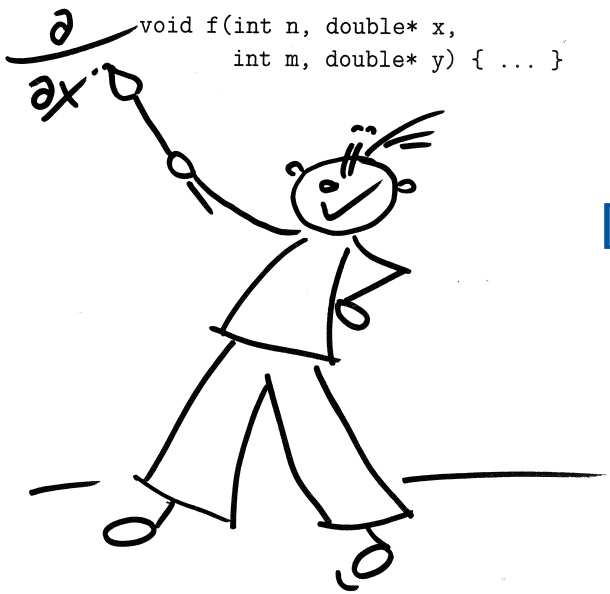
Software and Tools for Computational Engineering
RWTH AACHEN UNIVERSITY



Derivative Code by Overloading in C++

... with a clear focus on adjoints and building on Laurent's talk

Uwe Naumann

RWTH Aachen University, Germany

naumann@stce.rwth-aachen.de

Outline

Let the **C++ program** $\mathbf{y} := F(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$ be continuously differentiable at all points of interest with Jacobian $\nabla F(\mathbf{x}) \equiv \frac{dF}{d\mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{m \times n}$. Typically, F implements the numerical approximation of a function $\mathbf{y}(\mathbf{x})$ implicitly defined via $R(\mathbf{y}(\mathbf{x})) = 0$.

Overloading and template metaprogramming techniques, e.g, in dco/c++,[1] yield the **adjoint program**

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

through use of a custom (e.g, generic adjoint 1st-order scalar) data type

std::vector<dco::ga1s<T>::type> x(n),y(m);

over base type T for recording a (e.g, global) tape

dco::ga1s<T>::tape_t* dco::ga1s<T>::global_tape;

and its subsequent interpretation.

[1] www.nag.co.uk

Implementation

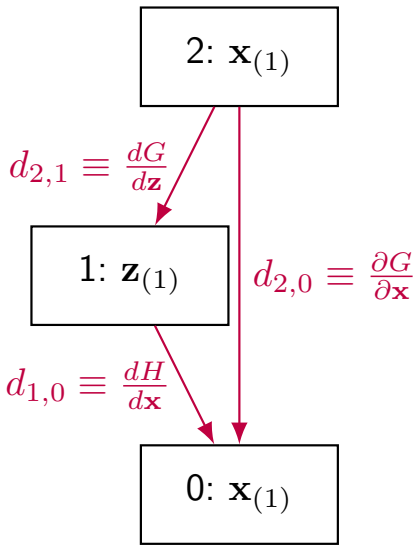
$\mathbf{y} = \underline{\mathbf{x} := F(\mathbf{x})} = G(H(\mathbf{x}), \mathbf{x}) \quad \Rightarrow \quad \mathbf{x}_{(1)} := \nabla F(\mathbf{x})^T \cdot \mathbf{x}_{(1)}$

Source Transformation

$\mathbf{z} := H(\mathbf{x})$
PUSH($\mathbf{x}$) // if TBR($\mathbf{x}$)
$\mathbf{x} := G(\mathbf{z}, \mathbf{x})$ // or move to end

POP($\mathbf{x}$)
$\mathbf{z}_{(1)} := \frac{dG}{d\mathbf{z}}^T \cdot \mathbf{x}_{(1)}$
$\mathbf{x}_{(1)} := \frac{\partial G}{\partial \mathbf{x}}^T \cdot \mathbf{x}_{(1)}$
$\mathbf{x}_{(1)} := \mathbf{x}_{(1)} + \frac{dH}{d\mathbf{x}}^T \cdot \mathbf{z}_{(1)}$

Overloading ($\rightarrow$ Tape [DAG])



$\Leftarrow$ chain rule: $\nabla F(\mathbf{x}) = \sum_{\text{path} \in \text{DAG}} \prod_{(i,j) \in \text{path}} d_{j,i}$

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

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

with Brownian Motion dW and for $x(p(0), 0) = x^0$.
Forward finite differences in time with time step $0 < \Delta t \ll T$ yield the Euler-Maruyama scheme

$$x^{i+1} := x^i + \Delta t \cdot p_i \cdot \sin(x^i \cdot i \cdot \Delta t) + \sqrt{\Delta t} \cdot p_i \cdot \cos(x^i \cdot i \cdot \Delta t) \cdot dW^i$$

for $i = 0, \ldots, n-1$, target time $T = n \cdot \Delta t$, parameter vector $\mathbf{p} = (p_i) \in \mathbb{R}^l$, and with random numbers dW^i drawn from the standard normal distribution $N(0, 1)$. The simulation of m Monte Carlo paths with precomputed random numbers $\texttt{dW} \in \mathbb{R}^{m \times n}$ is implemented as

```
template<typename AFLT_T, typename PFLT_T>
void f(AFLT_T& x, const std::vector<AFLT_T>& p,
       const std::vector<std::vector<PFLT_T>>& dW);
```

Case Study

Gradient with dco/c++: $\mathbf{p}_{(1)} := \frac{df}{d\mathbf{p}}(x,\mathbf{p})^T \cdot x_{(1)}$


```
1  #include "dco.hpp"
2
3  template<typename AFLT_T, typename PFLT_T>
4  void dxdp_a(AFLT_T& x_v, std::vector<AFLT_T>& p_v,
5       const std::vector<std::vector<PFLT_T>> &dW,
6       std::vector<AFLT_T> &dxdp ) {
7    typedef typename dco::ga1s<AFLT_T> DCO_M; // adjoint mode
8    typedef typename DCO_M::type DCO_T; // adjoint type
9    typedef typename DCO_M::tape_t DCO_TT; // tape type
10   int n=p_v.size();
11   DCO_T x=x_v;
12   std::vector<DCO_T> p(n); dco::value(p)=p_v;
13   DCO_M::global_tape=DCO_TT::create(); // create tape
14   DCO_M::global_tape->register_variable(p); // record active input
15   f(x,p,dW); // record active computation
16   dco::derivative(x)=1; // seed
17   DCO_M::global_tape->interpret_adjoint(); // interpret tape
18   dxdp=dco::derivative(p); // harvest
19   DCO_TT::remove(DCO_M::global_tape); // remove tape
20 }
```

Case Study

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

Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

	Time (s)	Memory (MB)	rel. Time
Primal	1.2	79	1
FFD	380	80	317
AAD by hand[2]	1.9	240	1.6
pathwise AAD by hand	2.2	80	1.8
AAD by dco/c++	1.7	728	1.4
pathwise AAD by dco/c++	1.9	86	1.6

FFD: Forward Finite Differences
AAD: Adjoint Algorithmic Differentiation

[2] using basic adjoint code generation rules [1]

Second-Order Adjoints

... actually: Arbitrary-Order Adjoints

W.l.o.g, let the C++ program $y := f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ be twice continuously differentiable with Hessian $\nabla^2 f(\mathbf{x}) \equiv \frac{d^2 f}{d\mathbf{x}^2}(\mathbf{x}) \in \mathbb{R}^{n \times n}$.

