Algorithmic Differentiation with dco/c++

Uwe Naumann

NAG Ltd. and RWTH Aachen University

About Myself

- Member (since 2008), Principal Scientist (2022), NAG Ltd., Oxford, UK
- Visiting Lecturer, University of Oxford, UK (2017)
- Professor of Computer Science, RWTH Aachen University, Aachen, Germany (since 2004)
- Assistant Computer Scientist, Argonne National Laboratory, Argonne, IL, US (2002–2004)
- Visiting Scientist, MIT, Cambridge, MA, US (2001)
- Senior Lecturer for Computer Science, University of Hertfordshire (UHerts), Hatfield, UK (2000–2001) $\Rightarrow$ Visiting Researcher
- Postdoctoral Researcher, INRIA, Sophia-Antipolis, France (1999–2000)
- MSc/PhD in Applied Mathematics, Technical University Dresden, Germany, University of York, UK, Chuo University, Tokyo. Japan, UHerts, UK (1990–1999, mentor: Andreas Griewank
- Military service in Bad Düben, East Germany (1988–1990)
- went to school in Chemnitz, East Germany and Dubna, Russia (1976–1988)

Contents

Outline

Motivation: Differentiable Programs

We consider sufficiently often differentiable computer programs implementing multivariate vector functions

$$F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^m : y = F(x, p)$$

with Jacobians

$$F' = F'(x, p) = (y_x, y_p) \equiv \left(\frac{dF}{dx}, \frac{dF}{dp} \right) \in \mathbb{R}^{m \times (n_x + n_p)} \ ,$$

Motivation: Differentiable Programs

We consider sufficiently often differentiable computer programs implementing multivariate vector functions

$$F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^m : y = F(x, p)$$

with Jacobians

$$F' = F'(x, p) = (y_x, y_p) \equiv \left(\frac{dF}{dx}, \frac{dF}{dp} \right) \in \mathbb{R}^{m \times (n_x + n_p)} ,$$

for example,

```
std::vector<double> F(
    const std::vector<double>& x, // state
    const std::vector<double>& p  // parameters
);
```

p is omitted whenever possible to simplify the notation $\Rightarrow y = F(x)$.

Motivation: Playground

Examples and live coding sessions are based on an implementation of the numerical approximation of the expected solution $\mathbb{E}(x) \in \mathbb{R}$ of a parameterized scalar stochastic initial value problem.

We consider the generalized Geometric Brownian Motion (gGBM) described by the stochastic differential equation (SDE)

$$dx = f_1(x(p_1(t), t), p_1(t), t))dt + f_2(x(p_2(t), t), p_2(t), t)dW$$

with drift and volatility parameterized by time-dependent $p_1 = p_1(t)$ and $p_2 = p_2(t)$, respectively, initial condition $x(0) = x^0$, unit target time $t = 1$ and Wiener Process dW.

Playground

Application of forward finite differences in time with time step $0 < \Delta t \ll 1$ to the SDE yields the Euler-Maruyama scheme

$$x^{i+1} = x^i + \Delta t \cdot f_1(x^i, p_1^i, i \cdot \Delta t) + \sqrt{\Delta t} \cdot f_2(x^i, p_2^i, i \cdot \Delta t) \cdot dW^i$$

for $i = 0, \ldots, n_s - 1$, $n_s > 0$, target time $n_s \cdot \Delta t = 1$, parameter vectors $p_j = (p_j^i) \in \mathbb{R}^{n_s}$, $j = 1, 2$, and with random numbers dW^i drawn from the standard normal distribution $N(0, 1)$.

Playground

Application of forward finite differences in time with time step $0 < \Delta t \ll 1$ to the SDE yields the Euler-Maruyama scheme

$$x^{i+1} = x^i + \Delta t \cdot f_1(x^i, p_1^i, i \cdot \Delta t) + \sqrt{\Delta t} \cdot f_2(x^i, p_2^i, i \cdot \Delta t) \cdot dW^i$$

for $i = 0, \ldots, n_s - 1$, $n_s > 0$, target time $n_s \cdot \Delta t = 1$, parameter vectors $p_j = (p_j^i) \in \mathbb{R}^{n_s}$, $j = 1, 2$, 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 $n_p > 0$ Euler-Maruyama paths.

We are interested in first- (and higher-order) sensitivities of

$$y = \mathbb{E}(x)$$

wrt. x^0, p_1, and p_2 to simulate use in the context of calibration.

Playground

```
1  template <typename T>
2  T f(
3      int np, // number of paths
4      int ns, // number of time steps per path
5      T x,                    // state
6      const vector<T>& p // parameters
7  ) {
8    default_random_engine g;
9    normal_distribution<float> d(0,1);
10   T y=0, x_in=x;
11   for (int i=0;i<np;++i) { // ensemble
12     x=x_in;
13     for (int j=0;j<ns;++j) // evolution
14       x+=exp(-p[j]*x)/ns+p[ns+j]*x*sqrt(1./ns)*d(g);
15     y+=x;
16   }
17   return y/np;
18 }
```

Playground

`SDE/main.cpp`

- inspect `SDE/f.h`
- build (`Makefile`)
- run
- time (`/usr/bin/time -v`)

np	ns	ERT (s)	RSS (MB)
10^5	10	< 0.1	4
10^5	100	0.2	4
10^5	1000	2.5	4
10^6	10	0.2	4
10^6	100	2.5	4
10^6	1000	24.4	4

Hands-on: Run and time on your computer.

ERT : Elapsed Run Time
RSS : Resident Set Size

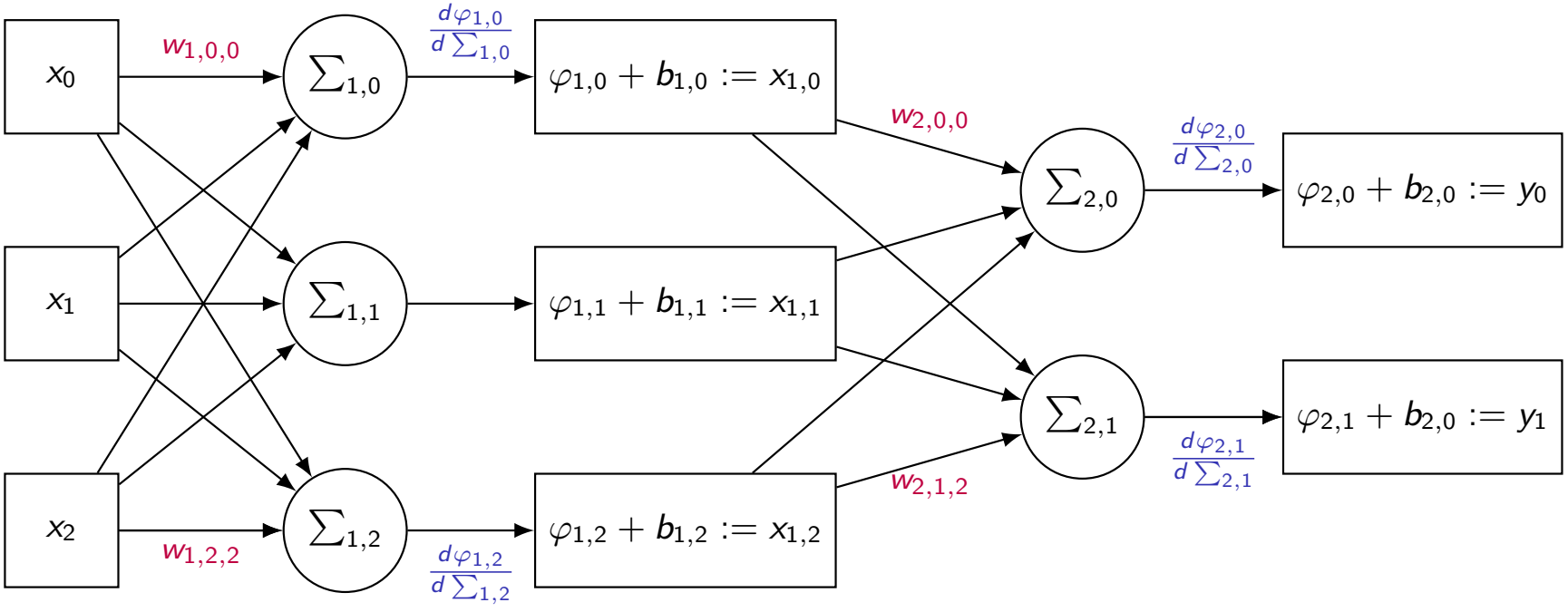
Motivation: Computational Cost of Gradients

$$f : \mathbb{R}^n \to \mathbb{R} \quad \Rightarrow \quad f' \in \mathbb{R}^{1 \times n}$$

	n	ERT (s)	RSS (MB)
primal	200	0.2	4
central finite differences	200	99.3	4
scalar tangent AD	200	52.9	4
black-box adjoint AD	200	0.8	1,061
vector tangent AD	200	5.8	4
primal	400	0.5	4
black-box adjoint AD	400	-	$> 16,000$
vector tangent AD	400	23.5	5
beyond black-box adjoint AD	400	1.5	5

Generalized Geometric Brownian Motion example with $n_p = 10^5$, $n_s = 100,200$

Example: (Deep) Artificial Neural Networks



Example: Backpropagation

Application of the chain rule to $F : \mathbb{R}^n \to \mathbb{R}^m$ defined as

$$y = \underbrace{F_3(F_2(\underbrace{F_1(x)}_{u \in \mathbb{R}^p}))}_{v \in \mathbb{R}^q} \quad \text{yields}$$

$$\frac{dy}{dx} = \frac{dy}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{dx} = \frac{dy}{dv} \cdot \left(\frac{dv}{du} \cdot \frac{du}{dx} \right) = \left(\frac{dy}{dv} \cdot \frac{dv}{du} \right) \cdot \frac{du}{dx}$$

due to associativity of matrix multiplication.

Example: Backpropagation

Application of the chain rule to $F : \mathbb{R}^n \to \mathbb{R}^m$ defined as

$$y = \underbrace{F_3(F_2(\underbrace{F_1(x)}_{u \in \mathbb{R}^p}))}_{v \in \mathbb{R}^q} \quad \text{yields}$$

$$\frac{dy}{dx} = \frac{dy}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{dx} = \frac{dy}{dv} \cdot \left(\frac{dv}{du} \cdot \frac{du}{dx} \right) = \left(\frac{dy}{dv} \cdot \frac{dv}{du} \right) \cdot \frac{du}{dx}$$

due to associativity of matrix multiplication.

However, assuming dense local Jacobians for $n = 10$, $p = 10$, $q = 10$, and $m = 1$, the respective operations counts (Ops; e.g. measured in terms of fused multiply-adds) compare as

$$\text{Ops}\left(\frac{dy}{dv} \cdot \left(\frac{dv}{du} \cdot \frac{du}{dx} \right) \right) = 1100 > 200 = \text{Ops}\left(\left(\frac{dy}{dv} \cdot \frac{dv}{du} \right) \cdot \frac{du}{dx} \right) .$$

Motivation: Adjoint Algorithmic Differentiation

Adjoint algorithmic differentiation (AD) generalizes backpropagation for arbitrary differentiable programs.

The C++ library dco/c++ supports AD more generally including

1. first-order tangents and adjoints over generic base data types
2. combinations thereof at arbitrary levels of the target code
3. second- and higher-order tangents and adjoints.

Very good performance can be achieved, that is,

$$\mathcal{R} \equiv \frac{\textsc{Ert}(f')}{\textsc{Ert}(f)} < 10$$

Bigger Picture

At NAG, we aim for a wholistic approach to (adjoint) AD of parameterized (financial) models.

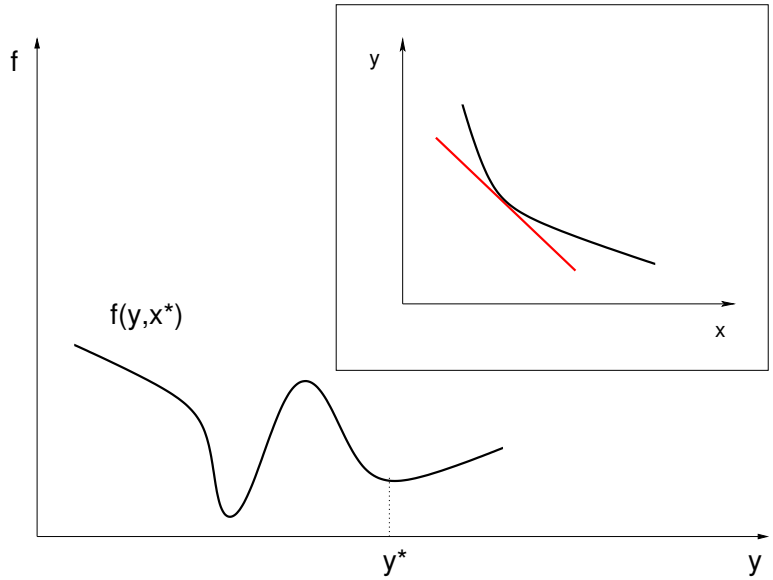
For example;

$$y(x) : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$$

defined as results y^* of calibration

$$\min_{y \in \mathbb{R}^{n_y}} f(x = x^*, y)$$

(e.g., to given market data) and subject to (a payoff) $P(y(x)) : \mathbb{R}^{n_x} \to \mathbb{R}$ yielding

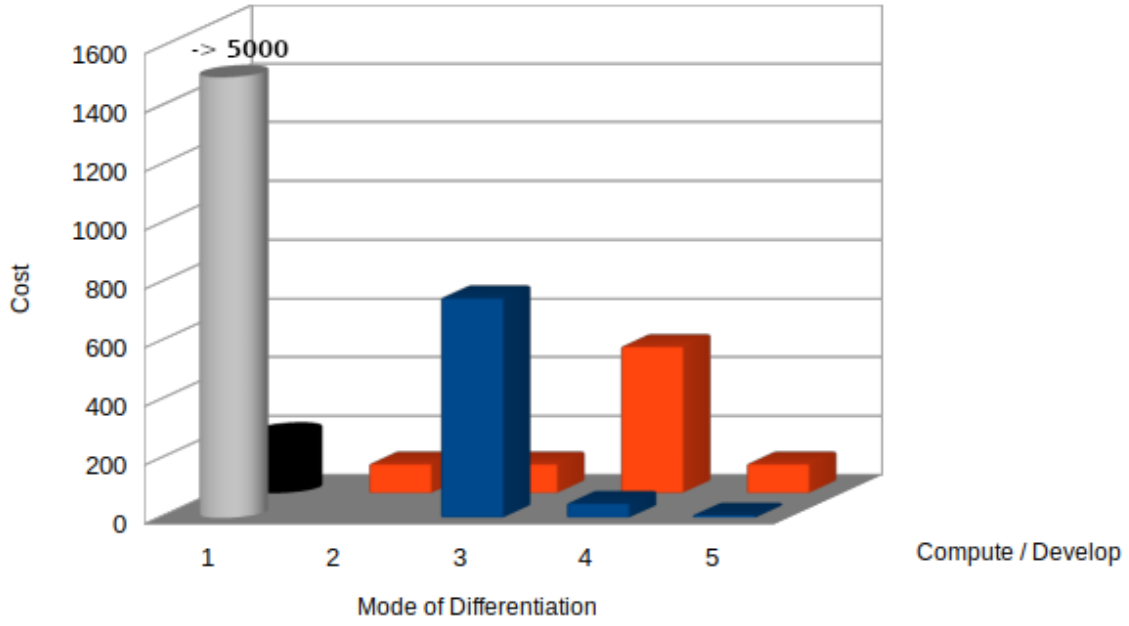


$$\frac{dP}{dy}(y^*) \cdot \frac{dy}{dx}(x) = \underbrace{P_y \cdot y_x \cdot I_{n_x}}_{\text{tangent} \approx \text{bump}} = \underbrace{1 \cdot P_y \cdot y_x}_{\text{adjoint}}$$

U.N.: *The Art of Differentiating Computer Programs ...* SIAM, 2012.

Bigger Picture: Computational vs. Development Cost

$$\underbrace{P_y(y^*)}_{\text{given}} \cdot y_x$$



1. Central finite differences
2. Black-box algorithmic adjoint
3. Robust black-box algorithmic adjoint (file tape)
4. Pathwise algorithmic adjoint
5. Symbolic adjoint

... with dco/c++

K.Leppkes, J. Lotz, U.N.: *dco/c++*. NAG TR2/20. `nag.com`

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)
2. Tangent (for accuracy, validation; $\mathcal{R} = \mathcal{O}(n_x)$)

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)
2. Tangent (for accuracy, validation; $\mathcal{R} = \mathcal{O}(n_x)$)
3. Feasible black-box adjoint ((relatively) cheap gradients; $\mathcal{R} \approx 100$)

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)
2. Tangent (for accuracy, validation; $\mathcal{R} = \mathcal{O}(n_x)$)
3. Feasible black-box adjoint ((relatively) cheap gradients; $\mathcal{R} \approx 100$)
4. Scalable adjoint (scalable cheap gradients; $\mathcal{R} \ll 100$)
   - 4.1 pathwise adjoints; early back-propagation
   - 4.2 preaccumulation; early/late preaccumulation
   - 4.3 checkpointing; late recording

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)
2. Tangent (for accuracy, validation; $\mathcal{R} = \mathcal{O}(n_x)$)
3. Feasible black-box adjoint ((relatively) cheap gradients; $\mathcal{R} \approx 100$)
4. Scalable adjoint (scalable cheap gradients; $\mathcal{R} \ll 100$)
   - 4.1 pathwise adjoints; early back-propagation
   - 4.2 preaccumulation; early/late preaccumulation
   - 4.3 checkpointing; late recording
5. Optimized adjoint (efficient scalable cheap gradients; $\mathcal{R} < 10$)
   - 5.1 (local) symbolic adjoints / extended set of elementals
   - 5.2 (local) parallelization / vectorization
   - 5.3 (local) adjoint code generation
   - 5.4 full combinatorics

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)
2. Tangent (for accuracy, validation; $\mathcal{R} = \mathcal{O}(n_x)$)
3. Feasible black-box adjoint ((relatively) cheap gradients; $\mathcal{R} \approx 100$)
4. Scalable adjoint (scalable cheap gradients; $\mathcal{R} \ll 100$)
   - 4.1 pathwise adjoints; early back-propagation
   - 4.2 preaccumulation; early/late preaccumulation
   - 4.3 checkpointing; late recording
5. Optimized adjoint (efficient scalable cheap gradients; $\mathcal{R} < 10$)
   - 5.1 (local) symbolic adjoints / extended set of elementals
   - 5.2 (local) parallelization / vectorization
   - 5.3 (local) adjoint code generation
   - 5.4 full combinatorics
6. Efficient scalable higher-order tangents and adjoints

Bigger Picture: Typical Work Flow

1. Finite difference approximation (given; relative runtime $\mathcal{R} = \mathcal{O}(n_x)$)
2. Tangent (for accuracy, validation; $\mathcal{R} = \mathcal{O}(n_x)$)
3. Feasible black-box adjoint ((relatively) cheap gradients; $\mathcal{R} \approx 100$)
4. Scalable adjoint (scalable cheap gradients; $\mathcal{R} \ll 100$)
   - 4.1 pathwise adjoints; early back-propagation
   - 4.2 preaccumulation; early/late preaccumulation
   - 4.3 checkpointing; late recording
5. Optimized adjoint (efficient scalable cheap gradients; $\mathcal{R} < 10$)
   - 5.1 (local) symbolic adjoints / extended set of elementals
   - 5.2 (local) parallelization / vectorization
   - 5.3 (local) adjoint code generation
   - 5.4 full combinatorics
6. Efficient scalable higher-order tangents and adjoints

We aim to avoid naive application of our AD software.

Outline

Tangent AD

- Single assignment code
- Directed acyclic graph
- Chain rule of differentiation
- Tangents
- Approximate Tangents (Finite Differences)
- Tangent AD with dco/c++

Single Assignment Code

For given values of its inputs all differentiable programs decompose into sequences of $q = p + m$ differentiable elemental functions (also: elementals) φ_j evaluated conceptually as a single assignment code (SAC)

$$v_j = \varphi_j(v_k)_{k \prec j} \quad \text{for } j = n+1, \ldots, n+q$$

and where $v_i = x_{i-1}$ for $i = 1, \ldots, n$, $y_{k-1} = v_{n+p+k}$ for $k = 1, \ldots, m$ and $k \prec j$ if v_k is an argument of φ_j,

Single Assignment Code

For given values of its inputs all differentiable programs decompose into sequences of $q = p + m$ differentiable elemental functions (also: elementals) φ_j evaluated conceptually as a single assignment code (SAC)

$$v_j = \varphi_j(v_k)_{k \prec j} \quad \text{for } j = n+1, \ldots, n+q$$

and where $v_i = x_{i-1}$ for $i = 1, \ldots, n$, $y_{k-1} = v_{n+p+k}$ for $k = 1, \ldots, m$ and $k \prec j$ if v_k is an argument of φ_j, e.g.

```
1  template<typename T>
2  void F(std::vector<T>& x) {
3    while (x[1]<=0) x[1]=exp(x[0]*x[1])+x[1];
4    x[0]=x[0]/x[1];
5    x[1]=pow(x[1],3);
6  }
```

called with x[0]=x[1]=-1 $\Rightarrow$ single iteration of **while**-loop.

Directed Acyclic Graph

The data dependences within a differential program $F = F(x)$ induce a directed acyclic graphs (DAG), in the following referred to as the tape $T = T(F, x) = (V, E)$ with integer vertices $i \in V = \{1, \ldots, |V|\}$ representing (instances of vectors of program) variables v_i and edges $(i, j) \in E \subseteq V \times V$.

The flow of control in F is determined by the given x.

Directed Acyclic Graph

The data dependences within a differential program $F = F(x)$ induce a directed acyclic graphs (DAG), in the following referred to as the tape $T = T(F, x) = (V, E)$ with integer vertices $i \in V = \{1, \ldots, |V|\}$ representing (instances of vectors of program) variables v_i and edges $(i, j) \in E \subseteq V \times V$.

The flow of control in F is determined by the given x.

The elementals induce a cover of T by bipartite subdags. Variables can be read but not written by different elementals, that is, edges emanating from a vertex can belong to distinct elementals. All incoming edges of a vertex are part of the same elemental.

In the simplest case, all elementals are scalar functions mapping the variables that correspond to the $|P_j|$ predecessors of vertex $j \in V$ to v_j.

Edges $(i, j) \in E$ are labelled with local partial derivatives $d_{j,i} \equiv \frac{dv_j}{dv_i}$.

