Finite-Build Stellarator Coil Design & Automatic Differentiation

Nick McGreivy[1,2] Stuart Hudson[1] Caoxiang Zhu[1]

[1]Princeton Plasma Physics Laboratory

[2]PhD Candidate
Program in Plasma Physics
Princeton University

Coffee & Chalk
July 16th 2020

Derivatives have been used in stellarator coil design for optimization and sensitivity analysis

For example:

- D.J. Strickler, L.A. Berry, & S.P. Hirshman (2002). Designing coils for compact stellarators *Fusion Sci. Technology*.
- T. Brown, J. Breslau, D. Gates, N. Pomphrey, & A. Zolfaghari (2015). *IEEE 26th Symp. on Fusion Engineering*.
- Caoxiang Zhu, Stuart R. Hudson, Yuntao Song, & Yuanxi Wan (2017). New method to design stellarator coils without the winding surface *Nuclear Fusion*.
- Caoxiang Zhu, Stuart R Hudson, Samuel A Lazerson, Yuntao Song, & Yuanxi Wan (2018). Hessian matrix approach for determining error field sensitivity to coil deviations *Plasma Physics and Controlled Fusion*.
- Caoxiang Zhu, Stuart R Hudson, Yuntao Song, & Yuanxi Wan (2018). Designing stellarator coils by a modified Newton method using FOCUS *Plasma Physics and Controlled Fusion*.
- E.J. Paul, M. Landreman, A. Bader, & W. Dorland (2018). An adjoint method for gradient-based optimization of stellarator coil shapes *Nuclear Fusion*.
- Matt Landreman & Elizabeth Paul (2018). Computing local sensitivity and tolerances for stellarator physics properties using shape gradients *Nuclear Fusion*.
- Hudson, S., Zhu, C., Pfefferle, D., & Gunderson, L. (2018). Differentiating the shape of stellarator coils with respect to the plasma boundary *Physics Letters. A*.
- Caoxiang Zhu, David A. Gates, Stuart R. Hudson, Haifeng Liu, Yuhong Xu, Akihiro Shimizu, & Shoichi Okamura (2019). Identification of important error fields in stellarators using the Hessian matrix method *Nuclear Fusion*.

We can compute derivatives three ways

1. Finite-difference derivatives (numerical derivatives)

- For an N-dimensional function, finite-difference requires $N + 1$ function evaluations to get the N-dimensional gradient
- Inexact due to truncation and round-off errors
- Simple

We can compute derivatives three ways

1. Finite-difference derivatives (numerical derivatives)

- For an N-dimensional function, finite-difference requires $N+1$ function evaluations to get the N-dimensional gradient
- Inexact due to truncation and round-off errors
- Simple

2. Analytic derivatives

- Either computed by hand or with symbolic differentiation
- Needs to be programmed by a human
- Efficient

3. 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.

This is a talk in two parts

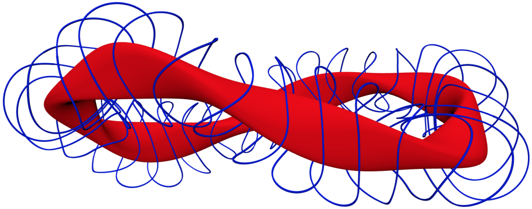
Organization of talk

- Part I: Finite-build coil design & results (15 minutes)
- Part II: Discussion of automatic differentiation

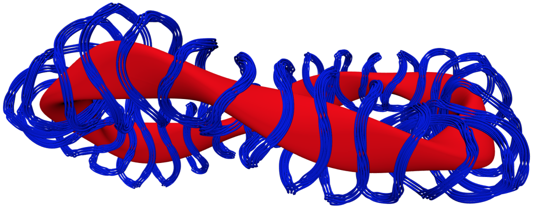
Part I:
Finite-build coil design & results

FOCUSADD: **FOCUS** with **AD** and finite-buil**D**

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



FOCUSADD (N. McGreivy, S. R. Hudson, C.Zhu 2020)



Stellarator Coil Design as an Optimization Problem

Choose the coil parameters $\boldsymbol{p}$ which minimize an objective function f which measures the deviation between target magnetic field and the magnetic field that your coils actually produce.

