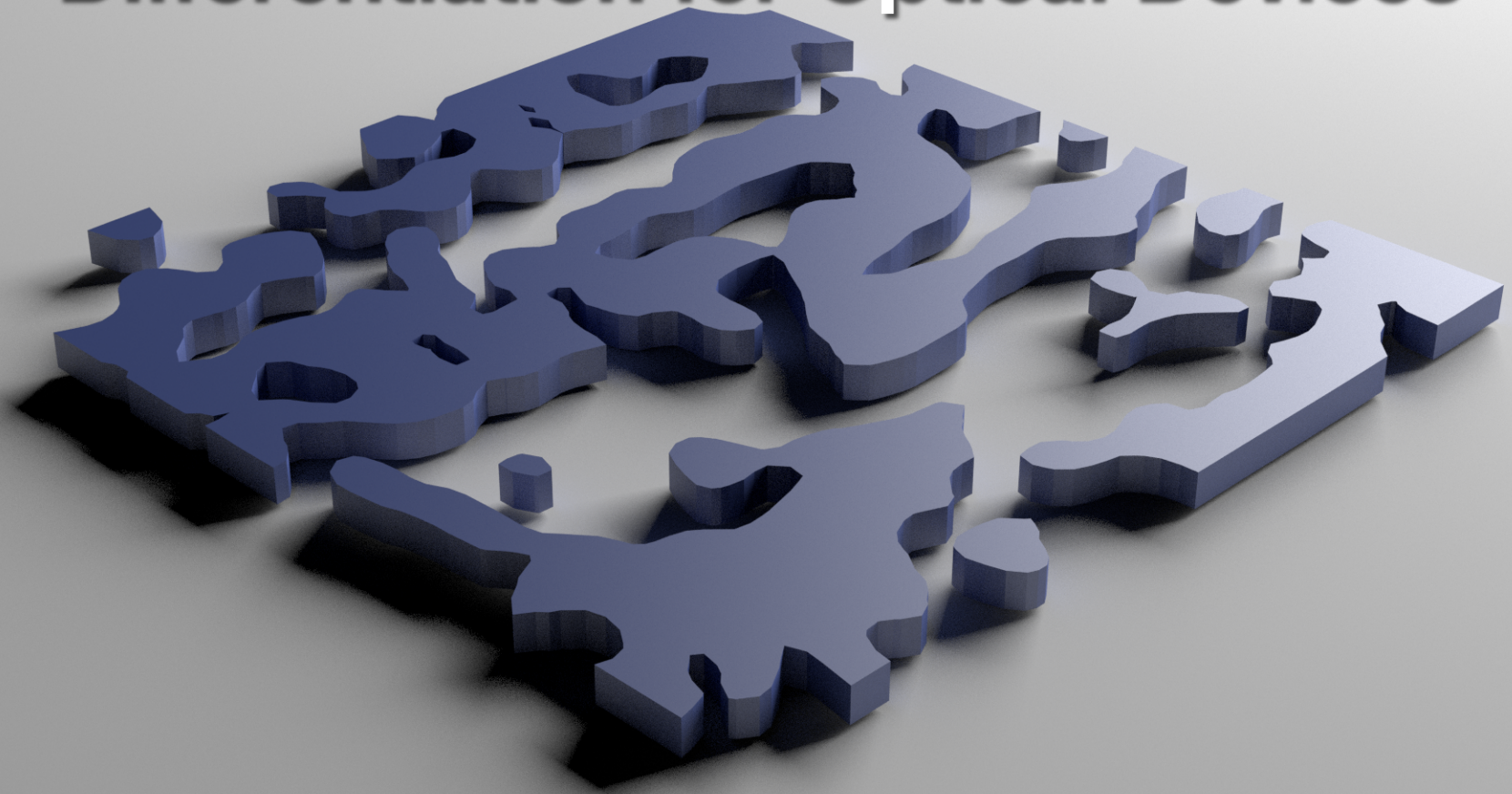
Inverse Design and Automatic Differentiation for Optical Devices

