Matrix Calculus lecture notes:

How can we use so many derivatives?

… a couple of applications

… and the “adjoint method”

Matrix Calculus, IAP 2023
Profs. Steven G. Johnson & Alan Edelman, MIT

Newton’s method: Nonlinear equations via Linearization

scalar out *scalar in*

18.01: solving **$f(x) = 0$**:

1. Linearize:

   $$f(x+\delta x) \approx f(x) + f'(x)\delta x$$

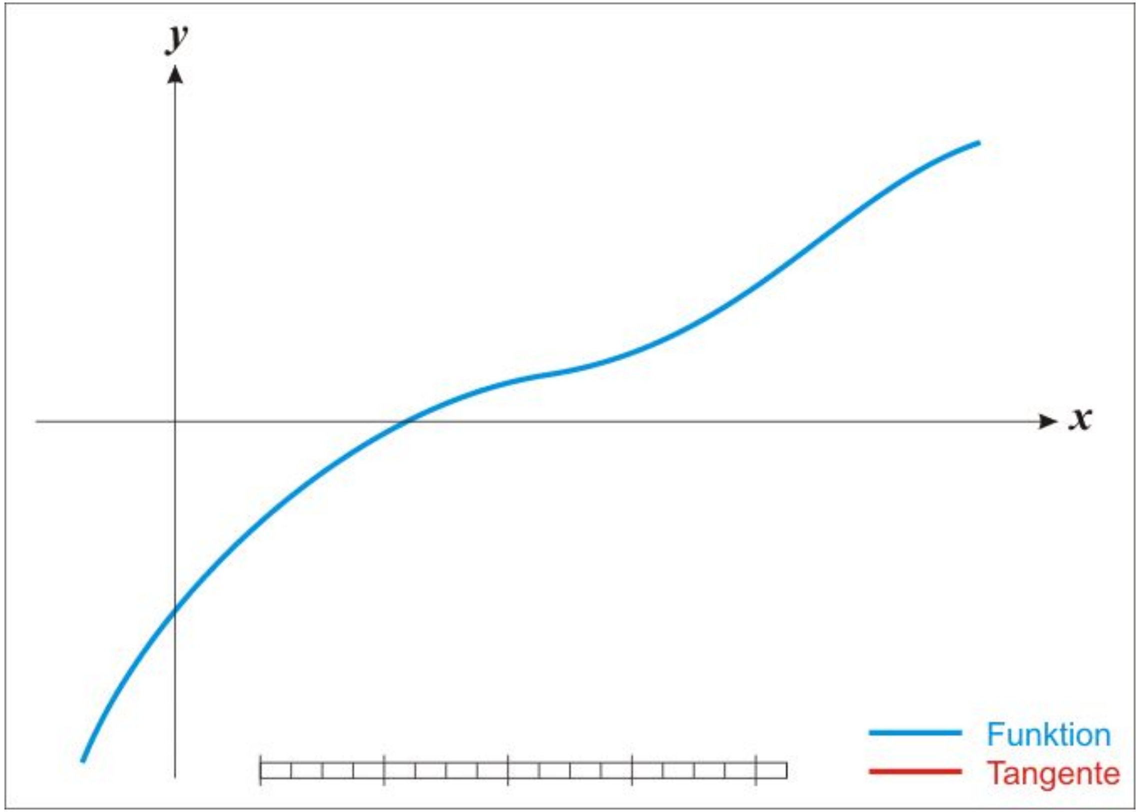
2. Solve linear equation

   $$f(x) + f'(x)\delta x = 0$$

   $$\Rightarrow \quad \delta x = -f(x)/f'(x)$$

3. Update x

   $$x \leftarrow x - f(x)/f'(x)$$



Multidimensional Newton's method: Real world is nonlinear!

vector out *vector in*

18.06: solving $\mathbf{f(x) = 0}$ where $x \in \mathbb{R}^n$ (input=vector) and f and $0 \in \mathbb{R}^n$ (output=vector)

Jacobian

1. Linearize:

   $$f(x+\delta x) \approx f(x) + \mathbf{f'(x)}\delta x$$

2. Solve linear equation

   $$f(x) + f'(x)\delta x = 0$$

   $$\Rightarrow \quad \delta x = -f'(x)^{-1} f(x)$$

   inverse Jacobian

3. Update x

   $$x \leftarrow x - f'(x)^{-1} f(x)$$

That's it! Once we have the Jacobian, just solve a linear system on each step.