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

This objective function traditionally includes the quadratic flux of $\boldsymbol{B}$ on a surface S:

$$f(\boldsymbol{p}) = \int_S \Big(\boldsymbol{B}(\boldsymbol{p}) \cdot \hat{\boldsymbol{n}}\Big)^2 dA + \text{Regularization term(s)}$$

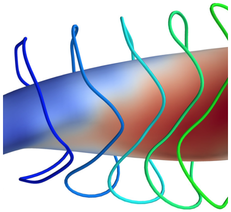
FOCUS: Coil Design Code

Represent coils as closed filamentary curves in space. $\boldsymbol{B}$ is given by the biot-savart law, and f is minimized using a gradient-based optimization method.

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

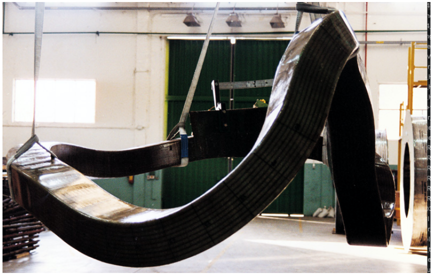
$$y^i(\theta) = \sum_{m=0}^{N_F-1} Y^i_{cm}\cos(m\theta) + Y^i_{sm}\sin(m\theta)$$

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



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

Motivation: How much does including finite coil thickness change optimized coils?



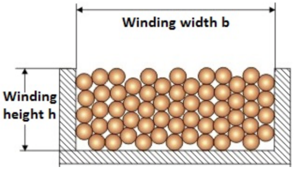
Deviations in B are second-order in the coil thickness

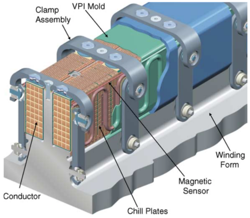
A simple calculation (not shown) shows that for a coil of thickness δ a distance L from the plasma, the change in magnetic field δB is of order ${}^{\delta^2}/_{L^2}$.

$$\delta B \sim \frac{\delta^2}{L^2}$$

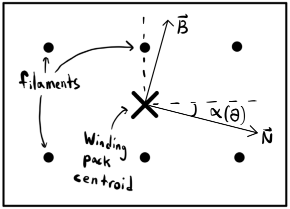
Multi-filament approximation to coil winding pack

Real coils are wound in a so-called ‘winding pack’.





One option: ‘multi-filament’ approximation to the coil winding pack. The filaments are placed on a grid defined by a choice of frame surrounding the coil centroid, and the rotation angle $\alpha(\theta)$ is parametrized with a Fourier series in poloidal angle θ.

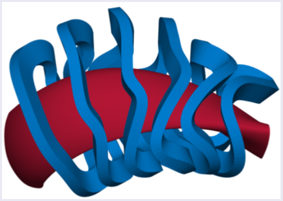


$$\alpha^i(\theta) = \sum_{m=0}^{N_{FR}-1} A^i_{cm}\cos(m\theta) + A^i_{sm}\sin(m\theta)$$

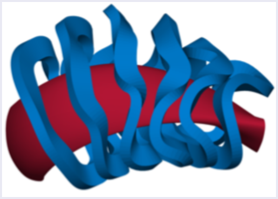
Optimizing rotation profile of finite-build coils: HSX