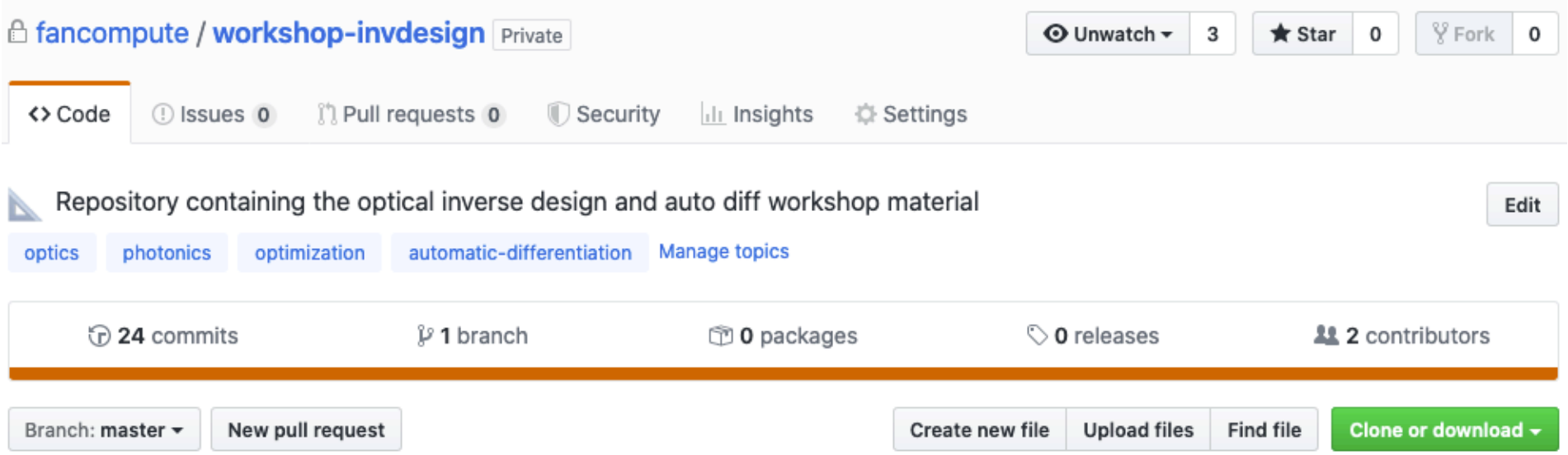


Ian Williamson and **Momchil Minkov**

Stanford University Optical Society Workshop

Nov 21, 2019

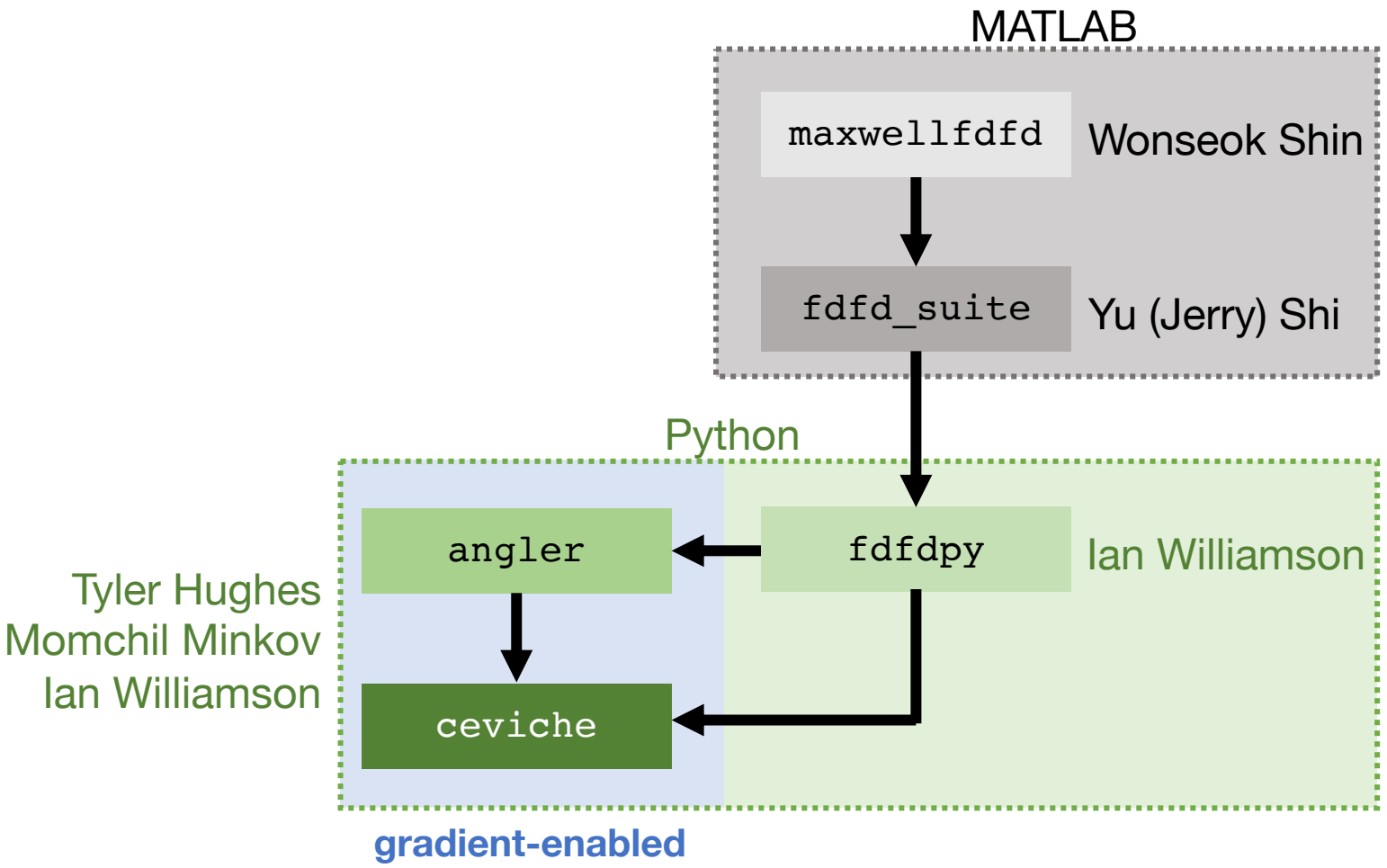
`https://github.com/fancompute/workshop-invdesign`



Workshop outline

- Brief crash course on optical simulation
  - Theory of numerically solving Maxwell's equations
  - Practical demonstration with our open source code
- Inverse design theory / practical introduction
  - Adjoint gradient computation
  - Automatic differentiation
  - Connection to machine learning
- Device parameterization techniques
  - Binarizing features
  - Optimization penalties

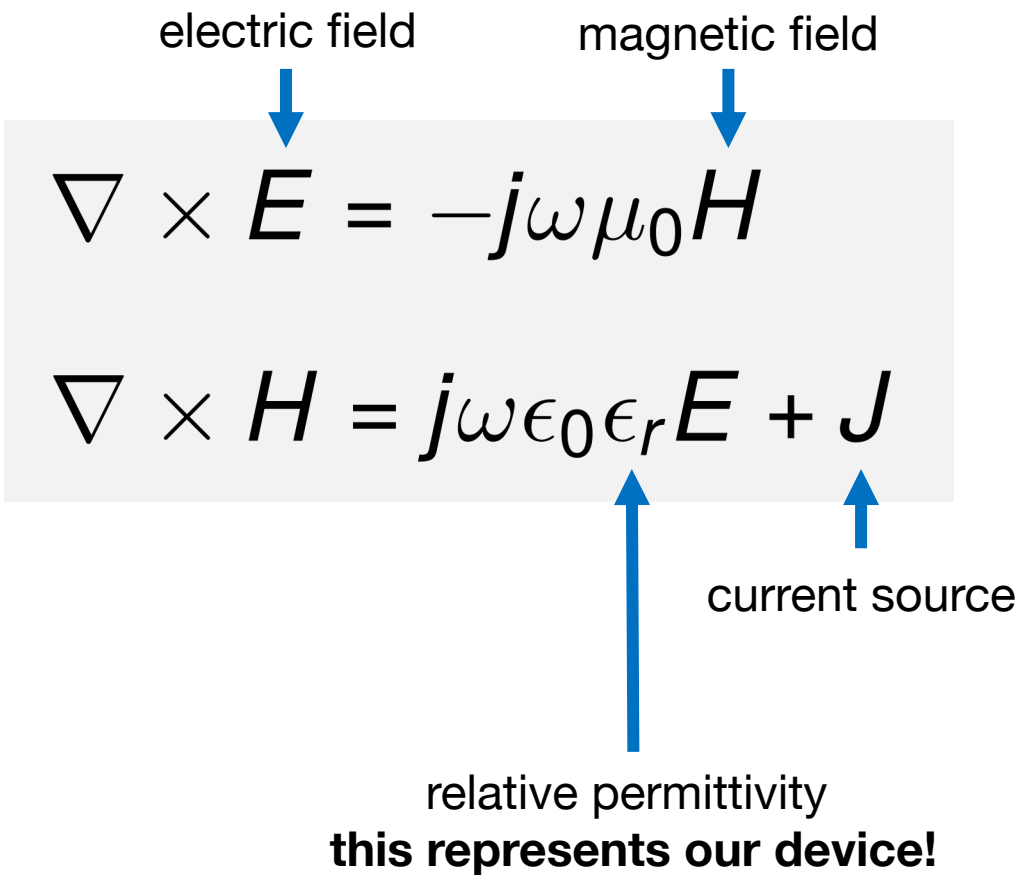
A history of optical simulation packages *(from our group)*



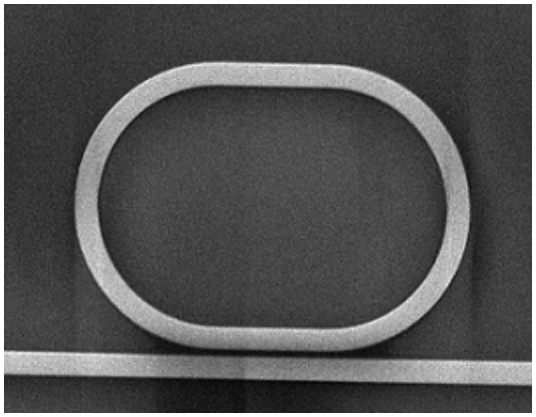
A quick crash course in optical simulation

How do we simulate optical devices?

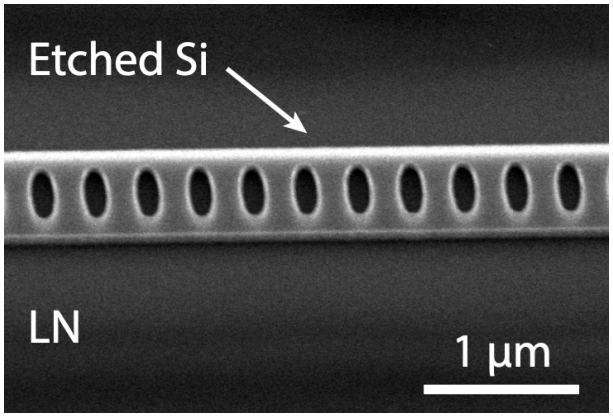
... we solve Maxwell's equations!



