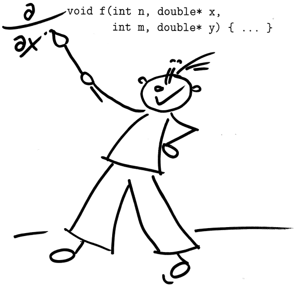


[Adjoint] Algorithmic Differentiation ([A]AD)

Risk Training Masterclass, London, 21-22 March 2018

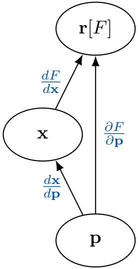
Uwe Naumann

STCE, RWTH Aachen University, Germany

What is a (1st-order) Tangent?

Primal $\mathbf{x} = \mathbf{x}(\mathbf{p}) \in \mathbb{R}^m$ defined implicitly:

$$\mathbf{r} \equiv F(\mathbf{x}, \mathbf{p}) = 0; \quad F : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$$



Tangent (forward sensitivities) $\mathbf{x}^{(1)} = \mathbf{x}^{(1)}(\mathbf{p}, \mathbf{p}^{(1)}) \in \mathbb{R}^m$:

$$\mathbf{r}^{(1)} \equiv F^{(1)}(\mathbf{x}^{(1)}, \mathbf{p}, \mathbf{p}^{(1)}) = 0; \quad F^{(1)} : \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$$

defined by

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

We distinguish between complete ($\frac{d\cdot}{d\cdot}$) and incomplete ($\frac{\partial\cdot}{\partial\cdot}$) sensitivities / (partial) derivatives.

$$r \equiv x^2 - p = 0$$

Implicit Function Theorem

$$r^{(1)} \equiv F^{(1)}(x^{(1)}, p, p^{(1)}) = 0$$

$$x^{(1)} = -\frac{dF}{dx}^{-1} \cdot \frac{\partial F}{\partial p} \cdot p^{(1)}$$
$$= -\frac{1}{2 \cdot x} \cdot (-1) \cdot p^{(1)}$$

primal $(r \to 0)$

symbolic tangent $(r^{(1)} \to 0)$
OC $= O(1) \cdot$ POC

$\check{x}$

approximate tangent $\tilde{x}^{(1)}$
OC $= O(1) \cdot$ POC

algorithmic tangent $\check{x}^{(1)}$
OC $= O(1) \cdot$ POC

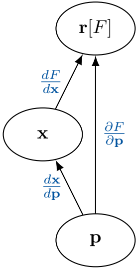
$$\check{x}^{(1)} \overset{?}{\approx} \tilde{x}^{(1)} \overset{?}{\approx} x^{(1)}$$

Notation: OC $\hat{=}$ operations count; POC $\hat{=}$ primal operations count

What is a (1st-order) Adjoint?

Adjoint (backward sensitivities) $\mathbf{p}_{(1)} = \mathbf{p}_{(1)}(\mathbf{x}_{(1)}, \mathbf{p}) \in \mathbb{R}^n$:

$$\mathbf{r}_{(1)} \equiv F_{(1)}(\mathbf{x}_{(1)}, \mathbf{p}, \mathbf{p}_{(1)}) = 0; \quad F_{(1)} : \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$$



defined by

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

implies consistency

$$\langle \mathbf{x}_{(1)}, \mathbf{x}^{(1)} \rangle = \langle \mathbf{p}_{(1)}, \mathbf{p}^{(1)} \rangle$$

$\Rightarrow$ necessary condition for numerical evaluation of tangents and adjoints

... hard to obtain by symbolic differentiation
... ensured by algorithmic differentiation

Story in a Nutshell

Symbolic, Algorithmic, Approximate Adjoints

$$r = x^2 - p = 0 \xrightarrow{\text{Implicit Function Theorem}}$$

$$r_{(1)} \equiv F_{(1)}(x_{(1)}, p, p_{(1)}) = 0$$

$$\begin{aligned} p_{(1)} &= -\frac{\partial F}{\partial p}^T \cdot \frac{dF}{dx}^{-T} \cdot x_{(1)} \\ &= -(-1) \cdot \frac{1}{2 \cdot x} \cdot x_{(1)} \end{aligned}$$

primal $(r \to 0)$

$$\check{x}$$

symbolic adjoint $(r_{(1)} \to 0)$
OC $= O(1) \cdot$ POC
PMR $\sim$ POC

approximate adjoint $\tilde{p}_{(1)}$
OC $\sim O(n) \cdot$ POC

algorithmic adjoint $\check{r}_{(1)}$
OC $\sim O(1) \cdot$ POC
PMR $\sim$ POC

$$\check{p}_{(1)} \overset{?}{\approx} \tilde{p}_{(1)} \overset{?}{\approx} p_{(1)}$$

$$\langle x^{(1)}, x_{(1)} \rangle \overset{?}{\approx} \langle p^{(1)}, p_{(1)} \rangle$$

$$\langle \check{x}^{(1)}, x_{(1)} \rangle \overset{!}{\approx} \langle p^{(1)}, \check{p}_{(1)} \rangle$$

Notation: PMR $\hat{=}$ persistent memory requirement

Tangents vs. Adjoints

Algebraic Perspective

Chain Rule on $F : \mathbb{R}^n \to \mathbb{R}^m$ (assuming differentiability of F, G and H)

$$\mathbf{y} = F(\underbrace{G(\underbrace{H(\mathbf{x})}_{\boldsymbol{u}\in\mathbb{R}^p})}_{\boldsymbol{v}\in\mathbb{R}^q}) \quad \Rightarrow \quad \frac{d\mathbf{y}}{d\mathbf{x}} = \frac{d\mathbf{y}}{d\boldsymbol{v}} \cdot \frac{d\boldsymbol{v}}{d\boldsymbol{u}} \cdot \frac{d\boldsymbol{u}}{d\mathbf{x}}$$

Tangent

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \frac{d\mathbf{y}}{d\boldsymbol{v}} \cdot \left(\frac{d\boldsymbol{v}}{d\boldsymbol{u}} \cdot \frac{d\boldsymbol{u}}{d\mathbf{x}} \right)$$

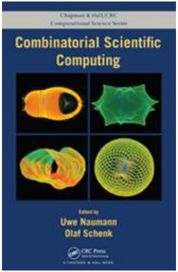
Adjoint

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \left(\frac{d\mathbf{y}}{d\boldsymbol{v}} \cdot \frac{d\boldsymbol{v}}{d\boldsymbol{u}} \right) \cdot \frac{d\boldsymbol{u}}{d\mathbf{x}}$$

Example: Dense local Jacobians for $n = 10$, $p = 10$, $q = 10$, and $m = 1$:

$$1100 \text{ fma} > 200 \text{ fma}$$

- V. Mosenkis, U. N.: *On Lower Bounds for Optimal Jacobian Accumulation.* To appear in OMS, 2018.
- U. N.: *Optimal Jacobian accumulation is NP-complete.* Math. Prog. 112(2):427–441, Springer, 2008.
- U. N.: *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph.* Math. Prog. 99(3):399–421, Springer, 2004.
- A. Griewank and U. N.: *Accumulating Jacobians as chained sparse matrix products.* Math. Prog. 95(3):555–571, Springer, 2003.



Case Study: SDE

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 Euler-Maruyama scheme

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

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

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

Case Study: SDE

Performance: $m = 10^4$, $n = 10^3$

$$dx = p(t) \cdot \sin(x(p(t), t) \cdot t)dt + p(t) \cdot \cos(x(p(t), t) \cdot t)dW; \;\; t \in [0, 1]$$

	Time (s)	Memory (MB)	rel. Time
Primal	1.2	79	1
FFD	380	80	317
AAD by hand	1.9	240	1.6
checkpointed AAD by hand	2.2	80	1.8
AAD by dco/c++	1.7	728	1.4
checkpointed AAD by dco/c++	1.9	86	1.6

FFD: Forward Finite Differences
AAD: Adjoint Algorithmic Differentiation

$\longrightarrow$ see code, race

Case Study: PDE

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

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

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

The sample solution codes use central finite difference discretization in space within explicit and implicit Euler time integration schemes.

We are interested in

$$\frac{dy(1, x)}{dy(0, x)}^T \cdot \mathbf{v} \ .$$

$\rightarrow$ see code

Case Study: LIBOR

We consider the same LIBOR market model which was used in

M.B. Giles and P. Glasserman: *Smoking adjoints: fast Monte Carlo Greeks.* RISK, January 2006.

to illustrate the benefits of AAD for simulations in finance.

See also

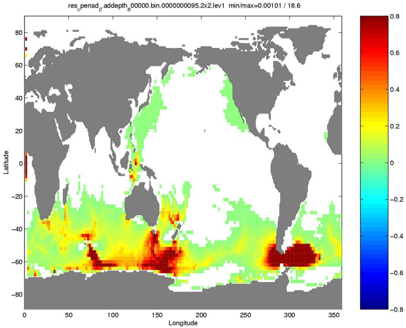
M.B. Giles: *Monte Carlo evaluation of sensitivities in computational finance.* In Elias A. Lipitakis, editor, HERCMA Conference, Athens 2007.

and

`http://people.maths.ox.ac.uk/~gilesm/codes/libor_AD/`

→ see code

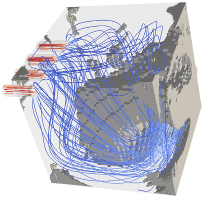
Let the run time of f be 1min. Let the spatially distributed parameter (e.g, bottom topography) be defined on a mesh with 10^6 cells. Finite difference approximation of the gradient of the average amount of water flowing through the Drake passage takes $O(10^6)$min (almost 2 years). The algorithmic adjoint MITgcm computes the gradient with machine accuracy in $O(1)$min (approx. 10min).

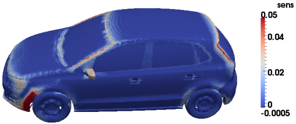


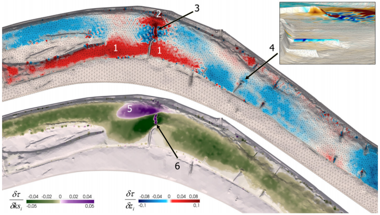
J. Utke, U.N. et al.: *OpenAD/F: A Modular Open-Source Tool for Automatic Differentiation of Fortran Codes*, ACM TOMS, 2008.