Work done by Wisconsin group

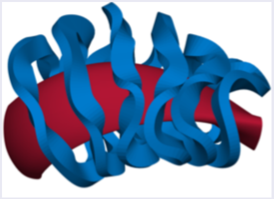
No Rotation



Some Regularization



No Regularization

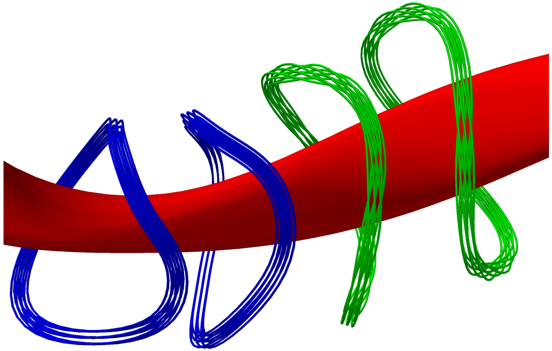


L. Singh, T. Kruger et al., *Optimization of Finite-build Stellarator Coils* (Accepted to JPP)

Finite-build coils with FOCUSADD

FOCUSADD efficiently optimizes two quantities

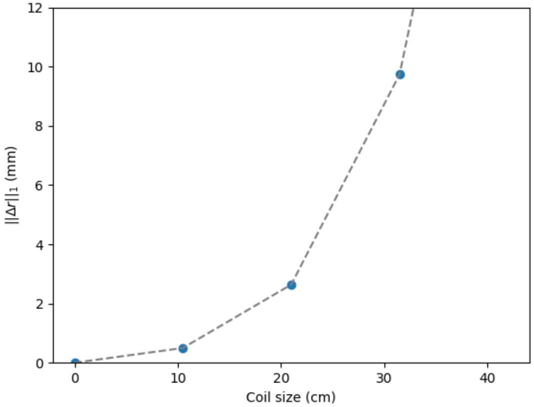
FOCUS optimizes the filamentary coil **centroid positions** $x^i(\theta)$, $y^i(\theta)$, and $z^i(\theta)$. The Wisconsin group optimizes the **rotation profile** $\alpha^i(\theta)$ around the coil centroid. FOCUSADD optimizes **both** quantities at once.



Optimized finite-build coils for rotating elliptical stellarator, with no rotation (left) and no regularization (right).

Would anyone ever build this?

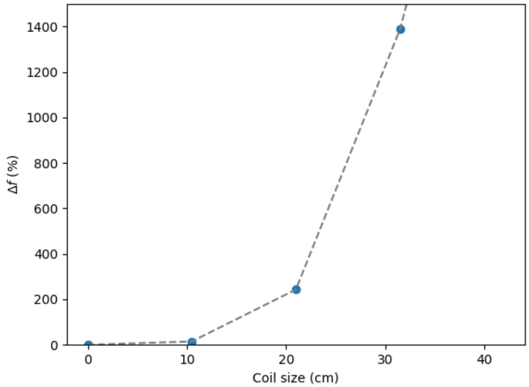
How much do coil positions shift with finite build?



$||\Delta r||_1 \equiv$ the L1 norm of the difference between the positions of the centroid of the filamentary optimized coils and the centroid of the finite-build optimized coils.

W7-X coil dimensions: 17.8 by 21.2cm

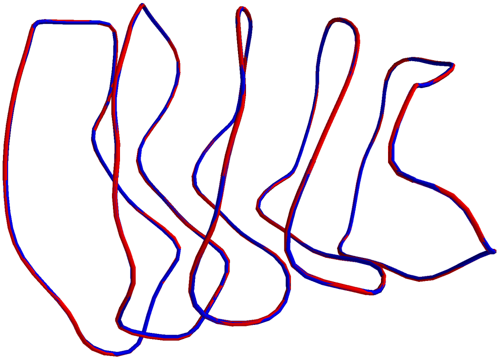
What is the effect on the quadratic flux?



$\Delta f \equiv$ the difference between the quadratic flux of finite-build coils which are optimized using a filamentary approximation, and the quadratic flux of finite-build coils which are optimized for their finite-build.

W7-X coil dimensions: 17.8 by 21.2cm

Does finite-build matter?