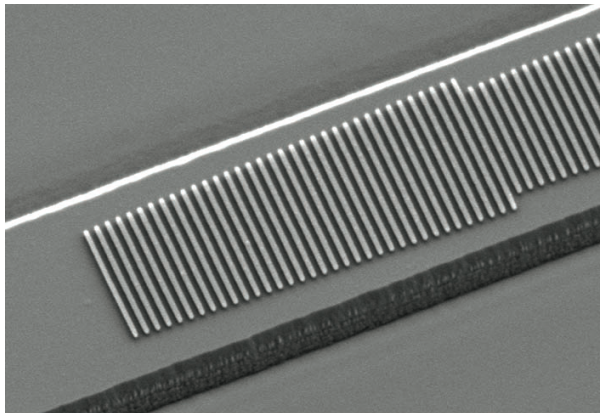
Welcome to the optical device zoo



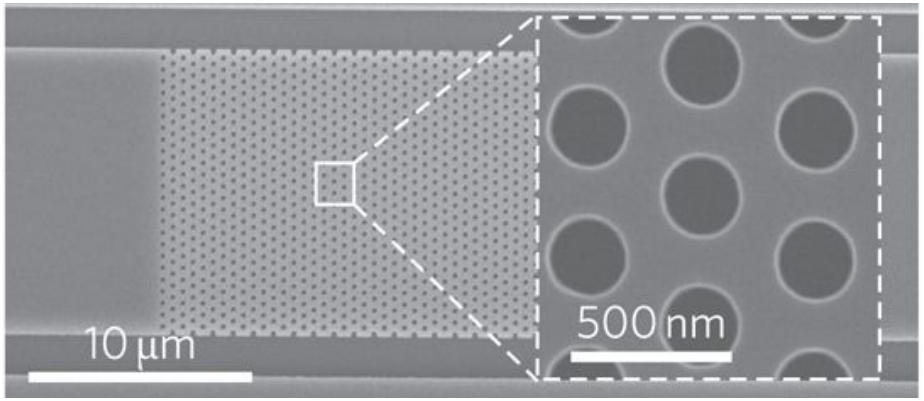
Stern Group (Northwestern Univ)



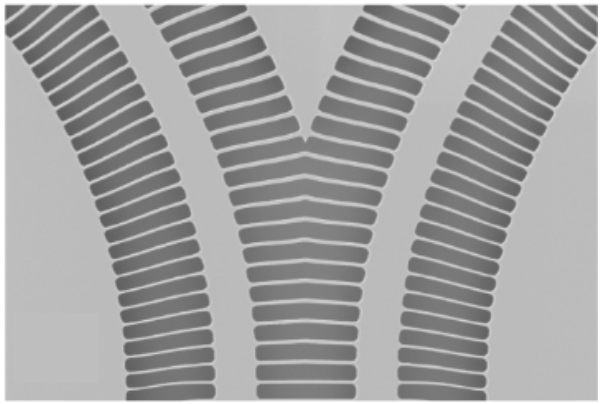
Witmer et al. Opt. Express 24, 5876-5885 (2016)



Wang et al. Nat. Comm. 8, 2098 (2017)



Kocaman et al. Nat. Pho. 5, 499–505 (2011)



Soler Penades et al. Opt. Express 24, 22908-22916 (2016)

Constructing a set of equations to solve

$$\nabla \times E = -j\omega\mu_0 H$$
$$\nabla \times H = j\omega\epsilon_0\epsilon_r E + J$$

Some assumptions we make:

- A two-dimensional domain (invariant along *z*-direction)
- All fields are also invariant along *z*-direction
- Electric-field polarized only along *z*-direction

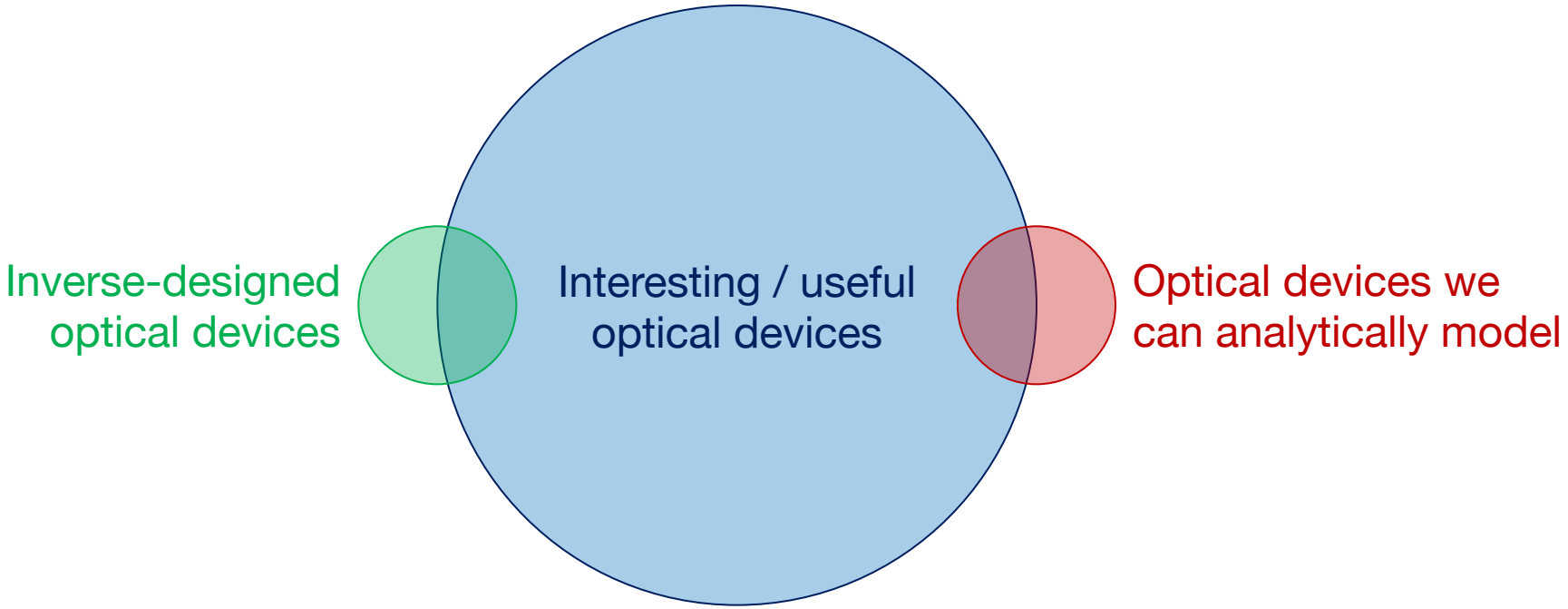
$$\partial_y E_z = -j\omega\mu_0 H_x$$
$$\partial_x E_z = j\omega\mu_0 H_y$$
$$\partial_x H_y - \partial_y H_x = j\omega\epsilon E_z + J_z$$

Constructing a set of equations to solve

$$\partial_y E_z = -j\omega\mu_0 H_x$$
$$\partial_x E_z = j\omega\mu_0 H_y$$
$$\partial_x H_y - \partial_y H_x = j\omega\epsilon E_z + J_z$$

$$\partial_x \partial_x E_z + \partial_y \partial_y E_z + \omega^2 \mu_0 \epsilon E_z = j\omega\mu_0 J_z$$

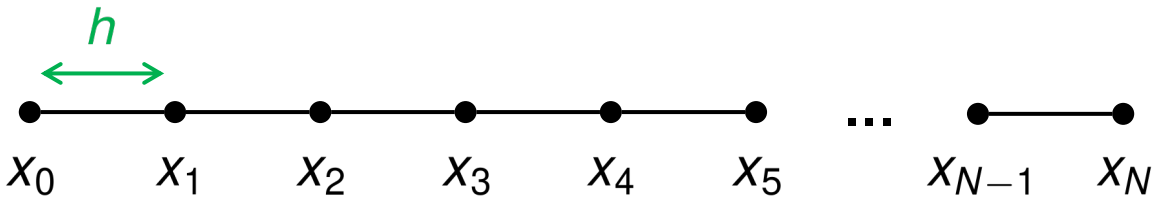
Where can these equations take us?



$$\partial_x \partial_x E_z + \partial_y \partial_y E_z + \omega^2 \mu_0 \epsilon E_z = j\omega\mu_0 J_z$$

The key idea behind finite differences

$$\frac{\partial y}{\partial x} = F(x) \longrightarrow \frac{\partial y}{\partial x} \approx \frac{y(x+h) - y(x)}{h}$$



- Very simple concept that can be scaled up to solve complex PDEs
- Convergence to the exact solution at a rate proportional to *h* (depends on “stencil”)

Discretizing Maxwell's equations

$$\partial_x \mu_0^{-1} \partial_x E_z + \partial_y \mu_0^{-1} \partial_y E_z + \omega^2 \epsilon E_z = j\omega J_z$$

continuous PDE

$$\left[D_x^b \mu_0^{-1} D_x^f + D_y^b \mu_0^{-1} D_y^f + \omega^2 \text{diag}(\epsilon) \right] e_z = (j\omega) j_z$$

discretized PDE

Along just **one** dimension, this looks like...

$$D_x^f = \frac{1}{\Delta x} \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ 1 & & & & -1 \end{bmatrix}$$

Along **two** dimensions, the representation is a bit more complex, *but the same idea applies!*

