Implicit differentiation and differentiable optimization

UCSD CSE 291
Tzu-Mao Li

Today

$$x^* = \arg\min_x f(x, \theta)$$

we want to differentiate this

Motivation

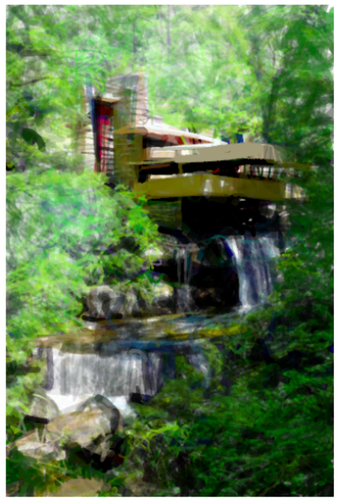
"Breathing Life Into Sketches Using Text-to-Video Priors", Gai*, Vinker*, Alaluf, Bermano, Cohen-Or, Shamir, Chechik https://livesketch.github.io/

The technology behind

Differentiable Vector Graphics Rasterization for Editing and Learning

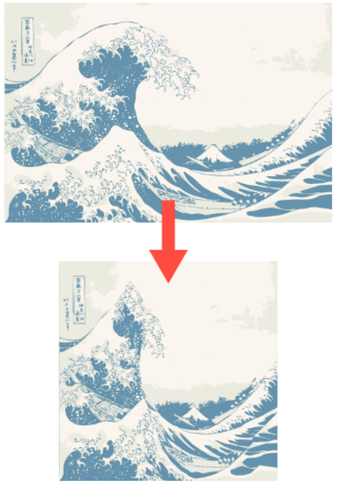
TZU-MAO LI, MIT CSAIL
MICHAL LUKÁČ, Adobe Research
MICHAËL GHARBI, Adobe Research
JONATHAN RAGAN-KELLEY, MIT CSAIL

(a) vector sculpting



(b) painterly rendering

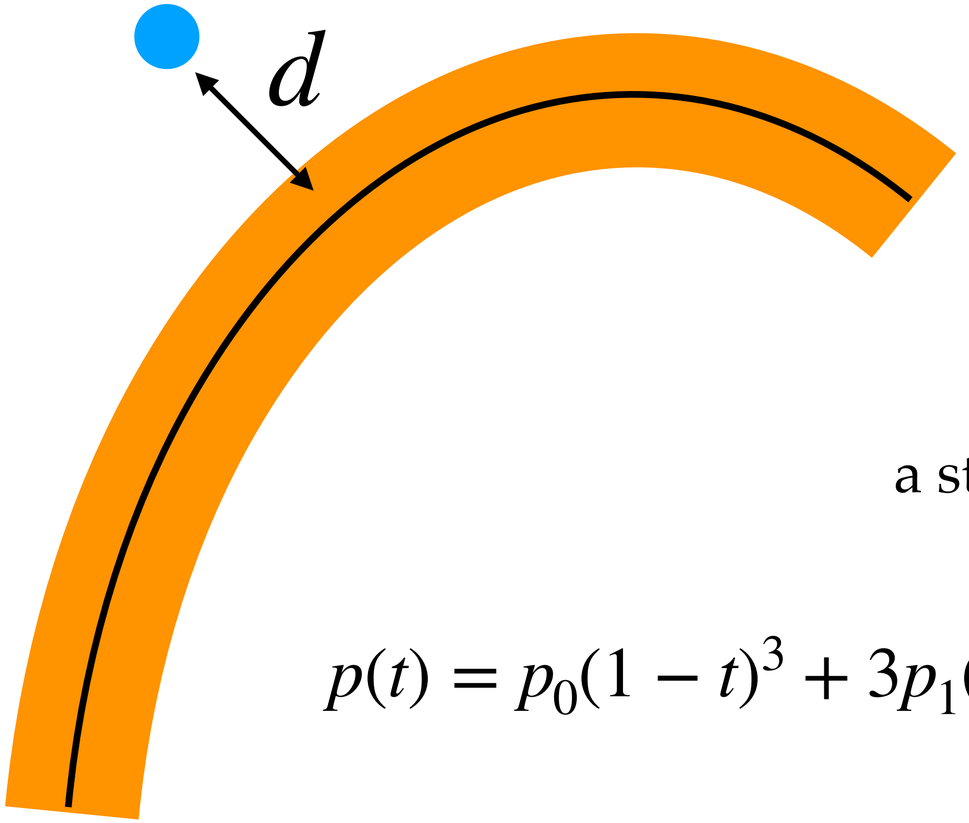
(c) image vectorization



(d) seam carving

(e) generative modeling

Motivation: differentiating point to stroke distance



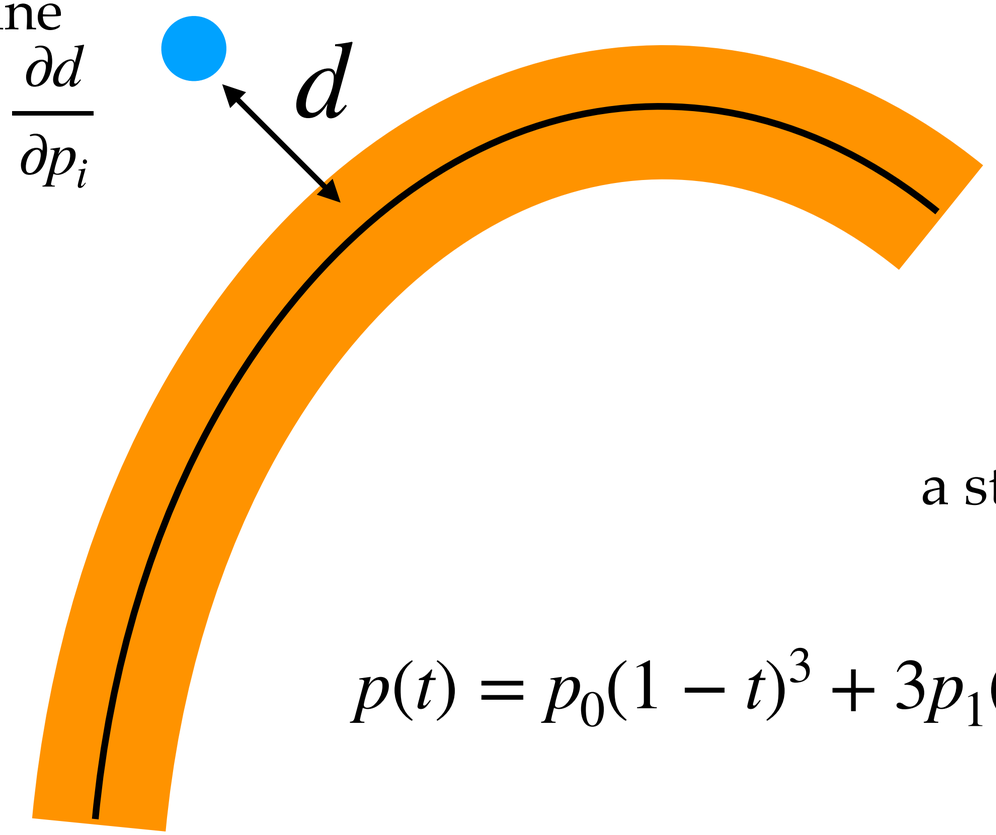
a stroke is usually represented by a cubic polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

