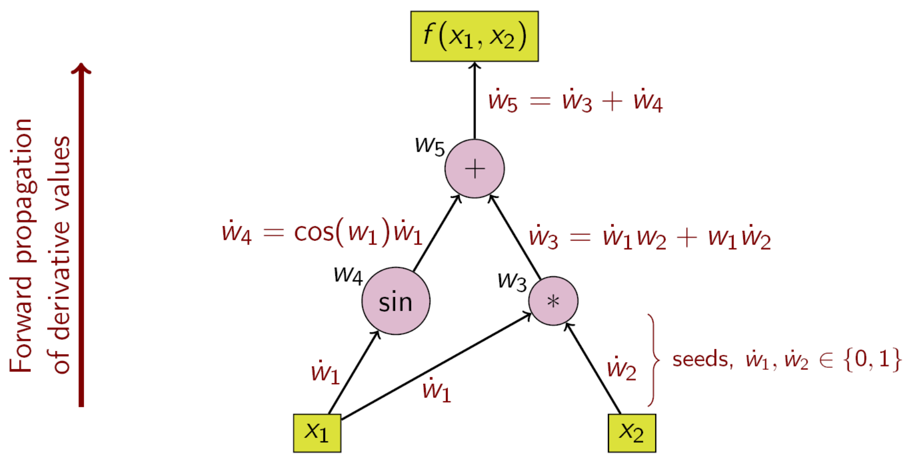
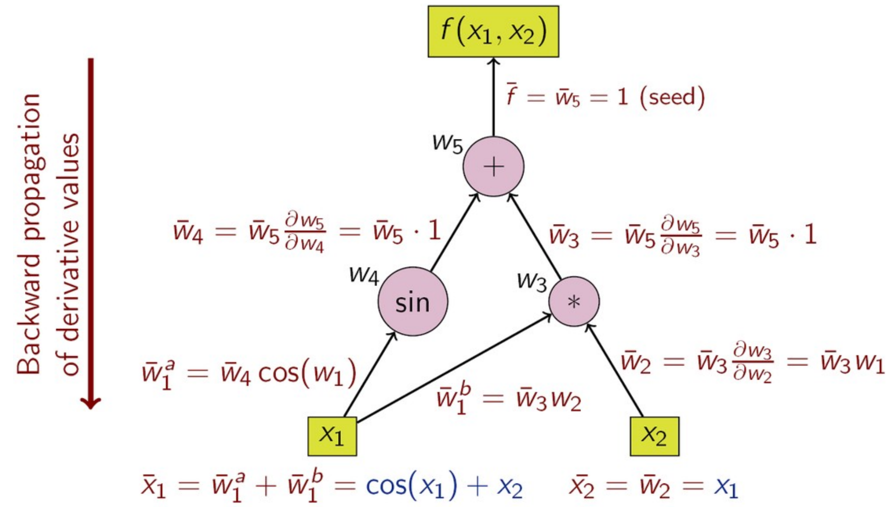
Automatic Differentiation

Presented by: Yubo “Paul” Yang






```
BinOp(
    left=Name(id='x1', ctx=Load()),
    op=Add(),
    right=BinOp(
        left=Name(id='x1', ctx=Load()),
        op=Mult(),
        right=Name(id='x2', ctx=Load())
    )
)
```

AD →

```
BinOp(
    left=Name(id='x1_d'),
    op=Add(),
    right=BinOp(
        left=BinOp(
            left=Name(id='x1', ctx=Load()),
            op=Mult(),
            right=Name(id='x2_d')
        ),
        op=Add(),
        right=BinOp(
            left=Name(id='x1_d'),
            op=Mult(),
            right=Name(id='x2', ctx=Load())
        )
    )
)
```

Motivation

JuliaDiff

Differentiation tools in Julia. JuliaDiff on GitHub.

Stop approximating derivatives!

Derivatives are required at the core of many numerical algorithms. Unfortunately, they are usually computed *inefficiently* and *approximately* by some variant of the finite difference approach

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, h \text{ small}.$$

This method is *inefficient* because it requires $\Omega(n)$ evaluations of $f : \mathbb{R}^n \to \mathbb{R}$ to compute the gradient $\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \cdots, \frac{\partial f}{\partial x_n}(x) \right)$, for example. It is *approximate* because we have to choose some finite, small value of the step length h, balancing floating-point precision with mathematical approximation error.

