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?

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 \( \pm \) directions to check for increase/decrease
    - Good luck in N-D... infinitely many directions

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 \( \pm \) directions to check for increase/decrease
    - Good luck in N-D... infinitely many directions
Wait, but we have a function of many variables. What do?

A couple options:

- Fully multivariate Newton-Raphson:
  \[ \mathbf{x}_{k+1} = \mathbf{x}_k - \nabla^2 f(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k) \]

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

- Newton coordinate-descent

Newton for minimization:

```python
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?

What is our objective function \( F \) to minimize?

```python
def falling_time(coords):
    t, speed = 0.0, 0.0
    prev = None
    for coord in coords:
        if prev != None:
            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
```

Let's define quadratic_roots...

```python
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
```

Aaaaaand put it all together

```python
def main(n=6):
    (y1, y2) = (1.0, 0.0)
    (x1, x2) = (0.0, 1.0)
    coords = [ [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()
```

Coordinate descent:

- Take \( x_1 \), use it to minimize \( F \), holding others fixed
- Take \( y_1 \), use it to minimize \( F \), holding others fixed
- Take \( x_2 \), use it to minimize \( F \), holding others fixed
- Take \( y_2 \), use it to minimize \( F \), holding others fixed
- ...
- Cycle through again

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

Error analysis: If \( x_{k+1} \) is the root and \( \epsilon_k = x_k - x_{k+1} \) is the error, then:

\[
(x_{k+1} - x_k) = (x_k - x_{k+1}) - \frac{f(x_k)}{f'(x_k)}
\]

(\text{Newton step})

\[
\epsilon_{k+1} = \epsilon_k - \frac{f(x_k)}{f'(x_k)}
\]

(\text{error step})

\[
\epsilon_{k+1} = \epsilon_k - \frac{f(x_k) + \frac{1}{2}f'(x_k) + \cdots}{f'(x_k)}
\]

(\text{Taylor series})

\[
\epsilon_{k+1} = \epsilon_k \frac{f''(x_k) + \cdots}{f'(x_k)}
\]

(simplify)

As \( \epsilon_k \to 0 \), the "..." terms are quickly dominated. Therefore:

- If \( f'(x_k) \approx 0 \), then \( \epsilon_{k+1} \approx \epsilon_k \) (slow: # of correct digits adds)
- Otherwise, we have \( \epsilon_{k+1} \approx \epsilon_k \) (fast: # of correct digits doubles)

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 \( y_1 = 0.5 \) instead of \( y_1 = 1.0 \)?

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:

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

However, knowing derivatives would be enough for analytic functions!

Addendum 2

By contrast: Unlike \( + \) and \( \times \), exponentiation is not well-understood!

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

- Nobody knows cost of computing \( x^n \) 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!

\[
f(x + ih) = f(x) + ih f'(x)
\]

\[
\text{Im}(f(x + ih)) = h f'(x)
\]

(\text{imaginary parts match})

\[
f'(x) = \frac{\text{Im}(f(x + ih))}{h}
\]

Features:

- More accurate: Avoids “catastrophic cancellation” in subtraction
- Faster (sometimes): \( f \) evaluated only once
- Difficult for \( \geq 2^{n} \) derivatives (need multicomplex numbers)
Hope you learned something new!

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