Coil centroids for optimized filamentary and finite-build (no rotation) w7x coils.

$$||\Delta r||_1 = 2.6\text{mm}$$

$$\Delta f = \frac{1.45 - .45}{.45} = 240\%$$

In conclusion, finite-build is important when

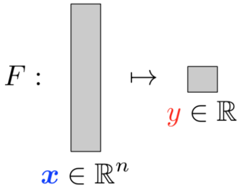
- Coil tolerances are small
- Coil-plasma distance is small
- Coils are large

Part II: Discussion of automatic differentiation (AD)

A functional viewpoint

Taking a derivative

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

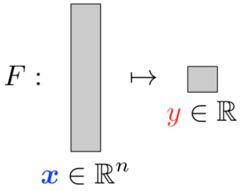


$$F = D \circ C \circ B \circ A \qquad y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

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

Taking a derivative

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



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

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

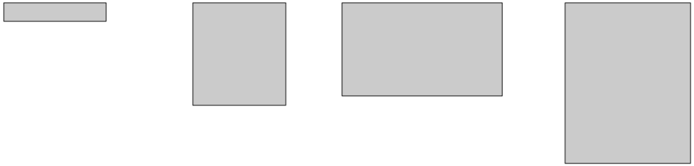
Primitive operations

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`.

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

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

$$\begin{tikzcd}
F'(\boldsymbol{x}) = & \frac{\partial y}{\partial \boldsymbol{c}} & \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} & \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} & \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \\
\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \arrow[ur] & \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b}) \arrow[ur] & & \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = B'(\boldsymbol{a}) \arrow[u] & \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = A'(\boldsymbol{x}) \arrow[u]
\end{tikzcd}$$

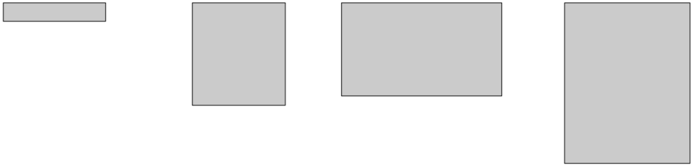


$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

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

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

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



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 $\mathcal{O}(mT)$.

AD: Forward and Reverse Modes

Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. Pushforward tangent vectors.

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

Reverse mode: vector-Jacobian products (VJPs), build Jacobian one row at a time. Pullback cotangent vectors.

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

AD: Forward and Reverse Modes

Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. Pushforward tangent vectors.

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

Reverse mode: vector-Jacobian products (VJPs), build Jacobian one row at a time. Pullback cotangent vectors.

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

From Jacobians to gradients

Often we want the gradient of a scalar function $F : \mathbb{R}^n \to \mathbb{R}$, which we can get by setting $\boldsymbol{v}^T = [1.0]$.

AD: Forward and Reverse Modes

Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. Pushforward tangent vectors.

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

Reverse mode: vector-Jacobian products (VJPs), build Jacobian one row at a time. Pullback cotangent vectors.

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

What makes it "automatic"?

The key is that your AD library knows the Jacobian of each primitive operation. This allows it to pushforward/pullback tangent/cotangent vectors for each primitive and compute the total JVP/VJP.

Why is AD useful?

1. Simplicity

- Finding analytic derivatives is time-consuming and often hard.
- Programming those derivatives is time-consuming.
- AD removes these steps.

Why is AD useful?

1. Simplicity

- Finding analytic derivatives is time-consuming and often hard.
- Programming those derivatives is time-consuming.
- AD removes these steps.

2. Ideal for gradient-based optimization

- For a scalar function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ which has time-complexity $\mathcal{O}(T)$, reverse mode AD computes the gradient in time $\mathcal{O}(T)$. This is as efficient as the best analytic methods.

Why is AD useful?

1. Simplicity

- Finding analytic derivatives is time-consuming and often hard.
- Programming those derivatives is time-consuming.
- AD removes these steps.

2. Ideal for gradient-based optimization

- For a scalar function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ which has time-complexity $\mathcal{O}(T)$, reverse mode AD computes the gradient in time $\mathcal{O}(T)$. This is as efficient as the best analytic methods.

3. Effortless gradients

- Easy to rapidly prototype new ideas and objectives.

