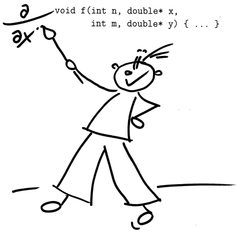


(Adjoint) Algorithmic Differentiation [(A)AD]

A Hands-On Introduction

Uwe Naumann

LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, Germany

and

The Numerical Algorithms Group Ltd., Oxford, UK

Progress

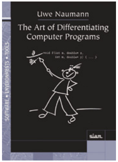
The Art of Differentiating Computer Programs

Bumping?

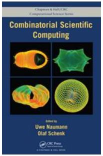
...

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



- inspired by sensitivity analysis, uncertainty quantification, calibration / optimization
- finite differences (first- and second-order), symbolic, algorithmic
- implementation by overloading, source trafo, hand-coding
- real code
- sensitivity analysis as modelling and software engineering tool
- what matters
  - user expertise
  - tool quality
  - tool sustainability and support

Let $y = F(\mathbf{x}),\ F : \mathbb{R}^n \rightarrow \mathbb{R}$:

1. tangent AD: $y^{(1)} = \nabla F \cdot \mathbf{x}^{(1)} \Rightarrow \nabla F$ at $O(n) \cdot \text{Cost}(F)$
2. adjoint AD: $\mathbf{x}_{(1)} = \nabla F^T \cdot y_{(1)} \Rightarrow \nabla F$ at $O(1) \cdot \text{Cost}(F)$
3. 2nd-order tangent AD: $y^{(1,2)} = {\mathbf{x}^{(1)}}^T \cdot \nabla^2 F \cdot \mathbf{x}^{(2)} \Rightarrow \nabla^2 F$ at $O(n^2) \cdot \text{Cost}(F)$
4. 2nd-order adjoint AD: $\mathbf{x}^{(2)}_{(1)} = y_{(1)} \cdot \nabla F^2 \cdot \mathbf{x}^{(2)} \Rightarrow \nabla^2 F$ at $O(n) \cdot \text{Cost}(F)$ and $\nabla^2 F \cdot \mathbf{x}^{(2)}$ at $O(1) \cdot \text{Cost}(F)$

Aims of this Course

You will learn how to

- implement tangent and adjoint versions of a Monte Carlo / Euler-Maruyama solver for parameterized scalar SDEs
- ensure feasibility of adjoint Monte Carlo simulation through pathwise adjoints
- “get serious” with AAD (tools, checkpointing, symbolic adjoints, design patterns, ...)

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.

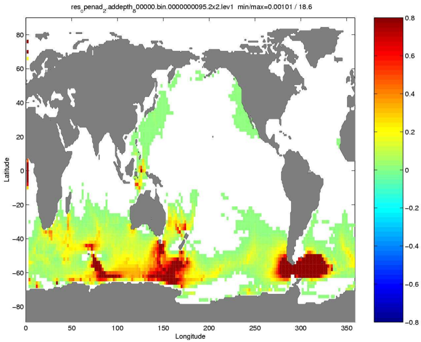
We are interested in sensitivities of the final state $\mathbb{E}(x)$ wrt. $\mathbf{p}$.

Race (Euler-Maruyama $m = 10^4, n = 10^2, l = 10^2$)

- primal: `primal.cpp` (inspect)
- bumping: `fd.cpp` (inspect)
- tangent: `tangent.cpp` (live)
- vector tangent: `tangent_vector.cpp` (inspect)
- adjoint: `adjoint.cpp` (live)
- pathwise adjoint: `adjoint_pathwise.cpp` (inspect)

mode	run time (s)	memory size (b)	accuracy
bump	$10.9 \sim O(l)$	$\sim P$	-
tangent	$21.5 \sim O(2 \cdot l)$	$\sim 2 \cdot P$	+
vector tangent	$13.6 \sim O(2 \cdot l)$	$\sim P + P \cdot l$	+
adjoint	$0.3 \sim O(1)$	$\sim 2 \cdot P + 2 \cdot m \cdot n \cdot 8$	+
pathwise adjoint	$0.5 \sim O(1)$	$\sim 2 \cdot P + 2 \cdot (m + n) \cdot 8$	+

where P denotes the memory requirement of the primal code.



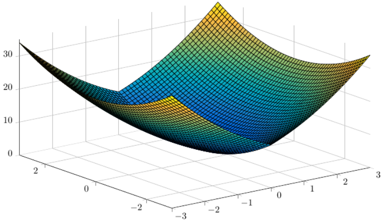
MITgcm, (EAPS, MIT)

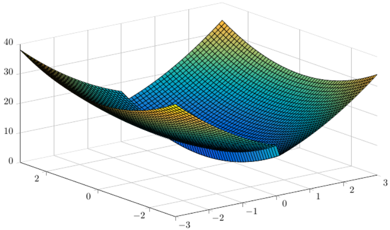
in collaboration with ANL, MIT, Rice, UColorado

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 :-)

- continuity
- differentiability?