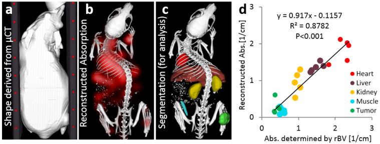
Adjoints Are Everywhere ...

e.g, Other Projects









Reading

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

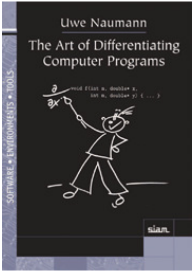
U.N., J. du Toit: *Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance*, JCF 2018.

K. Leppkes, J. Lotz, U.N.: dco/c++: Derivative Code by Overloading in C++. Under Review for ACM TOMS.

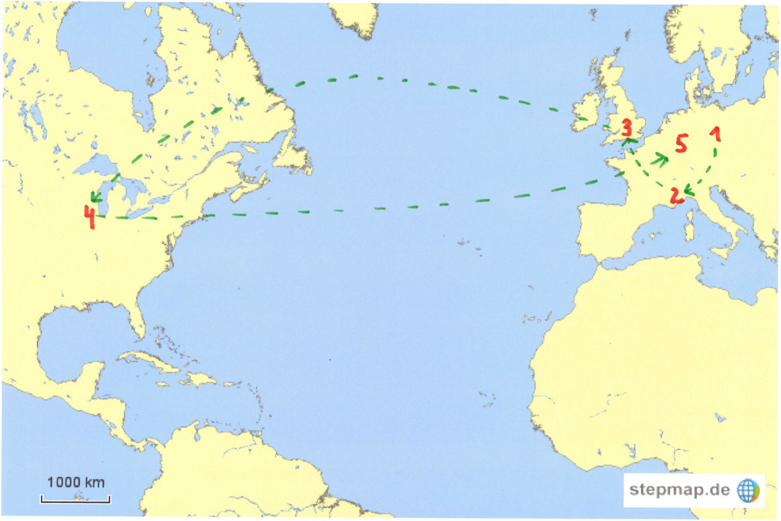
U.N.: Adjoint Code Design Patterns. Under Review for ACM TOMS.

K. Leppkes, J. Lotz, U.N., J. du Toit: Meta Adjoint Programming in C++, Technical Report AIB-2017-07, Dept. of Computer Science, RWTH Aachen University, Sep. 2017.

U.N., K. Leppkes: Low-Memory Algorithmic Adjoint Propagation. CSC18.



About Myself



Outline

Outline

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

1. tangent AD
   - $\mathbf{y}^{(1)} = \nabla F \cdot \mathbf{x}^{(1)} \Rightarrow \nabla F$ at $O(n) \cdot$ POC
   - approximate tangents by finite differences
2. adjoint AD
   - $\mathbf{x}_{(1)} = \nabla F^T \cdot \mathbf{y}_{(1)} \Rightarrow \nabla F$ at $O(m) \cdot$ POC
   - $m = 1 \Rightarrow$ cheap gradients at $O(1) \cdot$ POC
   - PMR $\sim$ POC
3. higher-level elemental functions, e.g, BLAS

Essential Ingredients

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. All elemental functions φ_j possess continuous (local) 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.
A linearized SAC is obtained by augmenting the elemental assignments with computations of the local partial derivatives $d_{j,i}$.

→ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

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

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 $Z \subsetneq V$.

A labeled DAG is obtained by attaching the $d_{j,i}$ to the corresponding edges (i, j) in the DAG.

In the following DAGs are assumed to be labelled.

$\rightarrow$ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

Essential Ingredients

Chain Rule

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

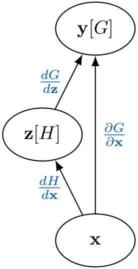
$$\mathbf{y} = F(\mathbf{x}) = G(H(\mathbf{x}), \mathbf{x}) = G(\mathbf{z}, \mathbf{x})$$

be such that both

$$G : D_G \subseteq \mathbb{R}^p \times \mathbb{R}^n \to I_G \subseteq \mathbb{R}^m$$

and

$$H : D_H \subseteq \mathbb{R}^n \to I_H \subseteq \mathbb{R}^p$$



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

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

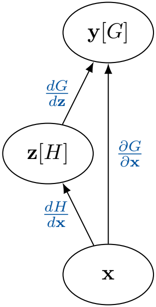
for all $\mathbf{x}^* \in D_F$ and $\mathbf{z}^* = H(\mathbf{x}^*)$.

Essential Ingredients

Chain Rule on DAG

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

DAG:



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

$\rightarrow$ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

Tangents

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

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

augments the computation of the primal function with the computation of a Jacobian-vector product:

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

The entire 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 $\mathbb{R}^n$.

Active and Passive (Program) Variables

Variables for which derivatives are computed are referred to as active; $\mathbf{x}$ is active input; $\mathbf{y}$ is active output.

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

Variables for which no derivatives are computed are referred to as passive.

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

Active variables are both varied and useful.

Define

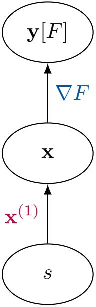
$$\boldsymbol{v}^{(1)} \equiv \frac{d\boldsymbol{v}}{ds}$$

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

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

Application of the chain rule to the tangent DAG yields $\mathbf{y}^{(1)} \in \mathbb{R}^m$ as a function of $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{x}^{(1)} \in \mathbb{R}^n$. (Note: forward edge back-elimination)



$\rightarrow$ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

Tangents by AD (Forward Mode)

Tangent SAC

Similar reasoning applied to the SAC yields ...

$$i = 0, \ldots, n-1: \quad \begin{pmatrix} v_i \\ v_i^{(1)} \end{pmatrix} := \begin{pmatrix} x_i \\ x_i^{(1)} \end{pmatrix}$$

“seed”

$$i = n, \ldots, q-1: \quad \begin{pmatrix} v_i \\ v_i^{(1)} \end{pmatrix} := \begin{pmatrix} \varphi_i(v_k)_{k \prec i} \\ \sum_{j \prec i} \frac{d\varphi_i(v_k)_{k \prec i}}{dv_j} \cdot v_j^{(1)} \end{pmatrix}$$

“propagate”

$$i = 0, \ldots, m-1: \quad \begin{pmatrix} y_i \\ y_i^{(1)} \end{pmatrix} := \begin{pmatrix} v_{n+p+i} \\ v_{n+p+i}^{(1)} \end{pmatrix}$$

“harvest”

→ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

Tangents by AD (Forward Mode)

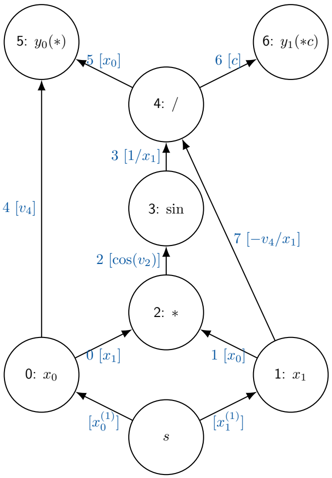
Code Generation Rules

1. duplicate active data segment
2. augment assignments with their tangents
3. leave flow of control unchanged
4. replace subprogram calls with their calls to their tangent versions

```
...
    for (int i=0;i<n;i++) {
      xt+=dt*sin(x*t)*pt[i]
          +dt*p[i]*t*cos(x*t)*xt
          +cos(x*t)*sqrt(dt)*dW[j][i]*pt[i]
          -p[i]*t*sin(x*t)*sqrt(dt)*dW[j][i]*xt;
      x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];
      t+=dt;
    }
...
```

→ SDE

Forward Edge Back-Elimination on Tangent DAG



Tangent DAG

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

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

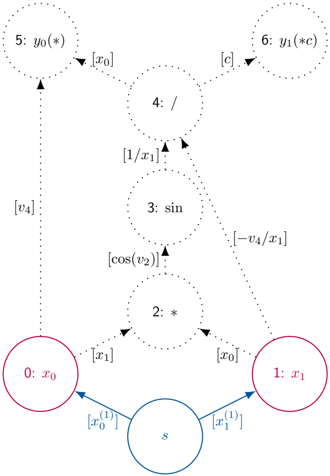
yielding SAC

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

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

Tangents by Overloading

Seed

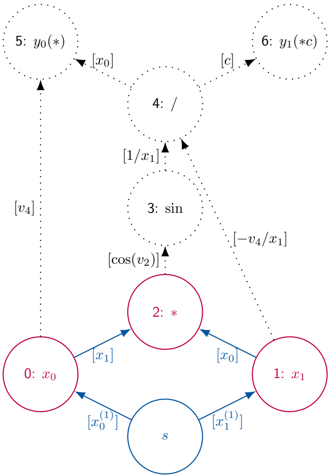


$$x_0 := ?$$
$$x_1 := ?$$

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

Tangents by Overloading

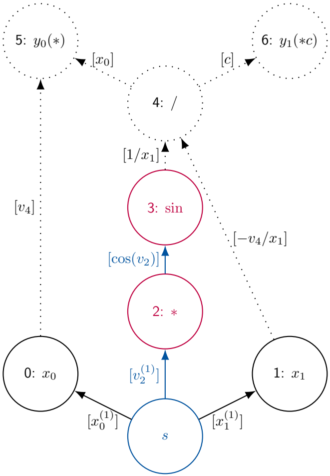
Propagate (Local Directional Derivatives)



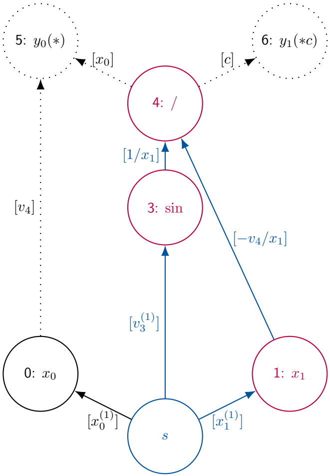
$$v_2 := x_0 * x_1$$
$$v_2^{(1)} := x_1 * x_0^{(1)} + x_0 * x_1^{(1)}$$

