# A Tutorial on Automatic Differentiation for Scientific Design: Practical, Elegant, and Powerful

Nick McGreivy

PhD Candidate
Program in Plasma Physics
Princeton University

$$\begin{tikzcd}
\boxed{\texttt{f(x) \{...\};}} \arrow[rr, "\text{Automatic differentiation}"] & & \boxed{\texttt{df(x) \{...\};}} \\
\boxed{y = f(x)} \arrow[u, "\text{human programmer}"'] \arrow[rr, dashed, "\text{symbolic differentiation (human/computer)}"'] & & \boxed{y' = f'(x)} \arrow[u, dashed]
\end{tikzcd}$$

March 5th 2021

## Automatic Differentiation (AD)

1. Practical
2. Elegant
3. Powerful

## Automatic Differentiation (AD)

1. Practical
2. Elegant
3. Powerful

## Automatic Differentiation (AD)

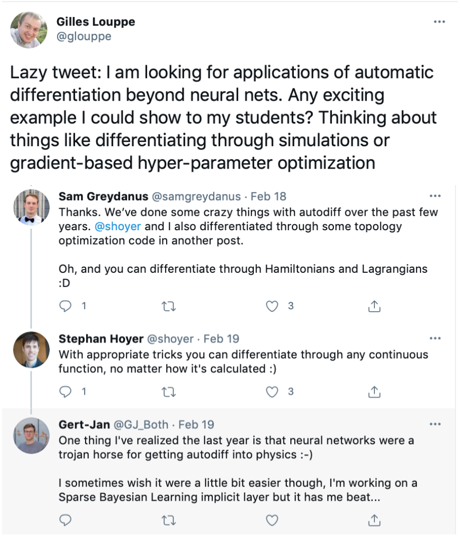
Also known as algorithmic differentiation or computational differentiation

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

## Where is AD being used?

A few highlights

- Machine Learning (Tensorflow, Pytorch are AD libraries specialized for ML)
- Learning protein structure (e.g., AlphaFold)
- Many-body Schrodinger equation (e.g., FermiNet)
- Stellarator coil design
- Differentiable ray tracing
- Model uncertainty & sensitivity
- Optimization of fluid simulations



**Gilles Louppe**
@glouppe

Lazy tweet: I am looking for applications of automatic differentiation beyond neural nets. Any exciting example I could show to my students? Thinking about things like differentiating through simulations or gradient-based hyper-parameter optimization

**Sam Greydanus** @samgreydanus · Feb 18
Thanks. We've done some crazy things with autodiff over the past few years. @shoyer and I also differentiated through some topology optimization code in another post.

Oh, and you can differentiate through Hamiltonians and Lagrangians :D

1 3

**Stephan Hoyer** @shoyer · Feb 19
With appropriate tricks you can differentiate through any continuous function, no matter how it's calculated :)

1 3

**Gert-Jan** @GJ_Both · Feb 19
One thing I've realized the last year is that neural networks were a trojan horse for getting autodiff into physics :-)

I sometimes wish it were a little bit easier though, I'm working on a Sparse Bayesian Learning implicit layer but it has me beat...

## Example: Stellarator Coil Design in 25 lines of code

(Go to code)

I don't explain what the code is doing, sorry. You can find the code on my GitHub under "focus_tiny". Although Chris Smiet has added to the code and added extensive comments, so a more appropriate name might now be "focus_footnotesize".

## Automatic Differentiation (AD)

1. Practical
2. Elegant
3. Powerful

## AD is Elegant

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

I am going to simplify the theory somewhat, as in practice AD uses either "forward mode" or "reverse mode". I'll talk about these concepts later.

## Elegant in theory: the multivariate chain rule

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

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

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

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

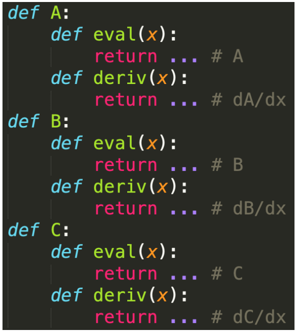
In the language of AD, $\boldsymbol{u}$ and $\boldsymbol{v}$ are primitive functions and $\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{v}}$ and $\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}$ are elementary partial derivatives.

