CSE 421 Winter 2025
Lecture 27:
Finale

Nathan Brunelle

http://www.cs.uw.edu/421

Other Approches you might hear about

Genetic algorithms:

- View each solution as a **string** (analogy with DNA)
- Maintain a **population of good solutions**
- Allow **random mutations** of single characters of individual solutions
- **Combine two solutions** by taking part of one and part of another (analogy with crossover in sexual reproduction)
- Get rid of solutions that have the worst values and make multiple copies of solutions that have the best values (analogy with natural selection -- survival of the fittest).

*Usually very slow. In the rare cases when they produce answers with better objective function values than other methods they tend to produce very **brittle** solutions – that are very bad with respect to small changes to the requirements.*

Deep Neural Nets and NP-hardness?

- **Artificial neural networks**
  - based on very elementary model of human neurons
  - **Set up a circuit of artificial neurons**
    - each artificial neuron is an analog circuit gate whose computation depends on a set of **connection strengths**
  - **Train the circuit**
    - Adjust the connection strengths of the neurons by giving many positive & negative training examples and seeing if it behaves correctly
  - **The network is now ready to use**

Despite their wide array of applications, they have not been shown to be useful for NP-hard problems.

Quantum Computing and NP-hardness?

Use physical processes at the quantum level to implement "weird" kinds of circuit gates based on unitary transformations

- Quantum objects can be in a "superposition" of many pure states at once
  - Can have n objects together in a superposition of 2^n states
- Each quantum circuit gate operates on the whole superposition of states at once
  - Inherent parallelism but classical randomized algorithms have a similar parallelism: *not enough on its own*
  - Advantage over classical: copies interfere with each other.
- Exciting direction - theoretically able to factor efficiently.
  Major practical problems wrt errors, decoherence to be overcome.
- *Small brute force improvement but unlikely to produce exponential advantage for NP.*

Summary: If you need to solve an NP-Hard Problem

- Look for assumptions that simplify the problem
- Made design decisions so that you can make simplifying assumptions
- Fix some parameter to a reasonable size, then write an algorithm that is exponential only in that parameter (and therefore polynomial when fixed)
- Give up on trying to find the best solution, and instead approximate
  - When a minimization/maximization problem
  - May be helpful to find/create assumptions for better approximation
  - Sometimes a LP can help with this!
- Give up on trying to write a polynomial time algorithm at all, and instead use a fast exponential time algorithm
  - For example, reduce your problem to SAT (if it's NP-complete), then use an out-of-the-box SAT solver

Minimum Vertex Cover as a "01 Program"

Minimum Vertex-Cover:

Given a graph $\boldsymbol{G = (V, E)}$

Find the largest $\boldsymbol{W \subseteq V}$ with $\boldsymbol{|W|}$ such that every edge of $\boldsymbol{G}$ has an endpoint in $\boldsymbol{W}$? ($\boldsymbol{W}$ is a vertex cover, a set of vertices that covers $\boldsymbol{E}$.)

Minimize: $\sum_{u \in V} x_u$

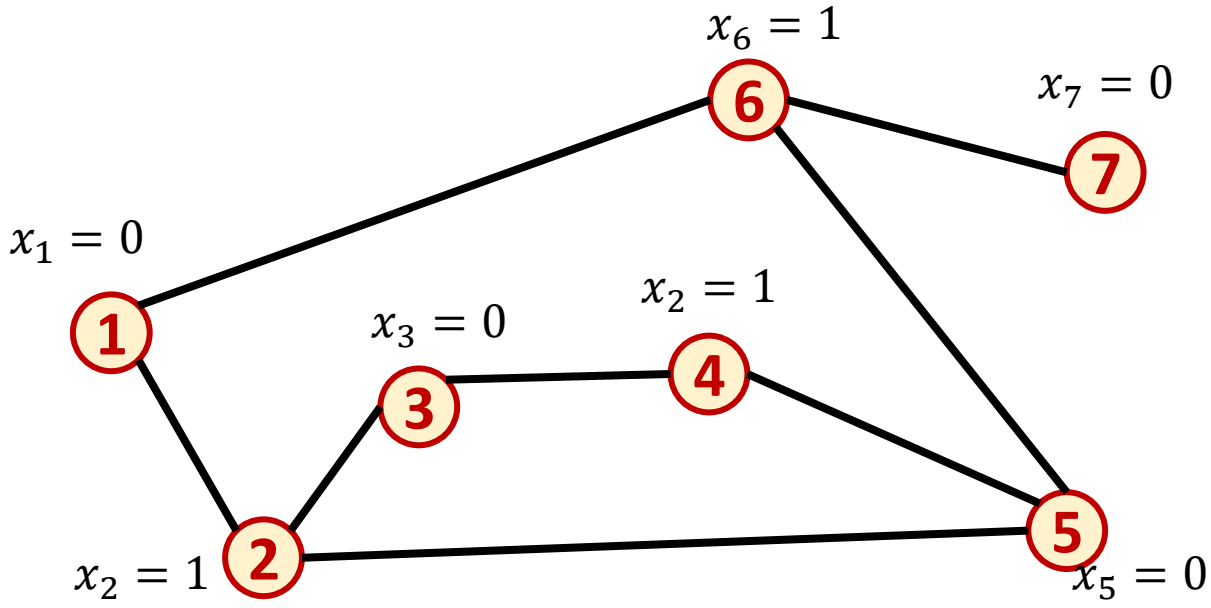
Subject to:

$x_u + x_v \geq 1$ for all $(u, v) \in E$

$x_u \in \{0,1\}$

x_u indicates whether to include vertex u

$x_u \in \{0,1\}$ is not expressible as an LP!