Nested first-order derivative types yield 2^{nd}-order adjoints

$$\mathbf{x}_{(1)}^{(2)} \mathrel{+}= y_{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(2)} \quad \text{(Hessian at } O(n) \cdot \text{Cost}(\mathbf{x}_{(1)}^{(2)})) \ ,$$

e.g, tangents of adjoints as

```
<dco::ga1s<dco::gt1s<T>::type>::type
```

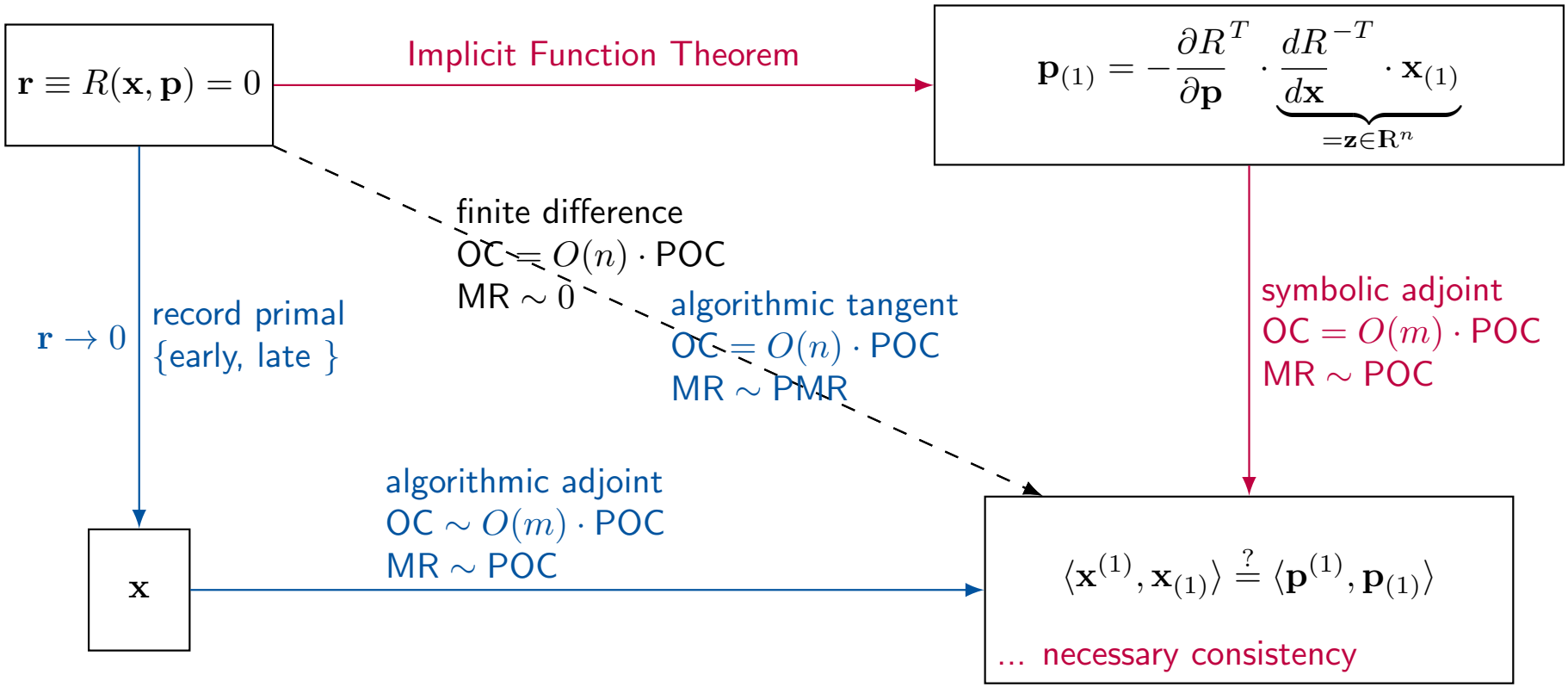
Deeper nesting yields adjoints of arbitrary order, e.g, tangents of adjoints of adjoints as

```
dco::ga1s<dco::ga1s<dco::gt1s<T>::type>::type>::type
```

Outline

Beyond Basic Adjoint AD

"Playground", e.g, $\mathbf{x} = \mathbf{x}(\mathbf{p}) : \mathbb{R}^n \to \mathbb{R}^m \Leftarrow R(\mathbf{x}, \mathbf{p}) = 0$



OC $\hat{=}$ op(eration)s count
POC $\hat{=}$ primal ops count
(P)MR $\hat{=}$ (primal) memory requirement

Note various combinations and nesting.

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

$$\underset{\in \mathbf{R}^n}{\boldsymbol{v}_{0(1)} = \mathbf{x}_{(1)}} := \nabla F(\mathbf{x})^T \cdot \underset{\in \mathbf{R}^m}{\mathbf{y}_{(1)}} = \nabla \varphi_1^T \cdot (\ldots (\nabla \varphi_q^T \cdot \boldsymbol{v}_{q(1)}) \ldots)$$

assuming availability of adjoint elementals

$$\boldsymbol{v}_{i-1\,(1)} = \nabla \varphi_i(\boldsymbol{v}_{i-1})^T \cdot \boldsymbol{v}_{i(1)}$$

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

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

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

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

The corresponding gap in the tape needs to be filled by the adjoint program

$$\mathbf{x}_{(1)} = \boldsymbol{v}_{0\,(1)} := \nabla\varphi_1^T \cdot \ldots \cdot \overbrace{\nabla\varphi_k^T(\boldsymbol{v}_{k-1}) \cdot \underbrace{(\nabla\varphi_{k+1}^T \cdot \ldots \cdot (\nabla\varphi_q^T \cdot \mathbf{y}_{(1)}) \ldots)}_{\boldsymbol{v}_{k\,(1)}}}^{\boldsymbol{v}_{k-1\,(1)}}$$

