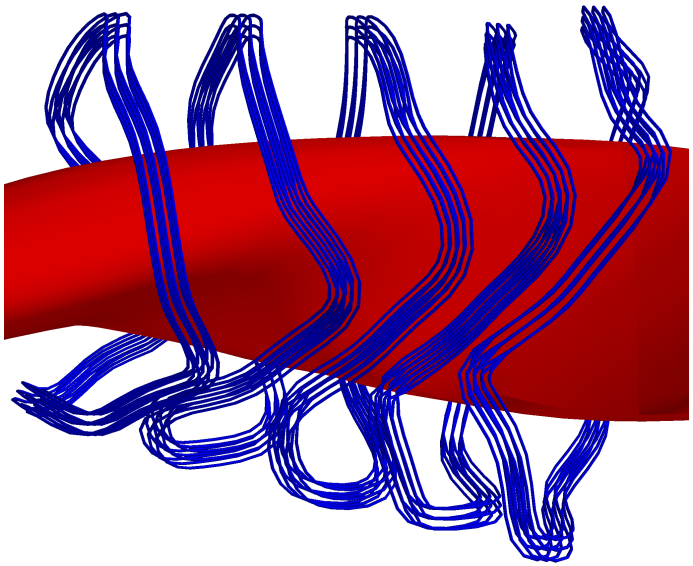
Automatic Differentiation for Stellarator Design

Nick McGreivy

PhD Candidate
Program in Plasma Physics
Princeton University



August 21st 2020

Automatic Differentiation (AD)

Also known as algorithmic differentiation or computational differentiation

- Automatic Differentiation (AD) is a technology for automatically computing the exact numerical derivatives of any differentiable function $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$, including arbitrarily complex simulations, represented by a computer program.
- To compute automatic derivatives of a function, you need to program that function using an AD software tool.

Why am I giving this talk?

Because AD is useful and you might want to use it in your research!

- Stellarator optimization (coils, magnetic fields)
- Understanding tolerances and sensitivities
- Linear perturbation analysis
- Anywhere else derivatives need to be computed

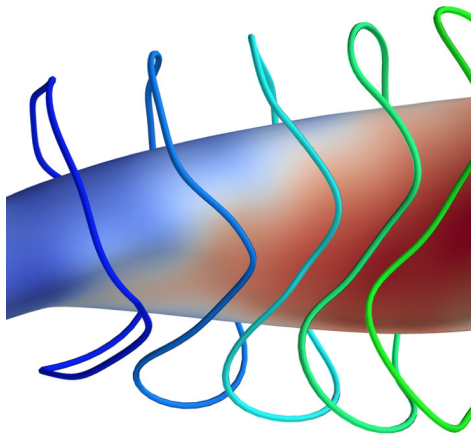
Part I:
Stellarator Coil Design in 25 lines of code

FOCUS: Coil Representation

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

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

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



$$\boldsymbol{p} = \left\{ X^i_{cm}, X^i_{sm}, Y^i_{cm}, Y^i_{sm}, Z^i_{cm}, Z^i_{sm} \right\}_{i=1,m=0}^{i=N_C,m=N_F-1}$$

(C. Zhu, S. R. Hudson, Y. Song, Y. Wan 2017)

Stellarator Coil Design as an Optimization Problem

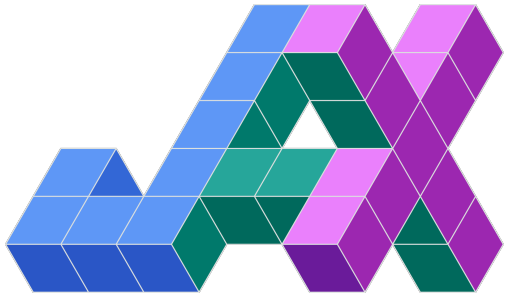
$$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{p})$$

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

$$\boldsymbol{B} = \boldsymbol{B}_{\text{coils}} + \cancelto{0}{\boldsymbol{B}_{\text{plasma}}}$$

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

JAX



- JAX is composable transformations of Python+NumPy programs: differentiate, vectorize, JIT to GPU/TPU, and more.

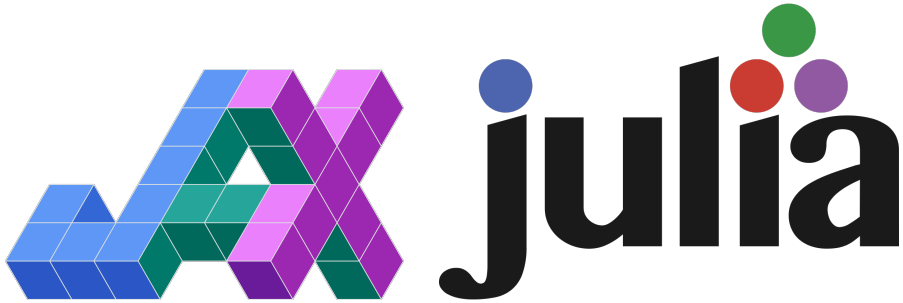
Go to implementation

Recap

We've written a parallelized, JIT-compiled FOCUS-like stellarator coil design code in about 25 lines of Python. This code:

- is vectorized with `vmap`
- does automatic differentiation with `grad`
- JIT-compiles to CPU or GPU with `jit`
- Parallelizes across multiple CPUs/GPUs with `pmap`

Part II: Discussion of automatic differentiation (AD)



Stan Math Library

differentiable C++ for linear algebra & probability

Taking a derivative

$$F : \mathbb{R}^n \to \mathbb{R}$$

$F :$  $\mapsto$ 

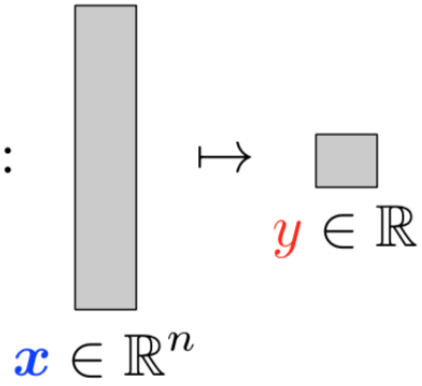
$\boldsymbol{x} \in \mathbb{R}^n$ $\quad y \in \mathbb{R}$

$$F = D \circ C \circ B \circ A \qquad y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

Taking a derivative

$$F : \mathbb{R}^n \to \mathbb{R}$$



$$F = D \circ C \circ B \circ A \qquad y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

Primitive operations

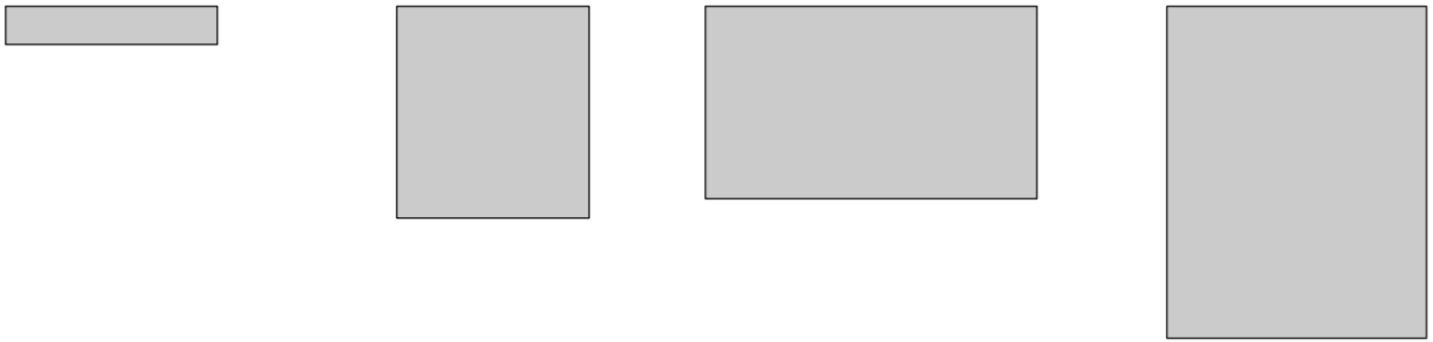
The functions A, B, C, and D should be understood as "primitive operations". These could be simple primitives like `sin`, `exp`, `div`, etc, but they could also be complicated primitives like `fft`, `odeint`, `solve` or even custom primitives like `nicksfunc`.