Tape Example

$$F : \mathbb{R}^2 \to \mathbb{R}^2$$

```
void F(std::vector<T>& x) {
  while (x[1]<=0)
    x[1]=exp(x[0]*x[1])+x[1];
  x[0]=x[0]/x[1];
  x[1]=pow(x[1],3);
}
```

$$v_1 = x_0$$

$$v_2 = x_1$$

$$v_3 = v_1 \cdot v_2$$

$$v_4 = e^{v_3}$$

$$v_5 = v_4 + v_2$$

$$v_6 = v_1 / v_5$$

$$v_7 = v_5^3$$

$$y_0 = v_6$$

$$y_1 = v_7$$

Tape Example

$$F : \mathbb{R}^2 \to \mathbb{R}^2$$

```
void F(std::vector<T>& x) {
  while (x[1]<=0)
    x[1]=exp(x[0]*x[1])+x[1];
  x[0]=x[0]/x[1];
  x[1]=pow(x[1],3);
}
```

$$v_1 = x_0$$

$$v_2 = x_1$$

$$v_3 = v_1 \cdot v_2$$

$$v_4 = e^{v_3}$$

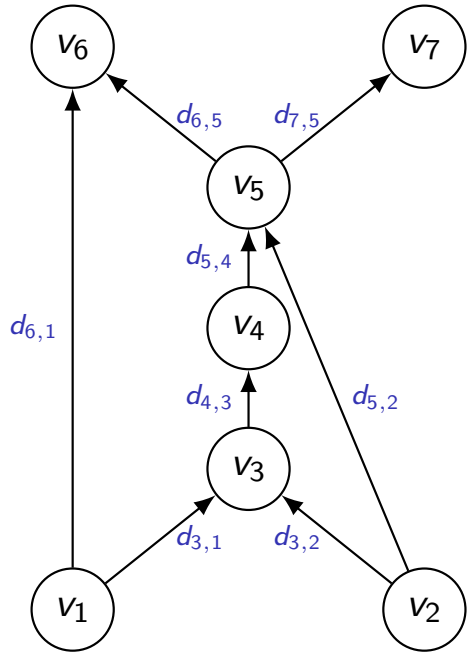
$$v_5 = v_4 + v_2$$

$$v_6 = v_1 / v_5$$

$$v_7 = v_5^3$$

$$y_0 = v_6$$

$$y_1 = v_7$$



Tape Example

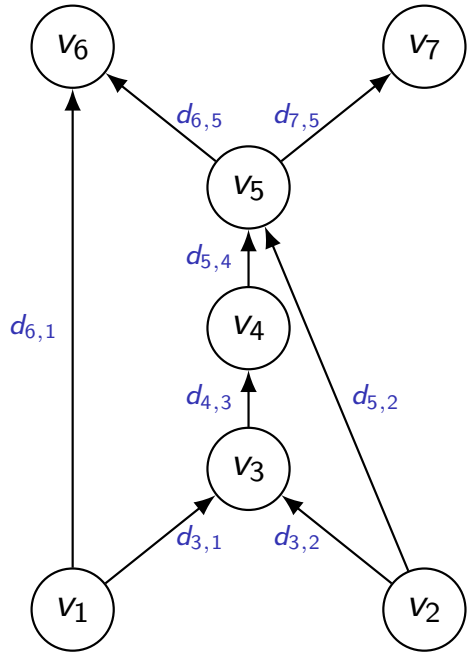
$F : \mathbb{R}^2 \to \mathbb{R}^2$

$$v_1 = x_0$$
$$v_2 = x_1$$

$$v_3 = v_1 \cdot v_2$$
$$v_4 = e^{v_3}$$
$$v_5 = v_4 + v_2$$
$$v_6 = v_1 / v_5$$
$$v_7 = v_5^3$$

$$y_0 = v_6$$
$$y_1 = v_7$$

```
void F(std::vector<T>& x) {
  while (x[1]<=0)
    x[1]=exp(x[0]*x[1])+x[1];
  x[0]=x[0]/x[1];
  x[1]=pow(x[1],3);
}
```



$F : \mathbb{R}^{200} \to \mathbb{R}^2$

$$v_1 = \left(x_0 \ldots x_{99}\right)^T$$
$$v_2 = \left(x_{100} \ldots x_{199}\right)^T$$

$$v_3 = v_1^T \cdot v_2$$
$$v_4 = e^{v_3}$$
$$v_5 = v_4 \cdot v_2$$
$$\begin{pmatrix} v_6 \\ v_7 \end{pmatrix} = \begin{pmatrix} v_1^T \cdot v_5 \\ v_5^T \cdot v_5 \end{pmatrix}$$

$$y_0 = v_6$$
$$y_1 = v_7$$

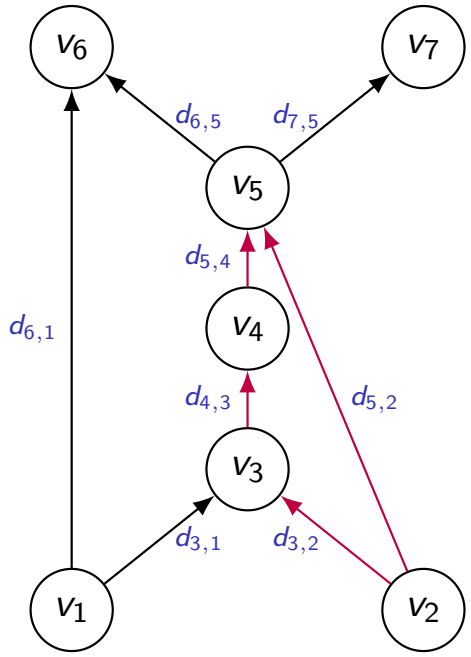
Chain Rule on Tape

The chain rule

$$F'(x) \equiv \frac{dy}{dx} = \sum_{\text{path}(x \to y) \in T(F,x)} \prod_{(i,j) \in \text{path}} \frac{\partial v_j}{\partial v_i}$$

applies to arbitrary pairs of vertices and corresponding subsets of V / subprograms of F, e.g.

$$\frac{dv_5}{dv_2} = d_{3,2} \cdot d_{4,3} \cdot d_{5,4} + d_{5,2}$$



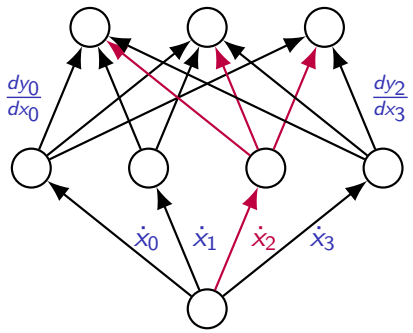
Tangents

Let $y = F(x)$. Tangents compute directional derivatives

$$\mathbb{R}^m \ni \dot{y} = \dot{F}(x, \dot{x}) \equiv F'(x) \cdot \dot{x} \ ,$$

without prior accumulation of the Jacobian $F'(x)$.

$$\begin{pmatrix} \dot{y}_0 \\ \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} \frac{dy_0}{dx_0} & \frac{dy_0}{dx_1} & \frac{dy_0}{dx_2} & \frac{dy_0}{dx_3} \\ \frac{dy_1}{dx_0} & \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \frac{dy_1}{dx_3} \\ \frac{dy_2}{dx_0} & \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \frac{dy_2}{dx_3} \end{pmatrix} \cdot \begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix}$$



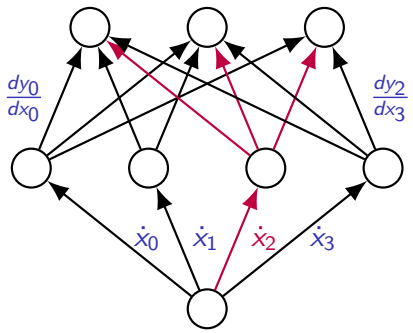
Tangents

Let $y = F(x)$. Tangents compute directional derivatives

$$\mathbb{R}^m \ni \dot{y} = \dot{F}(x, \dot{x}) \equiv F'(x) \cdot \dot{x} \ ,$$

without prior accumulation of the Jacobian $F'(x)$.

$$\begin{pmatrix} \dot{y}_0 \\ \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} \frac{dy_0}{dx_0} & \frac{dy_0}{dx_1} & \frac{dy_0}{dx_2} & \frac{dy_0}{dx_3} \\ \frac{dy_1}{dx_0} & \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \frac{dy_1}{dx_3} \\ \frac{dy_2}{dx_0} & \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \frac{dy_2}{dx_3} \end{pmatrix} \cdot \begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix}$$



Implementations of tangent AD typically augment F with $\dot{F}$.

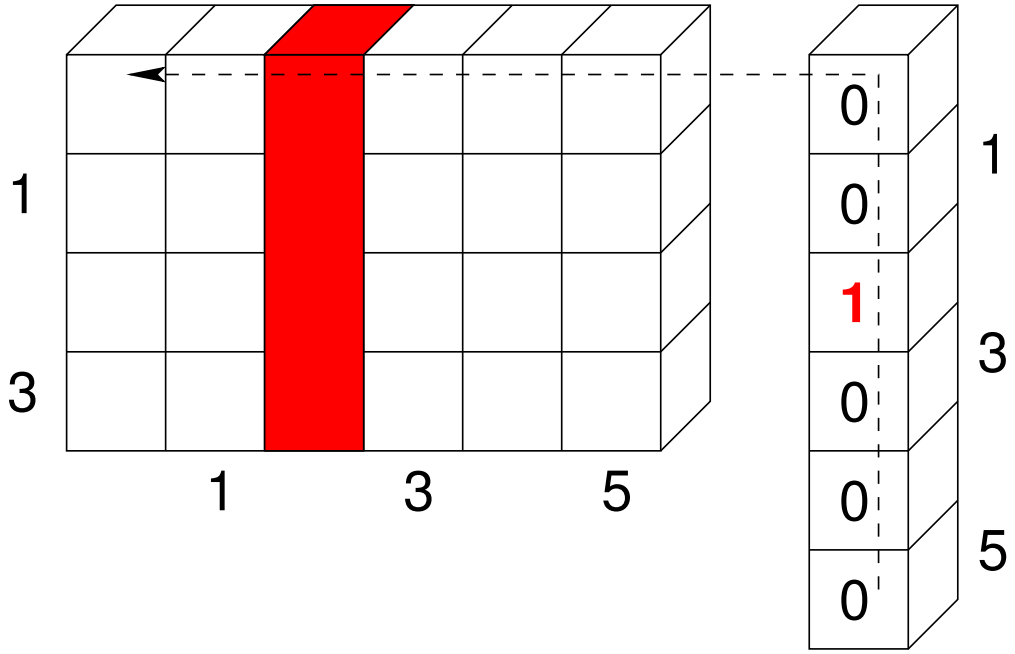
The Jacobian F' is accumulated column-wise by letting $\dot{x}$ range over the Cartesian basis vectors in $\mathbb{R}^n$.

Potential sparsity of the Jacobian can and should be exploited, e.g. 5×5 band structure.

Tangent AD: Cost of Jacobian Accumulation

The Jacobian F' can be accumulated at x by n evaluations of $\dot{F}$ with $\dot{x}$ ranging over the Cartesian basis vectors in $\mathbb{R}^n$.[1] Hence,

$$\mathrm{Cost}(F') = O(n) \cdot \mathrm{Cost}(\dot{F}) = O(n) \cdot \mathrm{Cost}(F) \ .$$



[1] This number can be decreased by detecting and exploiting potential sparsity F'.

Tangent AD

Tangent AD propagates tangents of all elementals evaluated by the primal SAC yielding the augmented primal SAC

$$i = 1, \ldots, n: \quad \begin{pmatrix} v_i \\ \dot{v}_i \end{pmatrix} = \begin{pmatrix} x_{i-1} \\ \dot{x}_{i-1} \end{pmatrix}$$

$$j = n+1, \ldots, n+q: \quad \begin{pmatrix} v_j \\ d_{j,i} \\ \dot{v}_j \end{pmatrix} = \begin{pmatrix} \varphi_j(v_k)_{k \prec j} \\ \frac{d\varphi_j(v_k)_{k \prec j}}{dv_i} \\ \sum_{i \prec j} d_{j,i} \cdot \dot{v}_i \end{pmatrix}$$

$$k = 1, \ldots, m: \quad \begin{pmatrix} y_{k-1} \\ \dot{y}_{k-1} \end{pmatrix} = \begin{pmatrix} v_{n+p+k} \\ \dot{v}_{n+p+k} \end{pmatrix}$$

Tangent AD

Tangent AD propagates tangents of all elementals evaluated by the primal SAC yielding the augmented primal SAC

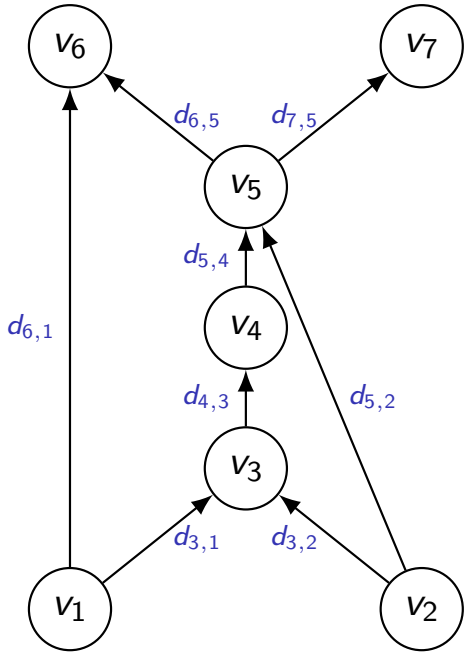
$$
\begin{aligned}
i = 1, \ldots, n: \quad & \begin{pmatrix} v_i \\ \dot{v}_i \end{pmatrix} = \begin{pmatrix} x_{i-1} \\ \dot{x}_{i-1} \end{pmatrix} \\
j = n+1, \ldots, n+q: \quad & \begin{pmatrix} v_j \\ d_{j,i} \\ \dot{v}_j \end{pmatrix} = \begin{pmatrix} \varphi_j(v_k)_{k \prec j} \\ \frac{d\varphi_j(v_k)_{k \prec j}}{dv_i} \\ \sum_{i \prec j} d_{j,i} \cdot \dot{v}_i \end{pmatrix} \\
k = 1, \ldots, m: \quad & \begin{pmatrix} y_{k-1} \\ \dot{y}_{k-1} \end{pmatrix} = \begin{pmatrix} v_{n+p+k} \\ \dot{v}_{n+p+k} \end{pmatrix}
\end{aligned}
$$

The above lends itself to implementation by operator and function overloading (e.g, in C++). The entire arithmetic can be overloaded for a custom data type $(v, \dot{v})$ comprising both value and tangent. Explicit storage of the tape is not necessary.

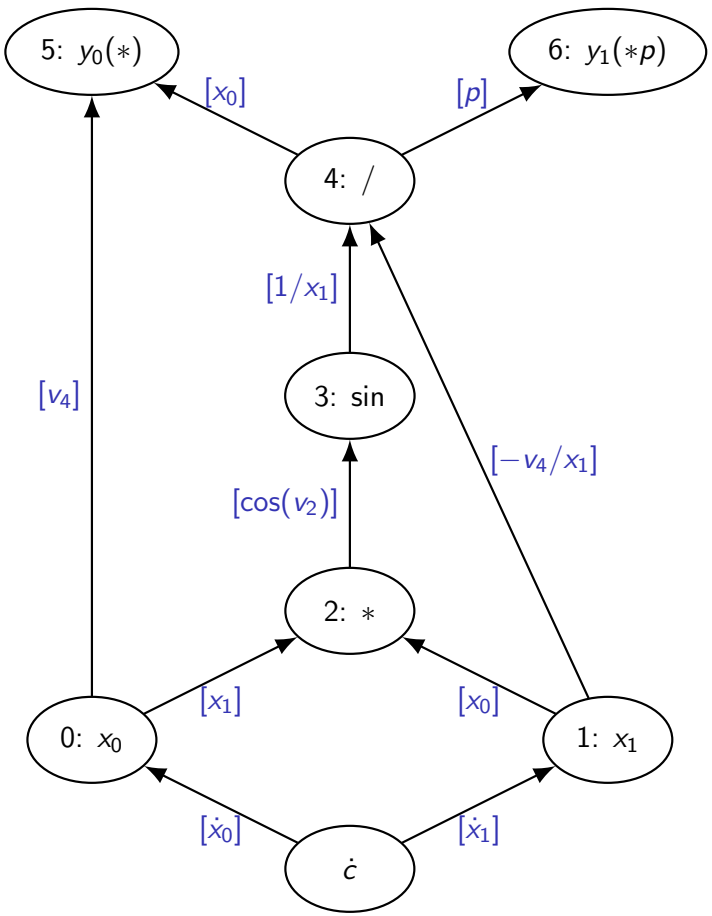
Tangent AD Example

Elementals are augmented with their tangents, e.g.

$v_1 = x_0;$		$\dot{v}_1 = \dot{x}_0$
$v_2 = x_1;$		$\dot{v}_2 = \dot{x}_1$
$v_3 = v_1 \cdot v_2;$	$d_{3,1} = v_2;$	$\dot{v}_3 = d_{3,1} \cdot \dot{v}_1 + d_{3,2} \cdot \dot{v}_2$
	$d_{3,2} = v_1$	
$v_4 = e^{v_3};$	$d_{4,3} = v_4;$	$\dot{v}_4 = d_{4,3} \cdot \dot{v}_3$
$v_5 = v_4 + v_2;$	$d_{5,4} = 1;$	$\dot{v}_5 = d_{5,4} \cdot \dot{v}_4 + d_{5,2} \cdot \dot{v}_2$
	$d_{5,2} = 1$	
$v_6 = v_1 / v_5;$	$d_{6,1} = 1/v_5;$	$\dot{v}_6 = d_{6,1} \cdot \dot{v}_1 + d_{6,5} \cdot \dot{v}_5$
	$d_{6,5} = -v_1/v_5^2$	
$v_7 = v_5^3;$	$d_{7,5} = 2 \cdot v_5^2;$	$\dot{v}_7 = d_{7,5} \cdot \dot{v}_5$
$y_0 = v_6;$		$\dot{y}_0 = \dot{v}_6$
$y_1 = v_7;$		$\dot{y}_1 = \dot{v}_7$



Tangent AD Movie



Tangent DAG

We consider

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

implemented as

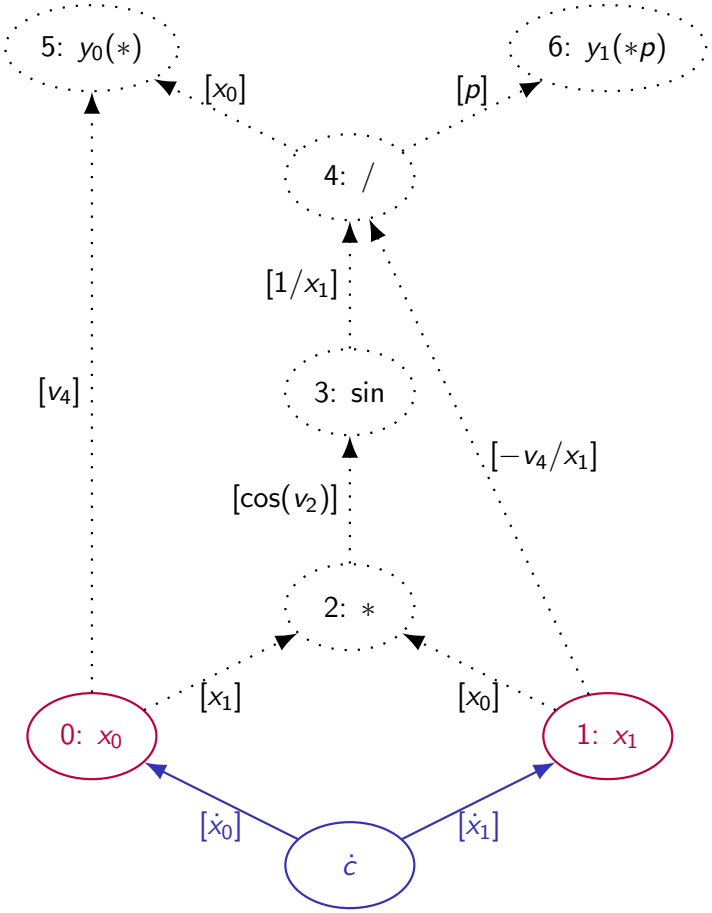
$$t = \sin(x_0 * x_1)/x_1$$
$$y_0 = x_0 * t;\ y_1 = t * p$$

yielding SAC

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

for some passive value p, i.e, no derivatives of or with respect to required; x, y, and t are active.

Tangent AD Movie: Seed



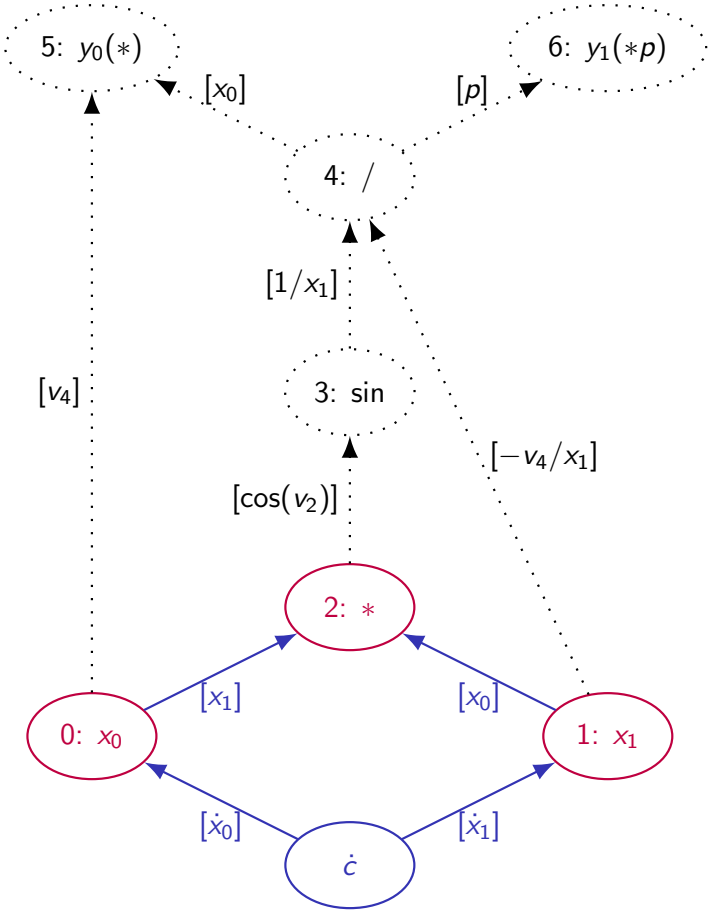
$x_0 = ?$

$x_1 = ?$

$\dot{x}_0 = ?$

$\dot{x}_1 = ?$

Tangent AD Movie: Propagate

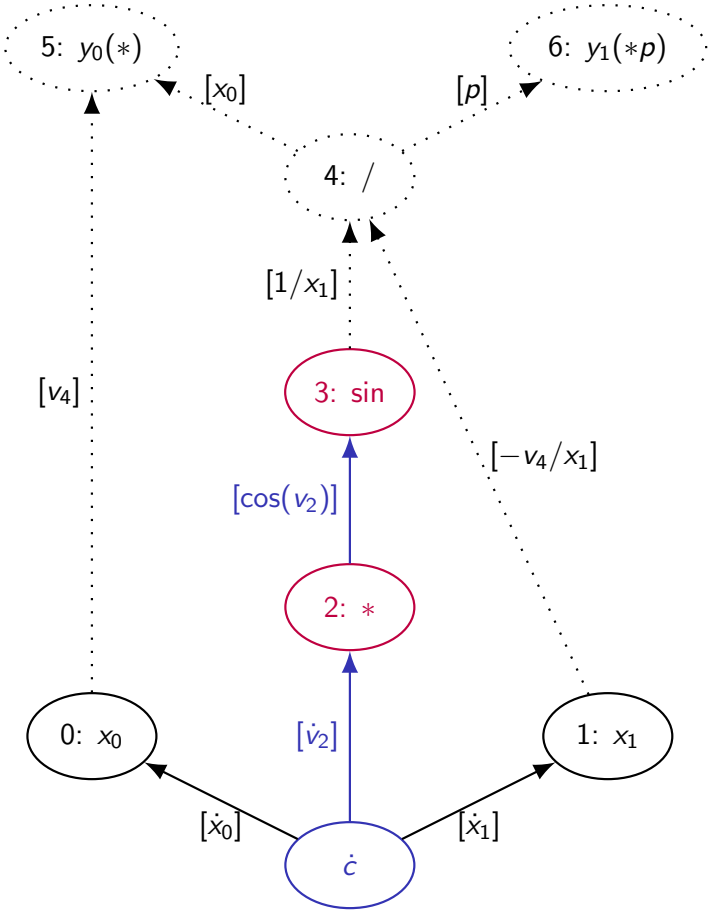


// overloaded $*$

$v_2 = x_0 * x_1$

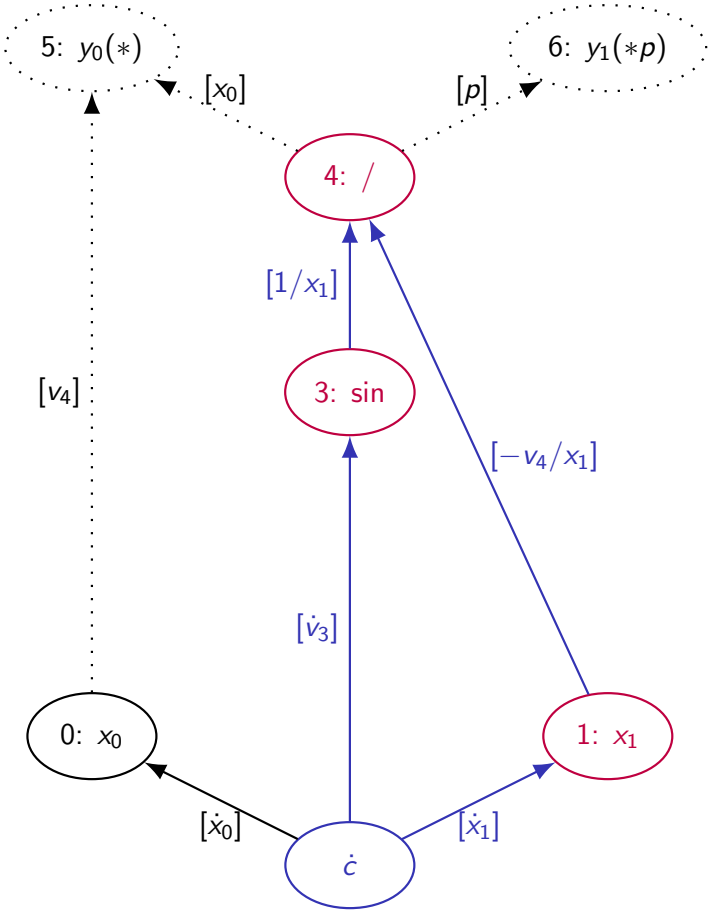
$\dot{v}_2 = x_1 * \dot{x}_0 + x_0 * \dot{x}_1$