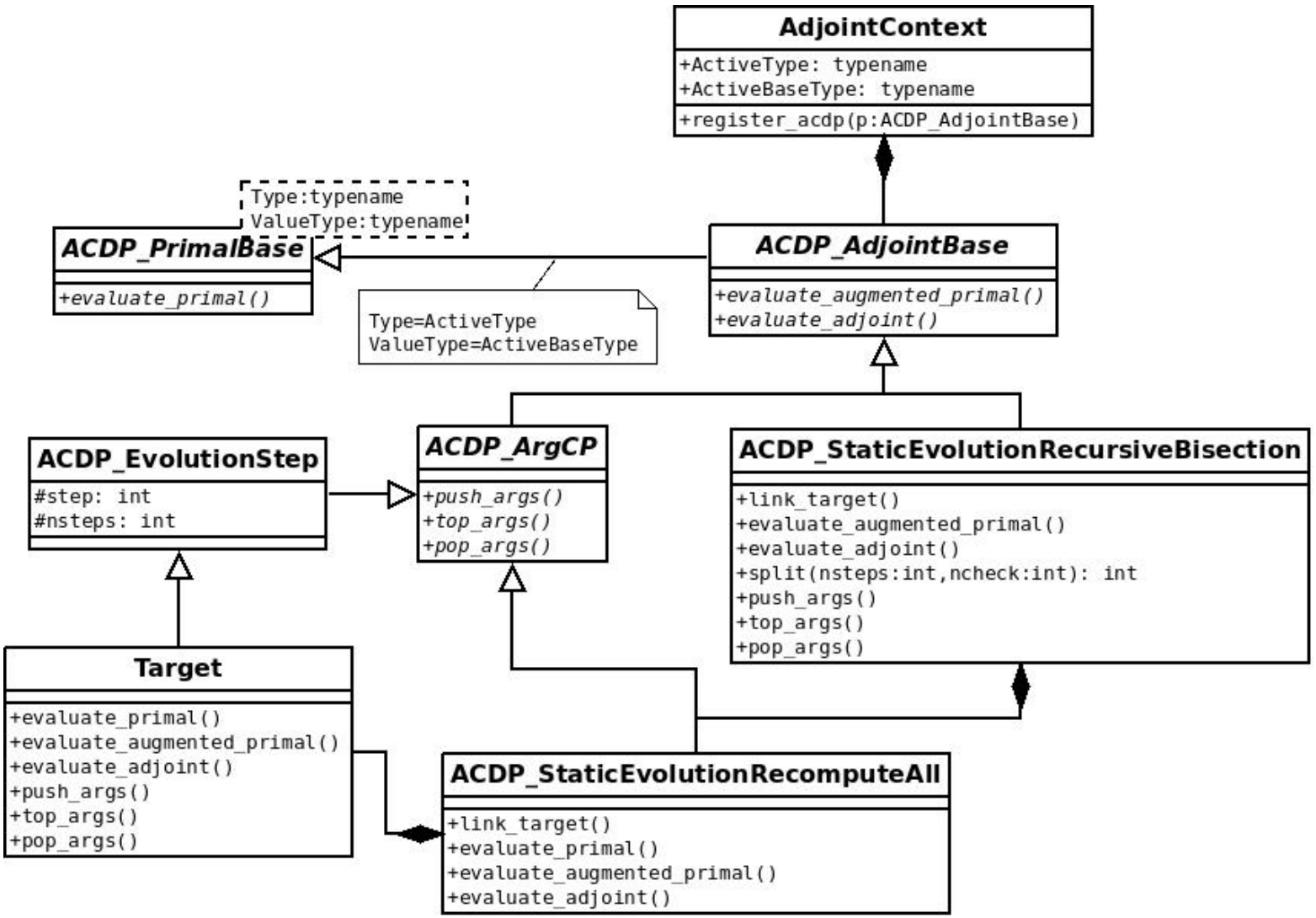
to be filled by a custom version of $\varphi_{k\,(1)}$.

For example, checkpointing methods decrease the maximum tape size by storing $\boldsymbol{v}_{k-1}$ in the forward section followed by the evaluation of the primal φ_k and postponing the generation of the tape for $\varphi_{(1)\,k}$ to the reverse section of $\varphi_{(1)}$.

Further examples include the implementation of symbolic adjoint elementals, preaccumulation, coupling with hand-written / generated adjoint source code and approximation of Jacobians of local black boxes by finite differences.

Adjoint Code Design Patterns

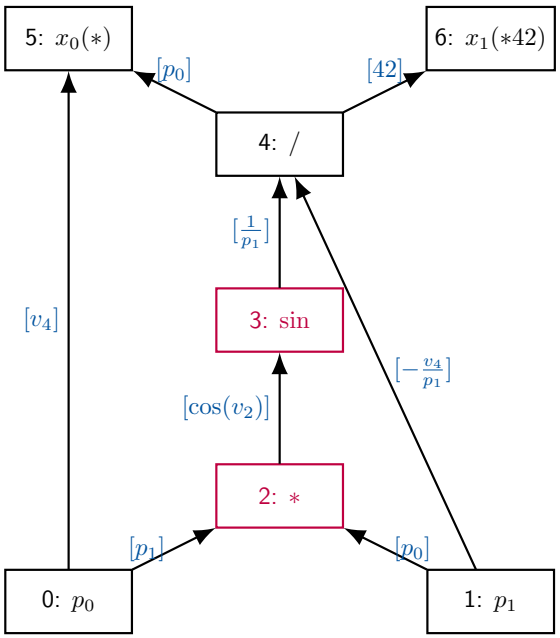
Recursive (e.g, Binomial) Checkpointing



[2]

Preaccumulation

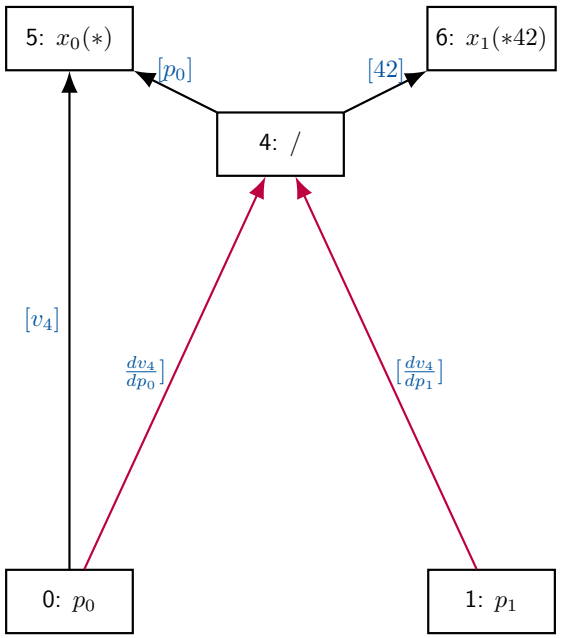
Reducing OC and MR



PMR=7+8=15

POC=5

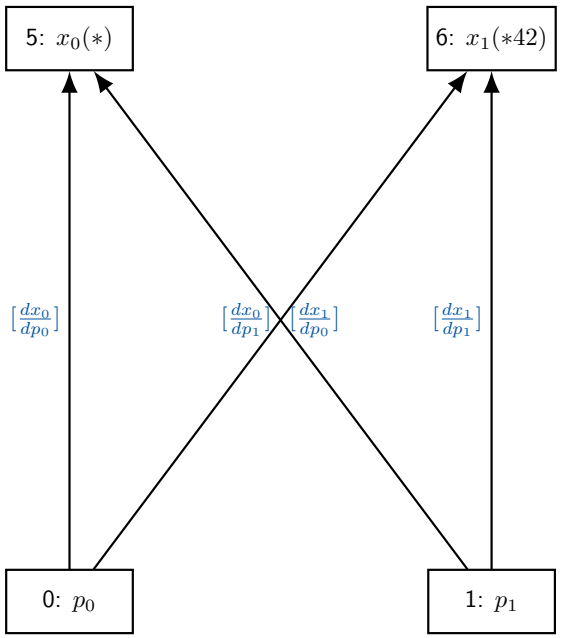
AOC=$2 \cdot 8 = 16$



PMR=$5 + 5 + \ldots$

POC=$5 + \ldots$

AOC=$1 \cdot 5 + \ldots$



$\ldots + 0 = 10$

$\ldots + 0 = 5$

$\ldots + 2 \cdot 5 = 15$

Let the primal matrix-matrix product

$$Y := A \cdot X$$

for $X \in \mathbb{R}^{n \times p}$, $Y \in \mathbb{R}^{m \times p}$, and $A \in \mathbb{R}^{m \times n}$ be implemented (naively) as a function `AX` as follows

```
1   ...
2   for (int i=0;i<m;i++)
3     for (int j=0;j<n;j++)
4       for (int k=0;k<p;k++)
5         Y[i][k]+=A[i][j]*X[j][k];
6   ...
```

Let $Y := A^T \cdot X$ be implemented as `ATX` and $Y := A \cdot X^T$ as `AXT`.