Discretizing Maxwell's equations

$$\begin{bmatrix} e_{00} & e_{01} & \cdots & e_{0N} \\ e_{10} & e_{11} & & \\ \vdots & & \ddots & \\ e_{N0} & & & e_{NN} \end{bmatrix}$$

spatial representation

$$\left[D_x^b \mu_0^{-1} D_x^f + D_y^b \mu_0^{-1} D_y^f + \omega^2 \mathrm{diag}(\epsilon)\right] e_z = (j\omega) j_z$$

discretized PDE

$$\begin{bmatrix} e_{00} & \cdots & e_{N0} & e_{01} & \cdots & e_{N1} & \cdots & e_{0N} & \cdots & e_{NN} \end{bmatrix}^T$$

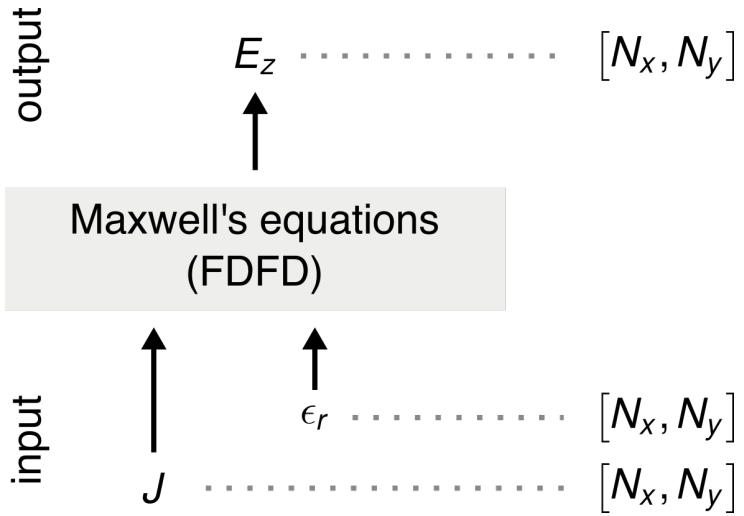
flattened representation

sparse linear problem: $A \cdot e = b \rightarrow A = f(\epsilon) \rightarrow e = A^{-1} b$

The `ceviche` package (at a high level)

$$\left[D_x^b \mu_0^{-1} D_x^f + D_y^b \mu_0^{-1} D_y^f + \omega^2 \text{diag}(\epsilon)\right] e_z = (j\omega) j_z$$

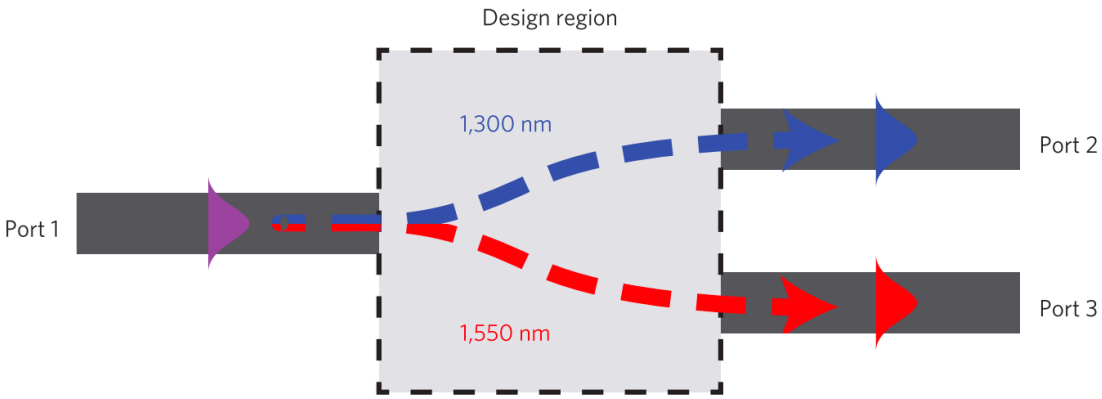
$$e = A^{-1} b$$

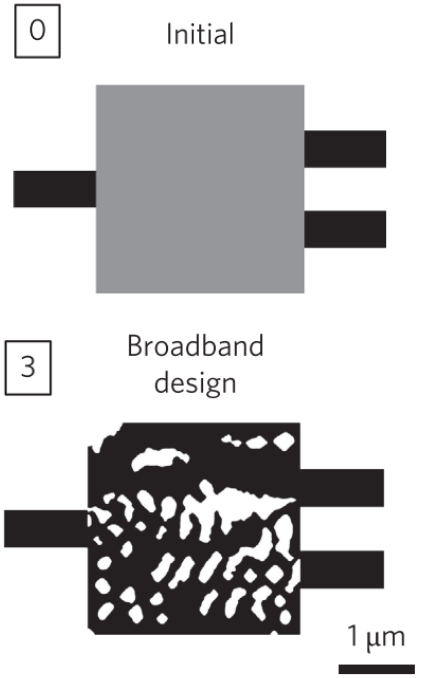


Notebook #01

Inverse design

- Assume that we can “mold” the permittivity
- Try to get the **best** possible structure for a particular desired functionality.

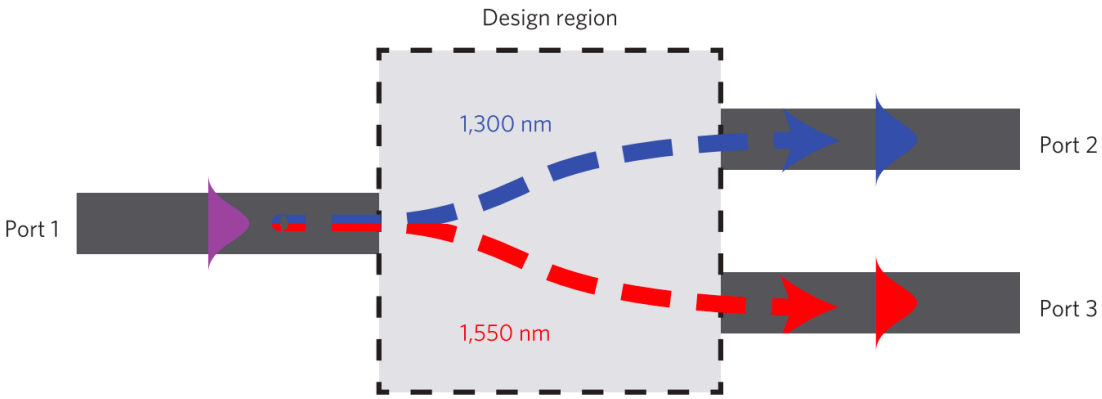


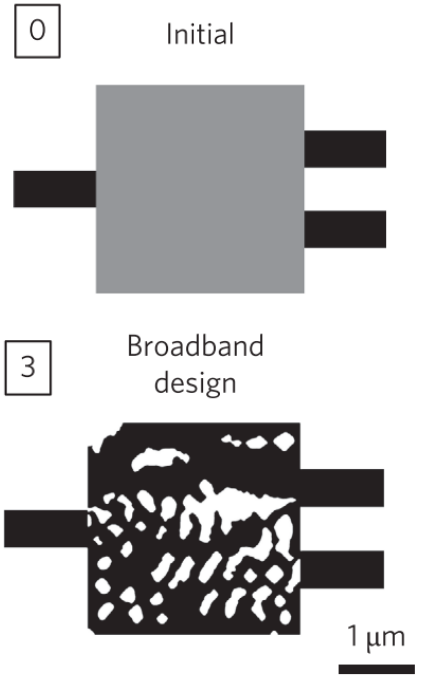


Piggott, A. et al., *Nature Photonics*, 374–377 (2015)

Inverse design

- We want the gradient of some objective function w.r.t. the permittivity at every point
- Efficiently!



