Integrating Scientific Simulations with Machine Learning Algorithms

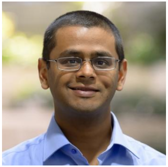
Ludger Paehler
Jan Hückelheim
Sri Hari Krishna Narayanan

SIAM CSE 2023

Welcome

Ludger Paehler

Jan Hückelheim



Krishna Narayanan

- Thanks for showing up on the last day!

What do we have in store

- Session 1:
  - Introduction
  - Seed matrices
  - Demo & Hands on: AD basics in PyTorch, Tapenade & Enzyme
- Session 2:
  - Interfacing in PyTorch
  - Demo & Hands on: Example 1
  - Demo & Hands on: Example 2

Resources: `https://tinyurl.com/siamcse23`

Our goals for this tutorial

Participants will . . .

- be able to use AD tools effectively,
- imagine what ML frameworks do "behind the scenes",
- understand how ML frameworks and simulation code can be interfaced
- understand enough about AD concepts that you can diagnose problems

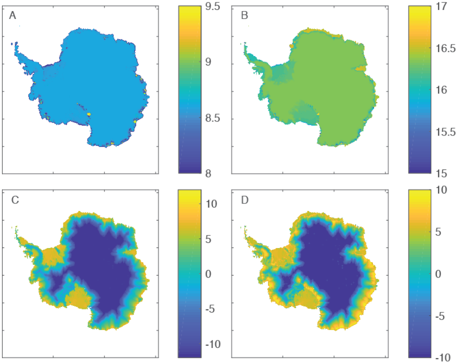
What we assume about you

Computational scientists facing challenging problems, through advanced modeling and simulation, using the most capable computers.

- large complex codes
- continuously developed
- including sophisticated math/physics
- using multiple libraries
- performance is essential
- parallelism involved
- familiar with ML frameworks

Why derivatives?

Sensitivity Analysis:

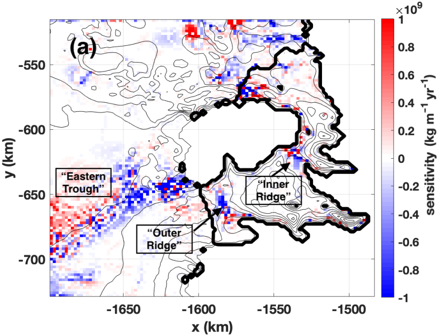


Logarithms of absolute value of adjoint sensitivities, for the Antarctic Ice Sheet. Control variables are the [A] initial ice thickness , [B] mean January precipitation [C] surface temperature, and [D] basal temperature.

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

Why derivatives?

Sensitivity Analysis:

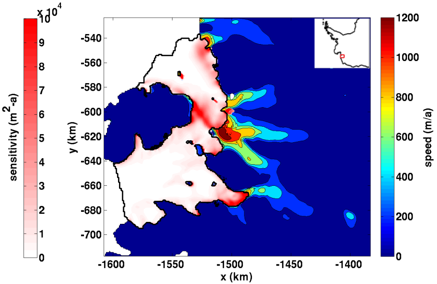


Sensitivity of total (area-integrated) melt to bathymetry in Dotson-Crosson experiment.

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

Why derivatives?

Sensitivity Analysis:

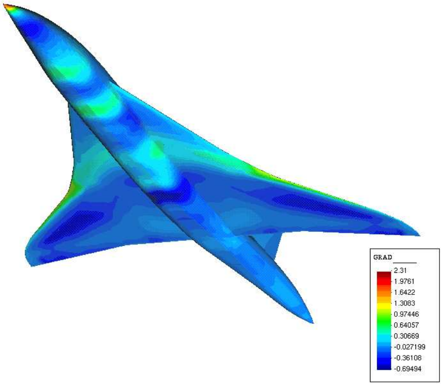


Adjoint sensitivity of loss of VAF to basal melting under the ice shelves adjacent to Smith Glacier.

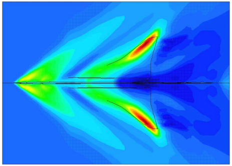
Find sensitivity of the computed field *wrt* one input parameter

Why derivatives?