What can we do instead?

One option is to explicitly write down a function which computes the exact derivatives by using the rules that we know from Calculus. However, this quickly becomes an error-prone and tedious exercise. **There is another way!** The field of automatic differentiation provides methods for automatically computing *exact* derivatives (up to floating-point error) given only the function f itself. Some methods use many fewer evaluations of f than would be required when using finite differences. In the best case, **the exact gradient of f can be evaluated for the cost of $O(1)$ evaluations of f itself**. The caveat is that f cannot be considered a black box; instead, we require either access to the source code of f or a way to plug in a special type of number using operator overloading.

Main Idea

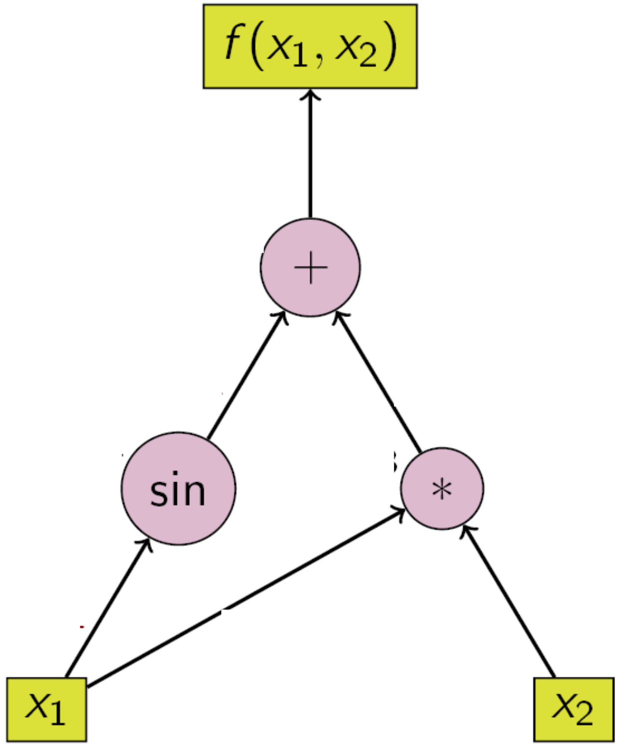
- Generalizable to: $f : \mathbb{R}^N \rightarrow \mathbb{R}^M$

- Many algebraic functions of the form: $f : \mathbb{R}^N \rightarrow \mathbb{R}$

Can be decomposed into smooth elementary operations.

- Smooth elementary operations have know exact derivatives.

- Chain rule!



Wikipedia example:

$$f(x_1, x_2) = \sin(x_1) + x1x2$$

Wikipedia Example: Forward Propagation

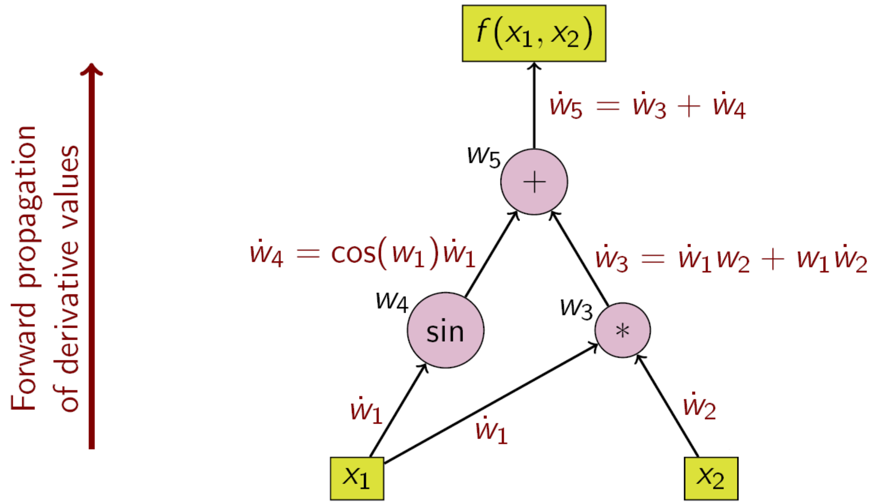
Start from independent variables

Say: w1=x1=0
w2=x2=0, want df/dx1

Step 1: seed

$$\dot{w}_1 = \frac{\partial x_1}{\partial x_1} = 1 \qquad \dot{w}_2 = \frac{\partial x_2}{\partial x_1} = 0$$

value(w2) → ∂x_2; independent → ∂x_1



Step 2: follow an arrow, use chain rule and known derivative
Say: left-most arrow $\dot{w}_4 = \cos w_1 \dot{w}_1 = \cos 0 = 1$

Keep following arrows, use chain rule and known derivatives

$$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2 = 1 * 0 + 0 * 0 = 0$$

$$\dot{w}_5 = \dot{w}_3 + \dot{w}_4 = 0 + 1 = \boxed{1}$$

Viola! The answer is 1!