Tangent AD Movie: Propagate



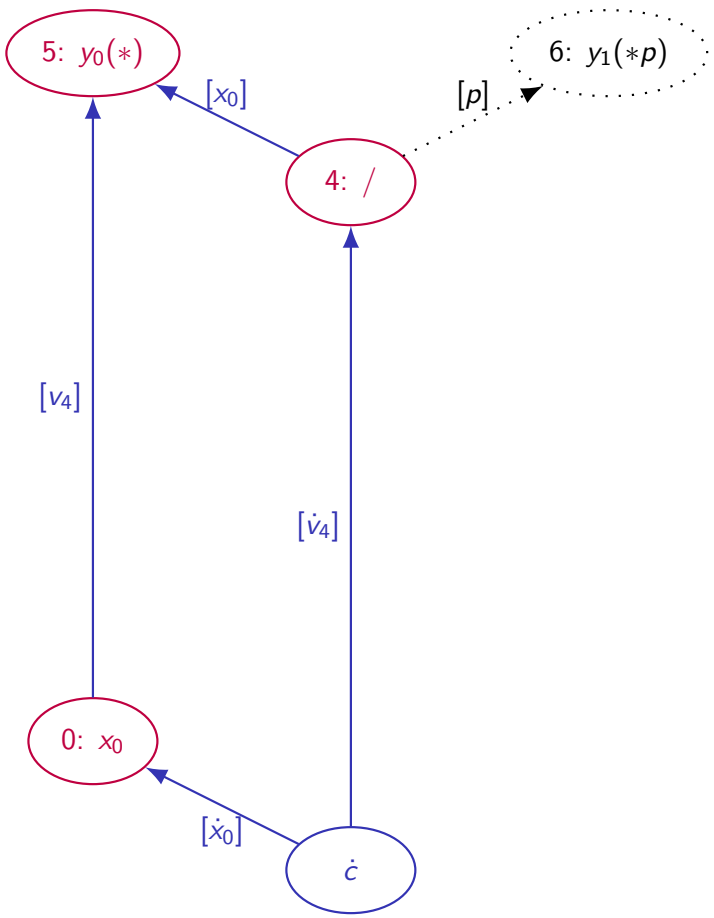
$$v_2 = x_0 * x_1$$
$$\dot{v}_2 = x_1 * \dot{x}_0 + x_0 * \dot{x}_1$$
// overloaded sin
$$v_3 = \sin(v_2)$$
$$\dot{v}_3 = \cos(v_2) * \dot{v}_2$$

Tangent AD Movie: Propagate



$$v_2 = x_0 * x_1$$
$$\dot{v}_2 = x_1 * \dot{x}_0 + x_0 * \dot{x}_1$$
$$v_3 = \sin(v_2)$$
$$\dot{v}_3 = \cos(v_2) * \dot{v}_2$$
// overloaded /
$$v_4 = v_3 / x_1$$
$$\dot{v}_4 = (\dot{v}_3 - v_4 * \dot{x}_1)/x_1$$

Tangent AD Movie: Propagate

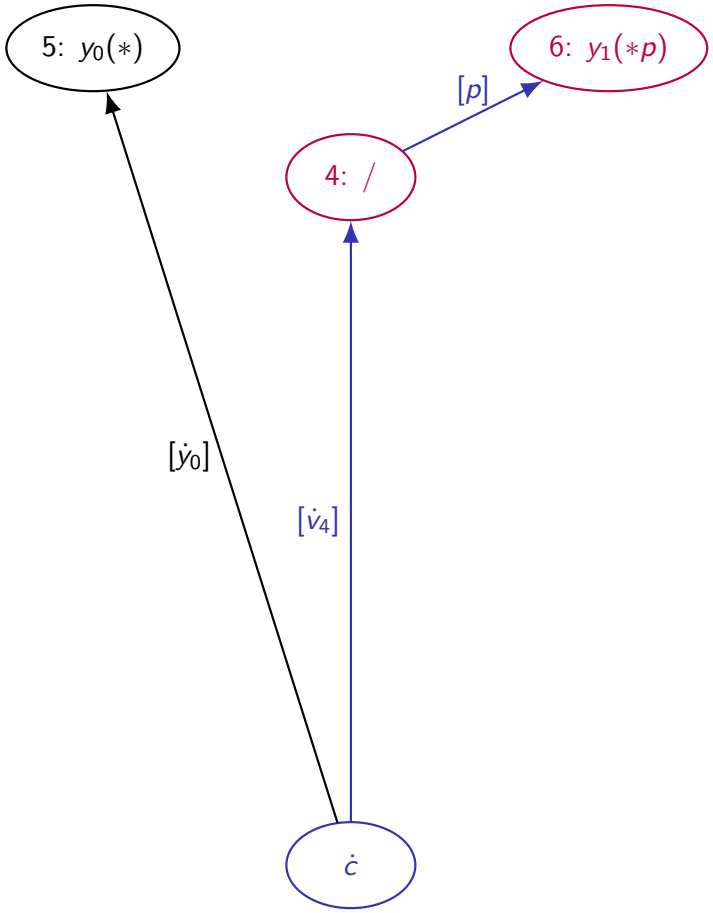


$$v_2 = x_0 * x_1$$
$$\dot{v}_2 = x_1 * \dot{x}_0 + x_0 * \dot{x}_1$$
$$v_3 = \sin(v_2)$$
$$\dot{v}_3 = \cos(v_2) * \dot{v}_2$$
$$v_4 = v_3 / x_1$$
$$\dot{v}_4 = (\dot{v}_3 - v_4 * \dot{x}_1)/x_1$$

// overloaded $*$

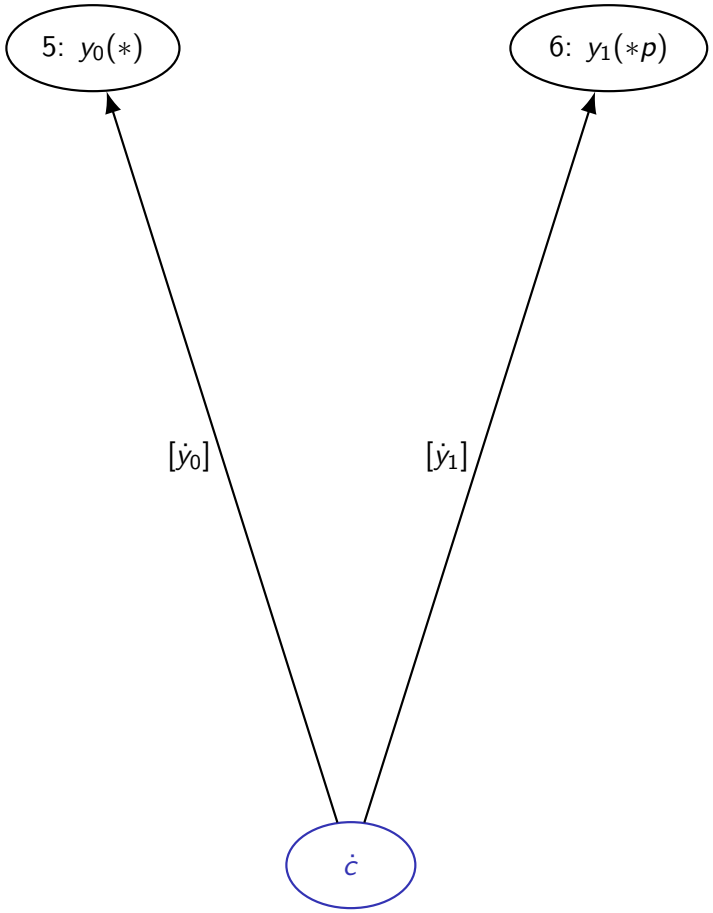
$$y_0 = x_0 * v_4$$
$$y_0^{(1)} = y_0^{(1)} + v_4 * \dot{x}_0 + x_0 * \dot{v}_4$$

Tangent AD Movie: Propagate



$$v_2 = x_0 * x_1$$
$$\dot{v}_2 = x_1 * \dot{x}_0 + x_0 * \dot{x}_1$$
$$v_3 = \sin(v_2)$$
$$\dot{v}_3 = \cos(v_2) * \dot{v}_2$$
$$v_4 = v_3 / x_1$$
$$\dot{v}_4 = (\dot{v}_3 - v_4 * \dot{x}_1) / x_1$$
$$y_0 = x_0 * v_4$$
$$y_0^{(1)} = y_0^{(1)} + v_4 * \dot{x}_0 + x_0 * \dot{v}_4$$
// overloaded $*$
$$y_1 = v_4 * p$$
$$y_1^{(1)} = y_1^{(1)} + p * \dot{v}_4$$

Tangent AD Movie: Harvest



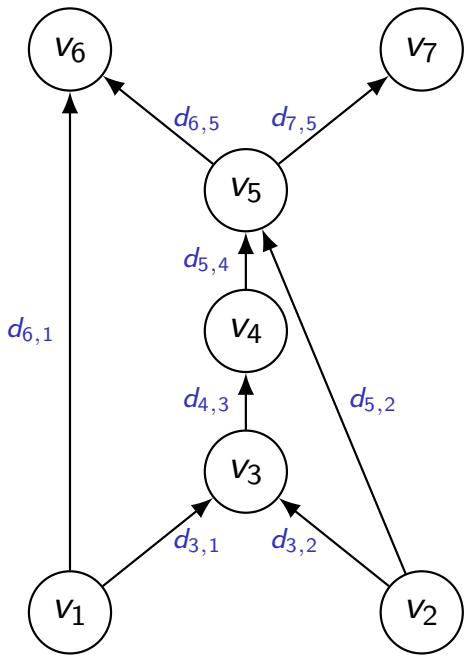
$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
\dot{v}_2 &= x_1 * \dot{x}_0 + x_0 * \dot{x}_1 \\
v_3 &= \sin(v_2) \\
\dot{v}_3 &= \cos(v_2) * \dot{v}_2 \\
v_4 &= v_3 / x_1 \\
\dot{v}_4 &= (\dot{v}_3 - v_4 * \dot{x}_1) / x_1 \\
y_0 &= x_0 * v_4 \\
y_0^{(1)} &= y_0^{(1)} + v_4 * \dot{x}_0 + x_0 * \dot{v}_4 \\
y_1 &= v_4 * p \\
y_1^{(1)} &= y_1^{(1)} + p * \dot{v}_4
\end{aligned}
$$

Cost of Tangent AD

(persistent) $\quad \textsc{Mem}(F) = \textsc{Mem}(\dot{F}) = 0$

$$\textsc{Ert}(F) \sim \textsc{Ops}(F) = |V| - n = 7 - 2 = 5$$

$$\begin{aligned}\textsc{Ert}(\dot{F}) \sim \textsc{Ops}(\dot{F}) &= |V| - n + |E| + |E| \\ &= 7 - 2 + 8 + 8 = 21\,.\end{aligned}$$



In reality we typically see

$$\textsc{Ert}(\dot{F}) \leq 2 \cdot \textsc{Ert}(F)\,.$$

Approximate Gradient by Finite Differences

Individual columns of the Jacobian can be approximated by (forward, backward, central) finite difference quotients as follows:

$$\nabla F(x) \approx_{\mathcal{O}(h)} \left(\frac{F(x + h \cdot e_i) - F(x)}{h} \right)_{i=0}^{n-1} \approx_{\mathcal{O}(h)} \left(\frac{F(x) - F(x - h \cdot e_i)}{h} \right)_{i=0}^{n-1}$$

$$\approx_{\mathcal{O}(h^2)} \left(\frac{F(x + h \cdot e_i) - F(x - h \cdot e_i)}{2 \cdot h} \right)_{i=0}^{n-1}$$

where $h = h(x_i) \in \mathbb{R}$ is a "suitable perturbation" typically picked as a compromise between accuracy and numerical stability, e.g,

$$h = \begin{cases} \sqrt{\epsilon} & x_i = 0 \\ \sqrt{\epsilon} \cdot |x_i| & x_i \neq 0 \end{cases}$$

with machine epsilon ϵ dependent on the floating-point precision.

Approximate Gradient by Central Finite Differences

`SDE/fd/main.cpp`

- inspect
- build (`Makefile`)
- run
- time (`/usr/bin/time -v`)

Experiments

np	ns	ERT (s)	RSS (MB)
10^4	10	0.1	4
10^4	100	10.1	4
10^4	200	39.6	4
10^4	300	99.6	4
10^5	10	1.0	4
10^5	100	99.3	4
10^6	10	10.6	4
10^6	100	≈ 1000	4

Hands-on: Run and time on your computer.

Gradient by Scalar Tangent AD with dco/c++

```
1  #include "f.h" // primal
```

Gradient by Scalar Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
```

Gradient by Scalar Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
```

Gradient by Scalar Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   using dco_type=dco::gt1s<dco_value_type>::type;
```

Gradient by Scalar Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   using dco_type=dco::gt1s<dco_value_type>::type;
12
13   // active program variables
14   dco_type x=1.; vector<dco_type> p(2*ns,1);
```

Gradient by Scalar Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   using dco_type=dco::gt1s<dco_value_type>::type;
12
13   // active program variables
14   dco_type x=1.; vector<dco_type> p(2*ns,1);
15
16   dco::derivative(x)=1; // seed
17   dco_type y=f(np,ns,x,p); // propagate
18   dco::derivative(x)=0; // prepare next seed
19   cout.precision(numeric_limits<dco_value_type>::digits10);
20   cout << "dy/dx=" << dco::derivative(y) << '\n'; // harvest
```

```
21    for (int i=0;i<2*ns;++i) {
22      dco::derivative(p[i])=1; // seed
23      y=f(np,ns,x,p); // propagate
24      cout << "dy/dp[" << i << "]="
25           << dco::derivative(y) << '\n'; // harvest
26      dco::derivative(p[i])=0; // prepare next seed
27    }
```

```
21     for (int i=0;i<2*ns;++i) {
22       dco::derivative(p[i])=1; // seed
23       y=f(np,ns,x,p); // propagate
24       cout << "dy/dp[" << i << "]="
25            << dco::derivative(y) << '\n'; // harvest
26       dco::derivative(p[i])=0; // prepare next seed
27     }
28
29     cout << "y=" << dco::value(y) << endl; // primal value
30
31     return 0;
32   } // end of driver
```

Gradient by Scalar Tangent AD with dco/c++

`SDE/gt1s/main.cpp`

- build (`Makefile`)
- run
- time (`/usr/bin/time -v`)
- compare with finite differences

Experiments

np	ns	Ert (s)	Rss (MB)
10^4	10	< 0.1	4
10^4	100	5.4	4
10^4	200	21.0	4
10^4	300	47.3	4
10^5	10	0.6	4
10^5	100	52.9	4
10^6	10	5.6	4
10^6	100	≈ 560	4

Hands-on: Run and time on your computer.

Vector Tangent AD

Vectors (also: ensembles) of k first-order tangents of

$$F : \mathbb{R}^n \to \mathbb{R}^m : \quad y = F(x) \,,$$

can be evaluated in vector tangent AD mode as

$$\dot{Y} = \dot{F}(x, \dot{X}) \equiv F'(x) \cdot \dot{X}$$

for $\dot{X} \in \mathbb{R}^{n \times k}$ and $\dot{Y} \in \mathbb{R}^{m \times k}$.

Performance of vector tangent mode typically exceeds that of scalar tangent mode even without the highly desirable explicit parallelization / vectorization. The memory requirement is increased.

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
```

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
```

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
```

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   const int VEC_SIZE=21; assert(VEC_SIZE>=2*ns+1);
12   using dco_type=dco::gt1v<dco_value_type,VEC_SIZE>::type;
```

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   const int VEC_SIZE=21; assert(VEC_SIZE>=2*ns+1);
12   using dco_type=dco::gt1v<dco_value_type,VEC_SIZE>::type;
13
14   // active program variables
15   dco_type x=1,y; vector<dco_type> p(2*ns,1);
```

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   const int VEC_SIZE=21; assert(VEC_SIZE>=2*ns+1);
12   using dco_type=dco::gt1v<dco_value_type,VEC_SIZE>::type;
13
14   // active program variables
15   dco_type x=1,y; vector<dco_type> p(2*ns,1);
16
17   // seed
18   dco::derivative(x)[0]=1;
19   for (int i=0;i<2*ns;++i) dco::derivative(p[i])[i+1]=1;
```

Gradient by Vector Tangent AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base and active types
10   using dco_value_type=double;
11   const int VEC_SIZE=21; assert(VEC_SIZE>=2*ns+1);
12   using dco_type=dco::gt1v<dco_value_type,VEC_SIZE>::type;
13
14   // active program variables
15   dco_type x=1,y; vector<dco_type> p(2*ns,1);
16
17   // seed
18   dco::derivative(x)[0]=1;
19   for (int i=0;i<2*ns;++i) dco::derivative(p[i])[i+1]=1;
20
21   y=f(np,ns,x,p); // propagate
```

Gradient by Vector Tangent AD with dco/c++

```
22    // harvest
23    cout.precision(numeric_limits<dco_value_type>::digits10);
24    cout << "dy/dx=" << dco::derivative(y)[0] << '\n';
25    for (int i=0;i<2*ns;++i)
26      cout << "dy/dp[" << i << "]="
27           << dco::derivative(y)[i+1] << '\n';
```

Gradient by Vector Tangent AD with dco/c++

```
22     // harvest
23     cout.precision(numeric_limits<dco_value_type>::digits10);
24     cout << "dy/dx=" << dco::derivative(y)[0] << '\n';
25     for (int i=0;i<2*ns;++i)
26       cout << "dy/dp[" << i << "]="
27            << dco::derivative(y)[i+1] << '\n';
28
29     cout << "y=" << dco::value(y) << endl; // primal value
30
31     return 0;
32   } // end of driver
```

Gradient by Vector Tangent AD with dco/c++

`SDE/gt1v/main.cpp`

- build (`Makefile`)
- run
- time (`/usr/bin/time -v`)
- compare with scalar tangent AD

Experiments

np	ns	ERT (s)	RSS (MB)
10^4	10	< 0.1	4
10^4	100	0.5	4
10^4	200	2.3	5
10^4	300	5.6	7
10^5	10	0.1	4
10^5	100	5.8	4
10^6	10	1.2	4
10^6	100	54.0	4

Hands-on: Run and time on your computer.

Hands-On

We consider a third-party implementation of the Heston Stochastic Volatility Model with Euler Discretization[2].

- inspect primal code
- build (`Makefile`)
- run

Use dco/c++ to compute the gradient

- in scalar tangent mode
- in vector tangent mode

Run experiments and compare performances.

[2] www.quantstart.com/articles/Heston-Stochastic-Volatility-Model-with-Euler-Discretisation-in-C/

Outline

Adjoint AD

Adjoints compute

$$\mathbb{R}^{1\times n} \ni \bar{x} = \bar{F}(x, \bar{y}) \equiv \bar{y} \cdot F'(x)$$

without prior accumulation of the Jacobian $F'(x)$.

Adjoint AD

Adjoints compute

$$\mathbb{R}^{1\times n} \ni \bar{x} = \bar{F}(x, \bar{y}) \equiv \bar{y} \cdot F'(x)$$

without prior accumulation of the Jacobian $F'(x)$.

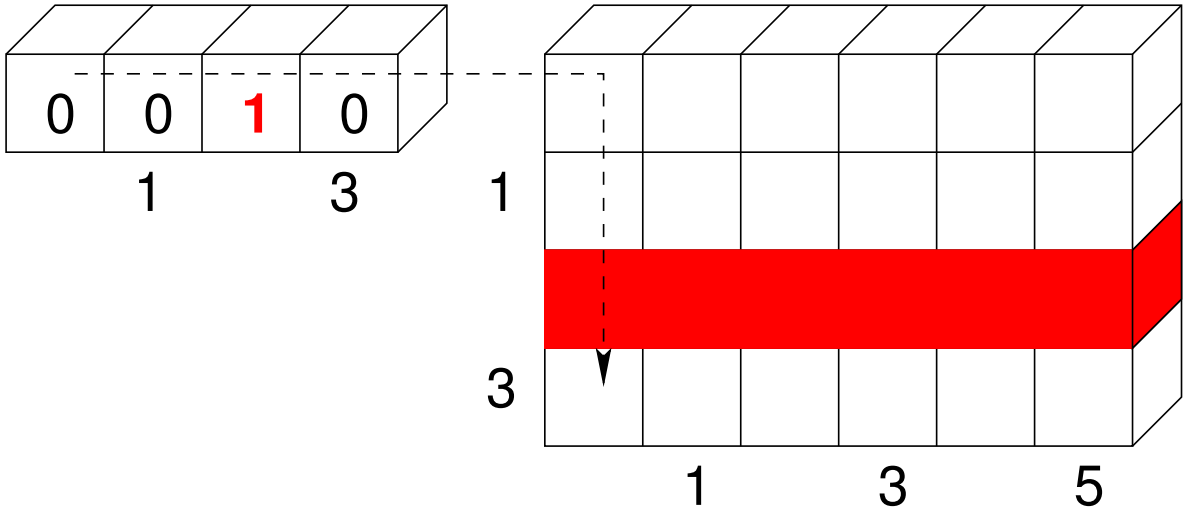
Example:

$$\begin{pmatrix} \bar{x}_0 & \bar{x}_1 & \bar{x}_2 & \bar{x}_3 \end{pmatrix} = \begin{pmatrix} \bar{y}_0 & \bar{y}_1 & \bar{y}_2 \end{pmatrix} \cdot \begin{pmatrix} \frac{dy_0}{dx_0} & \frac{dy_0}{dx_1} & \frac{dy_0}{dx_2} & \frac{dy_0}{dx_3} \\ \frac{dy_1}{dx_0} & \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \frac{dy_1}{dx_3} \\ \frac{dy_2}{dx_0} & \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \frac{dy_2}{dx_3} \end{pmatrix}$$

Adjoint AD: Cost of Jacobian Accumulation

The Jacobian $F' = F'(x)$ of $y = F(x)$ can be accumulated at x by m evaluations of $\bar{F}$ with $\bar{y}$ ranging over the Cartesian basis vectors in $\mathbb{R}^m$.[3] Hence,

$$\mathrm{Cost}(F') = O(m) \cdot \mathrm{Cost}(\bar{F}) = O(m) \cdot \mathrm{Cost}(F) \; .$$



[3] This number can be decreased by detecting and exploiting potential sparsity of F'.