**Bottom line**

$\boldsymbol{f}$ is built by composing primitive functions, and $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ is computed by multiplying elementary partial derivatives.

## Elegant in theory: AD is about composing derivatives

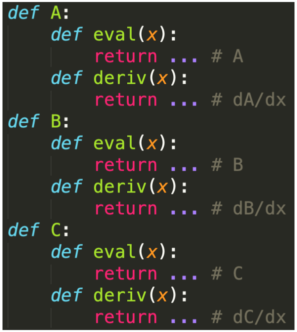
Suppose I create a library of primitive functions $A$, $B$, and $C$. I define how each primitive function is evaluated, and I define the derivative of each primitive function analytically:



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

## Elegant in theory: AD is about composing derivatives

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



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

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

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

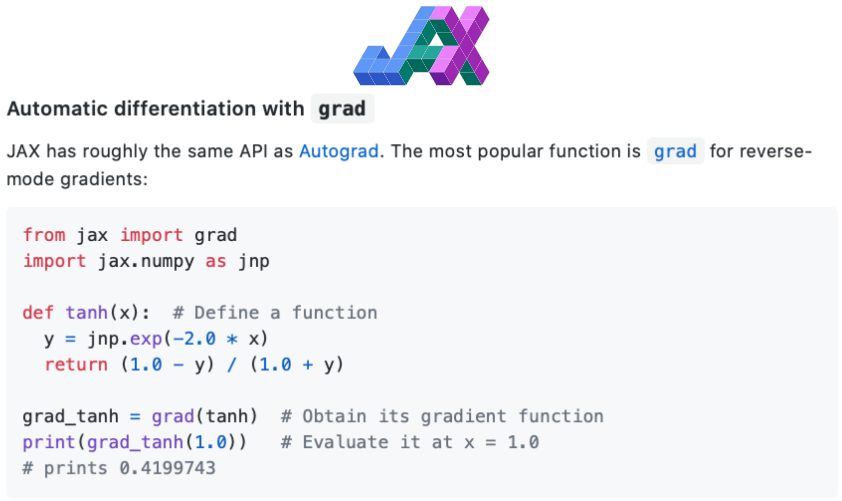
$$\frac{df_1}{dy} = \left(\frac{dC}{dx}\right)_{x=B(A(y))} \left(\frac{dB}{dx}\right)_{x=A(y)} \left(\frac{dA}{dx}\right)_{x=y}$$

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

$$f_2 = B(\ldots C(y))$$

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

## Elegant in practice: amazingly simple software tools



**Automatic differentiation with `grad`**

JAX has roughly the same API as Autograd. The most popular function is `grad` for reverse-mode gradients:

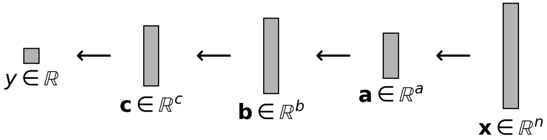
```
from jax import grad
import jax.numpy as jnp

def tanh(x):  # Define a function
  y = jnp.exp(-2.0 * x)
  return (1.0 - y) / (1.0 + y)

grad_tanh = grad(tanh)  # Obtain its gradient function
print(grad_tanh(1.0))   # Evaluate it at x = 1.0
# prints 0.4199743
```

## Differentiating a Function Composition

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

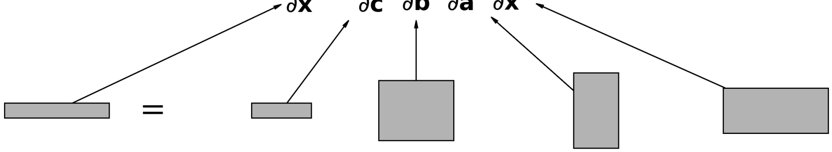
$$F = D \circ C \circ B \circ A$$

$$y = F(\mathbf{x})$$


$y \in \mathbb{R}$
$\mathbf{c} \in \mathbb{R}^c$
$\mathbf{b} \in \mathbb{R}^b$
$\mathbf{a} \in \mathbb{R}^a$
$\mathbf{x} \in \mathbb{R}^n$