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

Progress

1. The given implementation of $F : \mathbb{R}^n \rightarrow \mathbb{R}^m : \mathbf{y} = F(\mathbf{x})$, 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.

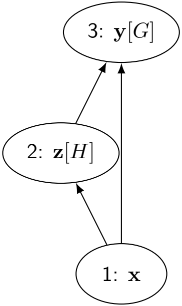
3. A linearized SAC (lSAC) is obtained by augmenting the elemental assignments with computations of the local partial derivatives $d_{j,i}$.
4. The SAC induces a directed acyclic graph (DAG) $G = G(F) = (V, E)$ with integer vertices $V = \{0, \ldots, n+q\}$ and edges $V \times V \supseteq E = \{(i,j) : i \prec j\}$.
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 (lDAG) is obtained by attaching the $d_{j,i}$ to the corresponding edges (i,j) in the DAG.

First-Order (A)AD

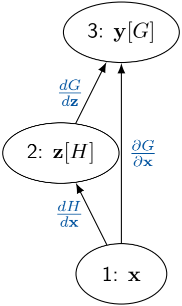
Prerequisites: Chain Rule on IDAG

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

DAG:



IDAG:



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

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

  Proof by reduction from ENSEMBLE COMPUTATION

- 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

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

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

Tangents

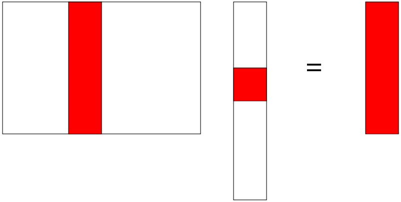
Mathematician's View

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

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

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 $\mathbb{R}^n$.

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 on which active outputs depend 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:

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

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

$$\begin{pmatrix} v_i \\ v_i^{(1)} \end{pmatrix} := \begin{pmatrix} x_i \\ x_i^{(1)} \end{pmatrix} \qquad \text{(seed)}$$

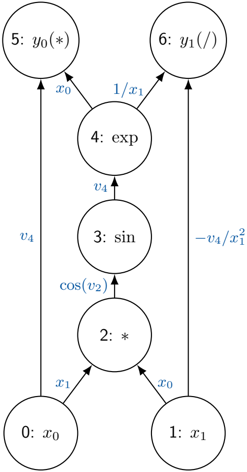
For $i = n, \ldots, q-1$

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

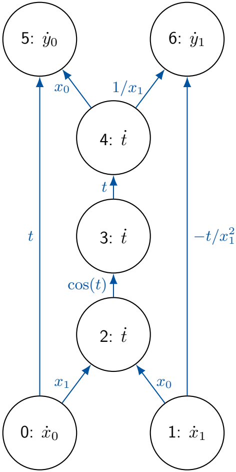
For $i = 0, \ldots, m-1$

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

Tangent assignments augment primal ...



$$\dot{t} = \dot{x}_0 \cdot x_1 + x_0 \cdot \dot{x}_1$$
$$t := x_0 \cdot x_1$$
$$\dot{t} = \cos(t) \cdot \dot{t}$$
$$t := \sin(t)$$
$$t := e^t$$
$$\dot{t} = t \cdot \dot{t}$$
$$\dot{y}_0 = \dot{x}_0 \cdot t + x_0 \cdot \dot{t}$$
$$y_0 := x_0 \cdot t$$
$$\dot{y}_1 = \dot{t}/x_1 - t \cdot \dot{x}_1/x_1^2$$
$$y_1 := t/x_1$$



Euler-Maruyama live ...

Vector Tangents

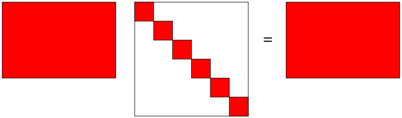
Computer Scientist's View (Simplified)

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!

Euler-Maruyama live ...

Adjoints

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

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

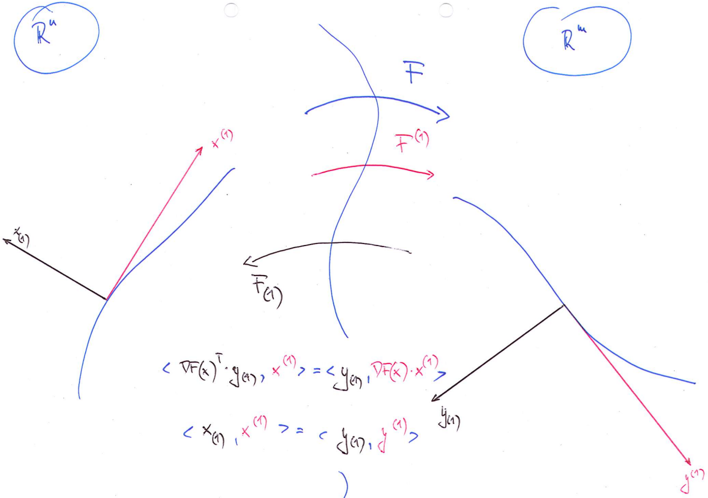
Theorem

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

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

Note invariant at each point in the program execution $\rightarrow$ validation.