Tangents by Overloading

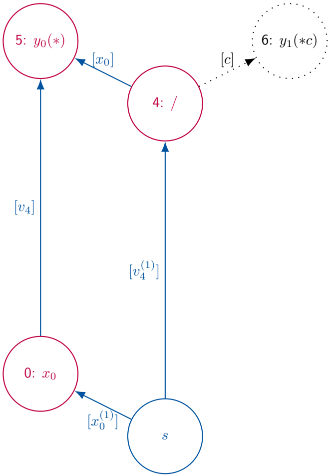
Propagate



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



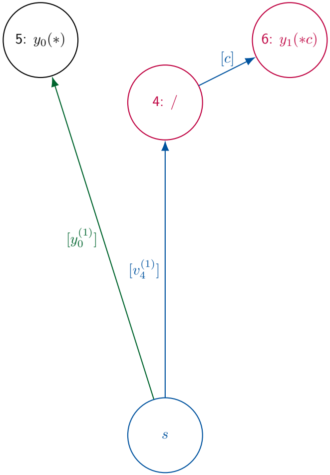
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_2^{(1)} &:= x_1 * x_0^{(1)} + x_0 * x_1^{(1)} \\
v_3 &:= \sin(v_2) \\
v_3^{(1)} &:= \cos(v_2) * v_2^{(1)} \\
v_4 &:= v_3/x_1 \\
v_4^{(1)} &:= (v_3^{(1)} - v_4 * x_1^{(1)})/x_1
\end{aligned}
$$



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

Tangents by Overloading

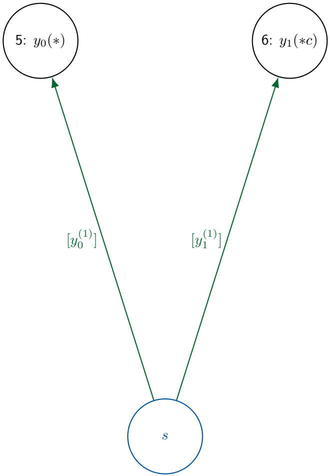
Propagate



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

Tangents by Overloading

Harvest



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

Tangents by dco/c++

Scalar Tangents (dco::gt1s<double>)

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

vector<double> driver(double& xv, vector<double>& pv,
    const vector<vector<double>>& dW) {
  int n=dW[0].size(); vector<double> g(n+1,0);
  DCO_T x0=xv; vector<DCO_T> p(n); dco::value(p)=pv; DCO_T x=x0;
  dco::derivative(x)=1; // seed
  euler_maruyama(x,p,dW); // propagate
  g[0]=dco::derivative(x); // harvest
  for (int i=0;i<n;i++) {
    x=x0; // reset
    dco::derivative(p[i])=1; // seed
    euler_maruyama(x,p,dW); // propagate
    g[i+1]=dco::derivative(x); // harvest
    dco::derivative(p[i])=0; // reset
  }
  return g;
}
```

Adjoints

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

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

augments the computation of the function with the computation of a shifted product of the transposed Jacobian with a vector:

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

... 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.

- context-sensitive adjoint

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

  if

  - subsequent active use of $\mathbf{x}$
  - previous active use of $\mathbf{y}$

  in primal

- context-free adjoint

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

  if

  - no subsequent active use of $\mathbf{x}$
  - no previous active use of $\mathbf{y}$

  in primal

Adjoints by AD

Adjoint DAG

Define

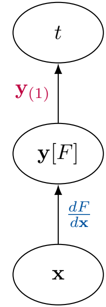
$$\boldsymbol{v}_{(1)} \equiv \frac{dt}{d\boldsymbol{v}}^T$$

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

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

Application of the chain rule to the adjoint DAG yields $\mathbf{x}_{(1)} \in \mathbb{R}^n$ as a function of $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y}_{(1)} \in \mathbb{R}^m$. (Note: reverse vertex elimination)



→ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

Similar reasoning applied to the SAC yields ...

$$i = 0, \ldots, n-1 : \quad v_i := x_i$$

"record" independent variables for harvesting

$$i = n, \ldots, q-1 : \quad v_i := \varphi_i(v_k)_{k \prec i}$$

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

$$i = 0, \ldots, m-1 : \quad y_i := v_{n+p+i}$$

"record" dependent variables for seeding

Software and Tools for Computational Engineering — RWTH AACHEN UNIVERSITY

$$i = 0, \ldots, m-1: \quad v_{n+p+i_{(1)}} := y_{i_{(1)}} \quad \text{“seed”}$$

$$i = q-1, \ldots, n: \quad v_{i_{(1)}} := \sum_{j:i \prec j} d_{j,i} \cdot v_{j_{(1)}} \quad \text{“propagate”}$$

$$i = 0, \ldots, n-1: \quad x_{i_{(1)}} := v_{i_{(1)}} \quad \text{“harvest”}$$

→ x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];

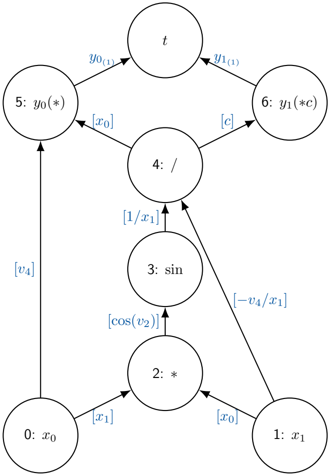
Adjoint Code Generation Rules

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, count loops and enumerate branches)
   1.4 enable recovery of primal results
2. adjoint section
   2.1 recovery 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, `z=cos(y); ... y=sin(x);`)
   2.5 recover primal results

Adjoint Code Generation Rules

```
// augmented primal
...
    for (int i=0;i<n;i++) {
      tbr_T.push(x);
      x+=dt*p[i]*sin(x*t)+p[i]*cos(x*t)*sqrt(dt)*dW[j][i];
      tbr_double.push(t);
      t+=dt;
    }
...
// adjoint
...
    for (int i=n-1;i>=0;i--) {
      t=tbr_double.top(); tbr_double.pop();
      x=tbr_T.top(); tbr_T.pop();
      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;
    }
...
```

→ SDE



Adjoint DAG

We consider

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

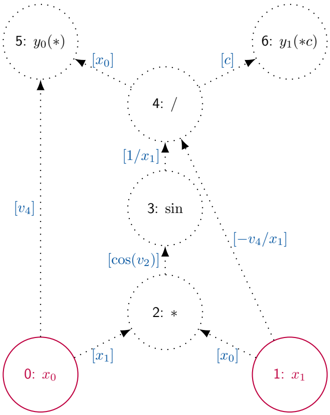
implemented as

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

yielding SAC

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

for some passive value c.

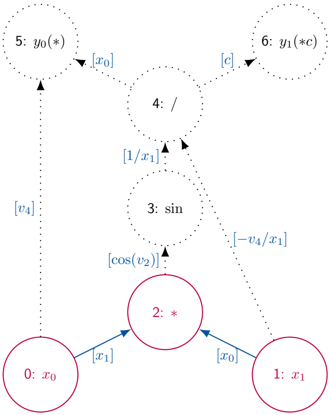


$x_0 :=?$

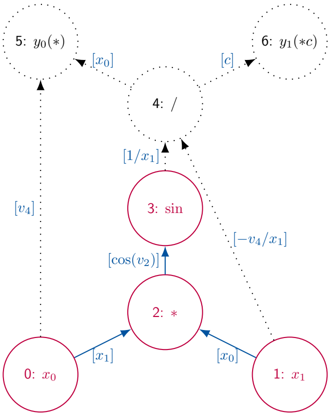
$x_1 :=?$

Adjoints by Overloading

Record (Tape)



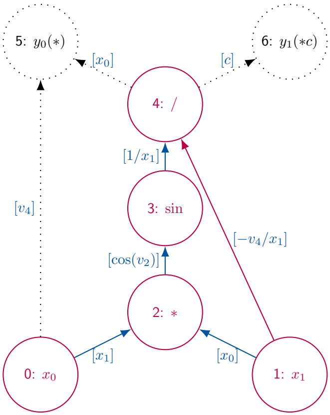
$$v_2 := x_0 * x_1$$



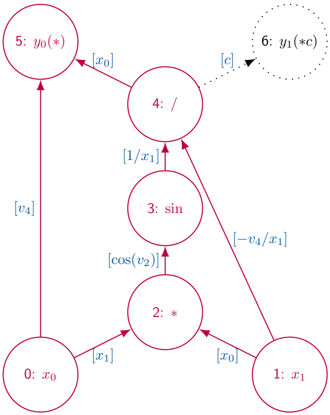
$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$

Adjoints by Overloading

Record (Tape)



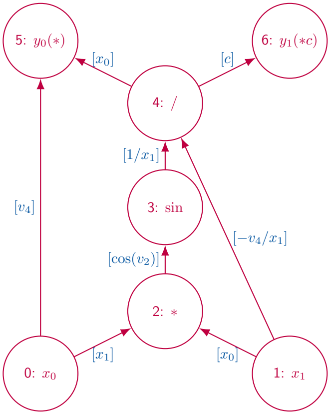
$$v_2 := x_0 * x_1$$
$$v_3 := \sin(v_2)$$
$$v_4 := v_3 / x_1$$



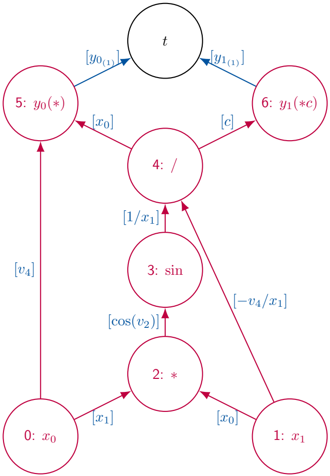
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3/x_1 \\
y_0 &:= x_0 * v_4
\end{aligned}
$$

Adjoints by Overloading

Record (Tape)

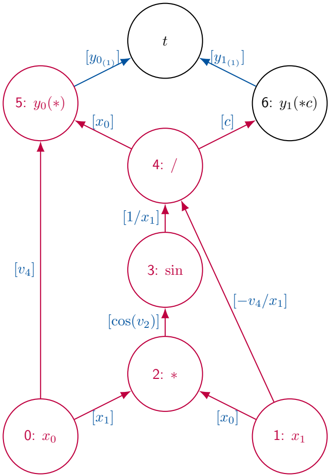


$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c
\end{aligned}
$$



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

Interpret (Tape)



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

Context-sensitivity:

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

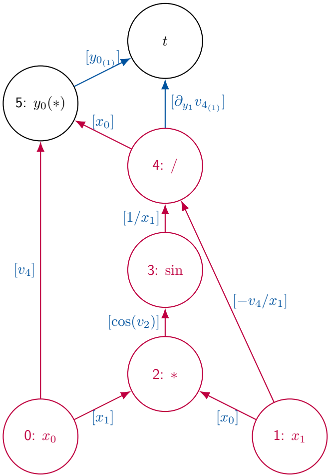
$$\Leftrightarrow$$

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

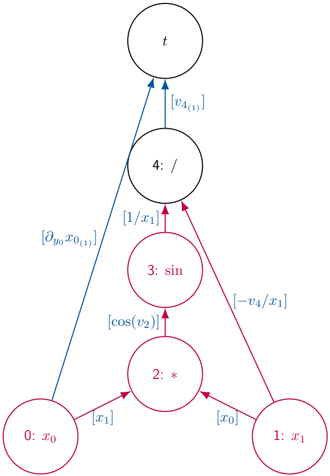
Note: Need to store DAG yields infeasible PMR in most application scenarios.

Adjoints by Overloading

Interpret (Tape)



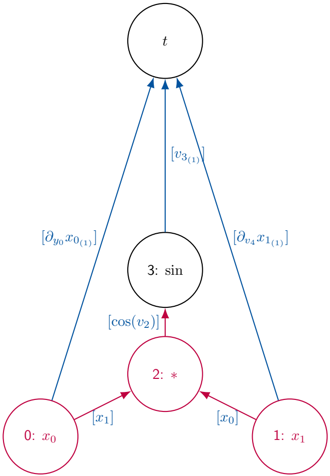
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}}
\end{aligned}
$$



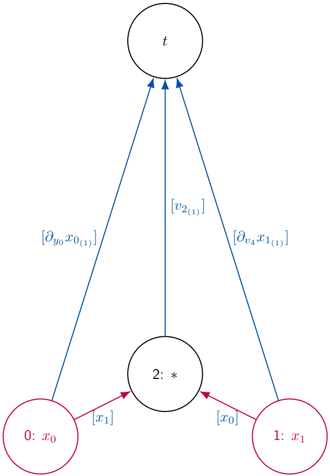
$$
\begin{aligned}
v_2 &:= x_0 * x_1 \\
v_3 &:= \sin(v_2) \\
v_4 &:= v_3 / x_1 \\
y_0 &:= x_0 * v_4 \\
y_1 &:= v_4 * c \\
v_{4_{(1)}} &+= c * y_{1_{(1)}} \\
v_{4_{(1)}} &+= x_0 * y_{0_{(1)}} \\
x_{0_{(1)}} &+= v_4 * y_{0_{(1)}} \\
u &:= 1/x_1 \\
v_{3_{(1)}} &+= u * v_{4_{(1)}} \\
x_{1_{(1)}} &-= v_4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

Adjoints by Overloading

Interpret (Tape)



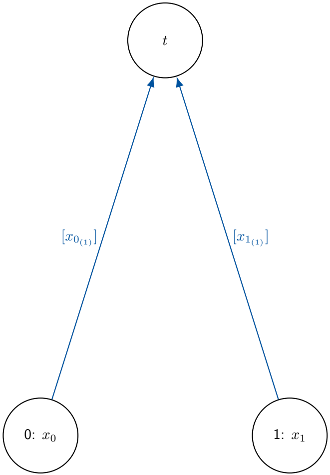
$$
\begin{aligned}
v_2 &:= x_0 * x_1\\
v_3 &:= \sin(v_2)\\
v_4 &:= v_3 / x_1\\
y_0 &:= x_0 * v_4\\
y_1 &:= v_4 * c\\
v_{4(1)} &+= c * y_{1(1)}\\
v_{4(1)} &+= x_0 * y_{0(1)}\\
x_{0(1)} &+= v_4 * y_{0(1)}\\
u &:= 1/x_1\\
v_{3(1)} &+= u * v_{4(1)}\\
x_{1(1)} &-= v_4 * u * v_{4(1)}\\
v_{2(1)} &+= \cos(x_2)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



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

Adjoints 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 by dco/c++

Scalar Adjoints (dco::ga1s<double>)

