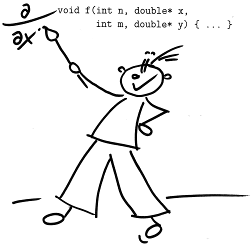
The Art of Differentiating Computer Programs

Uwe Naumann



LuFG Informatik 12
Software and **T**ools for **C**omputational **E**ngineering
RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de

Contents

AD Informally

AD Formally

AD by Overloading

Second-(Higher-)Order AD Formally

Advanced AD (with dco/c++)
- External Function Interface
- Adjoint Ensembles
- User-Defined Adjoints

AD Projects

Who knows how to differentiate ...

- y=x*x;

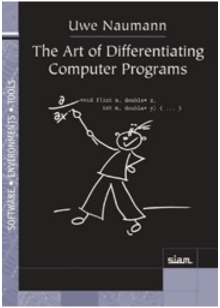
- ```
  void f(const double x, double& y) {
    y=x*x;
  }
  ```

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

AD ...

- ... assumes differentiability of a given implementation of a multivariate vector function $y = f(x) : \mathbb{R}^n \to \mathbb{R}^m$ (e.g., existence of Jacobian $\nabla f(x) \in \mathbb{R}^{m \times n}$ and Hessian $\nabla^2 f(x) \in \mathbb{R}^{m \times n \times n}$).
- ... transforms the given program into programs for computing arbitrary projections of first- and higher-order derivative tensors (e.g., $\nabla f(x) \cdot \dot{x}$ and $\nabla f(x)^T \cdot \bar{y}$).
- ... can be implemented by source-to-source transformation and/or operator overloading combined with generic (meta-)programming techniques.
- ... can be supported by corresponding software tools in order to (partially) automate its application.
- ... has been applied successfully to a large number of real-world codes from Computational Science, Engineering, and Finance; see, for example, proceedings of the six AD conferences.

While the basic idea is reasonably straight forward, the actual implementation (both user and developer perspective) yields several challenges.



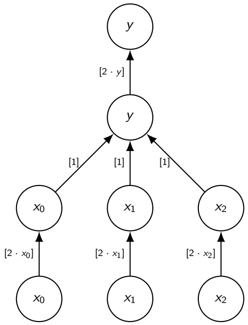
U. Naumann:
The Art of Differentiating Computer Programs.
An Introduction to Algorithmic Differentiation.
Number 24 in Software , Environments, and Tools, SIAM, 2012.

naumann@stce.rwth-aachen.de

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

where

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



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

$$\underbrace{\nabla f(x)}_{\in \mathbb{R}^n} \equiv \frac{\partial y}{\partial x} = \underbrace{\nabla f_3(y)}_{\in \mathbb{R}} \cdot \underbrace{\nabla f_2(x)}_{\in \mathbb{R}^{1\times n}} \cdot \underbrace{\nabla f_1(x)}_{\in \mathbb{R}^{n\times n}} = (2 \cdot y) \cdot (1 \ldots 1) \cdot \begin{pmatrix} 2 \cdot x_0 & & \\ & \vdots & \\ & & 2 \cdot x_{n-1} \end{pmatrix}$$

A (first-order forward) finite difference quotient

$$\frac{f(x + h \cdot \dot{x}) - f(x)}{h}$$

approximates the (first) directional (total) derivative

$$\begin{aligned}\dot{y} &= \nabla f(x) \cdot \dot{x} = \nabla f_3(y) \cdot \nabla f_2(x) \cdot \nabla f_1(x) \cdot \dot{x} \\ &= (2 \cdot y) \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \cdot x_0 & & \\ & \vdots & \\ & & 2 \cdot x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} \dot{x}_0 \\ \vdots \\ \dot{x}_{n-1} \end{pmatrix}.\end{aligned}$$

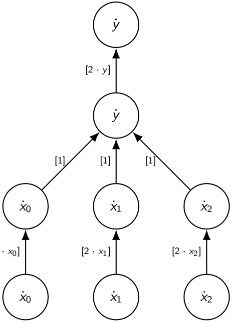
The whole gradient $\nabla f(x)$ can be approximated by letting $\dot{x}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ at a computational cost of $O(n) \cdot \text{Cost}(f)$.

Can we avoid truncation?

YES, WE CAN!

A (first-order) tangent code computes the same directional derivative with machine accuracy.

$$\dot{y} = (2 \cdot \underset{[=\sum_{i=0}^{n-1} x_i^2]}{y}) \cdot (1 \quad \dots \quad 1) \cdot \begin{pmatrix} 2 \cdot x_0 & & \\ & \vdots & \\ & & 2 \cdot x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} \dot{x}_0 \\ \vdots \\ \dot{x}_{n-1} \end{pmatrix}$$



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

1. duplicate active data segment

```
double v, t1_v;
```

2. augment with assignment-level tangent code

```
t1_z=cos(x*z)*(x*t1_z+t1_x*z);
z=sin(x*z);
```

3. leave intraprocedural flow of control unchanged

```
for (int i=0;i<n;++i) {
  t1_y[i]-=sin(x[i])*t1_x[i];
  y[i]+=cos(x[i]);
}
```

4. replace subprogram calls with tangent versions

```
f(n,x,m,y,t1_x,t1_y);
```

Can we reduce the computational cost?