Converges amazingly fast:
doubles #digits (squares error)
on each step ("quadratic convergence")!

Caveat: needs a starting guess
close enough to root
(google "Newton fractal"…)

Nonlinear optimization: min f(x), $x \in \mathbb{R}^n$

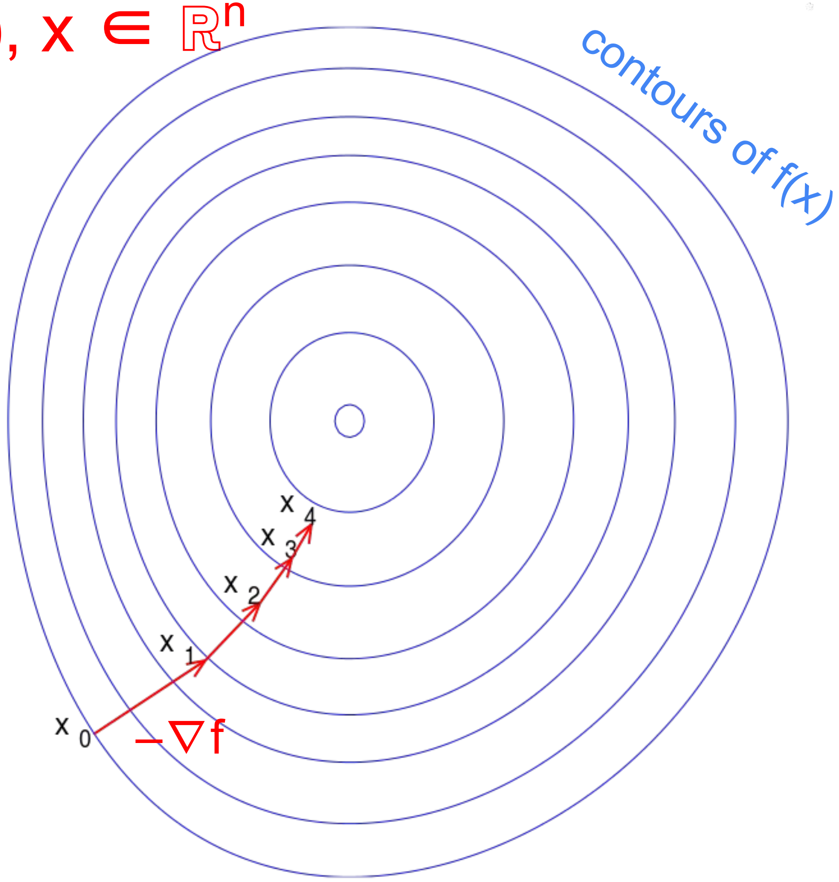
(or maximize)

$-\nabla f$ points downhill (steepest descent)

Even if we have $n=10^6$ parameters **x**, we can evolve them all simultaneously in the downhill direction.

Reverse-mode / adjoint / left-to-right / backpropagation: computing ∇f **costs** about same as evaluating f(x) once.

Makes large-scale optimization practical: training neural nets, optimizing shape of airplane wing, portfolio optimization…

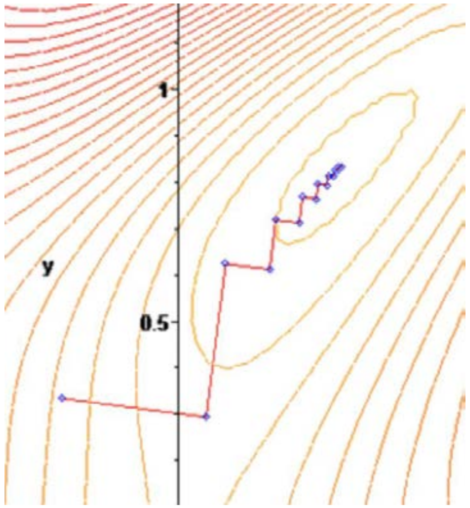


This image is in the public domain.