Adjoints by Gradient Taping

Recording

$$(y, T) = \overrightarrow{F}(x)$$

Backpropagation

$$\bar{X} = \overleftarrow{F}(\bar{Y}, T) \equiv \bar{Y} \cdot T$$

Adjoints by Gradient Taping

Recording

$$(y, T) = \overrightarrow{F}(x)$$

1: $T := (\varnothing, \varnothing)$

2: for $i = 1, \ldots, n$:

3: $\quad v_i = x_i$

4: $\quad T \mathrel{+}= (\{(i, \&\bar{v}_i)\}, \varnothing)$

5: for $j = n+1, \ldots, n+q$:

6: $\quad v_j := \varphi_j(v_k)_{k \prec j}$

7: $\quad T \mathrel{+}= (\{(j, \&\bar{v}_j)\}, \varnothing)$

8: $\quad \forall i \prec j$:

9: $\quad\quad d_{j,i} := \dfrac{d\varphi_j}{dv_i}(v_k)_{k \prec j}$

10: $\quad\quad T \mathrel{+}= (\varnothing, \{(i, j, d_{j,i})\})$

11: $\{y_1, \ldots, y_m\} \subseteq \{v_1, \ldots v_{n+q}\}$

Backpropagation

$$\bar{X} = \overleftarrow{F}(\bar{Y}, T) \equiv \bar{Y} \cdot T$$

Adjoints by Gradient Taping

Recording

$$(y, T) = \overrightarrow{F}(x)$$

$T := (\varnothing, \varnothing)$

for $i = 1, \ldots, n$:

  $v_i = x_i$

  $T \mathrel{+}= (\{(i, \&\bar{v}_i)\}, \varnothing)$

for $j = n+1, \ldots, n+q$:

  $v_j := \varphi_j(v_k)_{k \prec j}$

  $T \mathrel{+}= (\{(j, \&\bar{v}_j)\}, \varnothing)$

  $\forall i \prec j$:

    $d_{j,i} := \dfrac{d\varphi_j}{dv_i}(v_k)_{k \prec j}$

    $T \mathrel{+}= (\varnothing, \{(i, j, d_{j,i})\})$

$\{y_1, \ldots, y_m\} \subseteq \{v_1, \ldots v_{n+q}\}$

Backpropagation

$$\bar{X} = \overleftarrow{F}(\bar{Y}, T) \equiv \bar{Y} \cdot T$$

for $j = 1, \ldots, n+q$: $\bar{v}_j := 0$

$\{\bar{y}_1, \ldots, \bar{y}_m\} \subseteq \{\bar{v}_1, \ldots \bar{v}_{n+q}\}$

for $j = n+q, \ldots, n+1$

  $\forall i \prec j$:

    $T \mathrel{-}= (\varnothing, \{(i, j, d_{j,i})\})$

    $\bar{v}_i \mathrel{+}= \bar{v}_j \cdot d_{j,i}$

  $T \mathrel{-}= (\{(j, \&\bar{v}_j)\}, \varnothing)$

  $\bar{v}_j := 0$

for $i = 1, \ldots, n$: $\bar{x}_i = \bar{v}_i$

Alternatively: Adjoints by Value Taping

Recording

$$(y, T) = \overrightarrow{F}(x)$$

Backpropagation

$$\bar{X} = \overleftarrow{F}(\bar{Y}, T) \equiv \bar{Y} \cdot T$$

Alternatively: Adjoints by Value Taping

Recording

$$(y, T) = \overrightarrow{F}(x)$$

Backpropagation

$$\bar{X} = \overleftarrow{F}(\bar{Y}, T) \equiv \bar{Y} \cdot T$$

$T := (\varnothing, \varnothing)$

for $i = 1, \ldots, n$:

$\quad v_i = x_i$

$\quad T \mathrel{+}= (\{(i, v_i, \&\bar{v}_i)\}, \varnothing)$

for $j = n+1, \ldots, n+q$:

$\quad v_j := \varphi_j(v_k)_{k \prec j}$

$\quad T \mathrel{+}= (\{(j, v_j, \&\bar{v}_j)\}, \varnothing)$

$\quad \forall i \prec j : \quad T \mathrel{+}= (\varnothing, \{(i,j)\})$

$\{y_1, \ldots, y_m\} \subseteq \{v_1, \ldots v_{n+q}\}$

Alternatively: Adjoints by Value Taping

Recording

$$(y, T) = \overrightarrow{F}(x)$$

$T := (\varnothing, \varnothing)$

for $i = 1, \ldots, n$:

$\quad v_i = x_i$

$\quad T \mathrel{+}= (\{(i, v_i, \&\bar{v}_i)\}, \varnothing)$

for $j = n+1, \ldots, n+q$:

$\quad v_j := \varphi_j(v_k)_{k \prec j}$

$\quad T \mathrel{+}= (\{(j, v_j, \&\bar{v}_j)\}, \varnothing)$

$\quad \forall i \prec j : \quad T \mathrel{+}= (\varnothing, \{(i,j)\})$

$\{y_1, \ldots, y_m\} \subseteq \{v_1, \ldots v_{n+q}\}$

Backpropagation

$$\bar{X} = \overleftarrow{F}(\bar{Y}, T) \equiv \bar{Y} \cdot T$$

for $j = 1, \ldots, n+q : \quad \bar{v}_j := 0$

$\{\bar{y}_1, \ldots, \bar{y}_m\} \subseteq \{\bar{v}_1, \ldots \bar{v}_{n+q}\}$

for $j = n+q, \ldots, n+1$

$\quad \forall i \prec j :$

$\quad\quad T \mathrel{-}= (\varnothing, \{(i,j)\})$

$\quad\quad d_{j,i} := \dfrac{d\varphi_j}{dv_i}(v_k)_{k \prec j}$

$\quad\quad \bar{v}_i \mathrel{+}= \bar{v}_j \cdot d_{j,i}$

$\quad T \mathrel{-}= (\{(j, v_j, \&\bar{v}_j)\}, \varnothing)$

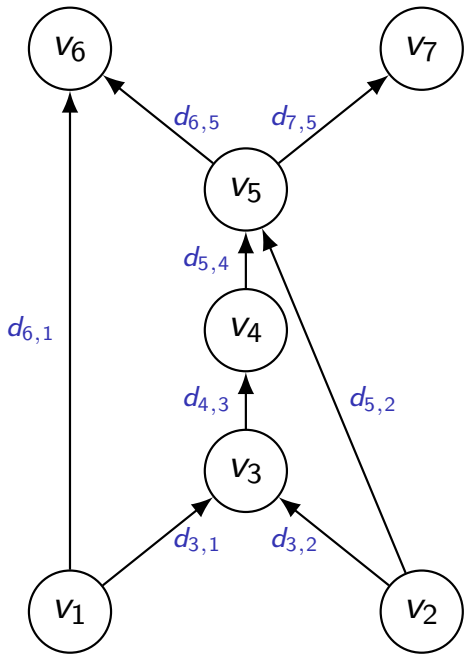
$\quad \bar{v}_j := 0$

for $i = 1, \ldots, n : \quad \bar{x}_i = \bar{v}_i$

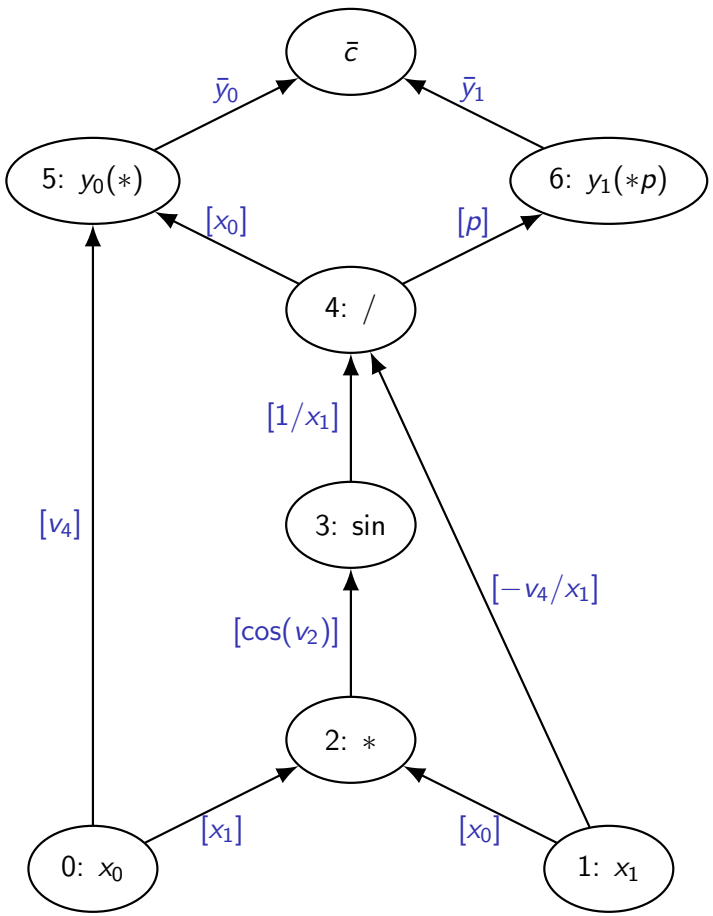
Adjoints by Gradient Taping: Example

Adjoint elementals are evaluated in reverse order, e.g.

$$
\begin{array}{lll}
\downarrow v_1 = x_0; & & \uparrow \bar{x}_0 = \bar{v}_1 \\
\downarrow v_2 = x_1; & & \uparrow \bar{x}_1 = \bar{v}_2 \\
\downarrow v_3 = v_1 \cdot v_2; & d_{3,1} = v_2; & \uparrow \bar{v}_1 = \bar{v}_1 + \bar{v}_3 \cdot d_{3,1} \\
 & d_{3,2} = v_1 & \uparrow \bar{v}_2 = \bar{v}_2 + \bar{v}_3 \cdot d_{3,2} \\
\downarrow v_4 = e^{v_3}; & d_{4,3} = v_4; & \uparrow \bar{v}_3 = \bar{v}_4 \cdot d_{4,3} \\
\downarrow v_5 = v_4 + v_2; & d_{5,4} = 1; & \uparrow \bar{v}_4 = \bar{v}_5 \cdot d_{5,4} \\
 & d_{5,2} = 1 & \uparrow \bar{v}_2 = \bar{v}_5 \cdot d_{5,2} \\
\downarrow v_6 = v_1 / v_5; & d_{6,1} = 1/v_5; & \uparrow \bar{v}_1 = \bar{v}_6 \cdot d_{6,1} \\
 & d_{6,5} = -v_1/v_5^2 & \uparrow \bar{v}_5 = \bar{v}_5 + \bar{v}_6 \cdot d_{6,5} \\
\downarrow v_7 = v_5^3; & d_{7,5} = 2 \cdot v_5^2; & \uparrow \bar{v}_5 = \bar{v}_7 \cdot d_{7,5} \\
\downarrow y_0 = v_6; & & \uparrow \bar{v}_6 = \bar{y}_0 \\
\downarrow y_1 = v_7; & & \uparrow \bar{v}_7 = \bar{y}_1
\end{array}
$$



Adjoint AD Movie



Adjoint DAG

We consider

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

implemented as

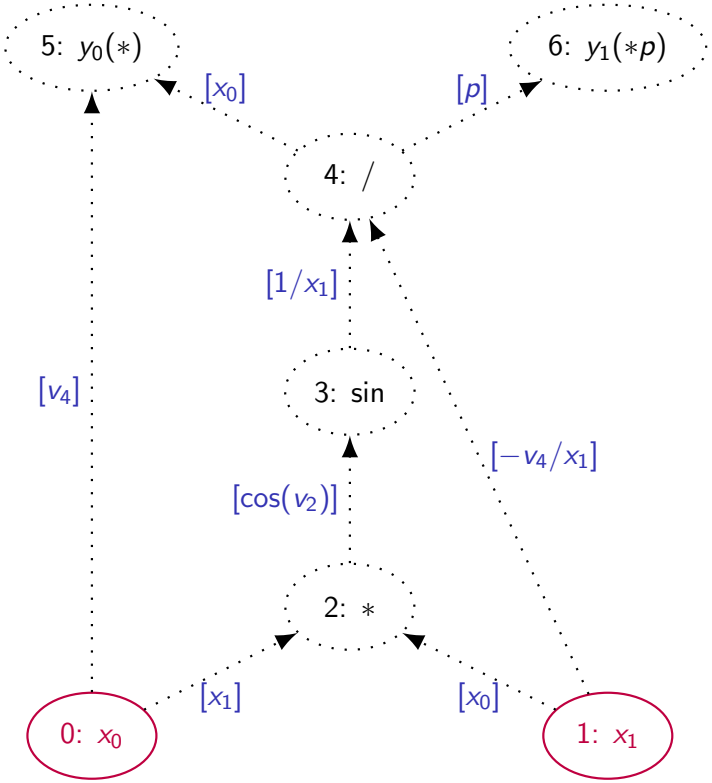
$$t = \sin(x_0 * x_1)/x_1$$
$$y_0 = x_0 * t$$
$$y_1 = t * p$$

yielding SAC

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

for some passive value p.

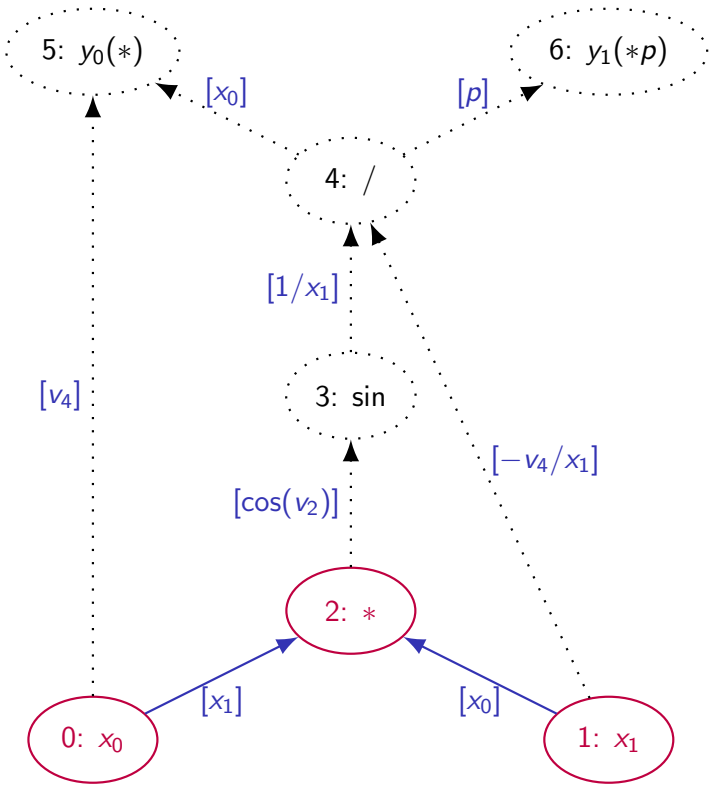
Adjoint AD Movie: Register



$x_0 = ?$

$x_1 = ?$

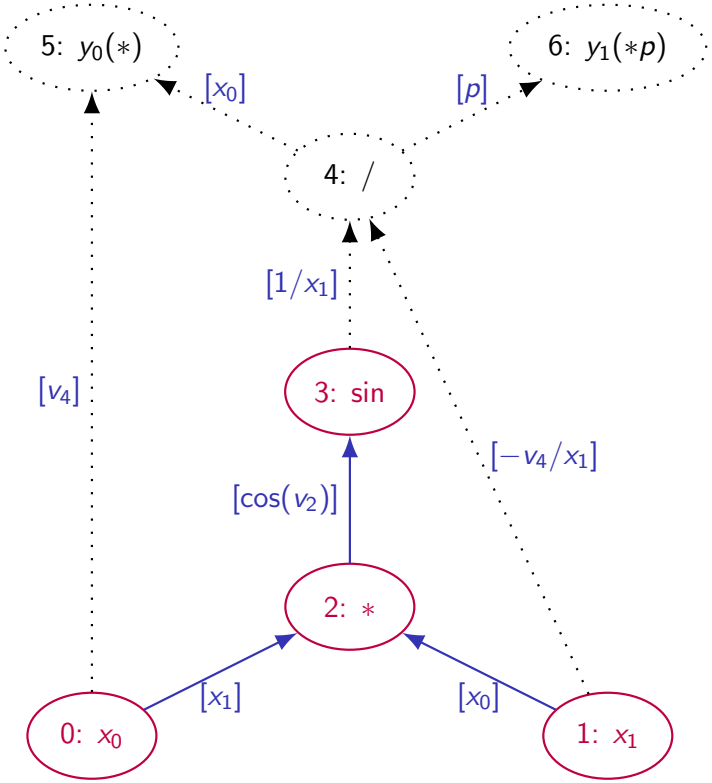
Adjoint AD Movie: Record



// overloaded $*$

$v_2 = x_0 * x_1$

Adjoint AD Movie: Record

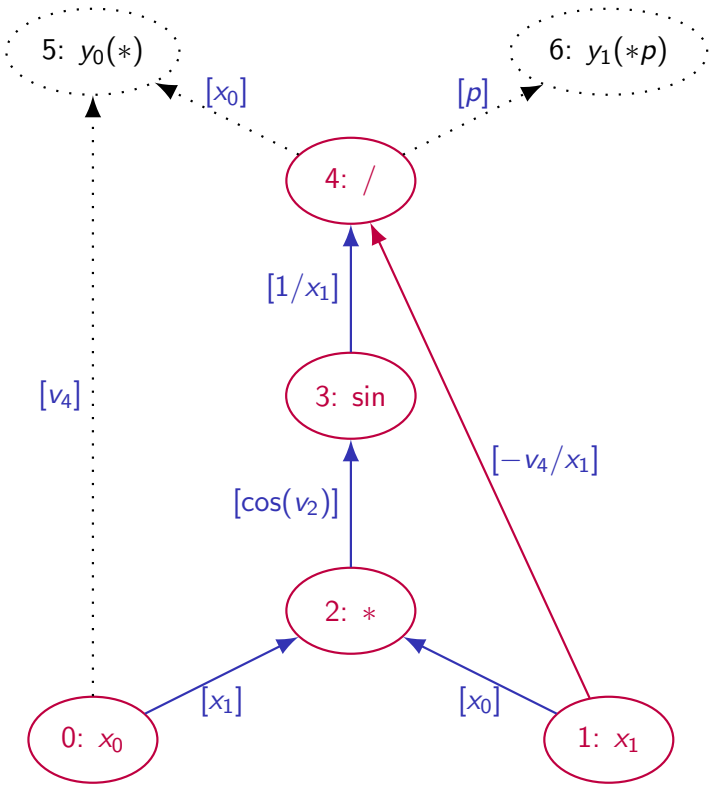


$v_2 = x_0 * x_1$

// overloaded sin

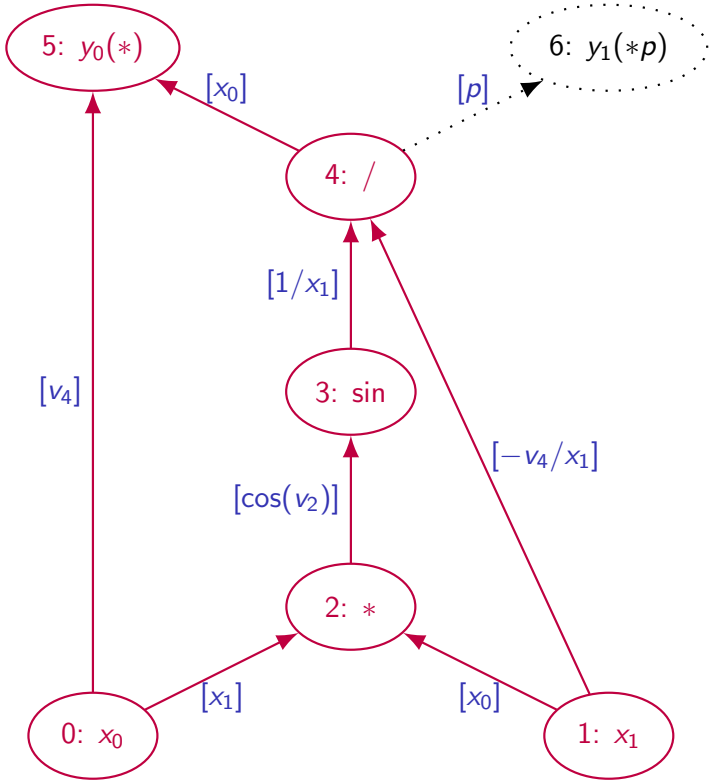
$v_3 = \sin(v_2)$

Adjoint AD Movie: Record



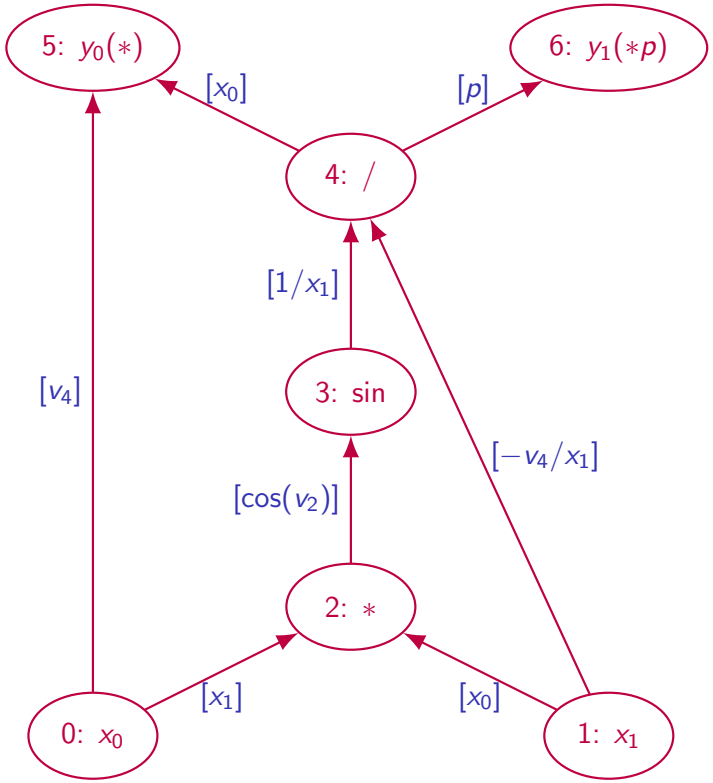
$v_2 = x_0 * x_1$
$v_3 = \sin(v_2)$
// overloaded /
$v_4 = v_3 / x_1$