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \qquad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b}) \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = B'(\boldsymbol{a}) \qquad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = A'(\boldsymbol{x})$$

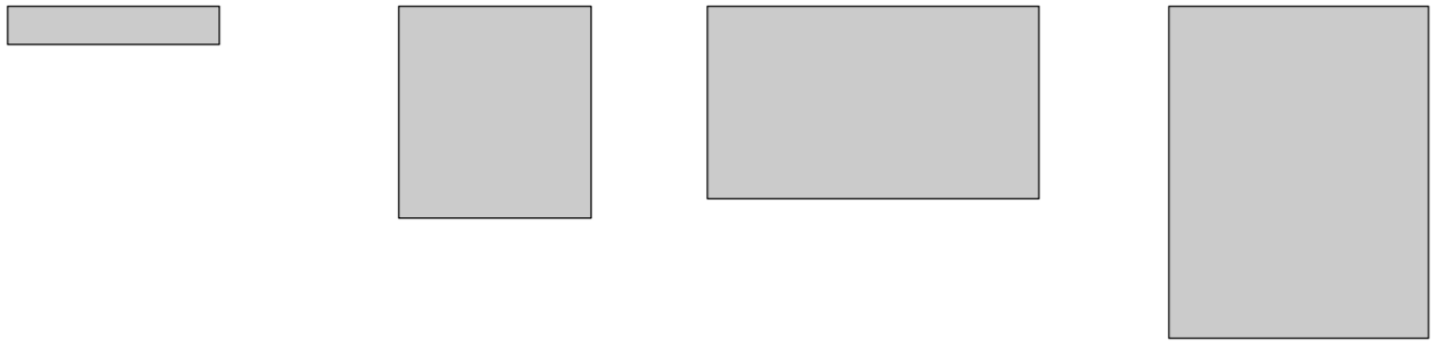


$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \qquad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b}) \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = B'(\boldsymbol{a}) \qquad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = A'(\boldsymbol{x})$$



Order matters!

If $F : \mathbb{R}^n \to \mathbb{R}^m$ takes time $\mathcal{O}(T)$ to compute, then multiplying from right to left takes time $\mathcal{O}(nT)$ and multiplying from left to right takes time $\mathcal{O}(mT)$.

Jacobian-vector product (JVP)

A Jacobian-vector (JVP) product with one column of an identity matrix gives one column of the Jacobian matrix. E.g., for a function $\mathbb{R}^n \to \mathbb{R}^m$, this is

$$\boldsymbol{F}'(\boldsymbol{x})\boldsymbol{v} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \vdots \\ \frac{\partial y_m}{\partial x_1} \end{bmatrix}$$

E.g., for a scalar function $\mathbb{R}^n \to \mathbb{R}$, a JVP gives a scalar.

$$\boldsymbol{F}'(\boldsymbol{x})\boldsymbol{v} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \frac{\partial y}{\partial x_1}$$

Vector-Jacobian product (VJP)

A vector-Jacobian product (VJP) with one row of an identity matrix gives one row of the Jacobian matrix. E.g., for a function $\mathbb{R}^n \to \mathbb{R}^m$, this is

$$\boldsymbol{v}^T \boldsymbol{F}'(\boldsymbol{x}) = \begin{bmatrix} 1 & 0 \dots 0 \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \vdots \\ \frac{\partial y_1}{\partial x_n} \end{bmatrix}$$

E.g., for a scalar function $\mathbb{R}^n \to \mathbb{R}$, a VJP gives the gradient.

$$\boldsymbol{v}^T \boldsymbol{F}'(\boldsymbol{x}) = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$$

AD: Forward and Reverse Modes

Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. $\boldsymbol{v}$ is a "tangent vector".

$$F'(\boldsymbol{x})\ \boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v}\right)\right)\right)$$

Reverse mode: vector-Jacobian products (VJPs), build Jacobian one row at a time. $\boldsymbol{v}$ is a "cotangent vector".

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \left(\left(\left(\boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\right)\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\right)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)\Bigg)$$

These are the "pushforward" and "pullback" maps from differential geometry, composed with tangent and cotangent vectors respectively.

AD: Forward and Reverse Modes

Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. $\boldsymbol{v}$ is a "tangent vector".

$$F'(\boldsymbol{x})\ \boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v}\right)\right)\right)$$

Reverse mode: vector-Jacobian products (VJPs), build Jacobian one row at a time. $\boldsymbol{v}$ is a "cotangent vector".

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \left(\left(\left(\boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\right)\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\right)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)$$

These are the "pushforward" and "pullback" maps from differential geometry, composed with tangent and cotangent vectors respectively.

What makes it "automatic"?

The key is that your AD library knows how to compute Jacobian-products for each primitive operation. This allows it to compose JVPs and VJPs for each primitive and compute the total JVP/VJP.

AD: Forward and Reverse Modes

Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. $\boldsymbol{v}$ is a "tangent vector".

$$F'(\boldsymbol{x})\ \boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v}\right)\right)\right)$$

Reverse mode: vector-Jacobian products (VJPs), build Jacobian one row at a time. $\boldsymbol{v}$ is a "cotangent vector".

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \left(\left(\left(\boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\right)\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\right)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)$$

These are the "pushforward" and "pullback" maps from differential geometry, composed with tangent and cotangent vectors respectively.

What if I want the full Jacobian, not a JVP or VJP?

AD tools can compute the full Jacobian as well, as well as higher derivatives. However, JVPs and VJPs are the building blocks of AD tools. I highly recommend the "JAX autodiff cookbook" for further explanation.

Fundamentals of AD: A summary

- Suppose $\boldsymbol{y} = f(\boldsymbol{x})$, where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^m$.
- If f is composed of primitive operations implemented by the AD tool, then if I give it an $\boldsymbol{x}$, AD computes the numerical value of the derivative at $\boldsymbol{x}$.
- Forward mode AD, by computing JVPs, can compute the Jacobian of f in time $\mathcal{O}(n)$.
- Reverse mode AD, by computing VJPs, can compute the Jacobian of f in time $\mathcal{O}(m)$.

Why is AD useful?

1. Simplicity

- Finding and programming analytic derivatives is time-consuming and error-prone.

Why is AD useful?

1. Simplicity

- Finding and programming analytic derivatives is time-consuming and error-prone.

2. Ideal for gradient-based optimization

- Reverse mode AD computes the gradient of a scalar function in time $\mathcal{O}(1)$. This is as efficient as the best analytic methods.

Why is AD useful?

1. Simplicity

- Finding and programming analytic derivatives is time-consuming and error-prone.

2. Ideal for gradient-based optimization

- Reverse mode AD computes the gradient of a scalar function in time $\mathcal{O}(1)$. This is as efficient as the best analytic methods.