Adjoints



$$\langle DF(x)^T \cdot y_{(1)}, x^{(1)} \rangle = \langle y_{(1)}, DF(x) \cdot x^{(1)} \rangle$$

$$\langle x_{(1)}, x^{(1)} \rangle = \langle y_{(1)}, y^{(1)} \rangle$$

E.g. $y = \sin(x)$; $y = x_0 \cdot x_1$; ...

Adjoints

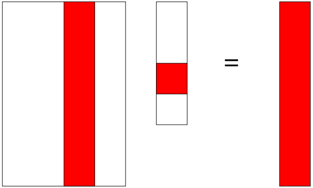
Mathematician's View

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

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

defines an adjoint directional derivative alongside with the function value:

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



... 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:

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

... 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.

1. Augmented Primal (enable data flow reversal)

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

$$v_i := x_i$$

$$\text{record } i \in V \ (v_{i_{(1)}} := x_{i_{(1)}})$$

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

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

$$\text{record } i \in V \ (v_{i_{(1)}} := 0)$$

$$\text{For } j \prec i : \text{record } (i,j) \in E \ (d_{j,i} := \frac{d\varphi_i(v_k)_{k \prec i}}{dv_j})$$

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

$$y_i := v_{n+p+i}$$

2. Adjoint

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

$$v_{n+p+i_{(1)}} := y_{i_{(1)}}$$

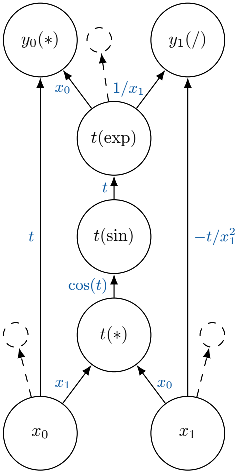
For $i = q-1, \ldots, n$

$$\forall\ (j,i) \in E : v_{j_{(1)}} := v_{j_{(1)}} + v_{i_{(1)}} \cdot d_{i,j}$$

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

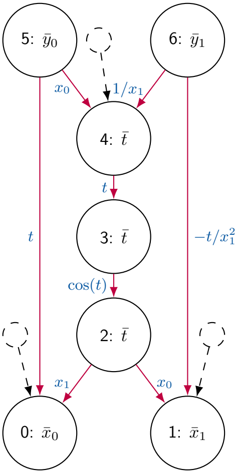
$$x_{i_{(1)}} := v_{i_{(1)}}$$

Mind overwrites and context ...



$t := x_0 \cdot x_1$
push(t); $t := \sin(t)$
$t := e^t$
$y_0 := x_0 \cdot t$
$y_1 := t/x_1$

$\bar{x}_1 \mathrel{+}= -t/x_1^2 \cdot \bar{y}_1$
$\bar{t} \mathrel{+}= 1/x_1 \cdot \bar{y}_1$
$\bar{y}_1 := 0$
$\bar{t} \mathrel{+}= x_0 \cdot \bar{y}_0$
$\bar{x}_0 \mathrel{+}= t \cdot \bar{y}_0$
$\bar{y}_0 := 0$
$\bar{t} := t \cdot \bar{t}$
pop(t); $\bar{t} := \cos(v_2) \cdot \bar{t}$
$\bar{x}_1 \mathrel{+}= x_0 \cdot \bar{t}$
$\bar{x}_0 \mathrel{+}= x_1 \cdot \bar{t}$



Euler-Maruyama live ...

Progress

Initially we consider multivariate scalar functions $y = F(\mathbf{x}) : D_F \subseteq \mathbb{R}^n \rightarrow I_F \subseteq \mathbb{R}$ 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.

