Automatic Differentiation for Scientific Discovery and Design: Useful, Elegant, and Underutilized

Nick McGreivy

PhD Candidate
Program in Plasma Physics
Princeton University

January 20th 2021

Twitter for SciComm: Useless, Inelegant, and Overutilized

Today's talk:

Connections between AD and adjoint method:

Automatic Differentiation (AD)

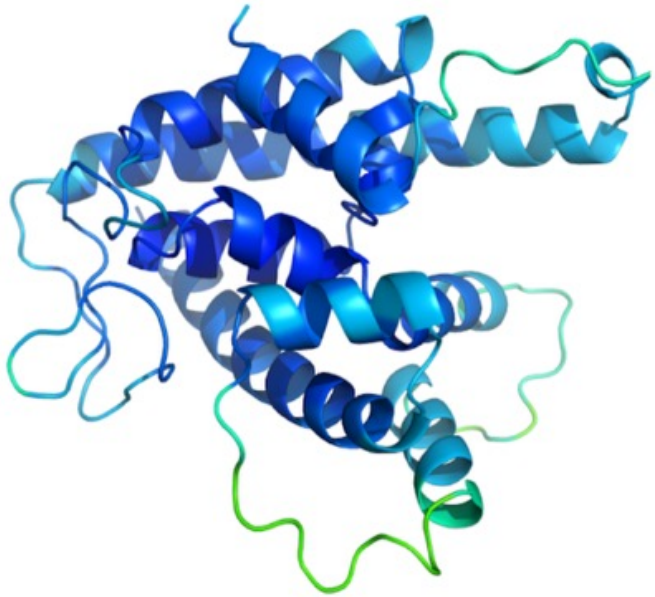
1. Useful
2. Elegant
3. Underutilized

Automatic Differentiation (AD)

1. Useful
2. Elegant
3. Underutilized

Protein folding: solve for f

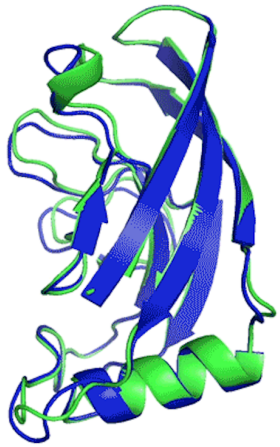
$$f\left(\text{Protein sequence}\right) = \text{3D structure}$$



Protein folding is a prediction problem: the goal is to learn a function f which, given a 1D sequence of amino acids, is able to predict the 3D structure of the protein.

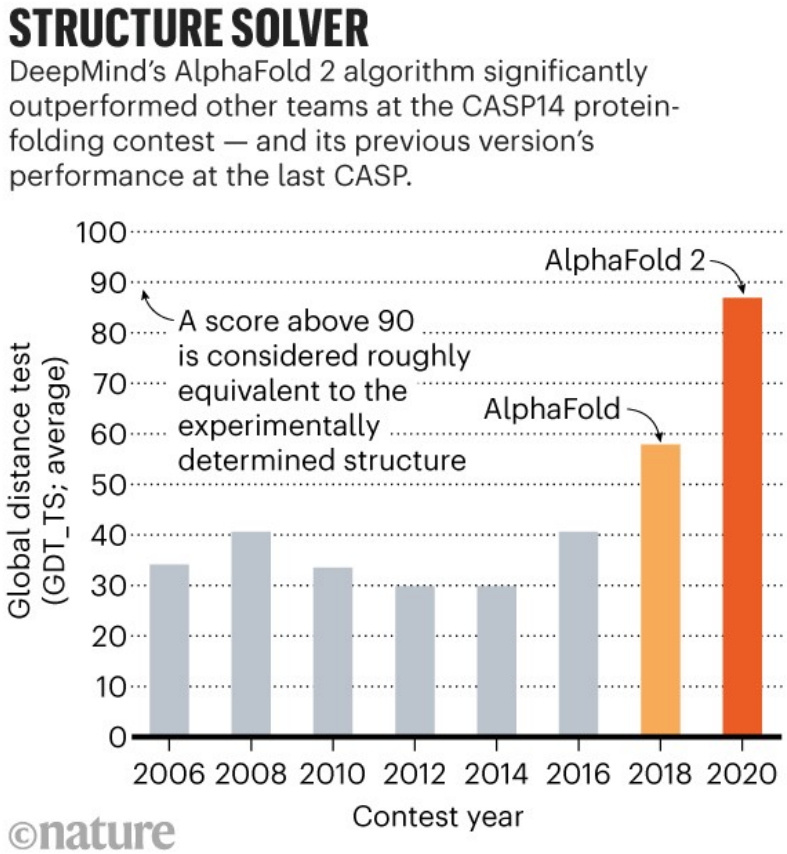
Alphafold 2 (2020)