3. Effortless gradients

- Easy to rapidly prototype new ideas and objectives.

Why would you not compute derivatives with AD?

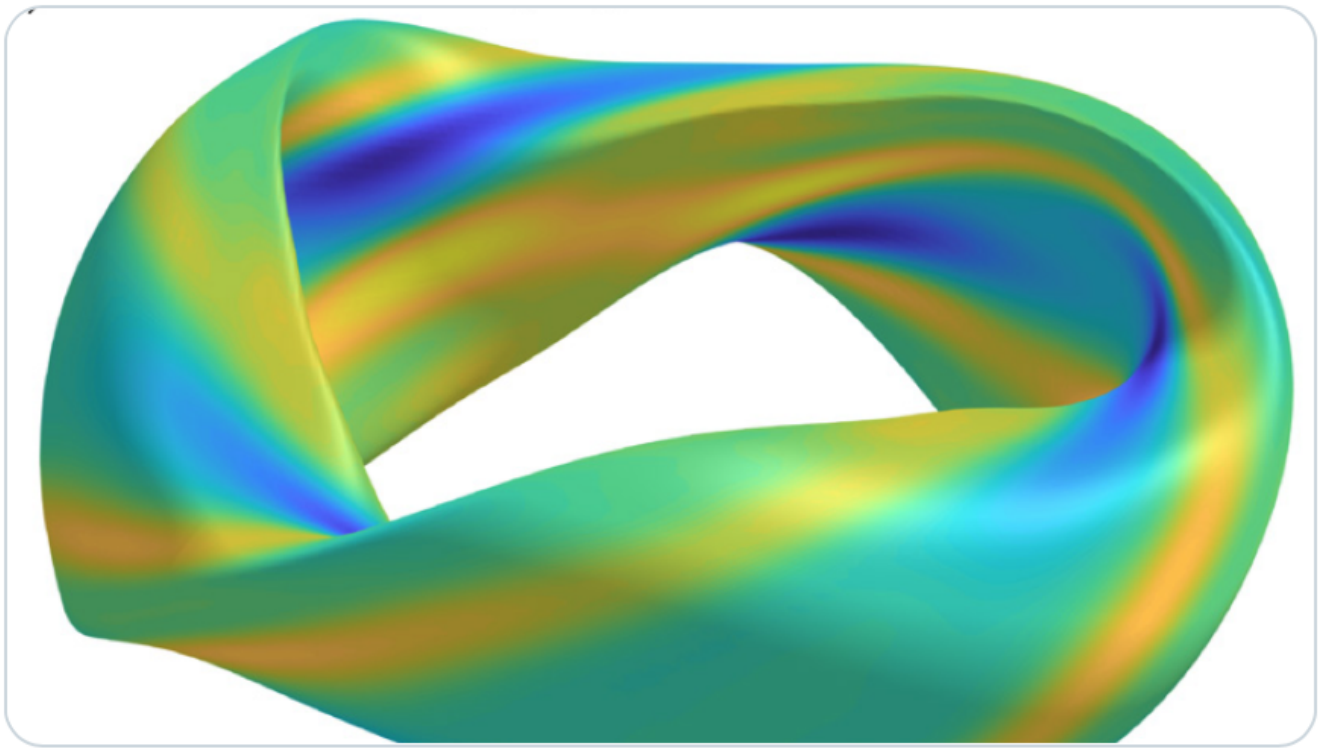
- You have a massive legacy code which you can't rewrite.
- You've already written your code and it efficiently computes analytic derivatives, so why change it? (Next time though, use AD...)
- You need highly specialized numerical routines which are not implemented by an AD tool. (However, you can always wrap a numerical routine with a custom `Primitive` if you really need it.)
- Memory cost of reverse mode can be large (checkpointing helps reduce this)
- "The automatic generation of optimal (in terms of robustness and efficiency) adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing." -Uwe Naumann (2011)

Nick McGreivy @NMcgreivy · Jul 22

The adjoint method for PDE-constrained optimization: the conclusions of my 9-month struggle to understand the word "adjoint".

I'll be talking about how automatic differentiation (AD) can help us better understand the adjoint method, and vice versa. Let's get started!

1/



2 43 141

Summary

- Automatic differentiation allows for the efficient computation of derivatives for stellarator design.
- AD is about (i) primitive operations and (ii) JVPs/VJPs.
- AD libraries know how to compute JVPs and VJPs for each primitive operation in the library.

Thank you

Additional Slides

Review: The adjoint method

Differentiating under constraints requires a linear solve

$$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p}) \text{ s.t. } \boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0 \text{ defines } \boldsymbol{u}. \text{ Linearize around } \boldsymbol{u}.$$

Review: The adjoint method

Differentiating under constraints requires a linear solve

$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p})$ s.t. $\boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0$ defines $\boldsymbol{u}$. Linearize around $\boldsymbol{u}$.

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}}$$

Review: The adjoint method

Differentiating under constraints requires a linear solve

$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p})$ s.t. $\boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0$ defines $\boldsymbol{u}$. Linearize around $\boldsymbol{u}$.

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{g}}{d\boldsymbol{p}} = 0 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$

Review: The adjoint method

Differentiating under constraints requires a linear solve

$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p})$ s.t. $\boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0$ defines $\boldsymbol{u}$. Linearize around $\boldsymbol{u}$.

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{g}}{d\boldsymbol{p}} = 0 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \Rightarrow \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$

Review: The adjoint method

Differentiating under constraints requires a linear solve

$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p})$ s.t. $\boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0$ defines $\boldsymbol{u}$. Linearize around $\boldsymbol{u}$.

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{g}}{d\boldsymbol{p}} = 0 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \Rightarrow \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} - \frac{\partial f}{\partial \boldsymbol{u}} \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$

Review: The adjoint method

Differentiating under constraints requires a linear solve

$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p})$ s.t. $\boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0$ defines $\boldsymbol{u}$. Linearize around $\boldsymbol{u}$.

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{g}}{d\boldsymbol{p}} = 0 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \Rightarrow \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} - \frac{\partial f}{\partial \boldsymbol{u}} \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \Rightarrow \frac{\partial f}{\partial \boldsymbol{u}} \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} = -\boldsymbol{\lambda}^T$$

Review: The adjoint method

Differentiating under constraints requires a linear solve

$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} f(\boldsymbol{u}(\boldsymbol{p}), \boldsymbol{p})$ s.t. $\boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = 0$ defines $\boldsymbol{u}$. Linearize around $\boldsymbol{u}$.