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

AD libraries for scientific computing

julia

Stan Math Library

differentiable C++ for linear algebra & probability

AD in machine learning

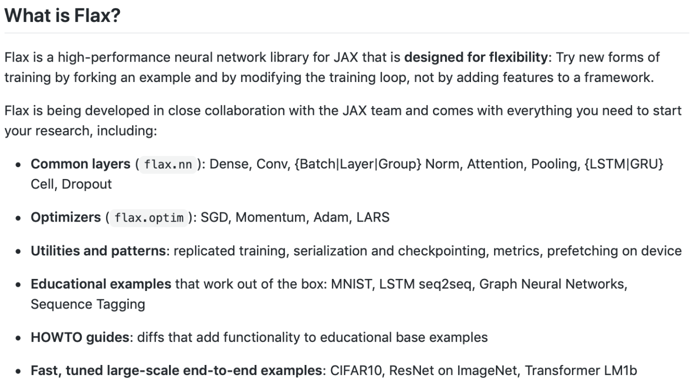
A few examples

PYTORCH

TensorFlow 2.0

AD in machine learning

A few examples



What is Flax?

Flax is a high-performance neural network library for JAX that is **designed for flexibility**: Try new forms of training by forking an example and by modifying the training loop, not by adding features to a framework.

Flax is being developed in close collaboration with the JAX team and comes with everything you need to start your research, including:

- **Common layers** (`flax.nn`): Dense, Conv, {Batch|Layer|Group} Norm, Attention, Pooling, {LSTM|GRU} Cell, Dropout
- **Optimizers** (`flax.optim`): SGD, Momentum, Adam, LARS
- **Utilities and patterns**: replicated training, serialization and checkpointing, metrics, prefetching on device
- **Educational examples** that work out of the box: MNIST, LSTM seq2seq, Graph Neural Networks, Sequence Tagging
- **HOWTO guides**: diffs that add functionality to educational base examples
- **Fast, tuned large-scale end-to-end examples**: CIFAR10, ResNet on ImageNet, Transformer LM1b

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

Comparison between JAX and Stan

Coming next week. . .

Preview

JAX has nearly everything you might want from an AD library for scientific computing, except it implements SPMD parallelization (which is useful in ML) rather than MPI parallization. Stan Math Library isn't quite as easy to use as JAX, but seems to have implemented MPI as well as many (but not all) of the desireable features of JAX. Stan may have fewer primitives implemented.

Checkpointing

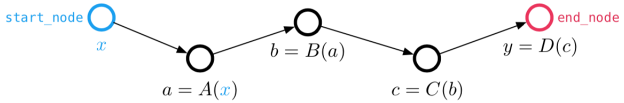
The problem with reverse mode AD

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.

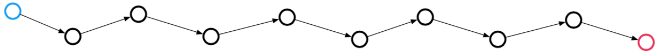
Checkpointing is a method to reduce memory requirements in exchange for increased runtime. It works by storing "checkpoints" at various points in the program and recomputing the data between checkpoints that would otherwise be stored in memory.

Checkpointing

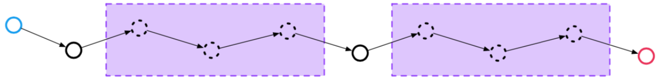
Our example function can be visualized as a graph:



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.



The adjoint method

Differentiating under constraints requires a linear solve

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

The adjoint method

Differentiating under constraints requires a linear solve

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

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

The adjoint method

Differentiating under constraints requires a linear solve

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

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

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

The adjoint method

Differentiating under constraints requires a linear solve

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

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

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

The adjoint method

Differentiating under constraints requires a linear solve

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

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

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

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

The adjoint method

Differentiating under constraints requires a linear solve

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

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

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

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

The adjoint method

Differentiating under constraints requires a linear solve

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

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

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

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

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

AD and the adjoint method for linear equations

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

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

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

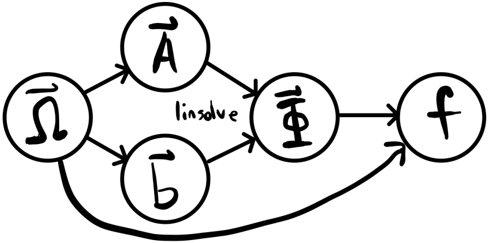
$$\begin{tikzcd}
& \vec{A} \arrow[dr] & & \\
\vec{\Omega} \arrow[ur] \arrow[dr] \arrow[rrr, bend right=60] & \text{linsolve} & \vec{\Phi} \arrow[r] & f \\
& \vec{b} \arrow[ur] & &
\end{tikzcd}$$