```
#include "dco.hpp"
typedef dco::ga1s<double> DCO_A_MODE; // adjoint mode
typedef DCO_A_MODE::type DCO_A; // adjoint type
typedef DCO_A_MODE::tape_t DCO_A_TAPE; // tape type

vector<double> driver(double& xv, vector<double>& pv,
    const vector<vector<double>>& dW) {
  int n=dW[0].size(); vector<double> g(n+1,0);
  DCO_A x0=xv; vector<DCO_A> p(n); dco::value(p)=pv;
  DCO_A_MODE::global_tape=DCO_A_TAPE::create(); // create tape
  DCO_A_MODE::global_tape->register_variable(x0); // record ...
  DCO_A_MODE::global_tape->register_variable(p); // ... active inputs
  DCO_A x=x0; // lock overwritten active input
  euler_maruyama(x,p,dW); // record intermediates
  DCO_A_MODE::global_tape->register_output_variable(x); // record ...
  dco::derivative(x)=1; // ... and seed active output
  DCO_A_MODE::global_tape->interpret_adjoint(); // propagate adjoints
  g[0]=dco::derivative(x0); // harvest from locked active input
  for (int i=0;i<n;i++) g[i+1]=dco::derivative(p[i]); // harvest
  DCO_A_TAPE::remove(DCO_A_MODE::global_tape); // remove tape
  return g;
}
```

Hands On

For given PDE and/or LIBOR codes ...

- ... write tangent code + driver
- ... use dco/c++ to generate tangent code; write driver
- ... write adjoint code + driver
- ... use dco/c++ to generate adjoint code; write driver
- ... cross-validate, race

Improvements

"Low Hanging Fruits"

- vector modes
- pathwise adjoints
- preaccumulation

Discussion

- higher-level elementals
- detection and exploitation of sparsity
- vector modes
- mixed precision
- nested tangents / adjoints / finite differences
- smoothing
- scripting and syntax-directed adjoints by interpretation

Outline

W.l.o.g, let $y = F(\mathbf{x}),\ F : \mathbb{R}^n \to \mathbb{R}$:

1. 2nd-order tangent AD: $y^{(1,2)} = \mathbf{x}^{(1)^T} \cdot \nabla^2 F \cdot \mathbf{x}^{(2)} \Rightarrow \nabla^2 F$ at $O(n^2) \cdot$ POC
2. 2nd-order adjoint AD: $\mathbf{x}^{(2)}_{(1)} = y_{(1)} \cdot \nabla F^2 \cdot \mathbf{x}^{(2)} \Rightarrow \nabla^2 F$ at $O(n) \cdot$ POC and $\nabla^2 F \cdot \mathbf{x}^{(2)}$ at $O(1) \cdot$ POC
3. three mathematically equivalent combinations of dco/c++ types for second-order adjoint
4. tensor projections for multivariate vector functions

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.

Second-Order Finite Differences

A second-order *central finite difference* quotient

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

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

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

Tangents

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-of-Tangent (TT) 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:

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

Note: In context of chain rule: $y^{(1)}$ and $y^{(2)}$ required and non-vanishing $\mathbf{x}^{(1,2)}$

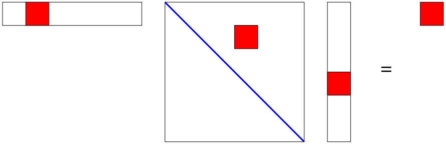
Directional differentiation in tangent mode of the first-order tangent model

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

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

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

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

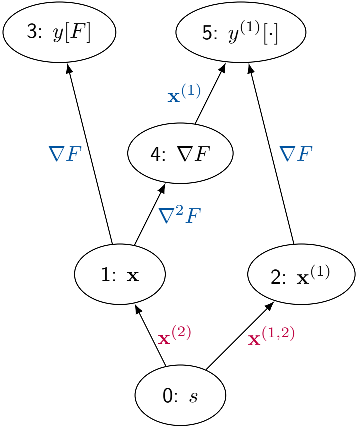


$$
{\mathbf{x}^{(1)}}^T \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}
$$

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

Tangents of Tangents

… on Tangent-Augmented Tangent DAG



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

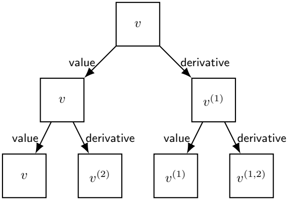
See also AD of Inner Product.

1. Apply tangent code generation rules to first-order tangent code
2. Write appropriate driver
3. Parallelize / vectorize accumulation of the Hessian (optional)

Tangents of Tangents by dco/c++

Cheat Sheet

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




```
dco::value(dco::value(v))==dco::passive_value(v)
dco::derivative(dco::value(v))
dco::value(dco::derivative(v))
dco::derivative(dco::derivative(v))
```

Tangents of Tangents by dco/c++

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


```cpp
#include "dco.hpp"
typedef dco::gt1s<double>::type DCO_T; // tangent type
typedef dco::gt1s<DCO_T>::type DCO_TT; // tangent-of-tangent type

vector<vector<double>> driver(double& xv, const vector<double> &pv,
    const vector<vector<double>>& dW) {
  int n=pv.size();
  vector<DCO_TT> p(n); dco::passive_value(p)=pv; // zero tangents
  vector<vector<double>> ddxdpp(n,vector<double>(n,0));
  for (int i=0;i<n;i++) {
    dco::derivative(dco::value(p[i]))=1; // seed
    for (int j=0;j<=i;j++) {
      dco::value(dco::derivative(p[j]))=1; // seed
      DCO_TT x=xv;
      euler_maruyama(x,p,dW); // overloaded primal
      ddxdpp[i][j]=dco::derivative(dco::derivative(x)); // harvest
      dco::value(dco::derivative(p[j]))=0; // reset
    }
    dco::derivative(dco::value(p[i]))=0; // reset
  }
  return ddxdpp;
}
```

Adjoints

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-of-Adjoint (TA) 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:

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

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

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

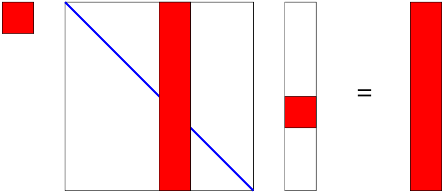
in direction $(\mathbf{x}^{(2)}\ y_{(1)}^{(2)})^T$ yields

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

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

Tangents of Adjoints

Accumulation of Hessian

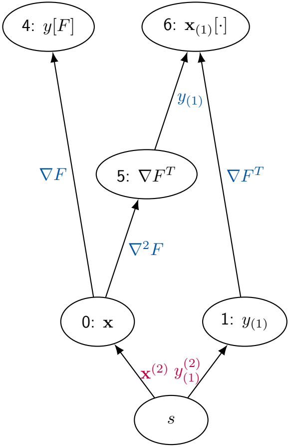


$$y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$$

... 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 $\mathbf{x}_{(1)}^{(2)}$.

Tangents of Adjoints

… on Tangent-Augmented Adjoint DAG



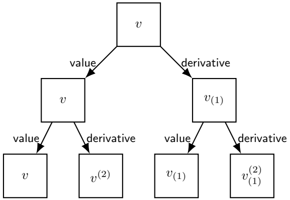
$$
\begin{aligned}
y^{(2)} &\equiv \frac{dy}{ds} \\
&= \frac{dF(\mathbf{x})}{d\mathbf{x}} \cdot \mathbf{x}^{(2)} \\
\mathbf{x}_{(1)}^{(2)} &\equiv \frac{d\mathbf{x}_{(1)}}{ds} \\
&= y_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)} + \nabla F(\mathbf{x})^T \cdot y_{(1)}^{(2)}
\end{aligned}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

1. Apply tangent code generation rules to first-order adjoint code
2. Write appropriate driver
3. Parallelize / vectorize accumulation of the Hessian (optional)

Tangents of Adjoints by dco/c++

Cheat Sheet

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