Motivation: differentiating point to stroke distance

at one stage of the pipeline

we may want to compute $\frac{\partial d}{\partial p_i}$



a stroke is usually represented by a cubic polynomial

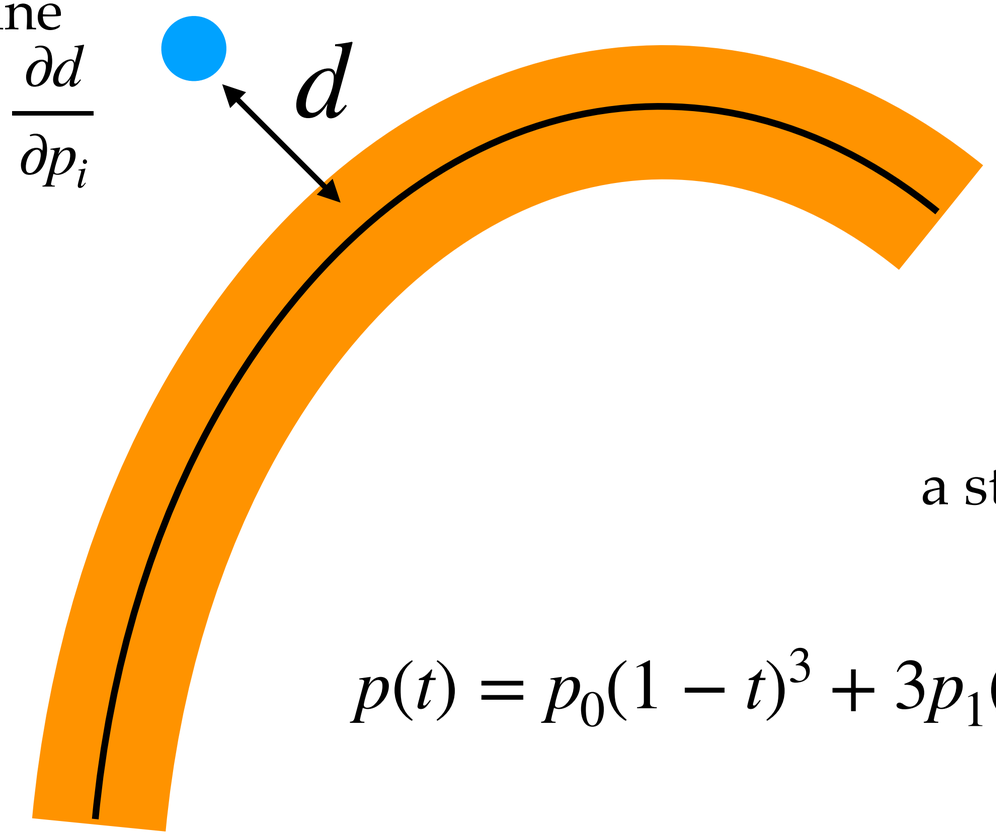
$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

Motivation: differentiating point to stroke distance

at one stage of the pipeline

we may want to compute $\frac{\partial d}{\partial p_i}$

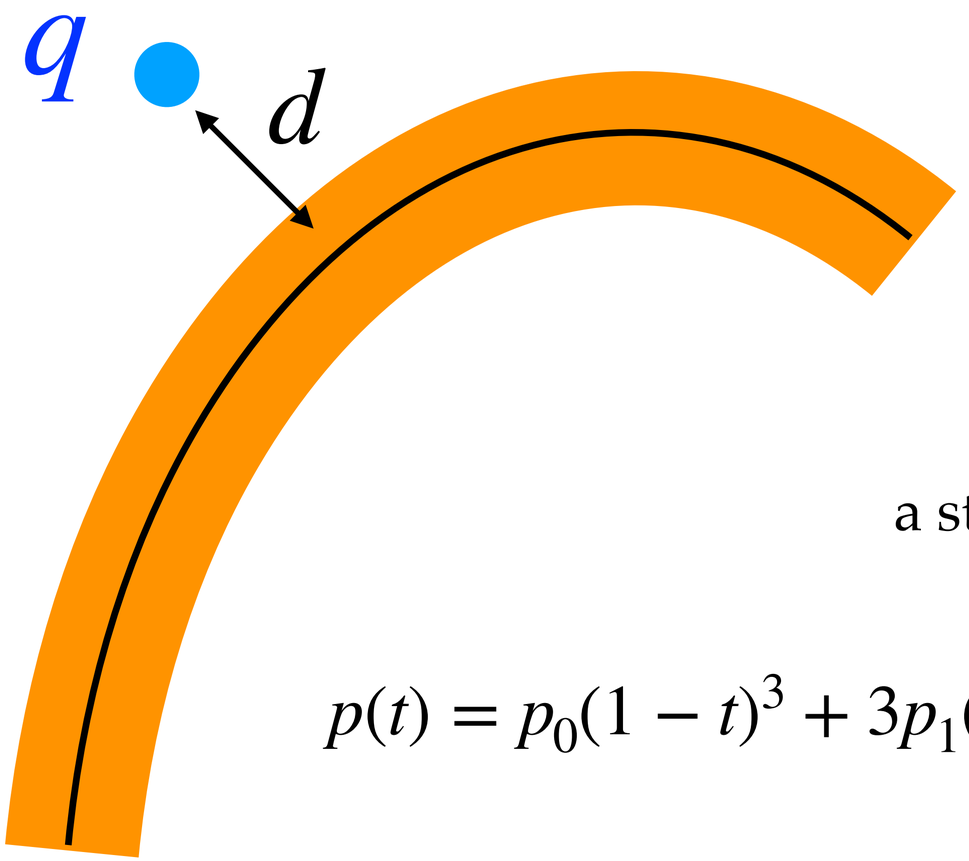
how do we compute d
in the first place?