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \frac{\partial f}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}}$$

$$\frac{d\boldsymbol{g}}{d\boldsymbol{p}} = 0 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \Rightarrow \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$

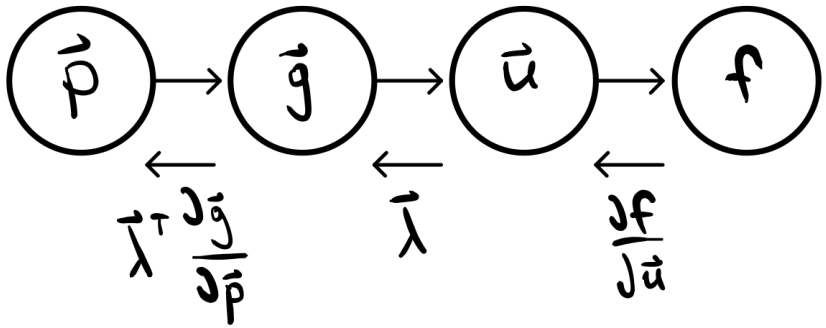
$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} - \frac{\partial f}{\partial \boldsymbol{u}} \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \Rightarrow \frac{\partial f}{\partial \boldsymbol{u}} \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^{-1} = -\boldsymbol{\lambda}^T$$

$$\boxed{\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}} \text{ where } \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^T \boldsymbol{\lambda} = -\frac{\partial f}{\partial \boldsymbol{u}}$$

Understanding the adjoint method with AD

The adjoint method computes the VJP of a constraint equation

$$\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}$$



- The constraint equation $\boldsymbol{g} = 0$ is a primitive operation taking $\boldsymbol{p}$ as input and outputting $\boldsymbol{u}$.
- $\frac{\partial f}{\partial \boldsymbol{u}}$ is the cotangent vector $\boldsymbol{v}$ and $\boldsymbol{\lambda}$ is the (cotangent) vector in the VJP $\boldsymbol{\lambda} = \boldsymbol{v}^T \boldsymbol{J}$.
- The equation $\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^T \boldsymbol{\lambda} = -\frac{\partial f}{\partial \boldsymbol{u}}$ is the adjoint equation for the primitive $\boldsymbol{g} = 0$.

What even is an adjoint equation?

Adjoint equation $\neq$ adjoint method

An adjoint equation is an equation which computes the VJP of a given primitive operation.

Suppose f_i is a primitive operation which takes input $\boldsymbol{x}_1 \in \mathbb{R}^a$ and has output $\boldsymbol{x}_2 \in \mathbb{R}^b$. Then the adjoint equation for f_i takes a cotangent vector $\boldsymbol{v}_2 \in \mathbb{R}^b$ as input and outputs a cotangent vector $\boldsymbol{v}_1 = \boldsymbol{v}_2^T \boldsymbol{J}_i \in \mathbb{R}^a$.

What even is an adjoint equation?

Adjoint equation $\neq$ adjoint method

An adjoint equation is an equation which computes the VJP of a given primitive operation.

Suppose f_i is a primitive operation which takes input $\boldsymbol{x}_1 \in \mathbb{R}^a$ and has output $\boldsymbol{x}_2 \in \mathbb{R}^b$. Then the adjoint equation for f_i takes a cotangent vector $\boldsymbol{v}_2 \in \mathbb{R}^b$ as input and outputs a cotangent vector $\boldsymbol{v}_1 = \boldsymbol{v}_2^T \boldsymbol{J}_i \in \mathbb{R}^a$.

A different interpretation of AD via the adjoint method

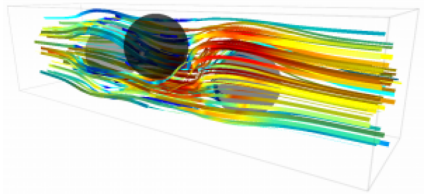
Reverse mode AD libraries know how to setup and solve adjoint equations and compose their solutions for each primitive operation in the library. For example, the adjoint equation for the primitive $y = \sin(x)$ is $dx = dy * \cos(x)$.

AD of FEM code: dolfin-adjoint & FEniCS

The dolfin-adjoint project automatically derives the discrete adjoint and tangent linear models from a forward model written in the Python interface to FEniCS and Firedrake.

dolfin-adjoint

FEniCS enables users to quickly translate scientific models into efficient finite element code. Firedrake is an automated system for the solution of partial differential equations using the finite element method (FEM).



Firedrake

dolfin-adjoint is not an AD library!

dolfin-adjoint is similar to an AD library, but operates at a much higher level of abstraction. It works because it focuses only on FEM code.

AD and the adjoint method for nonlinear equations

The adjoint method allows us to differentiate under the constraint $\boldsymbol{g} = 0$.

$$\boxed{\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}} \text{ where } \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^T \boldsymbol{\lambda} = -\frac{\partial f}{\partial \boldsymbol{u}}$$

AD and the adjoint method for nonlinear equations

The adjoint method allows us to differentiate under the constraint $\boldsymbol{g} = 0$.

$$\boxed{\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}} \text{ where } \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^T \boldsymbol{\lambda} = -\frac{\partial f}{\partial \boldsymbol{u}}$$

Suppose the equation $\boldsymbol{g} = 0$ is solved iteratively with Newton's method:

$$\boldsymbol{g}(\boldsymbol{u}^i, \boldsymbol{p}) + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \cdot (\boldsymbol{u}^{i+1} - \boldsymbol{u}^i) = 0 \Rightarrow \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \delta \boldsymbol{u} = -\boldsymbol{g}(\boldsymbol{u}^i, \boldsymbol{p})$$

AD and the adjoint method for nonlinear equations

The adjoint method allows us to differentiate under the constraint $\boldsymbol{g} = 0$.

$$\boxed{\frac{df}{d\boldsymbol{p}} = \frac{\partial f}{\partial \boldsymbol{p}} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}} \text{ where } \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right)^T \boldsymbol{\lambda} = -\frac{\partial f}{\partial \boldsymbol{u}}$$

Suppose the equation $\boldsymbol{g} = 0$ is solved iteratively with Newton's method:

$$\boldsymbol{g}(\boldsymbol{u}^i, \boldsymbol{p}) + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \cdot (\boldsymbol{u}^{i+1} - \boldsymbol{u}^i) = 0 \Rightarrow \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \delta \boldsymbol{u} = -\boldsymbol{g}(\boldsymbol{u}^i, \boldsymbol{p})$$

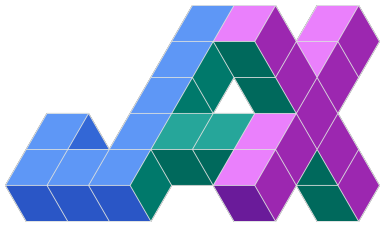
The Newton solver already computes $\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}$! So by solving $\boldsymbol{g} = 0$, we have everything we need to perform the adjoint method. A good AD tool will set up and solve this adjoint equation automatically.

Properties of a great AD library

- Feels like native programming
- Intuitive API
- Full set of primitive operations implemented
- Efficient linear algebra (modern implementations wrap Eigen, numpy, or JIT-compile)
- Control flow support (loops, if statements, recursion)
- Forward and reverse
- Higher-order derivatives
- GPU support
- Checkpointing
- Differentiation through linear, non-linear solves with the adjoint method
- User-defined primitives
- MPI parallelization

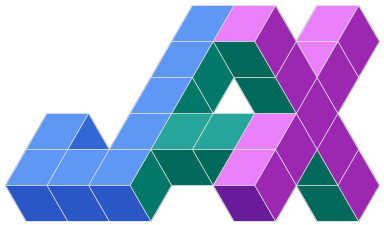
JAX: **J**ust **A**fter e**X**ecution (Python)

JAX is Numpy and Scipy with composable function transformations: JIT-compile (to CPU or GPU) with `jit`, vectorize functions with `vmap`, SPMD parallelization with `pmap`, and automatic differentiation with `grad`, `jacfwd`, and `jacrev`.



JAX: **J**ust **A**fter e**X**ecution (Python)

JAX is Numpy and Scipy with composable function transformations: JIT-compile (to CPU or GPU) with `jit`, vectorize functions with `vmap`, SPMD parallelization with `pmap`, and automatic differentiation with `grad`, `jacfwd`, and `jacrev`.



Not just an AD library!

JAX is super useful even if you aren't doing AD! You get the simplicity of Numpy and Scipy with the speed of JIT-compilation. Programming in JAX feels just like programming in Numpy.

Stan Math Library (C++)

The Stan Math Library is a C++ template library for automatic differentiation of any order using forward, reverse, and mixed modes. It includes a range of built-in functions for probabilistic modeling, linear algebra, and equation solving.