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

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



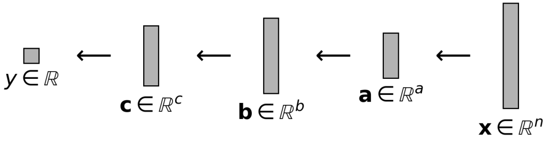
$$\frac{\partial y}{\partial \mathbf{x}} = \left[\frac{\partial y}{\partial x_1} \dots \frac{\partial y}{\partial x_n}\right] = \qquad \frac{\partial y}{\partial \mathbf{c}} \qquad \frac{\partial \mathbf{c}}{\partial \mathbf{b}} \qquad \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \qquad \frac{\partial \mathbf{a}}{\partial \mathbf{x}}$$

## Differentiating a Function Composition

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

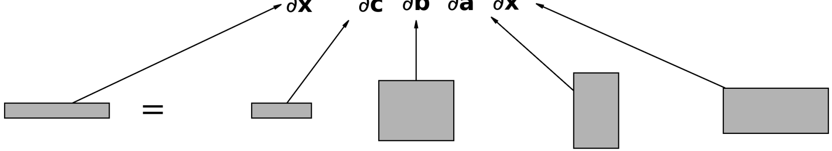
$$F = D \circ C \circ B \circ A$$

$$y = F(\mathbf{x})$$


$y \in \mathbb{R}$
$\mathbf{c} \in \mathbb{R}^c$
$\mathbf{b} \in \mathbb{R}^b$
$\mathbf{a} \in \mathbb{R}^a$
$\mathbf{x} \in \mathbb{R}^n$

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

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



$$\frac{\partial y}{\partial \mathbf{x}} = [\frac{\partial y}{\partial x_1} \ldots \frac{\partial y}{\partial x_n}] = \quad \frac{\partial y}{\partial \mathbf{c}} \quad \frac{\partial \mathbf{c}}{\partial \mathbf{b}} \quad \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \quad \frac{\partial \mathbf{a}}{\partial \mathbf{x}}$$

**Order matters!**

If $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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.

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

## 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

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

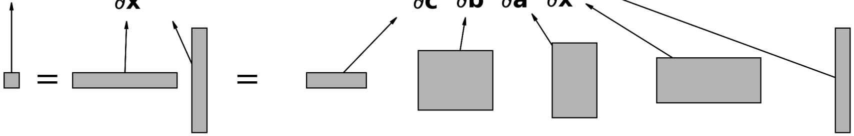
E.g., for a scalar function $\mathbb{R}^n \to \mathbb{R}$, a VJP gives the gradient.

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

## AD: Forward and Reverse Modes

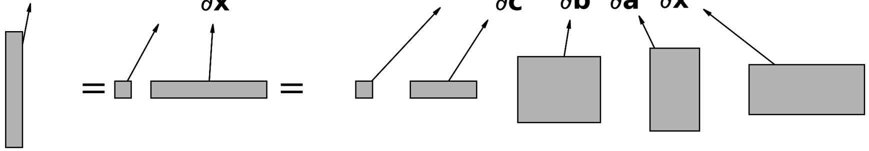
Forward mode: Jacobian-vector products (JVPs), build Jacobian one column at a time. $\boldsymbol{v}$ is a "tangent vector".

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



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

$$\mathbf{v}^T F'(\mathbf{x}) = \mathbf{v}^T\frac{\partial y}{\partial \mathbf{x}} = \mathbf{v}^T\frac{\partial y}{\partial \mathbf{c}}\frac{\partial \mathbf{c}}{\partial \mathbf{b}}\frac{\partial \mathbf{b}}{\partial \mathbf{a}}\frac{\partial \mathbf{a}}{\partial \mathbf{x}}$$



## 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 $\mathcal{O}(m)$.

## 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 $\mathcal{O}(m)$.

**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.

**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.

**Is there a connection to differential geometry?**

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

## 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 objective functions.

## Automatic Differentiation (AD)

1. Practical
2. Elegant
3. Powerful

## All of Computational Physics in One Slide

Linear equations

$$\boldsymbol{A}(\Omega)\boldsymbol{u} = \boldsymbol{b}(\Omega)$$

Time-dependent linear equations

$$\boldsymbol{A}(\boldsymbol{u}^t, \Omega)\boldsymbol{u}^{t+1} = \boldsymbol{b}(\boldsymbol{u}^t, \Omega)$$