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

$$\frac{df}{dx_1}(x_1, x_2) = \cos(x_1)\dot{x}_1 + x_2$$

Wikipedia Example: Backward Propagation

Start from dependent variable

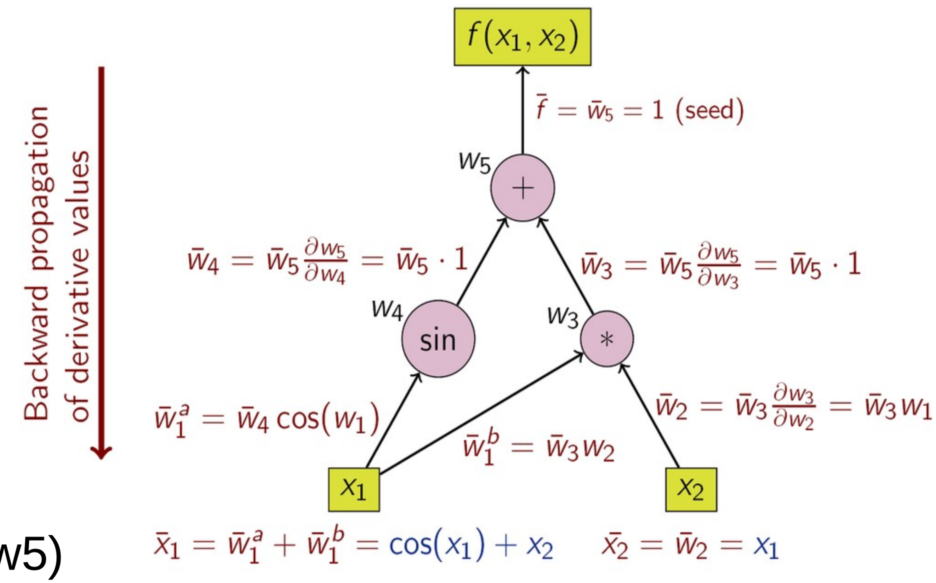
Say: f(x1,x2)=0, want df/dx1

Step 1: seed

dependent

Adjoint of f → $$\bar{w}_5 = \frac{\partial f}{\partial f} = 1$$

value (w5)



Step 2: follow an arrow *backwards*, use chain rule and known derivative. Say: left-most arrow

$$\bar{w}_4 \equiv \frac{dw_5}{dw_4} = \frac{\partial w_5}{\partial w_5}\frac{\partial w_5}{\partial w_4} = \bar{w}_5 * 1 = 1$$

Keep following arrows, use chain rule and known derivatives

Many arrows to follow! Results are functions of inputs!

Munch.. munch ...

$$\frac{df}{dx_1} = \bar{w}_1^a + \bar{w}_1^b = \cos x_1 + x_2$$

$$\frac{df}{dx_2} = \bar{w}_2 = x_1$$

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

$$\frac{df}{dx_1}(x_1, x_2) = \cos(x_1)\dot{x}_1 + x_2$$

Implementation

Source Code Transformation (SCT)

ad_sct.ipynb

```
BinOp(
    left=Name(id='x1', ctx=Load()),
    op=Add(),
    right=BinOp(
        left=Name(id='x1', ctx=Load()),
        op=Mult(),
        right=Name(id='x2', ctx=Load())
    )
)
```

$$y = x_1 + x_1 x_2$$

SCT

$$\bar{y} = \bar{x}_1 + x_1 \bar{x}_2 + \bar{x}_1 x_2$$

```
BinOp(
    left=Name(id='x1_d'),
    op=Add(),
    right=BinOp(
        left=BinOp(
            left=Name(id='x1', ctx=Load()),
            op=Mult(),
            right=Name(id='x2_d')
        ),
        op=Add(),
        right=BinOp(
            left=Name(id='x1_d'),
            op=Mult(),
            right=Name(id='x2', ctx=Load())
        )
    )
)
```