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

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

as

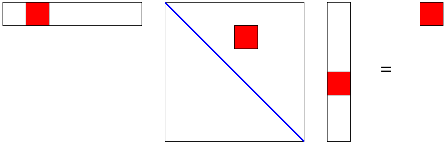
$$\begin{aligned}\frac{d^2 F}{dx_i dx_j}(\mathbf{x}^0) &\approx \frac{\frac{dF}{dx_i}(\mathbf{x}^0 + \mathbf{e}_j\cdot h) - \frac{dF}{dx_i}(\mathbf{x}^0 - \mathbf{e}_j\cdot h)}{2\cdot h} \\ &= \left[\frac{F(\mathbf{x}^0 + \mathbf{e}_j\cdot h + \mathbf{e}_i\cdot h) - F(\mathbf{x}^0 + \mathbf{e}_j\cdot h - \mathbf{e}_i\cdot h)}{2\cdot h}\right. \\ &\quad \left. - \frac{F(\mathbf{x}^0 - \mathbf{e}_j\cdot h + \mathbf{e}_i\cdot h) - F(\mathbf{x}^0 - \mathbf{e}_j\cdot h - \mathbf{e}_i\cdot h)}{2\cdot h}\right] / (2\cdot h).\end{aligned}$$

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:

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



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

... 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 $y^{(1,2)}$.

Note: Approximate Tangents of Tangents

A second derivative code

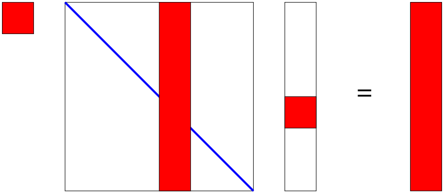
$$F^{(2)}_{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n,$$

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

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

as follows:

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



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

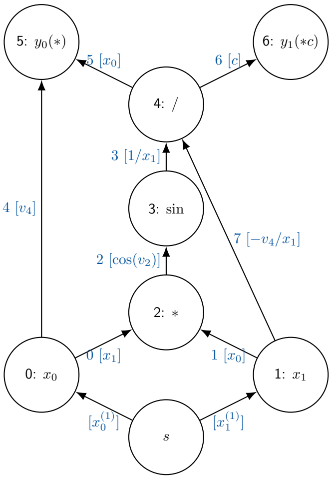
... accumulation of the whole Hessian column-wise by seeding input directions $\mathbf{x}^{(2)} \in \mathbb{R}^n$ with the Cartesian basis vectors in $\mathbb{R}^n$ for $\mathbf{x}_{(1)}^{(2)} = 0$, $y_{(1)} = 1$ and $y_{(1)}^{(2)} = 0$; harvesting from $\mathbf{x}_{(1)}^{(2)}$.

Note: Approximate Tangents of Adjoints

Progress

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)



Tangent IDAG

We consider

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

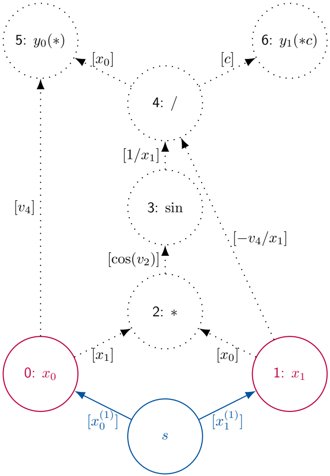
implemented as

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

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*.

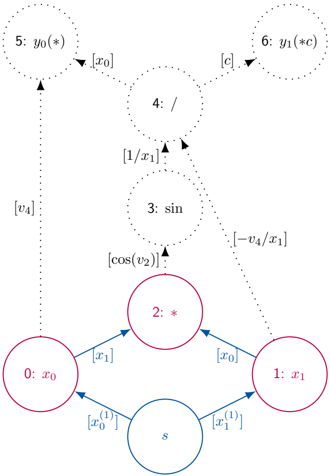


$$x_0 := ?$$
$$x_1 := ?$$

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

Tangents by Overloading

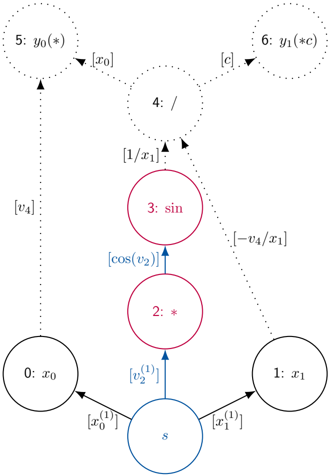
Propagate (Local Directional Derivatives)



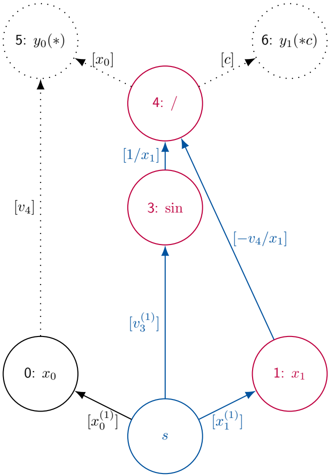
$$v_2 := x_0 * x_1$$
$$v_2^{(1)} := x_1 * x_0^{(1)} + x_0 * x_1^{(1)}$$

