CS 61A/CS 98-52

Mehrdad Niknami

University of California, Berkeley

Warning

FYI: This lecture might get a little... intense... and math-y

If it's hard, **don't panic!** It's okpy! They won't all be like this!

Just try to enjoy it, ask questions, & learn as much as you can. :)

Ready?!

Preliminaries

Last lecture was on **equation-solving**:

- "Given f and initial guess x_0, solve $f(x) = 0$"

This lecture is on **optimization**: $\arg\min_x F(x)$

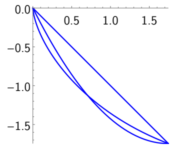
- "Given F and initial guess x_0, find x that minimizes $F(x)$"

Brachistochrone Problem

Let's solve a realistic problem.

It's the *brachistochrone* ("shortest time") problem:

1. Drop a ball on a ramp
2. Let it roll down
3. What shape minimizes the travel time?



$\implies$ How would **you** solve this?

Brachistochrone Problem

Ideally: Learn fancy math, derive the answer, plug in the formula.

Oh, sorry... did you say you're a *programmer*?

1. Math is hard
2. Physics is hard
3. We're lazy
4. Why learn something new when you can burn electricity instead?

OK but honestly the math **is** a little complicated...

- Calculus of variations... Euler-Lagrange differential eqn... maybe?
- Take Physics 105... have fun!
- Don't get wrecked

Brachistochrone Problem

Joking aside...

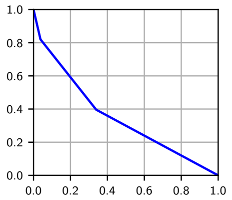
Most problems don't have a nice formula, so you'll need algorithms.

Let's get our hands dirty!
Remember Riemann sums?

This is similar:

1. Chop up the ramp into line segments (but hold ends fixed)
2. Move around the anchors to **minimize travel time**

Q: How do you do this?



Algorithm

Use Newton-Raphson!

...but wasn't that for finding *roots*? Not optimizing?

Actually, it's used for both:

- If F is differentiable, minimizing F **reduces to root-finding**:

$$F'(x) = f(x) = 0$$

- Caveat: must avoid maxima and inflection points
  - Easy in 1-D: only ± directions to check for increase/decrease
  - Good luck in N-D... infinitely many directions

Algorithm

Newton-Raphson method for **optimization**:

1. Assume F is **approximately quadratic**[1] (so $f = F'$ approx. linear)
2. Guess some x_0 intelligently
3. Repeatedly solve linear approximation[2] of $f = F'$:

$$\begin{aligned} f(x_k) - f(x_{k+1}) &= f'(x_k)\,(x_k - x_{k+1}) \\ f(x_{k+1}) &= 0 \\ \implies \quad x_{k+1} &= x_k - f'(x_k)^{-1}\, f(x_k) \end{aligned}$$

   We ignored F! Avoid maxima and inflection points! (How?)

4. ...Profit?

---

[1] Why are quadratics common? Energy/cost are quadratic ($K = \frac{1}{2}mv^2$, $P = I^2R$...)

[2] You'll see linearization **ALL** the time in engineering

Algorithm

Wait, but we have a function of **many** variables. What do?

A couple options:

1. Fully multivariate Newton-Raphson:

$$\vec{x}_{k+1} = \vec{x}_k - \vec{\nabla}\vec{f}(\vec{x}_k)^{-1}\,\vec{f}(\vec{x}_k)$$

   Taught in EE 219A, 227C, 144/244, etc... (need Math 53 and 54)

2. Newton coordinate-descent

Algorithm

Coordinate descent:

1. Take x_1, use it to minimize F, holding others fixed
2. Take y_1, use it to minimize F, holding others fixed
3. Take x_2, use it to minimize F, holding others fixed
4. Take y_2, use it to minimize F, holding others fixed
5. ...
6. Cycle through again

Doesn't work as often, but it works very well here.

