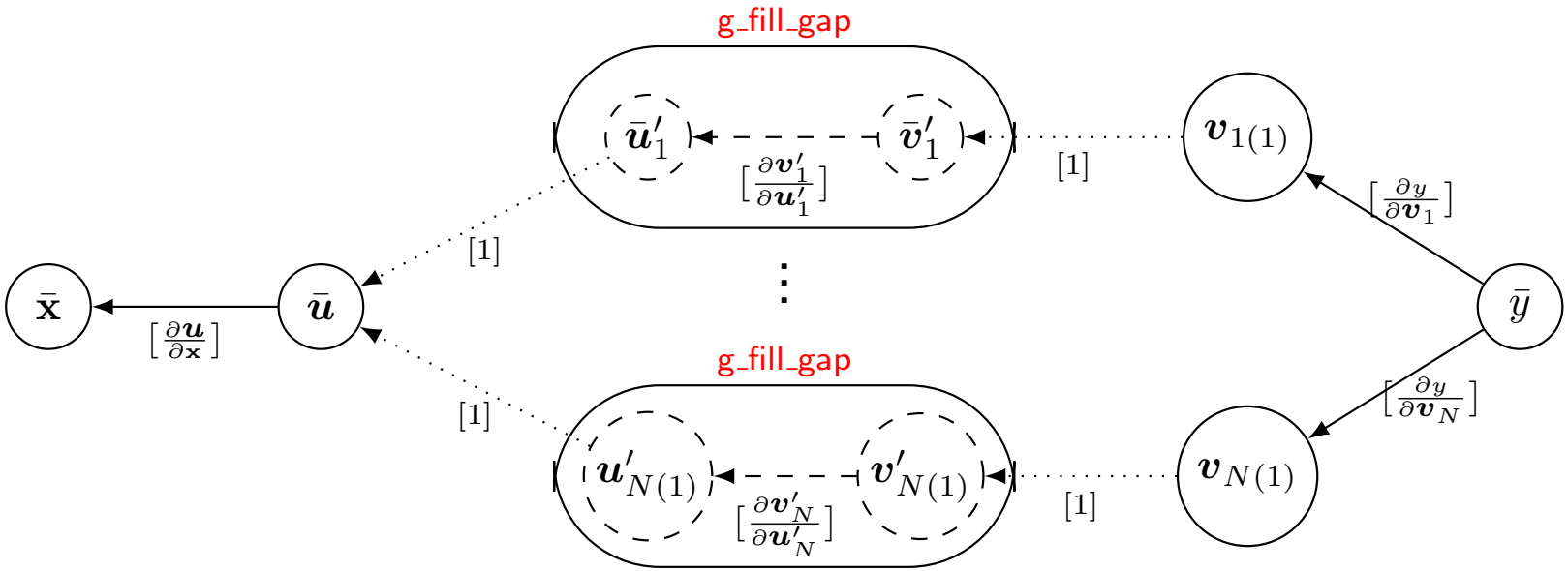
Iterative (Single-Level) Checkpointing



Implicit Euler: Checkpointing Evolution (Live)

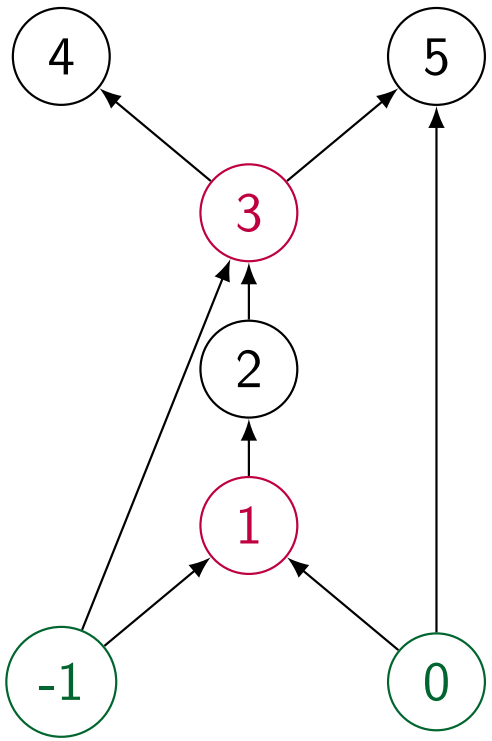
```
for (int j=0;j<n;j+=ncs) {
  cp.push(x);
  for (int i=j;i<min(j+ncs,n);i++)
    newton(x,p[i],i,dt);
}
...
for (int j=n/ncs*ncs-1;j>=0;j-=ncs) {
  x=cp.top(); cp.pop(); MEM-=sizeof(double);
  for (int i=j-ncs+1;i<=j;i++)
    newton(F,x,xa,p[i],pa[i],i,dt);
  for (int i=j;i>j-ncs;i--)
    newton(R,x,xa,p[i],pa[i],i,dt);
}
```


$f_{1(1)}$
$f_{2(1)}$
g_make_gap
g_make_gap
$\tilde{\mathbf{x}}$
$\mathbf{x}$
$\tilde{\boldsymbol{u}}$
$\boldsymbol{u}$
$[\frac{\partial \boldsymbol{u}}{\partial \mathbf{x}}]$
$[1]$
$[1]$
$\tilde{\boldsymbol{u}}$
$\tilde{\boldsymbol{v}}_1$
$\boldsymbol{u}'_1$
$\boldsymbol{v}'_1$
$[\frac{\partial \boldsymbol{v}'_1}{\partial \boldsymbol{u}'_1}]$
$\tilde{\boldsymbol{u}}_N$
$\tilde{\boldsymbol{v}}_N$
$\boldsymbol{u}'_N$
$\boldsymbol{v}'_N$
$[\frac{\partial \boldsymbol{v}'_N}{\partial \boldsymbol{u}'_N}]$
$\tilde{\boldsymbol{v}}_1$
$\boldsymbol{v}_1$
$[1]$
$\tilde{\boldsymbol{v}}_N$
$\boldsymbol{v}_N$
$[1]$
$[\frac{\partial y}{\partial \boldsymbol{v}_1}]$
$[\frac{\partial y}{\partial \boldsymbol{v}_N}]$
$\tilde{y}$
y


g_fill_gap
g_fill_gap
$\bar{\mathbf{x}}$
$\bar{\boldsymbol{u}}$
$\bar{y}$
$\bar{\boldsymbol{u}}'_1$
$\bar{\boldsymbol{v}}'_1$
$\boldsymbol{u}'_{N(1)}$
$\boldsymbol{v}'_{N(1)}$
$\boldsymbol{v}_{1(1)}$
$\boldsymbol{v}_{N(1)}$
$[\frac{\partial \boldsymbol{u}}{\partial \mathbf{x}}]$
$[\frac{\partial \boldsymbol{v}'_1}{\partial \boldsymbol{u}'_1}]$
$[\frac{\partial \boldsymbol{v}'_N}{\partial \boldsymbol{u}'_N}]$
$[\frac{\partial y}{\partial \boldsymbol{v}_1}]$
$[\frac{\partial y}{\partial \boldsymbol{v}_N}]$
[1]
[1]
[1]
[1]

Euler-Maruyama: Checkpointing Ensemble (Live)