Application of the adjoint code generation rules yields the adjoint version `AX_a`

```
1   ...
2   for (int i=0;i<m;i++)
3     for (int j=0;j<n;j++)
4       for (int k=0;k<p;k++) {
5         A_a[i][j]+=X[j][k]*Y_a[i][k]; // A_a+=AXT(Y_a,X);
6         X_a[j][k]+=A[i][j]*Y_a[i][k]; // X_a+=ATX(A,Y_a);
7       }
8   ...
```

of `AX` implementing $A_{(1)} \mathrel{+}= Y_{(1)} \cdot X^T, \quad X_{(1)} \mathrel{+}= A^T \cdot Y_{(1)}$.

Architecture-specific optimized implementations of `AXT` and `ATX` should be used for the computation of `AX_a`.

Systems of Linear Equations [3]

$$A \cdot \mathbf{x} = \mathbf{b}$$

At the primal solution $\mathbf{x} = \mathbf{x}(A, \mathbf{b})$ we derive

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

Systems of Parameterized Nonlinear Equations [7]

$$F(\mathbf{x}, \mathbf{p}) = 0$$

At the primal solution $\mathbf{x} = \mathbf{x}(\mathbf{p})$ the implicit function theorem yields

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

At the primal solution $\mathbf{x} = \mathbf{x}(\mathbf{p})$ the necessary first-order optimality condition

$$\frac{df}{d\mathbf{x}}(\mathbf{x}, \mathbf{p}) = 0$$

yields

$$\mathbf{z}_{(1)} := \frac{d^2 f}{d\mathbf{x}^2}^{-T} \cdot \mathbf{x}_{(1)} \; ; \quad \mathbf{p}_{(1)} := -\frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{p}}^{T} \cdot \mathbf{z}_{(1)} \; .$$

E.g, the NAG AD Library routine `e04gb_a1w` implements symbolic adjoint of length 1 (scalar) using working precision for

- unconstrained minimum of a sum of squares
- combined Gauss–Newton and quasi-Newton algorithm
- first derivatives provided by user.

Higher-Level Adjoint Elementals

$\min_{\mathbf{x}} f(\mathbf{x}(\mathbf{p}))$ s.t. $g(\mathbf{x}(\mathbf{p}), \mathbf{p}) = 0$

At the primal solution $(\mathbf{x}, \lambda) = (\mathbf{x}(\mathbf{p}), \lambda(\mathbf{p}))$ the KKT condition

$$\frac{d}{d\mathbf{x}}\mathcal{L}(\mathbf{x}, \mathbf{p}) \equiv \frac{d}{d\mathbf{x}}\left[f(\mathbf{x}) + \lambda^T \cdot g(\mathbf{x}, \mathbf{p})\right] = 0$$

yields (with appropriate tensor product $\langle \cdot, \cdot \rangle$ [1])

$$\mathbf{z}_{(1)} := \left(\frac{d^2 f}{d\mathbf{x}^2} + \langle \lambda, \frac{d^2 g}{d\mathbf{x}^2} \rangle\right)^{-T} \cdot \mathbf{x}_{(1)}$$

$$\mathbf{p}_{(1)} := -\left(\frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{p}} + \langle \lambda, \frac{\partial^2 g}{\partial \mathbf{x} \partial \mathbf{p}} \rangle\right)^{T} \cdot \mathbf{z}_{(1)} \, .$$

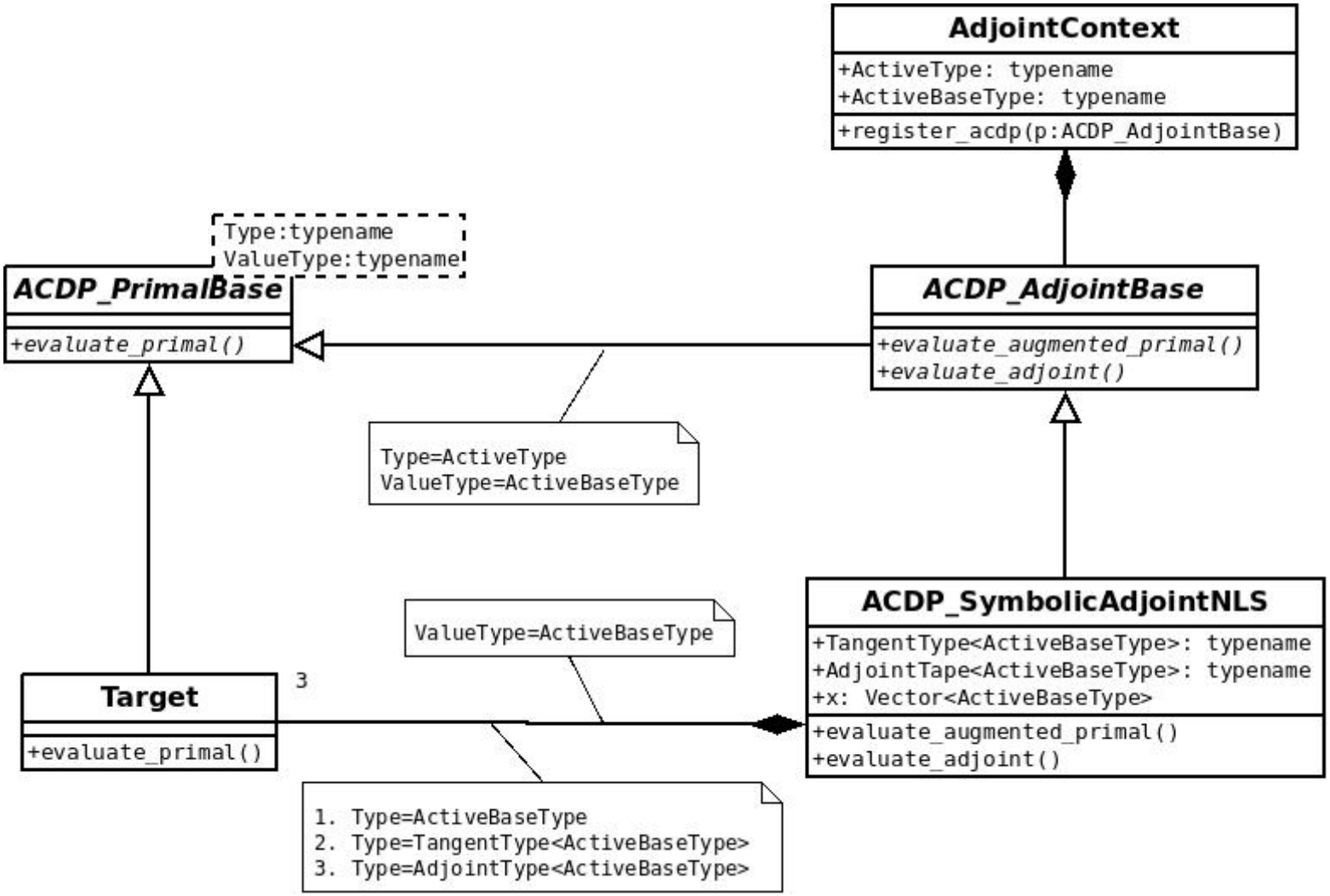
Moreover

$$\frac{df}{d\mathbf{p}}^T = \frac{d\mathbf{x}}{d\mathbf{p}}^T \cdot \frac{df}{d\mathbf{x}}^T := \frac{\partial g}{\partial \mathbf{p}}^T \cdot \lambda$$

identifies the Lagrange multipliers λ as seed of the adjoint constraint for investigating sensitivities of the optimum with respect to $\mathbf{p}$.