Nonlinear optimization: Lots of complications

- How far do we "step" in $-\nabla f$ direction?
  - Line search: $\min_\alpha f(x-\alpha\nabla f)$ — backtrack if not improved
  - *and/or* Limit step size to trust region, grow/shrink as needed
  - **Details are tricky** to get right
- Constraints: $\min f(x)$ subject to $g_k(x) \le 0$
  - Algorithms still need gradients ∇g_k!
- Faster convergence by "remembering" previous steps
  - Steepest-descent tends to "zig-zag" in narrow valleys
  - "Momentum" terms & conjugate gradients — simple "memory"
  - Fancier: estimate second derivative "Hessian matrix" from sequence of ∇f changes: BFGS algorithm
- Lots of refinements & competing algorithms …
  - **try out multiple** (pre-packaged) **algorithms** on your problem!

slow convergence:
zig-zagging downhill

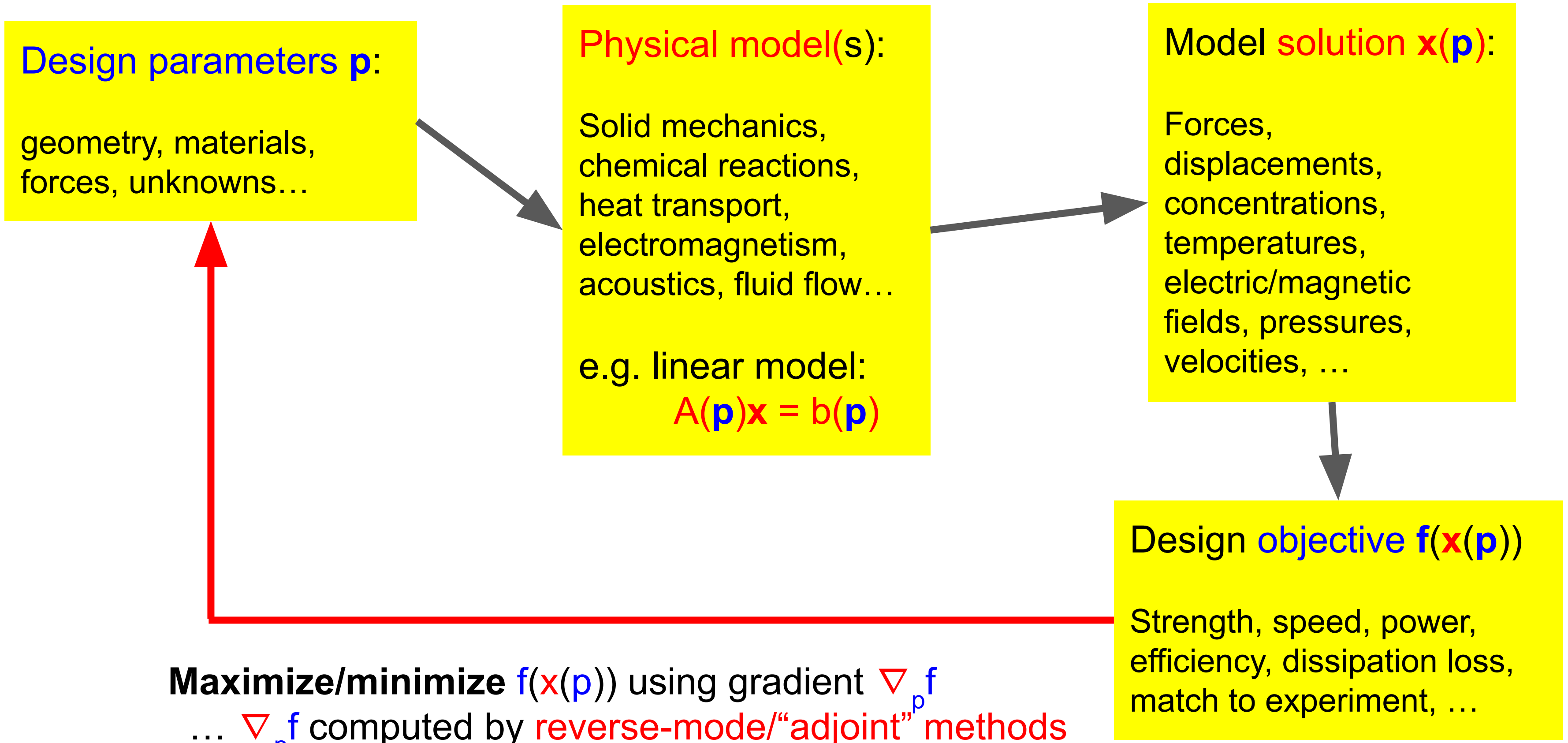


Some parting advice:

Often, the main trick is finding the right mathematical formulation of your problem — i.e. what function, what constraints, what parameters? — which lets you exploit the best algorithms.