Adjoint AD Movie: Record



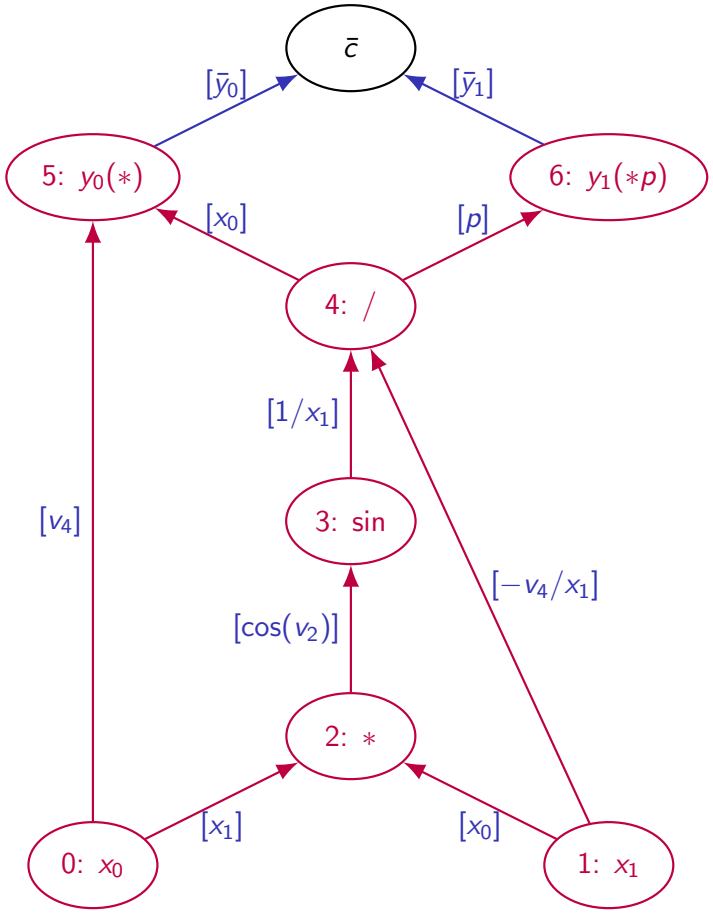
$$v_2 = x_0 * x_1$$
$$v_3 = \sin(v_2)$$
$$v_4 = v_3/x_1$$
// overloaded $*$
$$y_0 = x_0 * v_4$$

Adjoint AD Movie: Record



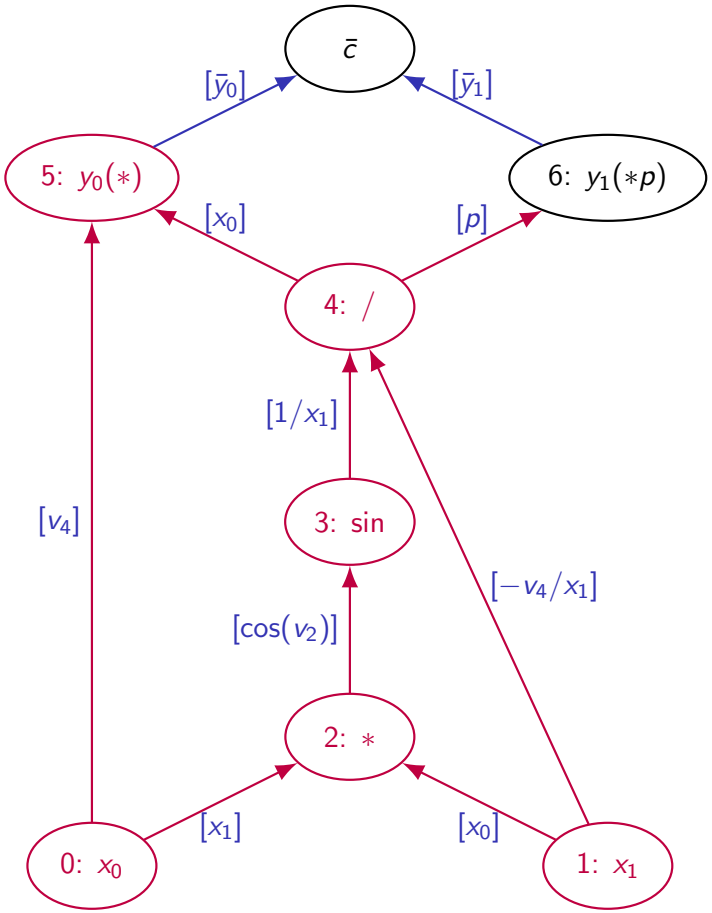
$$v_2 = x_0 * x_1$$
$$v_3 = \sin(v_2)$$
$$v_4 = v_3/x_1$$
$$y_0 = x_0 * v_4$$
// overloaded $*$
$$y_1 = v_4 * p$$

Adjoint AD Movie: Seed



$$v_2 = x_0 * x_1$$
$$v_3 = \sin(v_2)$$
$$v_4 = v_3/x_1$$
$$y_0 = x_0 * v_4$$
$$y_1 = v_4 * p$$
$$\bar{y}_0 = ?$$
$$\bar{y}_1 = ?$$
$$\bar{x}_0 = ?$$
$$\bar{x}_1 = ?$$
$$\bar{v}_2 = 0$$
$$\bar{v}_3 = 0$$
$$\bar{v}_4 = 0$$

Adjoint AD Movie: Interpret



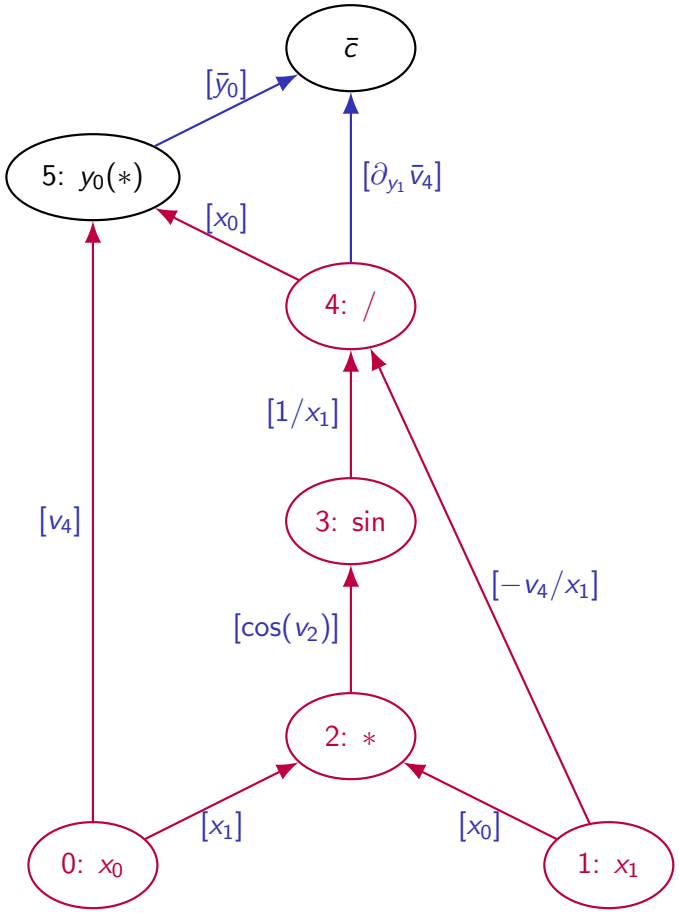
$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
v_3 &= \sin(v_2) \\
v_4 &= v_3 / x_1 \\
y_0 &= x_0 * v_4 \\
y_1 &= v_4 * p \\
\bar{v}_4 &\mathrel{+}= p * \bar{y}_1
\end{aligned}
$$

$$\bar{v}_4 \mathrel{+}= p * \bar{y}_1$$

$$\Leftrightarrow$$

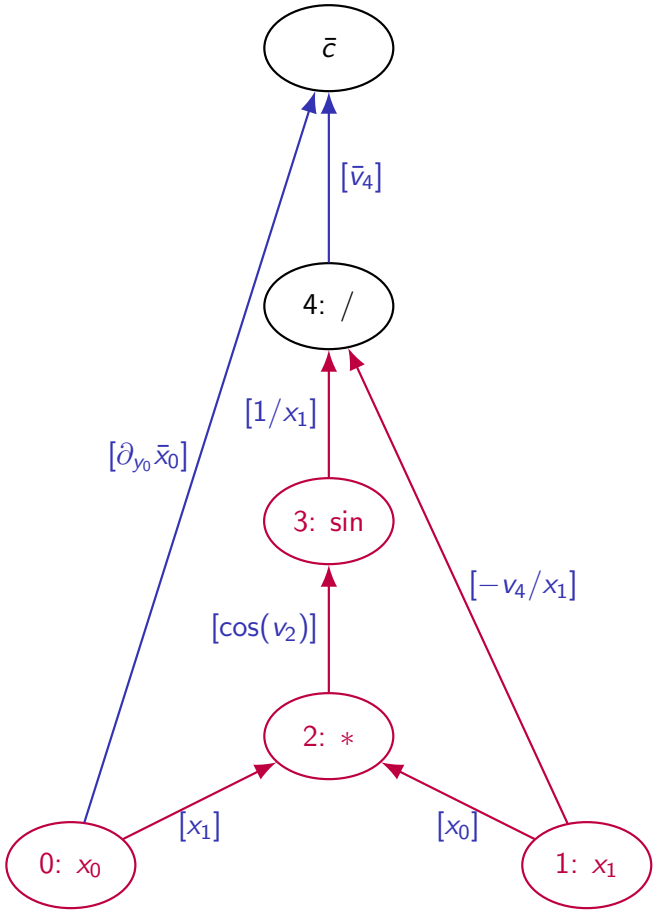
$$\bar{v}_4 = \bar{v}_4 + p * \bar{y}_1 .$$

Adjoint AD Movie: Interpret



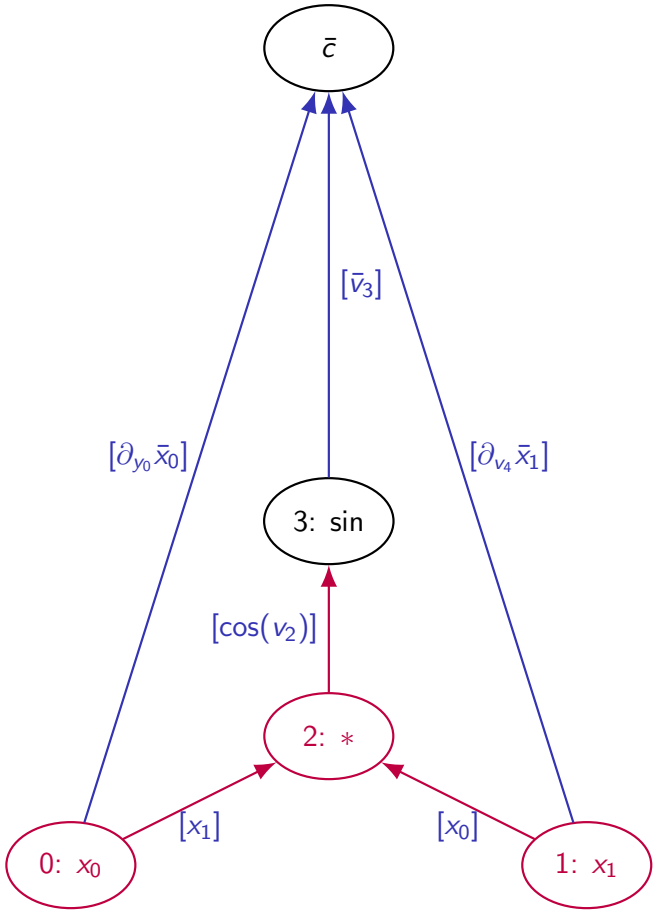
$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
v_3 &= \sin(v_2) \\
v_4 &= v_3 / x_1 \\
y_0 &= x_0 * v_4 \\
y_1 &= v_4 * p \\
\bar{v}_4 &\mathrel{+}= p * \bar{y}_1 \\
\bar{v}_4 &\mathrel{+}= x_0 * \bar{y}_0 \\
\bar{x}_0 &\mathrel{+}= v_4 * \bar{y}_0
\end{aligned}
$$

Adjoint AD Movie: Interpret



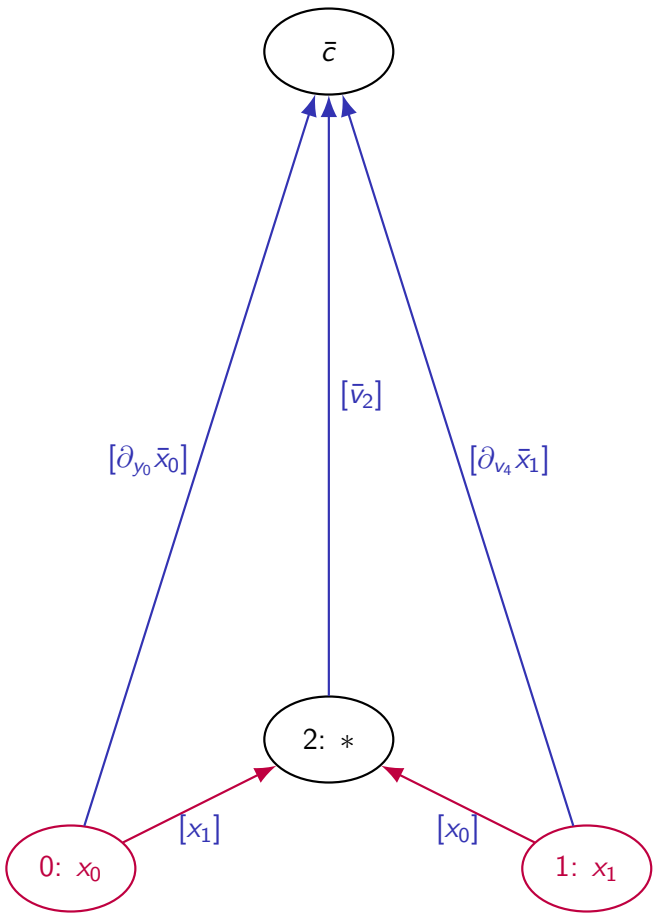
$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
v_3 &= \sin(v_2) \\
v_4 &= v_3 / x_1 \\
y_0 &= x_0 * v_4 \\
y_1 &= v_4 * p \\
\bar{v}_4 &\mathrel{+}= p * \bar{y}_1 \\
\bar{v}_4 &\mathrel{+}= x_0 * \bar{y}_0 \\
\bar{x}_0 &\mathrel{+}= v_4 * \bar{y}_0 \\
u &= 1/x_1 \\
\bar{v}_3 &\mathrel{+}= u * \bar{v}_4 \\
\bar{x}_1 &\mathrel{-}= v_4 * u * \bar{v}_4
\end{aligned}
$$

Adjoint AD Movie: Interpret



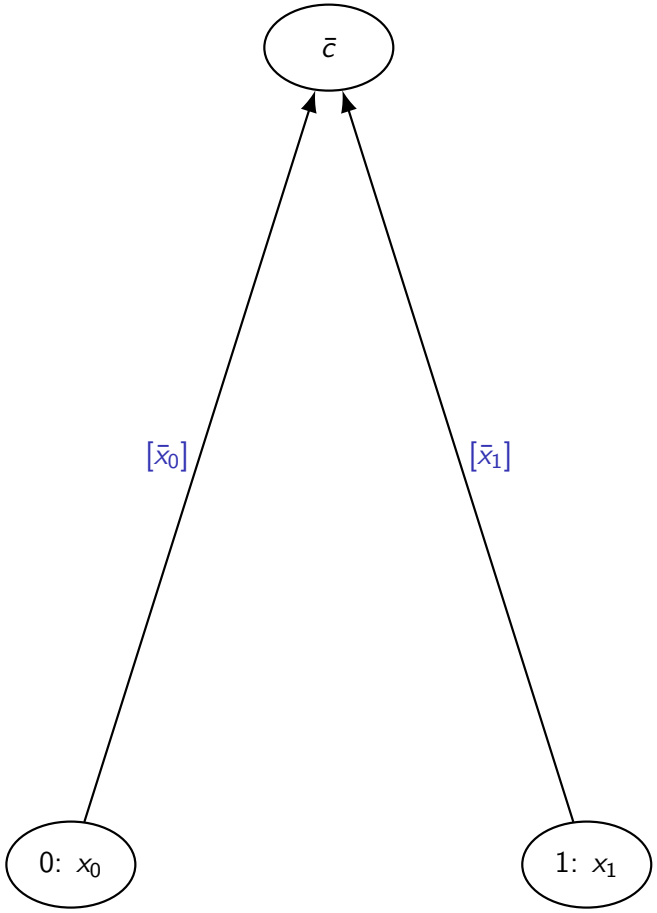
$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
v_3 &= \sin(v_2) \\
v_4 &= v_3 / x_1 \\
y_0 &= x_0 * v_4 \\
y_1 &= v_4 * p \\
\bar{v}_4 &\mathrel{+}= p * \bar{y}_1 \\
\bar{v}_4 &\mathrel{+}= x_0 * \bar{y}_0 \\
\bar{x}_0 &\mathrel{+}= v_4 * \bar{y}_0 \\
u &= 1 / x_1 \\
\bar{v}_3 &\mathrel{+}= u * \bar{v}_4 \\
\bar{x}_1 &\mathrel{-}= v_4 * u * \bar{v}_4 \\
\bar{v}_2 &\mathrel{+}= \cos(x_2) * \bar{v}_3
\end{aligned}
$$

Adjoint AD Movie: Interpret



$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
v_3 &= \sin(v_2) \\
v_4 &= v_3 / x_1 \\
y_0 &= x_0 * v_4 \\
y_1 &= v_4 * p \\
\bar{v}_4 &\mathrel{+}= p * \bar{y}_1 \\
\bar{v}_4 &\mathrel{+}= x_0 * \bar{y}_0 \\
\bar{x}_0 &\mathrel{+}= v_4 * \bar{y}_0 \\
u &= 1 / x_1 \\
\bar{v}_3 &\mathrel{+}= u * \bar{v}_4 \\
\bar{x}_1 &\mathrel{-}= v_4 * u * \bar{v}_4 \\
\bar{v}_2 &\mathrel{+}= \cos(x_2) * \bar{v}_3 \\
\bar{x}_0 &\mathrel{+}= x_1 * \bar{v}_2 \\
\bar{x}_1 &\mathrel{+}= x_0 * \bar{v}_2
\end{aligned}
$$

Adjoint AD Movie: Harvest



$$
\begin{aligned}
v_2 &= x_0 * x_1 \\
v_3 &= \sin(v_2) \\
v_4 &= v_3 / x_1 \\
y_0 &= x_0 * v_4 \\
y_1 &= v_4 * p \\
\bar{v}_4 &\mathrel{+}= p * \bar{y}_1 \\
\bar{v}_4 &\mathrel{+}= x_0 * \bar{y}_0 \\
\bar{x}_0 &\mathrel{+}= v_4 * \bar{y}_0 \\
u &= 1 / x_1 \\
\bar{v}_3 &\mathrel{+}= u * \bar{v}_4 \\
\bar{x}_1 &\mathrel{-}= v_4 * u * \bar{v}_4 \\
\bar{v}_2 &\mathrel{+}= \cos(x_2) * \bar{v}_3 \\
\bar{x}_0 &\mathrel{+}= x_1 * \bar{v}_2 \\
\bar{x}_1 &\mathrel{+}= x_0 * \bar{v}_2
\end{aligned}
$$

Cost of Adjoint AD

$$\textsc{Ert}(\bar{F}) = \mathcal{T} \cdot (|V| - n + |E|) + \bar{m} \cdot |E|)$$

$$\textsc{Mem}(\bar{F}) = |E| + |V|$$

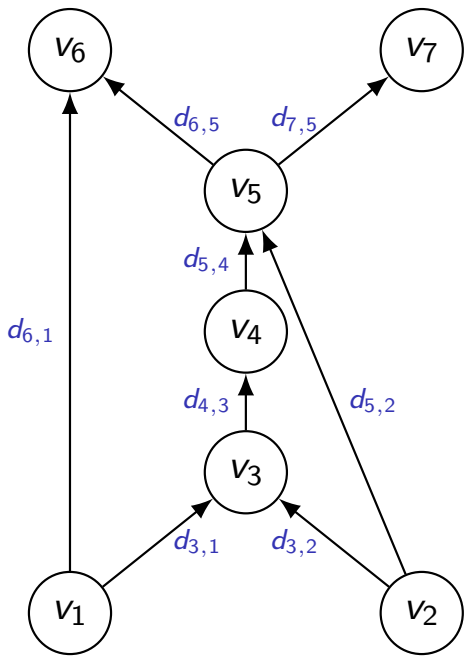
$$\textsc{Ert}(\bar{F}) = \frac{\textsc{Ert}(\bar{F})}{\textsc{Ert}(F)} \cdot \textsc{Ert}(F)$$

The minimization of the relative run time

$$0 < \mathcal{R} \equiv \frac{\textsc{Ert}(\bar{F})}{\textsc{Ert}(F)} \leq \infty$$

of an adjoint is a major challenge.

$\mathcal{T}$ quantifies the overhead induced by taping.



Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
```

Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
```

Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  // base type and active mode/type
6  using dco_value_type=double;
7  using dco_mode=dco::ga1s<dco_value_type>;
8  using dco_type=dco_mode::type;
```

Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  // base type and active mode/type
6  using dco_value_type=double;
7  using dco_mode=dco::ga1s<dco_value_type>;
8  using dco_type=dco_mode::type;
9
10 int main(int argc, char* argv[]) { // driver
11   assert(argc==3);
12   int np=stoi(argv[1]), ns=stoi(argv[2]);
```

Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  // base type and active mode/type
6  using dco_value_type=double;
7  using dco_mode=dco::ga1s<dco_value_type>;
8  using dco_type=dco_mode::type;
9
10 int main(int argc, char* argv[]) { // driver
11   assert(argc==3);
12   int np=stoi(argv[1]), ns=stoi(argv[2]);
13
14   // active program variables
15   dco_type x=1.; vector<dco_type> p(2*ns,1);
```

Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  // base type and active mode/type
6  using dco_value_type=double;
7  using dco_mode=dco::ga1s<dco_value_type>;
8  using dco_type=dco_mode::type;
9
10 int main(int argc, char* argv[]) { // driver
11   assert(argc==3);
12   int np=stoi(argv[1]), ns=stoi(argv[2]);
13
14   // active program variables
15   dco_type x=1.; vector<dco_type> p(2*ns,1);
16
17   dco::smart_tape_ptr_t<dco_mode> tape;  // tape
```

Gradient by Scalar Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  // base type and active mode/type
6  using dco_value_type=double;
7  using dco_mode=dco::ga1s<dco_value_type>;
8  using dco_type=dco_mode::type;
9
10 int main(int argc, char* argv[]) { // driver
11   assert(argc==3);
12   int np=stoi(argv[1]), ns=stoi(argv[2]);
13
14   // active program variables
15   dco_type x=1.; vector<dco_type> p(2*ns,1);
16
17   dco::smart_tape_ptr_t<dco_mode> tape;  // tape
18
19   // register active (independent) inputs
20   tape->register_variable(x);
21   for (int i=0;i<2*ns;i++) tape->register_variable(p[i]);
```

Gradient by Scalar Adjoint AD with dco/c++

```
22      dco_type y=f(np,ns,x,p); // record tape
```

Gradient by Scalar Adjoint AD with dco/c++

```
22      dco_type y=f(np,ns,x,p); // record tape
23
24      // register active (dependent) output
25      tape->register_output_variable(y);
```

Gradient by Scalar Adjoint AD with dco/c++

```
22      dco_type y=f(np,ns,x,p); // record tape
23
24      // register active (dependent) output
25      tape->register_output_variable(y);
26
27      dco::derivative(y)=1; // seed
```

Gradient by Scalar Adjoint AD with dco/c++

```
22      dco_type y=f(np,ns,x,p); // record tape
23
24      // register active (dependent) output
25      tape->register_output_variable(y);
26
27      dco::derivative(y)=1; // seed
28
29      tape->interpret_adjoint(); // interpret
```

Gradient by Scalar Adjoint AD with dco/c++

```
22      dco_type y=f(np,ns,x,p); // record tape
23
24      // register active (dependent) output
25      tape->register_output_variable(y);
26
27      dco::derivative(y)=1; // seed
28
29      tape->interpret_adjoint(); // interpret
30
31      // harvest
32      cout.precision(numeric_limits<dco_value_type>::digits10);
33      cout << "dy/dx=" << dco::derivative(x) << '\n';
34      for (int i=0;i<2*ns;i++)
35        cout << "dy/dp[" << i << "]="
36             << dco::derivative(p[i]) << '\n';
```

Gradient by Scalar Adjoint AD with dco/c++

```
22     dco_type y=f(np,ns,x,p); // record tape
23
24     // register active (dependent) output
25     tape->register_output_variable(y);
26
27     dco::derivative(y)=1; // seed
28
29     tape->interpret_adjoint(); // interpret
30
31     // harvest
32     cout.precision(numeric_limits<dco_value_type>::digits10);
33     cout << "dy/dx=" << dco::derivative(x) << '\n';
34     for (int i=0;i<2*ns;i++)
35       cout << "dy/dp[" << i << "]="
36            << dco::derivative(p[i]) << '\n';
37
38     cout << "y=" << dco::value(y) << endl; // primal value
39
40     return 0;
41   } // end of driver
```

Gradient by Scalar Adjoint AD with dco/c++

`SDE/ga1s/main.cpp`

- build (`Makefile`)
- run
- time (`/usr/bin/time -v`)
- compare with tangent AD

Experiments

np	ns	ERT (s)	RSS (MB)
10^4	10	< 0.1	14
10^4	100	0.1	109
10^4	200	0.2	214
10^4	300	0.3	320
10^5	10	0.1	111
10^5	100	0.8	1,061
10^6	10	0.8	1,089
10^6	100	-	$< 16,210$

Hands-on: Run and time on your computer.

Inspection of Scalar Adjoint AD with dco/c++

```
template<typename T>
void F(std::vector<T>& x) {
  while (x[1]<=0)
    x[1]=exp(x[0]*x[1])+x[1];
  x[0]=x[0]/x[1];
  x[1]=pow(x[1],3);
}
```

called with x[0]=x[1]=−1.

Tape	Adjoints ($\frac{dF_0}{dx}$):
7: 5, 1	0 // unused
6: 4, 1	−0.34
5: 3, 8.86	−0.58
4: 3, 0.34	0.34
4: 1, 0.58	1
3: 2, 1	0
3: 2, −2.72	1
3: 1, −2.72	0

See `simple/`.

Vector Adjoint AD

Vectors (also: ensembles) of k first-order adjoints of

$$F : \mathbb{R}^n \to \mathbb{R}^m : \quad y = F(x) \,,$$

can be evaluated in vector adjoint AD mode with machine accuracy as

$$\bar{X} = \bar{X} + \bar{F}(x, \bar{Y}) \equiv \bar{X} + \bar{Y} \cdot F'(x)$$

for $\bar{Y} \in \mathbb{R}^{k \times m}$ and $\bar{X} \in \mathbb{R}^{k \times n}$.

Performance of vector adjoint mode typically exceeds that of scalar adjoint mode even without the highly desirable explicit parallelization / vectorization. The memory requirement is increased.

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
```

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
```

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
```

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base type and active mode/type
10   using dco_value_type=double;
11   const int VEC_SIZE=1;
12   using dco_mode=dco::ga1v<dco_value_type,VEC_SIZE>;
13   using dco_type=dco_mode::type;
```

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base type and active mode/type
10   using dco_value_type=double;
11   const int VEC_SIZE=1;
12   using dco_mode=dco::ga1v<dco_value_type,VEC_SIZE>;
13   using dco_type=dco_mode::type;
14
15   // active program variables
16   dco_type x=1.; vector<dco_type> p(2*ns,1);
```

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base type and active mode/type
10   using dco_value_type=double;
11   const int VEC_SIZE=1;
12   using dco_mode=dco::ga1v<dco_value_type,VEC_SIZE>;
13   using dco_type=dco_mode::type;
14
15   // active program variables
16   dco_type x=1.; vector<dco_type> p(2*ns,1);
17
18   dco::smart_tape_ptr_t<dco_mode> tape;  // tape
```

Gradient by Vector Adjoint AD with dco/c++

```
1  #include "f.h" // primal
2
3  #include "dco.hpp" // dco/c++
4
5  int main(int argc, char* argv[]) { // driver
6    assert(argc==3);
7    int np=stoi(argv[1]), ns=stoi(argv[2]);
8
9    // base type and active mode/type
10   using dco_value_type=double;
11   const int VEC_SIZE=1;
12   using dco_mode=dco::ga1v<dco_value_type,VEC_SIZE>;
13   using dco_type=dco_mode::type;
14
15   // active program variables
16   dco_type x=1.; vector<dco_type> p(2*ns,1);
17
18   dco::smart_tape_ptr_t<dco_mode> tape;  // tape
19
20   // register active (independent) input
21   tape->register_variable(x);
22   for (int i=0;i<2*ns;i++) tape->register_variable(p[i]);
```

Gradient by Vector Adjoint AD with dco/c++

```
23      dco_type y=f(np,ns,x,p); // record tape
```

Gradient by Vector Adjoint AD with dco/c++

```
23      dco_type y=f(np,ns,x,p); // record tape
24
25      // register active (dependent) output
26      tape->register_output_variable(y);
```

Gradient by Vector Adjoint AD with dco/c++

```
23      dco_type y=f(np,ns,x,p); // record tape
24
25      // register active (dependent) output
26      tape->register_output_variable(y);
27
28      dco::derivative(y)[0]=1; // seed
```

Gradient by Vector Adjoint AD with dco/c++

```
23      dco_type y=f(np,ns,x,p); // record tape
24
25      // register active (dependent) output
26      tape->register_output_variable(y);
27
28      dco::derivative(y)[0]=1; // seed
29
30      tape->interpret_adjoint(); // interpret
```

Gradient by Vector Adjoint AD with dco/c++

```
23      dco_type y=f(np,ns,x,p); // record tape
24
25      // register active (dependent) output
26      tape->register_output_variable(y);
27
28      dco::derivative(y)[0]=1; // seed
29
30      tape->interpret_adjoint(); // interpret
31
32      // harvest
33      cout.precision(numeric_limits<dco_value_type>::digits10);
34      cout << "dy/dx=" << dco::derivative(x)[0] << '\n';
35      for (int i=0;i<2*ns;i++)
36        cout << "dy/dp[" << i << "]="
37             << dco::derivative(p[i])[0] << '\n';
```

Gradient by Vector Adjoint AD with dco/c++

```
23     dco_type y=f(np,ns,x,p); // record tape
24
25     // register active (dependent) output
26     tape->register_output_variable(y);
27
28     dco::derivative(y)[0]=1; // seed
29
30     tape->interpret_adjoint(); // interpret
31
32     // harvest
33     cout.precision(numeric_limits<dco_value_type>::digits10);
34     cout << "dy/dx=" << dco::derivative(x)[0] << '\n';
35     for (int i=0;i<2*ns;i++)
36       cout << "dy/dp[" << i << "]="
37            << dco::derivative(p[i])[0] << '\n';
38
39     cout << "y=" << dco::value(y) << endl; // primal value
40
41     return 0;
42   } // end of driver
```

Note missing benefit for $m = 1$.

Live: Variants of Adjoint AD with dco/c++

- mutable independent
  `SDE/ga1s/variants/mutable_indep`
- tape types and custom tape sizes
  `SDE/ga1s/variants/custom_tapesize`
- mixed tape base types
  `SDE/ga1s/variants/mixed_base_types`
- file tape
  `SDE/ga1s/variants/file_tape`

inspect, build (`Makefile`), run, time (`/usr/bin/time -v`), compare with default adjoint AD

Variants: Mutable Independent

- x is overwritten due to passing to f by reference.
- Default adjoint driver yields incorrect derivative wrt. x.
- Independent input value of x needs to be made read-only, e.g.,

```
dco_type x_in=1.; // read-only independent
tape->register_variable(x_in);
...
dco_type x=x_in; // mutable copy
```

Live: `SDE/ga1s/variants/mutable_indep`

Variants: Tape Types and Custom Tape Sizes

- The *blob* tape (default)
  - allocates half of available RAM (by default).
  - can be customized as follows:

```
dco::tape_options o; o.set_blob_size_in_gbyte(1);
dco::smart_tape_ptr_t<dco_mode> tape(o);
```

- The *chunk* tape (compiler flag: `-DDCO_CHUNK_TAPE`)
  - fills available RAM in chunks of preset size
  - can be customized as follows:

```
dco::tape_options o; o.set_chunk_size_in_mbyte(4);
dco::smart_tape_ptr_t<dco_mode> tape(o);
```

Live: `SDE/ga1s/variants/custom_tapesize`

Variants: Mixed Tape Base Types

- So far, we have been working with uniformly typed tapes and adjoints, i.e.,

```
using dco_value_type=double;
using dco_mode=dco::ga1s<dco_value_type>;

dco::smart_tape_ptr_t<dco_mode> tape;
```

- In fact, different types can be selected for values, partial derivatives and adjoints, e.g.,

```
using dco_partial_type=float;
using dco_adjoint_type=float;
using dco_mode=dco::ga1s<dco_value_type,
                         dco_partial_type,
                         dco_adjoint_type>;
```

Live: `SDE/ga1s/variants/mixed_base_types`

Variants: File Tape

- The RAM occupied by a chunk tape can be extended to the hard disc as

```
dco::tape_options o;
o.write_to_file()=true; o.set_chunk_size_in_mbyte(4);
dco::smart_tape_ptr_t<dco_mode> tape(o);
```

- Chunks written into files are removed automatically once tape leaves its scope unlessi the program is aborted prematurely, e.g., using

```
assert(false);
```

Live: `SDE/ga1s/variants/file_tape`

Hands-On

For the given implementation of the Heston Stochastic Volatility Model with Euler Discretization use dco/c++ to compute the gradient in scalar adjoint mode.

Run experiments with variants of adjoint mode including

- different tape types and sizes
- mixed base types
- file tape

Compare performances and relate to tangent modes.

Differential Invariant

The identity

$$\bar{x} \cdot \dot{x} = \bar{y} \cdot \dot{y} \quad (\bar{X} \cdot \dot{X} = \bar{Y} \cdot \dot{Y})$$

holds for any compact sub-program of F. ($\Rightarrow$ Use it for validation / *debugging*.)

Proof:

$$\bar{x} \cdot \dot{x} = (\bar{y} \cdot F') \cdot \dot{x} = \bar{y} \cdot (F' \cdot \dot{x}) = \bar{y} \cdot \dot{y}$$

Similarly, differential invariants can be derived for second- and higher-order AD.

U. N.: *Differential Invariants.* arXiv:2101.03334 [math.NA].

Outline

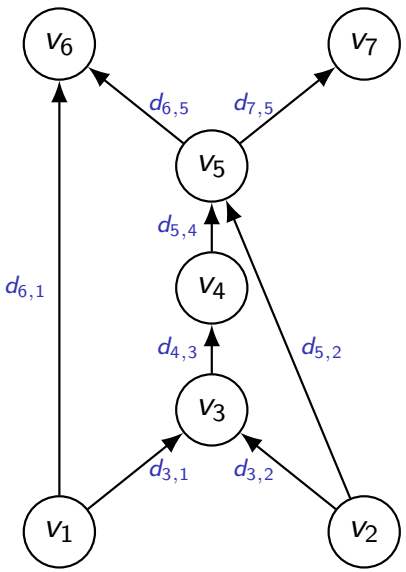
Beyond Black-Box Adjoints: Early Intervention

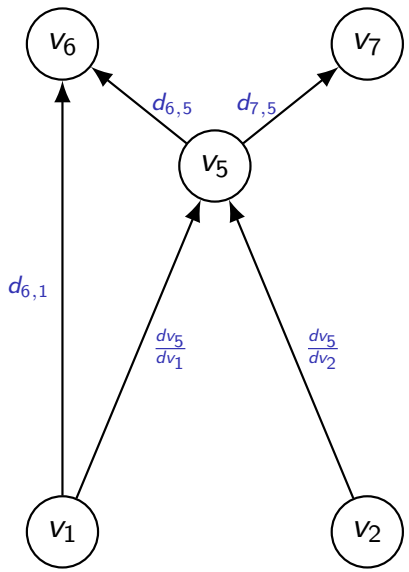
Contents:

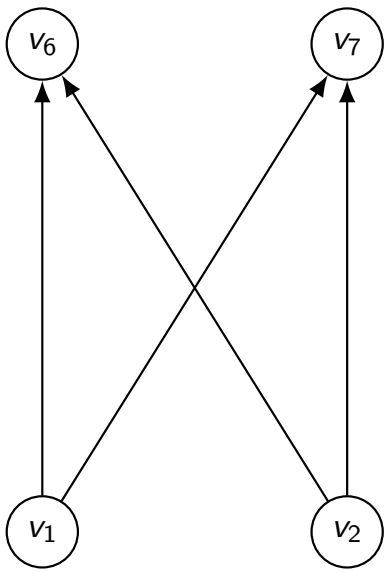
- Early Recording [and Late Back-Propagation] (Default)
- Early Preaccumuation
- Early Back-Propagation

Motivation: $y = F(x) = f(g(x), x)$ s.t. $g : x \mapsto v_5,\ f : (x_0, v_5) \mapsto y$

$(y, \bar{x}) = \bar{F}(x, I_2)$ $\qquad$ $(y, T^{g',f}) = \bar{g}(x, 1) \circ \overrightarrow{f}(x_0, v_5)$ $\qquad$ $\bar{x} = \overleftarrow{f}(I_2, T^{g',f})$







$\textsc{Mem}(\bar{F}) = 8 + 7^4 = 15$	$\textsc{Mem}(\bar{F}) = 5 + 5 + \ldots$	$\ldots + 0 = 10$
$OPS(\overrightarrow{F}) = 5 + 8 = 13$	$OPS(\overrightarrow{F}) = 5 + 8 + \ldots$	$\ldots + 0 = 13$
$OPS(\overleftarrow{F}) = 2 \cdot 8 = 16$	$OPS(\overleftarrow{F}) = 1 \cdot 5 + \ldots$	$\ldots + 2 \cdot 5 = 15$

[4] ... plus (invariant) memory requirement of primal

Terminology

AD can be applied naively to differentiable programs. Adjoint AD may require a more fine-grain treatment to ensure feasible memory requirement and optimal run time.

- The augmented primal section records the tape (default: early recording).
- The adjoint section (back-)propagates adjoints / interprets the tape (default: late back-propagation).
- Early intervention interrupts recording.
- Late intervention interrupts (recording and) back-propagation.
- (Partial) Preaccumulation yields (incomplete) local Jacobians of parts of the program.

Notation

- primal differentiable (sub-)program

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

- derivative (sub-)program

$$F' : \mathbb{R}^n \to \mathbb{R}^{m \times n} : F_x = F'(x) \quad (\text{or } (y, F_x) = F'(x))$$

- (vector) tangent (sub-)program

$$\dot{F} : \mathbb{R}^n \times \mathbb{R}^{n \times \dot{n}} \to \mathbb{R}^m \times \mathbb{R}^{m \times \dot{n}} : (y, \dot{Y}) = \dot{F}(x, \dot{X})$$

- (vector) adjoint (sub-)program

$$\bar{F} : \mathbb{R}^n \times \mathbb{R}^{\bar{m} \times m} \to \mathbb{R}^m \times \mathbb{R}^{\bar{m} \times n} : (y, \bar{X}) = \bar{F}(x, \bar{Y})$$

- augmented section of adjoint (sub-)program

$$\overrightarrow{F} : \mathbb{R}^n \to \mathbb{R}^m \times T : (y, T) = \overrightarrow{F}(x)$$

- adjoint section of adjoint (sub-)program

$$\overleftarrow{F} : \mathbb{R}^{\bar{m} \times m} \times T \to \mathbb{R}^{\bar{m} \times n} : \bar{X} = \overleftarrow{F}(\bar{Y}, T)$$

Further Notation

- $\textsc{Ert}(F)$ elapsed run time of primal (sub-)program
- $\textsc{Ert}(\dot{F})$ elapsed run time of tangent (sub-)program
- $\textsc{Ert}(\bar{F})$ elapsed run time of adjoint (sub-)program
- $\textsc{Ert}(\overrightarrow{F})$ elapsed run time of augmented primal section of adjoint (sub-)program
- $\textsc{Ert}(\overleftarrow{F})$ elapsed run time of adjoint section of adjoint (sub-)program
- $\textsc{Mem}(F) = 0$ persistent memory requirement of primal (sub-)program
- $\textsc{Mem}(\dot{F}) = 0$ persistent memory requirement of tangent (sub-)program
- $\textsc{Mem}(\bar{F})$ persistent memory of adjoint (sub-)program
- $\textsc{Mem}(\overrightarrow{F})$ persistent memory of augmented primal section of adjoint (sub-)program
- $\textsc{Mem}(\overleftarrow{F})$ persistent memory of adjoint section of adjoint (sub-)program
- $\hat{\textsc{Mem}}$ sharp upper bound on persistent memory available

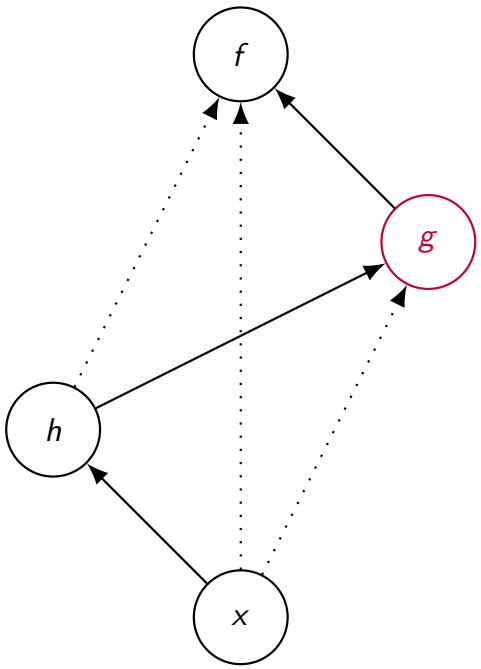
(Generic) Scenario

$$y = F(x) = f(g(h(x))) : \ \mathbb{R}^n \to \mathbb{R}^m$$

$$h : \mathbb{R}^n \to \mathbb{R}^{m_h}$$
$$g : \mathbb{R}^{n_g} \to \mathbb{R}^{m_g}$$
$$f : \mathbb{R}^{n_f} \to \mathbb{R}^m \ .$$

This generic scenario is used to illustrate the various modes of intervention applied to g.

W.l.o.g., further potential data dependences (dotted in the figure on the right) are omitted in order to limit notational complexity to the essential minimum, that is, $m_h = n_g$ and $m_g = n_f$.



(Generic) Scenario: Chain Rule

Jacobian:

$$\mathbb{R}^{m\times n} \ni F' \equiv \frac{dF}{dx} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x} = f' \cdot g' \cdot h'$$

Tangent:

$$\mathbb{R}^{m\times \dot{n}} \ni \dot{Y} = \frac{dF}{dx} \cdot \dot{X} = f' \cdot g' \cdot h' \cdot \dot{X}$$

Adjoint:

$$\mathbb{R}^{\bar{m}\times n} \ni \bar{X} = \bar{Y} \cdot \frac{dF}{dx} = \bar{Y} \cdot f' \cdot g' \cdot h'$$

Early Intervention

The objective of early intervention is a lower elapsed run time

$$\textsc{Ert}(\bar{F}) \rightarrow \min$$

and / or a reduction of the maximum resident set size

$$\textsc{Mem}(\bar{F}) \rightarrow \ (\leq \hat{\textsc{Mem}})$$

of the adjoint through (combination of)

- Early Back-Propagation
- Early Preaccumulation

Early Back-Propagation

The (elapsed run time of the) Early Back-Propagation pattern evolves as

$$
\begin{array}{lll}
1: & \textsc{Ert}[\bar{F}] = 0 & \\
2: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Ert}}[h] & (\text{record } T^h) \\
3: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Ert}}[g] & (\text{record } T^g) \\
4: & \textsc{Ert}[\bar{F}] \mathrel{+}= \textsc{Ert}[f] & (\text{evaluate } f) \\
5: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Ert}}[g] & (\text{interpret } T^g) \\
6: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Ert}}[h] & (\text{interpret } T^h)
\end{array}
$$

subject to $\textsc{Mem}[\bar{F}] \leq \hat{\textsc{Mem}}$.

Use Early Back-Propagation whenever $\bar{G}$ is known prior to the evaluation of $\bar{f}$ implying that $\bar{g}$ can be evaluated without recording and interpreting the tape of f.

Early Back-Propagation: Memory Requirement

$$\textsc{Mem}[\bar{F}] = \overrightarrow{\textsc{Mem}}[h] + \overrightarrow{\textsc{Mem}}[g] + \overleftarrow{\textsc{Mem}}[h] + \overleftarrow{\textsc{Mem}}[g] \tag{1}$$

Proof:

1:	$\textsc{Mem}[\bar{F}] = 0$	
2:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Mem}}[h]$	(record T^h)
3:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Mem}}[g]$	(record T^g)
4:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Mem}}[h] + \overleftarrow{\textsc{Mem}}[g]$	(allocate adjoints)
5:	$\textsc{Mem}[\bar{F}] \mathrel{+}= 0$	(evaluate f)
6:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overrightarrow{\textsc{Mem}}[g]$	(interpret T^g)
7:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overrightarrow{\textsc{Mem}}[h]$	(interpret T^h)
8:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overleftarrow{\textsc{Mem}}[h] + \overleftarrow{\textsc{Mem}}[g]$	(free adjoints) .

$\textsc{Mem}[\bar{F}]$ takes local maxima just prior to decrementation, that is, in line 5, implying Equation (1).

Early Back-Propagation: Pathwise Adjoints