Stan Math Library

differentiable C++ for linear algebra & probability

Checkpointing

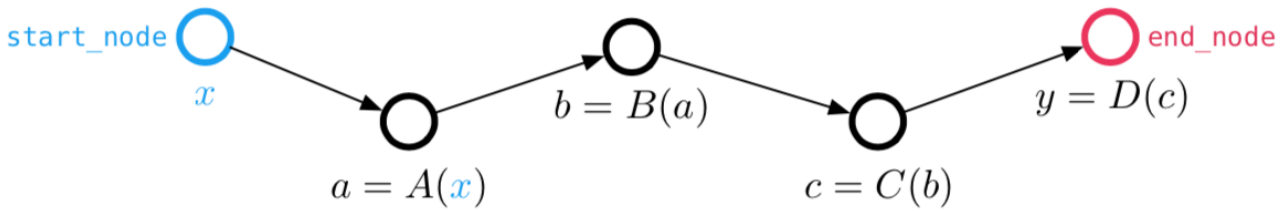
The problem with reverse mode AD

In order to compute a vector-Jacobian product (VJP) backwards, the data to calculate each primitive VJP much be stored in memory. Storing the data from every single primitive operation cause very large memory requirements for large computations.

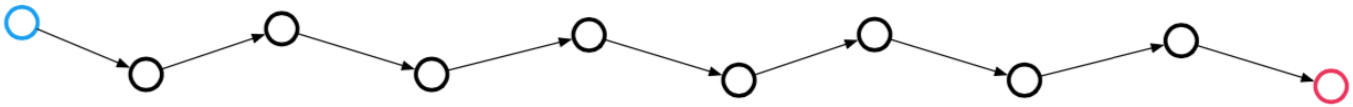
Checkpointing is a method to reduce memory requirements in exchange for increased runtime. It works by storing "checkpoints" at various points in the program and recomputing the data between checkpoints that would otherwise be stored in memory.

Checkpointing

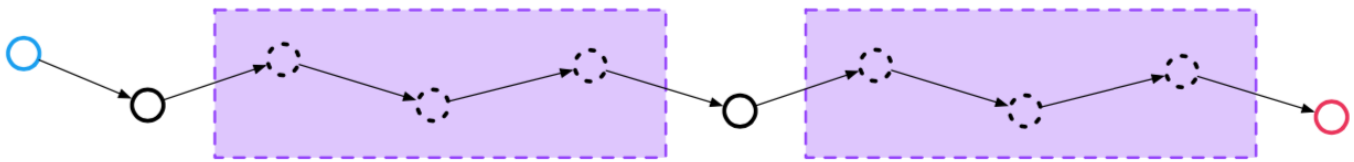
Our example function can be visualized as a graph:



Suppose we have a function which looks like this:

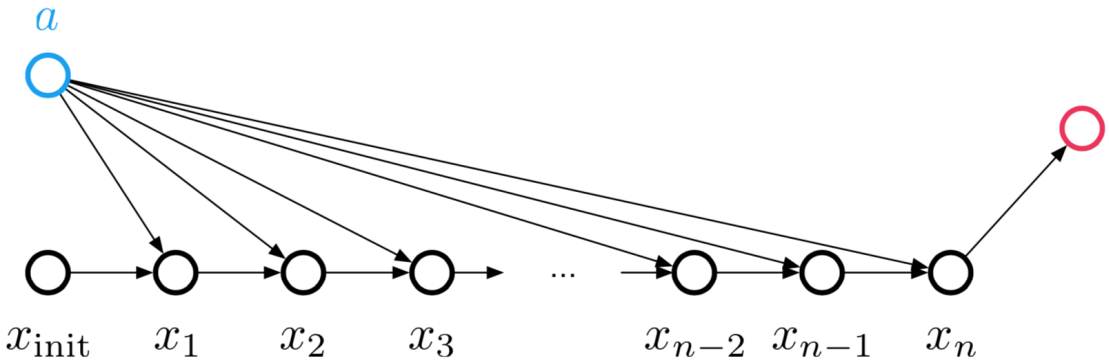


Let's use checkpointing to reduce the memory from the part of the computation in the purple boxes. We place checkpoints to the left of the purple boxes and recompute the functions in the purple boxes on the backwards pass.



AD and the adjoint method for iterative algorithms

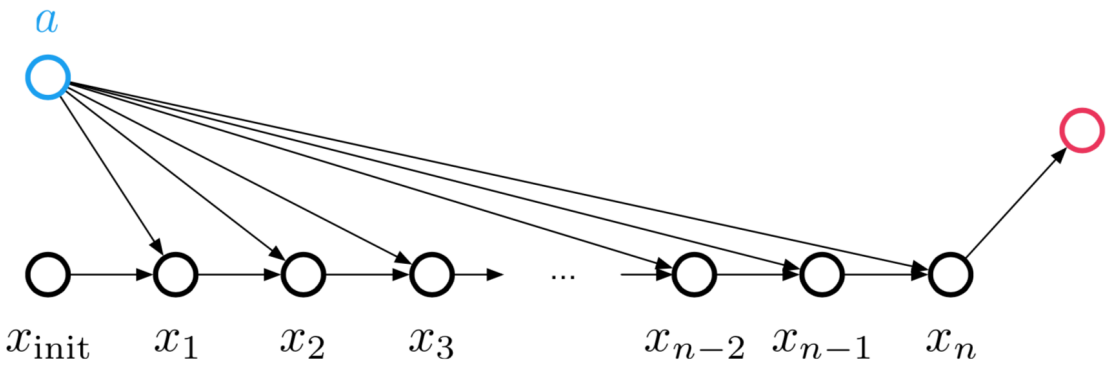
Suppose we have an iterative algorithm for solving some set of equations, and we want to compute the derivative of the solution with respect to some parameter a. Let x_{init} be an initial guess, and suppose the iterative algorithm runs n times before converging to a solution. This computation can be visualized with the following graph:



We could compute the derivative dx/da by using automatic differentiation on the entire computation. This might be very inefficient, both in terms of runtime and memory.

AD and the adjoint method for iterative algorithms

Suppose we have an iterative algorithm for solving some set of equations, and we want to compute the derivative of the solution with respect to some parameter a. Let x_{init} be an initial guess, and suppose the iterative algorithm runs n times before converging to a solution. This computation can be visualized with the following graph:



We could compute the derivative dx/da by using automatic differentiation on the entire computation. This might be very inefficient, both in terms of runtime and memory.

A clever trick

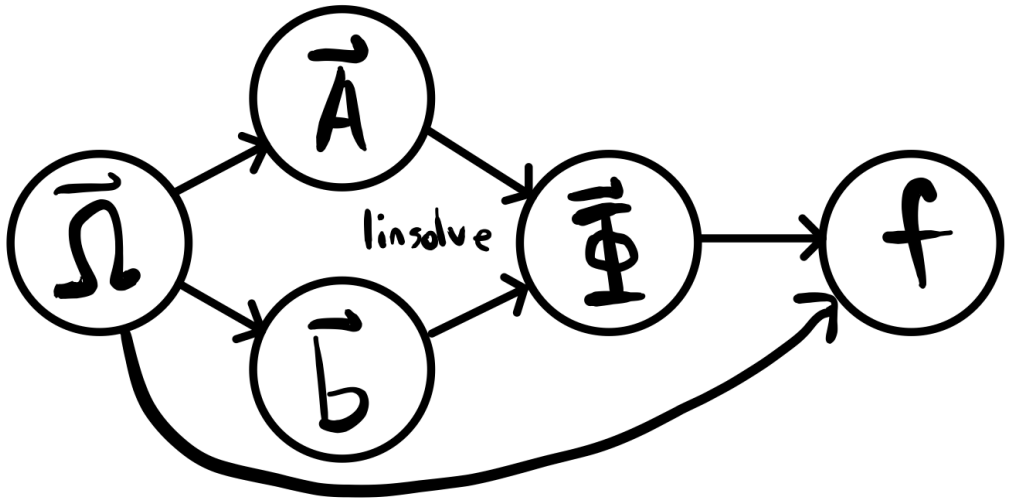
We can use the mathematical structure of the iteration to more efficiently compute the derivative. The key is that the derivative doesn't depend on x_{init}. This means that when computing the derivative, we can simply rerun the iteration with $x_{\text{init}} = x_n$, and compute the derivative of a single iteration of the algorithm. This is the trick for applying the adjoint method to systems of non-linear equations.