a stroke is usually represented by
a cubic polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

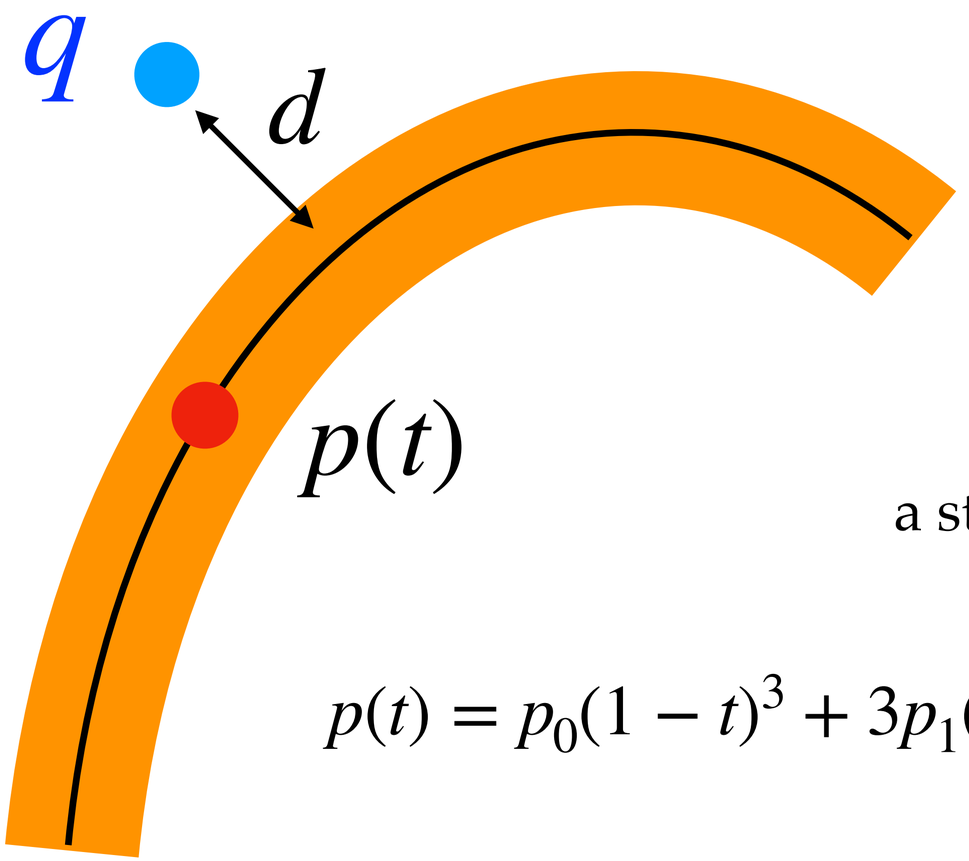
Motivation: differentiating point to stroke distance



a stroke is usually represented by a cubic polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

Motivation: differentiating point to stroke distance



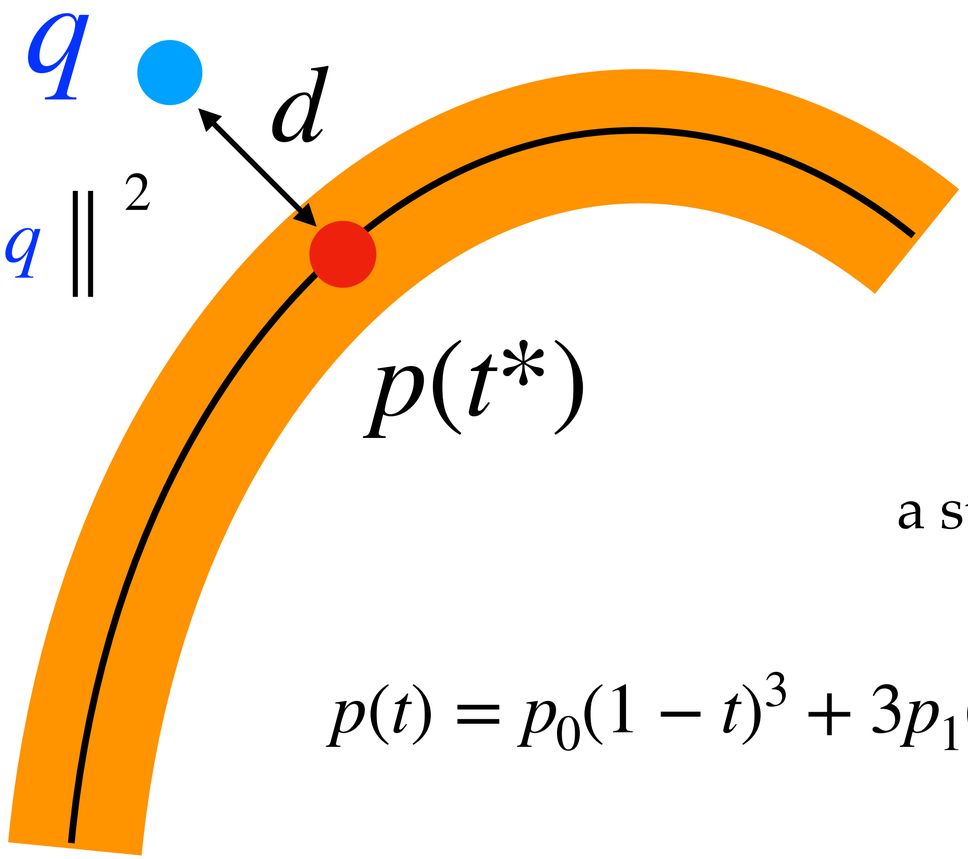
a stroke is usually represented by a cubic polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