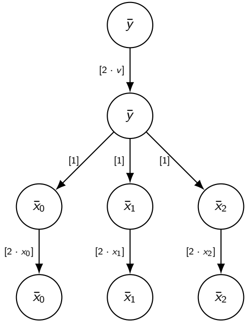
YES, WE CAN!

$$y = x_0 \cdot x_1$$

Alternatively, the transposed Jacobian can be computed as

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

$$= \begin{pmatrix} 2 \cdot x_0 & & \\ & \vdots & \\ & & 2 \cdot x_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \cdot (2 \cdot \underset{[=\sum_{i=0}^{n-1} x_i^2]}{y}) \cdot \bar{y}$$



The whole gradient $\nabla f(x)$ can be approximated by setting $\bar{y} = 1$ at the computational cost of $O(1) \cdot \text{Cost}(f)$.

1. duplicate active data segment
2. store lost (due to overwriting or leaving scope) required values in forward section

```
doubles.push(z);
z=sin(x*z);
```

3. assignment-level adjoint code in reverse section ...

```
z=doubles.top(); doubles.pop();
a1_x+=x*cos(x*z)*a1_z
a1_z=z*cos(x*z)*a1_z
```

4. reverse intraprocedural flow of control by (combinations of)
   4.1 enumeration of basic blocks
   4.2 counting loop iterations and enumeration of branches
   4.3 explicit reversal of **for**-loops

Summary: Tangents and Adjoints by AD
$y = f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ (w.l.o.g. $m = 1$)

- **Tangent(-Linear) Code** $f^{(1)}$ **(forward mode)**

$$y^{(1)} = \underbrace{\nabla f(\mathbf{x})}_{\in \mathbb{R}^n} \cdot \underbrace{\mathbf{x}^{(1)}}_{\in \mathbb{R}^n}$$

$$\Rightarrow \quad \nabla f \text{ at } O(n) \cdot Cost(f)$$

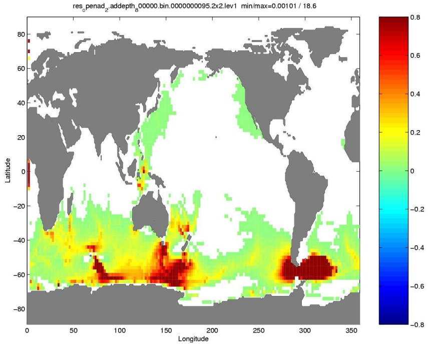
Note: Approximation by finite differences.

- **Adjoint Code** $f_{(1)}$ **(reverse mode)**

$$\mathbf{x}_{(1)} = \underbrace{y_{(1)}}_{\in \mathbb{R}} \cdot \nabla f(\mathbf{x})$$

$$\Rightarrow \quad \nabla f \text{ at } O(1) \cdot Cost(f)$$

(©J.Lotz)



MITgcm, (EAPS, MIT)

in collaboration with ANL, MIT, Rice, UColorado

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

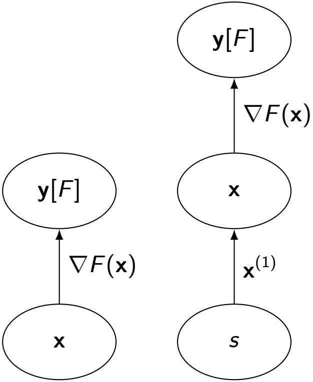
Plot: A tangent computation / finite difference approximation for 64,800 grid points at 1 min each would keep us waiting for a month and a half ... :-(((We did it in less than 10 minutes thanks to discrete adjoints computed by a differentiated version of the MITgcm :-)

Can we do it again (and again, and ...)?

YES WE CAN!

Tangent Model

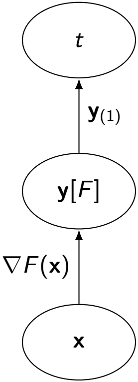
Graphically



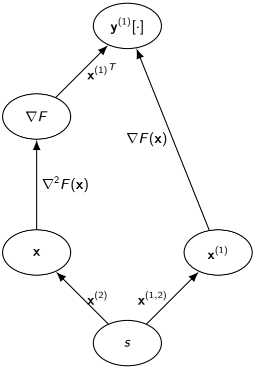
$$\mathbf{x}^{(1)} \equiv \frac{\partial \mathbf{x}}{\partial s} \quad \Rightarrow \quad \mathbf{y}^{(1)} \equiv \frac{\partial \mathbf{y}}{\partial s} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$$

Adjoint Model

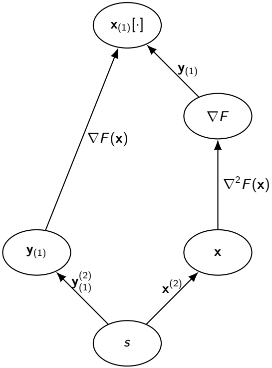
Graphically



$$\mathbf{y}_{(1)} \equiv \frac{\partial t}{\partial \mathbf{y}} \quad \Rightarrow \quad \mathbf{x}_{(1)} \equiv \frac{\partial t}{\partial \mathbf{x}} = \frac{\partial t}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{y}_{(1)} \cdot \nabla F(\mathbf{x})$$



Tangent Extension of the Linearized DAG of the Tangent Model
$\mathbf{y}^{(1)} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ of $\mathbf{y} = F(\mathbf{x})$



Tangent Extension of the Linearized DAG of the Adjoint Model
$\mathbf{x}_{(1)} = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$ of $\mathbf{y} = F(\mathbf{x})$