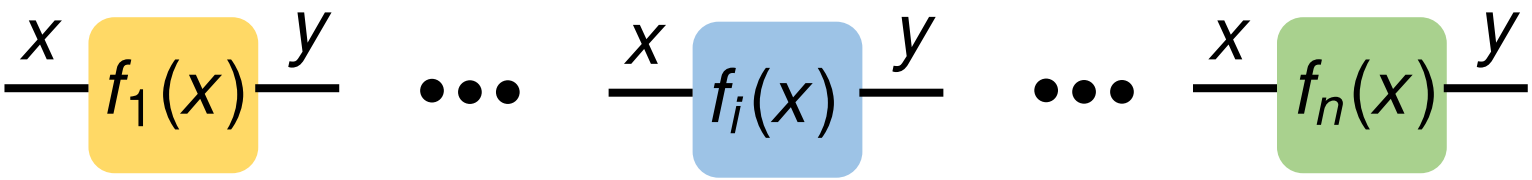


Piggott, A. et al., *Nature Photonics*, 374–377 (2015)

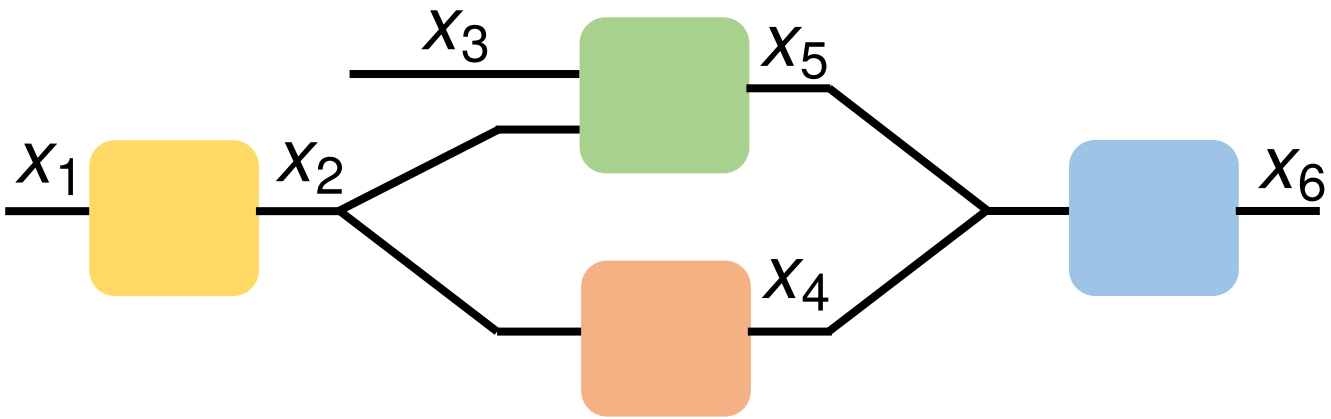
Automatic differentiation

or: differentiable programming

Usual programming: library of functions

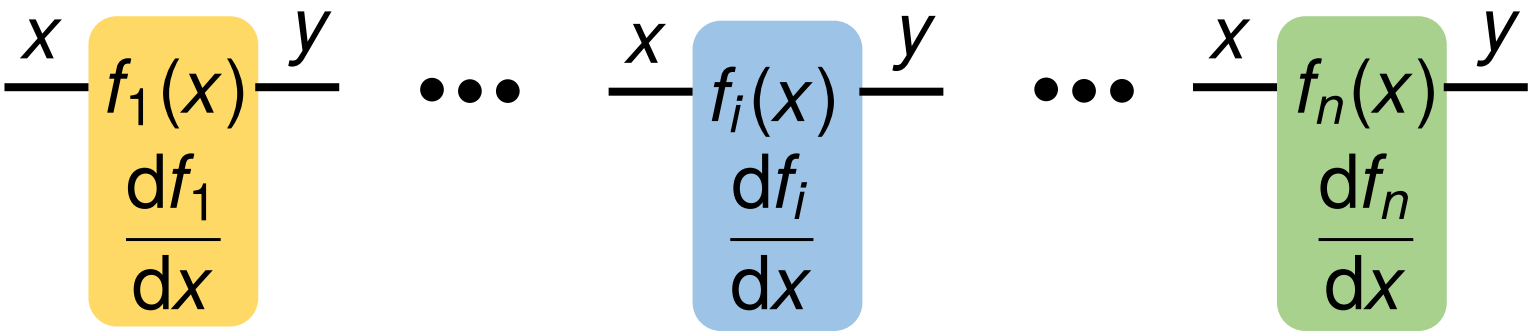


Program = “computational graph”

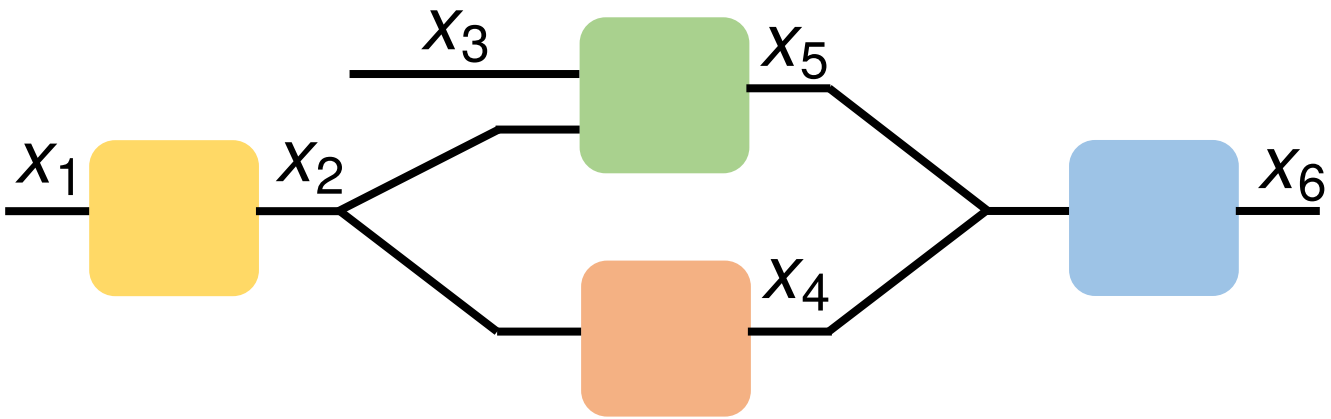


Automatic differentiation

Differentiable programming library:



Track a way to compute any dx_i/dx_j



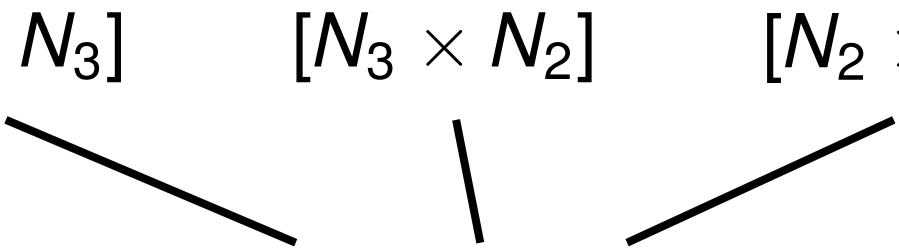
Forward/reverse autodiff

$$x_4 = f_3(f_2(f_1(x_1)))$$

$$\frac{\mathrm{d}x_4}{\mathrm{d}x_1} = \frac{\mathrm{d}x_4}{\mathrm{d}x_3}\frac{\mathrm{d}x_3}{\mathrm{d}x_2}\frac{\mathrm{d}x_2}{\mathrm{d}x_1}$$

$[N_4 \times N_3]$ $[N_3 \times N_2]$ $[N_2 \times N_1]$

Forward/reverse autodiff



$$\frac{\mathrm{d}x_4}{\mathrm{d}x_1} = \frac{\mathrm{d}x_4}{\mathrm{d}x_3}\frac{\mathrm{d}x_3}{\mathrm{d}x_2}\frac{\mathrm{d}x_2}{\mathrm{d}x_1}$$

$$\frac{\mathrm{d}x_4}{\mathrm{d}x_1} = \frac{\mathrm{d}x_4}{\mathrm{d}x_3}\left(\frac{\mathrm{d}x_3}{\mathrm{d}x_2}\frac{\mathrm{d}x_2}{\mathrm{d}x_1}\right) \quad \text{——} \quad \mathcal{O}(N_1N_2N_3 + N_1N_3N_4)$$

$$\frac{\mathrm{d}x_4}{\mathrm{d}x_1} = \left(\frac{\mathrm{d}x_4}{\mathrm{d}x_3}\frac{\mathrm{d}x_3}{\mathrm{d}x_2}\right)\frac{\mathrm{d}x_2}{\mathrm{d}x_1} \quad \text{——} \quad \mathcal{O}(N_2N_3N_4 + N_1N_2N_4)$$

$$\frac{dx_4}{dx_1} = \frac{dx_4}{dx_3}\left(\frac{dx_3}{dx_2}\frac{dx_2}{dx_1}\right) \quad \text{—} \quad \text{Forward-mode } \mathcal{O}(N_1N_2N_3 + N_1N_3N_4)$$

$$\frac{dx_4}{dx_1} = \left(\frac{dx_4}{dx_3}\frac{dx_3}{dx_2}\right)\frac{dx_2}{dx_1} \quad \text{—} \quad \text{Reverse-mode } \mathcal{O}(N_2N_3N_4 + N_1N_2N_4)$$