Motivation: differentiating point to stroke distance

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2$$

$$d = \left\| q - p(t^*) \right\|$$



a stroke is usually represented by a cubic polynomial

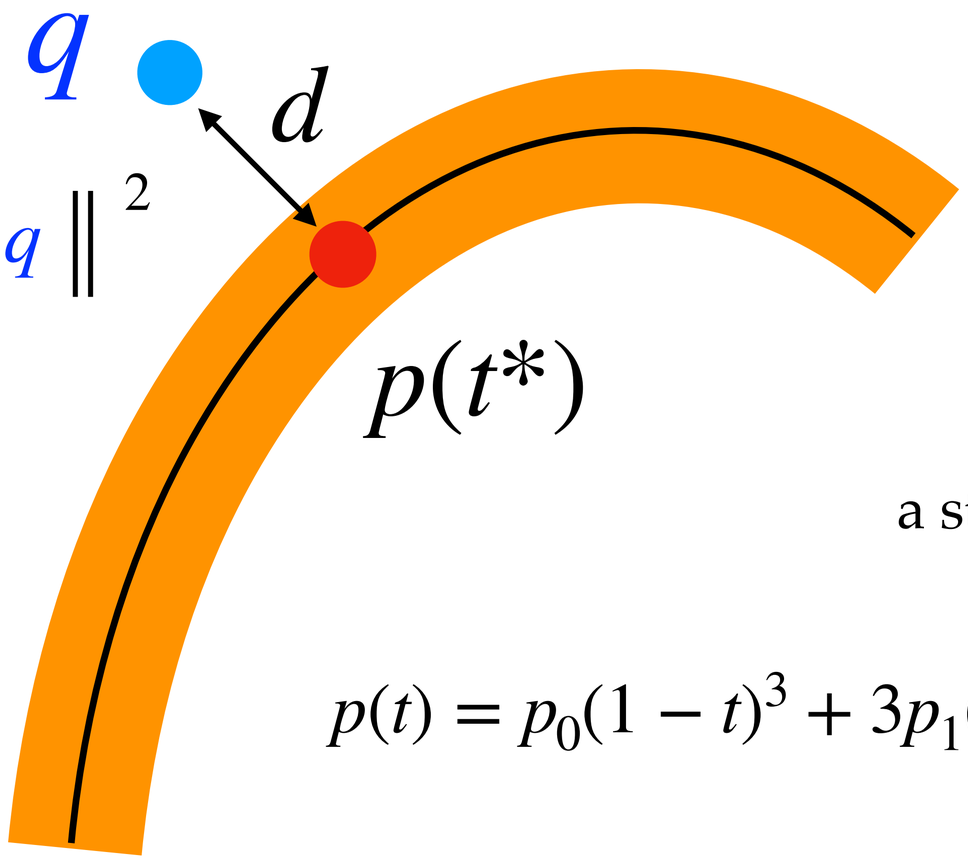
$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Motivation: differentiating point to stroke distance

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2$$

$$d = \left\| q - p(t^*) \right\|$$

$$\frac{\partial d}{\partial p_i} = \frac{\partial d}{\partial t^*}\frac{\partial t^*}{\partial p_i}$$



a stroke is usually represented by a cubic polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Solving a polynomial root

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2$$

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Solving a polynomial root

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2 \longrightarrow \frac{d}{dt} \left\| p(t) - q \right\|^2 = 0$$

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Solving a polynomial root

$$t^* = \arg\min_t \| p(t) - q \|^2 \longrightarrow \frac{d}{dt} \| p(t) - q \|^2 = 0$$

p is a 3rd order polynomial

$\| p - q \|^2$ is a 6th order polynomial

$\frac{d}{dt} \| p - q \|^2$ is a 5th order polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Solving a polynomial root

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2 \longrightarrow \frac{d}{dt} \left\| p(t) - q \right\|^2 = 0$$

p is a 3rd order polynomial

$\left\| p - q \right\|^2$ is a 6th order polynomial

$\frac{d}{dt} \left\| p - q \right\|^2$ is a 5th order polynomial

need to solve for the roots
of a 5th order polynomial!

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

No close form solution for 5th-order polynomials

Abel–Ruffini theorem

22 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with Abel's theorem.

In mathematics, the **Abel–Ruffini theorem** (also known as **Abel's impossibility theorem**) states that there is no solution in radicals to general polynomial equations of degree five or higher with arbitrary coefficients. Here, *general* means that the coefficients of the equation are viewed and manipulated as indeterminates.

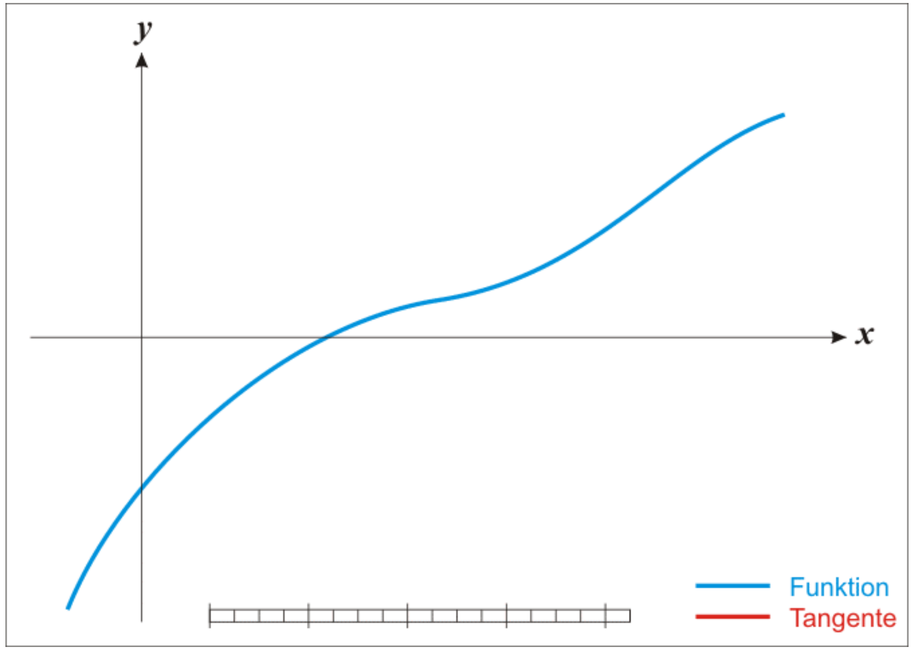
$$dt$$

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

Solving for roots of a 5th order polynomial

have to use an iterative method: start from an initial guess and refine

$$t_{n+1} = t_n - \frac{f(t_n, p_i)}{\partial_t f(t_n, p_i)}$$



https://en.wikipedia.org/wiki/Newton%27s_method

Solving for roots of a 5th order polynomial

To solve for $t^\star$, we take the derivative of the squared distance with respect to t and set it to zero:

$$2\,(p(t) - q) \cdot p'(t) = \rho(t) = 0. \tag{12}$$

This is a 5th order polynomial (p is 3rd order and p' is 2nd order) and does not have a closed-form solution. To solve for all real roots of the 5th order polynomial, we use a standard hybrid Newton-Raphson bisection method [Press et al. 2007]. This requires us to identify the intervals of all the real roots. To achieve this, we use the technique of *isolator polynomial* [Sederberg and Chang 1994]. Given a polynomial $\rho(t)$ (in our case a 5th order polynomial) and its derivative $\rho'(t)$, the long division of them yields two lower-order polynomials $a(t)$ and $b(t)$:

$$a(t) = \rho(t) - b(t)\rho'(t). \tag{13}$$

It turns out the real roots of a and b *isolate* the roots of ρ. To see this, consider any two adjacent real roots of ρ, r_0 and r_1 (if ρ only has one or zero real root, we know t is bounded by $[0, 1]$). We can show that there must be a root of either a or b in $[r_0, r_1]$. Since $\rho(r_0) = \rho(r_1) = 0$, we know $a(r_0)a(r_1) = b(r_0)b(r_1)\rho'(r_0)\rho'(r_1)$. Since r_0 and r_1 are roots of ρ, $\rho'(r_0)\rho'(r_1) \le 0$. Thus either $a(r_0)a(r_1) \le 0$ or $b(r_0)b(r_1) \le 0$. See Sederberg and Chang's paper for remarks on multiple roots.

We can construct a 5th order polynomial $\rho(t) = t^5 + Bt^4 + Ct^3 + Dt^2 + Et + F$ by rearranging Equation 12. By doing a long division between ρ and ρ', we obtain

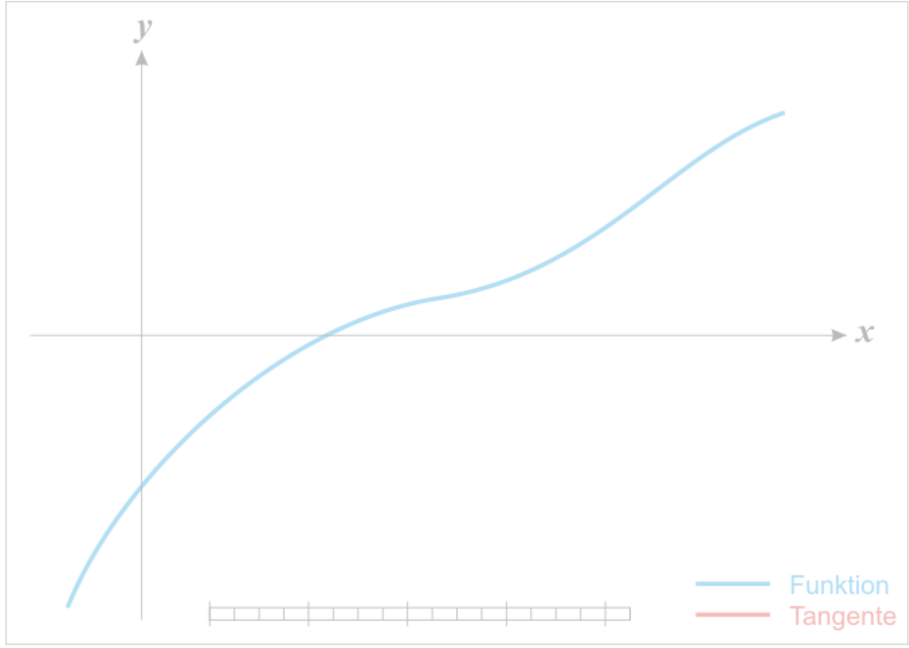
$$a(t) = \left(\frac{2C}{5} - \frac{4B^2}{25}\right)t^3 + \left(\frac{3D}{5} - \frac{3BC}{25}\right)t^2 + \left(\frac{4E}{5} - \frac{2BD}{25}\right)t + F - \frac{BE}{25}$$
$$b(t) = \frac{t}{5} + \frac{B}{25}. \tag{14}$$

Since a is a cubic polynomial and b is linear, we can solve for all the real roots of a and b between $(0, 1)$. This forms at most 5 intervals,

erative method: start from an initial guess and refine

$$\frac{f(t_n, p_i)}{f(t_n, p_i)}$$

see my paper on
how to construct an initial guess
using "Isolator Polynomials"



https://en.wikipedia.org/wiki/Newton%27s_method

We can encapsulate the argmin using an implicit function

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2 \longrightarrow \frac{d}{dt} \left\| p(t) - q \right\|^2 = 0$$

$$\downarrow$$

$$f(t^*(p_i), p_i) = 0 \longleftarrow t_{n+1} = t_n - \frac{f(t_n, p_i)}{\partial_t f(t_n, p_i)}$$

We can encapsulate the argmin using an implicit function

remember, we wanted to take derivative of t^* w.r.t. p_i

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2 \longrightarrow \frac{d}{dt} \left\| p(t) - q \right\|^2 = 0$$

$$\downarrow$$

$$f(t^*(p_i), p_i) = 0 \longleftarrow t_{n+1} = t_n - \frac{f(t_n, p_i)}{\partial_t f(t_n, p_i)}$$

The naive way: differentiating Newton iterations

start with an initial guess t_0

$$t_1 = t_0 - \frac{f(t_0, p_i)}{\partial_t f(t_0, p_i)}$$

$$t_2 = t_1 - \frac{f(t_1, p_i)}{\partial_t f(t_1, p_i)}$$

...

$$t_n = t_{n-1} - \frac{f(t_{n-1}, p_i)}{\partial_t f(t_{n-1}, p_i)}$$

The naive way: differentiating Newton iterations

start with an initial guess t_0

$$t_1 = t_0 + G(t_0, p_i)$$

$$t_2 = t_1 + G(t_1, p_i)$$

...

$$t_n = t_{n-1} + G(t_{n-1}, p_i)$$

The naive way: differentiating Newton iterations

start with an initial guess t_0

$$t_1 = t_0 + G(t_0, p_i)$$

$$t_2 = t_1 + G(t_1, p_i)$$

…

$$t_n = t_{n-1} + G(t_{n-1}, p_i)$$

reverse mode

```
dt[n] = …
dt[n-1] = dt[n]
DTG(t[n-1], &dt[n-1],
    p[i], &dp[i],
    dt[n])

dt[n-2] = dt[n-1]
DTG(t[n-2], &dt[n-2],
    p[i], &dp[i],
    dt[n-1])

…
```