Summary: Second-(and Higher-)Order AD
$y = f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$

- **Second-Order Tangent Code** (forward-over-forward AD)

$$y^{(1,2)} = \underset{\in \mathbf{R}^n}{\mathbf{x}^{(1)}}{}^T \cdot \underset{\in \mathbf{R}^{n \times n}}{\nabla^2 f(\mathbf{x})} \cdot \underset{\in \mathbf{R}^n}{\mathbf{x}^{(2)}}$$

$$\Rightarrow \quad \nabla^2 f \text{ at } O(n^2) \cdot Cost(f)$$

- **Second-Order Adjoint Code** (forward-over-reverse AD)

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

$$\Rightarrow \quad \nabla^2 f \cdot \mathbf{x}^{(2)} \text{ at } O(1) \cdot Cost(f) \text{ resp. } \nabla^2 f \text{ at } O(n) \cdot Cost(f)$$

- Higher-order tangent (ToTo...ToT) and adjoint (ToTo...ToA) models are derived recursively

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

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

   where $q = p + m$

2. elemental functions φ_j posess jointly continuous partial derivatives with respect to their arguments $(v_k)_{k \prec j}$ at alle points of interest

Let

$$\dot{v}_j \equiv \frac{\partial \varphi_j}{\partial s}\left((v_k)_{k \prec j}\right)$$

for some $s \in \mathbb{R}$ and $j = 0, \ldots, n+p+m-1$.

For given $\dot{x}_i$, $i = 0, \ldots, n-1$, tangent mode AD computes directional derivatives as follows:

- for $i = 0, \ldots, n-1:$ $\quad \dot{v}_i = \dot{x}_i; \; v_i = x_i$
- for $j = n, \ldots, n+q-1:$

$$\dot{v}_j = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \cdot \dot{v}_i$$
$$v_j = \varphi_j\left((v_k)_{k \prec j}\right)$$

- for $k = 0, \ldots, m-1:$ $\quad \dot{y}_k = \dot{v}_{n+p+k}; \; y_k = v_{n+p+k}$

Let $F : \mathbb{R}^2 \to \mathbb{R}^2$, $\mathbf{y} = F(\mathbf{x})$ be implemented as

$$\begin{aligned} x_0 &= x_0 \cdot x_1 \\ y_0 &= \sin(x_0) \\ y_1 &= x_0 - x_1 \end{aligned}$$

yielding the SAC

$$\begin{aligned} v_0 &= x_0; \;\; v_1 = x_1 \\ v_2 &= v_0 \cdot v_1 \\ y_0 = v_3 &= \sin(v_2) \\ y_1 = v_4 &= v_2 - v_1 . \end{aligned}$$

Consider the extended function $\mathcal{F} : \mathbb{R}^5 \rightarrow \mathbb{R}^5$ defined as

$$\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ y_0 \\ y_1 \end{pmatrix} = \mathcal{F}\left(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}\right) = \Phi_3(\Phi_2(\Phi_1(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix})$$

where r_i, $i = 2, 3, 4$ are random and

$$\Phi_1(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}) = (\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ r_3 \\ r_4 \end{pmatrix}); \quad \Phi_2(\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ r_3 \\ r_4 \end{pmatrix}) = (\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ v_3 \\ r_4 \end{pmatrix}); \quad \Phi_3(\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ v_3 \\ r_4 \end{pmatrix}) = (\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix})$$

for

$$v_2 = v_0 \cdot v_1; \quad v_3 = \sin(v_2); \quad v_4 = v_2 - v_1.$$

If F is continuously differentiable at $\mathbf{x}$, then so is $\mathcal{F}$ and, hence,

$$\nabla\mathcal{F} = \nabla\mathcal{F}(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}) = \nabla\Phi_3(\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ v_3 \\ r_4 \end{pmatrix}) \cdot \nabla\Phi_2(\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ r_3 \\ r_4 \end{pmatrix}) \cdot \nabla\Phi_1(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix})$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(v_2) & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ x_1 & x_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

since

$$v_4 = v_2 - v_1; \qquad v_3 = \sin(v_2); \qquad v_2 = x_0 \cdot x_1.$$

Moreover, ...

$$\begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ \dot{v}_2\\ \dot{y}_0\\ \dot{y}_1\end{pmatrix} = \nabla\mathcal{F}\cdot\begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ \dot{r}_2\\ \dot{r}_3\\ \dot{r}_4\end{pmatrix}$$

$$= \begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&-1&1&0&0\end{pmatrix}\cdot\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&\cos(v_2)&0&0\\0&0&0&0&1\end{pmatrix}\cdot\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\x_1&x_0&0&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}\cdot\begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ \dot{r}_2\\ \dot{r}_3\\ \dot{r}_4\end{pmatrix}$$

$$= \begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&-1&1&0&0\end{pmatrix}\cdot\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&\cos(v_2)&0&0\\0&0&0&0&1\end{pmatrix}\cdot\begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ x_1\cdot\dot{x}_0+x_0\cdot\dot{x}_1\\ \dot{r}_3\\ \dot{r}_4\end{pmatrix}$$

$$= \begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&-1&1&0&0\end{pmatrix}\cdot\begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ \dot{v}_2\\ \cos v_2\cdot\dot{v}_2\\ \dot{r}_4\end{pmatrix} = \begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ \dot{v}_2\\ \dot{v}_3\\ \dot{v}_2-\dot{x}_1\end{pmatrix} = \begin{pmatrix}\dot{x}_0\\ \dot{x}_1\\ \dot{v}_2\\ \dot{v}_3\\ \dot{v}_4\end{pmatrix}$$

"Q.E.D"

Let

$$\bar{v}_j \equiv \frac{\partial t}{\partial \varphi_j}\left((v_k)_{k \prec j}\right)$$

for some $t \in \mathbb{R}$ and $j = 0, \ldots, n+p+m-1$.