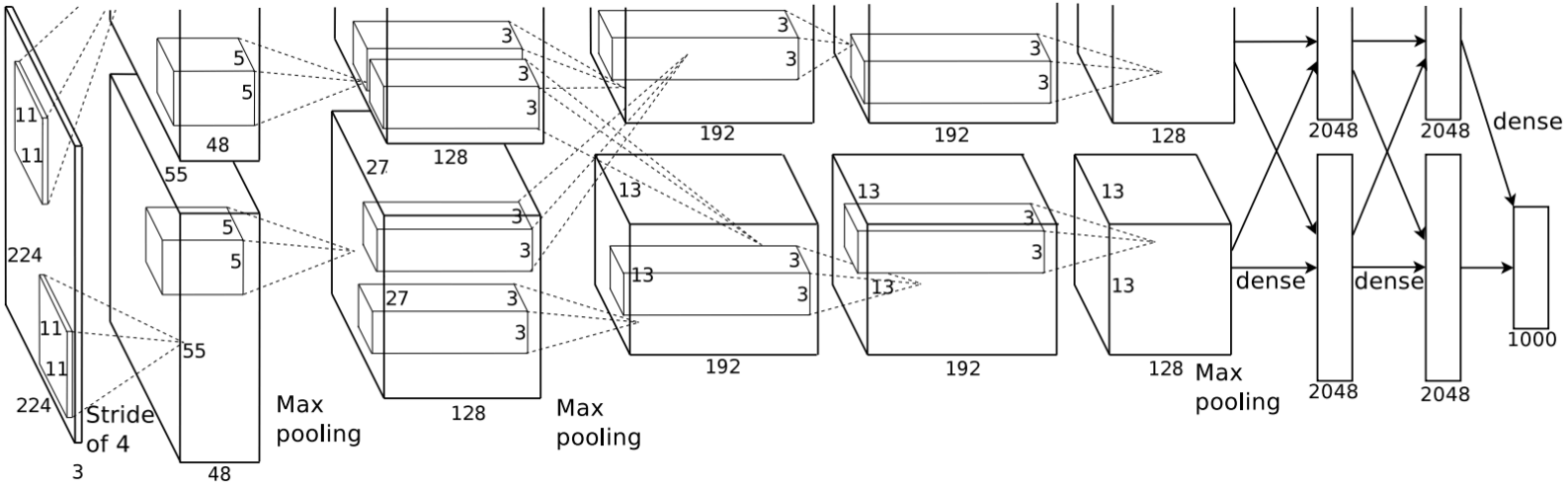
Reverse-mode autodiff always better when

$N_4 \ll N_1$ (few outputs, many inputs)

In optimization problems, this is always the case! Final objective function is just one number.

Reverse-mode autodiff

It's backpropagation in machine learning lingo!

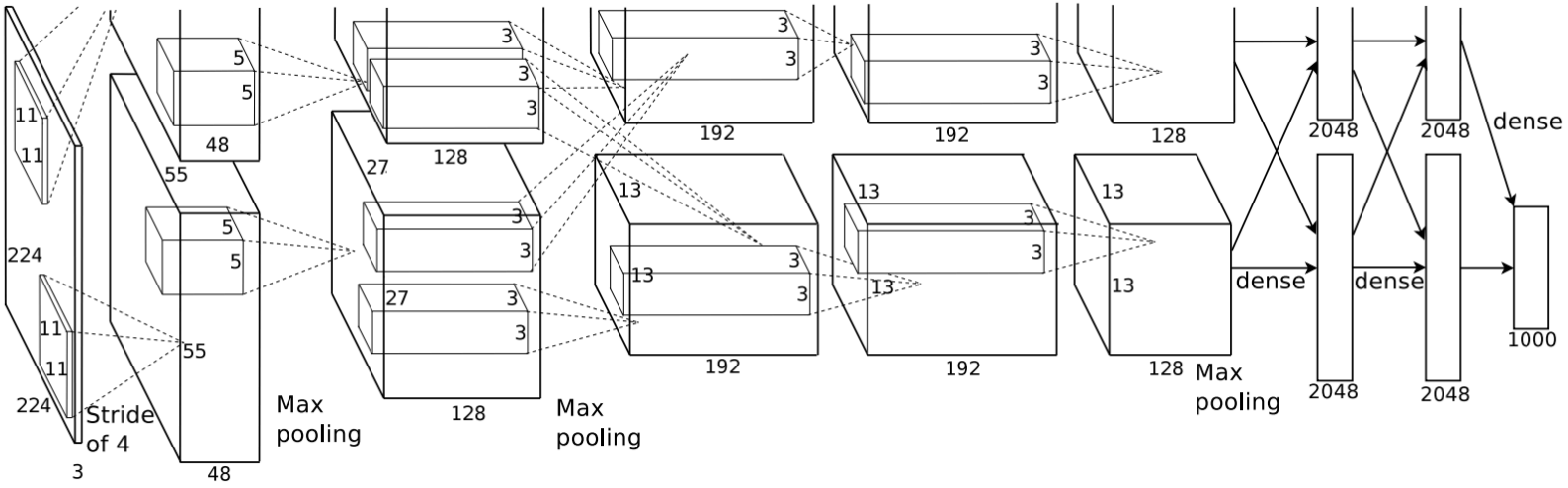


62 **million** parameters… one final output!

"AlexNet", Krizhevsky et al., NIPS (2012)

Reverse-mode autodiff

It's backpropagation in machine learning lingo!

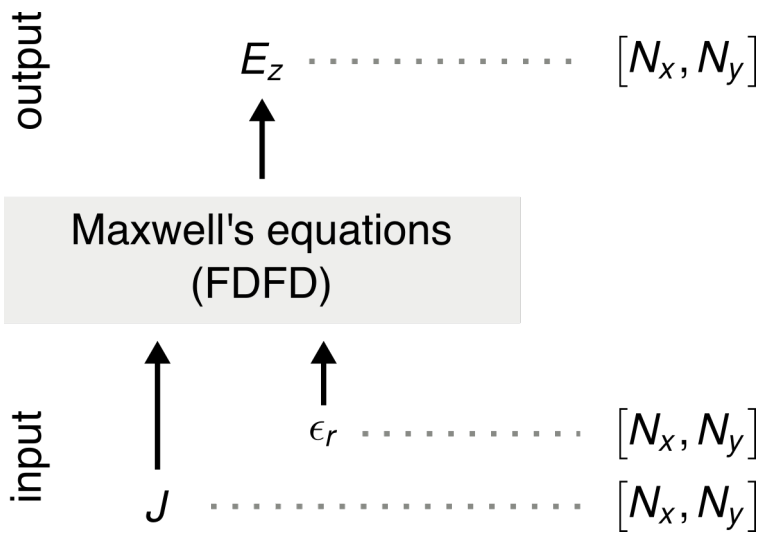


TensorFlow, PyTorch, Autograd et al. are all, in their heart, automatic differentiation libraries

"AlexNet", Krizhevsky et al., NIPS (2012)

Adjoint variable method

In photonics inverse design, it refers to how we "backprop" through the field computation