```
1  template<>
2  dco_type f(
3      int np, int ns, dco_type x, const vector<dco_type>& p
4  ) {
5    default_random_engine g;
6    normal_distribution<float> d(0,1);
7    dco_type y=0, x_in=x; auto tape=dco::tape(x);
8    // avoid repeated interpretation of upstream tape
9    auto pos=tape->get_position();
10   for (int i=0;i<np;++i) {
11     x=x_in;
12     for (int j=0;j<ns;++j)
13       x+=exp(-p[j]*x)/ns+p[ns+j]*x*sqrt(1./ns)*d(g);
14     dco::derivative(x)=1./np; // known \bar{G}
15     tape->interpret_adjoint_and_reset_to(pos);
16     y+=x;
17   }
18   return y/np;
19 }
```

Live: `code/SDE/ga1s/variants/pathwise`

Early Preaccumulation (in Adjoint Mode)

The elapsed run time of the Early Preaccumulation (in Adjoint Mode) pattern evolves as

$$
\begin{array}{lll}
1: & \textsc{Ert}[\bar{F}] = 0 & \\
2: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Ert}}[h] & \text{(record } T^h\text{)} \\
3: & \textsc{Ert}[\bar{F}] \mathrel{+}= \textsc{Ert}[g'] & \text{(preaccumulate } g'\text{)} \\
4: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Ert}}[g'] & \text{(record } T^{g'}\text{)} \\
5: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Ert}}[f] & \text{(record } T^f\text{)} \\
6: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Ert}}[f] & \text{(interpret } T^f\text{)} \\
7: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Ert}}[T^{g'}] & \text{(interpret } T^{g'}\text{)} \\
8: & \textsc{Ert}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Ert}}[h] & \text{(interpret } T^h\text{)}
\end{array}
$$

subject to $\textsc{Mem}[\bar{F}] \leq \hat{\textsc{Mem}}$ and where g' preaccumulates the (partial) Jacobian of g, which subsequently needs to be recorded to become part of the global tape.

Early Preaccumulation (in Adjoint Mode)

The corresponding persistent memory requirement evolves as

1:	$\textsc{Mem}[\bar{F}] = 0$	
2:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Mem}}[h]$	(record T^h)
3:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Mem}}[g]$	(record for preaccumulation)
4:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overleftarrow{\textsc{Mem}}[h] + \overleftarrow{\textsc{Mem}}[g]$	(allocate adjoints)
5:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overrightarrow{\textsc{Mem}}[g']$	(preaccumulate $T^{g'}$)
6:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Mem}}[T^{g'}]$	(record $T^{g'}$)
7:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \overrightarrow{\textsc{Mem}}[f]$	(record T^f)
8:	$\textsc{Mem}[\bar{F}] \mathrel{+}= \max\{\overleftarrow{\textsc{Mem}}[f] + \overleftarrow{\textsc{Mem}}[g'] - \overleftarrow{\textsc{Mem}}[g], 0\}$	(grow memory for adjoints?)
9:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overrightarrow{\textsc{Mem}}[f]$	(interpret T^f)
10:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overrightarrow{\textsc{Mem}}[T^{g'}]$	(interpret $T^{g'}$)
11:	$\textsc{Mem}[\bar{F}] \mathrel{-}= \overrightarrow{\textsc{Mem}}[h]$	(interpret T^h)
12:	$\textsc{Mem}[\bar{F}] = 0$	(free adjoints)

Early Preaccumulation: Local Jacobians

```
1  template<>
2  dco_type f(
3      int np, int ns, dco_type x, const vector<dco_type>& p
4  ) {
5    dco_mode::jacobian_preaccumulator_t jp(dco::tape(x)); // !
6    default_random_engine g;
7    normal_distribution<float> d(0,1);
8    dco_type y=0, x_in=x;
9    for (int i=0;i<np;++i) {
10     x=x_in;
11     jp.start(); // start recording local tape
12     for (int j=0;j<ns;++j)
13       x+=exp(-p[j]*x)/ns+p[ns+j]*x*sqrt(1./ns)*d(g);
14     jp.register_output(x); register local output(s)
15     jp.finish(); // preaccumulate local Jacobian
16     y+=x;
17   }
18   return y/np;
19 }
```

Live: `code/SDE/ga1s/variants/jacobian_preaccumulator_t`

Early Preaccumulation: Options

Preaccumulation by local

- jacobian_preaccumulator_t ("convenience feature")
- adjoint AD (sparsity?)
- tangent AD ($n_g \leq \frac{\textsc{Ert}(\bar{g}))}{\textsc{Ert}(\dot{g})} \cdot m_g$ or $|T \setminus T^f| > \overline{\textsc{Mem}}$; sparsity)
- finite difference (smoothing?)
- hand coding (better local performance?)
- elimination techniques (scarcity?)
- *different tool (e.g. not C++ or dco/map for GPGPU)*

Outline

Basic Linear Algebra Subprograms: Outline

- axpy
  - tangents and adjoints
- inner vector product
  - tangents and adjoints
  - implementation and validation
- matrix-vector product
  - tangents and adjoints
  - implementation and validation
- matrix-matrix product
  - tangents and adjoints
  - implementation and validation
- consequences

axpy: Tangent and Adjoint

The tangent of the axpy operation $z := a \cdot x + y$ with active $a, x, y, z \in \mathbb{R}$ is computed as

$$z^{(1)} := x \cdot a^{(1)} + a \cdot x^{(1)} + y^{(1)} \tag{2}$$

for $a^{(1)}, x^{(1)}, y^{(1)}, z^{(1)} \in \mathbb{R}$.

The corresponding adjoint is computed as

$$\begin{aligned} a_{(1)} &:= z_{(1)} \cdot x \\ x_{(1)} &:= z_{(1)} \cdot a \\ y_{(1)} &:= z_{(1)} \end{aligned} \tag{3}$$

for $a_{(1)}, x_{(1)}, y_{(1)}, z_{(1)} \in \mathbb{R}$.

Inner Vector Product: Tangent and Adjoint

The tangent of an inner vector product

$$y := \langle \mathbf{a}, \mathbf{x} \rangle \equiv \mathbf{a}^T \cdot \mathbf{x} = \sum_{i=0}^{n-1} a_i \cdot x_i$$

with active $\mathbf{a} \in \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^n$ and $y \in \mathbb{R}$ is computed as

$$y^{(1)} := \langle \mathbf{a}^{(1)}, \mathbf{x} \rangle + \langle \mathbf{a}, \mathbf{x}^{(1)} \rangle \tag{4}$$

with $y^{(1)} \in \mathbb{R}$ and $\mathbf{a}^{(1)}, \mathbf{x}^{(1)} \in \mathbb{R}^n$.

The corresponding adjoint is computed as

$$\begin{aligned} \mathbf{a}_{(1)} &:= y_{(1)} \cdot \mathbf{x}^T \\ \mathbf{x}_{(1)} &:= y_{(1)} \cdot \mathbf{a}^T \end{aligned} \tag{5}$$

with $y_{(1)} \in \mathbb{R}$ and $\mathbf{a}_{(1)}, \mathbf{x}_{(1)} \in \mathbb{R}^{1 \times n}$.

Inner Vector Product: Implementation

```
1
2  template<typename T>
3  T aTx( la::vector_t<T> a, la::vector_t<T> x) {
4    return a.dot(x);
5  }
6
7  template<typename T>
8  T aTx_sym_tan(
9      la::vector_t<T> a, la::vector_t<T> a_t,
10     la::vector_t<T> x, la::vector_t<T> x_t
11 ) {
12   return a_t.dot(x)+a.dot(x_t);
13 }
14
15 template<typename T>
16 void aTx_sym_adj(
17     Eigen::vector_t<T> a, Eigen::row_vector_t<T>& a_a,
18     Eigen::vector_t<T> x, Eigen::row_vector_t<T>& x_a, T y_a
19 ) {
20   a_a=y_a*x.transpose();
21   x_a=y_a*a.transpose();
22 }
```

Inner Vector Product: Validation

```
template<typename T>
void approximate_tangent(
    Eigen::vector_t<T> a, Eigen::vector_t<T> a_t,
    Eigen::vector_t<T> x, Eigen::vector_t<T> x_t,
    T y_t
) {
  const T h=sqrt(std::numeric_limits<T>::epsilon());
  Eigen::vector_t<T> ap=a+h*a_t, xp=x+h*x_t;
  cout << "\napproximate tangent (ffd)" << endl;
  cout << (aTx(ap,xp)-aTx(a,x))/h << "==" << y_t << '?' <<
    endl;
}

template<typename T>
void validate_invariant(
    Eigen::vector_t<T> a_t, Eigen::row_vector_t<T> a_a,
    Eigen::vector_t<T> x_t, Eigen::row_vector_t<T> x_a,
    T y_t, T y_a
) {
  cout << "\ndifferential invariant:" << endl;
  cout << a_a*a_t+x_a*x_t << "==" << y_t*y_a << '?' << endl;
}
```

Matrix-Vector Product: Tangent and Adjoint

The tangent of a matrix-vector product $\mathbf{y} := A \cdot \mathbf{x}$ is computed as

$$\mathbf{y}^{(1)} := A^{(1)} \cdot \mathbf{x} + A \cdot \mathbf{x}^{(1)} \ . \tag{6}$$

Given $A, A^{(1)} \in \mathbb{R}^{m \times n}$ and $x, x^{(1)} \in \mathbb{R}^n$ yield $\mathbf{y}^{(1)} \in \mathbb{R}^m$.

The corresponding adjoint is computed as

$$\begin{aligned} \mathbf{x}_{(1)} &:= \mathbf{y}_{(1)} \cdot A \\ A_{(1)} &:= \mathbf{x} \cdot \mathbf{y}_{(1)} \ . \end{aligned} \tag{7}$$

Given $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $\mathbf{y}_{(1)} \in \mathbb{R}^{1 \times m}$ yield $\mathbf{x}_{(1)} \in \mathbb{R}^{1 \times n}$ and $A_{(1)} \in \mathbb{R}^{n \times m}$.

Matrix-Vector Product: Implementation

```
template<typename T>
Eigen::vector_t<T> Ax(
    Eigen::matrix_t<T> A, Eigen::vector_t<T> x
) { return A*x; }

template<typename T>
la::vector_t<T> Ax_sym_tan(
    la::matrix_t<T> A, la::matrix_t<T> A_t,
    la::vector_t<T> x, la::vector_t<T> x_t
) { return A*x_t+A_t*x; }

template<typename T>
void Ax_sym_adj(
    Eigen::matrix_t<T> A, Eigen::matrix_t<T>& A_a,
    Eigen::vector_t<T> x, Eigen::row_vector_t<T>& x_a,
    Eigen::row_vector_t<T> y_a
) { A_a=x*y_a; x_a=y_a*A; }
```

See full sample code for dco/c++ tangent and adjoint and for validation.

Matrix-Matrix Product: Tangent and Adjoint

The tangent of a matrix-matrix product $Y := A \cdot X$ is computed as

$$Y^{(1)} := A^{(1)} \cdot X + A \cdot Y^{(1)} \,. \tag{8}$$

Given $A, A^{(1)} \in \mathbb{R}^{m \times p}$ and $X, X^{(1)} \in \mathbb{R}^{p \times n}$ yield $Y^{(1)} \in \mathbb{R}^{m \times n}$.

The corresponding adjoint is computed as

$$\begin{aligned} X_{(1)} &:= Y_{(1)} \cdot A \\ A_{(1)} &:= X \cdot Y_{(1)} \,. \end{aligned} \tag{9}$$

Given $A \in \mathbb{R}^{m \times p}$, $X \in \mathbb{R}^{p \times n}$ and $Y_{(1)} \in \mathbb{R}^{n \times m}$ yield $A_{(1)} \in \mathbb{R}^{p \times m}$ and $X_{(1)} \in \mathbb{R}^{n \times p}$.

Matrix-Matrix Product: Implementation

```
template<typename T>
la::matrix_t<T> AX(
    la::matrix_t<T> A, la::matrix_t<T> X
) { return A*X; }

template<typename T>
la::matrix_t<T> AX_sym_tan(
    la::matrix_t<T> A, la::matrix_t<T> A_t,
    la::matrix_t<T> X, la::matrix_t<T> X_t
) { return A*X_t+A_t*X; }

template<typename T>
void AX_sym_adj(
    Eigen::matrix_t<T> A, Eigen::matrix_t<T>& A_a,
    Eigen::matrix_t<T> X, Eigen::matrix_t<T>& X_a,
    Eigen::matrix_t<T> Y_a
) { A_a=X*Y_a; X_a=Y_a*A; }
```

See full sample code for dco/c++ tangent and adjoint and for validation.

Matrix Factorization, e.g. $A = L \cdot L^T$

From

$$\begin{pmatrix} \alpha & a^T \\ a & A_* \end{pmatrix} = \begin{pmatrix} \rho & 0 \\ r & L_* \end{pmatrix} \begin{pmatrix} \rho & r^T \\ 0 & L_*^T \end{pmatrix}$$

with $\alpha, \rho \in \mathbb{R}$, $a, r \in \mathbb{R}^{n-1}$, $A_* \in \mathbb{R}^{(n-1)\times(n-1)}$ and lower triangular $L_* \in \mathbb{R}^{(n-1)\times(n-1)}$ it follows that

$$\alpha = \rho^2; \quad a = r \cdot \rho; \quad A_* = r \cdot r^T + L_* \cdot L_*^T$$

and hence the factorization step

$$\begin{aligned} \rho &:= \sqrt{\alpha} \qquad (A \text{ is s.p.d.} \Leftrightarrow \alpha > 0) \\ r &:= \frac{a}{\rho} \\ L_* \cdot L_*^T &:= A_* - r \cdot r^T \end{aligned}$$

to be performed recursively.

$A = L \cdot L^T$: In-Place Factorization

$$
\begin{aligned}
&\text{for } i = 0, \ldots, n-1: \\
&\qquad A_{i,i} := \sqrt{A_{i,i}} \\
&\qquad \text{if } i < n-1: \\
&\qquad\qquad A_{i+1\ldots n,i} := \frac{A_{i+1\ldots n,i}}{A_{i,i}} \\
&\qquad\qquad A_{i+1\ldots n,i+1\ldots n} := A_{i+1\ldots n,i+1\ldots n} - A_{i+1\ldots n,i} \cdot A_{i+1\ldots n,i}^T
\end{aligned}
$$

or, more compactly,

$$
\begin{aligned}
&\text{for } i = 0, \ldots, n-1: \\
&\qquad \alpha_i := \sqrt{\alpha_i} \\
&\qquad \text{if } i < n-1: \\
&\qquad\qquad a_i := \frac{a_i}{\alpha_i} \\
&\qquad\qquad A_i := A_i - a_i \cdot a_i^T \ .
\end{aligned}
$$

$A = L \cdot L^T$: Implementation

```
1   template <typename T>
2   Eigen::matrix_t<T> LLT( Eigen::matrix_t<T> A) {
3     int n=A.rows();
4     for (int i=0; i<n; i++) {
5       A(i,i)=sqrt(A(i,i));
6       if (i<n-1) {
7         A.col(i).tail(n-i-1)/=A(i,i);
8         A.block(i+1,i+1,n-i-1,n-i-1)-=A.col(i).tail(n-i-1)*A.
            col(i).tail(n-i-1).transpose();
9       }
10    }
11    return Eigen::TriangularView<Eigen::matrix_t<T>,Eigen::
        Lower>(A);
12  }
```

or, preferrably,

```
1   template<typename T>
2   Eigen::matrix_t<T> LLT_Eigen(Eigen::matrix_t<T> A) {
3     return A.llt().matrixL();
4   }
```

Tangent $A = L \cdot L^T$

$$
\begin{aligned}
&\text{for } i = 0, \ldots, n-1 : \\
&\qquad \alpha_i^{(1)} := \frac{\alpha_i^{(1)}}{2 \cdot \sqrt{\alpha_i}} \\
&\qquad \alpha_i := \sqrt{\alpha_i} \\
&\qquad \text{if } i < n-1 : \\
&\qquad\qquad a_i^{(1)} := \frac{a_i^{(1)}}{\alpha_i} - \frac{a_i \cdot \alpha_i^{(1)}}{\alpha_i^2} \\
&\qquad\qquad a_i := \frac{a_i}{\alpha_i} \\
&\qquad\qquad A_i^{(1)} := A_i^{(1)} - a_i^{(1)} \cdot a_i^T - a_i \cdot a_i^{(1)^T} \\
&\qquad\qquad A_i := A_i - a_i \cdot a_i^T \ .
\end{aligned}
$$

... it does not get (much) better.

Symbolic Tangent of $A = L \cdot L^T$

From

$$A^{(1)} = L^{(1)} \cdot L^T + L \cdot {L^{(1)}}^T, \quad \text{where } A^{(1)} = {A^{(1)}}^T$$

it follows that

$$L^{-1} \cdot A^{(1)} \cdot L^{-T} = L^{-1} \cdot L^{(1)} + {L^{(1)}}^T \cdot L^{-T} = L^{-1} \cdot L^{(1)} + (L^{-1} \cdot L^{(1)})^T$$

yielding

$$\Phi(L^{-1} \cdot A^{(1)} \cdot L^{-T}) = L^{-1} \cdot L^{(1)}$$

and hence

$$L^{(1)} = L \cdot \Phi(L^{-1} \cdot A^{(1)} \cdot L^{-T})$$

where

$$\Phi(A) = \begin{cases} a_{i,j} & i > j \\ \frac{a_{i,i}}{2} & i = j \\ 0 & \text{otherwise.} \end{cases}$$

I. Murray: *Differentiation of the Cholesky decomposition.* arXiv:1602.07527 [stat.CO], 2016.

Symbolic Tangent of $A = L \cdot L^T$

Note that

$$\Phi(A) = \frac{1}{2} \cdot \sum_{i=1}^{n} E_i \cdot A \cdot E_i + \sum_{i=2}^{n} \sum_{j=1}^{i-1} E_i \cdot A \cdot E_j$$

where

$$\mathbb{R}^{n \times n} \ni E_i = (e^i_{j,k}) \equiv \begin{cases} 1 & i = j = k \\ 0 & \text{otherwise.} \end{cases}$$

Consequently,

$$\begin{aligned} L^{(1)} &= L \cdot \Phi(L^{-1} \cdot A^{(1)} \cdot L^{-T}) \\ &= L \cdot \left(\frac{1}{2} \cdot \sum_{i=1}^{n} E_i \cdot (L^{-1} \cdot A^{(1)} \cdot L^{-T}) \cdot E_i \right. \\ &\quad \left. + \sum_{i=2}^{n} \sum_{j=1}^{i-1} E_i \cdot (L^{-1} \cdot A^{(1)} \cdot L^{-T}) \cdot E_j \right) \end{aligned}$$

Adjoint $A = L \cdot L^T$

In-place factorization implies to-be-recorded (tbr) status of overwritten variables, that is,

$$
\begin{aligned}
&\text{for } i = 0, \ldots, n-1 : \\
&\qquad \alpha_i := \sqrt{\alpha_i} \qquad \text{(tbr)} \\
&\qquad \text{if } i < n-1 : \\
&\qquad\qquad a_i := \frac{a_i}{\alpha_i} \qquad \text{(tbr)} \\
&\qquad\qquad A_i := A_i - a_i \cdot a_i^T \ .
\end{aligned}
$$

Note linear use of A_i and no aliasing with prior nonlinear uses.

Adjoint $A = L \cdot L^T$

for $i = 0, \ldots, n-1$:

  push(α_i)

  $\alpha_i := \sqrt{\alpha_i}$

  if $i < n-1$:

    push(a_i)

    $a_i := \dfrac{a_i}{\alpha_i}$

    $A_i := A_i - a_i \cdot a_i^T$

for $i = n-1, \ldots, 0$:

  if $i < n-1$:

    $a_{i(1)} := a_{i(1)} - A_{i(1)} \cdot a_i - A_{i(1)}^T \cdot a_i$

    pop(a_i)

    $\alpha_{i(1)} := \alpha_{i(1)} - a_{i(1)} \cdot \dfrac{a_i}{\alpha_i^2}$

    $a_{i(1)} := \dfrac{a_{i(1)}}{\alpha_i}$

  pop(α_i)

  $\alpha_{i(1)} := \dfrac{\alpha_{i(1)}}{2 \cdot \sqrt{\alpha_i}}$

Tangent and Adjoint $A = L \cdot L^T$: Implementation

Live:

- LLT_alg_tan from tangent code generation rules on LLT
- LLT_ffd_tan for validation
- LLT_dco_tan on LLT_Eigen for validation and comparison of run times
- LLT_alg_adj from adjoint code generation rules on LLT
- LLT_dco_adj on LLT_Eigen for validation and comparison of run times
- use of differential_invariant for validation

Linear Solvers: Outline

- consequence from AD of matrix products
- tangents and adjoints
- $A \cdot x = L \cdot U \cdot x$
  - tangents and adjoints
  - illustration
  - implementation
  - validation
- $A \cdot x = L \cdot L^T \cdot x$
- $A \cdot x = Q \cdot R \cdot x$
- error analysis

A Consequence from AD of Matrix Products

$$Y^{(1)} = A \cdot X^{(1)} \cdot B \;\Rightarrow\; X_{(1)} = B \cdot Y_{(1)} \cdot A \tag{10}$$

The tangent $Y^{(1)} = A \cdot X^{(1)} \cdot B$ of $Y := A \cdot X \cdot B$ for active $X \in \mathbb{R}^{n \times q}$, $Y \in \mathbb{R}^{m \times p}$ and passive $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{q \times p}$ implies the adjoint $X_{(1)} = B \cdot Y_{(1)} \cdot A$.

Proof

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

follows from the application of Eqns. (8) and (9) to $Y = Z \cdot B$ with passive B.

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

follows from application of Eqns. (8) and (9) to $Z = A \cdot X$ with passive A.

Substitution of $Z^{(1)}$ and $Z_{(1)}$ yields Eqn. (10). ■

Another Consequence

$$Y^{(1)} = \sum_{i=1}^{k} A_i \cdot X_i^{(1)} \cdot B_i \;\Rightarrow\; X_{i(1)} = B_i \cdot Y_{(1)} \cdot A_i,\; i = 1, \ldots, k \qquad (11)$$

The tangent $Y^{(1)} = \sum_{i=1}^{k} A_i \cdot X_i^{(1)} \cdot B_i$ of $Y = \sum_{i=1}^{k} A_i \cdot X_i \cdot B_i$ with active $X_i \in \mathbb{R}^{n_i \times q_i}$, $Y \in \mathbb{R}^{m \times p}$ and with passive $A_i \in \mathbb{R}^{m \times n_i}$, $B_i \in \mathbb{R}^{q_i \times p}$ implies the adjoint $X_{i(1)} = B_i \cdot Y_{(1)} \cdot A_i$ for $i = 1, \ldots, k$.

Proof

From

$$Y_i^{(1)} = A_i \cdot X_i^{(1)} \cdot B_i$$

follows with Eqn. (10)

$$X_{i(1)} = B_i \cdot Y_{i(1)} \cdot A_i$$

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

Moreover, $Y^{(1)} = \sum_{i=1}^{k} Y_i^{(1)}$ implies $Y_{i(1)} = Y_{(1)}$ for $i = 1, \ldots, k$ and hence Eqn. (11). ■

Linear Solvers: Tangents and Adjoints

Tangents of systems $A \cdot \mathbf{x} = \mathbf{b}$ of n linear equations with invertible $A \in \mathbb{R}^{n \times n}$ and right-hand side $\mathbf{b} \in \mathbb{R}^n$ are evaluated at the primal solution $\mathbf{x} := A^{-1} \cdot \mathbf{b} \in \mathbb{R}^n$ as

$$\mathbf{x}^{(1)} := A^{-1} \cdot (\mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{x}) \,. \tag{12}$$

Corresponding adjoints are are evaluated as

$$\begin{aligned} \mathbf{b}_{(1)} &:= \mathbf{x}_{(1)} \cdot A^{-1} \\ A_{(1)} &:= -\mathbf{x} \cdot \mathbf{b}_{(1)} \end{aligned} \tag{13}$$

Avoid explicit inversion for efficiency!

Tangent and Adjoint Linear Solvers: Proof

Differentiation of $A \cdot \mathbf{x} = \mathbf{b}$ wrt. A and $\mathbf{b}$ yields the tangent system

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

which immediately implies Eqn. (12).

From Eqn. (12) it follows that

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

with identity $I_n \in \mathbb{R}^{n \times n}$.

Eqn. (11) yields

$$\begin{aligned} \mathbf{b}_{(1)} &:= I_n \cdot \mathbf{x}_{(1)} \cdot A^{-1} \\ A_{(1)} &:= -\mathbf{x} \cdot \underbrace{\mathbf{x}_{(1)} \cdot A^{-1}}_{=\mathbf{b}_{(1)}} \end{aligned}$$

and hence Eqn. (13). ■

Linear Solvers: Differential Invariant