For given $\dot{y}_i$, $i = 0, \ldots, m-1$, adjoint mode AD computes adjoint derivatives as follows:

- for $i = 0, \ldots, n-1$: $\quad v_i = x_i$
- for $j = n, \ldots, n+q-1$: $\quad v_j = \varphi_j\left((v_k)_{k \prec j}\right)$
- for $k = 0, \ldots, m-1$: $\quad y_k = v_{n+p+k}$
- for $k = 0, \ldots, m-1$: $\quad \bar{v}_{n+p+k} = \bar{y}_k$
- for $i = 0, \ldots, n-1$: $\quad \bar{v}_i = \bar{x}_i$
- for $j = n+q-1, \ldots, n$:

$$\forall i \prec j : \bar{v}_i = \bar{v}_i + \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \cdot \bar{v}_j$$

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

- for $i = 0, \ldots, n-1$: $\quad \bar{x}_i = \bar{v}_i$

Standard matrix arithmetic arithmetic yields

$$\nabla \mathcal{F}^T = \nabla \mathcal{F}\left(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}\right)^T = \nabla \Phi_1\left(\begin{pmatrix} x_0 \\ x_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}\right)^T \cdot \nabla \Phi_2\left(\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ r_3 \\ r_4 \end{pmatrix}\right)^T \cdot \nabla \Phi_3\left(\begin{pmatrix} x_0 \\ x_1 \\ v_2 \\ v_3 \\ r_4 \end{pmatrix}\right)^T$$

$$= \begin{pmatrix} 1 & 0 & x_1 & 0 & 0 \\ 0 & 1 & x_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & \cos(v_2) & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Moreover, ...

$$\begin{pmatrix}\bar{x}_0\\ \bar{x}_1\\ \bar{v}_2\\ \bar{y}_0\\ \bar{y}_1\end{pmatrix} = \nabla\mathcal{F}^T \cdot \begin{pmatrix}\bar{x}_0\\ \bar{x}_1\\ \bar{v}_2\\ \bar{y}_3\\ \bar{y}_4\end{pmatrix}$$

$$= \begin{pmatrix}1 & 0 & x_1 & 0 & 0\\ 0 & 1 & x_0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 1 & \cos(v_2) & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & -1\\ 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0\end{pmatrix} \cdot \begin{pmatrix}\bar{x}_0\\ \bar{x}_1\\ \bar{v}_2\\ \bar{y}_0\\ \bar{y}_1\end{pmatrix}$$

$$= \begin{pmatrix}1 & 0 & x_1 & 0 & 0\\ 0 & 1 & x_0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 1 & \cos(v_2) & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}\bar{x}_0\\ \bar{x}_1 - \bar{y}_1\\ \bar{v}_2 + \bar{y}_1\\ \bar{y}_0\\ 0\end{pmatrix}$$

$$= \begin{pmatrix}1 & 0 & x_1 & 0 & 0\\ 0 & 1 & x_0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 1\end{pmatrix} \cdot \begin{pmatrix}\bar{x}_0\\ \bar{x}_1 - \bar{y}_1\\ \bar{v}_2 + \bar{y}_1 + \cos(v_2)\cdot\bar{y}_0\\ 0\\ 0\end{pmatrix} = \dots$$

$$\ldots = \begin{pmatrix} \bar{x}_0 + x_1 \cdot (\bar{v}_2 + \bar{y}_1 + \cos(v_2) \cdot \bar{y}_0) \\ \bar{x}_1 - \bar{y}_1 + x_0 \cdot (\bar{v}_2 + \bar{y}_1 + \cos(v_2) \cdot \bar{y}_0) \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

corresponding to

$$\begin{aligned}
\bar{v}_2 &= \bar{v}_2 + 1 \cdot \bar{y}_1 \\
\bar{x}_1 &= \bar{x}_1 - 1 \cdot \bar{y}_1 \\
\bar{y}_1 &= 0 \\
\bar{v}_2 &= \bar{v}_2 + \cos(v_2) \cdot \bar{y}_0 \\
\bar{y}_0 &= 0 \\
\bar{x}_1 &= \bar{x}_1 + x_0 \cdot \bar{v}_2 \\
\bar{x}_0 &= \bar{x}_0 + x_1 \cdot \bar{v}_2 \\
\bar{v}_2 &= 0
\end{aligned}$$

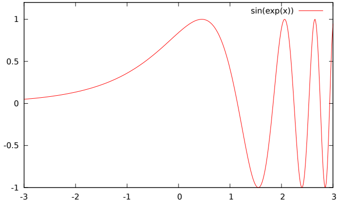
Note the use of v_2 (stored in x_0) prior to input value of x_0.