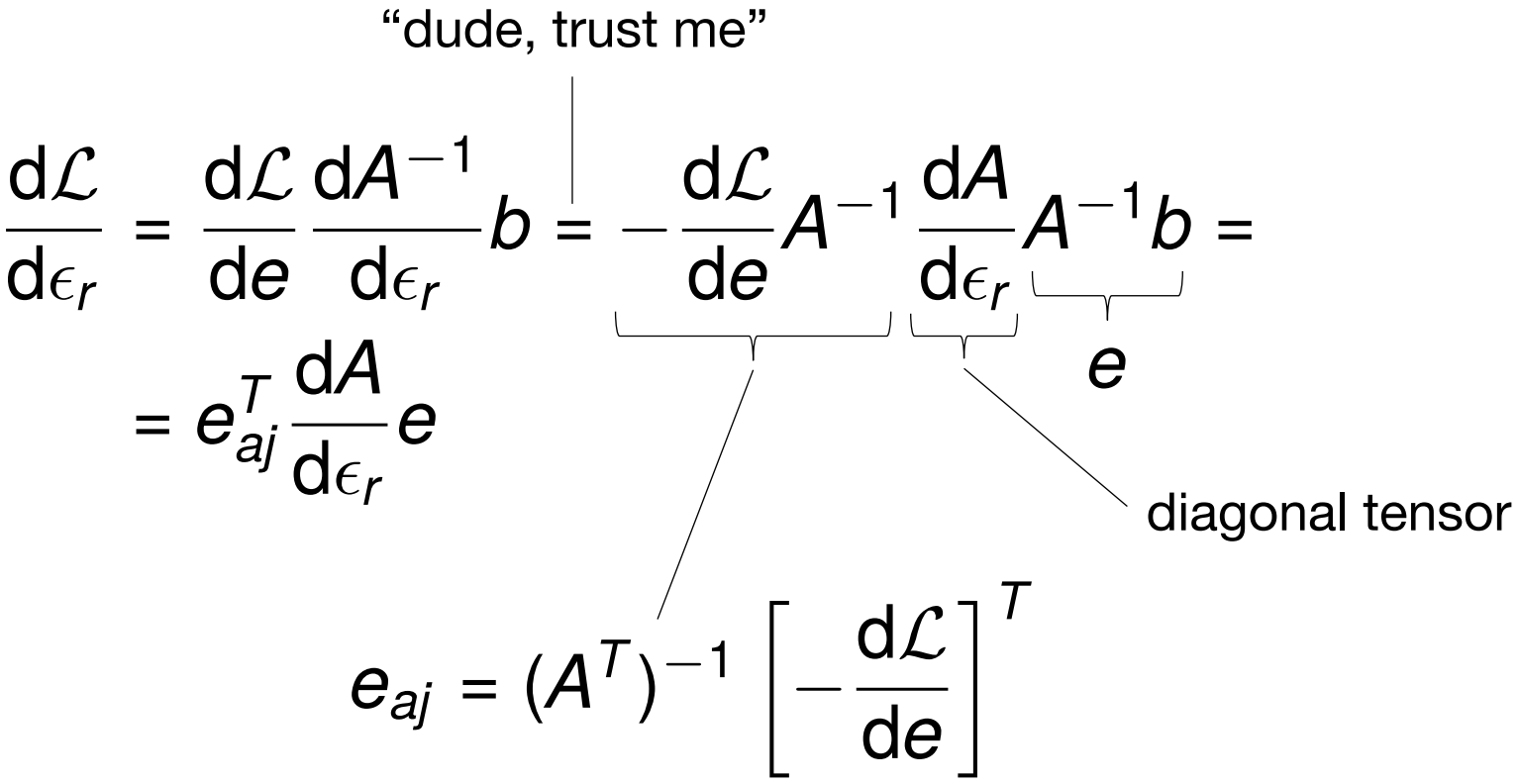


$$Ae = b \qquad e = A^{-1}b$$

$$\frac{d\mathcal{L}}{d\epsilon_r} = \frac{d\mathcal{L}}{de}\frac{de}{d\epsilon_r}$$

Adjoint variable method

In photonics inverse design, it refers to how we "backprop" through the field computation



Adjoint variable method

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\epsilon_r} = e_{aj}^T \frac{\mathrm{d}A}{\mathrm{d}\epsilon_r} e \qquad e_{aj} = (A^T)^{-1} \left[-\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}e} \right]^T$$

For a “reciprocal” system (very often), $A^T = A$

$$e = A^{-1} b \qquad e_{aj} = A^{-1} b_{aj}$$

The adjoint field is a solution to **the same Maxwell** problem, but with a **different source** that depends on the objective function and the original field.

Autograd

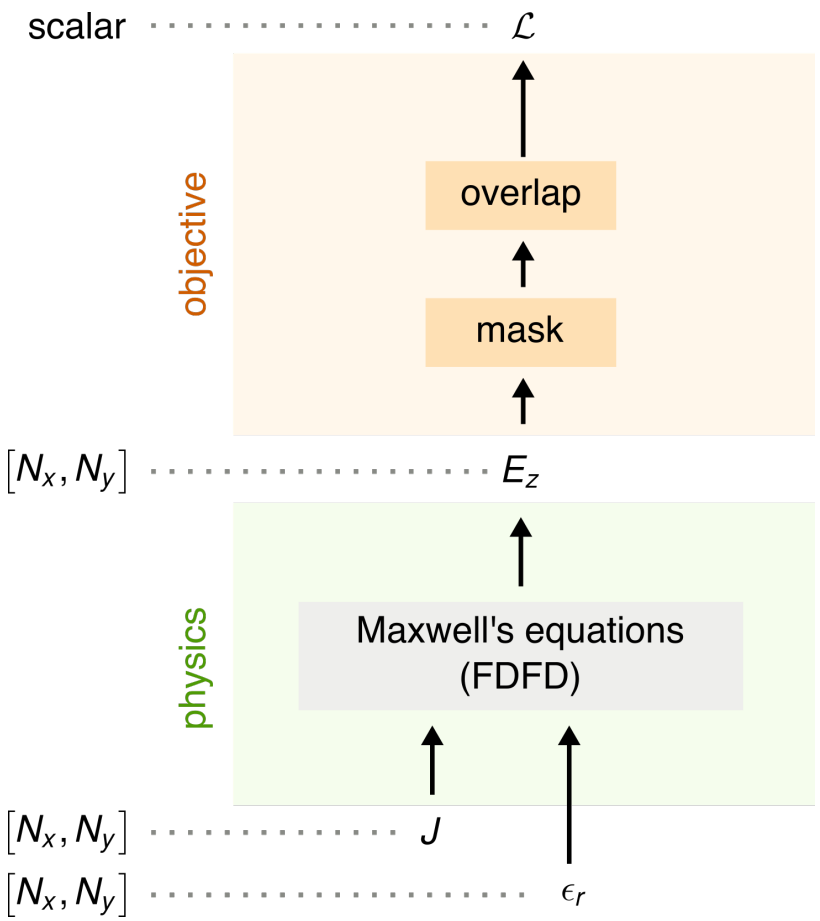
github.com/HIPS/autograd

- The *autograd* package from Harvard (HIPS) is an autodiff library for NumPy and SciPy
- In many cases, you can just use it blindly, and then get all the gradients you need!
- You can also extend it by telling *autograd* how to differentiate through custom functions