Operator Overloading (OO)

Dual number

$$x \to \langle x, x' \rangle$$

Dual Arithmetic

$$\langle u, u' \rangle + \langle v, v' \rangle = \langle u + v, u' + v' \rangle$$
$$\langle u, u' \rangle - \langle v, v' \rangle = \langle u - v, u' - v' \rangle$$
$$\langle u, u' \rangle * \langle v, v' \rangle = \langle uv, u'v + uv' \rangle$$
$$\langle u, u' \rangle \,/\, \langle v, v' \rangle = \left\langle \frac{u}{v}, \frac{u'v - uv'}{v^2} \right\rangle \quad (v \neq 0)$$
$$\sin\langle u, u' \rangle = \langle \sin(u), u' \cos(u) \rangle$$
$$\cos\langle u, u' \rangle = \langle \cos(u), -u' \sin(u) \rangle$$
$$\exp\langle u, u' \rangle = \langle \exp u, u' \exp u \rangle$$
$$\log\langle u, u' \rangle = \langle \log(u), u'/u \rangle \quad (u > 0)$$
$$\langle u, u' \rangle^k = \langle u^k, k u^{k-1} u' \rangle \quad (u \neq 0)$$
$$|\langle u, u' \rangle| = \langle |u|, u' \operatorname{sign} u \rangle \quad (u \neq 0)$$

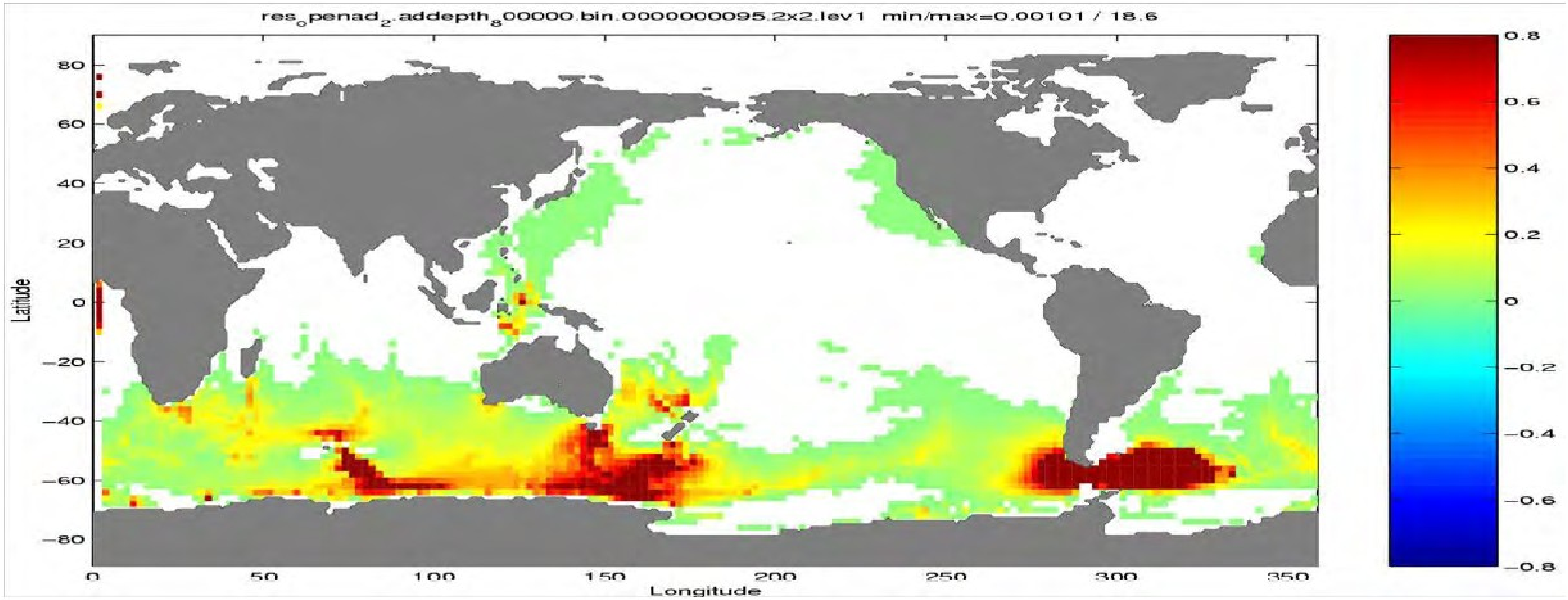
Applications

- Parameter Tuning
- Sensitivity Analysis
- Mesh Quality Optimization
- Nonlinear PDEs