```
dco::value(dco::value(v))==dco::passive_value(v)
dco::derivative(dco::value(v))
dco::value(dco::derivative(v))
dco::derivative(dco::derivative(v))
```

Tangents of Adjoints by dco/c++

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

```
#include "dco.hpp"
typedef dco::gt1s<double>::type DCO_T; // tangent type
typedef dco::ga1s<DCO_T> DCO_TA_MODE; // tangent of adjoint mode
typedef DCO_TA_MODE::type DCO_TA; // tangent of adjoint type
typedef DCO_TA_MODE::tape_t DCO_TA_TAPE; // tape
typedef DCO_TA_TAPE::position_t DCO_TA_TAPE_POSITION; // tape position

vector<vector<double>> driver(double& xv, const vector<double> &pv,
    const vector<vector<double>>& dW) {
  int n=pv.size();
  vector<DCO_TA> p(n); dco::passive_value(p)=pv;
  vector<vector<double>> ddxdpp(n,vector<double>(n,0));
  DCO_TA_MODE::global_tape=DCO_TA_TAPE::create(); // create tape
  DCO_TA_MODE::global_tape->register_variable(p); // register active input
  DCO_TA_TAPE_POSITION tpos=DCO_TA_MODE::global_tape->get_position(); // mark
  for (int i=0;i<n;i++) {
    dco::derivative(dco::value(p[i]))=1; // seed tangent
    DCO_TA x=xv;
    euler_maruyama(x,p,dW); // overloaded augmented primal
    DCO_TA_MODE::global_tape->register_output_variable(x); // register ...
```

Tangents of Adjoints by dco/c++

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

```
      dco::value(dco::derivative(x))=1; // and seed adjoint output
      DCO_TA_MODE::global_tape->
        interpret_adjoint_and_reset_to(tpos); // propagate
      for (int j=0;j<=i;j++)
        ddxdpp[i][j]=dco::derivative(dco::derivative(p[j])); // harvest
      for (int j=0;j<n;j++) {
        dco::derivative(dco::derivative(p[j]))=0; // reset
        dco::value(dco::derivative(p[j]))=0; // reset
      }
      dco::derivative(dco::value(p[i]))=0; // reset
    }
    DCO_TA_TAPE::remove(DCO_TA_MODE::global_tape); // remove tape
    return ddxdpp;
}
```

Adjoints of Tangents

Computer Scientist's View (Simplified)

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-of-Tangent (AT) 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:

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

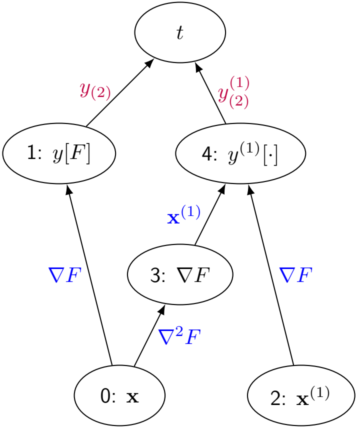
Directional differentiation in adjoint mode of the first-order tangent model

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

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

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

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

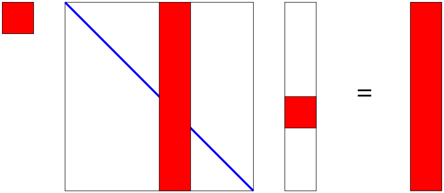


$$\mathbf{x}_{(2)}^{(1)} \equiv \frac{dt}{d\mathbf{x}^{(1)}}^T = \nabla F(\mathbf{x}))^T \cdot y_{(2)}^{(1)}$$

$$\mathbf{x}_{(2)} \equiv \frac{dt}{d\mathbf{x}}^T$$

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

See also Eqn. (117).



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

... 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 $\mathbf{x}_{(2)}$.

Adjoints of Adjoints

Computer Scientist's View (Simplified)

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-of-Adjoint (AA) 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:

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

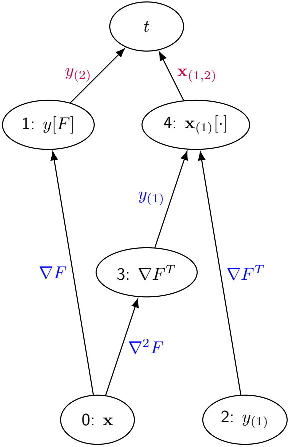
Directional differentiation in adjoint mode of the first-order adjoint model

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

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

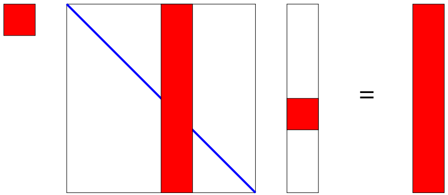
$$\begin{pmatrix} \mathbf{x}_{(2)} \\ y_{(1,2)} \end{pmatrix} \equiv \frac{d\begin{pmatrix} y \\ \mathbf{x}_{(1)} \end{pmatrix}}{d(\mathbf{x}\ y_{(1)})}^T \cdot \begin{pmatrix} y_{(2)} \\ \mathbf{x}_{(1,2)} \end{pmatrix} = \begin{pmatrix} \frac{dy}{d\mathbf{x}}^T \cdot y_{(2)} + \frac{d\mathbf{x}_{(1)}}{d\mathbf{x}}^T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

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



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

See also Eqn. (117).



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

... 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

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

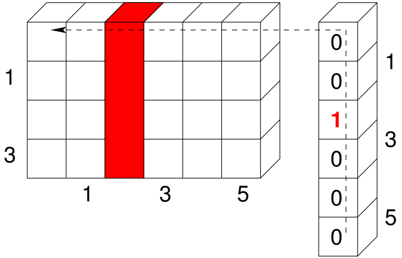
The Hessian is a three-tensor, that is

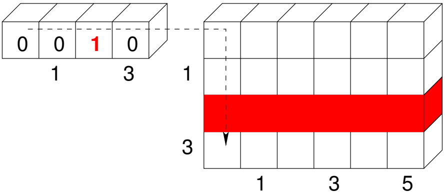
$$\nabla^2 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n}.$$

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

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

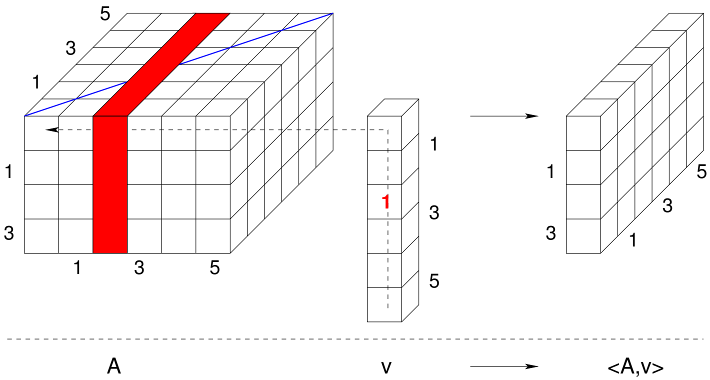
Tangent Projection	Adjoint Projection





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

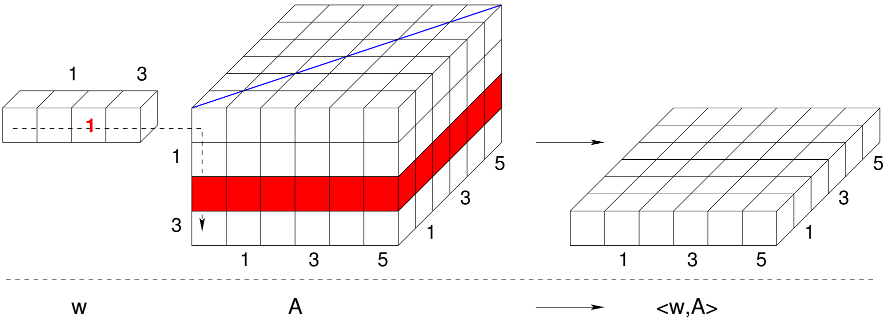
$< w, A > \equiv A^T \cdot w = (w^T \cdot A)^T$



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

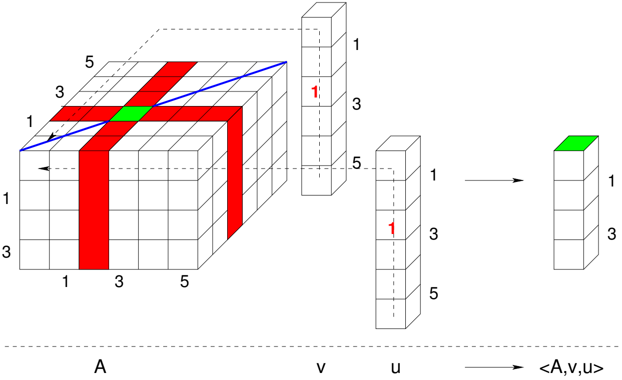
Hessian Projections

(First-Order) Adjoint Projection



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

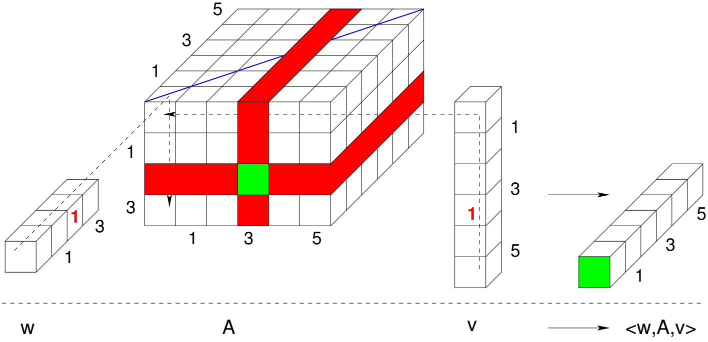
Second-Order Tangent Projection



Note:

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

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



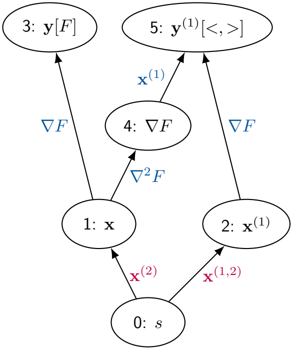
Note:

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

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

Tangents of Tangents

... on Tangent-Augmented Tangent DAG

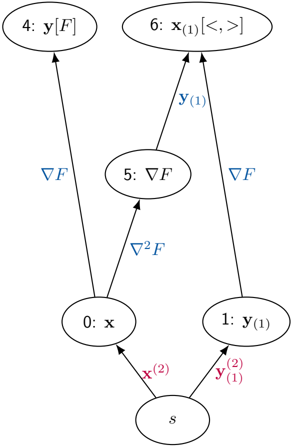


$$
\begin{aligned}
\mathbf{y}^{(1,2)} = \frac{d\mathbf{y}^{(1)}}{ds} &= \frac{d\nabla F \cdot \mathbf{x}^{(1)}}{ds} \\
&= \frac{d < \nabla F, \mathbf{x}^{(1)} >}{ds} \\
&=< \frac{d\nabla F}{ds}, \mathbf{x}^{(1)} > \\
&=<< \frac{d\nabla F}{d\mathbf{x}}, \frac{d\mathbf{x}}{ds} >, \mathbf{x}^{(1)} > \\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

for passive $\mathbf{x}^{(1)}$ $(\mathbf{x}^{(1,2)} = 0)$.

Tangents of Adjoints

… on Tangent-Augmented Adjoint DAG

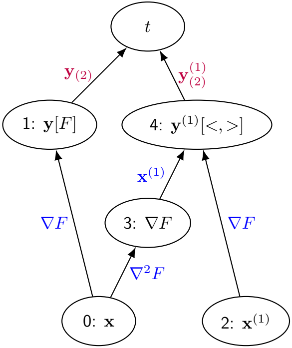


$$
\begin{aligned}
\mathbf{x}_{(1)}^{(2)} &= \frac{d\mathbf{x}_{(1)}}{ds} = \frac{d\nabla F^T \cdot \mathbf{y}_{(1)}}{ds} \\
&= \frac{d < \mathbf{y}_{(1)}, \nabla F >}{ds} \\
&=< \mathbf{y}_{(1)}, \frac{d\nabla F}{ds} > \\
&=< \mathbf{y}_{(1)}, < \frac{d\nabla F}{d\mathbf{x}}, \frac{d\mathbf{x}}{ds}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

for passive $\mathbf{y}_{(1)}$ ($\mathbf{y}_{(1)}^{(2)}$).

Adjoints of Tangents

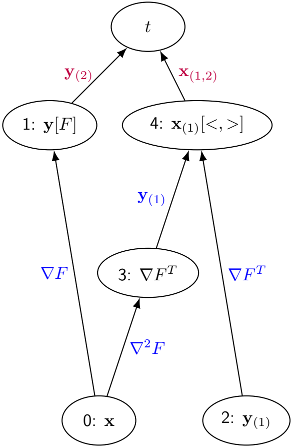
… on Adjoint-Augmented Tangent DAG



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

Adjoints of Adjoints

… on Adjoint-Augmented Adjoint DAG



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

Outlook: Higher Derivatives

e.g, Tangents of Tangents of Adjoints

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

Hands On

For given PDE and/or LIBOR codes ...

- ... write second-order tangent code + driver
- ... use dco/c++ to generate second-order tangent code; write driver
- ... write second-order adjoint code + driver
- ... use dco/c++ to generate second-order adjoint code; write driver
- ... cross-validate, race

Outline

Recall ...

The adjoint of a program $\mathbf{y} = \mathbf{x}_q = F(\mathbf{x} = \mathbf{x}_0)$ computes

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

assuming availability of adjoint elemental functions (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 adjoint AD (AAD) is the implementation of adjoint versions of the intrinsic operations $(+, *, \ldots)$ and functions $(\sin, \exp, \ldots)$ of the given programming language.

Their naive combination yields algorithmic adjoint programs, which may turn out infeasible for various reasons. Hierarchies in granularity and mathematical semantics must be exploited in "real world" AAD.

Adjoint Elementals and Programs

An adjoint elemental $F_{i(1)}$ 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 a partially ordered sequence of evaluations of adjoint elementals.

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

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

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

The tape of the adjoint program is a (partially ordered) concatenation of the tapes of the adjoint elementals. Basic AAD records the entire tape homogeneously based on algorithmic adjoint elementals followed by its use for the propagation of adjoints.

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 $\mathbf{x}_{k-1}$ in the forward section followed by the evaluation of the primal F_k and postponing the generation of the tape for $F_{(1)_k}$ to the reverse section of $F_{(1)}$.

Further examples include the implementation of symbolic adjoint elementals, preaccumulation and approximation of Jacobians of local black boxes by finite differences.

AD of Implicit Functions

Let $F(\mathbf{x}(\mathbf{p}), \mathbf{p}) = 0$ with $F : \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^{n_\mathbf{p}} \to \mathbb{R}^{n_\mathbf{x}}$ continuously differentiable wrt. both $\mathbf{x}$ and $\mathbf{p}$. Then from

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

follows (*Implicit Function Theorem*)

$$\frac{d\mathbf{x}}{d\mathbf{p}} = -\frac{dF}{d\mathbf{x}}^{-1} \cdot \frac{\partial F}{\partial \mathbf{p}}$$

implying tangents

$$\mathbf{x}^{(1)} \equiv \frac{d\mathbf{x}}{d\mathbf{p}} \cdot \mathbf{p}^{(1)} = -\frac{dF}{d\mathbf{x}}^{-1} \cdot \underbrace{\frac{\partial F}{\partial \mathbf{p}} \cdot \mathbf{p}^{(1)}}_{=:\mathbf{z}^{(1)}}$$

and context-free adjoints

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

Tangent Nonlinear Equations

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\boxed{x^2 - p = 0} \arrow[r, "\frac{d}{dp}\cdot p^{(1)}"] \arrow[d, "{(x^*, x^{*(1)}) = S(p, p^{(1)})}"'] & \boxed{\left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) \cdot p^{(1)} = 2 \cdot x \cdot \underbrace{\left(\frac{dx}{dp} \cdot p^{(1)}\right)}_{=x^{(1)}} - p^{(1)} = 0} \arrow[d, "{\tilde{S}(x^*, p^{(1)})}"] \\
\boxed{x^* \approx \sqrt{p}} \arrow[r, "S(p+\Delta p)", "x^{*(1)}"'] & \boxed{= \frac{p^{(1)}}{2 \cdot x} \quad ?}
\end{tikzcd}
$$

Tangent Nonlinear Equations by Hand

Algorithmic Tangent

