

Automatic Differentiation as a Tool for Computational Science

Jan Hückelheim
Sri Hari Krishna Narayanan

SIAM AN Student Days 2024



U.S. DEPARTMENT OF
ENERGY

Office of
Science

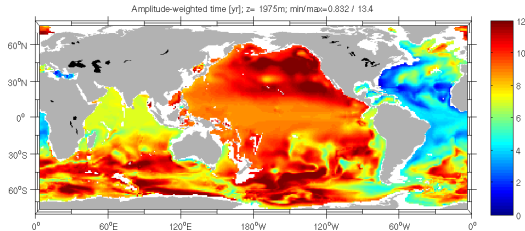
What do we have in store

- ▶ Presented Today:
 - ▶ Introduction
 - ▶ Demo & Hands on: AD basics
 - ▶ Demo & Hands on: Using AD for optimization
- ▶ Further Material:
 - ▶ Seed matrices
 - ▶ Memory requirements
 - ▶ AD for parallel programs
 - ▶ Know what you are differentiating
 - ▶ Adding AD to existing code

Resources: <https://tinyurl.com/siaman24ad>

Why derivatives?

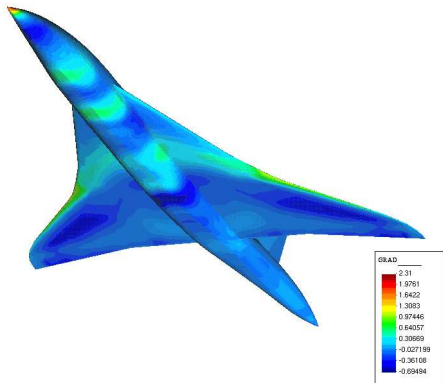
Sensitivity Analysis:



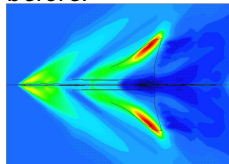
Find sensitivity of the computed field *wrt* one input parameter

Why derivatives?

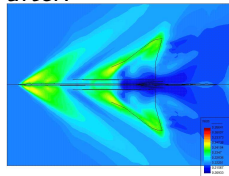
Gradient-driven optimization



before:



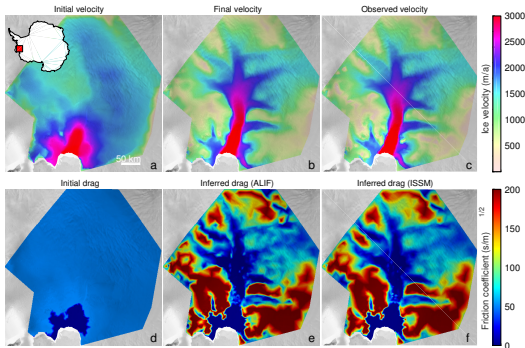
after:



Why derivatives?

Inverse problems:

from measurements and model, estimate hidden parameters



Other uses: Reduced models, Error estimation, Mesh adaption, Uncertainty Quantification, Backpropagation for ML training. . .

What is AD?

The chain rule applied to algorithms

See any algorithm/program $P: \{I_1; I_2; \dots I_p; \}$ as:

$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m \quad F = f_p \circ f_{p-1} \circ \dots \circ f_1$$

Define for short:

$$V_0 = \mathbf{input} \quad \text{and} \quad V_k = f_k(V_{k-1})$$

Apply the chain rule:

$$F'(V_0) = f'_p(V_{p-1}) \times f'_{p-1}(V_{p-2}) \times \dots \times f'_1(V_0)$$

...and transform P to make it compute that.

Cost considerations

$$F'(V_0) = f'_p(V_{p-1}) \times f'_{p-1}(V_{p-2}) \times \cdots \times f'_1(V_0)$$

is often expensive:

- ▶ in computation time
- ▶ in storage space

What can save us:

- ▶ The shapes of the f'_k matter
- ▶ The final usage may not require the full F' but only a projection

Classical projections of F'

- ▶ $F' \times \dot{V}_0$, “forward” or “tangent” mode
- ▶ $\overline{V}_p \times F'$ “reverse” or “adjoint” mode

When full F' needed, use multi-directional AD
→ $F' \times Id$ (or $Id \times F'$),
→ possibly compressed as $F' \times S$ (or $S \times F'$)

For higher-order derivatives, differentiate F'
If directional, differentiate $F' \times \dot{V}_0$

Demo (with Tapenade)

rosenbrock.f90 :

```
REAL*8 FUNCTION ROSENBROCK(x,n) RESULT(y)
  INTEGER :: n
  REAL*8 :: x(0:n)
  y = SUM(100.d0*(x(1:n) - x(0:n-1)**2)**2
          + (1-x(0:n-1))**2)
END FUNCTION ROSENBROCK
```