The naive way: differentiating Newton iterations

start with an initial guess t_0

$$t_1 = t_0 + G(t_0, p_i)$$

$$t_2 = t_1 + G(t_1, p_i)$$

…

$$t_n = t_{n-1} + G(t_{n-1}, p_i)$$

reverse mode

needs to allocate size n arrays!

```
dt[n] = …
dt[n-1] = dt[n]
DTG(t[n-1], &dt[n-1],
    p[i], &dp[i],
    dt[n])

dt[n-2] = dt[n-1]
DTG(t[n-2], &dt[n-2],
    p[i], &dp[i],
    dt[n-1])

…
```

A better way to differentiate argmin: use the implicit function form!

remember, we wanted to take derivative of t^* w.r.t. p_i

$$t^* = \arg\min_t \| p(t) - q \|^2 \longrightarrow \frac{d}{dt} \| p(t) - q \|^2 = 0$$

$$\downarrow$$

$$f(t^*(p_i), p_i) = 0$$

$$\longleftarrow$$

$$t_{n+1} = t_n - \frac{f(t_n, p_i)}{\partial_t f(t_n, p_i)}$$

Assumption: the Newton iteration reaches convergence

$$t_1 = t_0 + G(t_0, p_i)$$

$$t_2 = t_1 + G(t_1, p_i)$$

$$f(t^*(p_i), p_i) = 0$$

...

$$t_n = t_{n-1} + G(t_{n-1}, p_i)$$

$$t_n = t^*$$

Throw away all Newton iteration history

$$f(t^*(p_i), p_i) = 0$$

Math: implicit function theorem!

$$\frac{\partial}{\partial p_i} f(t^*(p_i), p_i) = 0$$

$$f(t^*(p_i), p_i) = 0$$

Math: implicit function theorem!

$$f(t^*(p_i), p_i) = 0$$

$$\frac{\partial}{\partial p_i} f(t^*(p_i), p_i) = 0$$

$$\partial_{t^*} f \partial_{p_i} t^* + \partial_{p_i} f = 0$$

Math: implicit function theorem!

$$f(t^*(p_i), p_i) = 0$$

$$\frac{\partial}{\partial p_i} f(t^*(p_i), p_i) = 0$$

$$\partial_{t^*} f \partial_{p_i} t^* + \partial_{p_i} f = 0$$

$$\boxed{\frac{\partial t^*}{\partial p_i} = -\frac{\partial_{p_i} f}{\partial_{t^*} f}}$$

Math: implicit function theorem!

$$f(t^*(p_i), p_i) = 0$$

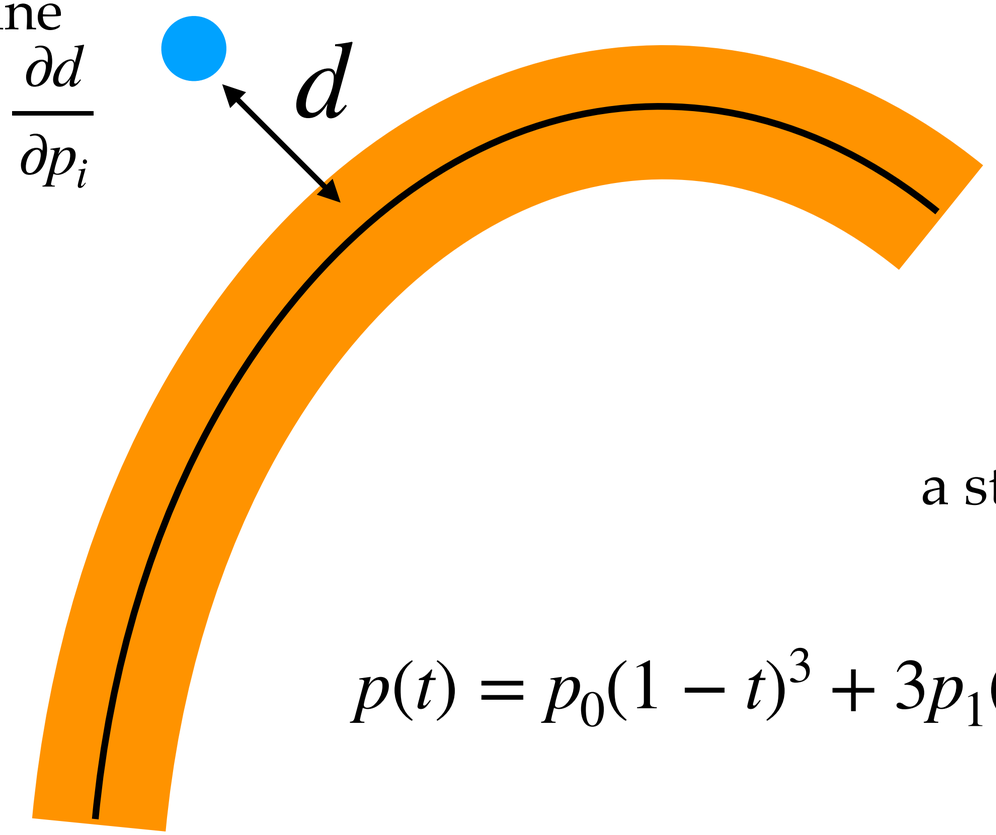
only need the information at t^*!!

$$\frac{\partial t^*}{\partial p_i} = -\frac{\partial_{p_i} f}{\partial_{t^*} f}$$