"Q.E.D"

AD by Overloading

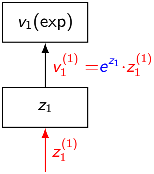
Example: $y = f(x) = \sin(e^x)$

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



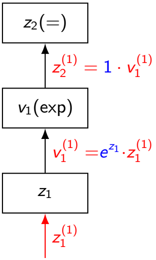
(generated with Gnuplot)

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



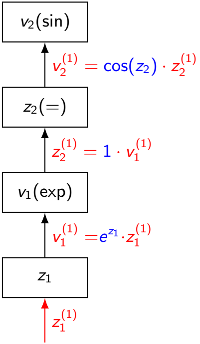
Tangents by Overloading

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



Tangents by Overloading

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

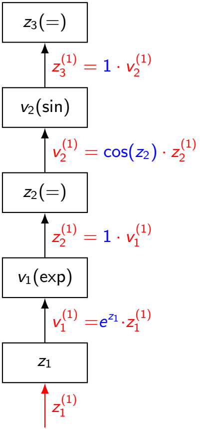


Tangents by Overloading

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

Driver:

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



The given implementation

```
void f(int n, const double* x, int n_p, const double* x_p
       int m, double* y, int m_p, double* y_p);
```

of

$$f : \mathbb{R}^n \times \mathbb{R}^{n_p} \to \mathbb{R}^m \times \mathbb{R}^{m_p} : \quad \begin{pmatrix} y \\ y_p \end{pmatrix} = f(x, x_p)$$

needs to be preprocessed (manually and/or automatically) yielding a new implementation `f'`.

The scope of this preprocessing depends on the AD context. In the simplest case a mere change of the types of all *active* program variables to the desired `dco/c++` type is sufficient. More substantial modifications may be required, e.g., in the context of *checkpointing* or when defining *user-defined intrinsics*.

To compute

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

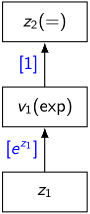
- activate
  - **#include** "dco.hpp"
  - redeclare floating-point variables in F as of type dco::tt1s<**double**>::type
- seed
  - set directional derivatives of active inputs
- run
  - call active F
- harvest
  - get directional derivatives of active outputs

→ Live: Example

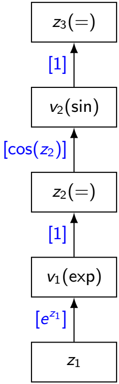
Adjoints by Overloading

... the dco approach (conceptually) ...

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

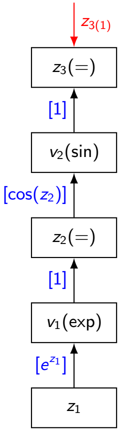


```
void a1_f(a1s::type &[z,a1_z]) {
  [z,TAPE]=exp(z);
  [z,TAPE]=sin(z);
}
...
```

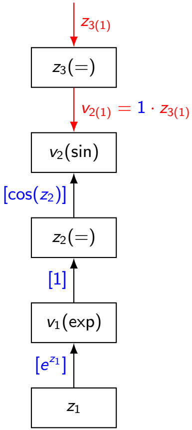


Adjoints by Overloading

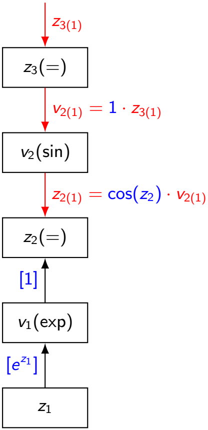
```
void a1_f(a1s::type &[z,a1_z]) {
   [z,TAPE]=exp(z);
   [z,TAPE]=sin(z);
}
...
a1_z=1;
TAPE.interpret();
...
```



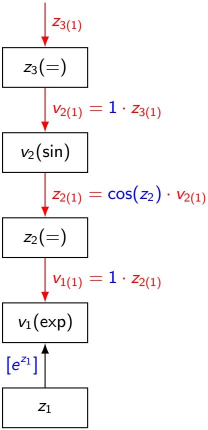
```
void a1_f(a1s::type &[z,a1_z]) {
  [z,TAPE]=exp(z);
  [z,TAPE]=sin(z);
}
...
a1_z=1;
TAPE.interpret();
...
```



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

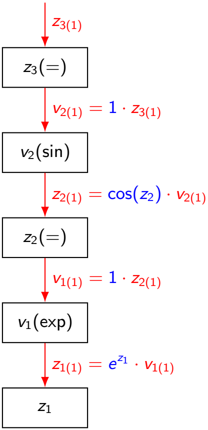


```
void a1_f(a1s::type &[z,a1_z]) {
  [z,TAPE]=exp(z);
  [z,TAPE]=sin(z);
}
...
a1_z=1;
TAPE.interpret();
...
```



Adjoints by Overloading

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



To compute

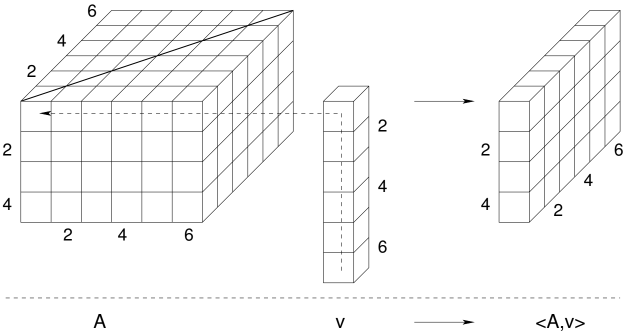
$$\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$$