- ▶ $n + 1$ inputs x , scalar output y
- ▶ Looking for $\frac{dy}{dx}$

Demo with Tapenade: tangent

```
$ tapenade rosenbrock.f90  
$> -head "rosenbrock(y)/(x)" -d
```

rosenbrock_d.f90 :

```
arg1d(:) = 100.d0*2*(x(1:n)-x(0:n-1)**2)  
          *(xd(1:n)-2*x(0:n-1)*xd(0:n-1))  
          - 2*(1-x(0:n-1))*xd(0:n-1)  
arg1 = 100.d0*(x(1:n) - x(0:n-1)**2)**2  
       + (1-x(0:n-1))**2  
yd = SUM(arg1d(:))  
y = SUM(arg1(:))
```

driverTgt.f90 initializes xd to 0,0,0,0,1,0,0,0,0,0 (n=9)

```
$ gfortran rosenbrock_d.f90 driverTgt.f90 -o tangent.exe  
$ ./tangent.exe  
Rosenbrock tangent: -260.4622
```

Demo with Tapenade: gradient

```
$ tapenade rosenbrock.f90  
$> -head "rosenbrock(y)/(x)" -b
```

rosenbrock_b.f90 :

```
xb = 0.0_8  
tempb = 2*(x(1:n)-x(0:n-1)**2)*100.d0*yb  
xb(0:n-1) = xb(0:n-1) - 2*(1-x(0:n-1))*yb  
xb(1:n) = xb(1:n) + tempb  
xb(0:n-1) = xb(0:n-1) - 2*x(0:n-1)*tempb
```

```
$ gfortran rosenbrock_b.f90 driverGrad.f90 -o gradient.exe  
$ ./gradient.exe  
Rosenbrock gradient: 15.9751 50.3524  
-109.0627 -447.2795 -260.4622 -113.7724  
-421.9504 -121.0793 66.6307 6.3868
```

Focus on forward mode: $F' \times \dot{V}_0$

$$F' \times \dot{V}_0 = f'_p(V_{p-1}) \times f'_{p-1}(V_{p-2}) \times \cdots \times f'_1(V_0) \times \dot{V}_0$$

\dot{V}_0 is a vector \Rightarrow compute from right to left!

This corresponds to P's original order

\Rightarrow interleave derivative and primal computation.

Easy!

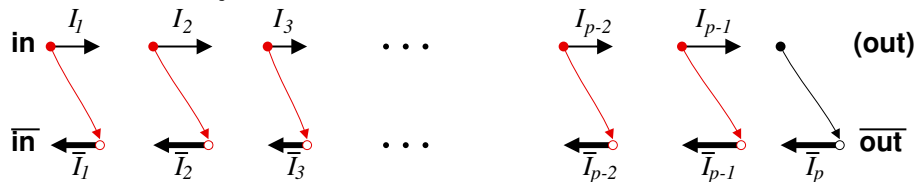


Focus on reverse mode: $\overline{V_p} \times F'$

$$\overline{V_p} \times F' = \overline{V_p} \times f'_p(V_{p-1}) \times f'_{p-1}(V_{p-2}) \times \cdots \times f'_1(V_0)$$

Vector now on the left \Rightarrow compute from left to right

Not so easy, but worth the effort!



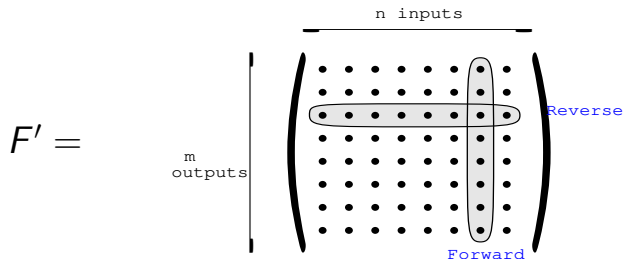
A **forward sweep**, and then a **backward sweep**

The derivative instructions form the **backward sweep**

"**Data-flow reversal**" to get V_k 's in **reverse order**

Cost model

$$F : \text{in} \in \mathbb{R}^n \rightarrow \text{out} \in \mathbb{R}^m$$



- ▶ full F' cost grows like n using the forward mode
Good if $n \leq m$
- ▶ full F' cost grows like m using the reverse mode
Good if $n \gg m$ (e.g. $m = 1$ for a gradient)

Alternatives to AD

- ▶ Symbolic differentiation, Continuous adjoint. . .
Uses equations, not code ;
Duplicates discretization & coding work
- ▶ Finite Differences
Robust wrt coding style & non-differentiable code ;
inaccurate (2nd order contributions) ; only forward mode
- ▶ Complex-step
Astute use of Complex arithmetics ;
similar to AD ; only forward mode

Tool landscape

- ▶ Various AD models \Rightarrow various AD tools:
 - Store derivatives stuck to primals or in separate variables ;
 - Tape all forward computation or only chosen values ;
 - Preaccumulate partials ; Prefer recomputation to storage
- ▶ Various languages \Rightarrow Various tool interfaces:
 - AD by separate tool or embedded in application language ;
 - AD in the compiler ; Differentiated code visible or not ;
 - AD on restricted language or DSL

End of basics

Let's look in detail !