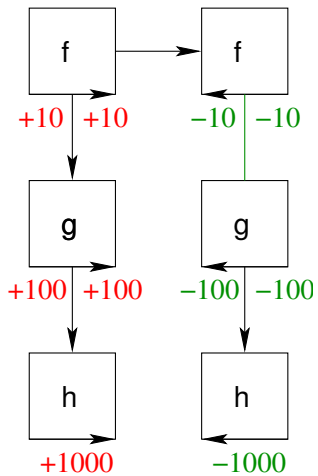
```
for (int j=m-1;j>=0;j--) {
    // augmented primal path
    x=x0; t=0;
    for (int i=0;i<n;i++) {
      tbr.push(x); MEM+=sizeof(double); PMR=max(PMR,MEM);
      x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];
      t+=dt;
    }
    s+=x; x=x0;

    // adjoint path
    x0a+=xa; xa=0; xa+=sa; t=T;
    for (int i=n-1;i>=0;i--) {
      t-=dt;
      x=tbr.top(); tbr.pop(); MEM-=sizeof(double);
      pa[i]+=(dt*sin(x*t)+cos(x*t)*sqrt(dt)*dW[j][i])*xa;
      xa=(1+dt*p[i]*t*cos(x*t)-p[i]*t*sin(x*t)*sqrt(dt)*dW[j][i])*xa;

    }
  }
```



$v_5, v_4, \ldots, v_{-1}$

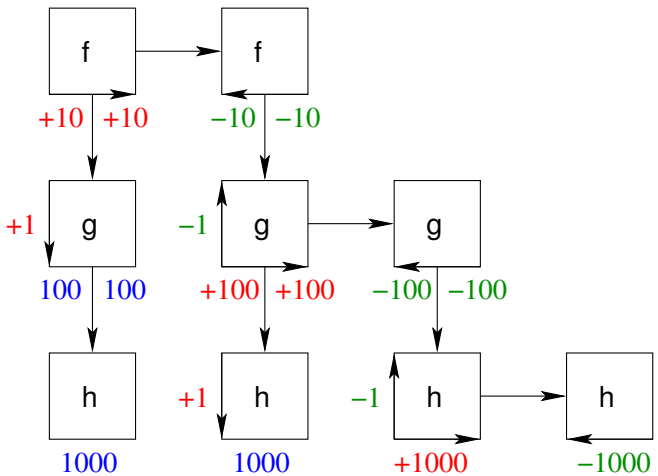
- U.N.: DAG Reversal is NP-Complete, J. Disc. Alg. 7(4), 402-410 (2009).
- U.N.: Call Tree Reversal is NP-Complete, LNCSE 64, 13-22 (2008).
- J. Lotz et al.: Mixed Integer Programming for Call Tree Reversal, SIAM CSC (2016).

R=(0,0):



MEM=1220, OPS=0

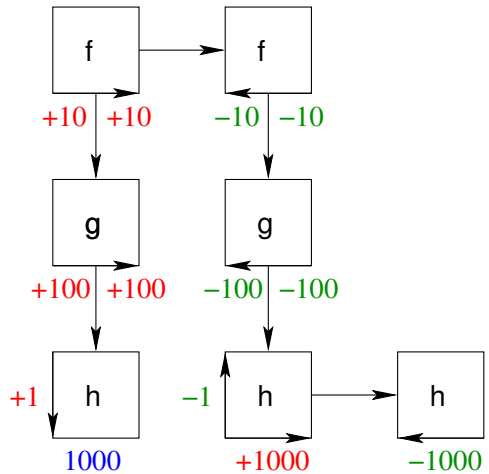
R=(1,1):



MEM=1110, OPS=2200

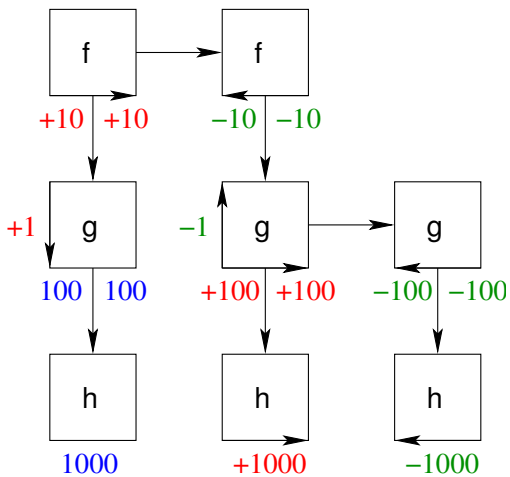
Smallest Memory Increase starts from $\text{R} = \mathbf{1}$ and yields ...
Largest Memory Decrease (LMD) starts from $\text{R} = \mathbf{0}$ and yields ...

R=(0,1):