Nonlinear equations

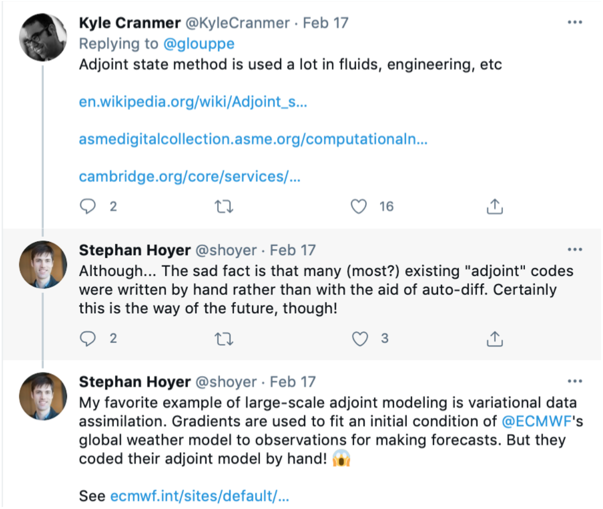
$$\boldsymbol{g}(\boldsymbol{u}, \Omega) = 0 \Rightarrow -\boldsymbol{g}(\boldsymbol{u}_i, \Omega) = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}(\boldsymbol{u}_{i+1} - \boldsymbol{u}_i)$$

Time-dependent nonlinear equations

$$\boldsymbol{g}(\boldsymbol{u}^t, \boldsymbol{u}^{t+1}, \Omega) = 0 \Rightarrow$$

$$-\boldsymbol{g}(\boldsymbol{u}^t, \boldsymbol{u}_i^{t+1}, \Omega) = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}(\boldsymbol{u}_{i+1}^{t+1} - \boldsymbol{u}_i^{t+1})$$

# Adjoint Methods



**Kyle Cranmer** @KyleCranmer · Feb 17
Replying to @glouppe
Adjoint state method is used a lot in fluids, engineering, etc

en.wikipedia.org/wiki/Adjoint_s...

asmedigitalcollection.asme.org/computationaln...

cambridge.org/core/services/...

2 16

**Stephan Hoyer** @shoyer · Feb 17
Although... The sad fact is that many (most?) existing "adjoint" codes were written by hand rather than with the aid of auto-diff. Certainly this is the way of the future, though!

2 3

**Stephan Hoyer** @shoyer · Feb 17
My favorite example of large-scale adjoint modeling is variational data assimilation. Gradients are used to fit an initial condition of @ECMWF's global weather model to observations for making forecasts. But they coded their adjoint model by hand! 😱

See ecmwf.int/sites/default/...

## 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 $\boldsymbol{u}$.

## 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

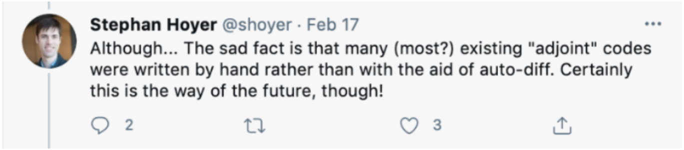
The adjoint method computes the VJP of a constraint equation

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

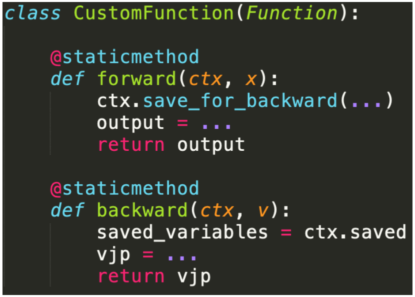
$$\begin{tikzcd}
\vec{p} \arrow[r, shift left] & \vec{g} \arrow[l, shift left, "\vec{\lambda}^T \frac{\partial \vec{g}}{\partial \vec{p}}"] \arrow[r, shift left] & \vec{u} \arrow[l, shift left, "\vec{\lambda}"] \arrow[r, shift left] & f \arrow[l, shift left, "\frac{\partial f}{\partial \vec{u}}"]
\end{tikzcd}$$

- 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 $\boldsymbol{g} = 0$.