Stolen slides start here
Credit: Boyana Norris

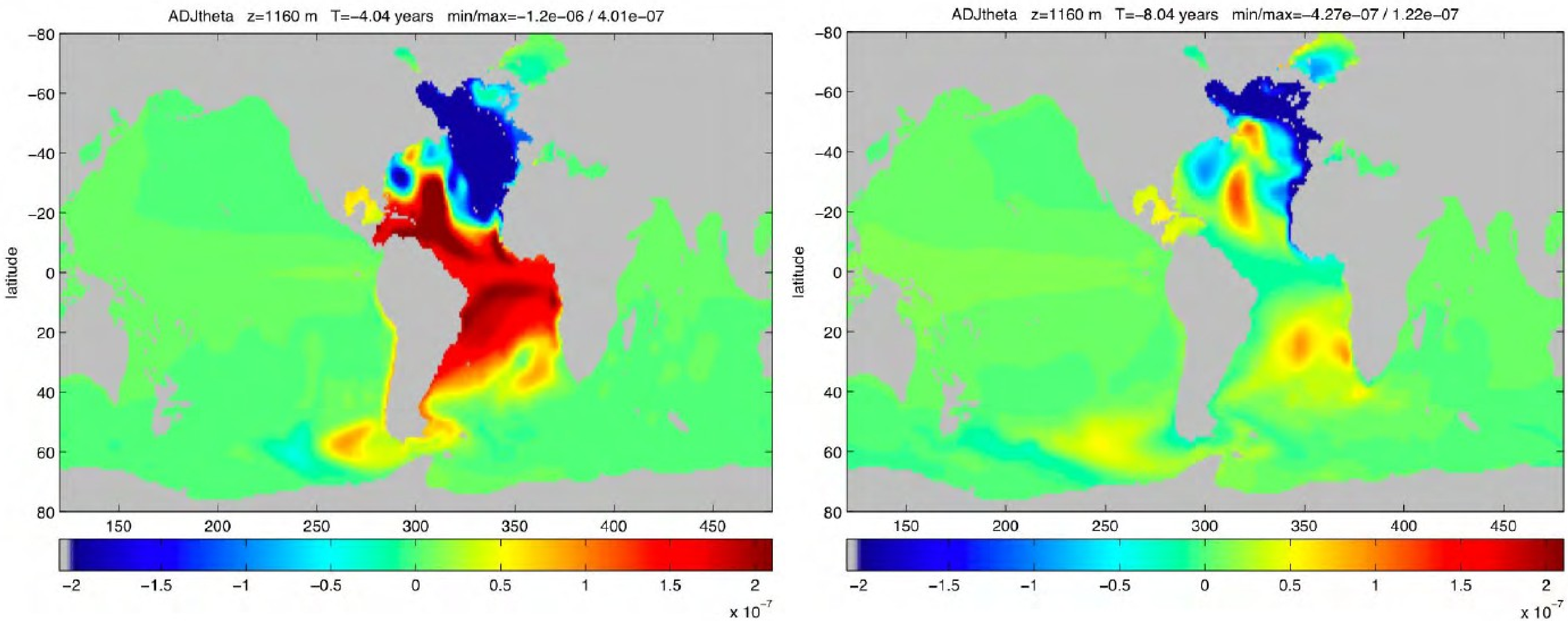
Application: Sensitivity analysis in simplified climate model

- Sensitivity of flow through Drake Passage to ocean bottom topography
  - Finite difference approximations: 23 days
  - Naïve automatic differentiation: 2 hours 23 minutes
  - Smart automatic differentiation: 22 minutes