Minimum Vertex Cover as a LP

Minimum Vertex-Cover:

Given a graph $\boldsymbol{G = (V, E)}$

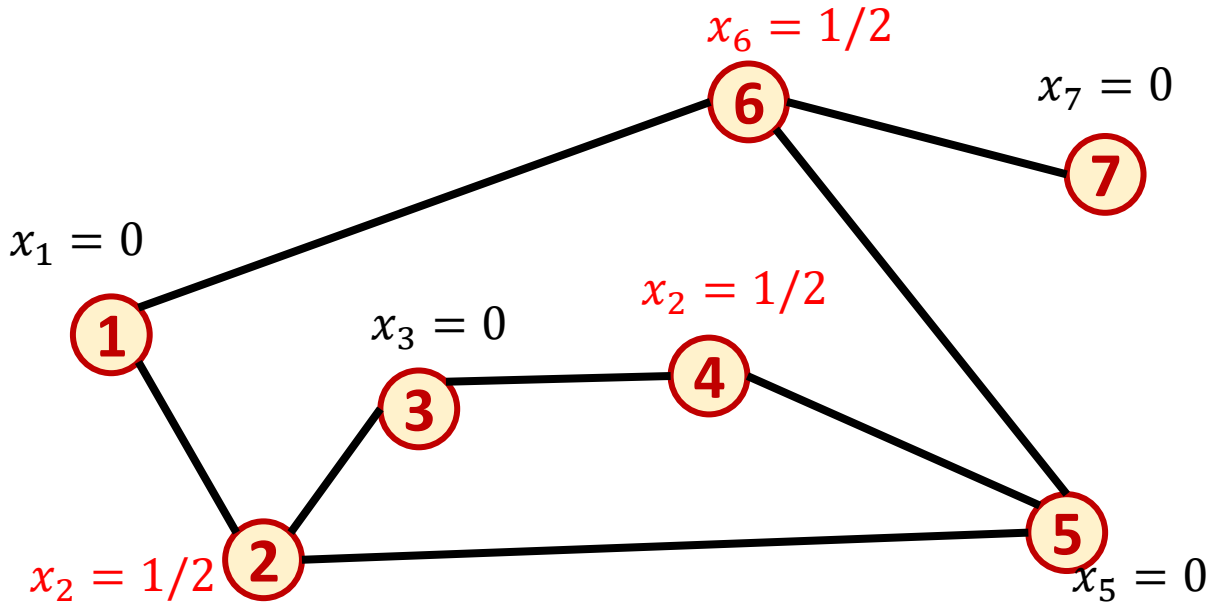
Find the largest $\boldsymbol{W \subseteq V}$ with $\boldsymbol{|W|}$ such that every edge of $\boldsymbol{G}$ has an endpoint in $\boldsymbol{W}$? ($\boldsymbol{W}$ is a vertex cover, a set of vertices that covers $\boldsymbol{E}$.)

Minimize: $\sum_{u \in V} x_u$

Subject to:

$$x_u + x_v \geq 1 \text{ for all } (u, v) \in E$$
$$x_u \leq 1$$
$$x_u \geq 0$$

Solution: let x_u be a value between 0 and 1, then round



Why is this a Vertex Cover?

Why does this produce a VC?

$x_u + x_v \geq 1$ guarantees at least one of x_u, x_v is $\geq \frac{1}{2}$, so at least one end point is selected

Minimize: $\sum_{u \in V} x_u$

Subject to:

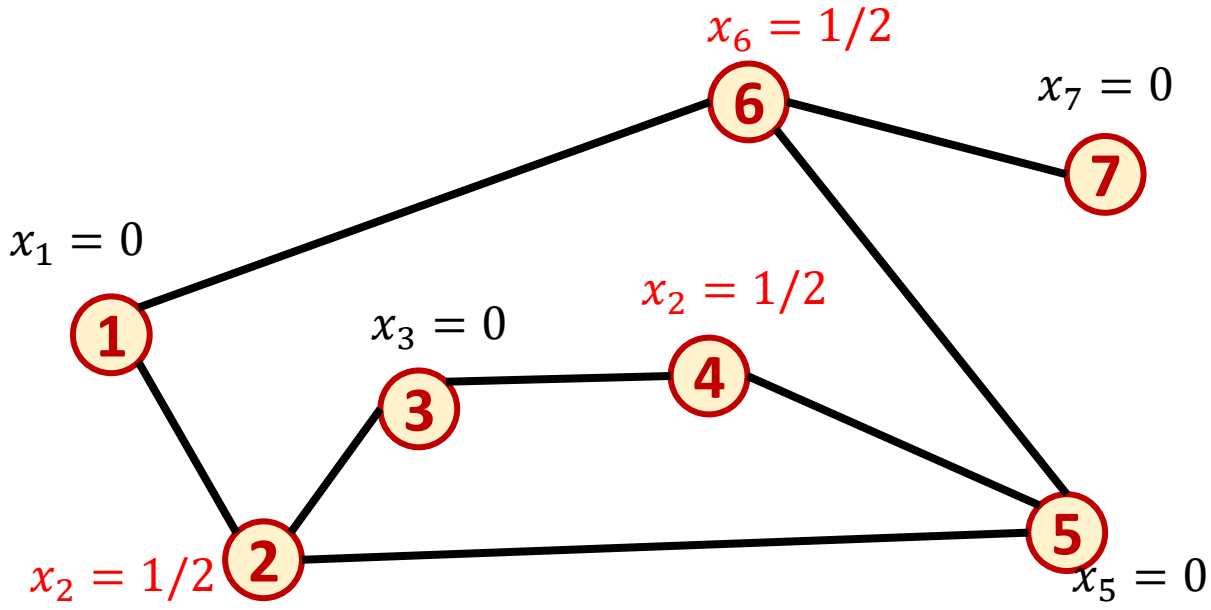
$x_u + x_v \geq 1$ for all $(u, v) \in E$

$x_u \leq 1$

$x_u \geq 0$

Solve, then round each x_u

if $x_u \geq \frac{1}{2}$ then set $x_u = 1$



How good is it?