```
template<typename T> // primal
void newton(T& x, const T& p, const T& eps) {
  while (abs(x*x-p)>eps) x=x-(x*x-p)/(2*x);
}

template<typename T>
void tangent_newton(T& xv, T& xt, const T& pv, const T& pt, const T& eps) {
  while (abs(xv*xv-pv)>eps) {
    xt+=pt/(2*xv)-(3./4.+pv/(4*xv*xv))*xt;
    xv-=(xv*xv-pv)/(2*xv);
  }
}
```

```
template<typename T> // primal
void newton(T& x, const T& p, const T& eps) {
  while (abs(x*x-p)>eps) x=x-(x*x-p)/(2*x);
}

template<typename T> // symbolic tangent
void tangent_newton(const T& xv, T& xt, const T& pt) {
  xt=pt/(2*xv);
}

int main(int c, char* v[]) {
  ...
  newton(xv,pv,eps); // primal
  tangent_newton(xv,xt,pt); // symbolic tangent
  ...
}
```

Adjoint Nonlinear Equations

$$
\begin{tikzcd}
\boxed{x^2 - p = 0} \arrow[r, "x_{(1)} \cdot \frac{d}{dp}"] \arrow[d, "(x^*, G) = \overrightarrow{S}(p)"'] & \boxed{x_{(1)} \cdot \left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) = 2 \cdot x \cdot \left(\underbrace{x_{(1)} \cdot \frac{dx}{dp}}_{=p_{(1)}}\right) - x_{(1)} = 0} \arrow[d, "\tilde{S}(x^*, x_{(1)})"] \\
\boxed{x^* \approx \sqrt{p}} \arrow[r, "S(p + \Delta p)", "\overleftarrow{S}(G, x_{(1)})"'] & \boxed{= \frac{x_{(1)}}{2 \cdot x} \ ?}
\end{tikzcd}
$$

Adjoint Nonlinear Equations by Hand

Algorithmic Adjoint

```
template<typename T>
void adjoint_newton(T& xv, T& xa, const T& pv, T& pa, const T& eps) {
  stack<T> tbr_T;
  int i=0;
  while (abs(xv*xv-pv)>eps) {
    tbr_T.push(xv);
    xv-=(xv*xv-pv)/(2*xv);
    i++;
  }
  double y=xv;
  for (int j=0;j<i;j++) {
    xv=tbr_T.top(); tbr_T.pop();
    pa+=xa/(2*xv);
    xa-=(3./4.+pv/(4*xv*xv))*xa;
  }
  xv=y;
}
```

Adjoint Nonlinear Equations by Hand

Symbolic Adjoint