T1037 / 6vr4
90.7 GDT
(RNA polymerase domain)



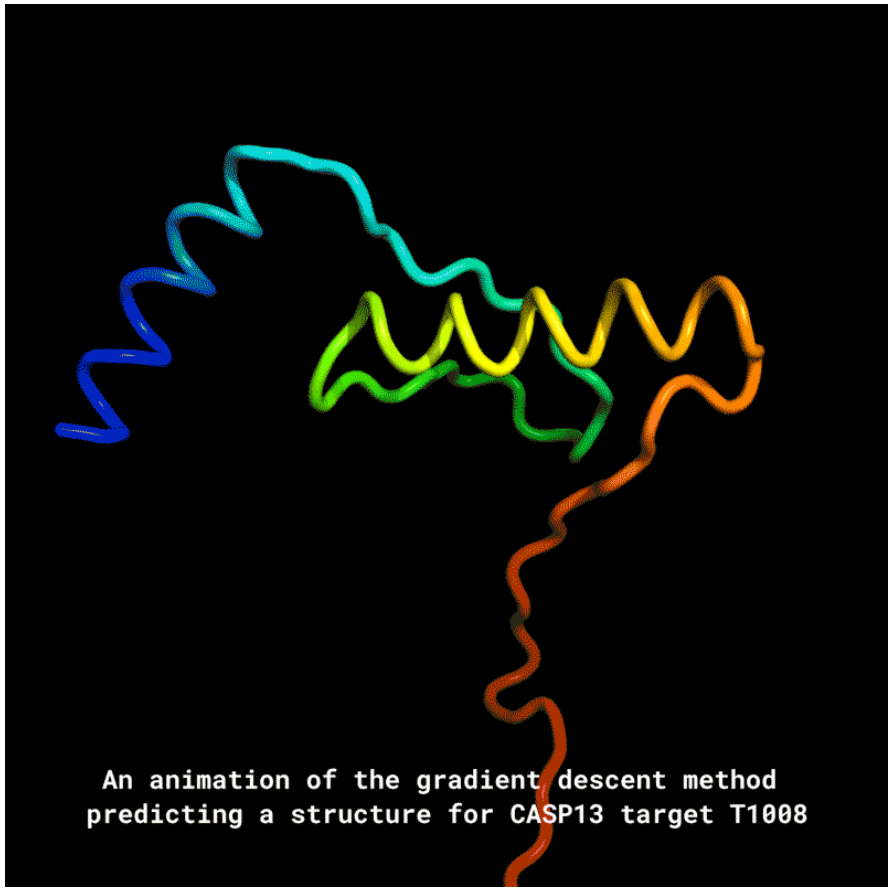
T1049 / 6y4f
93.3 GDT
(adhesin tip)