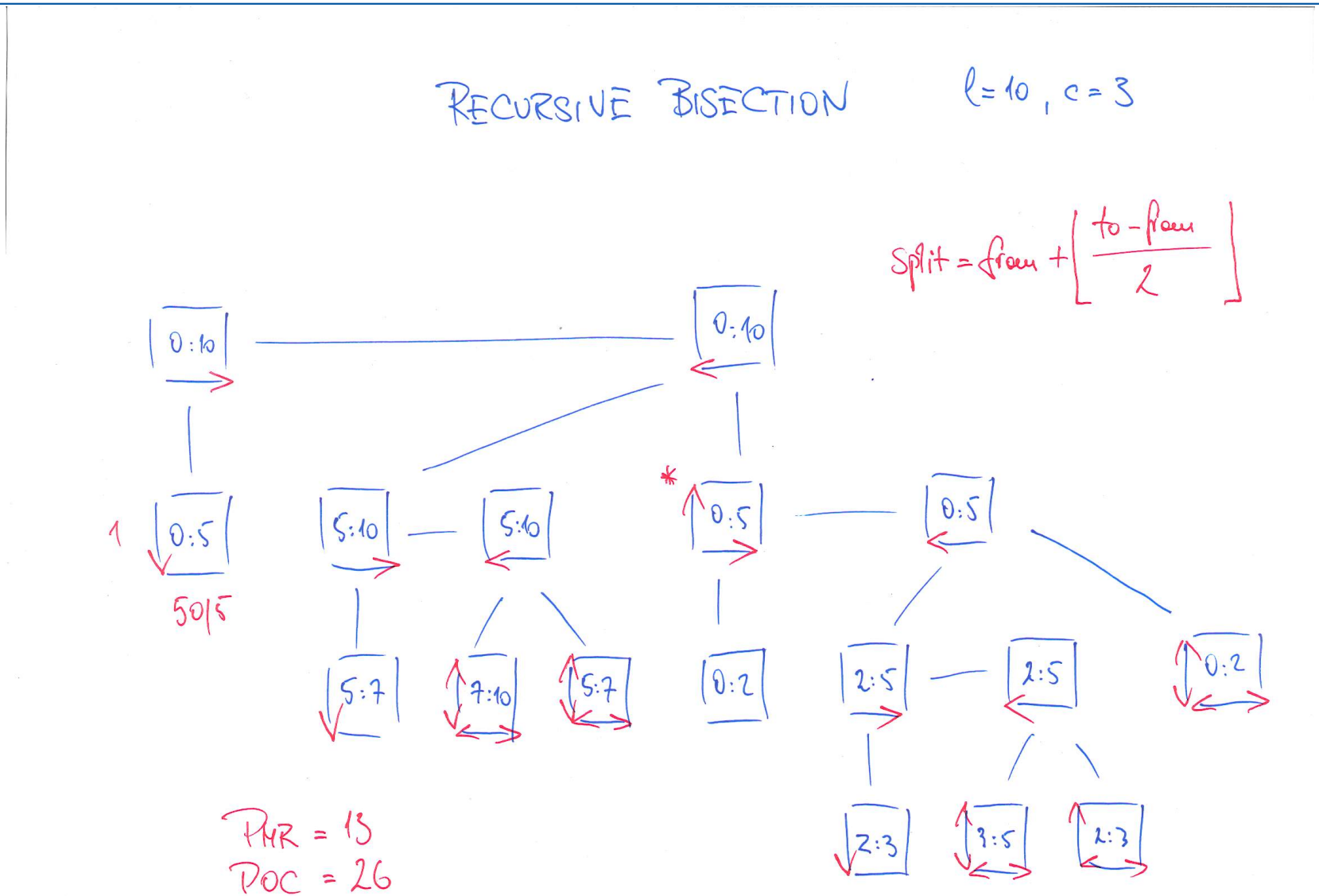
MEM=1110, OPS=1000

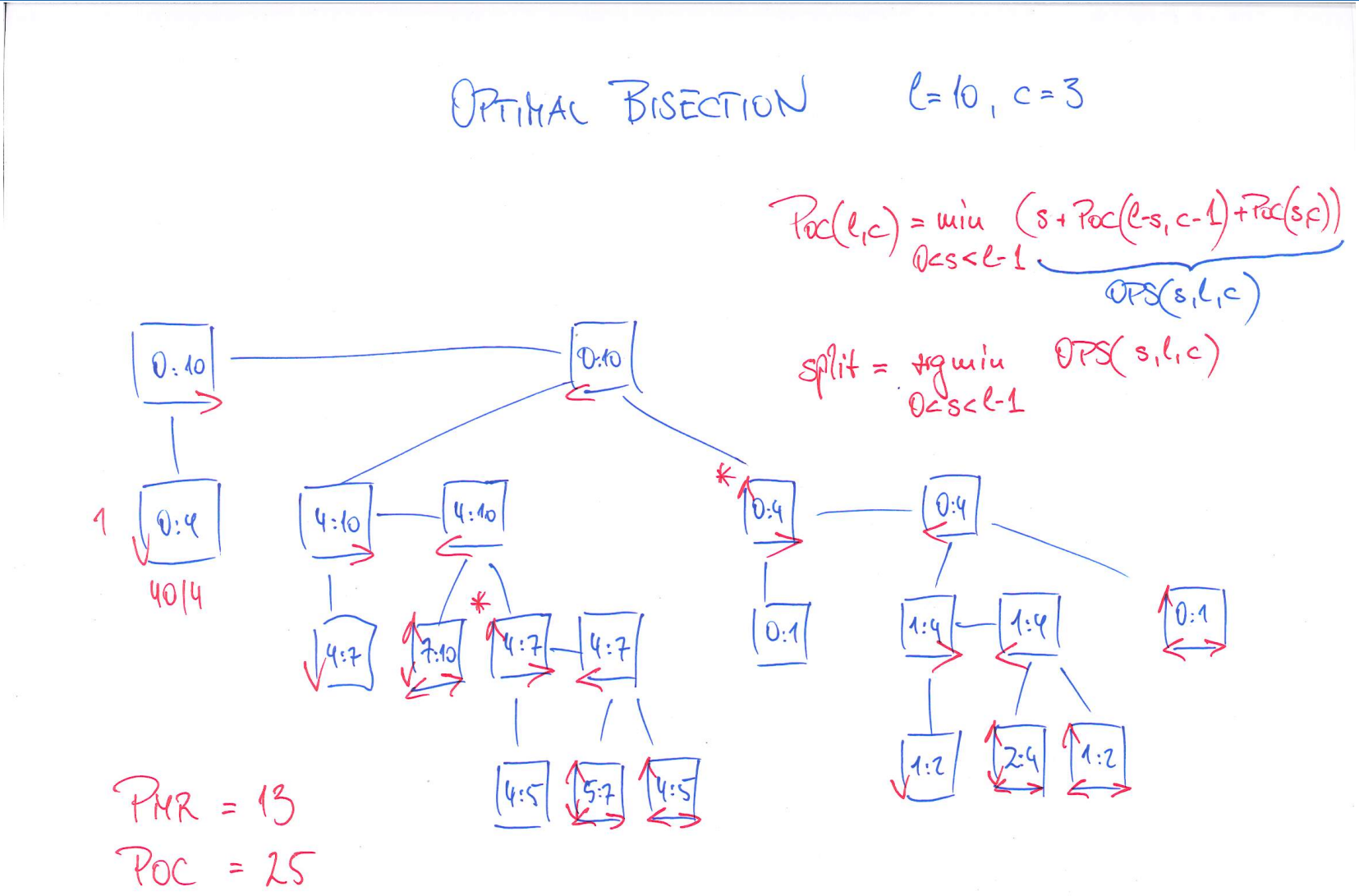
R=(1,0):



MEM=1120, OPS=1200

Largest Memory Increase (LMI) remains at $\text{R} = \mathbf{1}$ as $\text{R} = (1, 0)$ infeasible


RECURSIVE BISECTION
l=10, c=3
split = from + ⌊(to - from)/2⌋
0:10
0:10
1
0:5
5:10
5:10
*
0:5
0:5
5:7
7:10
5:7
0:2
2:5
2:5
0:2
2:3
3:5
2:3
50|5
PHR = 13
POC = 26

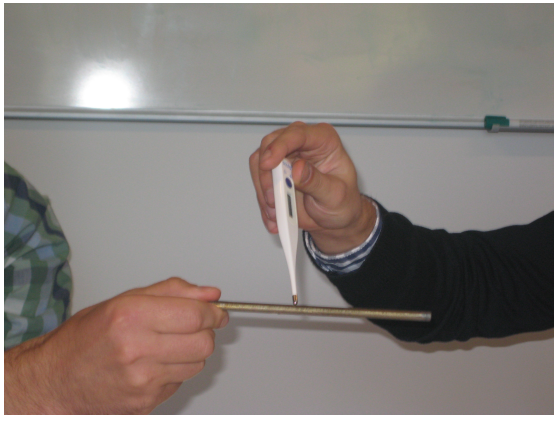

OPTIMAL BISECTION l=10, c=3
Poc(l,c) = min 0≤s<l-1 (s + Poc(l-s, c-1) + Poc(s,c))
OPS(s,l,c)
split = argmin 0≤s<l-1 OPS(s,l,c)
0:10
0:4
40/4
4:10
7:10
4:7
4:5
5:7
0:1
1:4
1:2
2:4
PMR = 13
POC = 25

1. Find $y = y(x,t)$ s.t.
$$\frac{dy}{dt} = c(y,x,t) \cdot \frac{d^2y}{dx^2} \ .$$
Discretization in space using n grid points; performing m time steps.
2. For given primal $\mathbb{R}^n \ni \mathbf{y}^m = \mathbf{y}^m(\mathbf{y}^0)$ and adjoints $\mathbf{y}^0_{(1)}$ and $\mathbf{y}^m_{(1)}$ find
$$\mathbf{y}^0_{(1)} := \mathbf{y}^0_{(1)} \left(\frac{d\mathbf{y}^m}{d\mathbf{y}^0}(\mathbf{y}^0) \right)^T \cdot \mathbf{y}^m_{(1)}$$

Initially we consider nonlinear c. Linear scenarios include self-adjointness as a special case.

Case Study: Diffusion

Primal: Motivation



We study the development of a quantity $y = y(x,t) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ over space $x \in [0,1]$ and time $t \in [0,1]$ modeled by the partial differential (diffusion) equation (pde)

$$\frac{dy}{dt} = c(y,x,t) \cdot \frac{d^2y}{dx^2}$$

with initial and boundary conditions

$$y(x, t = 0) = 1$$

and

$$y(x = 0, t) = y(x = 1, t) = 0,$$

respectively.