- Let LP refer to the value of the LP solution (i.e. $\sum_{u \in V} x_u$)
- Let ALG refer to the size of the VC we select (i.e. number of x_u rounded up)
- Let OPT refer to the size of the minimum vertex cover
- $\boldsymbol{ALG \leq 2 \cdot LP}$ because we don't do worse than doubling when rounding
- $\boldsymbol{LP \leq OPT}$ because the true minimum vertex cover is a feasible solution to the linear program
- $\boldsymbol{ALG \leq 2 \cdot OPT}$ by combining

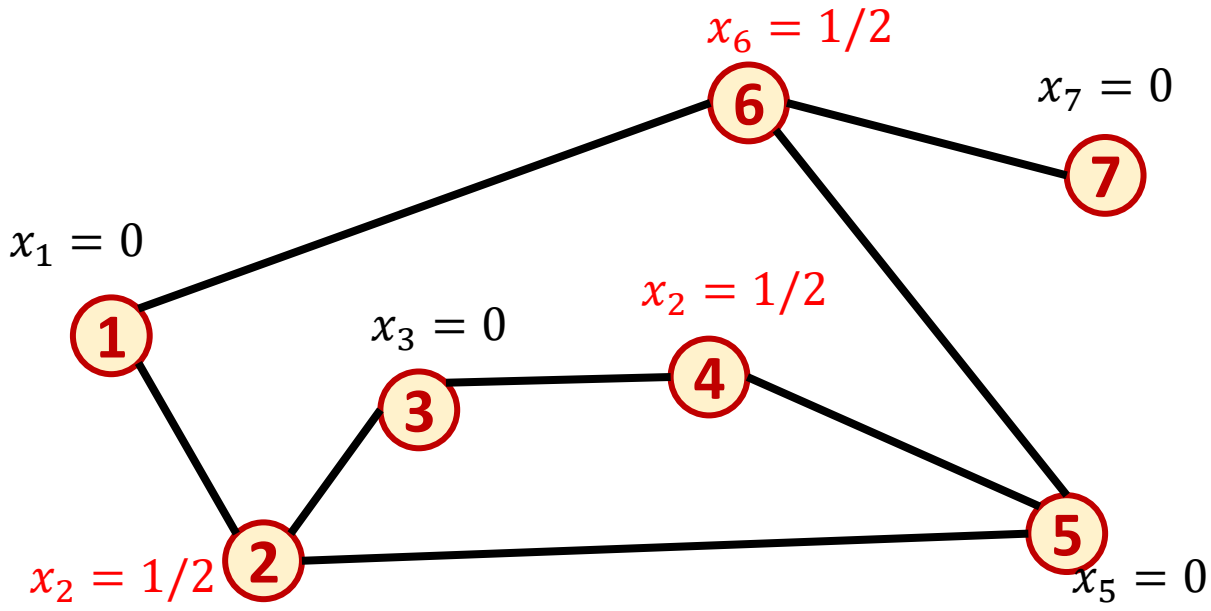
Minimize: $\sum_{u \in V} x_u$

Subject to:

$$x_u + x_v \geq 1 \text{ for all } (u, v) \in E$$
$$x_u \leq 1$$
$$x_u \geq 0$$

Solve, then round each x_u

if $x_u \geq \frac{1}{2}$ then set $x_u = 1$



Algorithms for Linear Programs

Simplex Algorithm

- Simple
- Often fast in practice
- Not polynomial time (on pathological counterexamples)

Ellipsoid Algorithm

- More complicated
- First polynomial time algorithm, but not always fast

Interior Point Methods

- Even more complicated based on differential equation ideas
- Polynomial time, fast in practice; simplex better for small input size

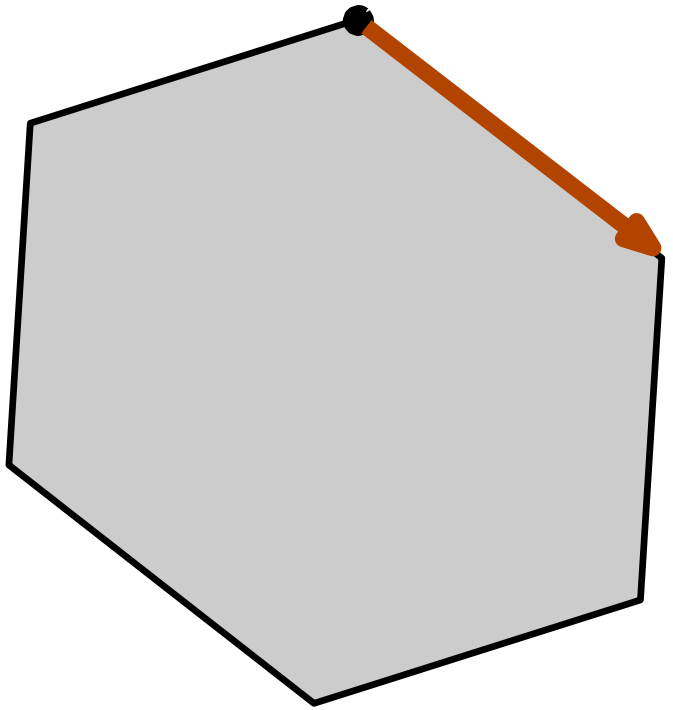
The Simplex Algorithm

Simplex Algorithm:

- Start with a vertex of the polytope
- In each step move to a neighboring vertex that is *lower* (larger $\boldsymbol{c}^\top \boldsymbol{x}$).

Creates a path running along the edges and vertices on the outside of the polytope

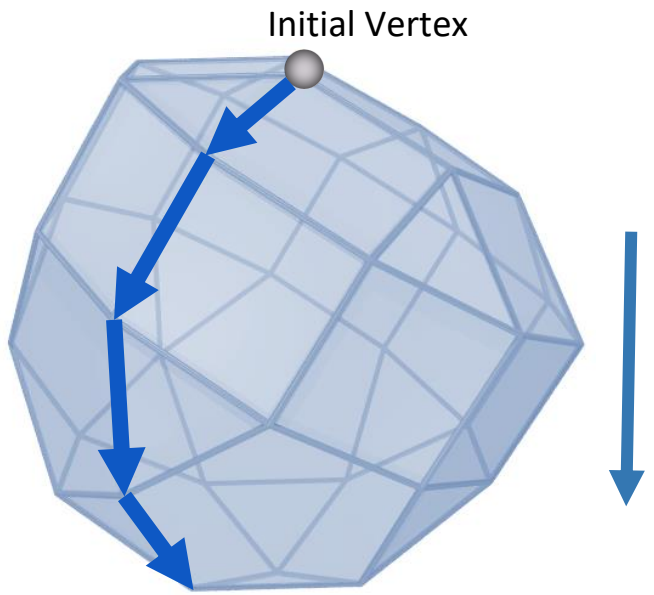
- Since the polytope is convex, this will never get stuck before reaching the lowest point.



The Simplex Algorithm: The downside

Simplex Algorithm:

- Start with a vertex of the polytope
- In each step move to a neighboring vertex that is *lower* (larger $\boldsymbol{c}^{\mathsf{T}}\boldsymbol{x}$).



Creates a path running along the edges and vertices on the outside of the polytope

- Since the polytope is convex, this will never get stuck before reaching the lowest point.

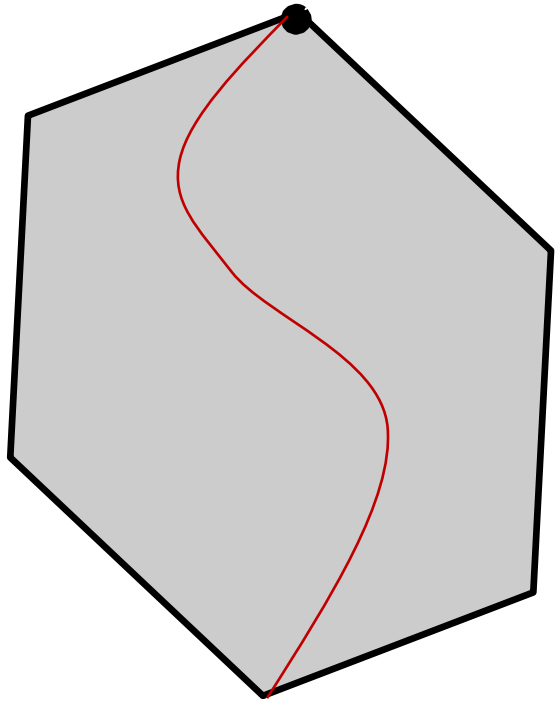
Problem: Many paths to choose from; # of vertices on path can be exponential!

Interior Point Algorithms

Interior Point Idea:

- Start with a point in the polytope, either a vertex or in the interior
- Follow approximations to a curving "central path" that
  - tunnels through the polytope
  - avoids the boundary using loss functions

  and eventually gets to the optimum

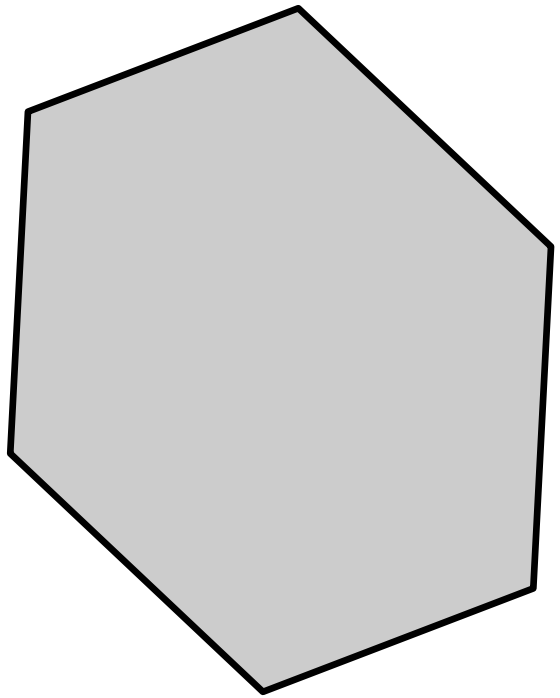


Can be implemented efficiently using data structure tricks. Also leads to best randomized algorithms for network flow. Super complicated, I'd rather skip it.