Application: Preliminary results for MITgcm

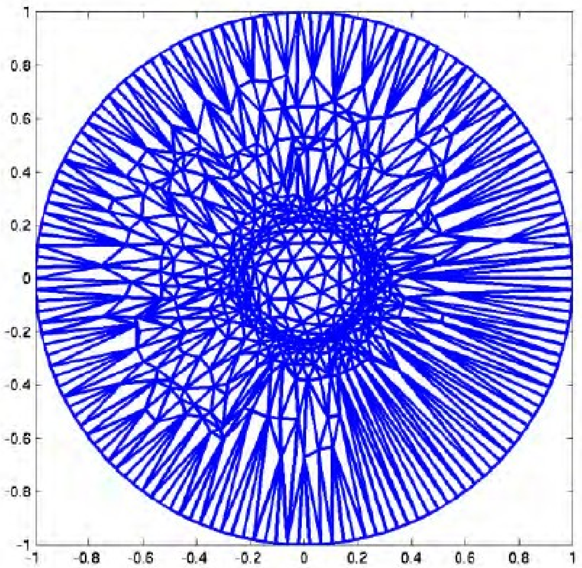
- Time for one simulation run (20 years at 4 degree resolution): 51.75 hrs
- Time for one gradient computation using AD: 204.2 hrs (8.5 days)



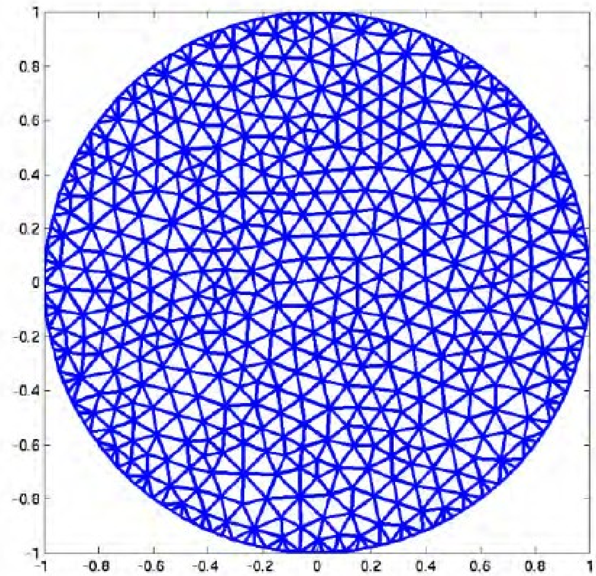
Application: mesh quality optimization

- Optimization used to move mesh vertices to create elements as close to equilateral triangles/tetrahedra as possible
- Semi-automatic differentiation is 10-25% faster than hand-coding for gradient and 5-10% faster than hand-coding for Hessian
- Automatic differentiation is a factor 2-5 times faster than finite differences