Tangents by Overloading

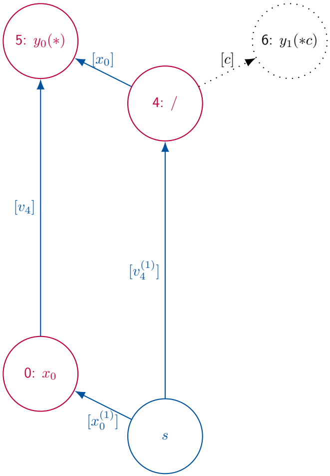
Propagate



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



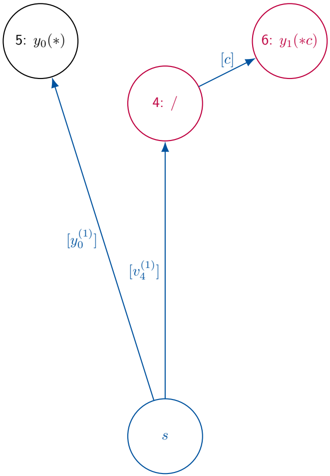
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1
\end{aligned}
$$



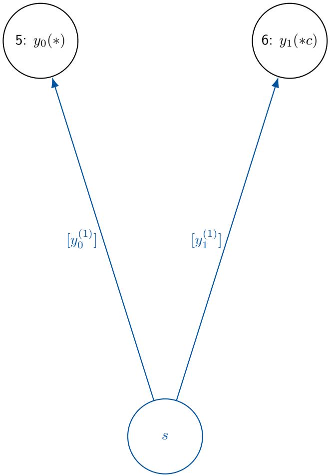
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1 \\
y_0 &:= x_0 * v_4 \\
y_0^{(1)} &:= v_4 * x_0^{(1)} + x_0 * v_4^{(1)}
\end{aligned}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

Tangents by Overloading

Propagate



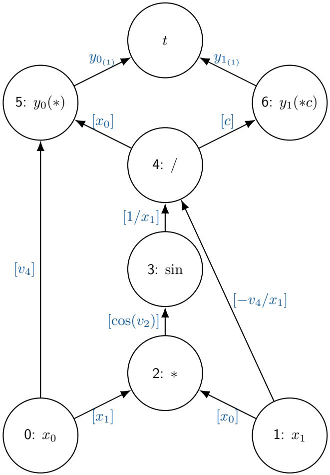
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3 / x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)}) / x_1 \\
y_0 &:= x_0 * v_4 \\
y_0^{(1)} &:= v_4 * x_0^{(1)} + x_0 * v_4^{(1)} \\
y_1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



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

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

```
void driver(const vector<double>& xv, double &yv, vector<double> &g) {
  typedef dco::gt1s<double>::type DCO_T;
  size_t n=xv.size();
  DCO_T y=0;
  for (size_t i=0;i<n;i++) {
    vector<DCO_T> x(n,0);
    for (size_t j=0;j<n;j++) x[j]=xv[j];
    dco::derivative(x[i])=1; // seed directions
    f(x,y); // overloaded primal
    g[i]=dco::derivative(y); // harvest directional derivatives
  }
  yv=dco::value(y); // extract function value
}
```



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

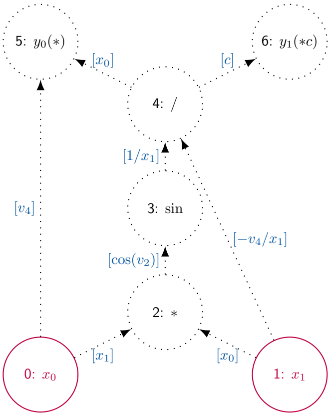
$$t := \sin(x_0 * x_1)/x_1$$
$$y_0 := x_0 * t$$
$$y_1 := t * c$$

yielding SAC

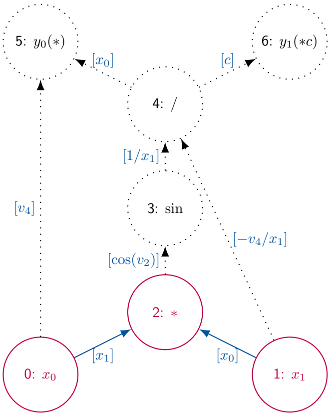
$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3/x_1$$
$$y_0 := x_0 * v_4$$
$$y_1 := v_4 * c$$

for some passive value c.