- activate
  - **#include** "dco.hpp"
  - instantiate tape for base type **double**
  - redeclare floating-point variables in F as of type dco::ga1s<**double**>::type
- tape
  - register active inputs $\mathbf{x}$
  - call active F
  - mark active outputs $\mathbf{y}$
- seed
  - set adjoints $\mathbf{x}_{(1)}$ and $\mathbf{y}_{(1)}$ of active inputs and outputs, respectively
- interpret
  - run tape interpreter
- harvest
  - get adjoints $\mathbf{x}_{(1)}$ of active inputs

→ Live: Example

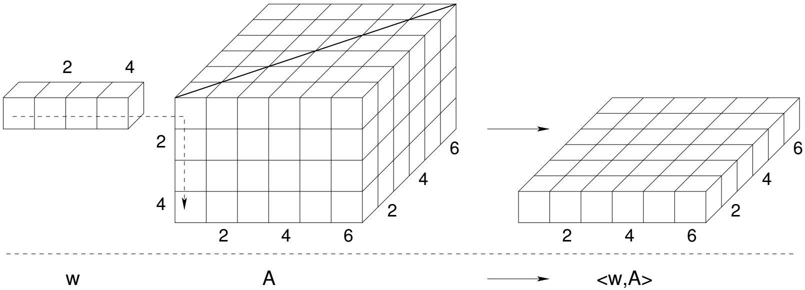
Second (and Higher) Derivatives

Tangent Projection ($A \equiv \nabla^2 F,\ F : \mathbb{R}^6 \to \mathbb{R}^4$)

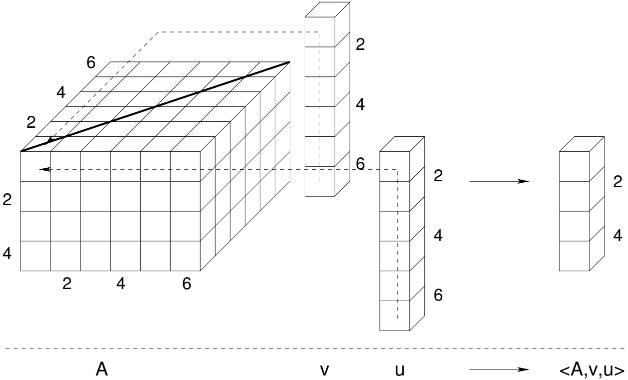


Second (and Higher) Derivatives

Adjoint Projection ($A \equiv \nabla^2 F,\ F : \mathbb{R}^6 \to \mathbb{R}^4$)



Second-Order Tangent Projection ($A \equiv \nabla^2 F,\ F : \mathbb{R}^6 \to \mathbb{R}^4$)



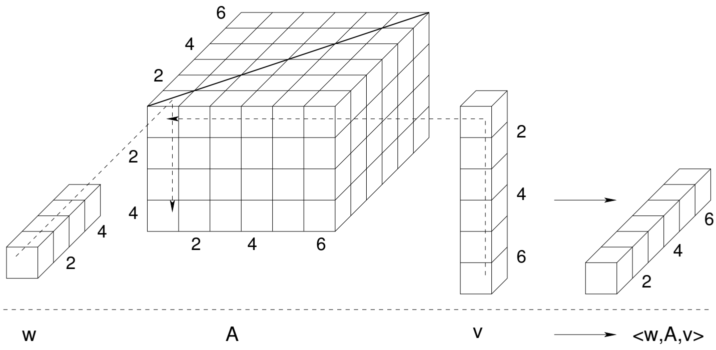
Note:

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

due to symmetry.

Second (and Higher) Derivatives

Second-Order Adjoint Projection ($A \equiv \nabla^2 F$, $F : \mathbb{R}^6 \to \mathbb{R}^4$)



Note:

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

due to symmetry.

To compute

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

- activate
  - **#include** "dco.hpp"
  - redeclare floating-point variables in F as of type dco::tt1s<dco::tt1s<**double**>::type>::type
- seed
  - set tangents $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ of active input
  - set second-order tangent $\mathbf{x}^{(1,2)}$ of active input
- run
  - call active F
- harvest
  - get second-order tangent $\mathbf{y}^{(1,2)}$ of active output

$\rightarrow$ Live: Example

Generic Second-Order Adjoint Mode

To compute

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

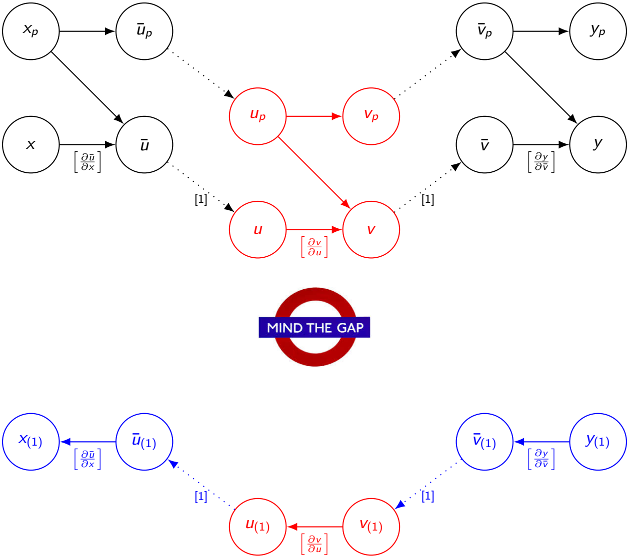
- activate
  - **#include** "dco.hpp"
  - instantiate tape for base type dco::tt1s<**double**>::type
  - redeclare floating-point variables in F as of type dco::ta1s<tt1s<**double**>::type>::type
- seed tangent
  - set tangent $\mathbf{x}^{(2)}$ for active input
- tape
  - register active input $\mathbf{x}$
  - call active F
  - mark active output $\mathbf{y}$
- seed adjoints
  - set adjoint $\mathbf{y}_{(1)}$ of active output
  - set second-order adjoints $\mathbf{x}_{(1)}^{(2)}$ and $\mathbf{y}_{(1)}^{(2)}$ of active input and output, respectively
- interpret
  - run tape interpreter
- harvest
  - get second-order adjoint $\mathbf{x}_{(1)}^{(2)}$ of active input