```
template<typename T> // primal
void newton(T& x, const T& p, const T& eps) {
  while (abs(x*x-p)>eps) x=x-(x*x-p)/(2*x);
}

template<typename T> // symbolic adjoint
void adjoint_newton(const T& xv, T& xa, T& pa) {
  pa+=xa/(2*xv); xa=0;
}

int main(int c, char* v[]) {
  ...
  newton(xv,pv,eps); // primal
  adjoint_newton(xv,xa,pa); // symbolic adjoint
  ...
}
```

AD of Inner Product

Let $y = \boldsymbol{a}^T \cdot \mathbf{x}$.

Tangent

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

Context-Sensitive Adjoint

$$\begin{aligned}
\boldsymbol{a}_{(1)} += y_{(1)} \cdot \mathbf{x} \\
\mathbf{x}_{(1)} += \boldsymbol{a} \cdot y_{(1)} \\
y_{(1)} = 0
\end{aligned}$$

Proof via algorithmic adjoint ...

AD of Matrix-Vector Product

Let $\mathbf{y} = A \cdot \mathbf{x}$.

Tangent

$$\mathbf{y}^{(1)} = A^{(1)} \cdot \mathbf{x} + A \cdot \mathbf{x}^{(1)}$$

Context-Sensitive Adjoint

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

Proof via element-wise inner products ...

AD of Matrix-Matrix Product

Let $Y = A \cdot X$.

Tangent

$$Y^{(1)} = A^{(1)} \cdot X + A \cdot X^{(1)}$$

Context-Sensitive Adjoint

$$\begin{aligned} X_{(1)} &+= A^T \cdot Y_{(1)} \\ A_{(1)} &+= Y_{(1)} \cdot X^T \\ Y_{(1)} &= 0 \end{aligned}$$

Proof via (concurrent) column-wise matrix-vector products

Generalization

Let $A \in \mathbb{R}^{m \times n}$, $X^{(1)} \in \mathbb{R}^{n \times q}$, and $B \in \mathbb{R}^{q \times p}$. Then $Y^{(1)} \in \mathbb{R}^{m \times p}$ and

$$Y^{(1)} = A \cdot X^{(1)} \cdot B \quad \Rightarrow \quad X_{(1)} = A^T \cdot Y_{(1)} \cdot B^T$$

for $Y_{(1)} \in \mathbb{R}^{m \times p}$ and $X_{(1)} \in \mathbb{R}^{n \times q}$.

Proof: From matrix-matrix product ...

$$Z^{(1)} = A \cdot X^{(1)} \quad \Leftrightarrow \quad X_{(1)} = A^T \cdot Z_{(1)}$$

$$Y^{(1)} = Z^{(1)} \cdot B \quad \Leftrightarrow \quad Z_{(1)} = Y_{(1)} \cdot B^T$$

and substitution. ■

AD of Linear Systems

Let $A\mathbf{x} = \mathbf{b}$ with invertable $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^n$ and $\mathbf{x} = \mathbf{x}(A, \mathbf{b}) \in \mathbb{R}^n$.

Eqn. (118) yields tangents $\mathbf{x}^{(1)}$ as solutions of the linear system

$$A \cdot \mathbf{x}^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{x} \, .$$

Context-free adjoints follow immediately from $\mathbf{x}^{(1)} = A^{-1} \cdot \mathbf{b}^{(1)} - A^{-1} \cdot A^{(1)} \cdot \mathbf{x}$ as

$$\mathbf{b}_{(1)} = A^{-T} \cdot \mathbf{x}_{(1)}$$
$$A_{(1)} = (-A^{-T} \cdot \mathbf{x}_{(1)} \cdot \mathbf{x}^T =) - \mathbf{b}_{(1)} \cdot \mathbf{x}^T \, .$$

AD of Nonlinear Systems

Let $F(\mathbf{x}(\mathbf{p}),\mathbf{p})=0$ with $F:\mathbb{R}^{n_\mathbf{x}}\times\mathbb{R}^{n_\mathbf{p}}\rightarrow\mathbb{R}^{n_\mathbf{x}}$ continuously differentiable wrt. both $\mathbf{x}$ and $\mathbf{p}$.

Tangents and adjoints are defined at the solution as

$$\mathbf{x}^{(1)}=-\frac{dF}{d\mathbf{x}}^{-1}\cdot\frac{\partial F}{\partial\mathbf{p}}\cdot\mathbf{p}^{(1)}$$

i.e, as solution of linear system $\frac{dF}{d\mathbf{x}}\cdot\mathbf{x}^{(1)}=-\frac{\partial F}{\partial\mathbf{p}}\cdot\mathbf{p}^{(1)}$, and

$$\mathbf{p}_{(1)}=-\frac{\partial F}{\partial\mathbf{p}}^{T}\cdot\frac{dF}{d\mathbf{x}}^{-T}\cdot\mathbf{x}_{(1)}\ ,$$

i.e, as solution of linear system $\frac{dF}{d\mathbf{x}}^{T}\cdot\mathbf{z}_{(1)}=-\mathbf{x}_{(1)}$ followed by evaluation of the adjoint $\mathbf{p}_{(1)}=\frac{\partial F}{\partial\mathbf{p}}^{T}\cdot\mathbf{z}_{(1)}$.

U.N., K. Leppkes, J. Lotz, M. Towara: Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. ACM Trans. Math. Soft., 2015.

Let $\frac{df(\mathbf{x}(\mathbf{p}),\mathbf{p})}{d\mathbf{x}} = 0$ with $\frac{df}{d\mathbf{x}} : \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^{n_\mathbf{p}} \to \mathbb{R}^{n_\mathbf{x}}$ continuously differentiable wrt. both $\mathbf{x}$ and $\mathbf{p}$.

Tangents and adjoints are defined at the solution as

$$\mathbf{x}^{(1)} = -\frac{d^2 f}{d\mathbf{x}^2}^{-1} \cdot \frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{p}} \cdot \mathbf{p}^{(1)}$$

i.e, as solution of linear system $\frac{d^2 f}{d\mathbf{x}^2} \cdot \mathbf{x}^{(1)} = -\frac{\partial f^2}{\partial \mathbf{x} \partial \mathbf{p}} \cdot \mathbf{p}^{(1)}$, and

$$\mathbf{p}_{(1)} = -\frac{\partial f^2}{\partial \mathbf{x} \partial \mathbf{p}}^T \cdot \frac{df^2}{d\mathbf{x}^2}^{-1} \cdot \mathbf{x}_{(1)} ,$$

i.e, as solution of linear system $\frac{d^2 f}{d\mathbf{x}^2} \cdot \mathbf{z}_{(1)} = -\mathbf{x}_{(1)}$ followed by evaluation of the second-order adjoint $\mathbf{p}_{(1)} = \frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{p}}^T \cdot \mathbf{z}_{(1)}$.

Algorithmic Adjoint ODE

... over Symbolic Adjoint Nonlinear System I

Implicit Euler integration of the ODE

$$\frac{d\mathbf{x}}{dt} = G(\mathbf{x})$$

for $\mathbf{x} \in \mathbb{R}^n$ and with given initial value $\mathbf{x}(0) = \mathbf{x}^0$ yields

$$\frac{\mathbf{x}^i - \mathbf{x}^{i-1}}{t^i - t^{i-1}} = G(\mathbf{x}^i)$$

and hence the solution of the nonlinear system

$$F(\mathbf{x}^i, \mathbf{x}^{i-1}) = \mathbf{x}^i - \mathbf{x}^{i-1} - (t^i - t^{i-1}) \cdot G(\mathbf{x}^i) = 0$$

for $i = 1, \ldots, m$. To evaluate adjoints of the ODE's final wrt. initial condtions the symbolic adjoint nonlinear system

$$\frac{dF}{d\mathbf{x}^i}^T \cdot \mathbf{x}_{(1)}^{i-1} = \left(I_n - (t^i - t^{i-1}) \cdot \frac{dG}{d\mathbf{x}^i}^T \right) \cdot \mathbf{x}_{(1)}^{i-1} = \mathbf{x}_{(1)}^i$$

needs to be solved for $i = m, \ldots, 1$ as

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

is followed by evaluation of the adjoint

$$\mathbf{x}^{i-1}_{(1)} = \frac{\partial F}{\partial \mathbf{x}^{i-1}}^T \cdot \mathbf{z}_{(1)} = -I_n \cdot \mathbf{z}_{(1)}$$

implying

$$\begin{aligned}
-I_n \cdot \mathbf{x}^{i-1}_{(1)} &= \mathbf{z}_{(1)} \\
\frac{dF}{d\mathbf{x}^i}^T \cdot (-I_n) \cdot \mathbf{x}^{i-1}_{(1)} &= -\mathbf{x}^i_{(1)} \\
-I_n \cdot \frac{dF}{d\mathbf{x}^i}^T \cdot \mathbf{x}^{i-1}_{(1)} &= -\mathbf{x}^i_{(1)} \\
\frac{dF}{d\mathbf{x}^i}^T \cdot \mathbf{x}^{i-1}_{(1)} &= \mathbf{x}^i_{(1)} \ .
\end{aligned}$$

Symbolic Adjoint ODE

... turns out to be the same ... I

Algorithmic Differentiation of the explicit Euler scheme

$$\mathbf{x}^{i+1} := \mathbf{x}^i + (t^{i+1} - t^i) \cdot G(\mathbf{x}^i), \quad i = 0, \ldots, m-1$$

for the primal ODE in adjoint mode yields

$$\mathbf{x}^i_{(1)} := \mathbf{x}^{i+1}_{(1)} + (t^{i+1} - t^i) \cdot \frac{dG}{d\mathbf{x}}^T (\mathbf{x}^i) \cdot \mathbf{x}^{i+1}_{(1)}$$

for $i = m-1, \ldots, 0$ and hence

$$\frac{\mathbf{x}^i_{(1)} - \mathbf{x}^{i+1}_{(1)}}{t^i - t^{i+1}} = -\frac{dG}{d\mathbf{x}}^T (\mathbf{x}^i) \cdot \mathbf{x}^{i+1}_{(1)}$$

that is, for $m \to \infty$ $(t^{i+1} \to t^i)$ the explicit Euler scheme for the adjoint ODE

$$\frac{d\mathbf{x}_{(1)}}{dt} = -\frac{dG}{d\mathbf{x}}^T \cdot \mathbf{x}_{(1)}, \quad \mathbf{x}^m_{(1)} = \mathbf{x}_{(1)}(T) \; .$$

Note that the primal $\mathbf{x}^i$ are accessed in reverse order of their computation. Implicit Euler integration yields