Adjoint Code Design Patterns

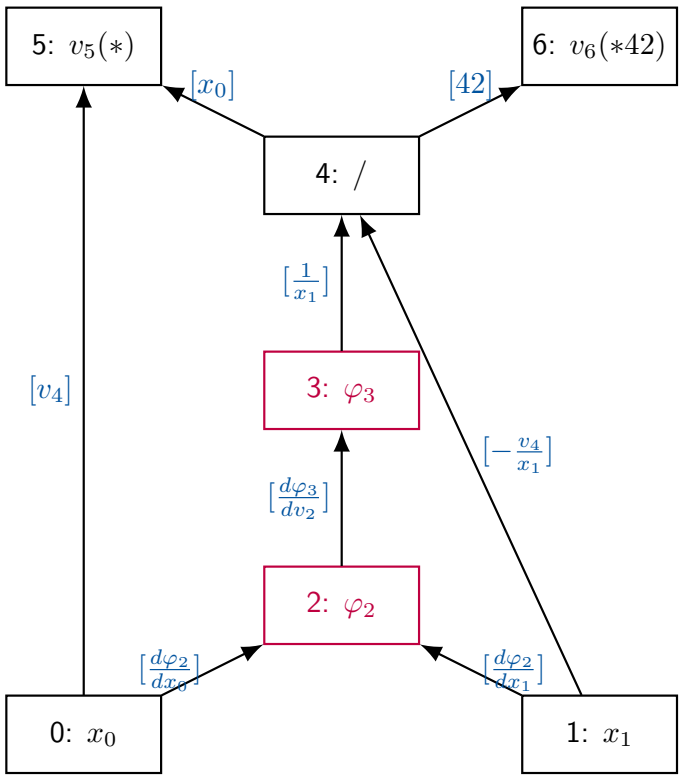
Symbolic Adjoint $F(\mathbf{x}, \mathbf{p}) = 0$



[2]

Debugging Adjoints

$\langle \mathbf{x}^{(1)}, \mathbf{x}_{(1)} \rangle = v^{(1)} \cdot v_{(1)}$



$$\langle \mathbf{x}^{(1)}, \mathbf{x}_{(1)} \rangle \stackrel{?}{=} v_2^{(1)} \cdot v_{2(1)}$$

$$\langle \mathbf{x}^{(1)}, \mathbf{x}_{(1)} \rangle \stackrel{?}{=} v_3^{(1)} \cdot v_{3(1)}$$

$$\langle \mathbf{x}^{(1)}, \mathbf{x}_{(1)} \rangle \stackrel{?}{=} v_4^{(1)} \cdot v_{4(1)}$$

$$\vdots$$

requires `dco::gta1s<T>::type` and primal type generic external adjoints $\varphi_{2(1)}$ and $\varphi_{3(1)}$.

- shared memory parallelism (e.g, MPI)
- distributed memory parallelism (e.g, OpenMP)
- vectorization (e.g, AVX)
- massively parallel acceleration (e.g, GPU)
- cross-language adjoints (e.g, LLVM)
- static/dynamic program analysis (e.g, MIP for CTR)
- globalization of adjoints (e.g, convex relaxations)

Outline

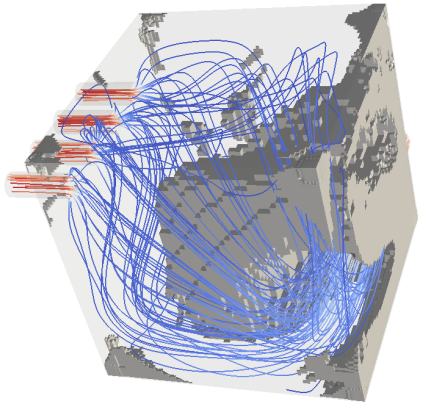
The quality of an adjoint AD solution / tool is defined by

- level of mathematical and software technological abstraction
- robustness wrt. language features of target code
- efficiency of adjoint propagation
- scalability on modern parallel computer architectures
- flexibility wrt. design scenarios
- sustainability wrt. dynamics in user requirements, personnel, hard- and software

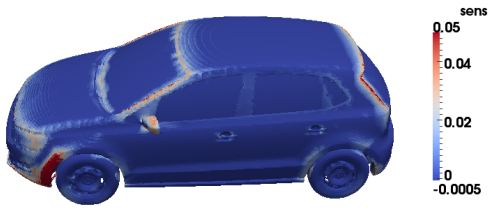
Outline

Selected Recent Applications

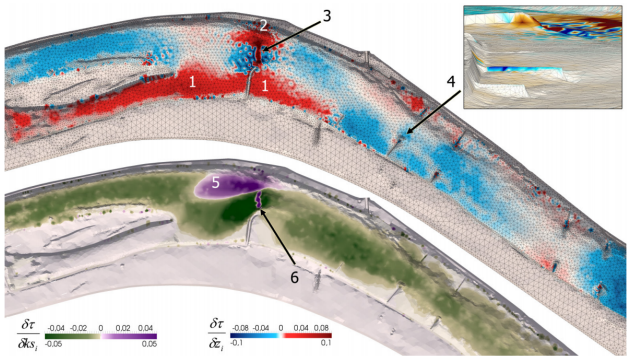
See `www.autodiff.org` for related work by others!



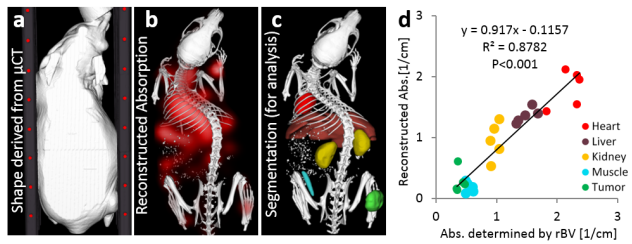
[Towara 2018]



[Towara, N. 2015]



[Kopmann, Riehme, N. 2017]

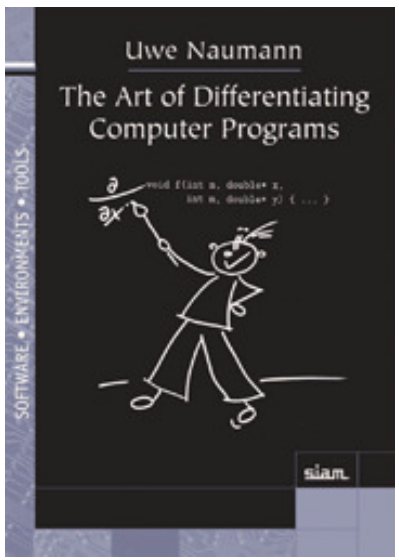


[Gremse, Hoefter, N. 2016]

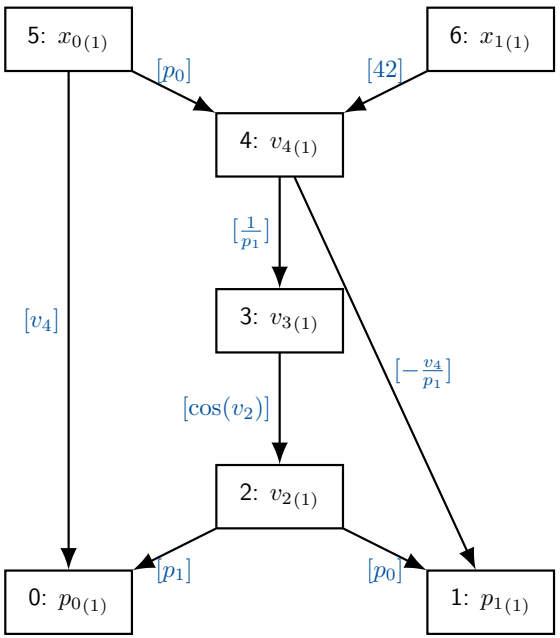
Selected Recent Publications

See `www.autodiff.org` for related work by others!

