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?!
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?

How would *you* solve this?

⇒ 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
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?
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
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 $f = F'$:

   $$f(x_k) - f(x_{k+1}) = f'(x_k) (x_k - x_{k+1})$$
   $$f(x_{k+1}) = 0$$

   $$\implies x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

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

A couple options:

1. Fully multivariate Newton-Raphson:

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

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**: 

```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?
Algorithm

Computing derivatives numerically:

```python
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 \( \frac{f(x + h) - f(x)}{h} \)?

- Breaking the intrinsic asymmetry reduces accuracy

\[ \sim \textit{Words of Wisdom} \sim \]

If your problem has \{fundamental feature\} that your solution doesn’t, you’ve created more problems.
What is our **objective function** $F$ to minimize?

```python
def falling_time(coords):  # coords = [[x1,y1], [x2,y2], ...]
    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...

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

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

(Newton step)

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

(error step)

\[
\epsilon_{k+1} = \epsilon_k - \frac{f(x_\infty) + \epsilon_k f'(x_\infty) + \frac{1}{2} \epsilon_k^2 f''(x_\infty) + \ldots}{f'(x_\infty) + \epsilon_k f''(x_\infty) + \ldots}
\]

(Taylor series)

\[
\epsilon_{k+1} = \frac{1}{2} \frac{\epsilon_k^2 f''(x_\infty) + \ldots}{f'(x_\infty) + \epsilon_k f''(x_\infty) + \ldots}
\]

(simplify)

As $\epsilon_k \to 0$, the “…” 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)
Some failure modes:

- $f$ is flat near root: too slow
- $f'(x) \approx 0$ = shoots off into infinity (n.b. if $x \neq 0$ 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 $y_1 = 0.5$ instead of $y_1 = 1.0$?
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: \( f'(x) = \lim_{h \to 0} (f(x + h) - f(x))/h, \ h \in \mathbb{C}. \)
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 \( x \leq 0 \)

However, knowing derivatives \textbf{would} be enough for \textit{analytic} functions!
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\(^4\)) \(\Rightarrow\) you know where to start
    - Minimal root separation (Mahler) \(\Rightarrow\) you know when to stop
    - Guaranteed root-finding (Sturm) \(\Rightarrow\) you can binary-search

\[^4\text{If } \sum_{k=0}^n a_n-kx^k = 0 \text{ then } |x| \leq 2 \max_k \sqrt[k]{|a_k/a_n|}\]
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!
**Fun fact:** If $f$ is analytic, you can compute $f'$ by evaluating $f$ **only once**!

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

\[
f(x + ih) \approx f(x) + i \ h \ f'(x) \\
\text{Im}(f(x + ih)) \approx h \ f'(x) \quad \text{(imaginary parts match)} \\
f'(x) \approx \frac{\text{Im}(f(x + ih))}{h}
\]

**Features:**

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

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