$$\frac{\mathbf{x}^i_{(1)} - \mathbf{x}^{i+1}_{(1)}}{t^i - t^{i+1}} = -\frac{dG}{d\mathbf{x}}^T(\mathbf{x}^i) \cdot \mathbf{x}^i_{(1)}$$

and hence $\mathbf{x}^i_{(1)}$ as the solution of the linear system

$$\mathbf{x}^i_{(1)} + (t^i - t^{i+1}) \cdot \frac{dG}{d\mathbf{x}}^T(\mathbf{x}^i) \cdot \mathbf{x}^i_{(1)} = \left(I + (t^i - t^{i+1}) \cdot \frac{dG}{d\mathbf{x}}^T(\mathbf{x}^i)\right) \cdot \mathbf{x}^i_{(1)} =$$

$$\left(I - (t^{i+1} - t^i) \cdot \frac{dG}{d\mathbf{x}}^T(\mathbf{x}^i)\right) \cdot \mathbf{x}^i_{(1)} = \mathbf{x}^{i+1}_{(1)}$$

for $i = m-1, \ldots, 0$. Note equivalence of symbolic adjoint ODE to its algorithmic adjoint over symbolic adjoint nonlinear solver.

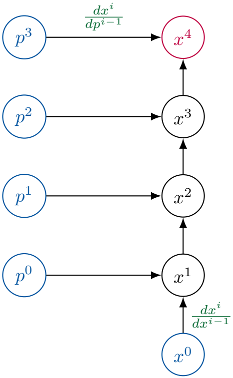
Explicit Euler integration of the adjoint ODE yields

$$\begin{aligned}\mathbf{x}_{(1)}^{i-1} &= \mathbf{x}_{(1)}^{i} - ((t^{i-1} - t^{i}) \cdot \frac{dG}{d\mathbf{x}}^{T}(\mathbf{x}^{i}) \cdot \mathbf{x}_{(1)}^{i} \\ &= \mathbf{x}_{(1)}^{i} + ((t^{i} - t^{i-1}) \cdot \frac{dG}{d\mathbf{x}}^{T}(\mathbf{x}^{i}) \cdot \mathbf{x}_{(1)}^{i}\end{aligned}$$

for $i = m-1, \ldots, 0$.

Hands-On

Implement symbolic tangent and adjoint versions of the given implict Euler PDE code.



(PMR,POC)

$(5,0) \leftarrow$ register $x^0, \mathbf{p}$

$(8,1) \leftarrow$ compute x^1 and record

$(11,2) \leftarrow$ compute x^2 and record

$(14,3) \leftarrow$ compute x^3 and record

$(\mathbf{17},4) \leftarrow$ compute x^4 and record; return x^4; set $x^4_{(1)}$

$(14,4) \leftarrow$ compute $x^3_{(1)}, p^3_{(1)}$; release x_4

$(11,4) \leftarrow$ compute $x^2_{(1)}, p^2_{(1)}$; release x_3

$(8,4) \leftarrow$ compute $x^1_{(1)}, p^1_{(1)}$; release x_2

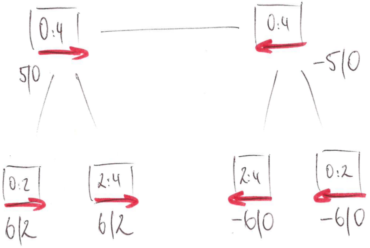
$(5,4) \leftarrow$ compute $x^0_{(1)}, p^0_{(1)}$; release x_1

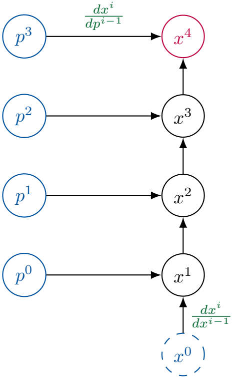
$(0,\mathbf{4}) \leftarrow$ return $x^0_{(1)}, \mathbf{p}_{(1)}$, release $x_0, \mathbf{p}$

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

Checkpointing

Motivation: Store-All





(PMR,POC)

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

$(9,4) \leftarrow$ compute x^3; compute x^4 and record; return x^4

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

$(9,7) \leftarrow$ restore x^0; compute x^2; compute x^3 and record

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

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

$(6,9) \leftarrow$ compute $x^1_{(1)}, p^1_{(1)}$; release x^2

$(8,10) \leftarrow$ restore and free x^0; compute x^1 and record

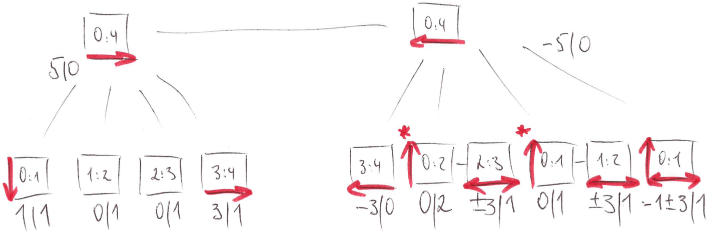
$(5,10) \leftarrow$ compute $x^0_{(1)}, p^0_{(1)}$; release x^1

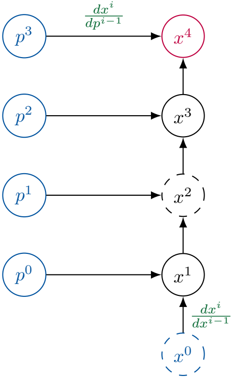
$(0,\mathbf{10}) \leftarrow$ return $x^0_{(1)}, \mathbf{p}_{(1)}$; release x^0, $\mathbf{p}$

Note: Recording single steps

Checkpointing

Motivation: Recompute-All





(PMR,POC)

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

$(6,2) \leftarrow$ compute x^2

$(\mathbf{12},4) \leftarrow$ compute x^4 and record

$(9,4) \leftarrow$ set $x^4_{(1)}$; compute $x^3_{(1)}, p^3_{(1)}$; release x^4

$(6,4) \leftarrow$ compute $x^2_{(1)}, p^2_{(1)}$; release x^3

$(11,6) \leftarrow$ restore and free x^0; compute x^2 and record

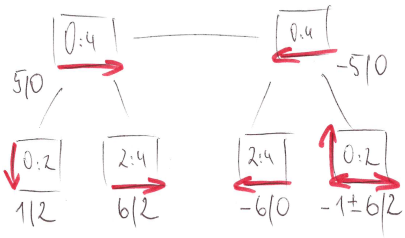
$(8,6) \leftarrow$ compute $x^1_{(1)}, p^1_{(1)}$; release x^2

$(5,6) \leftarrow$ compute $x^0_{(1)}, p^0_{(1)}$; release x^1

$(0,\mathbf{6}) \leftarrow$ return $x^0_{(1)}, \mathbf{p}_{(1)}$; release x^0, $\mathbf{p}$

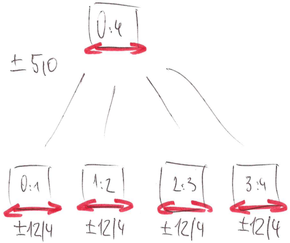
Checkpointing

Motivation: Trade-Off



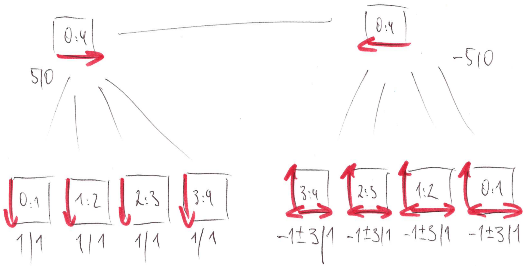
Checkpointing Ensembles

Recall: Pathwise Adjoint SDE



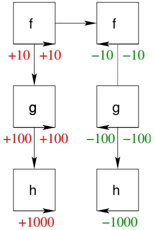
Checkpointing Evolutions

Live: Pathwise Checkpointed Adjoint SDE



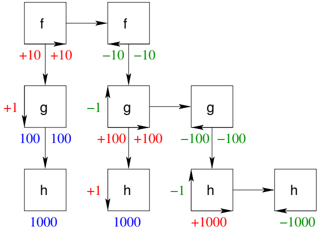
Example: Let $\overline{\text{PMR}} = 1110$ ...

R=(0,0):



PMR=1220, POC=0

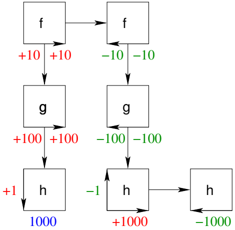
R=(1,1):



PMR=1110, POC=2200

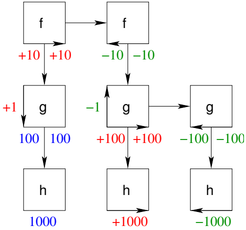
Smallest Memory Increase starts from R $= \mathbf{1}$ and yields ...
Largest Memory Decrease (LMD) starts from R $= \mathbf{0}$ and yields ...

R=(0,1):



PMR=1110, POC=1000

R=(1,0):



PMR=1120, POC=1200

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

Hands On

Implement an equidistant checkpointing scheme for given algorithmic adjoint PDE code.

Outline

Further AAD

- AAD on GPUs / meta-adjoint programming
- Adjoint code design patterns
- NAG AD Library
- Further dco/c++
  - file tape
  - multiple (e.g, thread-local) tapes
  - multiple (e.g, thread-local) adjoint vectors and (parallel) interpretation of same tape
  - minimum number of adjoint program variables
  - just-in-time code generation and compilation / linking
  - inner product invariance debugging

Outline

Conclusion

The quality of an adjoint AD solution / tool is defined by

- robustness wrt. language features of target code
- efficiency of adjoint propagation
- flexibility wrt. design scenarios
- sustainability wrt. dynamics in user requirements, personnel, hard- and software

AD is fun ...

“Optimality”

I

I’m sittin’ in front of the computer screen.
Newton’s second iteration is what I’ve just seen.
It’s not quite the progress that I would expect
from a code such as mine – no doubt it must be perfect!
Just the facts are not supportive, and I wonder ...

My linear solver is state-of-the-art.
It does not get better wherever I start.
For differentiation is there anything else?
Perturbing the inputs – can’t imagine this fails.
I pick a small Epsilon, and I wonder ...

I wonder how, but I still give it a try.
The next change in step size is bound to fly.
’cause all I’d like to see is simply optimality.
Epsilon, in fact, appears to be rather small.
A factor of ten should improve it all.

“Optimality”

II

’cause all I’d like to see is nearly optimality.

A DAD ADADA DAD ADADA DADAD.

A few hours later my talk’s getting rude.
The sole thing descending seems to be my mood.
How can guessing the Hessian only take this much time?
N squared function runs appear to be the crime.
The facts support this thesis, and I wonder ...

Isolation due to KKT
Isolation – why not simply drop feasibility?

The guy next door’s been sayin’ again and again:
An adjoint Lagrangian might relieve my pain.
Though I don’t quite believe him, I surrender.

I wonder how but I still give it a try:

“Optimality” III

Gradients and Hessians in the blink of an eye.
Still all I’d like to see is simply optimality.
Epsilon itself has finally disappeared.
Reverse mode AD works, no matter how weird,
and I’m about to see local optimality.

Yes, I wonder, I wonder ...

I wonder how but I still give it a try:
Gradient and Hessians in the blink of an eye.
Still all I’d like to see ...
I really need to see ...
now I can finally see my cherished optimality :-)

www.stce.rwth-aachen.de/research/the-art

Follow-up

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

naumann@stce.rwth-aachen.de

and

https://www.nag.co.uk/content/nag-and-algorithmic-differentiation