A given spatial discretization $\mathbf{x} \in \mathbb{R}^n$ yields a discrete solution $\mathbf{y}^1 \equiv \mathbf{y}(\mathbf{x}, 1) \in R^n$ as a function of the discrete initial condition $\mathbf{y}^0 \equiv \mathbf{y}(\mathbf{x}, 0) \in R^n$. We aim to compute the adjoint

$$\mathbb{R}^n \ni \mathbf{y}^0_{(1)} + = < \mathbf{y}^1_{(1)}, \frac{d\mathbf{y}^1}{d\mathbf{y}^0} > \ .$$

For example, the adjoint allows us to calibrate an unknown or uncertain initial condition efficiently to given obserations of the solution.

We use central finite differences to approximate the second derivative of y wrt. x. The domain $[0, 1] \ni x$ is divided into n sub-intervals of equal length

$$\Delta x = \frac{1}{n}$$

yielding a spatial discretization $\mathbf{x} = (x_j)_{j=0}^{n-1}$ with $n - 2$ inner points $x_1, \ldots, x_{n-2}$ in addition to the two boundary points $x_0 = 0$ and $x_{n-1} = 0$. Second spatial derivatives are approximated by second-order finite differences based on central finite difference approximation of the first derivatives at the center points

$$a_j \equiv \frac{x_{j-1} + x_j}{2} \quad \text{and} \quad b_j \equiv \frac{x_j + x_{j+1}}{2}$$

of the corresponding intervals.
From

$$\frac{\partial y}{\partial x}(a_j, t) \approx \frac{y(x_j, t) - y(x_{j-1}, t)}{\Delta x} = \frac{y_j - y_{j-1}}{\Delta x}$$

for $j = 1, \ldots, n-2$ and

$$\frac{\partial y}{\partial x}(b_j, t) \approx \frac{y(x_{j+1}, t) - y(x_j, t)}{\Delta x} = \frac{y_{j+1} - y_j}{\Delta x}$$

for $j = 1, \ldots, n-2$ follows

$$\frac{\partial^2 y}{\partial x^2}(x_j, t) \approx \frac{\frac{\partial y}{\partial x}(b_j, t) - \frac{\partial y}{\partial x}(a_j, t)}{\Delta x} = \frac{y_{j+1} - 2 \cdot y_j + y_{j-1}}{(\Delta x)^2}$$

for $j = 1, \ldots, n-2$ and hence the system of ordinary differential equations (ode)

$$\frac{\partial y_j}{\partial t} = g_j(\mathbf{y}) \,, \quad j = 0, \ldots, n-1 \,, \tag{2}$$

where

$$g_j = 0 \qquad j \in \{0, n-1\}$$

$$g_j = \frac{c(y_j, x, t)}{(\Delta x)^2} \cdot (y_{j+1} - 2 \cdot y_j + y_{j-1}) \qquad j \in \{1, \ldots, n-2\}.$$

The residual vanishes at both boundary points due to constant Dirichlet-type boundary conditions.

The derivative of y wrt. time t on left hand side of Equation (2) is approximated by backward finite differences yielding the *backward Euler method*. Decomposition of the unit time interval $[0, 1] \ni t$ into m sub-intervals of equal length

$$\Delta t = \frac{1}{m}$$

yields a temporal discretization $\mathbf{t} = (t_i)_{i=0}^m$ and $m + 1$ states $y_j^0, \ldots, y_j^m$ for $j = 0, \ldots, n$. From

$$\frac{\partial y_j^k}{\partial t} \approx \frac{y_j^k - y_j^{k-1}}{\Delta t}$$

for $j = 1, \ldots, n - 2$ follows

$$\frac{\mathbf{y}^k - \mathbf{y}^{k-1}}{\Delta t} = g(\mathbf{y}^k)$$

yielding the recurrence

$$f(\mathbf{y}^k, \mathbf{y}^{k-1}) \equiv \mathbf{y}^k - \mathbf{y}^{k-1} - \Delta t \cdot g(\mathbf{y}^k) = 0 \tag{3}$$

for $k = 1, \ldots, m$.

Solutions of the system of parameterized nonlinear equations in Equation (3) yield $\mathbf{y}^k \in \mathbb{R}^n$ for parameter $\mathbf{y}^{k-1} \in \mathbb{R}^n$ and $k = 1, \ldots, m$.

$\mathbf{y}^k := \text{Newton}(\mathbf{y}_0^k, \mathbf{y}^{k-1})$

Newton's algorithm for the solution of the parameterized system of nonlinear equation $f(\mathbf{y}, \mathbf{p}) = 0$ with state $\mathbf{y} = \mathbf{y}^k$, parameter $\mathbf{p} = \mathbf{y}^{k-1} \in \mathbb{R}^n$ and initial condition $\mathbf{y}_0^k$:

1: $0 < \epsilon \ll 1$
2: $j := 0$
3: **while** $|f(\mathbf{y}_j^k, \mathbf{y}^{k-1})| > \epsilon$ **do**
4: $\quad \mathbf{y}_{j+1}^k := \mathbf{y}_j^k - \frac{df}{d\mathbf{y}}(\mathbf{y}_j^k, \mathbf{y}^{k-1})^{-1} \cdot f(\mathbf{y}_j^k, \mathbf{y}^{k-1})$
5: $\quad j := j + 1$
6: **end while**

The Newton step $\Delta \mathbf{y}^k$ is computed as the solution of the system of linear equations

$$\frac{df}{d\mathbf{y}}(\mathbf{y}_j^k, \mathbf{y}^{k-1}) \cdot \Delta \mathbf{y}^k = f(\mathbf{y}_j^k, \mathbf{y}^{k-1}) \; .$$

We use standard LU decomposition followed by forward and backward substitution, e.g, for $A \cdot \mathbf{x} = \mathbf{b}$:

$$\begin{aligned} A &= L \cdot U \\ L \cdot \mathbf{y} &= \mathbf{b} \\ U \cdot \mathbf{x} &= \mathbf{y} \end{aligned}$$

Application of Newton's algorithm to Equation (3) yields the backward Euler method for solving Equation (2).

$\mathbf{y}^m := \text{Euler}(m, \mathbf{y}^0)$

Backward Euler method for approximating a solution $\mathbf{y}^m = \mathbf{y}(t=1)$ of the system of ode $\frac{d\mathbf{y}}{dt} = g(\mathbf{y})$ with m time steps for a given initial condition $\mathbf{y}^0 = \mathbf{y}(t=0)$:

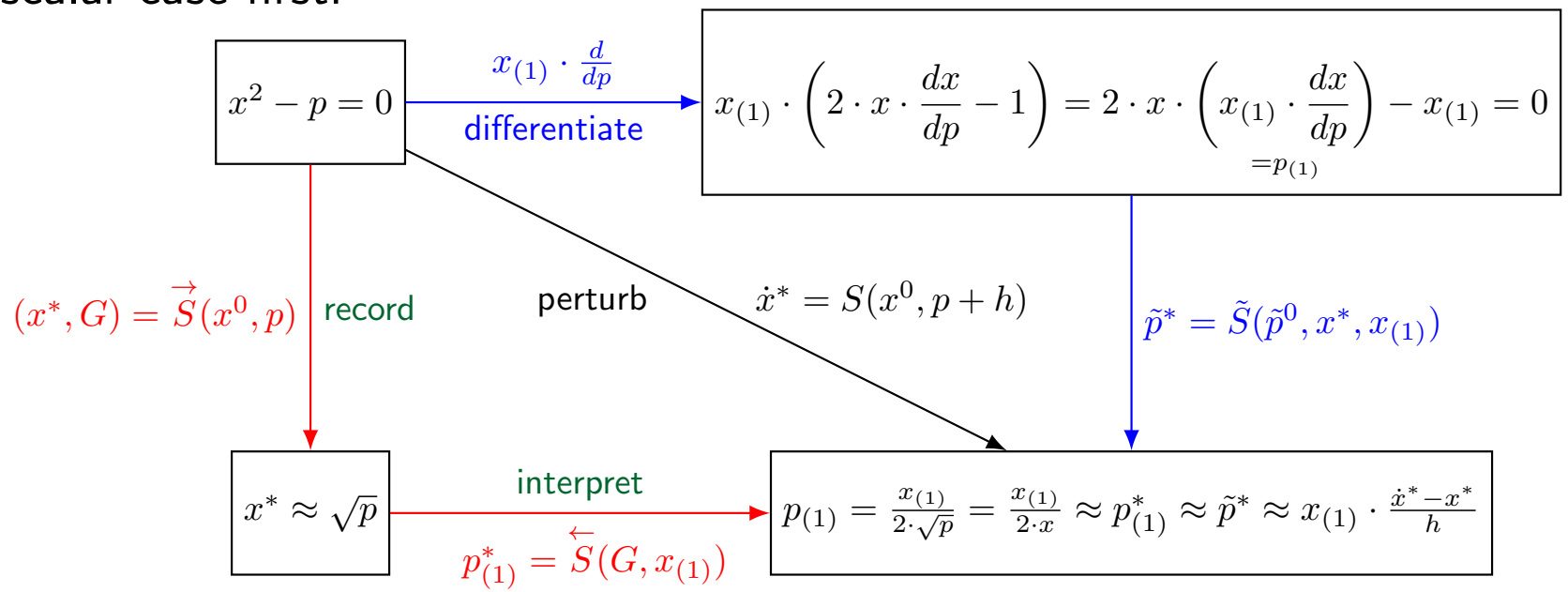
1: $0 < \epsilon \ll 1$,
2: $\Delta t := m^{-1}$
3: $k := 1$
4: **while** $k \leq m$ **do**
5: $\quad j := 0$
6: $\quad$ **while** $|\mathbf{y}_j^k - \mathbf{y}^{k-1} - \Delta t \cdot g(\mathbf{y}_j^k)| > \epsilon$ **do**
7: $\quad\quad \mathbf{y}_{j+1}^k := \mathbf{y}_j^k - (I - \Delta t \cdot \frac{dg}{d\mathbf{y}}(\mathbf{y}_j^k))^{-1} \cdot (\mathbf{y}_j^k - \mathbf{y}^{k-1} - \Delta t \cdot g(\mathbf{y}_j^k))$
8: $\quad\quad j := j + 1$
9: $\quad$ **end while**
10: $\quad k := k + 1$
11: **end while**

$\mathbf{y}^0_{(1)} := \text{Euler}^{\approx}_{(1)}(m, \mathbf{y}^0, \mathbf{y}^m_{(1)})$

Central finite difference approximation of the adjoint initial condition $\mathbf{y}^0_{(1)} = \mathbf{y}_{(1)}(t=0)$ for a given adjoint $\mathbf{y}^m_{(1)} = \mathbf{y}_{(1)}(t=1)$ of an approximate solution $\mathbf{y}^m$ of the system of ode $\frac{d\mathbf{y}}{dt} = g(\mathbf{y})$ obtained after m implicit Euler time steps for a given initial condition $\mathbf{y}^0$:

1: $i := 0$
2: **while** $i < n$ **do**
3: $\quad 0 < h \ll 1$
4: $\quad \mathbf{y}^{m-} := \text{Euler}(m, \mathbf{y}^0 - \mathbf{e}_i \cdot h)$
5: $\quad \mathbf{y}^{m+} := \text{Euler}(m, \mathbf{y}^0 + \mathbf{e}_i \cdot h)$
6: $\quad \mathbf{y}^0_{(1)} := \mathbf{y}^0_{(1)} + \mathbf{e}_i \cdot \left((\mathbf{y}^m_{(1)})^T \cdot (\mathbf{y}^{m+} - \mathbf{y}^{m-})/(2 \cdot h) \right)$
7: $\quad$ i:=i+1
8: **end while**

The key ingerdient of algorithmic adjoint diffusion is an adjoint version of the solver for the system of nonlinear equations. For motivation, we consider the scalar case first:



Assumptions: sole use of p (increment $p_{(1)}$ otherwise); no previous use of x (reset $x_{(1)}$ to zero otherwise)

Higher-level elementals improve efficiency, robustness, and scalability of algorithmic adjoints. We consider a sequence of steps leading us to elemental solvers for systems of nonlinear equations.

- Explicit Functions
  - Inner Vector Product
  - Matrix Vector Product
- Implicit Functions
  - Inner Vector Product
  - Linear Equation
  - System of Linear Equations
  - Nonlinear Equation
  - System of Nonlinear Equations

For

$$y = \boldsymbol{a}^T \cdot \mathbf{x}, \quad \boldsymbol{a}, \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}$$

we get

1: $y := 0$
2: **for** $i = 0, \ldots, n-1$ **do**
3: $\quad y := y + a_i \cdot x_i$
4: **end for**
5:
6: **for** $i = n-1, \ldots, 0$ **do**
7: $\quad a_{(1)_i} := a_{(1)_i} + x_i \cdot y_{(1)_j}$
8: $\quad x_{(1)_i} := x_{(1)_i} + a_i \cdot y_{(1)_j}$
9: **end for**
10: $y_{(1)} := 0$

Note: Unless explicitly stated otherwise we assume distinctly named variables to be unaliased.

1: $y := \boldsymbol{a}^T \cdot \mathbf{x}$ (DOT)

2: $\boldsymbol{a}_{(1)} := \boldsymbol{a}_{(1)} + \tilde{\boldsymbol{a}}_{(1)} = \boldsymbol{a}_{(1)} + \mathbf{x} \cdot y_{(1)}$ (AXPY)

$$\frac{dy}{d\boldsymbol{a}} = \frac{d\mathbf{x}^T \cdot \boldsymbol{a}}{d\boldsymbol{a}} = \mathbf{x}^T$$

$$\tilde{\boldsymbol{a}}_{(1)} = \frac{dy}{d\boldsymbol{a}}^T \cdot y_{(1)} = \mathbf{x} \cdot y_{(1)}$$

3: $\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + \tilde{\mathbf{x}}_{(1)} = \mathbf{x}_{(1)} + \boldsymbol{a} \cdot y_{(1)}$ (AXPY)

$$\frac{dy}{d\mathbf{x}} = \frac{d\boldsymbol{a}^T \cdot \mathbf{x}}{d\mathbf{x}} = \boldsymbol{a}^T$$

$$\tilde{\mathbf{x}}_{(1)} = \frac{dy}{d\mathbf{x}}^T \cdot y_{(1)} = \boldsymbol{a} \cdot y_{(1)}$$

4: $y_{(1)} := 0$

Note: We use overset tildes to denote the context-free adjoint as opposed to its context-sensitive version.