```
template<typename T>
void validate_invariant(
    Eigen::matrix_t<T> A_t, Eigen::matrix_t<T> A_a,
    Eigen::vector_t<T> b_t, Eigen::row_vector_t<T> b_a,
    Eigen::vector_t<T> x_t, Eigen::row_vector_t<T> x_a
) {
  T v=b_a*b_t;
  for (int i=0;i<A_a.rows();i++) v+=A_a.row(i)*A_t.col(i);
  cout << "differential invariant:" << endl;
  cout << v << "==" << x_a*x_t << '?' << endl;
}
```

Note serialized versions of $A^{(1)}$ and $A_{(1)}$.

Tangent Linear Solvers: E.g. $A \cdot x = L \cdot U \cdot x$

$$L \cdot U = A$$
$$L \cdot z = \mathbf{b}$$
$$U \cdot \mathbf{x} = z$$
$$L \cdot z^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot \mathbf{x}$$
$$U \cdot \mathbf{x}^{(1)} = z^{(1)}$$

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} 2 & 1 \\ 0 & 0.5 \end{pmatrix}}_{U} = \underbrace{\begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix}}_{A}$$

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} 1 \\ -0.5 \end{pmatrix}}_{z} = \underbrace{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}_{b}$$

$$\underbrace{\begin{pmatrix} 2 & 1 \\ 0 & 0.5 \end{pmatrix}}_{U} \cdot \underbrace{\begin{pmatrix} 1 \\ -1 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 1 \\ -0.5 \end{pmatrix}}_{z}$$

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} -1 \\ 2.5 \end{pmatrix}}_{z^{(1)}} = \underbrace{\begin{pmatrix} 0 \\ 0 \end{pmatrix}}_{b^{(1)}} - \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{A^{(1)}} \cdot \underbrace{\begin{pmatrix} 1 \\ -1 \end{pmatrix}}_{x}$$

$$\underbrace{\begin{pmatrix} 2 & 1 \\ 0 & 0.5 \end{pmatrix}}_{U} \cdot \underbrace{\begin{pmatrix} -3 \\ 5 \end{pmatrix}}_{x^{(1)}} = \underbrace{\begin{pmatrix} -1 \\ 2.5 \end{pmatrix}}_{z^{(1)}}$$

Tangent $A \cdot x = L \cdot U \cdot x$: Implementation

Live:

- `Axb_LU.cpp`: implementation with Eigen
  - primal: `A.lu().solve(b);`
  - algorithmic tangent by dco/c++
  - symbolic tangent
  - approximate tangent by forward finite differences
- `Axb_LU_steps.cpp`: symbolic tangent unrolled (as on previous slide)
- `Axb_LU_ex.cpp`: example from previous slide

Adjoint Linear Solvers: $A \cdot x = L \cdot U \cdot x$

$$L \cdot U = A$$
$$L \cdot z = \mathbf{b}$$
$$U \cdot \mathbf{x} = z$$
$$z_{(1)} \cdot U = \mathbf{x}_{(1)}$$
$$\mathbf{b}_{(1)} \cdot L = z_{(1)}$$
$$A_{(1)} := -\mathbf{x} \cdot \mathbf{b}_{(1)}$$

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} 2 & 1 \\ 0 & 0.5 \end{pmatrix}}_{U} = \underbrace{\begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix}}_{A}$$

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} 1 \\ -0.5 \end{pmatrix}}_{z} = \underbrace{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}_{b}$$

$$\underbrace{\begin{pmatrix} 2 & 1 \\ 0 & 0.5 \end{pmatrix}}_{U} \cdot \underbrace{\begin{pmatrix} 1 \\ -1 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 1 \\ -0.5 \end{pmatrix}}_{z}$$

$$\underbrace{\begin{pmatrix} 0.5 & 1 \end{pmatrix}}_{z_{(1)}} \cdot \underbrace{\begin{pmatrix} 2 & 1 \\ 0 & 0.5 \end{pmatrix}}_{U} = \underbrace{\begin{pmatrix} 1 & 1 \end{pmatrix}}_{x_{(1)}}$$

$$\underbrace{\begin{pmatrix} -1 & 1 \end{pmatrix}}_{b_{(1)}} \cdot \underbrace{\begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}}_{L} = \underbrace{\begin{pmatrix} 0.5 & 1 \end{pmatrix}}_{z_{(1)}}$$

$$\underbrace{\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}}_{A_{(1)}} := -\underbrace{\begin{pmatrix} 1 \\ -1 \end{pmatrix}}_{x} \cdot \underbrace{\begin{pmatrix} -1 & 1 \end{pmatrix}}_{b_{(1)}}$$

Adjoint $A \cdot x = L \cdot U \cdot x$: Implementation

Live:

- `Axb_LU.cpp`: implementation with Eigen
  - primal: `A.lu().solve(b);`
  - algorithmic adjoint by dco/c++
  - symbolic adjoint
  - differential invariant for validation
- `Axb_LU_steps.cpp`: symbolic adjoint unrolled (as on previous slide)
- `Axb_LU_ex.cpp`: example from previous slide

Nonlinear Solvers: Outline

- Tangents
  - Algorithm
  - Illustration
- Adjoints
  - Algorithm
  - Illustration
- Convex Objectives
  - Algorithms
  - Illustration

Tangents of Nonlinear Solvers

Let $r = R(x(p), p) = 0$ with $R : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ be continuously differentiable wrt. both x and p.

From

$$\frac{dR}{dp} = \frac{\partial R}{\partial p} + \frac{dR}{dx} \cdot \frac{dx}{dp} = 0$$

follows

$$\underbrace{\frac{dx}{dp}}_{\in \mathbb{R}^{n_x \times n_p}} = - \underbrace{\frac{dR}{dx}^{-1}}_{\in \mathbb{R}^{n_x \times n_x}} \cdot \underbrace{\frac{\partial R}{\partial p}}_{\in \mathbb{R}^{n_x \times n_p}}$$

and hence the tangent

$$x^{(1)} \equiv \frac{dx}{dp} \cdot p^{(1)} = \underbrace{-\frac{dR}{dx}^{-1} \cdot \frac{\partial R}{\partial p} \cdot p^{(1)}}_{\frac{dR}{dx} \cdot x^{(1)} = -\frac{\partial R}{\partial p} \cdot p^{(1)}} .$$

Tangents of Nonlinear Solvers: Algorithm

1. Solve the primal system $R(x(p), p) = 0$ to the required accuracy yielding an approximate root $x^* = S(R, x, p)$.
2. Compute the Jacobian $\frac{dR}{dx}$ of the residual wrt. the state, e.g. using dco/c++ in vector tangent mode (dco::gt1v<T,VS>::type with vector size VS).
3. Evaluate the tangent $z^{(1)} := \frac{\partial R}{\partial p} \cdot p^{(1)}$, e.g. using dco/c++ in tangent mode (dco::gt1s<T>::type).
4. Solve the system of linear equations $\frac{dR}{dx} \cdot x^{(1)} = -z^{(1)}$.

Algorithmic differentiation of the iterative primal solver in tangent mode can be avoided at the primal solution. Error analysis should be employed to account for the availability of an approximate primal solution.

Tangents of Nonlinear Solvers: Illustration

$$R = R(x,p) \equiv x^2 - p = 0$$

$$\xrightarrow{\frac{d}{dp} \cdot p^{(1)}}$$

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

$$S^{(1)}(R, p, p^{(1)}, x)$$

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

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

$$x^{*(1)} \approx \frac{p^{(1)}}{2 \cdot x^*} \approx \frac{p^{(1)}}{2 \cdot x} \ ?$$

Illustration: Primal Code

```
1  #include "dco.hpp"
2
3  template<typename T>
4  T R(T x, T p) { return x*x-p; }
5
6  template<typename T>
7  T dRdx(T x, T p) {
8    typename dco::gt1s<T>::type x_dco=x, p_dco=p;
9    dco::derivative(x_dco)=1;
10   return dco::derivative(R(x_dco,p_dco));
11 }
12
13 template<typename T>
14 T S(T x, T p, float eps) {
15   do x=x-R(x,p)/dRdx(x,p); while (fabs(R(x,p))>eps);
16   return x;
17 }
18
19 int main() {
20   double x=2, p=2;
21   cout << "x*=" << S(x,p,1e-7) << endl;
22   return 0;
23 }
```

Illustration: Algorithmic Tangent Code

```
1  int main() {
2    dco::gt1s<double>::type x=2, p=2;
3    dco::derivative(p)=1;
4    x=S(x,p,1e-7);
5    cout << "x*=" << dco::value(x) << endl;
6    cout << "dx/dp(x*)=" << dco::derivative(x) << endl;
7    return 0;
8  }
```

Illustration: Symbolic Tangent

```
1   template<typename T>
2   T dRdp_pt(T x, T p, T pt) {
3     typename dco::gt1s<T>::type x_dco=x, p_dco=p;
4     dco::derivative(p_dco)=pt;
5     return dco::derivative(R(x_dco,p_dco));
6   }
7
8   int main() {
9     double x=2, p=2, pt=1;
10    x=S(x,p,1e-7);
11    cout << "x*=" << x << endl;
12    cout << "dx/dp(x*)=" << -dRdp_pt(x,p,pt)/dRdx(x,p) << endl
        ;
13    return 0;
14  }
```

Adjoints of Nonlinear Solvers

Let $r = R(x(p), p) = 0$ with $R : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ be continuously differentiable wrt. both x and p.

From

$$\frac{dR}{dp} = \frac{\partial R}{\partial p} + \frac{dR}{dx} \cdot \frac{dx}{dp} = 0$$

follows

$$\underbrace{\frac{dx}{dp}}_{\in \mathbb{R}^{n_x \times n_p}} = - \underbrace{\frac{dR}{dx}^{-1}}_{\in \mathbb{R}^{n_x \times n_x}} \cdot \underbrace{\frac{\partial R}{\partial p}}_{\in \mathbb{R}^{n_x \times n_p}}$$

and hence the adjoint

$$p_{(1)} \equiv x_{(1)} \cdot \frac{dx}{dp} = \underbrace{-x_{(1)} \cdot \frac{dR}{dx}^{-1}}_{z_{(1)} \cdot \frac{dR}{dx} = -x_{(1)}} \cdot \frac{\partial R}{\partial p} \, .$$

Adjoints of Nonlinear Solvers: Algorithm

1. Solve the primal system $R(x(p), p) = 0$ to the required accuracy yielding an approximate root $x^* = S(R, x, p)$.

2. Compute the Jacobian $\frac{dR}{dx}$ of the residual wrt. the state, e.g. using dco/c++ in vector tangent mode.

3. Solve the system of linear equations $z_{(1)} \cdot \frac{dR}{dx} = -x_{(1)}$ for the given adjoint of the primal solution $x_{(1)}$ yielding $z_{(1)} \in \mathbb{R}^{1 \times n_x}$.

4. Evaluate the adjoint $p_{(1)} = z_{(1)} \cdot \frac{\partial R}{\partial p}$, e.g. using dco/c++ in adjoint mode (dco::ga1s<T>::type).

Algorithmic differentiation of the iterative primal solver in adjoint mode can be avoided at the primal solution. Error analysis should be employed to account for the availability of an approximate primal solution.

Adjoints of Nonlinear Solvers: Illustration

$$R = R(x,p) \equiv x^2 - p = 0$$

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

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

$$(x^*, Tape) = \overrightarrow{S}(R, p, x)$$

$$\tilde{S}_{(1)}(x^*, x_{(1)})$$

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

$$\overleftarrow{S}(Tape, x_{(1)})$$

$$p_{(1)} \approx \frac{x_{(1)}}{2 \cdot x^*} \approx \frac{x_{(1)}}{2 \cdot x} \ ?$$

Illustration: Algorithmic Adjoint

```
int main() {
  using dco_mode=dco::ga1s<double>;
  using dco_type=dco_mode::type;
  dco_type x=2, p=2;
  dco::smart_tape_ptr_t<dco_mode> tape;
  tape->register_variable(p);
  x=S(x,p,1e-7);
  cout << "x*=" << dco::value(x) << endl;
  dco::derivative(x)=1;
  tape->interpret_adjoint();
  cout << "dx/dp(x*)=" << dco::derivative(p) << endl;
  return 0;
}
```

Illustration: Symbolic Adjoint

```
1  template<typename T>
2  T ra_dRdp(T x, T p, T ra) {
3    using dco_mode=typename dco::ga1s<T>;
4    using dco_type=typename dco_mode::type;
5    dco_type x_dco=x, p_dco=p;
6    dco::smart_tape_ptr_t<dco_mode> tape;
7    tape->register_variable(p_dco);
8    x_dco=R(x_dco,p_dco);
9    dco::derivative(x_dco)=ra;
10   tape->interpret_adjoint();
11   return dco::derivative(p_dco);
12 }
13
14 int main() {
15   double x=2, p=2, xa=1;
16   x=S(x,p,1e-7);
17   cout << "x*=" << x << endl;
18   cout << "dx/dp(x*)=" << ra_dRdp(x,p,-xa/dRdx(x,p)) << endl
       ;
19   return 0;
20 }
```

Tangents and Adjoints of Convex Objectives

Let $x^* = x^*(p) = \min_x f(x(p), p)$ for twice continuously differentiable $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}$ wrt. both x and p yielding

$$\frac{df}{dx}(x^*) = 0 \quad \text{and} \quad \frac{d^2 f}{dx^2}(x^*) > 0 \,.$$

From

$$\frac{d^2 f}{dxdp}(x^*) = \frac{\partial \frac{df}{dx}}{\partial p}(x^*) + \frac{d^2 f}{dx^2}(x^*) \cdot \frac{dx}{dp}(x^*) = 0$$

follows

$$\underbrace{\frac{dx}{dp}}_{\in \mathbb{R}^{n_x \times n_p}} = -\underbrace{\frac{d^2 f}{dx^2}^{-1}}_{\in \mathbb{R}^{n_x \times n_x}} \cdot \underbrace{\frac{\partial \frac{df}{dx}}{\partial p}}_{\in \mathbb{R}^{n_x \times n_p}}$$

...

Tangents and Adjoints of Convex Objectives

... and hence the tangent

$$x^{(1)} \equiv \frac{dx}{dp} \cdot p^{(1)} = \frac{d^2 f}{dx^2}^{-1} \cdot \underbrace{\frac{\partial \frac{df}{dx}}{\partial p} \cdot p^{(1)}}_{z^{(1)}}$$

and the adjoint

$$p_{(1)} \equiv x_{(1)} \cdot \frac{dx}{dp} = \underbrace{-x_{(1)} \cdot \frac{d^2 f}{dx^2}^{-1}}_{z_{(1)} \cdot \frac{d^2 f}{dx^2} = -x_{(1)}} \cdot \frac{\partial \frac{df}{dx}}{\partial p} \ .$$

Tangents of Convex Objectives: Algorithm

1. Compute the gradient $R = R(x, p) = \frac{df}{dx}(x, p)$ of the objective wrt. the state, e.g. using dco/c++ in adjoint mode.
2. Solve the primal system $R(x(p), p) = 0$ to the required accuracy yielding an approximate root $x^* = S(R, x, p)$.
3. Compute the Jacobian $\frac{dR}{dx}$ of the residual wrt. the state, e.g. using dco/c++ in vector tangent mode.
4. Evaluate the tangent $z^{(1)} := \frac{\partial R}{\partial p} \cdot p^{(1)}$, e.g. using dco/c++ in tangent mode.
5. Solve the system of linear equations $\frac{dR}{dx} \cdot x^{(1)} = -z^{(1)}$.

Algorithmic differentiation of the iterative primal solver in tangent mode can be avoided at the primal solution. Error analysis should be employed to account for the availability of an approximate primal solution.

Tangents of Convex Objectives: Illustration

The objective function $f(x,p) = e^x - p \cdot x$ has a minimum at

$$\frac{df}{dx}(x,p) = e^x - p = 0 \quad \left(\frac{d^2 f}{dx^2}(x,p) = e^x > 0 \ \forall x \in \mathbb{R}\right) .$$

$$\begin{tikzcd}
\boxed{R = R(x,p) \equiv e^x - p = 0} \arrow[r, "\frac{d}{dp} \cdot p^{(1)}"] \arrow[d, "S^{(1)}(R,p,x)"'] \arrow[dr] & \boxed{\left(e^x \cdot \frac{dx}{dp} - 1\right) \cdot p^{(1)} = e^x \cdot \underbrace{\frac{dx}{dp} \cdot p^{(1)}}_{x^{(1)}} - p^{(1)} = 0} \arrow[d, "\tilde{S}^{(1)}(x^*, p^{(1)})"] \\
\boxed{x^* \approx \log p} & \boxed{x^{(1)} \approx \frac{p^{(1)}}{e^{x^*}} \approx \frac{p^{(1)}}{p} \ ?}
\end{tikzcd}$$

Adjoints of Convex Objectives: Algorithm

1. Compute the gradient $R = R(x, p) = \frac{df}{dx}(x, p)$ of the objective wrt. the state, e.g. using dco/c++ in adjoint mode.
2. Solve the primal system $R(x(p), p) = 0$ to the required accuracy yielding an approximate root $x^* = S(R, x, p)$.
3. Compute the Jacobian $\frac{dR}{dx}$ of the residual wrt. the state, e.g. using dco/c++ in vector tangent mode.
4. Solve the system of linear equations $z_{(1)} \cdot \frac{dR}{dx} = -x_{(1)}$ for the given adjoint of the primal solution $x_{(1)}$ yielding $z_{(1)} \in \mathbb{R}^{1 \times n_x}$.
5. Evaluate the adjoint $p_{(1)} = z_{(1)} \cdot \frac{\partial R}{\partial p}$, e.g. using dco/c++ in adjoint mode (dco::ga1s<T>::type).

Algorithmic differentiation of the iterative primal solver in adjoint mode can be avoided at the primal solution. Error analysis should be employed to account for the availability of an approximate primal solution.

Adjoints of Convex Objectives: Illustration

$$R = R(x,p) \equiv e^x - p = 0 \xrightarrow{x_{(1)} \cdot \frac{d}{dp}} x_{(1)} \cdot \left(e^x \cdot \frac{dx}{dp} - 1 \right) = e^x \cdot \underbrace{x_{(1)} \cdot \frac{dx}{dp}}_{p_{(1)}} - x_{(1)} = 0$$

$$R = R(x,p) \equiv e^x - p = 0 \xrightarrow{(x^*, Tape) = \overrightarrow{S}(R,p,x)} x^* \approx \log p$$

$$x_{(1)} \cdot \left(e^x \cdot \frac{dx}{dp} - 1 \right) = e^x \cdot \underbrace{x_{(1)} \cdot \frac{dx}{dp}}_{p_{(1)}} - x_{(1)} = 0 \xrightarrow{\tilde{S}_{(1)}(x, x_{(1)})} p_{(1)} \approx \frac{x_{(1)}}{e^{x^*}} \approx \frac{x_{(1)}}{p} \ ?$$

$$x^* \approx \log p \xrightarrow{\overleftarrow{S}(Tape, x_{(1)})} p_{(1)} \approx \frac{x_{(1)}}{e^{x^*}} \approx \frac{x_{(1)}}{p} \ ?$$

Illustration: Source

```
1  #include "dco.hpp"
2
3  template<typename T>
4  T f(T x, T p) { return exp(x)-p*x; }
5
6  template<typename T>
7  T R(T x, T p) {
8    using dco_mode=typename dco::ga1s<T>;
9    using dco_type=typename dco_mode::type;
10   dco_type x_dco=x, p_dco=p, y_dco;
11   dco::smart_tape_ptr_t<dco_mode> tape;
12   tape->register_variable(x_dco);
13   y_dco=f(x_dco,p_dco);
14   dco::derivative(y_dco)=1;
15   tape->interpret_adjoint();
16   return dco::derivative(x_dco);
17 }
```

... tangent and adjoint codes remain unchanged.

Nonlinear Solvers: Is there an Issue?

From the differential invariant $p_{(1)} \cdot p^{(1)} = x_{(1)} \cdot x^{(1)}$ it follows that

$$(-x_{(1)} \cdot R_x^{-1} \cdot R_p) \cdot p^{(1)} \quad \stackrel{?}{=} \quad \underbrace{(x_{(1)} \cdot S_p)}_{\text{not available!}} \cdot p^{(1)} = x_{(1)} \cdot (S_p \cdot p^{(1)}) \, .$$

U.N.: *Differential Invariants.* arXiv:2101.03334, 2021.

Hence, for validation

- pick $x_{(1)}$ (randomly)
- compute $\bar{p}_{(1)} \equiv -x_{(1)} \cdot R_x^{-1} \cdot R_p$ (symbolic adjoint)
- pick $p^{(1)}$ (randomly)
- compute / approximate $\dot{x}^{(1)} \equiv S_p \cdot p^{(1)}$ (algorithmic / approximate tangent)
- compare $\bar{p}_{(1)} \cdot p^{(1)} \stackrel{?}{=} x_{(1)} \cdot \dot{x}^{(1)}$.

If there is an issue ... e.g., first-order error analysis

Convex Objectives: Is there an Issue?

From the differential invariant $p_{(1)} \cdot p^{(1)} = x_{(1)} \cdot x^{(1)}$ it follows that

$$(-x_{(1)} \cdot f_{xx}^{-1} \cdot f_{xp}) \cdot p^{(1)} \quad \stackrel{?}{=} \quad \underbrace{(x_{(1)} \cdot S_p)}_{\text{not available!}} \cdot p^{(1)} = x_{(1)} \cdot (S_p \cdot p^{(1)}) \; .$$

Hence, for validation

- pick $x_{(1)}$ (randomly)
- compute $\bar{p}_{(1)} \equiv -x_{(1)} \cdot f_{xx}^{-1} \cdot f_{xp}$
- pick $p^{(1)}$ (randomly)
- compute / approximate $\dot{x}^{(1)} \equiv S_p \cdot p^{(1)}$
- compare $\bar{p}_{(1)} \cdot p^{(1)} \stackrel{?}{=} x_{(1)} \cdot \dot{x}^{(1)}$.

If there is an issue ... e.g., first-order error analysis

Outline

Conclusion

What's next?

- Error analysis for symbolic tangents and adjoints
- AD by hand (selectively)
- external function interface
- second-(and higher-)order tangents and adjoints
- checkpointing
  - evolutions
  - call trees
- AD mission planning
- dco/c++ v4.0
  - easy-to-use interfaces
  - code generation
  - explicit vectorization
- NAG AD Library
- interval adjoints