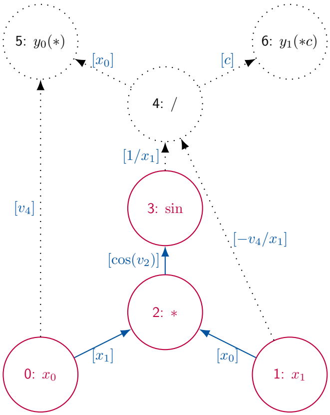
Register (Independent Inputs with Tape)



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



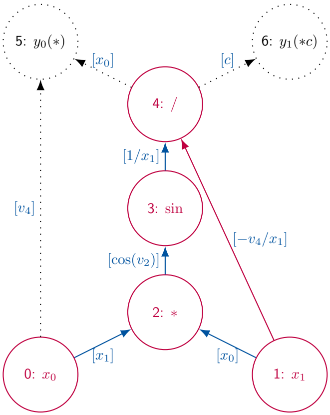
$$v_2 := x_0 * x_1$$



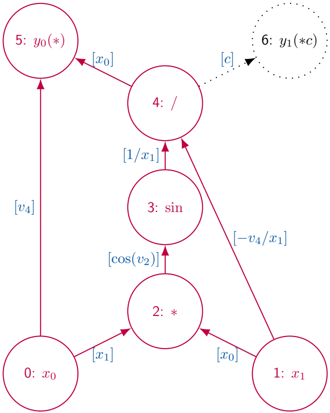
$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$

Adjoints by Overloading

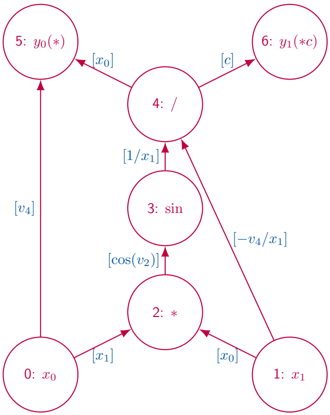
Record (Tape)



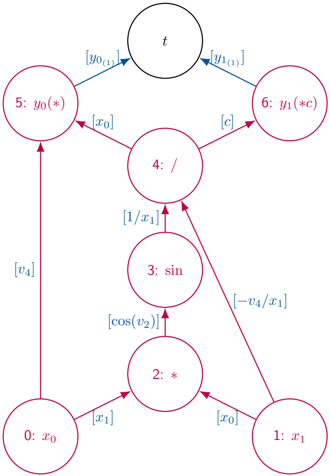
$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3/x_1$$



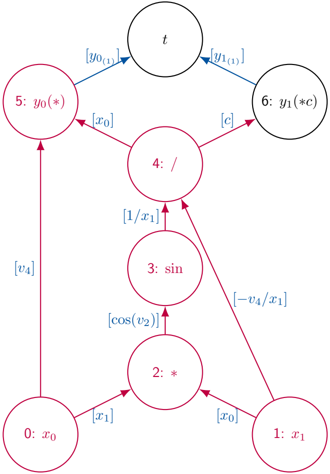
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4
\end{aligned}
$$



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



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



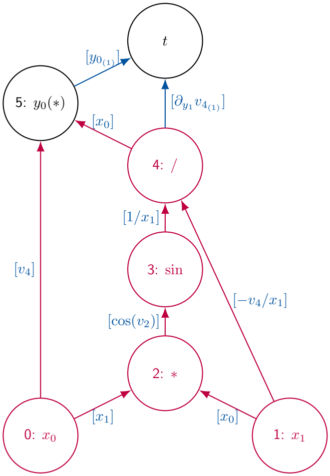
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}}
\end{aligned}
$$

Note C++ Syntax:

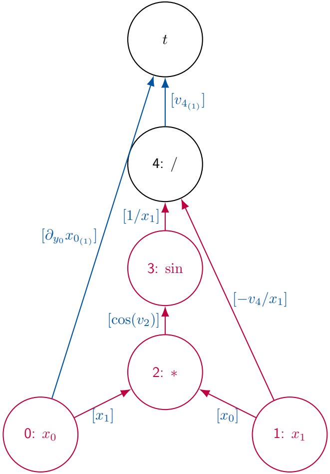
$$v_{4_{(1)}} += c * y_{1_{(1)}}$$

$$\Leftrightarrow$$

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

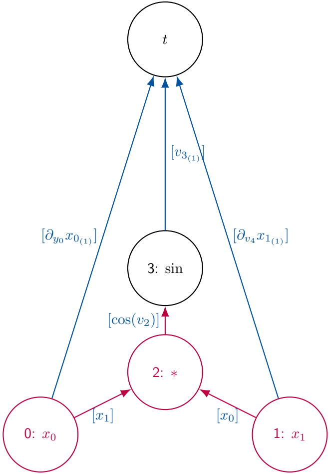


$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}}
\end{aligned}
$$



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

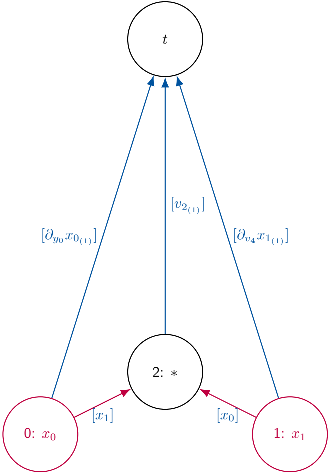
Interpret (Tape)