## Why AD is Powerful: Custom Primitives


**Stephan Hoyer** @shoyer · Feb 17

Although... The sad fact is that many (most?) existing "adjoint" codes were written by hand rather than with the aid of auto-diff. Certainly this is the way of the future, though!


```
class CustomFunction(Function):

    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(...)
        output = ...
        return output

    @staticmethod
    def backward(ctx, v):
        saved_variables = ctx.saved
        vjp = ...
        return vjp
```

### Bottom line

Automatic differentiation can perform the adjoint method using custom primitive functions. This allows AD to be used in computational physics.

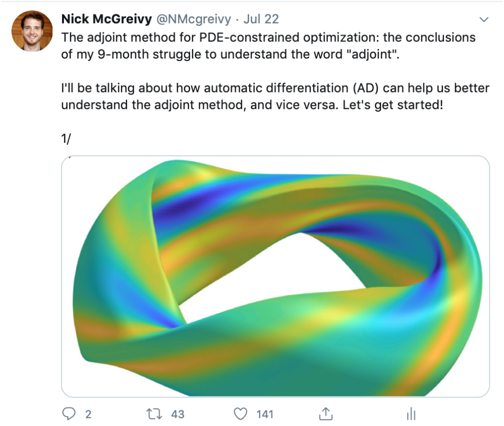
## Why would you not compute derivatives with AD?

- 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)

## 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.



**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/

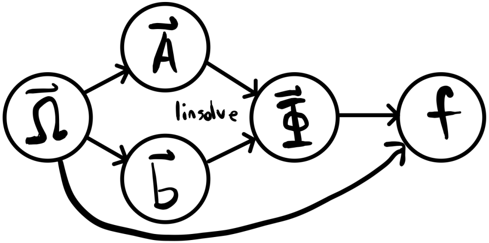
## Additional Slides

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

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

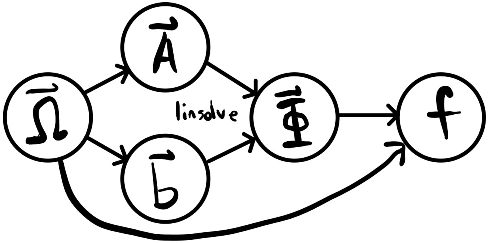


## Review: AD and the adjoint method for linear equations

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

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

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



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

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 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

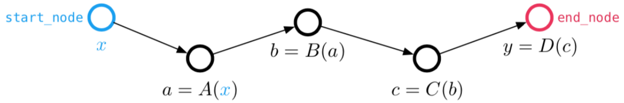
**The problem with reverse mode AD**

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.

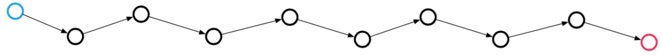
Checkpointing is a method to reduce memory requirements in exchange for increased runtime. It works by storing "checkpoints" at various points in the program and recomputing the data between checkpoints that would otherwise be stored in memory.

## Checkpointing

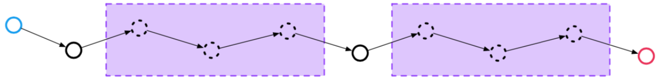
Our example function can be visualized as a graph:



Suppose we have a function which looks like this:

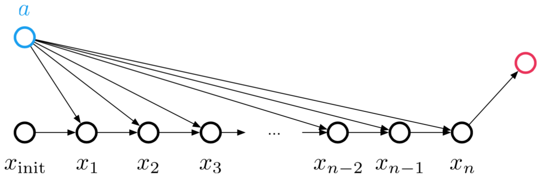


Let's use checkpointing to reduce the memory from the part of the computation in the purple boxes. We place checkpoints to the left of the purple boxes and recompute the functions in the purple boxes on the backwards pass.



## AD and the adjoint method for iterative algorithms

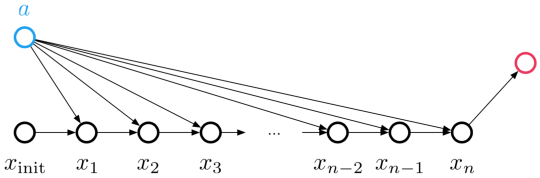
Suppose we have an iterative algorithm for solving some set of equations, and we want to compute the derivative of the solution with respect to some parameter $a$. Let $x_{\text{init}}$ be an initial guess, and suppose the iterative algorithm runs $n$ times before converging to a solution. This computation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