AD and the adjoint method for linear equations

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

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

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


$$\begin{tikzcd} & \vec{A} \arrow[dr] & & \\ \vec{\Omega} \arrow[ur] \arrow[dr] \arrow[rrr, bend right=60] & & \vec{\Phi} \arrow[r] & f \\ & \vec{b} \arrow[ur] & & \end{tikzcd}$$

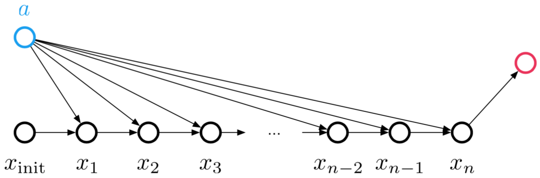
Using the adjoint method, we have

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

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

AD and the adjoint method for iterative algorithms

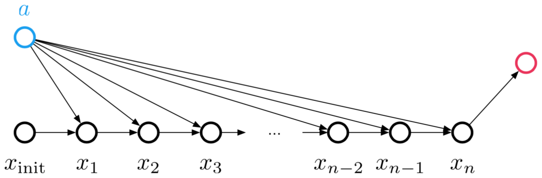
Suppose we have an iterative algorithm for solving some set of equations, and we want to compute the derivative of the solution with respect to some parameter a. Let x_{init} be an initial guess, and suppose the iterative algorithm runs n times before converging to a solution. This computation can be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

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.

Automatically generating adjoint equations

What is an adjoint equation?

The adjoint equation is the equation which defines the derivative of a given mathematical equation. For example: the adjoint equation for the linear system $\boldsymbol{A}\boldsymbol{\Phi} = \boldsymbol{b}$ was shown to be $\boldsymbol{A}^T\boldsymbol{\lambda} = {}^{\partial f}/_{\partial \boldsymbol{\Phi}}$, and the adjoint equation for the nonlinear system $\boldsymbol{g} = 0$ was shown to be $({}^{\partial \boldsymbol{g}}/_{\partial \boldsymbol{u}})^T\boldsymbol{\lambda} = {}^{\partial f}/_{\partial \boldsymbol{u}}$

It's also possible to setup and solve adjoint equations for the solution of ODEs and PDEs. (Details coming next week. . .)

Another perspective on AD

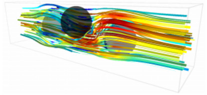
AD libraries know how to setup and solve adjoint equations for each primitive operation in the library.

dolfin-adjoint

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 is a popular open-source (LGPLv3) computing platform for solving partial differential equations (PDEs). 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.

Summary

- Automatic differentiation allows the efficient computation of derivatives in stellarator coil design. This has allowed the development of a new finite-build coil design code, FOCUSADD.
- Accounting for the finite build of coils is important when coil tolerances are small, when the coil-plasma distance is small, and/or when coil thicknesses are large.
- Automatic differentiation libraries know how to setup and solve adjoint equations for each primitive operation in the library.
- Primitive operations can be composed together arbitrarily to push forward tangent vectors to compute Jacobian-vector products or pull back cotangent vectors to compute vector-Jacobian products.

Thank you

Additional Slides

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.

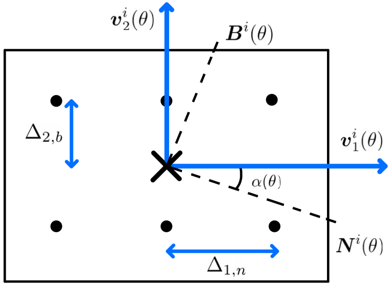
$$x^i(\theta) = \sum_{m=0}^{N_F-1} X^i_{cm}\cos(m\theta) + X^i_{sm}\sin(m\theta)$$

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

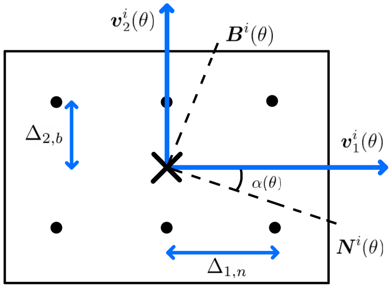
$$z^i(\theta) = \sum_{m=0}^{N_F-1} Z^i_{cm}\cos(m\theta) + Z^i_{sm}\sin(m\theta)$$

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.