Experimental result
Computational prediction

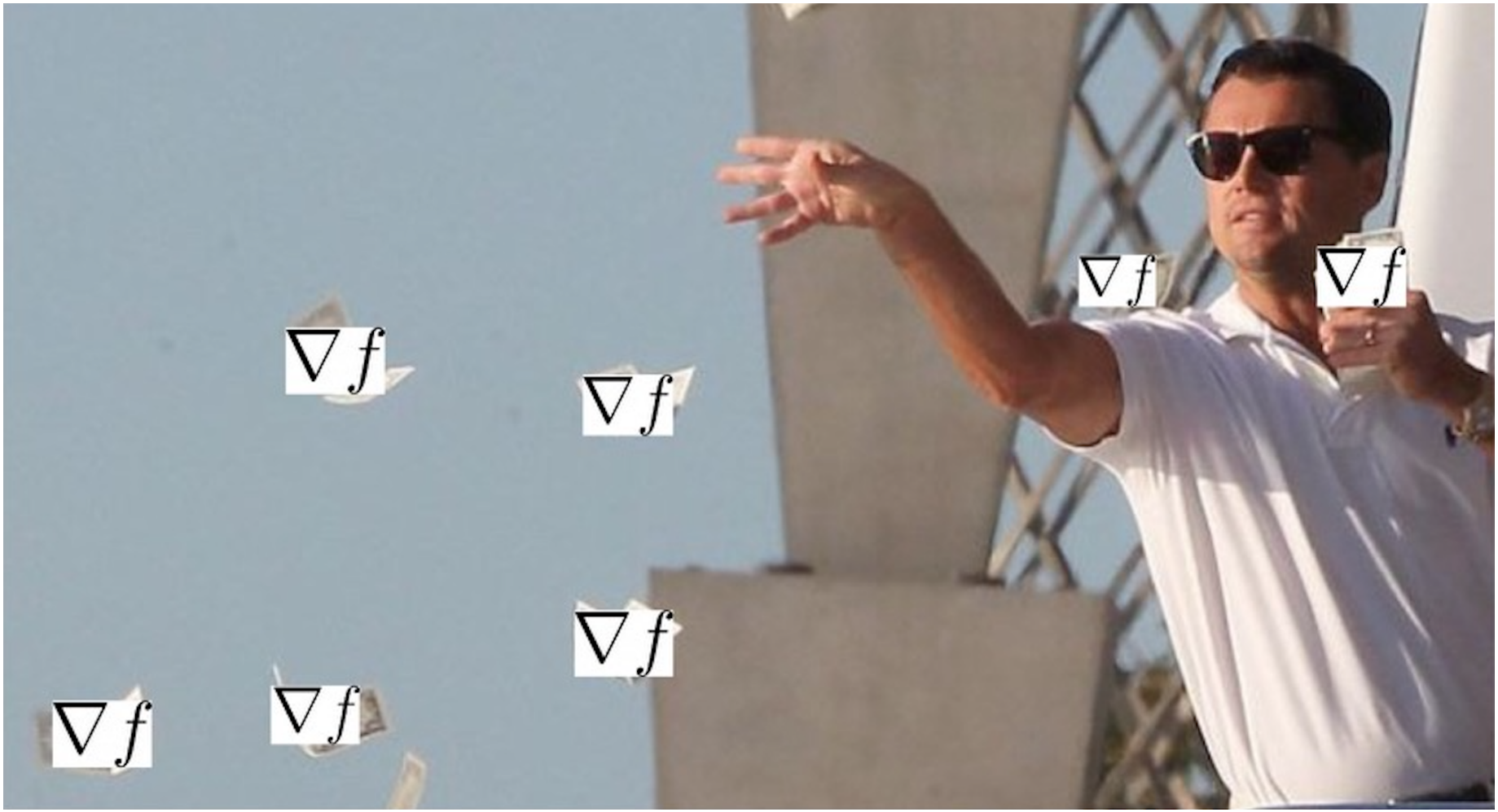


"It's a breakthrough of the first order, certainly one of the most significant scientific results of my lifetime." -Mohammed AlQuraishi, Assistant Professor of Systems Biology at Columbia University

Gradient descent on gradient descent



Useful: to feel like Leonardo DiCaprio



Don't believe me, believe these random people on twitter

Akshay Agrawal
@akshaykagrawal

There have been lots of predictions that deep learning will revolutionize many fields. What's discussed less is how large an impact automatic differentiation (@PyTorch, @TensorFlow, JAX, etc) has had.

9:16 AM · Nov 3, 2020 · Twitter Web App

gully
@gully_

The enduring gift of Machine Learning/AI to the astronomy community is an underdog:

High quality automatic differentiation frameworks.

Autodiff is so much more widely applicable than we have seen so far, and I'm jazzed to see where it goes in the future.

Automatic Differentiation (AD)

1. Useful
2. Elegant
3. Underutilized

AD is Elegant

- In theory: fundamentally, AD is based on the chain rule. We'll see how this allows us to compute the derivatives of arbitrary functions composed of known building blocks.
- In practice: the high-quality AD tools developed for ML research have made taking gradients simple and effortless. I'll demonstrate this with a simple stellarator coil design code.