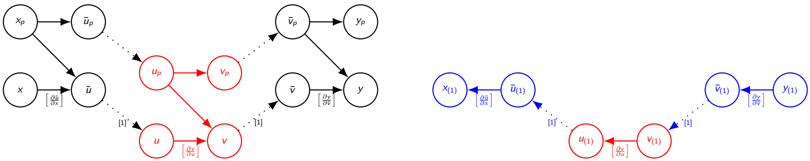
$\rightarrow$ Live: Example

Is it really that easy?

WELL, ...

General Case $(y, y_p) = f(x, x_p)$





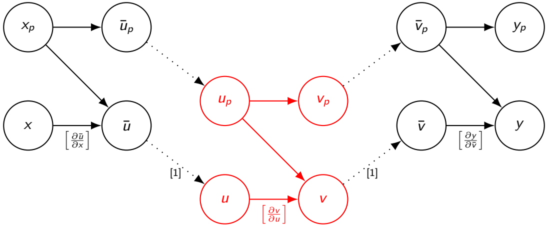
A gap in the tape is introduced by calling a user-defined function make_gap to record the following gap_data:

- Tape location of active gap inputs $\bar{u}$ in order to write $\bar{u}_{(1)} := u_{(1)}$ correctly;
- adjoint gap input checkpoint $\subset (u, u_p, v, v_p)$ in order to initialize interpretation of the gap correctly;
- tape location of active gap outputs $\bar{v}$ in order to initialize $v_{(1)} := \bar{v}_{(1)}$ and, hence, interpretation of gap correctly; requires execution of $g(u, u_p, v, v_p)$.

This data is stored in the tape together with a reference to a user-defined function interpret_gap to increment $\bar{u}_{(1)}$ with $\left(\frac{\partial v}{\partial u}\right)^T \cdot v_{(1)}$.

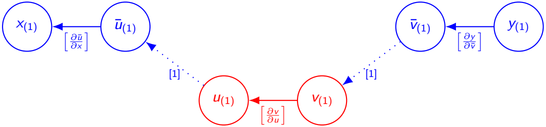
User function make_gap:

- $u :=$ gap_data−>register_input$(\bar{u})$; $u_p := \bar{u}_p$
- gap_data−>write_to_checkpoint(z^-), where $z^- \in (\bar{u}, \bar{u}_p, \emptyset)$
- $g(u, u_p, v, v_p)$
- $\bar{v} :=$ gap_data−>register_output(v)
- gap_data−>write_to_checkpoint(z^+), where $z^+ \in (v, v_p, \emptyset)$
- register_external_function(&interpret_gap,gap_data)



User function interpret_gap :

- gap_data−>read_from_checkpoint(z), where $z \equiv (z^-, z^+)$
- $v_{(1)}$:= gap_data−>get_output_adjoint()
- $g_{(1)}(z, v_{(1)}, v_p, u_{(1)})$
- gap_data−>increment_input_adjoint($u_{(1)}$)



```
void g(int l, int k, const double* u, double* v) { ... }

void f(int n, int m, const double* x, double* y) {
  ...
  g(l,k,u,v);
  ...
}
```

yields

```
|_ f
   |_ g
```

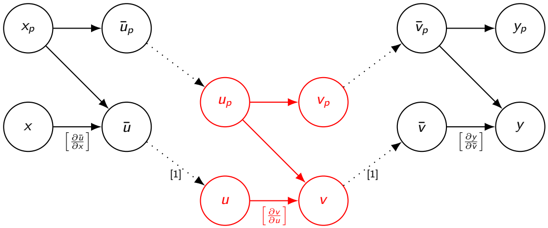
```
|_ f' (MAKE_TAPE)
|    |_ g' (STORE_INPUTS)
|    |_ g
|
|_ f' (INTERPRET_TAPE)
     |_ g' (RESTORE_INPUTS)
     |_ g' (MAKE_TAPE)
     |_ g' (INTERPRET_TAPE)
```

Assumption: f and g are transformed into f' and g', respectively. The new routines implement the four modes (MAKE_TAPE, INTERPRET_TAPE, STORE_INPUTS, RESTORE_INPUTS) that are required by the checkpointed adjoint code.