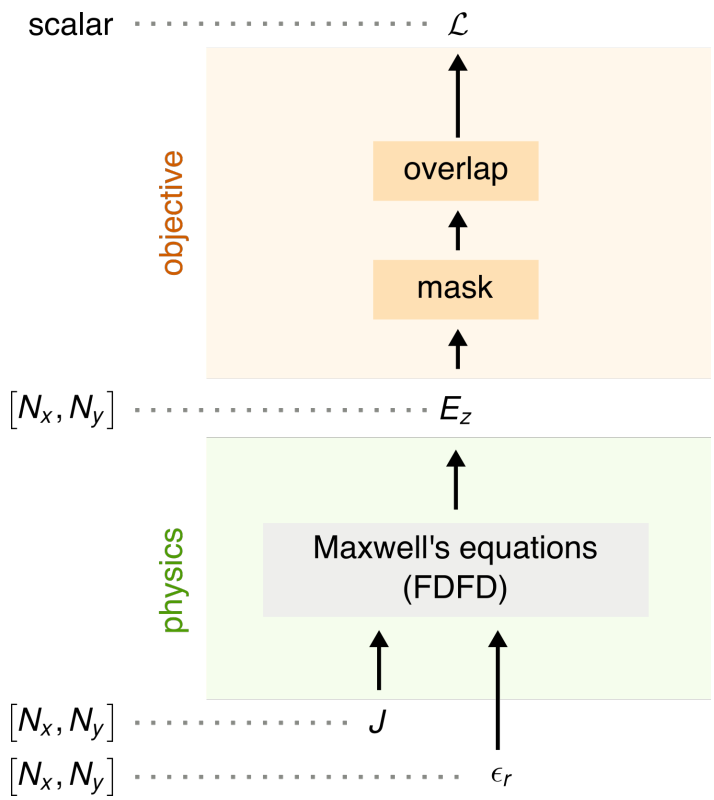
Ceviche

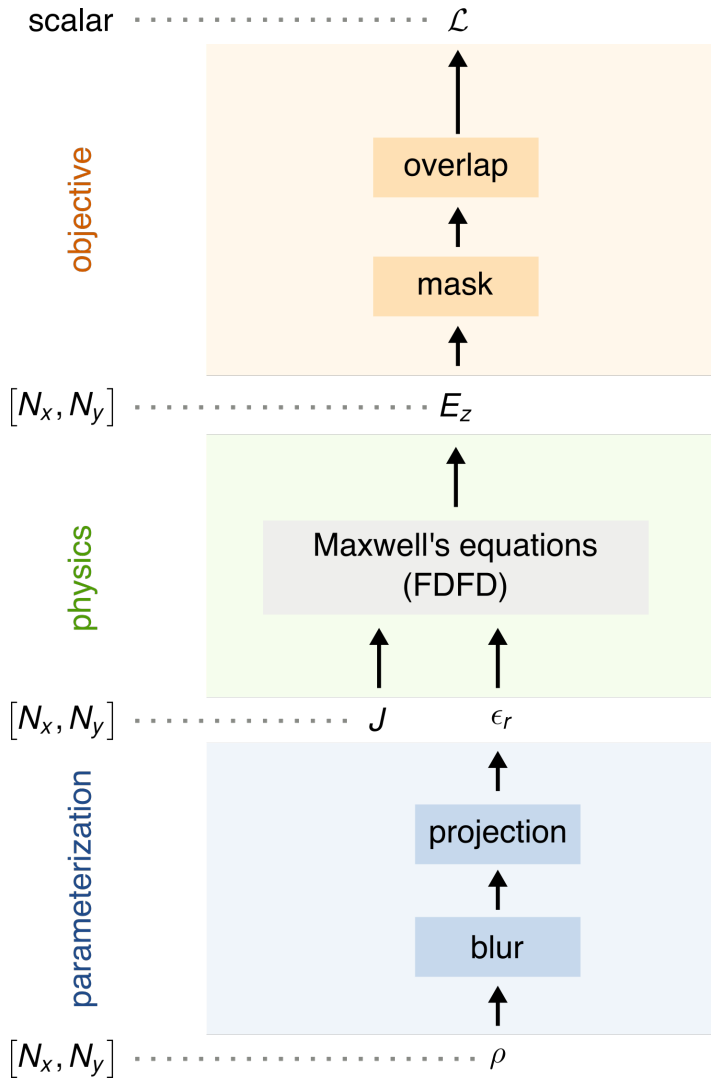
In *ceviche* we've interfaced the AVM with *autograd*

This allows great flexibility in building optimization graphs!

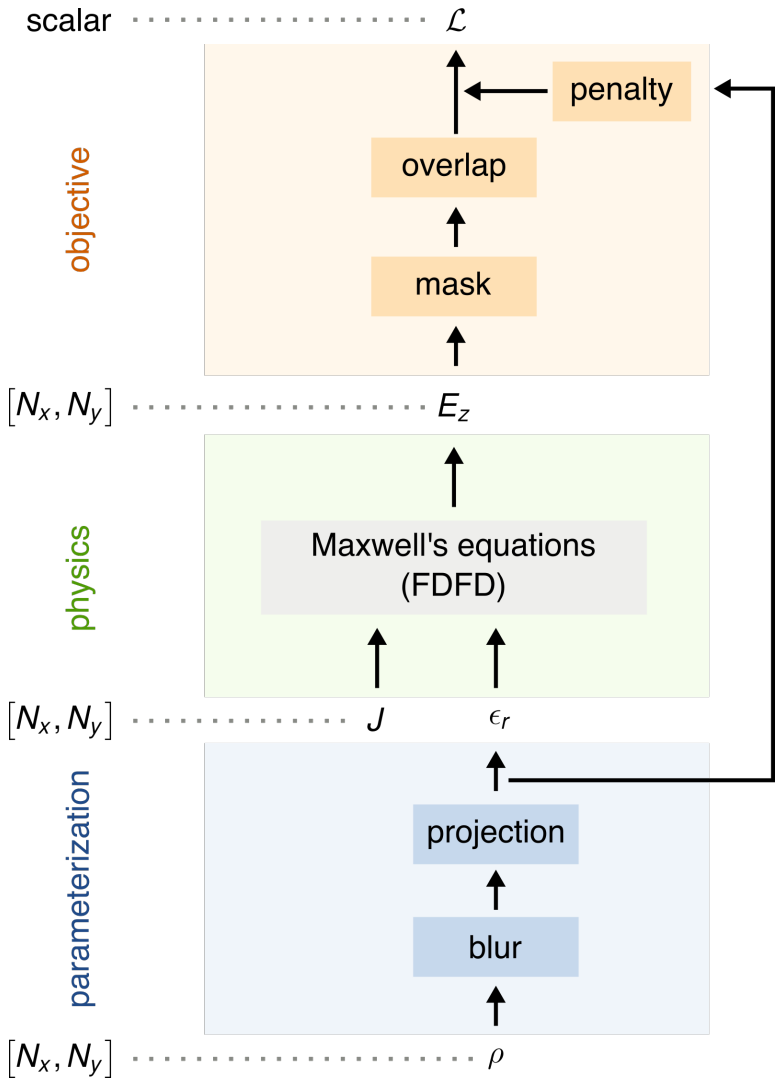


Notebook #02

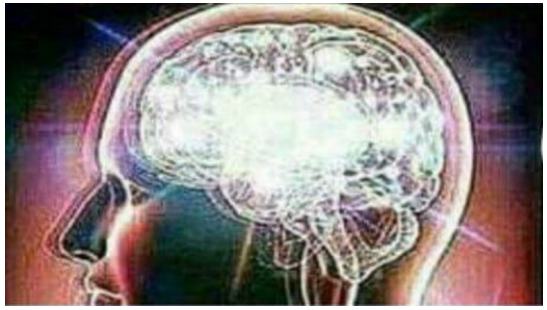

scalar
$\mathcal{L}$
objective
overlap
mask
$[N_x, N_y]$
E_z
physics
Maxwell's equations
(FDFD)
$[N_x, N_y]$
J
$[N_x, N_y]$
ϵ_r


scalar
$\mathcal{L}$
objective
overlap
mask
$[N_x, N_y]$
E_z
physics
Maxwell's equations
(FDFD)
$[N_x, N_y]$
J
ϵ_r
parameterization
projection
blur
$[N_x, N_y]$
ρ

Notebook #03


scalar
$\mathcal{L}$
objective
penalty
overlap
mask
$[N_x, N_y]$
E_z
physics
Maxwell's equations
(FDFD)
$[N_x, N_y]$
J
ϵ_r
parameterization
projection
blur
$[N_x, N_y]$
ρ

The **adjoint variable method** lets us efficiently compute the gradients in *PDE-constrained optimization problems* (inverse design)



The **backpropagation** algorithm is the equivalent of the *adjoint method* for training neural networks

Automatic differentiation is the generalization of both *backpropagation* and the *adjoint method* to arbitrary *computational graphs (programs)*

We can utilize **automatic differentiation** to efficiently optimize complex models that freely combine physics, machine learning, and other computations

Final thoughts (homework?)

- Viewing parameterization as a sequence of ***composable transformations*** is powerful!
  - ML libraries can be useful for things other than ML!
- Hyper parameters are important!
  - *penalty term*, *binarization*, etc.
  - *Scheduling* (adjusting hyper params during progress of optimization)
- Must go to simulations which capture 3D features to make real devices
  - but doesn’t necessarily mean that you can’t prototype in 2D!

Other resources

`metanet.stanford.edu`

A collection of resources being setup by Jiaqi Jiang / Jonathan Fan

Stanford University
- Shanhui Fan
- Jonathan Fan
- Jelena Vuckovic

UC Berkeley
- Eli Yablonovitch

Princeton University
- Alejandro Rodriguez

MIT
- Steven Johnson

Technical University of Denmark
- Ole Sigmund

... and **many** others!

`https://github.com/fancompute/workshop-invdesign`