For $\mathbf{y} = A \cdot \mathbf{x}, \quad A \in \mathbb{R}^{m \times n}, \ \mathbf{x} \in \mathbb{R}^n, \ \mathbf{y} \in \mathbb{R}^m$ we get

1: **for** $j = 0, \ldots, m-1$ **do**
2: $y_j := 0$
3: **for** $i = 0, \ldots, n-1$ **do**
4: $y_j := y_j + A_{j,i} \cdot x_i$
5: **end for**
6: **end for**
7:
8: **for** $j = m-1, \ldots, 0$ **do**
9: **for** $i = n-1, \ldots, 0$ **do**
10: $A_{(1)_{j,i}} := A_{(1)_{j,i}} + x_i \cdot y_{(1)_j}$
11: $x_{(1)_i} := x_{(1)_i} + A_{j,i} \cdot y_{(1)_j}$
12: **end for**
13: $y_{(1)_j} := 0$
14: **end for**

1: $\mathbf{y} := A \cdot \mathbf{x}$ (GEMV)
2: $A_{(1)} := A_{(1)} + \tilde{A}_{(1)} = A_{(1)} + \mathbf{y}_{(1)} \cdot \mathbf{x}^T$ (GER)

$$
\begin{aligned}
y_j &= A_j \cdot \mathbf{x} \text{ (DOT)} \\
\tilde{A}^T_{(1)_j} &= \mathbf{x} \cdot y_{(1)_j} \quad (A_j \text{ is row vector}) \\
\tilde{A}_{(1)_j} &= y_{(1)_j} \cdot \mathbf{x}^T \\
\tilde{A}_{(1)} &= \mathbf{y}_{(1)} \cdot \mathbf{x}^T
\end{aligned}
$$

3: $\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + \tilde{\mathbf{x}}_{(1)} = \mathbf{x}_{(1)} + A^T \cdot \mathbf{y}_{(1)}$ (GEMV)

$$
\begin{aligned}
y_j &= A_j \cdot \mathbf{x} \text{ (DOT)} \\
\tilde{\mathbf{x}}_{(1)_j} &= A_j^T \cdot y_{(1)_j} \\
\tilde{\mathbf{x}}_{(1)} &= A^T \cdot \mathbf{y}_{(1)}
\end{aligned}
$$

4: $\mathbf{y}_{(1)} := 0$

For

$$a \cdot x = b, \quad a, x, b \in \mathbb{R}$$

we get

1: $x := \frac{b}{a}$
2:
3: $b_{(1)} := b_{(1)} + \frac{1}{a} \cdot x_{(1)}$
4: $a_{(1)} := a_{(1)} - \frac{x}{a} \cdot x_{(1)}$
5: $x_{(1)} := 0$

1: $b_{(1)} := b_{(1)} + \tilde{b}_{(1)} = b_{(1)} + \frac{1}{a} \cdot x_{(1)}$

$$\frac{d(a \cdot x)}{db} = a \cdot \frac{dx}{db} = \frac{db}{db} = 1$$
$$\tilde{b}_{(1)} = \frac{dx}{db} \cdot x_{(1)} = \frac{1}{a} \cdot x_{(1)}$$

2: $a_{(1)} := a_{(1)} + \tilde{a}_{(1)} = a_{(1)} - \frac{x}{a} \cdot x_{(1)}$

$$\frac{d(a \cdot x)}{da} = \frac{d(a \cdot x)}{dx} \cdot \frac{dx}{da} + \frac{\partial(a \cdot x)}{\partial a} = a \cdot \frac{dx}{da} + x = \frac{db}{da} = 0$$
$$\tilde{a}_{(1)} = \frac{dx}{da} \cdot x_{(1)} = -\frac{x}{a} \cdot x_{(1)}$$

3: $x_{(1)} := 0$

For

$$A \cdot \mathbf{x} = \mathbf{b}, \quad A \in \mathbb{R}^{n \times n}, \ \mathbf{b}, \mathbf{x} \in \mathbb{R}^n$$

we get

1: $\mathbf{x} := A^{-1} \cdot \mathbf{b}$ (SOLVE)
2:
3: $\mathbf{b}_{(1)} := \mathbf{b}_{(1)} + A^{-T} \cdot \mathbf{x}_{(1)}$ (SOLVE)
4: $A_{(1)} := A_{(1)} - \mathbf{b}_{(1)} \cdot \mathbf{x}^T$ (GER)
5: $\mathbf{x}_{(1)} := 0$

For

$$f(y, \mathbf{p}) = 0, \quad \mathbf{p} \in \mathbb{R}^l, \; y = y(\mathbf{p})$$

we get

$$\frac{df}{d\mathbf{p}} = \frac{\partial f}{\partial \mathbf{p}} + \frac{df}{dy} \cdot \frac{dy}{d\mathbf{p}} = 0 \, .$$

Transposition yields

$$\left(\frac{df}{d\mathbf{p}}\right)^T = \left(\frac{\partial f}{\partial \mathbf{p}}\right)^T + \frac{df}{dy} \cdot \left(\frac{dy}{d\mathbf{p}}\right)^T = 0$$

and hence

$$\tilde{\mathbf{p}}_{(1)} = \frac{dy}{d\mathbf{p}} \cdot y_{(1)} = -\frac{\frac{\partial f}{\partial \mathbf{p}}}{\frac{df}{dy}} \cdot y_{(1)}$$

$$\mathbf{p}_{(1)} := \mathbf{p}_{(1)} + \tilde{\mathbf{p}}_{(1)}$$

Differentiation of the parameterized system of nonlinear equations

$$\mathbb{R}^n \ni F(\mathbf{y}(\mathbf{p}), \mathbf{p}) = 0$$

at the solution $\mathbf{y}_* = \mathbf{y}_*(\mathbf{p}) \in \mathbb{R}^n$ with respect to parameters $\mathbf{p} \in \mathbb{R}^l$ yields

$$\frac{dF}{d\mathbf{p}} = \frac{\partial F}{\partial \mathbf{p}} + \frac{dF}{d\mathbf{y}} \cdot \frac{d\mathbf{y}}{d\mathbf{p}} = 0.$$

Transposition yields

$$\left(\frac{dF}{d\mathbf{p}}\right)^T = \left(\frac{\partial F}{\partial \mathbf{p}}\right)^T + \left(\frac{d\mathbf{y}}{d\mathbf{p}}\right)^T \cdot \left(\frac{dF}{d\mathbf{y}}\right)^T = 0$$

and hence

$$\left(\frac{\partial F}{\partial \mathbf{p}}\right)^T \cdot \mathbf{z} = \left(\frac{d\mathbf{y}}{d\mathbf{p}}\right)^T \cdot \underbrace{\left(\frac{dF}{d\mathbf{y}}\right)^T \cdot \mathbf{z}}_{:=-\mathbf{y}_{(1)}}.$$

Hence, computation of the adjoint $\mathbf{p}_{(1)} = \left(\frac{d\mathbf{y}}{d\mathbf{p}}\right)^T \cdot \mathbf{y}_{(1)}$ amounts to the solution of the linear system

$$\frac{dF}{d\mathbf{y}}^T \cdot \mathbf{z} = -\mathbf{y}_{(1)}$$

followed by a single evaluation of $F_{(1)}$ to obtain

$$\mathbf{p}_{(1)} = \frac{\partial F}{\partial \mathbf{p}}^T \cdot \mathbf{z}.$$

The Jacobian $\frac{dF}{d\mathbf{y}}$ requires $O(n)$ evaluations of the tangent residual $F^{(1)}$ while potential sparsity should be exploited.