- [1] →
- [2] U.N.: Adjoint Code Design Patterns, under review.
- [3] U.N.: A Note on Adjoint Linear Algebra, under review.
- [4] M. Towara, U.N.: SIMPLE Adjoint Message Passing, OMS 2018.
- [5] U.N., J. du Toit: Adjoint Algorithmic Differentiation Tool Support for Typical Numerical Patterns in Computational Finance, JCF 2018.
- [6] U.N., K. Leppkes: Low-Memory Algorithmic Adjoint Propagation, SIAM CSC, 2018.
- [7] U.N., K. Leppkes, J. Lotz, M. Towara: Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations, ACM TOMS, 2015.



Outline



Adjoint interpretation (of DAG) yields

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

e.g, with $v_{i(1)} = 0$ for $i = 2, 3, 4$

$$\begin{aligned} v_{4(1)} &+= 42 \cdot x_{1(1)} \\ v_{4(1)} &+= p_0 \cdot x_{0(1)} \\ p_{0(1)} &+= v_4 \cdot x_{0(1)} \\ &\dots \end{aligned}$$

Edges (local partial derivatives; PMR$= |E|$) are accessed sequentially.

Vertices (adjoints; PMR$= |V|$) are accessed "randomly."

Reduction of Adjoint Memory Requirement

Problem Description and Summary of Results [6]

- infeasible memory requirement of naive adjoints
- automatically, stream DAG to slower[3] storage instead of nontrivial checkpointing
- "page faults" due to random access to adjoints
- Idea: separate sequentially from randomly accessed data; stream sequentially accessed data to slower storage; keep randomly accessed data in fast storage $\Rightarrow$ minimize adjoint memory requirement (AMR)

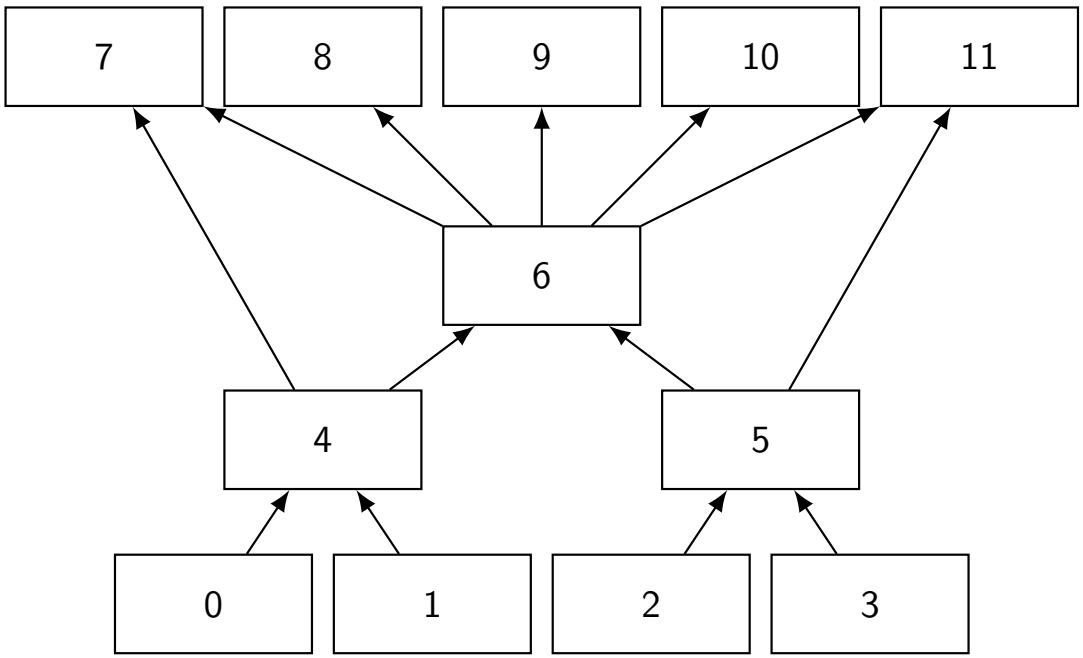
$$\text{AMR} \overset{?}{\ll} |V|$$

- AMR = # program variables ($\rightarrow$ source transformation)
- AMR $= \max_{(i,j)\in E}(j-i) \Rightarrow$ Directed Bandwidth
- decomposition by perpetuation
- case study: Discrete Adjoint OpenFOAM

[3] speed of access

Reduction of Adjoint Memory Requirement

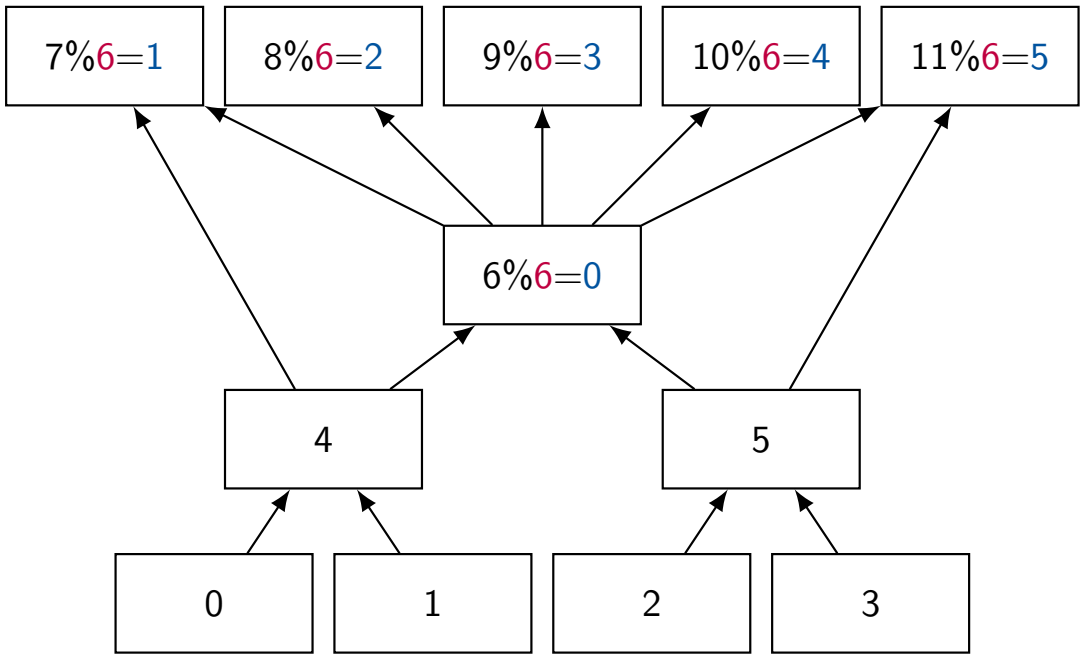
E.g, Variant of *Bat* DAG $\Rightarrow$ AMR $= 12$