Algorithm

Newton step for **minimization**:

```
def newton_minimizer_step(F, coords, h):
    delta = 0.0
    for i in range(1, len(coords) - 1):
        for j in range(len(coords[i])):
            def  f(c): return derivative(F, c, i, j, h)
            def df(c): return derivative(f, c, i, j, h)
            step = -f(coords) / df(coords)
            delta += abs(step)
            coords[i][j] += step
    return delta
```

Side note: Notice a potential bug? What's the fix?
Notice a 33% inefficiency? What's the fix?

Algorithm

Computing derivatives numerically:

```
def derivative(f, coords, i, j, h):
    x = coords[i][j]
    coords[i][j] = x + h;  f2 = f(coords)
    coords[i][j] = x - h;  f1 = f(coords)
    coords[i][j] = x
    return (f2 - f1) / (2 * h)
```

Why not `(f(x + h) - f(x)) / h`?

- Breaking the intrinsic asymmetry reduces accuracy

~ Words of Wisdom ~

If your problem has {`fundamental feature`} that your solution doesn't, you've created more problems.

Algorithm

What is our **objective function** F to minimize?

```
def falling_time(coords): # coords = [[x1,y1], [x2,y2], ...]
    t, speed = 0.0, 0.0
    prev = None
    for coord in coords:
        if prev != None:
            dy = coord[1] - prev[1]
            d = ((coord[0] - prev[0]) ** 2 + dy ** 2) ** 0.5
            accel = -9.80665 * dy / d
            for dt in quadratic_roots(accel, speed, -d):
                if dt > 0:
                    speed += accel * dt
                    t += dt
        prev = coord
    return t
```

Algorithm

Let's define quadratic_roots...

```
def quadratic_roots(two_a, b, c):
    D = b * b - 2 * two_a * c
    if D >= 0:
        if D > 0:
            r = D ** 0.5
            roots = [(-b + r) / two_a, (-b - r) / two_a]
        else:
            roots = [-b / two_a]
    else:
        roots = []
    return roots
```

Algorithm

Aaaaaand put it all together

```
def main(n=6):
    (y1, y2) = (1.0, 0.0)
    (x1, x2) = (0.0, 1.0)
    coords = [  # initial guess: straight line
        [x1 + (x2 - x1) * i / n,
         y1 + (y2 - y1) * i / n]
        for i in range(n + 1)
    ]
    f = falling_time
    h = 0.00001
    while newton_minimizer_step(f, coords, h) > 0.01:
        print(coords)

if __name__ == '__main__':
    main()
```

Algorithm

(Demo)

Analysis

Error analysis: If x_∞ is the root and $\epsilon_k = x_k - x_\infty$ is the error, then:

$$\begin{aligned}
(x_{k+1} - x_\infty) &= (x_k - x_\infty) - \frac{f(x_k)}{f'(x_k)} && \text{(Newton step)}\\
\epsilon_{k+1} &= \epsilon_k - \frac{f(x_k)}{f'(x_k)} && \text{(error step)}\\
\epsilon_{k+1} &= \epsilon_k - \frac{\cancel{f(x_\infty)} + \epsilon_k f'(x_\infty) + \frac{1}{2}\epsilon_k^2 f''(x_\infty) + \cdots}{f'(x_\infty) + \epsilon_k f''(x_\infty) + \cdots} && \text{(Taylor series)}\\
\epsilon_{k+1} &= \frac{\frac{1}{2}\epsilon_k^2 f''(x_\infty) + \cdots}{f'(x_\infty) + \epsilon_k f''(x_\infty) + \cdots} && \text{(simplify)}
\end{aligned}$$

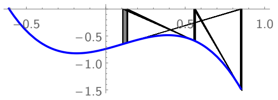
As $\epsilon_k \to 0$, the "$\cdots$" terms are quickly dominated. Therefore:

- If $f'(x_\infty) \approx 0$, then $\epsilon_{k+1} \propto \epsilon_k$ (slow: # of correct digits **adds**)
- Otherwise, we have $\epsilon_{k+1} \propto \epsilon_k^2$ (fast: # of correct digits **doubles**)

Analysis

Some failure modes:

- f is flat near root: too slow
- $f'(x) \approx 0$ = shoots off into infinity (n.b. `if x != 0` **not** a solution)
- Stable oscillation trap



Intuition: Think **adversarially**: create "tricky" f that *looks* root-less

- Obviously this is possible... just put the root far away
- Therefore Newton-Raphson *can't* be foolproof

Final thoughts

Notes: There are subtleties I brushed under the rug:

- The physics is **much** more complicated (why?)
- The numerical code can break easily (why?)

Can't tell why?

What happens if `y1 = 0.5` instead of `y1 = 1.0`?

Addendum 1

There's *never* a one-size-fits-all solution

- Must *always* know **something** about problem structure

Typical assumptions (stronger assumptions = better results):

- Vaguely predictable: **Continuity**
- Somewhat predictable: **Differentiability**
- Pretty predictable: **Smoothness** (infinite-differentiability)
- *Extremely* predictable: **Analyticity** (approximable by **polynomial**)
  - Function "equals" its infinite Taylor series
  - Also said to be *holomorphic*[3]

---

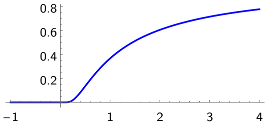
[3] Equivalent to *complex-differentiability*: $f'(x) = \lim_{h\to 0} (f(x+h) - f(x))/h$, $h \in \mathbb{C}$.

Addendum 1

Q: Does knowing $f(x_1)$, $f'(x_1)$, $f''(x_1)$, $\ldots$ let you predict $f(x_2)$?

A: Obviously! ...not :) counterexample:

$$f(x) = \begin{cases} e^{-1/x} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$



Indistinguishable from 0 for $x \le 0$

However, knowing derivatives **would** be enough for **analytic** functions!

Addendum 2

Fun facts:

- Why are *polynomials* fundamental? Why not, say, exponentials?
  - Pretty much **everything** is built on addition & multiplication!
  - Study of polynomials = **study of addition & multiplication**
- Polynomials are **awesome**
  - Polynomials can approximate real-world functions very well
  - Pretty much **everything** about polynomials has been solved
    - Global root bound (Fujiwara[4]) $\implies$ you know where to **start**
    - Minimal root separation (Mahler) $\implies$ you know when to **stop**
    - Guaranteed root-finding (Sturm) $\implies$ you can **binary-search**

---

[4] If $\sum_{k=0}^{n} a_{n-k} x^k = 0$ then $|x| \le 2 \max_k \sqrt[k]{|a_k/a_n|}$

Addendum 2

By contrast: Unlike + and ×, exponentiation is **not** well-understood!

Table-maker's dilemma (Prof. William Kahan):

- Nobody knows cost of computing x^y with correct rounding (!)
- We don't even know if it's possible with *finite* memory (!!!)

So, polynomials are *really* nice!

Addendum 3

Fun fact: If f is analytic, you can compute f' by evaluating f **only once**!

Any guesses how? **Complex-step differentiation!**

$$\begin{aligned}
f(x+ih) &\approx f(x) + i\,h\,f'(x) & \\
\mathrm{Im}\big(f(x+ih)\big) &\approx h\,f'(x) && \text{(imaginary parts match)}\\
f'(x) &\approx \frac{\mathrm{Im}\big(f(x+ih)\big)}{h} &
\end{aligned}$$

Features:

- More accurate: Avoids "catastrophic cancellation" in subtraction
- Faster (sometimes): f evaluated only once
- Difficult for ≥ 2^{nd} derivatives (need *multicomplex numbers*)

Done!

Hope you learned something new!

P.S.: Did you prefer the coding part? Or the math part?