I am going to simplify the theory somewhat, as in practice AD uses either "forward mode" or "reverse mode". Those important concepts are not discussed in this talk.

Elegant in theory: the multivariate chain rule

Suppose we have a multivariate function $\boldsymbol{f}$ which has input $\boldsymbol{x}$:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{u}(\boldsymbol{v}(\boldsymbol{x}))$$

The Jacobian is given by the chain rule, which multiplies elementary Jacobian matrices:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{v}} \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}$$

In the language of AD, $\boldsymbol{u}$ and $\boldsymbol{v}$ are primitive functions and $\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{v}}$ and $\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}$ are elementary partial derivatives. $\boldsymbol{f}$ is built by composing primitive functions, and $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ is computed by multiplying elementary partial derivatives.

Elegant in theory: AD is about composing derivatives

Suppose I create a library of primitive functions A, B, and C. I define how each primitive function is evaluated, and I define the derivative of each primitive function analytically:

```
def A:
    def eval(x):
        return ... # A
    def deriv(x):
        return ... # dA/dx
def B:
    def eval(x):
        return ... # B
    def deriv(x):
        return ... # dB/dx
def C:
    def eval(x):
        return ... # C
    def deriv(x):
        return ... # dC/dx
```

Elegant in theory: AD is about composing derivatives

Suppose I create a library of primitive functions A, B, and C. I define how each primitive function is evaluated, and I define the derivative of each primitive function analytically:

```
def A:
    def eval(x):
        return ... # A
    def deriv(x):
        return ... # dA/dx
def B:
    def eval(x):
        return ... # B
    def deriv(x):
        return ... # dB/dx
def C:
    def eval(x):
        return ... # C
    def deriv(x):
        return ... # dC/dx
```

An AD software package uses a library, such as the one on the left, to compute the derivative of any function which is made up of the primitive functions A, B, and C; e.g.

$$f_1 = C(B(A(y)))$$

$$\frac{df_1}{dy} = \left(\frac{dC}{dx}\right)_{x=B(A(y))} \left(\frac{dB}{dx}\right)_{x=A(y)} \left(\frac{dA}{dx}\right)_{x=y}$$

is one possible function composition whose derivative AD computes. But our AD package can compute the derivative of any function made up of these building blocks; e.g.

$$f_1 = B(\ldots C(y))$$

$$\frac{df_2}{dy} = \left(\frac{dB}{dx}\right)_{x=B(\ldots C(y))} \cdots \left(\frac{dC}{dx}\right)_{x=y}$$

Elegant in practice: coil design in 25 lines of code

FOCUS (Caoxiang Zhu et al 2018 Nucl. Fusion) coil representation:

$$\boldsymbol{r}^i(\theta) = \sum_{m=0}^{N_F-1} \boldsymbol{R}^i_{cm} \cos(m\theta) + \boldsymbol{R}^i_{sm} \sin(m\theta)$$

The objective function is the quadratic flux:

$$f(\boldsymbol{p}) = \int_S \left(\boldsymbol{B}(\boldsymbol{p}) \cdot \hat{\boldsymbol{n}}\right)^2 dA$$

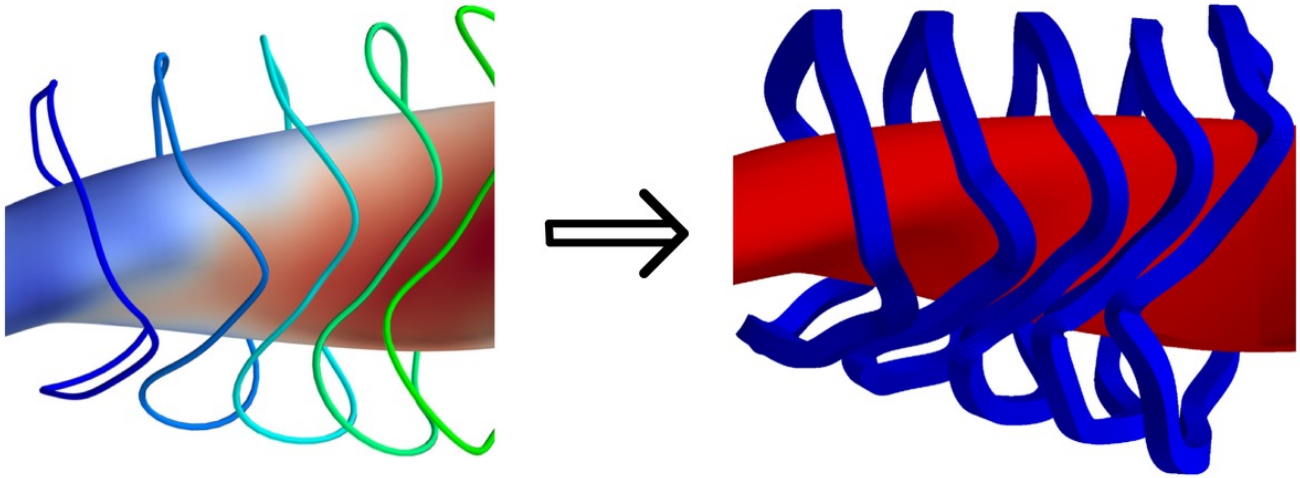
Perform gradient-based optimization:

$$\boldsymbol{p}' = \boldsymbol{p} - \eta \boldsymbol{\nabla} f$$

Automatic Differentiation (AD)

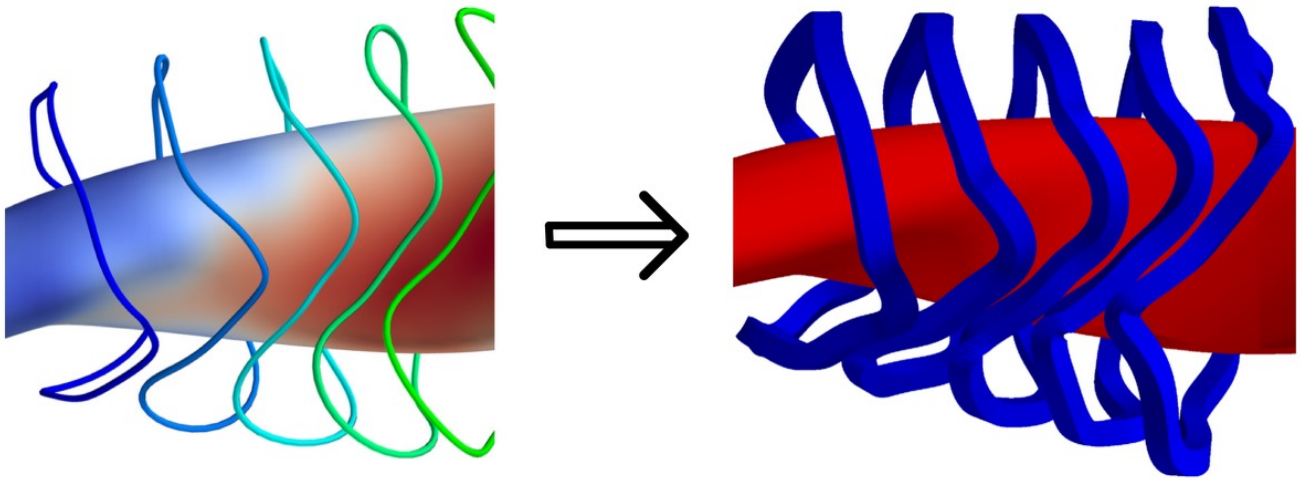
1. Useful
2. Elegant
3. Underutilized

Underutilized in coil design: FOCUSADD and Finite Build



N. McGreivy et al 2021 Nucl. Fusion

Underutilized in coil design: FOCUSADD and Finite Build

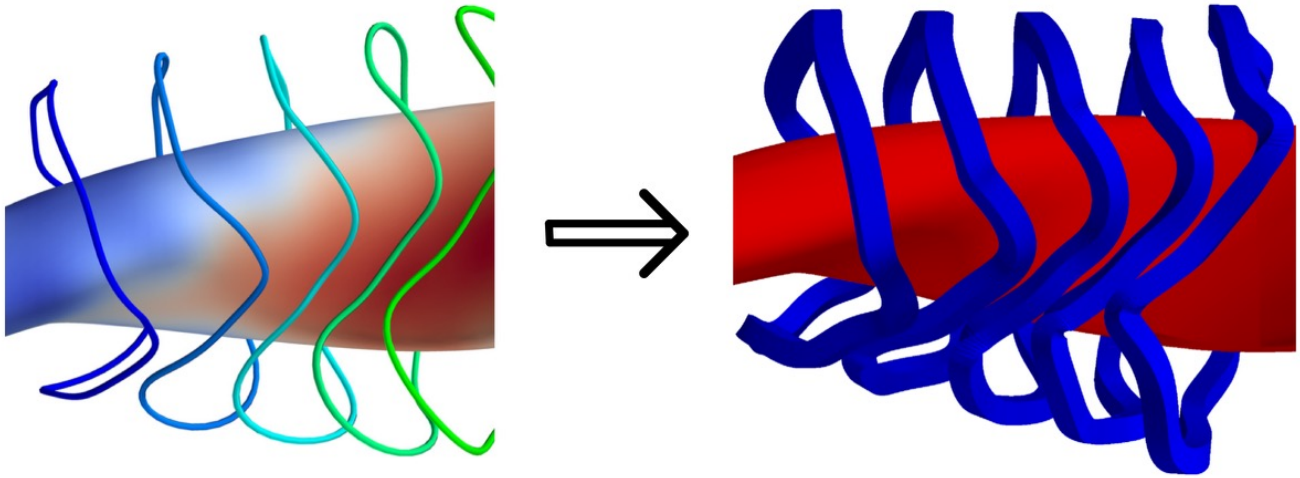


N. McGreivy et al 2021 Nucl. Fusion

New paradigm in coil design: whatever we can compute, we can optimize

Using AD has two major advantages: (1) Instead of worrying about *how* to compute gradients of an objective function, we only have to worry about *what* objective function we want to optimize. (2) We can rapidly iterate on different objective functions, meaning we can explore a much larger optimization space.

Underutilized in coil design: FOCUSADD and Finite Build



N. McGreivy et al 2021 Nucl. Fusion

New paradigm in coil design: whatever we can compute, we can optimize

Using AD has two major advantages: (1) Instead of worrying about *how* to compute gradients of an objective function, we only have to worry about *what* objective function we want to optimize. (2) We can rapidly iterate on different objective functions, meaning we can explore a much larger optimization space.

A new paradigm in coil design is good, but we can do better...

Single-stage optimization is the next big kahuna

Combined coil-plasma optimization requires passing gradients between plasma solvers and coil codes:

$$\frac{df}{d\text{coil}} = \frac{\partial f}{\partial \text{coil}} + \frac{\partial f}{\partial \text{plasma}} \frac{\partial \text{plasma}}{\partial \text{coil}}$$