Ellipsoid Method

Idea:

- If I had a way of finding any point in the feasible region then I can find the optimal vertex
  - Using a “binary search” strategy
- I can find a point in the feasible region by identifying using progressively smaller “balls” which contain the region



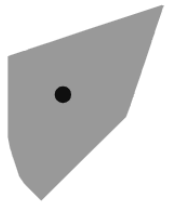
Using binary search

$y = T$

$y = -T$

Check if polytope is empty using FindPoint

$y = T$



$y = -T$

Add new constraint

$y = T$

$y \leq 0$

$y = -T$

Call FindPoint

$y = T$

$y \leq 0$

$y = -T$

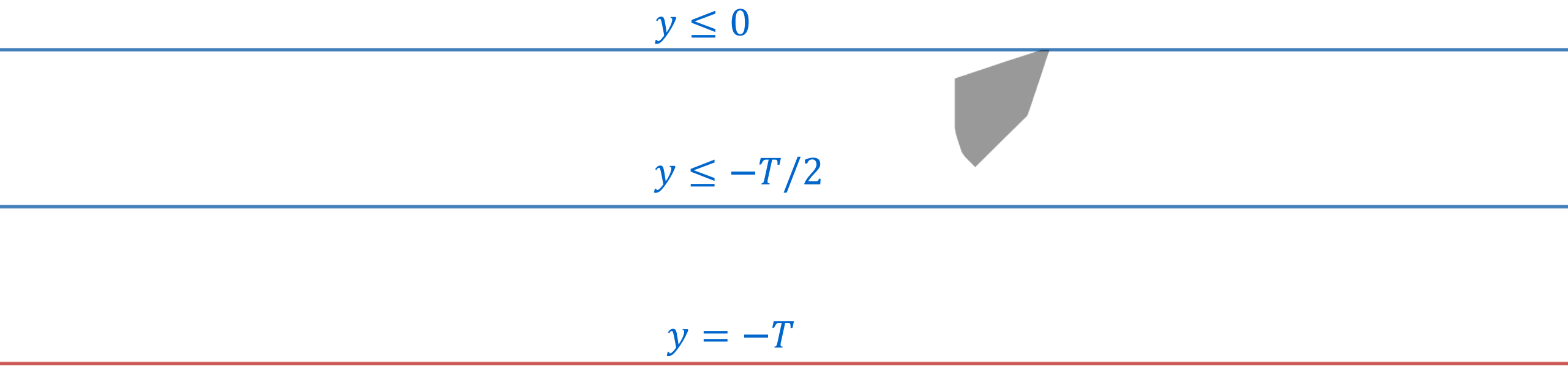
Add new constraint

$y \leq 0$

$y \leq -T/2$

$y = -T$

FindPoint: Polytope is empty!



Add new constraint

$y \leq 0$

$y \leq -T/4$

$y \leq -T/2$

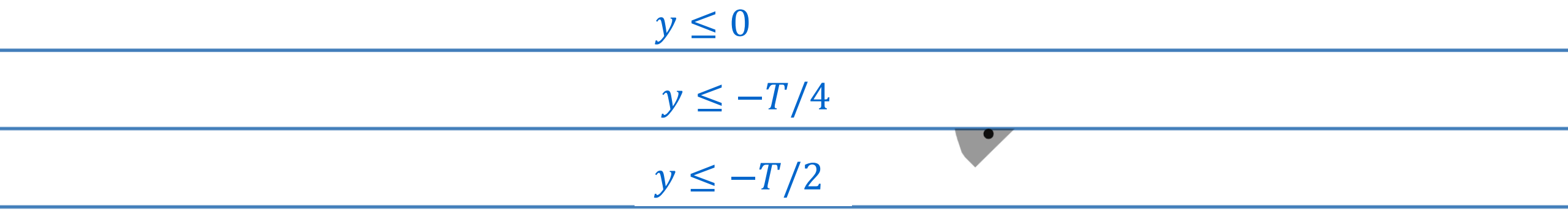
Add new constraint

$$y \leq 0$$

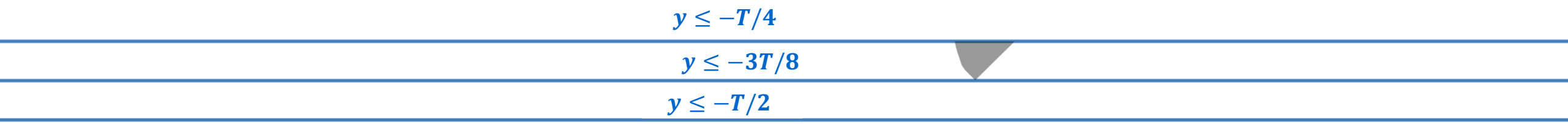
$$y \leq -T/4$$

$$y \leq -T/2$$

Find point



Add new constraint



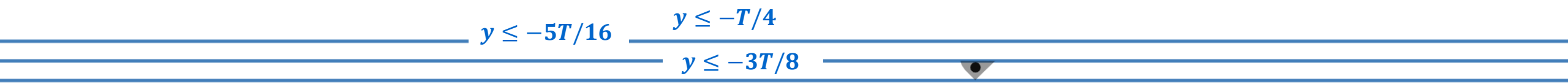
Find point: Polytope is empty!