Recap

at one stage of the pipeline

we may want to compute $\frac{\partial d}{\partial p_i}$



a stroke is usually represented by
a cubic polynomial

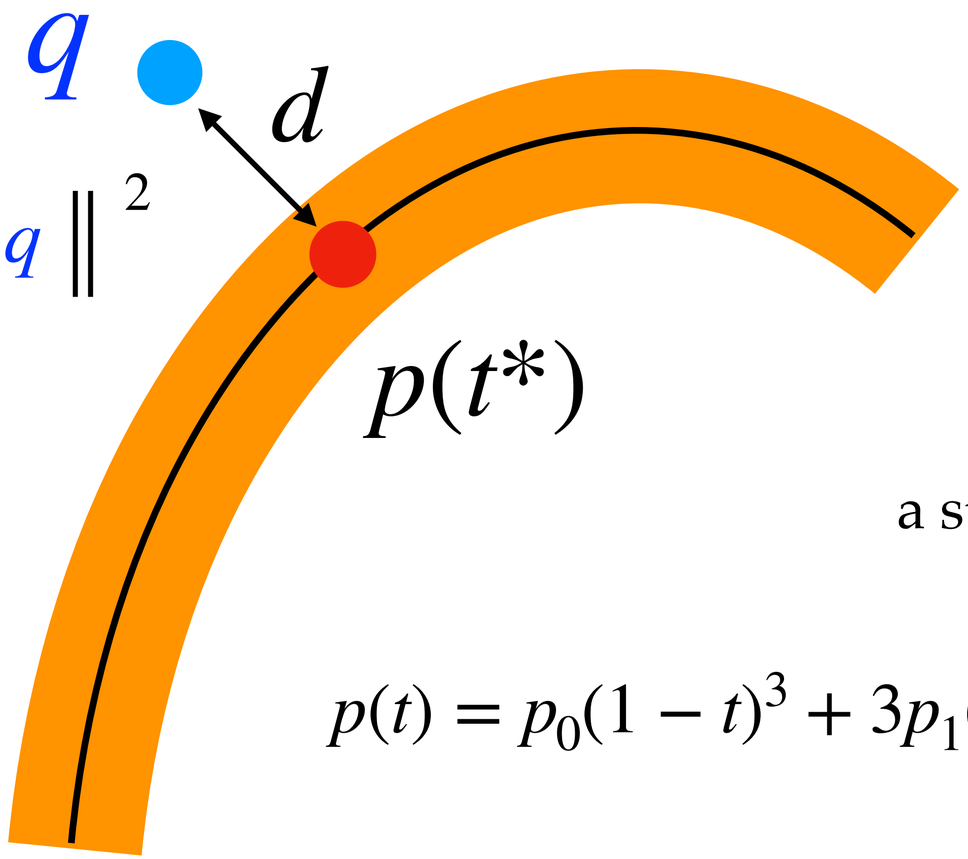
$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2t + 3p_2(1-t)t^2 + p_3t^3$$

Recap

$$t^* = \arg\min_t \left\| p(t) - q \right\|^2$$

$$d = \left\| q - p(t^*) \right\|$$

$$\frac{\partial d}{\partial p_i} = \frac{\partial d}{\partial t^*} \frac{\partial t^*}{\partial p_i}$$



a stroke is usually represented by
a cubic polynomial

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Recap

$$t^* = \arg\min_t \| p(t) - q \|^2 \longrightarrow \frac{d}{dt} \| p(t) - q \|^2 = 0$$

p is a 3rd order polynomial

$\| p - q \|^2$ is a 6th order polynomial

$\frac{d}{dt} \| p - q \|^2$ is a 5th order polynomial

need to solve for the roots
of a 5th order polynomial!

$$p(t) = p_0(1-t)^3 + 3p_1(1-t)^2 t + 3p_2(1-t)t^2 + p_3 t^3$$

Recap

remember, we wanted to take derivative of t^* w.r.t. p_i

$$t^* = \arg\min_t \| p(t) - q \|^2 \longrightarrow \frac{d}{dt} \| p(t) - q \|^2 = 0$$

$$\downarrow$$

$$t_{n+1} = t_n - \frac{f(t_n, p_i)}{\partial_t f(t_n, p_i)} \longleftarrow f(t^*(p_i), p_i) = 0$$

Recap

remember, we wanted to take derivative of t^* w.r.t. p_i

$$t^* = \arg\min_t \| p(t) - q \|^2 \longrightarrow \frac{d}{dt} \| p(t) - q \|^2 = 0$$

$$f(t^*(p_i), p_i) = 0$$

$$t_{n+1} = t_n - \frac{f(t_n, p_i)}{\partial_t f(t_n, p_i)}$$

assuming the Newton iteration converged, then $t_{n+1} = t^*$

$$\frac{\partial t^*}{\partial p_i} = -\frac{\partial_{p_i} f}{\partial_{t^*} f}$$

Take home message

- for any argmin/root finding problem that requires an iterative solver, assuming the solver converges, we can differentiate through the argmin/root finding without remembering any of the solver states!!

$$f(t^*(p_i), p_i) = 0$$

$$\frac{\partial t^*}{\partial p_i} = -\frac{\partial_{p_i} f}{\partial_{t^*} f}$$

More applications

Differentiable Solver for Time-Dependent Deformation Problems With Contact

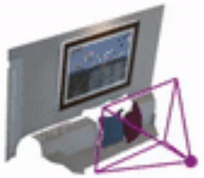
in simulation, finding a new state is
often formulated as solving an implicit function

implicit differentiation
allows us to very efficiently
differentiate simulation

Differentiable solver for time-dependent deformation problems with contact

ZIZHOU HUANG*, New York University, USA
DAVI COLLI TOZONI*, NTop and New York University, USA
ARVI GJOKA, New York University, USA
ZACHARY FERGUSON, Massachusetts Institute of Technology and New York University, USA
TESEO SCHNEIDER, University of Victoria, Canada
DANIELE PANOZZO, New York University, USA
DENIS ZORIN, New York University, USA