Review: AD and the adjoint method for linear equations

Goal: $\boldsymbol{\Omega}^* = \arg\min_{\boldsymbol{\Omega}} f(\boldsymbol{\Phi}(\boldsymbol{\Omega}), \boldsymbol{\Omega})$

Use GD: $\boldsymbol{\Omega}^{n+1} = \boldsymbol{\Omega}^n - \eta \frac{\partial f}{\partial \boldsymbol{\Omega}}$

s.t. $\boldsymbol{A}(\Omega)\boldsymbol{\Phi} = \boldsymbol{b}(\Omega)$

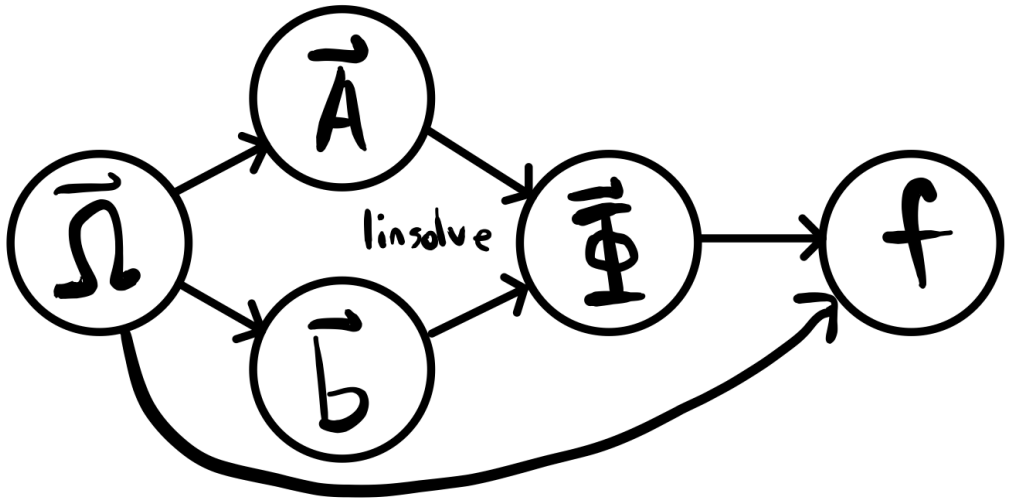


Review: AD and the adjoint method for linear equations

Goal: $\boldsymbol{\Omega}^* = \arg\min_{\boldsymbol{\Omega}} f(\boldsymbol{\Phi}(\boldsymbol{\Omega}), \boldsymbol{\Omega})$

Use GD: $\boldsymbol{\Omega}^{n+1} = \boldsymbol{\Omega}^n - \eta \frac{\partial f}{\partial \boldsymbol{\Omega}}$

s.t. $\boldsymbol{A}(\Omega)\boldsymbol{\Phi} = \boldsymbol{b}(\Omega)$



Using the adjoint method, we have

$$\boxed{\frac{df}{d\boldsymbol{\Omega}} = \frac{\partial f}{\partial \boldsymbol{\Omega}} + \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{\Omega}} - \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{A}}{\partial \boldsymbol{\Omega}} \boldsymbol{\Phi}} \quad \text{where } \boldsymbol{A}^T \boldsymbol{\lambda} = \frac{\partial f}{\partial \boldsymbol{\Phi}}$$

AD tools setup and solve the adjoint equation of a linear system automatically, e.g. JAX uses `np.linalg.solve` and `grad`.

Should AD be partially adopted by the stellarator optimization community? It's worth considering.

This is an enormously complex question. I'd love to discuss this in depth after the talk or offline.

Should AD be partially adopted by the stellarator optimization community? It's worth considering.

This is an enormously complex question. I'd love to discuss this in depth after the talk or offline.

Pros

- Gradient-based optimization of high-dimensional non-convex objective functions has been successful in many domains.
- AD and the adjoint method work together particularly well.
- If $N = 50$, does a factor of 10-20 increase in computational speed matter?
- Exact derivative needed?
- Much easier to rewrite an existing code we understand than write a new code.

Should AD be partially adopted by the stellarator optimization community? It's worth considering.

This is an enormously complex question. I'd love to discuss this in depth after the talk or offline.

Pros

- Gradient-based optimization of high-dimensional non-convex objective functions has been successful in many domains.
- AD and the adjoint method work together particularly well.
- If $N = 50$, does a factor of 10-20 increase in computational speed matter?
- Exact derivative needed?
- Much easier to rewrite an existing code we understand than write a new code.

Cons

- Is there sufficient demand for rewriting STELLOPT with an AD tool? Does our team have the right expertise?
- The stellarator community seems to like FORTRAN. Bad for AD.
- Does the right tool exist?
- Is gradient-free Bayesian optimization better? Bayesian optimization with gradients? Do we have resources to try all the above?
- Memory manageable?

AD in machine learning (ML):

A 2009 blog post made a convincing argument that ML researchers should use AD.

Criticisms of blog post:

- Computing derivatives distracts from what the field actually wants to accomplish.
- Valuable researcher time is wasted.
- Leads to preference for functions they are capable of manually deriving gradients for.

AD was eventually fully adopted by the ML community.

Structure of FOCUSADD (1 of 6)

Coil centroid is parametrized in free space with a Fourier series, as in FOCUS. Here $\boldsymbol{r}_i(\theta)$ is the position of the i coil centroid.

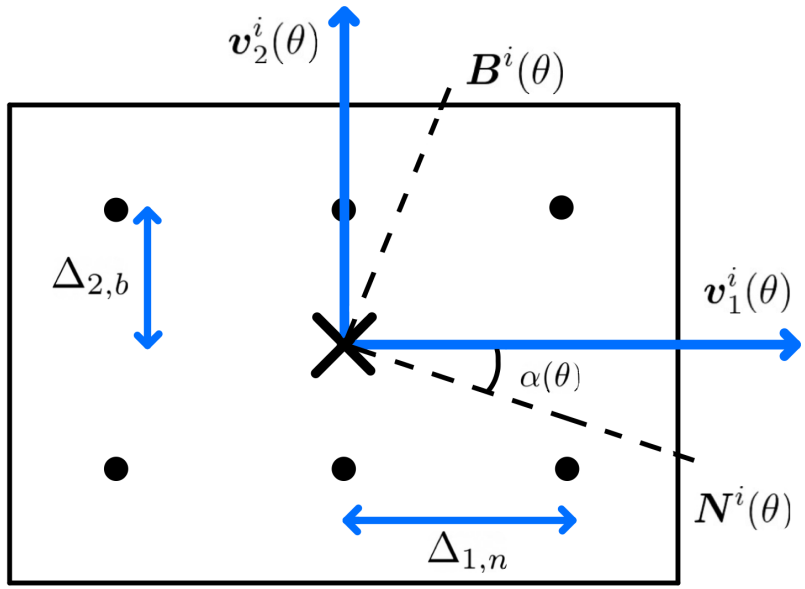
$$x^i(\theta) = \sum_{m=0}^{N_F-1} X^i_{cm}\cos(m\theta) + X^i_{sm}\sin(m\theta)$$