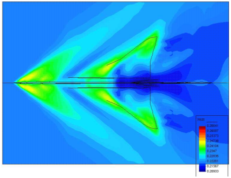
Gradient-driven optimization



before:



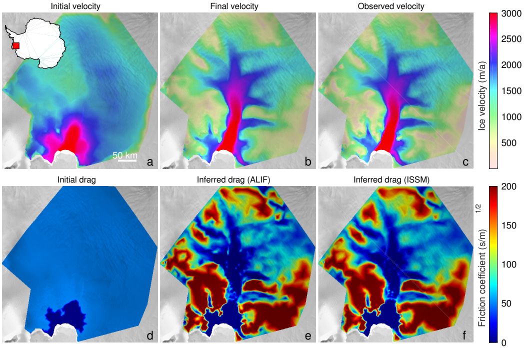
after:



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`: $\{I_1; I_2; \ldots I_p;\}$ as:

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

Define for short:

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

Apply the chain rule:

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

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

Cost considerations

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

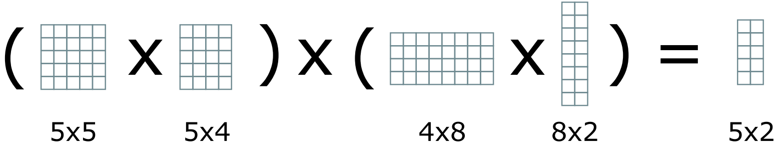
is often expensive:

- 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

Cost considerations



- How should we group the various matrices?
- The order matters!
- Classical projections of F'
  - $F' \times \dot{V}_0$, "forward" or "tangent" mode
  - $\overline{V_p} \times F'$ "reverse" or "adjoint" mode

Demo (with Tapenade)

```
SUBROUTINE FOO(x, y)
  IMPLICIT NONE
  REAL :: x, y
  INTRINSIC SIN
  y = SIN(x)**2
END SUBROUTINE FOO
```

- Can you compute the derivatives yourself?

Demo with Tapenade: tangent

```
SUBROUTINE FOO_D(x, xd, y, yd)
  IMPLICIT NONE
  REAL :: x, y
  REAL :: xd, yd
  INTRINSIC SIN
  REAL :: temp
  temp = SIN(x)
  yd = 2*temp*COS(x)*xd
  y = temp*temp
END SUBROUTINE FOO_D
```

- Note the common sub-expression elimination
- This computes the primal as well as the derivatives

Demo with Tapenade: adjoint

```
SUBROUTINE FOO_B(x, xb, y, yb)
  IMPLICIT NONE
  REAL :: x, y
  REAL :: xb, yb
  INTRINSIC SIN
  xb = xb + COS(x)*2*SIN(x)*yb
  yb = 0.0
END SUBROUTINE FOO_B
```

- The adjoint and the tangent are very similar
- Note that the primal code is not computed alongisde the adjoint

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

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

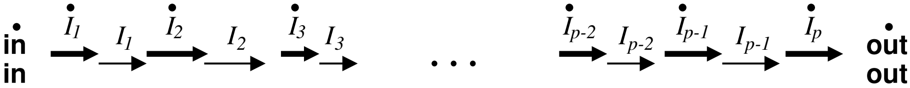
$\dot{V}_0$ is a vector $\Rightarrow$ compute from right to left!

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

This corresponds to `P`'s original order

$\Rightarrow$ interleave derivative and primal computation.

Easy!



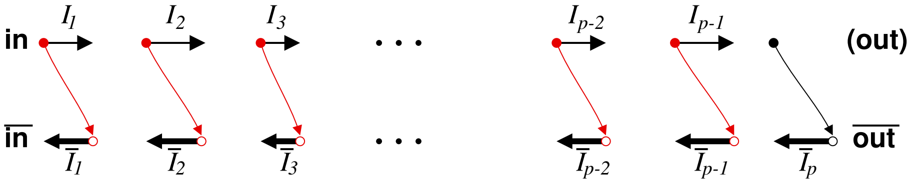
Focus on reverse mode: $\overline{V_p} \times F'$

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

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

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

Not so easy, but worth the effort!



A forward sweep, and then a backward sweep

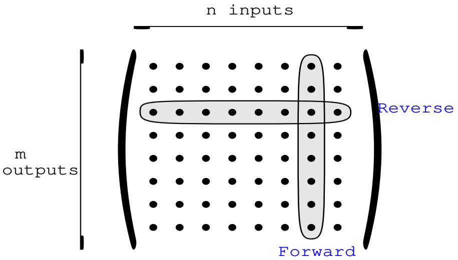
The derivative instructions form the backward sweep

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

Cost model

$$F \quad : \quad \textbf{in} \in \mathbb{R}^{n} \;\rightarrow\; \textbf{out} \in \mathbb{R}^{m}$$

$F' =$



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

End of basics

Let's look in detail !