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

Adjoints by Overloading

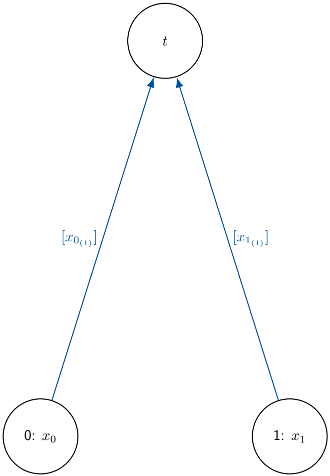
Interpret (Tape)



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

Adjoints by Overloading

Harvest



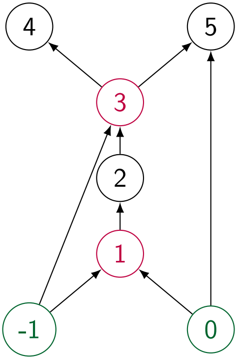
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
u &:= 1/x_1 \\
v_{3_{(1)}} &+= u * v_{4_{(1)}} \\
x_{1_{(1)}} &-= v_4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

Adjoints with dco/c++

Driver

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

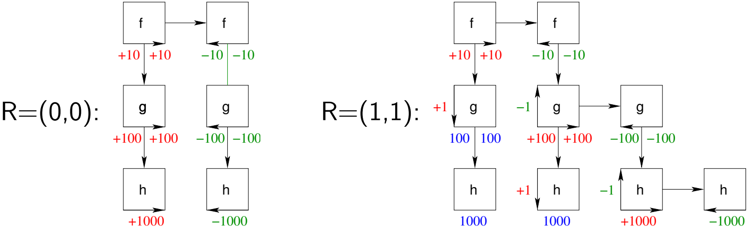
```
void driver(const vector<double> &xv, double &yv, vector<double> &g) {
  typedef dco::ga1s<double> DCO_M; // dco mode
  typedef DCO_M::type DCO_T; // dco type
  typedef DCO_M::tape_t DCO_TAPE_T; /dco tape type
  size_t n=xv.size();
  vector<DCO_T> x(n); DCO_T y;
  DCO_M::global_tape=DCO_TAPE_T::create(); // tape creation
  for (size_t i=0;i<n;i++) { // independent tape entries
    x[i]=xv[i]; DCO_M::global_tape->register_variable(x[i]);
  }
  f(x,y); // overloaded primal
  DCO_M::global_tape->register_output_variable(y); // dependent tape entry
  yv=dco::value(y); dco::derivative(y)=1; // seed
  DCO_M::global_tape->interpret_adjoint(); // tape interpretation
  for (size_t i=0;i<n;i++) { g[i]=dco::derivative(x[i]); } // harvest
  DCO_TAPE_T::remove(DCO_M::global_tape); // release tape
}
```



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

- 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).

Example: Let $\overline{\text{MEM}} = 1110$ ...



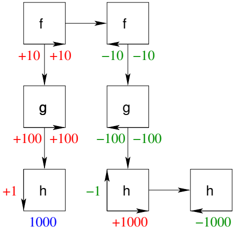
MEM=1220, OPS=0

MEM=1110, OPS=2200

Example: Let $\overline{\text{MEM}} = 1110$ ... Greedy Heuristics

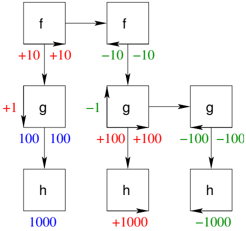
Smallest Memory Increase starts from R $= \mathbf{1}$ and yields ...
Largest Memory Decrease (LMD) starts from R $= \mathbf{0}$ and yields ...

R=(0,1):



MEM=1110, OPS=1000

R=(1,0):



MEM=1120, OPS=1200

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

Algorithmic Differentiation (AD) is based on Symbolic Differentiation (SD).
AD approaches vary in terms of choice of SD level.

- U.N. et al.: Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of linear equations, RWTH Technical Report AIB-2012-10 (2012).
- U.N. et al.: Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations, ACM TOMS 41 (4), 26 (2015).
- N. Safiran et al.: Algorithmic Differentiation of Numerical Methods: Second-Order Adjoint Solvers for Parameterized Systems of Nonlinear Equations, Procedia Computer Science 80, 2231-2235 (2016).
- J. Lotz et al.: A Case Study in Adjoint Sensitivity Analysis of Parameter Calibration, Procedia Computer Science 80, 201-211 (2016).