…but if you have many (> 10) parameters,
always use an **analytical gradient** (not finite differences!)
… computed efficiently in **reverse mode**

Engineering/physical optimization


Design parameters **p**:
geometry, materials, forces, unknowns…
Physical model(s):
Solid mechanics, chemical reactions, heat transport, electromagnetism, acoustics, fluid flow…
e.g. linear model:
$A(\mathbf{p})\mathbf{x} = b(\mathbf{p})$
Model solution $\mathbf{x}(\mathbf{p})$:
Forces, displacements, concentrations, temperatures, electric/magnetic fields, pressures, velocities, …
Design objective $\mathbf{f}(\mathbf{x}(\mathbf{p}))$
Strength, speed, power, efficiency, dissipation loss, match to experiment, …

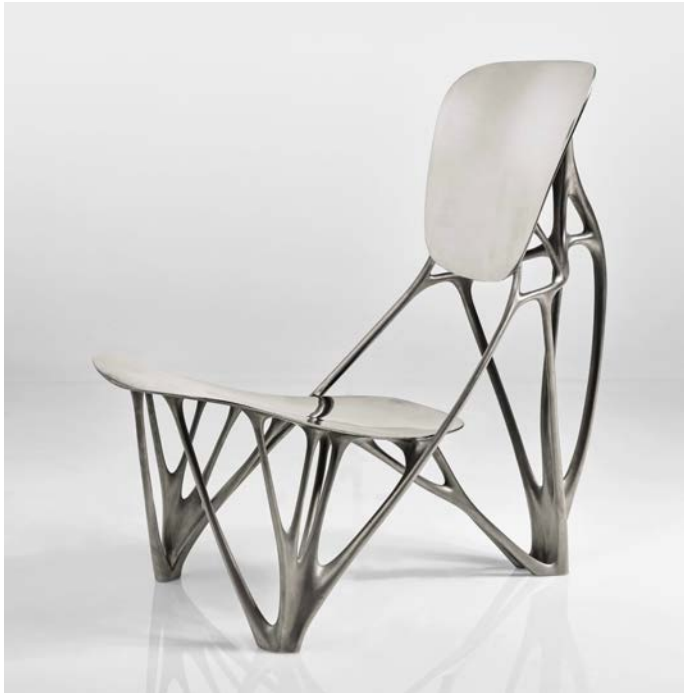
Maximize/minimize $f(x(p))$ using gradient $\nabla_p f$

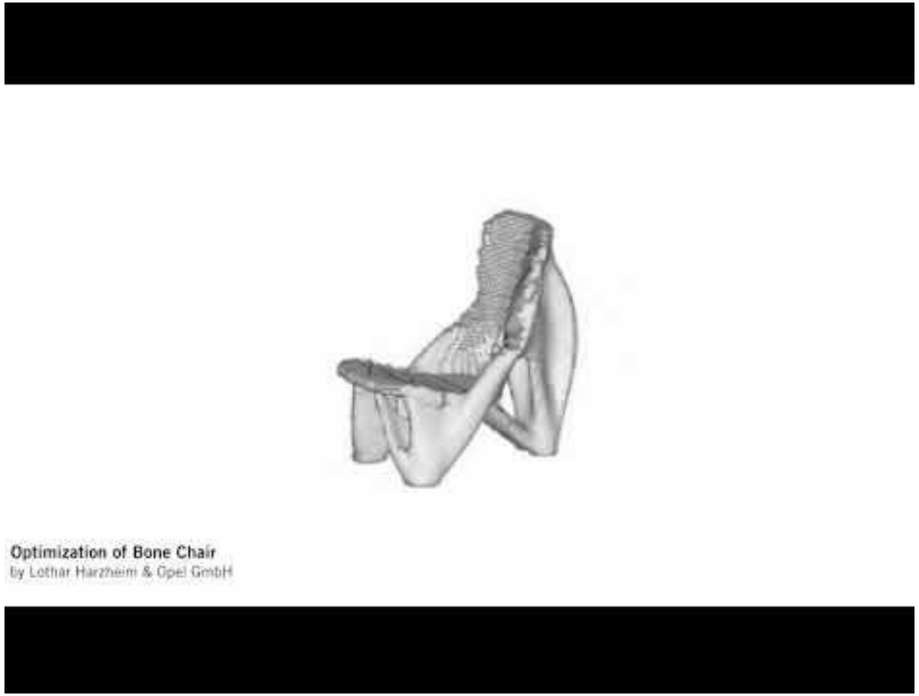
… $\nabla_p f$ computed by reverse-mode/“adjoint” methods