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

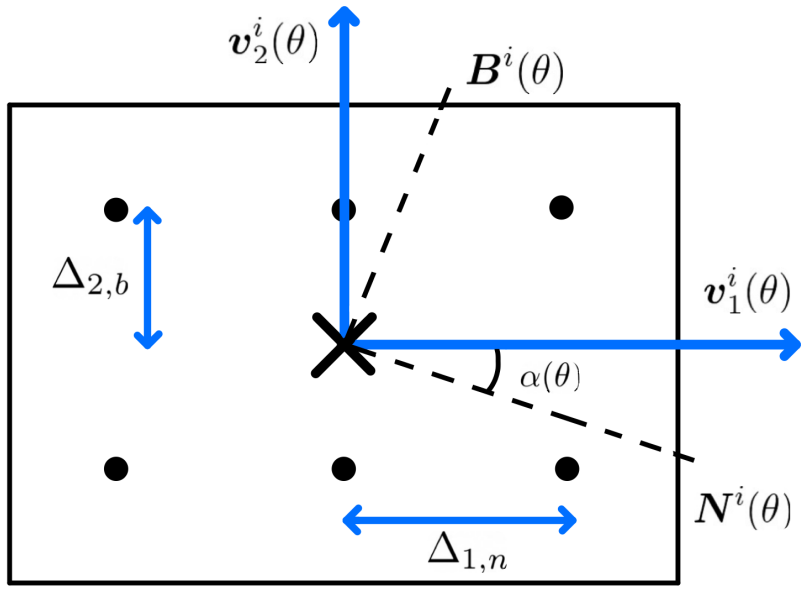
$$z^i(\theta) = \sum_{m=0}^{N_F-1} Z^i_{cm}\cos(m\theta) + Z^i_{sm}\sin(m\theta)$$

Structure of FOCUSADD (2 of 6)

The multi-filament winding pack surrounds the coil centroid. For the ith coil, the axes of the winding pack $\boldsymbol{v}_1^i$ and $\boldsymbol{v}_2^i$ are rotated by an angle α^i relative to the normal $\boldsymbol{N}^i$ and binormal $\boldsymbol{B}^i$ vectors. The normal vector is defined as the component perpendicular to the tangent of the vector from the coil center-of-mass to the local point.



Structure of FOCUSADD (3 of 6)



α is parametrized by another Fourier series, giving the coil the freedom to twist in space.

$$\alpha^i(\theta) = \frac{N_R\theta}{2} + \sum_{m=0}^{N_{FR}-1} A^i_{cm}\cos(m\theta) + A^i_{sm}\sin(m\theta)$$

Structure of FOCUSADD (4 of 6)

Once we have $\boldsymbol{v}_1^i$ and $\boldsymbol{v}_2^i$, we can compute the position of the $N_n \times N_b$ filaments for each coil. We do this using the following formula for the nth and bth filaments, where n runs from 0 to $N_n - 1$ and b runs from 0 to $N_b - 1$. Here, l_n is the spacing between the filaments in the $\boldsymbol{v}_1$ direction, and l_b is the spacing between the filaments in the $\boldsymbol{v}_2$ direction.

$$\boldsymbol{r}_{n,b}^i(\theta) = \boldsymbol{r}_{central}^i + \left[n - \frac{N_n - 1}{2}\right] l_n \boldsymbol{v}_1^i(\theta) + \left[b - \frac{N_b - 1}{2}\right] l_b \boldsymbol{v}^i(\theta)$$

So far we've only computed vacuum fields, using the Biot-Savart law.

$$\boldsymbol{B}(\boldsymbol{r}) = \sum_{i=1}^{N_c} \sum_{n=1}^{N_1} \sum_{b=1}^{N_2} \mu_0 I_{n,b}^i \oint \frac{d\boldsymbol{l}_{n,b}^i \times (\boldsymbol{r} - \boldsymbol{r}_{n,b}^i)}{|\boldsymbol{r} - \boldsymbol{r}_{n,b}^i|^3}$$

Structure of FOCUSADD (5 of 6)

The optimization is performed using gradient descent on an objective function f_{total}, given by a sum of physics objectives and engineering objectives.

$$f_{total}(\boldsymbol{p}) = f_{Phys}(\boldsymbol{p}) + \lambda_{Eng} f_{Eng}(\boldsymbol{p})$$

The simplest possible physics objective was chosen, the squared surface-normal magnetic field integrated over the surface.

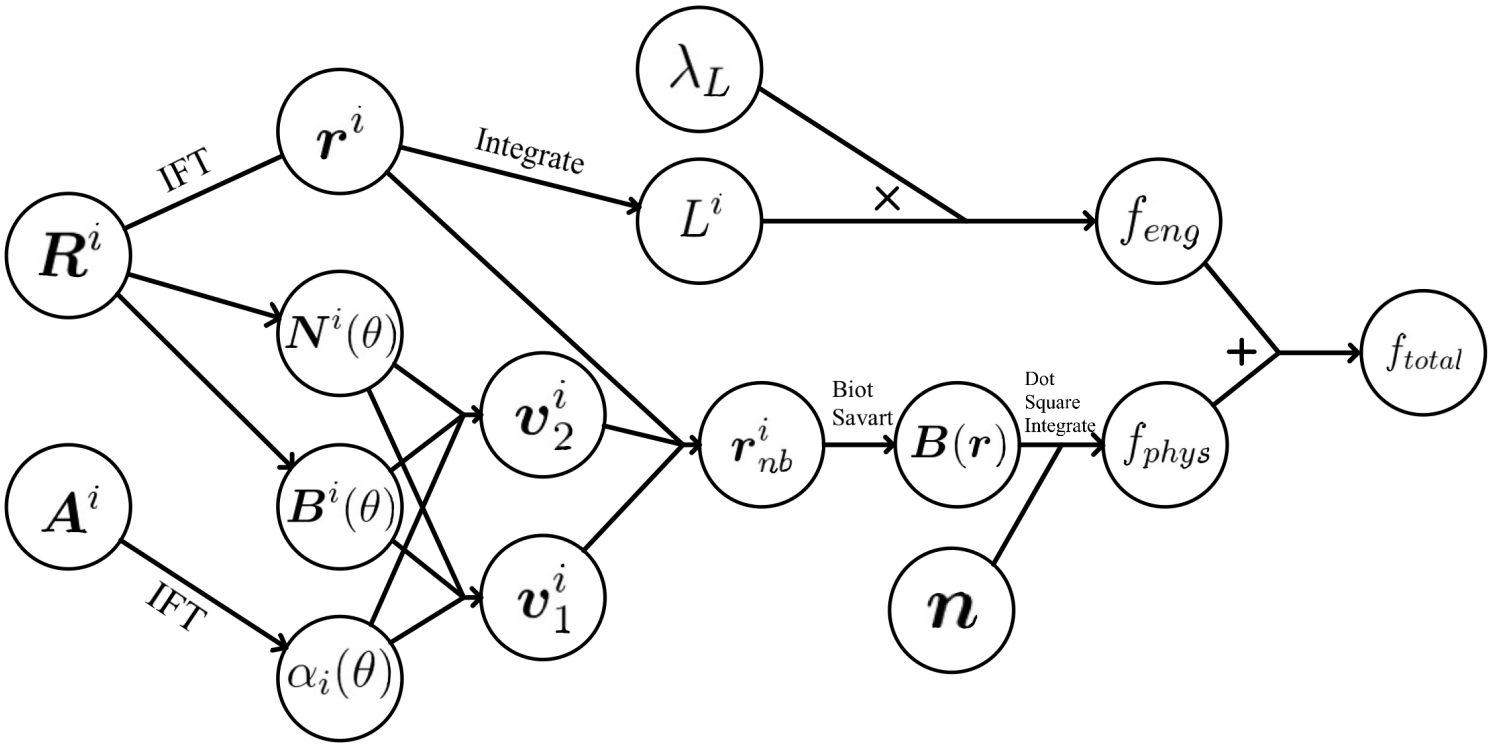
$$f_{Phys} \equiv \int_S (\boldsymbol{B} \cdot \boldsymbol{n})^2 dA$$