$$y \leq -T/4$$

$$y \leq -3T/8$$

$$y \leq -T/2$$

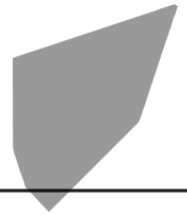
Add new constraint… Find point …

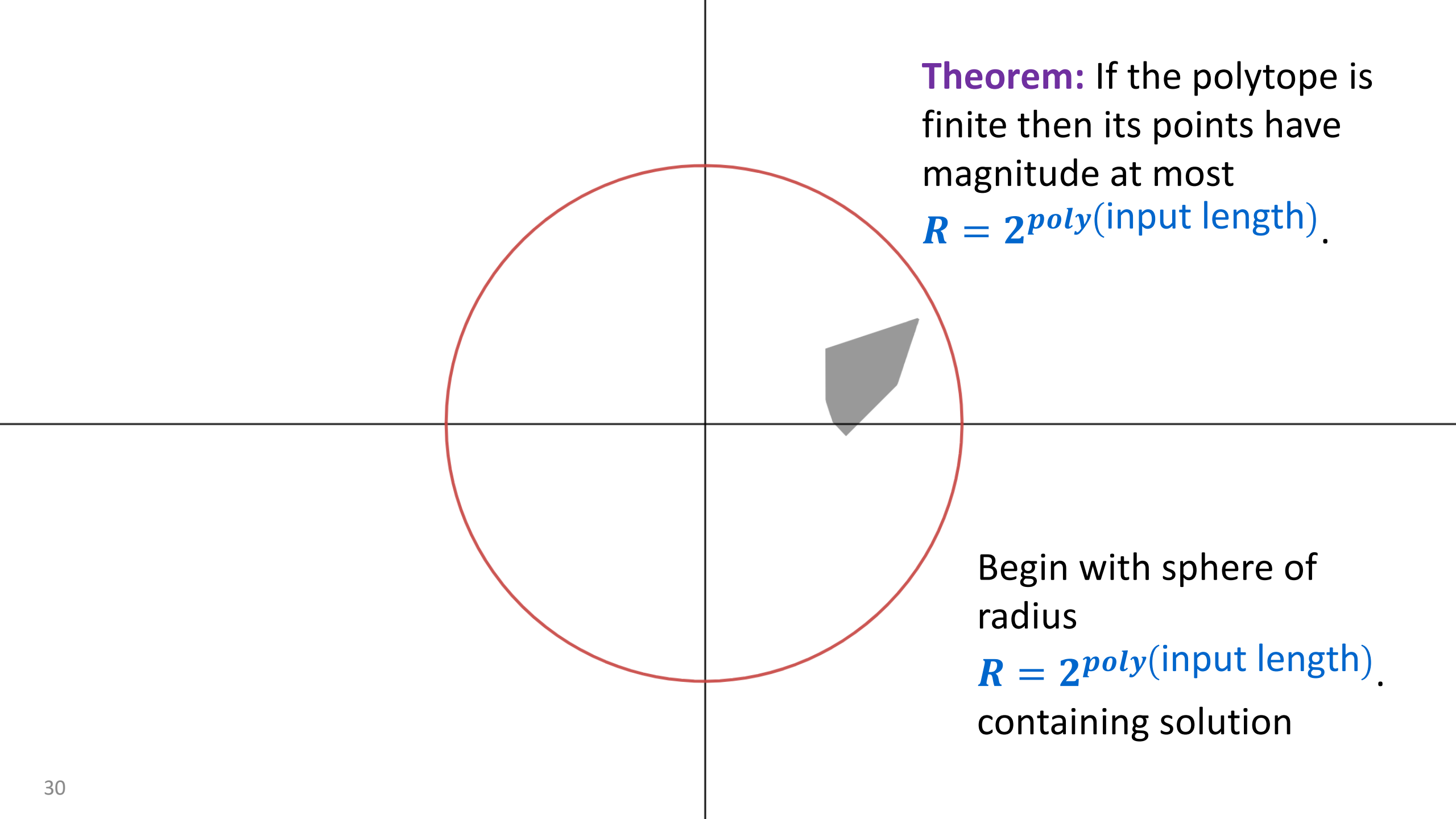


Conclusion: It is enough to give an algorithm to find a point in a polytope.

Ellipsoid algorithm for finding points in polytopes

Idea: Iteratively find ellipsoids where the density of the polytope within each ellipsoid is larger and larger, until a point is found