External Function

Checkpointing (make_gap)

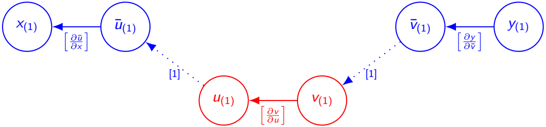
- $u \equiv \bar{u}$ (g'(INTERPRET_TAPE,...) increments $\bar{u}_{(1)}$); $u_p := \bar{u}_p$
- $g(u, u_p, v, v_p)$
- $\bar{v}$:= gap_data−>register_output(v)
- gap_data−>write_to_checkpoint($\bar{u}$, $\bar{u}_p$)
- register_external_function(&interpret_gap,gap_data)



External Function

Checkpointing (interpret_gap)

- gap_data−>read_from_checkpoint(u, u_p)
- $g'(\text{MAKE\_TAPE}, u, u_p)$
- $v_{(1)} :=$ gap_data−>get_output_adjoint()
- $g'(\text{INTERPRET\_TAPE}, v_{(1)})$ (interpretation increments $\bar{u}_{(1)}$)

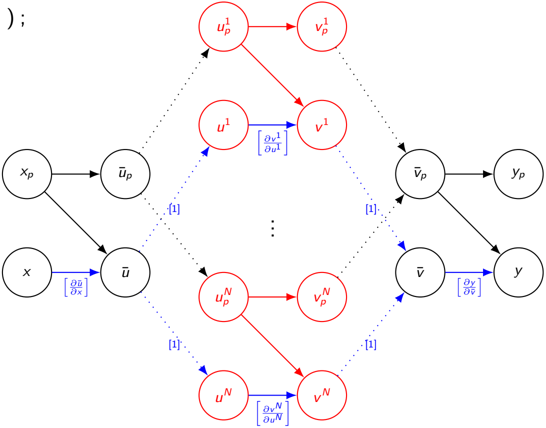


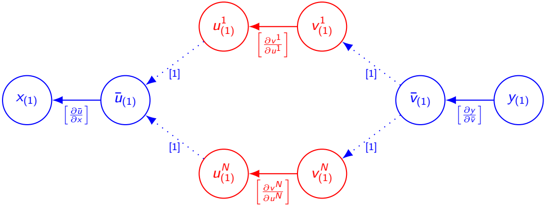
```
template<typename ATYPE>
void f(int n, ATYPE& x) {
  for (int i=0;i<n;i++) x=sin(x);
}
...

int main() {
  ...
  f(n/3,x);
  f_make_gap(n/3,x);
  f(n-n/3*2,x);
  ...
}
```

- n iterations of $x = \sin(x)$ split into thirds
- checkpointed adjoint: `a1s_joint_reversal`
- naive adjoint: single call of `f(n,x)` instead

```
void f(...) {
  ...
  vb=0;
  for (int i=0;i<N;i++) {
    g(u,u_p[i],v[i],v_p[i]);
    vb+=v[i];
    ...
  }
  ...
}
```





- plain adjoint AD likely to run out of memory for large N
- path-wise adjoints reduce memory requirement siginifcantly
- multiple tapes enable thread-parallelism
- optionally: individual paths and their adjoints on accelerators

External Function

Adjoint Ensemble (Implementation)

```
template<typename ATYPE>
void f(int m, const ATYPE& x, const double& r, ATYPE& y) {
  y=0;
  for (int i=0;i<m;i++) y+=sin(x+r);
}
```

- ensemble over x in (random) r
- naive adjoint `a1s_adjoint_ensemble/plain`
- adjoint ensemble `a1s_adjoint_ensemble`

External Function

User-Defined Adjoints (Motivating Example)

$$
\begin{tikzcd}
\boxed{r(x,p) \equiv x^2 - p = 0} \arrow[r, "x_{(1)} \cdot \frac{\partial x}{\partial p}"] \arrow[d, "x = \text{SOLVE}(x^0, p)"] \arrow[dr, "p_{(1)} = \text{SOLVE}_{(1)}(x^0, x_{(1)}, p)"] & \boxed{x_{(1)} \cdot \left(2 \cdot x \cdot \frac{\partial x}{\partial p} - 1\right) = 2 \cdot x \cdot x_{(1)} \cdot \frac{\partial x}{\partial p} - x_{(1)} = 0} \arrow[d, "p_{(1)} = \text{SOLVE}(x, x_{(1)})"] \\
\boxed{x \approx \sqrt{p}} \arrow[ur, dotted] & \boxed{x_{(1)} \cdot \frac{\partial x}{\partial p} = \frac{x_{(1)}}{2 \cdot \sqrt{p}} = \frac{x_{(1)}}{2 \cdot x} \approx p_{(1)} \approx p_{(1)}}
\end{tikzcd}
$$

exact inexact continuous adjoint discrete adjoint

- solving $x^2 - p = 0$ by Newton's method

```
template<typename ATYPE>
void my_sqrt(ATYPE p, ATYPE& x) {
  const double eps=1e-15;
  ATYPE x_old=x+1;
  while (abs(x-x_old)>eps) {
    x_old=x;
    x=x_old-(x_old*x_old-p)/(2*x_old);
  }
}
```

- solving (linear) adjoint equation explicitly

```
void my_adjoint_sqrt(double& a1s_p, double x, double a1s_x) {
  a1s_p=a1s_x/(2*x);
}
```

- discrete adjoint `a1s_user_defined_intrinsic/discrete`
- continuous adjoint `a1s_user_defined_intrinsic`

- EU: AboutFlow, Scorpio, Fortissimo (OpenFOAM, ACE+)
- Tier-1 Investment Banks: Risk management
- NAG, DFG: Adjoint numerical methods (NAG Library)
- DFG: Hybrid discrete adjoints
- BAW, EDF: Waterways engineering (Telemac/Sisyphe)
- MPI-M: Oceanography (ICON GCM)
- JADE: DAEs with optimality conditions
- GRS: Toward robust global (deterministic) optimization
- FNR, Base: Adjoint MPI / OpenMP