Bat [N. 2004]

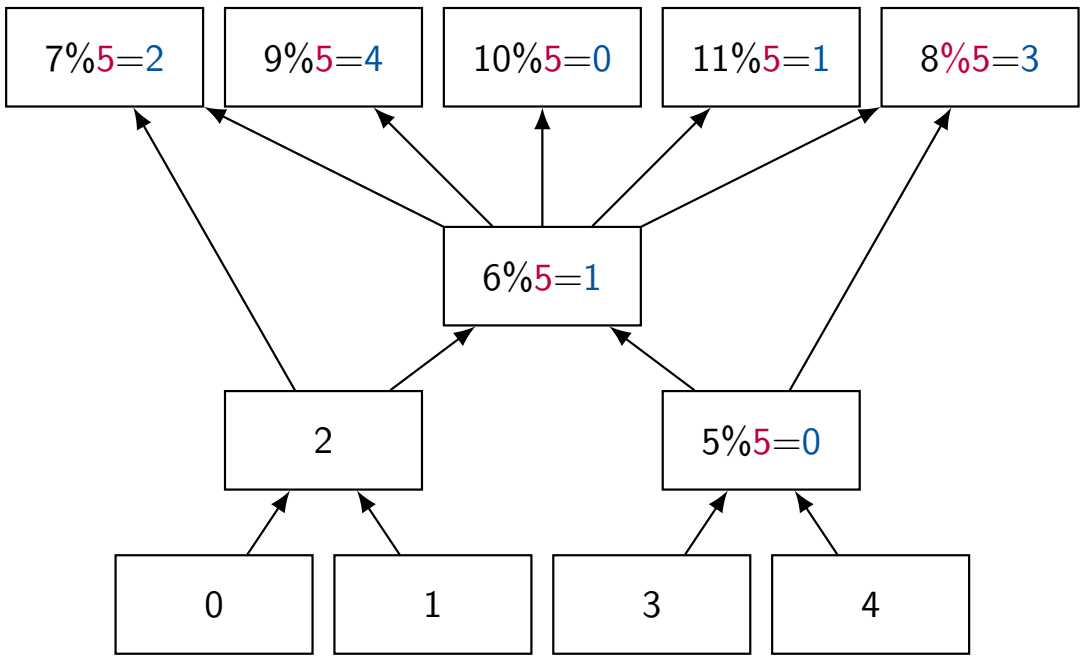
Reduction of Adjoint Memory Requirement

... $\%\max_{(i,j)\in E}(j-i) \Rightarrow \text{AMR} = 6$



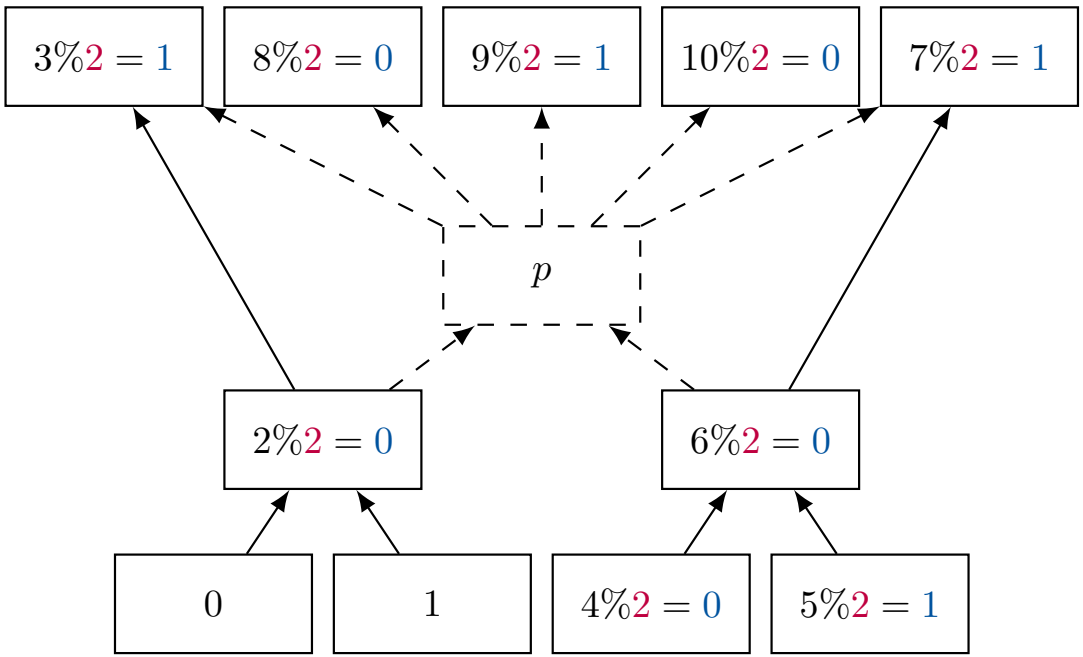
Reduction of Adjoint Memory Requirement

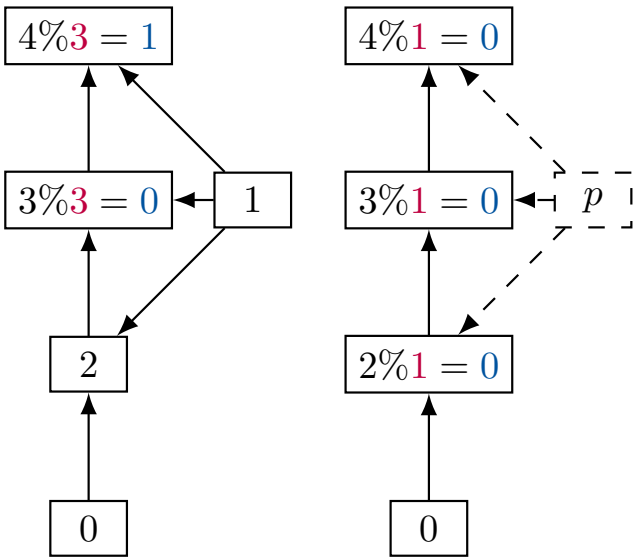
... Directed Bandwidth ⇒ AMR = 5



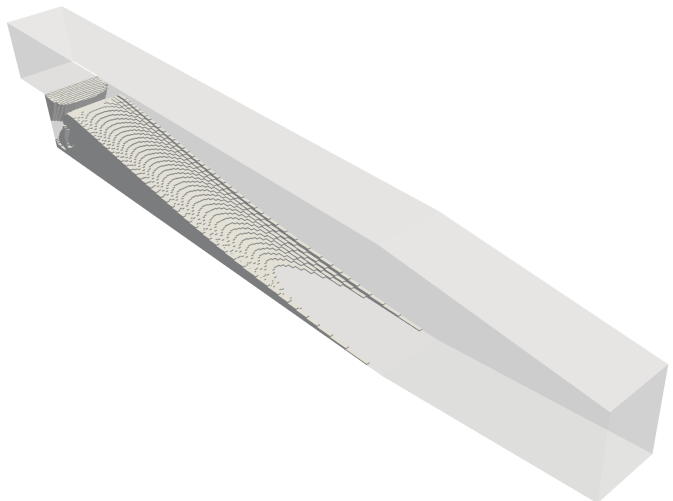
Reduction of Adjoint Memory Requirement

... Perpetuation $\Rightarrow$ AMR $= 3$





Sensitivity of pressure loss from inlet (left) to outlet (right) over $122,250$ cells.