More applications



differentiate through 3D reconstruction
for training better feature matching

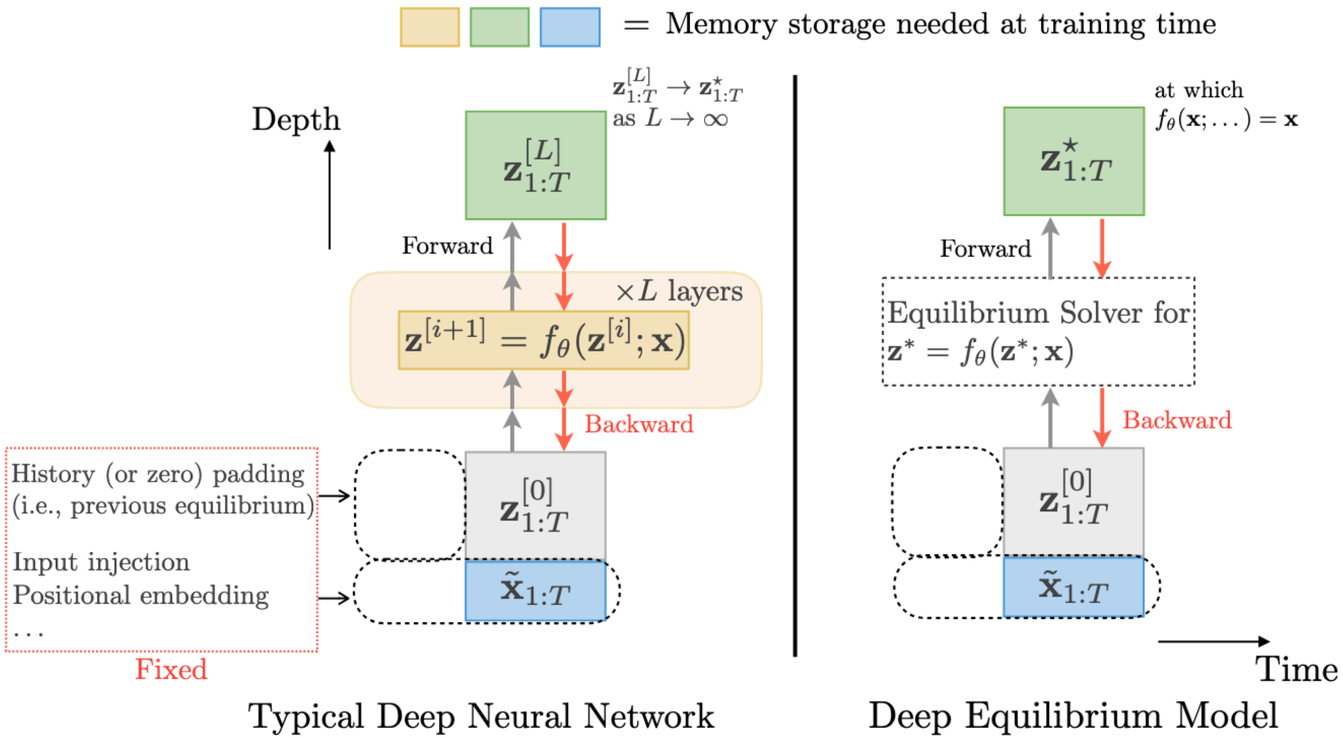
∇SLAM: Automagically differentiable SLAM

https://gradslam.github.io

Krishna Murthy J.*[1,2,3], Soroush Saryazdi*[4], Ganesh Iyer[5], and Liam Paull†[1,2,3,6]

[1]Université de Montréal, [2]Mila, [3]Robotics and Embodied AI Lab (REAL), [4]Concordia University,
[5]Carnegie Mellon University, [6]Candian CIFAR AI Chair

More applications



a deep learning model where
each layer is a root finding process
saves lots of memory during backprop!

Deep Equilibrium Models

Shaojie Bai
Carnegie Mellon University

J. Zico Kolter
Carnegie Mellon University
Bosch Center for AI

Vladlen Koltun
Intel Labs

Multidimensional case

$$F(x^*(\theta), \theta) = 0$$

$$x^* \in \mathbb{R}^d$$

$$\theta \in \mathbb{R}^n$$

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

Multidimensional case

$$F(x^*(\theta), \theta) = 0$$

$$x^* \in \mathbb{R}^d$$

$$\theta \in \mathbb{R}^n$$

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

$$\underset{\mathbb{R}^{d\times d}}{\partial_{x^*} F} \underset{\mathbb{R}^{d\times n}}{\partial_\theta x^*} = - \partial_\theta F$$

$$\mathbb{R}^{d\times n}$$

Multidimensional case

$$F(x^*(\theta), \theta) = 0$$

$$x^* \in \mathbb{R}^d$$

$$\theta \in \mathbb{R}^n$$

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

$$\underset{\mathbb{R}^{d\times d}}{\left[\partial_{x^*} F\right]^T} \underset{\mathbb{R}^{d\times 1}}{u} = - \underset{\mathbb{R}^{d\times 1}}{dx^*(\theta)}$$

$$d\theta = u^T \partial_\theta F$$

$dx^*(\theta)$ is the adjoint of $x^*(\theta)$
$d\theta$ is the adjoint of θ

Multidimensional case

$$F(x^*(\theta), \theta) = 0$$

$$x^* \in \mathbb{R}^d$$

$$\theta \in \mathbb{R}^n$$

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

still need to solve a linear system!
usually can be solved in the same way as solving F=0

$$\underset{\mathbb{R}^{d\times d}}{\left[\partial_{x^*} F\right]^T} \underset{\mathbb{R}^{d\times 1}}{u} = \underset{\mathbb{R}^{d\times 1}}{-\, dx^*(\theta)}$$

$$d\theta = u^T \partial_\theta F$$

$dx^*(\theta)$ is the adjoint of $x^*(\theta)$
$d\theta$ is the adjoint of θ

Implicit differentiation is more numerically accurate than unrolling

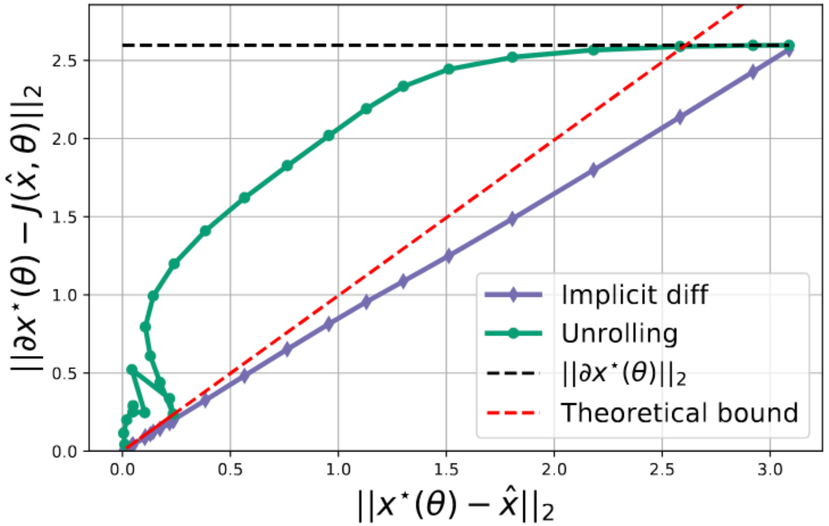


Figure 3: Jacobian estimate errors. Empirical error of implicit differentiation follows closely the theoretical upper bound. Unrolling achieves a much worse error for comparable iterate error.

assuming that the derivative you want is the derivative at the true minimum

Efficient and Modular Implicit Differentiation

Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, Jean-Philippe Vert
Google Research

Integrating implicit differentiation into AD systems

https://jaxopt.github.io/stable/

Efficient and Modular Implicit Differentiation

Mathieu Blondel, Quentin Berthet, Marco Cuturi*, Roy Frostig,
Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, Jean-Philippe Vert*
Google Research