(A. Giuliani et. al., "Single-stage gradient-based stellarator coil design.", arXiv:2010.02033)

Single-stage optimization is the next big kahuna

Combined coil-plasma optimization requires passing gradients between plasma solvers and coil codes:

$$\frac{df}{d\text{coil}} = \frac{\partial f}{\partial \text{coil}} + \frac{\partial f}{\partial \text{plasma}} \frac{\partial \text{plasma}}{\partial \text{coil}}$$

(A. Giuliani et. al., "Single-stage gradient-based stellarator coil design.", arXiv:2010.02033)

It's not my job to design SIMSOPT, but if it were . . .

I'd implement SIMSOPT as a collection of primitive functions using AD.

Single-stage optimization is the next big kahuna

Combined coil-plasma optimization requires passing gradients between plasma solvers and coil codes:

$$\frac{df}{d\text{coil}} = \frac{\partial f}{\partial \text{coil}} + \frac{\partial f}{\partial \text{plasma}} \frac{\partial \text{plasma}}{\partial \text{coil}}$$

(A. Giuliani et. al., "Single-stage gradient-based stellarator coil design.", arXiv:2010.02033)

It's not my job to design SIMSOPT, but if it were . . .

I'd implement SIMSOPT as a collection of primitive functions using AD.

AD makes passing Jacobians between compositions of primitive functions easy. The strategy is to implement a collection of modular primitive functions. Under this framework, adjoint methods are implemented as primitive functions. Primitives are then used as building blocks which are composed as needed to form single-stage (or two-stage) optimizations.

Thanks! Questions?