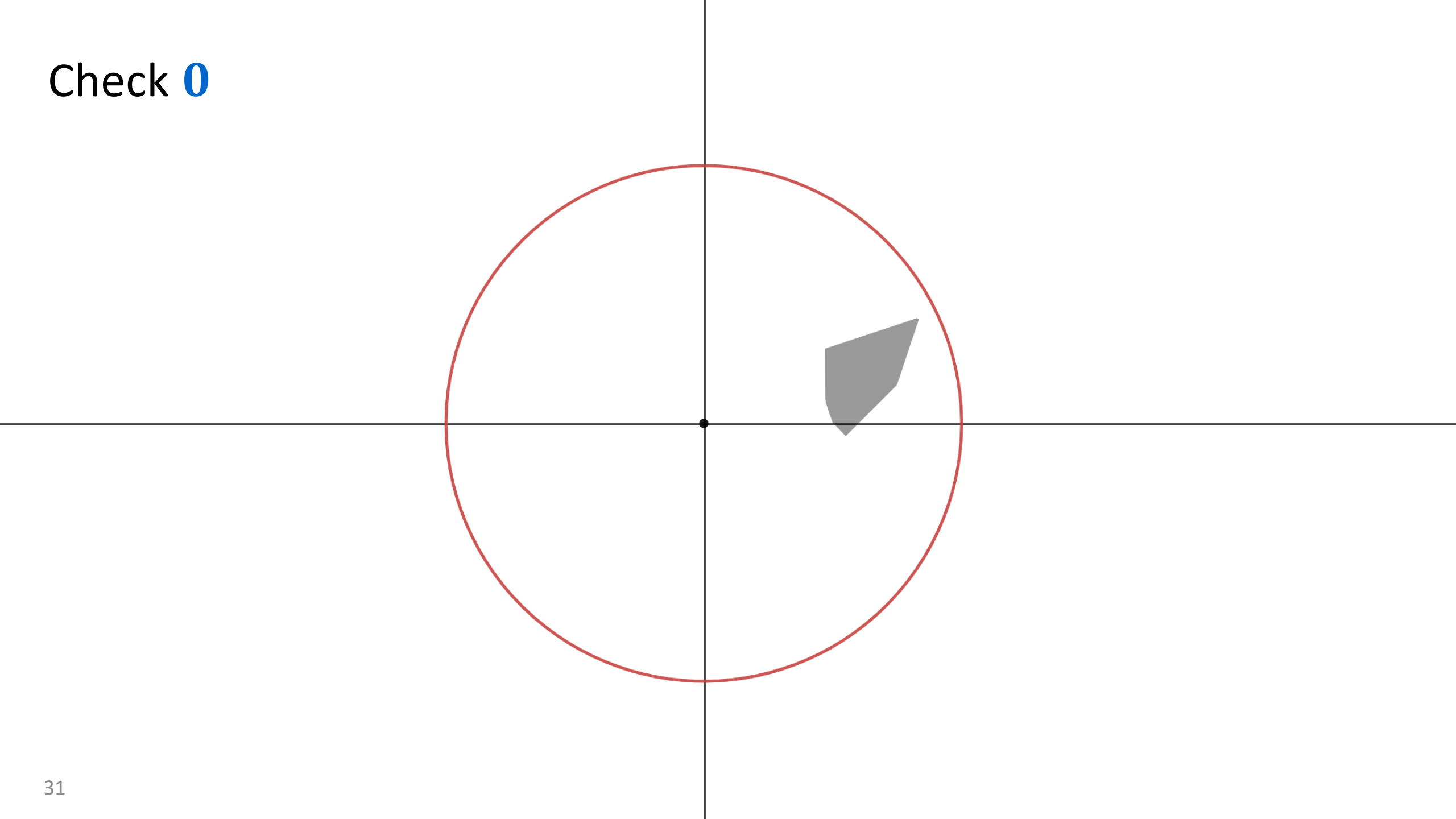




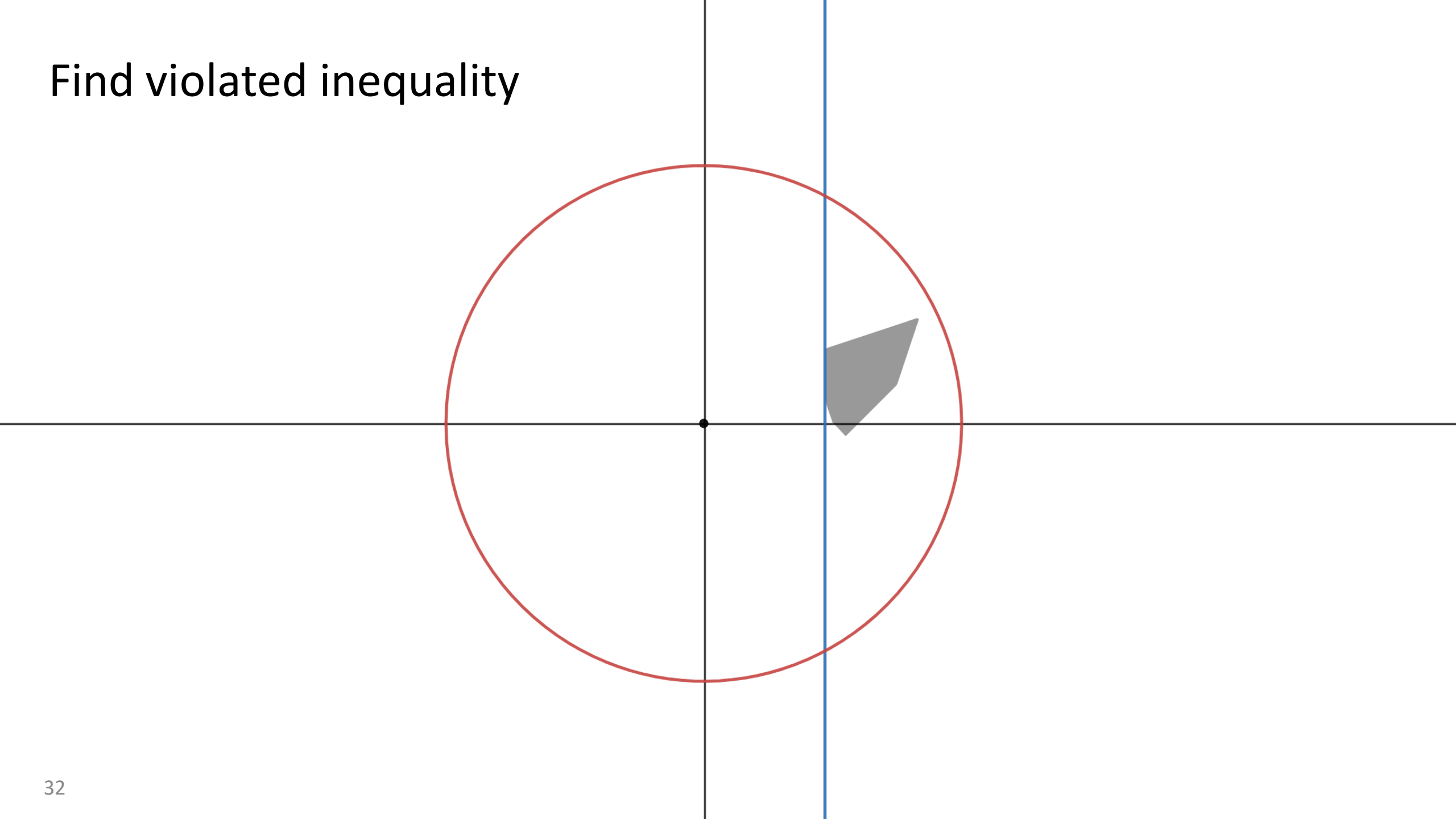
Theorem: If the polytope is finite then its points have magnitude at most $\boldsymbol{R = 2^{poly}}$(input length).