(Embedded) Symbolic Adjoints

Case Study: Root Finding

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
x^2 - p = 0 \arrow[r, "x_{(1)} \cdot \frac{d}{dp}", "\text{differentiate}"'] \arrow[d, "(x^*, G) = \overrightarrow{S}(x^0, p)"', "\text{record}"] \arrow[dr, "\text{perturb}"', "\dot{x}^* = S(x^0, p+h)"] & x_{(1)} \cdot \left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) = 2 \cdot x \cdot \underbrace{\left(x_{(1)} \cdot \frac{dx}{dp}\right)}_{=p_{(1)}} - x_{(1)} = 0 \arrow[d, "\tilde{p}^* = \tilde{S}(\tilde{p}^0, x^*, x_{(1)})"] \\
x^* \approx \sqrt{p} \arrow[r, "\text{interpret}", "p^*_{(1)} = \overleftarrow{S}(G, x_{(1)})"'] & p_{(1)} = \frac{x_{(1)}}{2 \cdot \sqrt{p}} = \frac{x_{(1)}}{2 \cdot x} \approx p^*_{(1)} \approx \tilde{p}^* \approx x_{(1)} \cdot \frac{\dot{x}^* - x^*}{h}
\end{tikzcd}
$$

$$\underset{x(p)}{\operatorname{argmin}} \ \frac{1}{3}\cdot x^3 - p\cdot x + q$$

differentiate first-order optimality condition

$x_{(1)} \cdot \frac{d}{dp}$

$$x_{(1)} \cdot \left(2\cdot x \cdot \frac{dx}{dp} - 1\right) = 2\cdot x \cdot \underbrace{\left(x_{(1)}\cdot\frac{dx}{dp}\right)}_{=p_{(1)}} - x_{(1)} = 0$$

$(x^*, G) = \overrightarrow{S}(x^0, p)$ record

perturb $\dot{x}^* = S(x^0, p+h)$

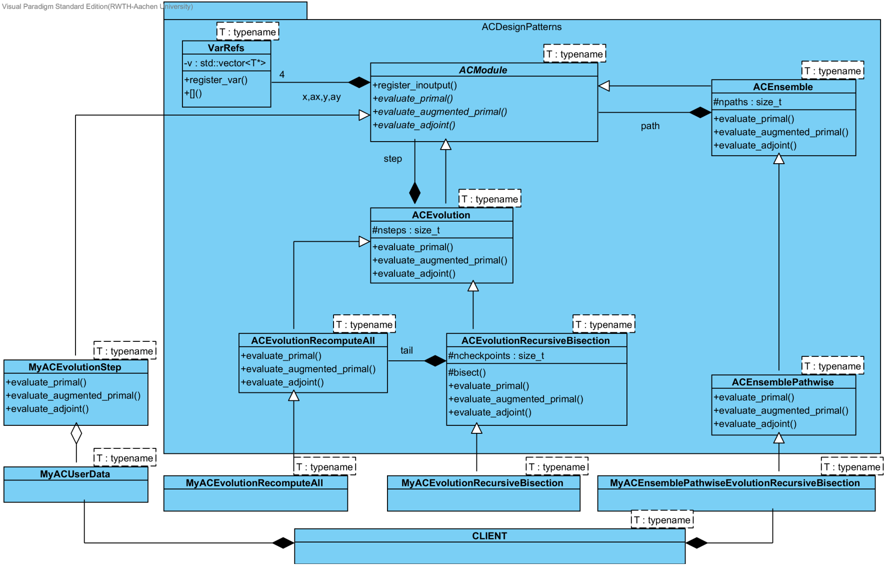
$\tilde{p}^* = \tilde{S}(\tilde{p}^0, x^*, x_{(1)})$

$$x^* \approx \sqrt{p}$$

interpret

$p^*_{(1)} = \overleftarrow{S}(G, x_{(1)})$

$$p_{(1)} = \frac{x_{(1)}}{2\cdot\sqrt{p}} = \frac{x_{(1)}}{2\cdot x} \approx p^*_{(1)} \approx \tilde{p}^* \approx x_{(1)} \cdot \frac{\dot{x}^* - x^*}{h}$$



Visual Paradigm Standard Edition(RWTH-Aachen University)
ACDesignPatterns
T : typename
VarRefs
-v : std::vector<T*>
+register_var()
+[]()
4
x,ax,y,ay
T : typename
ACModule
+register_inoutput()
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
T : typename
ACEnsemble
#npaths : size_t
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
path
step
T : typename
ACEvolution
#nsteps : size_t
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
T : typename
ACEvolutionRecomputeAll
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
tail
T : typename
ACEvolutionRecursiveBisection
#ncheckpoints : size_t
#bisect()
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
T : typename
ACEnsemblePathwise
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
T : typename
MyACEvolutionStep
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
T : typename
MyACUserData
T : typename
MyACEvolutionRecomputeAll
T : typename
MyACEvolutionRecursiveBisection
T : typename
MyACEnsemblePathwiseEvolutionRecursiveBisection
T : typename
CLIENT