$\mathbf{y}_{(1)}^{k-1} := \text{Newton}_{(1)}(\mathbf{y}_0^k, \mathbf{y}^{k-1}, \mathbf{y}_{(1)}^k)$

Symbolic adjoint of Newton's algorithm for the solution of the parameterized system of nonlinear equation $f(\mathbf{y}, \mathbf{p}) = 0$ with state $\mathbf{y} = \mathbf{y}^k$, parameter $\mathbf{p} = \mathbf{y}^{k-1} \in \mathbb{R}^n$, initial condition $\mathbf{y}_0^k$, and adjoint solution $\mathbf{y}_{(1)} = \mathbf{y}_{(1)}^k$

1: $0 < \epsilon \ll 1$
2: $j := 0$
3: **while** $|f(\mathbf{y}_j^k, \mathbf{y}^{k-1})| > \epsilon$ **do**
4: $\quad \mathbf{y}_{j+1}^k := \mathbf{y}_j^k - \frac{df}{d\mathbf{y}}(\mathbf{y}_j^k, \mathbf{y}^{k-1})^{-1} \cdot f(\mathbf{y}_j^k, \mathbf{y}^{k-1})$
5: $\quad j := j + 1$
6: **end while**
7: $\mathbf{y}^k = \mathbf{y}_{j-1}^k$
8: $\mathbf{z} := \frac{df}{d\mathbf{y}}(\mathbf{y}^k, \mathbf{y}^{k-1})^{-T} \cdot (-\mathbf{y}_{(1)}^k)$
9: $\mathbf{y}_{(1)}^{k-1} := \frac{df}{d\mathbf{p}}(\mathbf{y}^k, \mathbf{y}^{k-1})^T \cdot \mathbf{z}$

$\mathbf{y}^0_{(1)} := \text{Euler}_{(1)}(m, \mathbf{y}^0, \mathbf{y}^m_{(1)})$

Algorithmic adjoint backward Euler method for computing the adjoint initial condition $\mathbf{y}^0_{(1)}$ for a given adjoint $\mathbf{y}^m_{(1)}$ of an approximate solution $\mathbf{y}^m$ of the system of ode $\frac{d\mathbf{y}}{dt} = g(\mathbf{y})$ obtained after m implicit Euler time steps for a given initial condition $\mathbf{y}^0$:

$0 < \epsilon \ll 1,$
$\Delta t := m^{-1}$

$k := 1$
while $k \leq m$ **do**
  $j := 0$
  while $|\mathbf{y}^k_j - \mathbf{y}^{k-1} - \Delta t \cdot g(\mathbf{y}^k_j)| > \epsilon$ **do**
    $\mathbf{y}^k_{j+1} := \mathbf{y}^k_j - (I - \Delta t \cdot \frac{dg}{d\mathbf{y}}(\mathbf{y}^k_j))^{-1} \cdot (\mathbf{y}^k_j - \mathbf{y}^{k-1} - \Delta t \cdot g(\mathbf{y}^k_j))$
    $j := j + 1$
  end while
  $k := k + 1$

end while

$k := m$
while $k \geq 1$ **do**
$\mathbf{y}_{(1)}^{k-1} := (I - \Delta t \cdot \frac{dg}{d\mathbf{y}}(\mathbf{y}_j^k))^{-T} \cdot \mathbf{y}_{(1)}^k$
$k := k - 1$
end while

If c is linear in y then so is the entire residual $g(\mathbf{y})$. Hence, it can be expanded into a first-order Taylor series at point $\mathbf{y} \equiv 0$ ($y_j = 0$ for $j = 1, \ldots, n-2$) as follows:

$$\frac{d\mathbf{y}}{dt} = \underbrace{g(0, \mathbf{x}, \mathbf{t})}_{=0} + \frac{dg}{d\mathbf{y}}(\mathbf{x}, \mathbf{t}) \cdot (\mathbf{y} - 0) = \frac{dg}{d\mathbf{y}}(\mathbf{x}, \mathbf{t}) \cdot \mathbf{y}.$$

The residual at $\mathbf{y} \equiv 0$ vanishes identically. Linearity of the residual in $\mathbf{y}$ implies the independence of its first derivative from $\mathbf{y}$, that is, the Jacobian $\frac{dg}{d\mathbf{y}} = \frac{dg}{d\mathbf{y}}(\mathbf{x}, \mathbf{t})$ is independent of the state of the system $\mathbf{y}$ at any time $t_i, i = 0, \ldots, m$.

Hence,

$$\frac{\mathbf{y}^{k+1} - \mathbf{y}^k}{\Delta t} = \frac{dg}{d\mathbf{y}}(\mathbf{x}, \mathbf{t}) \cdot \mathbf{y}^{k+1}$$

where the Jacobian $\frac{dg}{d\mathbf{y}}(\mathbf{x}, \mathbf{t})$ of the residual can be computed as the residual $g(\mathbf{y}, \mathbf{x}, \mathbf{t})$ evaluated at the Cartesian basis directions $\mathbf{y} = \mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$ due to linearity.

Linear c II

The solution of the resulting system of linear equations

$$(\Delta t \cdot \frac{dg}{d\mathbf{y}}(\mathbf{x}, \mathbf{t}) - I) \cdot \mathbf{y}^{k+1} = -\mathbf{y}^k$$

is obtained by a single LU-decomposition followed by forward and backward substitutions during each backward Euler step.

We are interested in adjoints of $\mathbf{y}^m$ with respect to $\mathbf{y}^0$ and the parameter $C = C(\mathbf{x}, \mathbf{t}) \in \mathbb{R}^{n \times m}$:

$$\begin{pmatrix} \tilde{\mathbf{y}}^0_{(1)} \\ \tilde{C}_{(1)} \end{pmatrix} = < \mathbf{y}_{(1)}, \frac{d\mathbf{y}^m}{d(\mathbf{y}^0, C)} >$$

The residual becomes $g = g(C, \mathbf{y}) : \mathbb{R}^{n \times m} \times \mathbb{R}^n \to \mathbb{R}^n$. It turns out to be linear both in $\mathbf{y}$ and C. Backward finite difference discretization in time yields the backward Euler method:

1: $A = A(C) := \mathsf{dgdy}(C) =< \frac{d\mathsf{R}}{d\mathbf{y}}(C), I >= \left(\mathsf{R}(C, \mathbf{e}_i)\right)_{i=1}^{n}$
2: $A := L + U - I = \mathsf{LU}(\Delta t \cdot A - I)$
3: **for** $j = 1, \ldots, m$ **do**
4: $\quad \mathbf{y} := -\mathbf{y}$
5: $\quad \mathbf{y} := \mathsf{FB}(A, \mathbf{y}) = U^{-1} \cdot (L^{-1} \cdot (\mathbf{y}))$
6: **end for**

with in-place LU factorization $\mathsf{LU} : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$, forward-backward substitution $\mathsf{FB} : \mathbb{R}^{n \times n} \times \mathbb{R}^n \to \mathbb{R}^n$ and identity $I \in \mathbb{R}^{n \times n}$.

1: $A := \mathsf{dgdy}(C)$
2: $A := \Delta t \cdot A - I$
3: $A := \mathsf{LU}(A)$
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad \mathbf{y} := -\mathbf{y}$
6: $\quad \mathsf{PUSH}(\mathbf{y});\ \mathbf{y} := \mathsf{FB}(A, \mathbf{y})$
7: **end for**
8:
9: **for** $j = m, \ldots, 1$ **do**
10: $\quad \mathsf{POP}(\mathbf{y});\ (A_{(1)}, \mathbf{y}_{(1)}) := \mathsf{FB}_{(1)}(A, A_{(1)}, \mathbf{y}, \mathbf{y}_{(1)})$
11: $\quad \mathbf{y}_{(1)} := -\mathbf{y}_{(1)}$
12: **end for**
13: $A_{(1)} := \mathsf{LU}_{(1)}(A, A_{(1)})$
14: $A_{(1)} := \Delta t \cdot A_{(1)}$
15: $C_{(1)} := \mathsf{dgdy}_{(1)}(C, C_{(1)}, A_{(1)})$