$\boldsymbol{v}_2^i(\theta)$
$\boldsymbol{B}^i(\theta)$
$\Delta_{2,b}$
$\boldsymbol{v}_1^i(\theta)$
$\alpha(\theta)$
$\boldsymbol{N}^i(\theta)$
$\Delta_{1,n}$

Structure of FOCUSADD (3 of 6)



$\boldsymbol{v}_2^i(\theta)$
$\boldsymbol{B}^i(\theta)$
$\Delta_{2,b}$
$\boldsymbol{v}_1^i(\theta)$
$\alpha(\theta)$
$\boldsymbol{N}^i(\theta)$
$\Delta_{1,n}$

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.

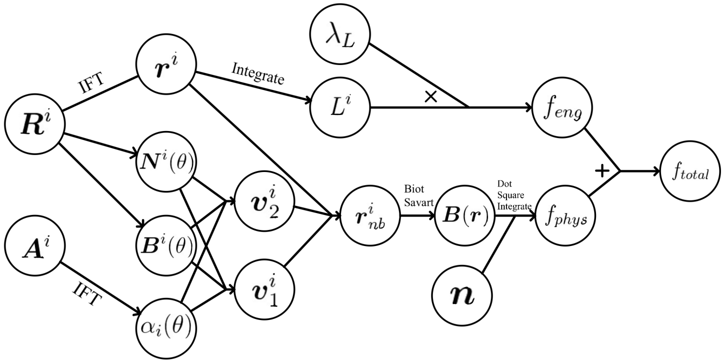
$$f_{Phys} \equiv \int_S (\boldsymbol{B} \cdot \boldsymbol{n})^2 dA$$

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

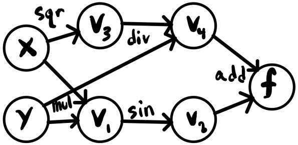
- Suppose $\boldsymbol{y} = f(\boldsymbol{x})$, where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^m$. Then first-order AD computes the numerical value of the Jacobian $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ at a particular value of $\boldsymbol{x}$.
- Suppose 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:

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

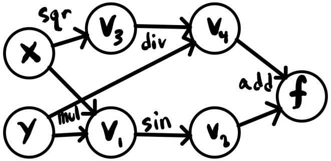
Step 2:

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

Step 2, continued:

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

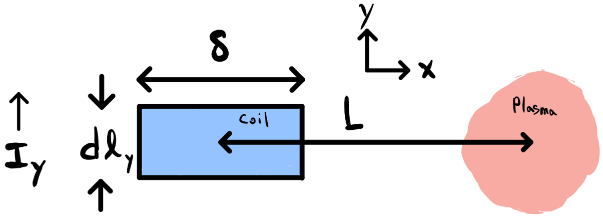
$$f(x, y) = \sin(xy) + x^2/y$$


$$\begin{tikzcd} x \arrow[r, "sqr"] \arrow[dr] & v_3 \arrow[r, "div"] & v_4 \arrow[dr, "add"] & \\ y \arrow[r, "mul"] \arrow[urr] & v_1 \arrow[r, "sin"] & v_2 \arrow[r] & f \end{tikzcd}$$

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

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.