$|E| \sim 1TB$, AMR $\sim 100GB \rightarrow 500MB$[a]

[a] Run time penalty $< 10\%$

Beyond 1st-Order Algorithmic Adjoints

E.g, Hessian by $\mathbf{p}_{(1)}^{(2)} \mathrel{+}= x_{(1)} \cdot \nabla^2 f(\mathbf{p}) \cdot \mathbf{p}^{(2)}$


```
1  template<typename AFLT_T, typename PFLT_T>
2  void ddxdpdp_t(AFLT_T& x_v, std::vector<AFLT_T>& p_v,
3           const std::vector<std::vector<PFLT_T>> &dW,
4           std::vector<std::vector<AFLT_T>> &ddxdpp) {
5    typedef typename dco::gt1s<AFLT_T>::type DCO_T;
6    int n=p_v.size();
7    std::vector<DCO_T> p(n), dxdp(n); dco::value(p)=p_v;
8    for (int i=0;i<n;i++) {
9      DCO_T x=x_v;
10     dco::derivative(p[i])=1;
11     dxdp_a(x,p,dW,dxdp); // first-order adjoint driver
12     ddxdpp[i]=dco::derivative(dxdp);
13     dco::derivative(p[i])=0;
14   }
15 }
```

dco/c++ allows for seamless instantiation of arbitrary combinations of tangent and (non-black-box) adjoint modes.

Beyond 1st-Order Algorithmic Adjoints

E.g, Hessian by $\mathbf{p}_{(2)}^{(1)} \mathrel{+}= x_{(2)}^{(1)} \cdot \nabla^2 f(\mathbf{p}) \cdot \mathbf{p}^{(1)}$


```
1  template<typename AFLT_T, typename PFLT_T>
2  void ddxdpdp_a(
3          AFLT_T& x_v, std::vector<AFLT_T>& p_v,
4          const std::vector<std::vector<PFLT_T>> &dW,
5          std::vector<std::vector<AFLT_T>> &ddxdpdp
6          ) {
7    typedef typename dco::ga1s<AFLT_T> DCO_M;
8    typedef typename DCO_M::type DCO_T;
9    typedef typename DCO_M::tape_t DCO_TT;
10   int n=p_v.size(); std::vector<DCO_T> p(n); dco::value(p)=p_v;
11   for (int i=0;i<n;i++) {
12     DCO_T x=x_v;
13     std::vector<DCO_T> dxdp(n,0);
14     DCO_M::global_tape=DCO_TT::create();
15     DCO_M::global_tape->register_variable(p);
16     dxdp_t(x,p,dW,dxdp); // first-order tangent driver
17     dco::derivative(dxdp[i])=1;
18     DCO_M::global_tape->interpret_adjoint();
19     ddxdpdp[i]=dco::derivative(p);
20     DCO_TT::remove(DCO_M::global_tape);
21   }
22 }
```

Beyond 1st-Order Algorithmic Adjoints

E.g, Hessian by $\mathbf{p}_{(2)} \mathrel{+}= x_{(1)} \cdot \nabla^2 f(\mathbf{p}) \cdot \mathbf{p}_{(1,2)}$


```
1  template<typename AFLT_T, typename PFLT_T>
2  void ddxdpdp_a(
3          AFLT_T& x_v, std::vector<AFLT_T>& p_v,
4          const std::vector<std::vector<PFLT_T>> &dW,
5          std::vector<std::vector<AFLT_T>> &ddxdpdp
6          ) {
7    typedef typename dco::ga1s<AFLT_T> DCO_M;
8    typedef typename DCO_M::type DCO_T;
9    typedef typename DCO_M::tape_t DCO_TT;
10   int n=p_v.size(); std::vector<DCO_T> p(n); dco::value(p)=p_v;
11   for (int i=0;i<n;i++) {
12     DCO_T x=x_v;
13     std::vector<DCO_T> dxdp(n,0);
14     DCO_M::global_tape=DCO_TT::create();
15     DCO_M::global_tape->register_variable(p);
16     dxdp_a(x,p,dW,dxdp); // first-order adjoint driver
17     dco::derivative(dxdp[i])=1;
18     DCO_M::global_tape->interpret_adjoint();
19     ddxdpdp[i]=dco::derivative(p);
20     DCO_TT::remove(DCO_M::global_tape);
21   }
22 }
```

The tangent

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

holds at the primal solution $\mathbf{x}$ (see tangent matrix-vector product).

The corresponding adjoint

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

follows from

$$Y^{(1)} = \sum_i A_i \cdot X_i^{(1)} \cdot B_i \quad \Leftrightarrow \quad X_{i(1)} = A_i^T \cdot Y_{(1)} \cdot B_i^T \ .$$

[3]

$$\frac{dF}{d\mathbf{p}}(\mathbf{x}, \mathbf{p}) \cdot \mathbf{p}^{(1)} = 0$$

implies at the primal solution $\mathbf{x} = \mathbf{x}(\mathbf{p})$

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

The adjoint

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

follows from

$$Y^{(1)} = \sum_i A_i \cdot X_i^{(1)} \cdot B_i \quad \Leftrightarrow \quad X_{i(1)} = A_i^T \cdot Y_{(1)} \cdot B_i^T \ .$$

[3]

KKT

$$0 = \frac{d\mathcal{L}}{d(\mathbf{x}, \lambda)}(\mathbf{x}, \mathbf{p}, \lambda) = \frac{d\left(f(\mathbf{x}, \mathbf{p}) + \lambda^T \cdot g(\mathbf{x}, \mathbf{p})\right)}{d(\mathbf{x}, \lambda)}$$

implies

$$\frac{d\left(f(\mathbf{x}, \mathbf{p}) + \lambda^T \cdot g(\mathbf{x}, \mathbf{p})\right)}{d\mathbf{x}} = 0$$

$$g(\mathbf{x}, \mathbf{p}) = 0$$

at a solution $(\mathbf{x}, \lambda) = (\mathbf{x}(\mathbf{p}), \lambda(\mathbf{p}))$.

Further Details II

Hence, at the solution

$$\begin{aligned}\frac{d\mathcal{L}}{d\mathbf{p}}(\mathbf{x}, \mathbf{p}, \lambda) &= \frac{df}{d\mathbf{p}}(\mathbf{x}, \mathbf{p}) \\ &= \frac{df}{d\mathbf{x}} \cdot \frac{d\mathbf{x}}{d\mathbf{p}} + \frac{\partial f}{\partial \mathbf{p}} + \lambda^T \cdot \left(\frac{dg}{d\mathbf{x}} \cdot \frac{d\mathbf{x}}{d\mathbf{p}} + \frac{\partial g}{\partial \mathbf{p}} \right) \\ &= \underbrace{\left(\frac{df}{d\mathbf{x}} + \lambda^T \cdot \frac{dg}{d\mathbf{x}} \right)}_{=0} \cdot \frac{d\mathbf{x}}{d\mathbf{p}} + \frac{\partial f}{\partial \mathbf{p}} + \lambda^T \cdot \frac{\partial g}{\partial \mathbf{p}}\end{aligned}$$

implying

$$\frac{df}{d\mathbf{p}}(\mathbf{x}) = \lambda^T \cdot \frac{\partial g}{\partial \mathbf{p}}$$

as in this case

$$\frac{\partial f}{\partial \mathbf{p}} = 0 \, .$$