A simple engineering objective was chosen, namely the total length of the coils.

$$f_{Eng} \equiv \sum_{i=1}^{N} L_i$$

Structure of FOCUSADD (6 of 6)

The following computational graph describes the structure of the computation performed by FOCUSADD. In my AD tool (JAX), I simply compute f_{total}, then type "grad" to get the gradient.



Important facts about AD

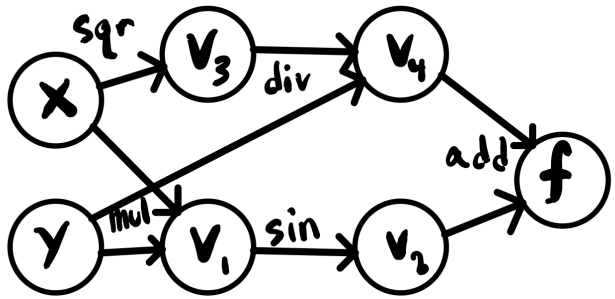
- Suppose $\boldsymbol{y} = f(\boldsymbol{x})$, where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^m$. Then first-order AD computes the numerical value of the Jacobian $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ at a particular value of $\boldsymbol{x}$.
- Suppose f takes time T to compute.
- Forward mode AD computes the Jacobian of f in time $\mathcal{O}(nT)$.
- Reverse mode AD computes the Jacobian of f in time $\mathcal{O}(mT)$.

How does AD work?

Step 1: Compute function

The AD tool computes the function f, one elementary operation at a time. A representation of the 'computational graph' is built.

$$f(x, y) = \sin(xy) + x^2/y$$



Step 2: Compute derivatives in the reverse order

Compute the derivative of the output of f with respect to each variable v_i by traversing the graph in reverse order. This is exactly the chain rule, applied in a clever way.

$$\frac{\partial f}{\partial v_i} = \sum_{\substack{j \in \text{children} \\ \text{of } i}} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

The details

Step 1:

$x = 2.0$
$y = 3.0$
$v_1 = x * y = 6.0$
$v_2 = \sin(v_1) = -0.297$
$v_3 = x^2 = 4.0$
$v_4 = v_3/y = 1.333$
$f = v_2 + v_4 = 1.054$

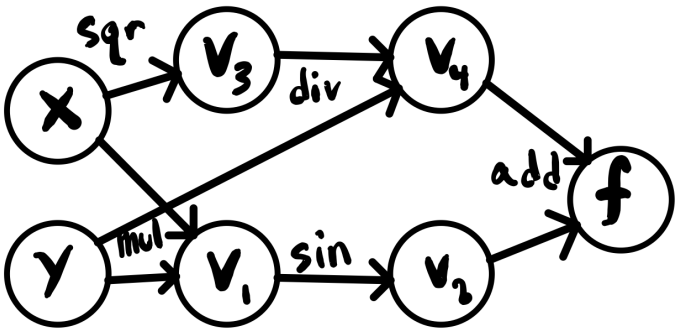
Step 2:

$\frac{\partial f}{\partial f} = 1.0$
$\frac{\partial f}{\partial v_4} = 1.0$
$\frac{\partial f}{\partial v_3} = \frac{\partial f}{\partial v_4}\frac{\partial v_4}{\partial v_3} = 0.333$
$\frac{\partial f}{\partial v_2} = 1.0$
$\frac{\partial f}{\partial v_1} = \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial v_1} = 0.960$

Step 2, continued:

$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial y} + \frac{\partial f}{\partial v_4}\frac{\partial v_4}{\partial y}$
$\frac{\partial f}{\partial y} = 1.476$
$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial x} + \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial x}$
$\frac{\partial f}{\partial x} = 4.214$

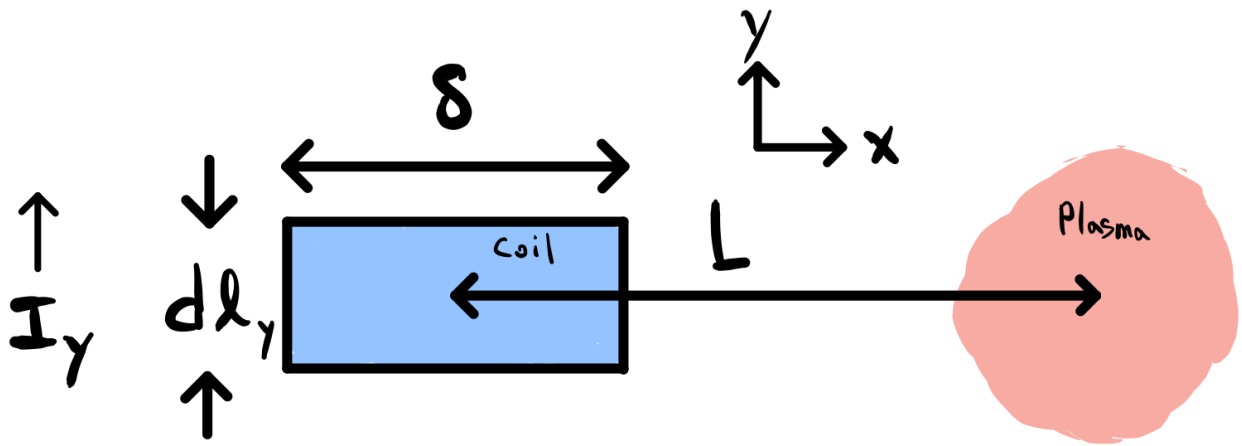
$$f(x, y) = \sin(xy) + x^2/y$$



$$\frac{\partial f}{\partial v_i} = \sum_{\substack{j \in \text{children} \\ \text{of } i}} \frac{\partial f}{\partial v_j}\frac{\partial v_j}{\partial v_i}$$

Deviations in the magnetic field are second-order in the coil thickness divided by the coil-plasma distance

Suppose we have a coil carrying current in the y-direction with thickness δ a distance L away from our plasma.



$$\text{Biot-Savart: } \frac{dB_z}{d\ell_y} = -\frac{\mu_0 I_y}{4\pi}\int_{-\delta/2}^{\delta/2}\frac{dx}{(L+x)^2}$$

$$dB_z \approx -\frac{\mu_0 I_y d\ell_y}{4\pi L^2}\int_{-\delta/2}^{\delta/2}\left(1 - \frac{2x}{L}^{0} + \frac{3x^2}{L^2}\right)dx \approx dB_{filament}\left(1 + \frac{\delta^2}{4L^2}\right)$$