Before

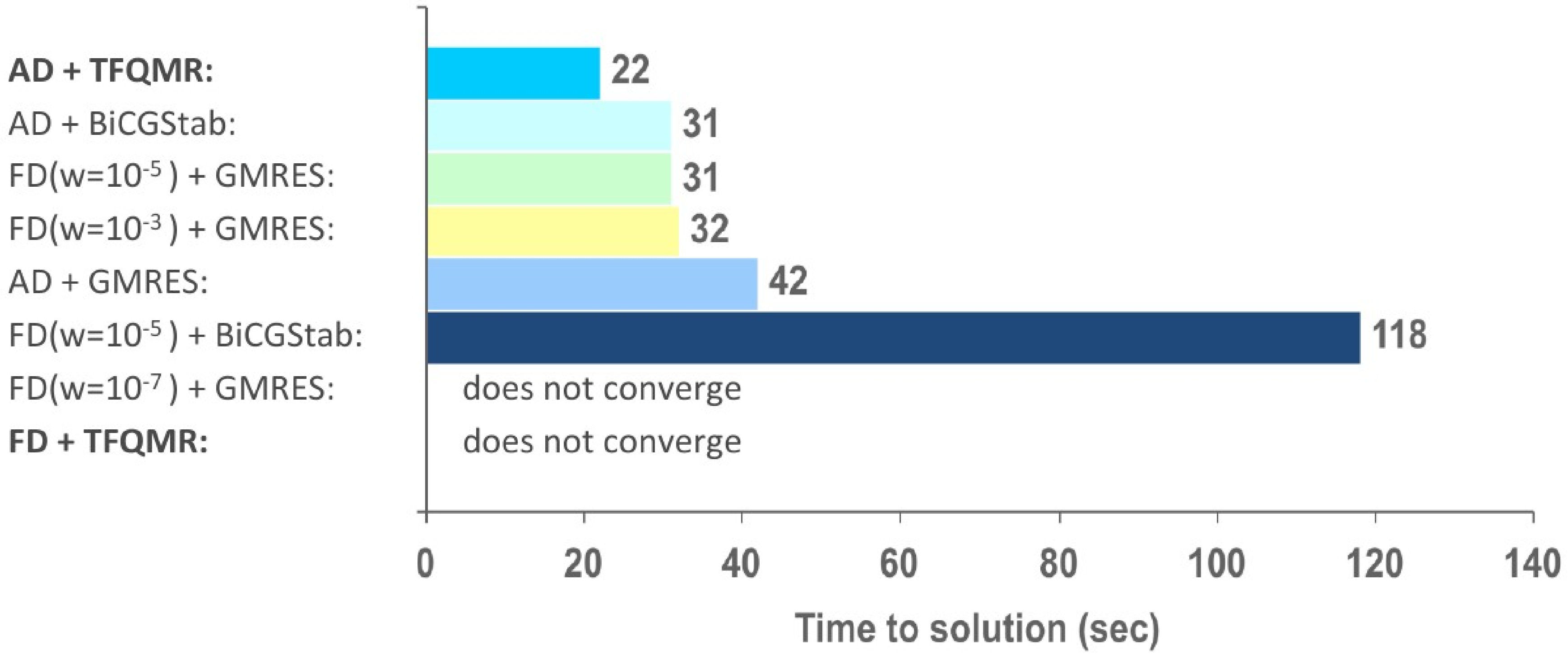


After



Application: solution of nonlinear PDEs

- Jacobian-free Newton-Krylov solution of model problem (driven cavity)



AD = automatic differentiation

FD = finite differences

W = noise estimate for Brown-Saad

Points of nondifferentiability

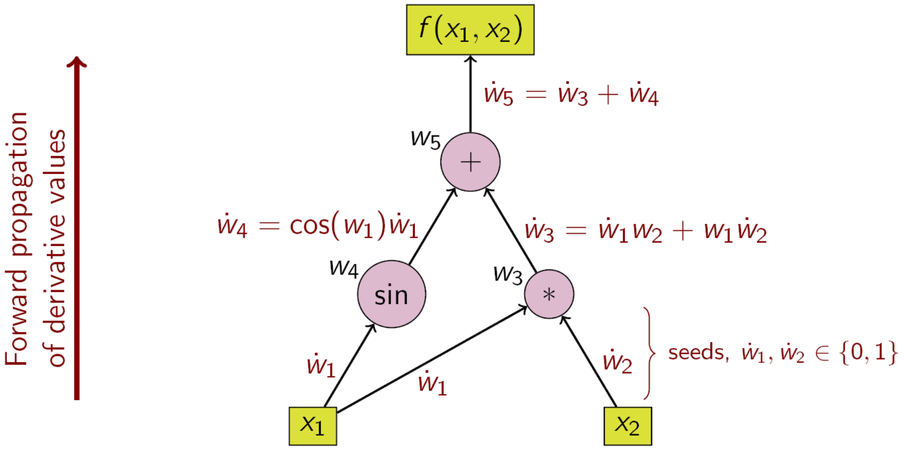
- Due to intrinsic functions
  - Several intrinsic functions are defined at points where their derivatives are not, e.g.:
    - abs(x), sqrt(x) at x=0
    - max(x,y) at x=y
  - Requirements:
    - Record/report exceptions
    - Optionally, continue computation using some generalized gradient
  - ADIFOR/ADIC approach
    - User-selected reporting mechanism
    - User-defined generalized gradients, e.g.:
      - [1.0,0.0] for max(x,0)
      - [0.5,0.5] for max(x,y)
    - Various ways of handling
      - Verbose reports (file, line, type of exception)
      - Terse summary (like IEEE flags)
      - Ignore
- Due to conditional branches
  - May be able to handle using trust regions

Conclusion

- If $f : \mathbb{R}^N \to \mathbb{R}^M$ is a chain of smooth elementary operations.

Then disassembly + chain rule + reassembly = AD.

- Source code transformation:
  Good for forward and reverse
  Hard to implement
  Memory intensive



- Operator Overloading:
  Good for forward mode
  Easier to implement
  Memory efficient

- Strengths of AD:
  Exact, Fast, No human error

- Pitfall of AD:
  Non-differentiability