Case Study: Diffusion

Adjoint: Elemental Linear Solver

1: $A := \mathsf{dgdy}(C)$
2: $A := \Delta t \cdot A - I$
3: $A := \mathsf{LU}(A)$
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad \mathbf{y} := -\mathbf{y}$
6: $\quad \mathbf{y} := \mathsf{FB}(A, \mathbf{y}) = U^{-1} \cdot (L^{-1} \cdot (\mathbf{y}))$
7: $\quad \mathsf{PUSH}(\mathbf{y})$
8: **end for**
9:
10: **for** $j = 1, \ldots, m$ **do**
11: $\quad \mathsf{POP}(\mathbf{y})$
12: $\quad \mathbf{y}_{(1)} := L^{-T} \cdot (U^{-T} \cdot (\mathbf{y}_{(1)}))$
13: $\quad \mathbf{y}_{(1)} := -\mathbf{y}_{(1)}$
14: $\quad A_{(1)} += \mathbf{y}_{(1)} \cdot \mathbf{y}^T$
15: **end for**
16: $A_{(1)} := \Delta t \cdot A_{(1)}$
17: $C_{(1)} := \mathsf{dgdy}_{(1)}(C, C_{(1)}, A_{(1)})$

Adjoint: Simplification due to Linearity of R in C

Note that $C_{(1)} := \mathsf{dgdy}_{(1)}(C_{(1)}, A_{(1)})$ independent of C yielding

1: $A := \mathsf{dgdy}(C)$
2: $A := \Delta t \cdot A - I$
3: $L + U - I := \mathsf{LU}(A)$
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad \mathbf{y} := -\mathbf{y}$
6: $\quad \mathbf{y} := \mathsf{FB}(L, U, \mathbf{y})$
7: $\quad \mathsf{PUSH}(\mathbf{y})$
8: **end for**
9:
10: **for** $j = 1, \ldots, m$ **do**
11: $\quad \mathsf{POP}(\mathbf{y})$
12: $\quad \mathbf{y}_{(1)} := L^{-T} \cdot (U^{-T} \cdot (\mathbf{y}_{(1)}))$
13: $\quad \mathbf{y}_{(1)} := -\mathbf{y}_{(1)}$
14: $\quad C_{(1)} := C_{(1)} + < \mathbf{y}_{(1)}, \frac{dg}{dC}(\Delta t \cdot \mathbf{y}) >$
15: **end for**

For

$$\frac{dy}{dt} = \frac{d^2y}{dx^2}$$

the algorithmic adjoint simplifies to

1: $A :=$ dgdy
2: $A := \Delta t \cdot A - I$
3: $L + U - I :=$ LU(A)
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad \mathbf{y}_{(1)} := -\mathbf{y}_{(1)}$
6: $\quad \mathbf{y}_{(1)} :=$ FB$(L, U, \mathbf{y}_{(1)})$
7: **end for**

due to selfadjointness of the diffusion operator as $A = A^T$.

Selfadjointness for independent $c \in \mathbb{R}$

1: $A := \mathsf{dgdy}(c)$
2: $A := \Delta t \cdot A - I$
3: $L + U - I := \mathsf{LU}(A)$
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad \mathbf{y} := -\mathbf{y}$
6: $\quad \mathbf{y} := \mathsf{FB}(L, U, \mathbf{y})$
7: $\quad \mathsf{PUSH}(\mathbf{y})$
8: **end for**
9:
10: **for** $j = 1, \ldots, m$ **do**
11: $\quad \mathsf{POP}(\mathbf{y})$
12: $\quad \mathbf{y}_{(1)} := -\mathbf{y}_{(1)}$
13: $\quad \mathbf{y}_{(1)} := \mathsf{FB}(L, U, \mathbf{y}_{(1)})$
14: $\quad c_{(1)} := c_{(1)} + < \mathbf{y}_{(1)}, \frac{dR}{dc}(\Delta t \cdot \mathbf{y}) >$
15: **end for**

Case Study: Diffusion

Selfadjointness for time-dependent $c = c(t) \rightarrow \mathbf{c} = (c_j)_{j=0}^m$

1: **for** $j = 1, \ldots, m$ **do**
2: $A := \mathsf{dgdy}(c_j)$
3: $A := \Delta t \cdot A - I$
4: $L + U - I := \mathsf{LU}(A)$
5: $\mathsf{PUSH}(L + U - I)$
6: $\mathbf{y} := -\mathbf{y}$
7: $\mathbf{y} := \mathsf{FB}(L, U, \mathbf{y})$
8: $\mathsf{PUSH}(\mathbf{y})$
9: **end for**
10:
11: **for** $j = 1, \ldots, m$ **do**
12: $\mathsf{POP}(\mathbf{y})$
13: $\mathsf{POP}(L + U - I)$
14: $\mathbf{y}_{(1)} := -\mathbf{y}_{(1)}$
15: $\mathbf{y}_{(1)} := \mathsf{FB}(L, U, \mathbf{y}_{(1)})$
16: ${c_{(1)}}_j := {c_{(1)}}_j + < \mathbf{y}_{(1)}, \frac{dg}{d\mathbf{c}}(\Delta t \cdot \mathbf{y}) >$
17: **end for**

References I


M. Beckers, V. Mosenkis, and U. Naumann.
Adjoint mode computations of subgradients for McCormick relaxations.
In *Recent Advances in Algorithmic Differentiation*, number 87 in Lecture Notes in Computational Science and Engineering (LNCSE), pages 103–113. Springer, 2012.


A. Griewank and U. Naumann.
Accumulating Jacobians as chained sparse matrix products.
Mathematical Programming, 95(3):555–571, 2003.


R. Hannemann-Tamás, J. Tillack, M. Schmitz, M. Förster, J. Wyes, K. Nöh, E. von Lieres, U. Naumann, W. Wiechert, and W. Marquardt.
First- and second-order parameter sensitivities of a metabolically and isotopically non-stationary biochemical network model.
In *Electronic Proceedings of the 9th International Modelica Conference, Munich, Sep 3-5, 2012*. Modelica Association, 2012.

References II

J. Lotz, M. Schwalbach, and U. Naumann.
A case study in adjoint sensitiviy analysis of parameter calibration.
In Procedia Computer Science, editor, *International Conference on Computational Science (ICCS 2016)*, 2016.

U. Merkel, J. Riehme, and U. Naumann.
Reverse engineering of initial and boundary conditions with Telemac and algorithmic differentiation.
WASSERWIRTSCHAFT, 103(12):22–27, 2013.

U. Naumann.
Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph.
Mathematical Programming, 99(3):399–421, 2004.

U. Naumann.
Optimal Jacobian accumulation is NP-complete.
Mathematical Programming, 112(2):427–441, 2008.

References III

U. Naumann and J. du Toit.
Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance.
Journal of Computational Finance, 2016.

U. Naumann and O. Schenk, editors.
Combinatorial Scientific Computing, Computational Science Series.
Chapman & Hall / CRC Press, Taylor and Francis Group, 2012.

M. Towara, M. Schanen, and U. Naumann.
MPI-parallel discrete adjoint OpenFOAM.
Procedia Computer Science, 51:19–28, 2015.

J. Ungermann, J. Blank, J. Lotz, K. Leppkes, Lars Hoffmann, T. Guggenmoser, M. Kaufmann, P. Preusse, U. Naumann, and M. Riese.
A 3-d tomographic retrieval approach with advection compensation for the air-borne limb-imager GLORIA.
Atmospheric Measurement Techniques, 4(11):2509–2529, 2011.

References IV


V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. Antonopoulos, N. Bellas, S. Lalisa, and U. Naumann.
Towards automatic significance analysis for approximate computing.
In *International Symposium on Code Generation and Optimization*, pages 182–193. IEEE/ACM, 2016.


A. Vlasenko, P. Korn, J. Riehme, and U. Naumann.
Estimation of data assimilation error: A shallow-water model study.
Monthly Weather Review, 142:2502–2520, 2014.