Example: "Topology optimization" of a chair

…optimizing every voxel to support weight with minimal material

(either voxel "density" or a "level-set" function)





Adjoint differentiation
(yet another example of left-to-right/reverse-mode differentiation)

Example: gradient of scalar $f(x(p))$ where $A(p)x=b$, i.e. $f(A(p)^{-1}b)$

- $df = f'(x)\, dx = f'(x)\, d(A^{-1})\, b = -\, f'(x)\, A^{-1}\, dA\, A^{-1}\, b$

  row vec matrix row vec = x

  = “adjoint” solution $\mathbf{v}^T$

- “Adjoint method:” Just multiply left-to-right! $df = -\,\mathbf{(}f'(x)\, A^{-1}\mathbf{)}\, dA\, x$
  - i.e. solve “adjoint equation” $A^T\mathbf{v} = f'(x)^T$ for v (“*adjoint*” meaning “*transpose*”)
  - …then $df = v^T\, dA\, x$
  - For any given parameter p□, ∂f/∂p□ = v^T ∂A/∂p□ x (& usually ∂A/∂p□ is very sparse)

- i.e. Takes only two solves to get both f and ∇f
  - Solve Ax=b once to get f(x), then solve *one* more time with A^T for v
  - … then *all* derivatives ∂f/∂p□ are just some cheap dot products

Don't use right-to-left **"forward-mode"** derivatives with **lots of parameters**!

$\partial f/\partial p\square = -\underbrace{f'(x)}_{\text{row vector}}\,(A^{-1}\,\underbrace{(\partial A/\partial p\square\ x)}_{=\ \text{vector}}) =$ one solve **per parameter** $p\square$!

solve

(different rhs)

Right-to-left (a.k.a. forward mode) better when **1 input** & many outputs.

Left-to-right (a.k.a. backward mode, adjoint, backpropagation) better when **1 output** & many inputs

(Note: Using dual numbers is forward mode. Most AD uses the term "forward" if it is forward mode. e.g. ForwardDiff.jl in Julia is forward mode. jax.jacfwd in Python is forward mode.)

Don’t use finite differences with **lots of parameters**!

$\partial f/\partial p_{\square} \approx [\ f(p + \varepsilon\ e_{\square}) - f(p)\ ] / \varepsilon$ (e$_{\square}$ = unit vector, ε = small number)

= requires one solve $x(p + \varepsilon\ e_{\square})$ for each parameter $p_{\square}$

… even worse if you use fancier finite-difference approximations

Adjoint differentiation with nonlinear equations

Example: gradient of scalar $f(x(p))$ where $x(p) \in \mathbb{R}^n$ solves $g(p,x) = 0 \in \mathbb{R}^n$

- $g(p,x) = 0 \Rightarrow dg = \partial g/\partial p \, dp + \partial g/\partial x \, dx = 0 \Rightarrow dx = -(\partial g/\partial x)^{-1} \, \partial g/\partial p \, dp$

  [a.k.a. “implicit-function theorem”]

  Jacobian, matrix

  = inverse Jacobian, also used in Newton solver for x!

- $df = f'(x) \, dx = - \left(f'(x) \, (\partial g/\partial x)^{-1} \right) \partial g/\partial p \, dp$

  = “adjoint” solution $\mathbf{v}^T$

  $\Rightarrow$ adjoint equation: $(\partial g/\partial x)^T \mathbf{v} = f'(x)^T$

- i.e. Takes only two solves to get both f and ∇f
  - one **nonlinear solve for x**, and one linear solve for v!
  - … then *all* derivatives ∂f/∂p□ are just some cheap dot products

You need to understand adjoint methods even if you use AD

- Helps understand **when to use** forward vs. reverse mode!
- Many physical models call large software packages written over decades in various languages, and **cannot be differentiated automatically** by AD
    - You often just need to supply a “vector–Jacobian product” $y^T dx$ for physics, or even just *part* of the physics, and then AD will differentiate the rest and apply the chain rule for you
- Often models involve **approximate calculations**, but AD tools don’t know this & spend extra effort trying to differentiate the *error* in your approximation
    - If you solve for x by an iterative method (e.g. Newton), it is inefficient for AD to backpropagate *through* the iteration … instead, you want take derivative of the underlying equation $g(p,x) = 0$
    - For discretized physics (e.g. a finite-element methods), it is often more efficient (and sufficiently accurate) to apply adjoint method to continuous physics (“differentiate-then-discretize”)