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

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

Analytic magnetic fields for circular coils

From Static and Dynamic Electricity (1950) by W.R. Smythe, p. 270-271, we have

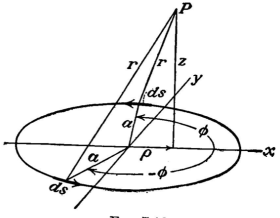


Fig. 7.10.

$$B_\rho = \frac{\mu I}{2\pi}\frac{z}{\rho[(a+\rho)^2+z^2]^{\frac{1}{2}}}\left[-K + \frac{a^2+\rho^2+z^2}{(a-\rho)^2+z^2}E\right]$$
$$B_z = \frac{\mu I}{2\pi}\frac{1}{[(a+\rho)^2+z^2]^{\frac{1}{2}}}\left[K + \frac{a^2-\rho^2-z^2}{(a-\rho)^2+z^2}E\right]$$

where $K(k)$ and $E(k)$ are complete elliptic integrals of the first and second kind and $k^2 \equiv 4a\rho/[(a+\rho)^2+z^2]$. In the plane of the coil, we have $z=0$ and $B_\rho = 0$, giving

$$B_z = \frac{\mu_0 I}{2\pi(a+\rho)}\left[K(k) + \frac{a+\rho}{a-\rho}E(k)\right]$$

Taylor expand in the $z = 0$ plane

$$B_z = \frac{\mu_0 I}{2\pi(a+\rho)}\left[K(k) + \frac{a+\rho}{a-\rho}E(k)\right]$$

We can rewrite this in terms of $\epsilon \equiv \rho/a$ and use ϵ as an expansion parameter.

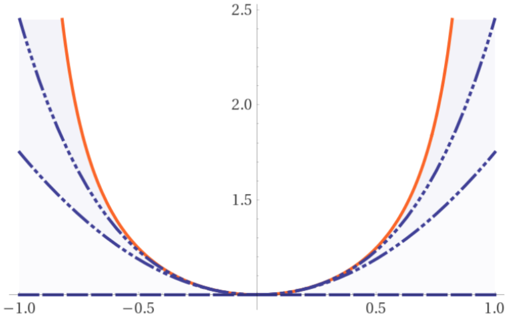
$$B_z = \frac{\mu_0 I}{2\pi a}\left[\frac{1}{1+\epsilon}K(k) + \frac{1}{1-\epsilon}E(k)\right] \quad k^2 = \frac{4\epsilon}{(1+\epsilon)^2}$$

Since k^2 is small, we can use $K(k) = \frac{\pi}{2}(1 + \frac{1}{4}k^2 + \frac{9}{64}k^4 + \dots)$ and $E(k) = \frac{\pi}{2}(1 - \frac{1}{4}k^2 - \frac{3}{64}k^4 - \dots)$. Working out the expansion gives us

$$B_z \approx \frac{\mu_0 I}{2\pi a}\left[1 + \frac{3\epsilon^2}{4} + \frac{45\epsilon^4}{64}\right]$$

How good of an approximation is this?

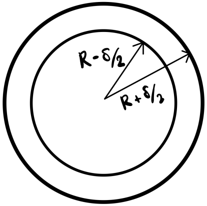
$$B_z \approx \frac{\mu_0 I}{2a}\left[1 + \frac{3\epsilon^2}{4} + \frac{45\epsilon^4}{64}\right]$$



This approximation is robust for $\epsilon = \rho/a \lessapprox 0.6$.

From filamentary coils to finite-build coils

Suppose our coil is an annulus of radius R and thickness δ carrying total current I with constant volumetric current density. Then the magnetic field at $z = 0$ and radius ρ is an integral over dB_z from $r = R - \delta/2$ to $r = R + \delta/2$.



$$B_z \approx \frac{\mu_0 I}{2R\delta} \int_{r=R-\delta/2}^{r=R+\delta/2} \left(1 + \frac{3x^2}{4} + \frac{45x^4}{64}\right) dr$$

Expanding this integral in δ/R, to lowest order this is

$$B_z \approx \frac{\mu_0 I}{2R}\left[1 + \frac{3\rho^2}{4R^2}\left(1 + \frac{\delta^2}{4R^2}\right) + \frac{45\rho^4}{64R^4}\left(1 + \frac{5\delta^2}{6R^2}\right)\right]$$