Begin with sphere of radius $\boldsymbol{R = 2^{poly}}$(input length). containing solution

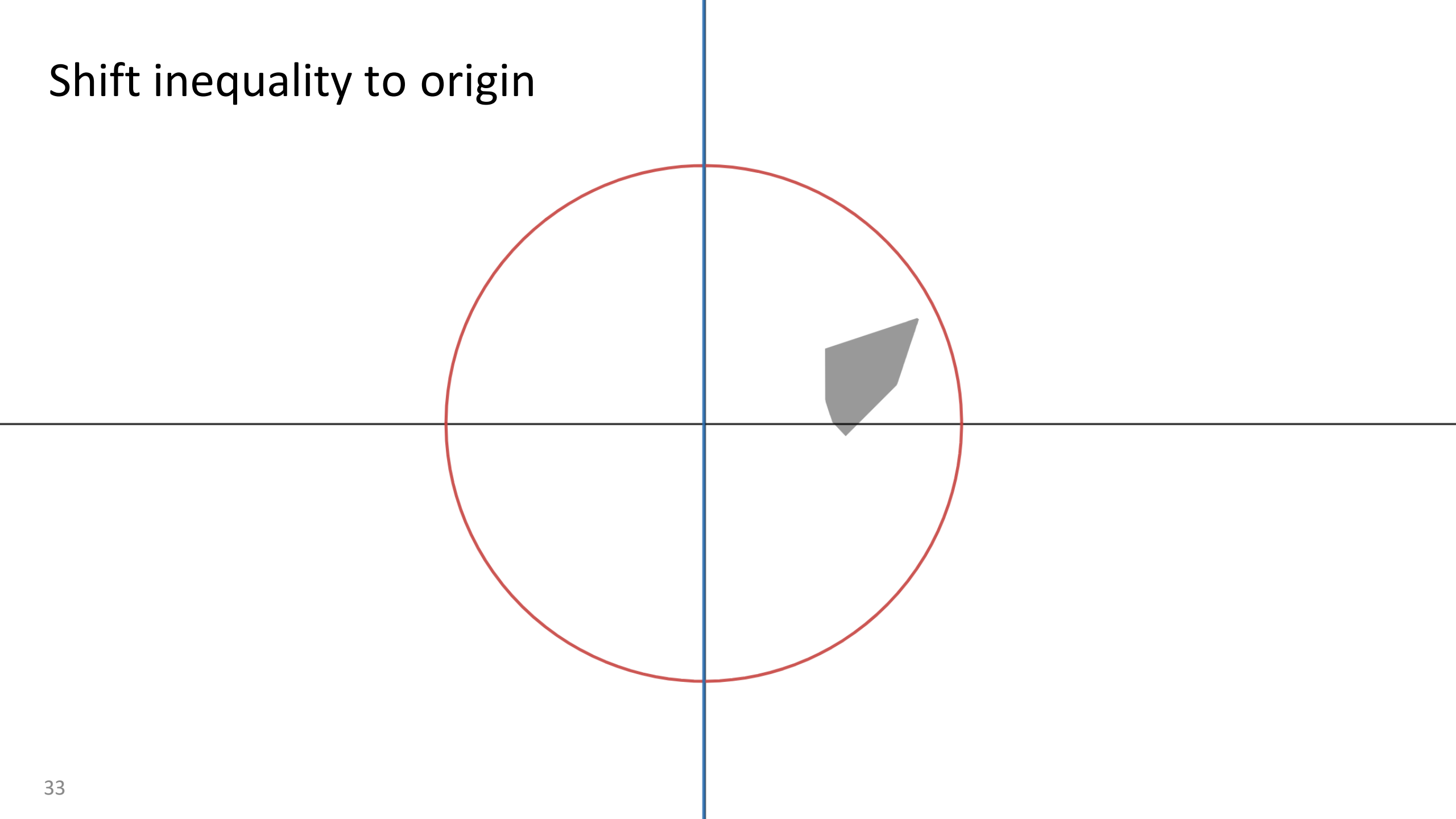
Check **0**



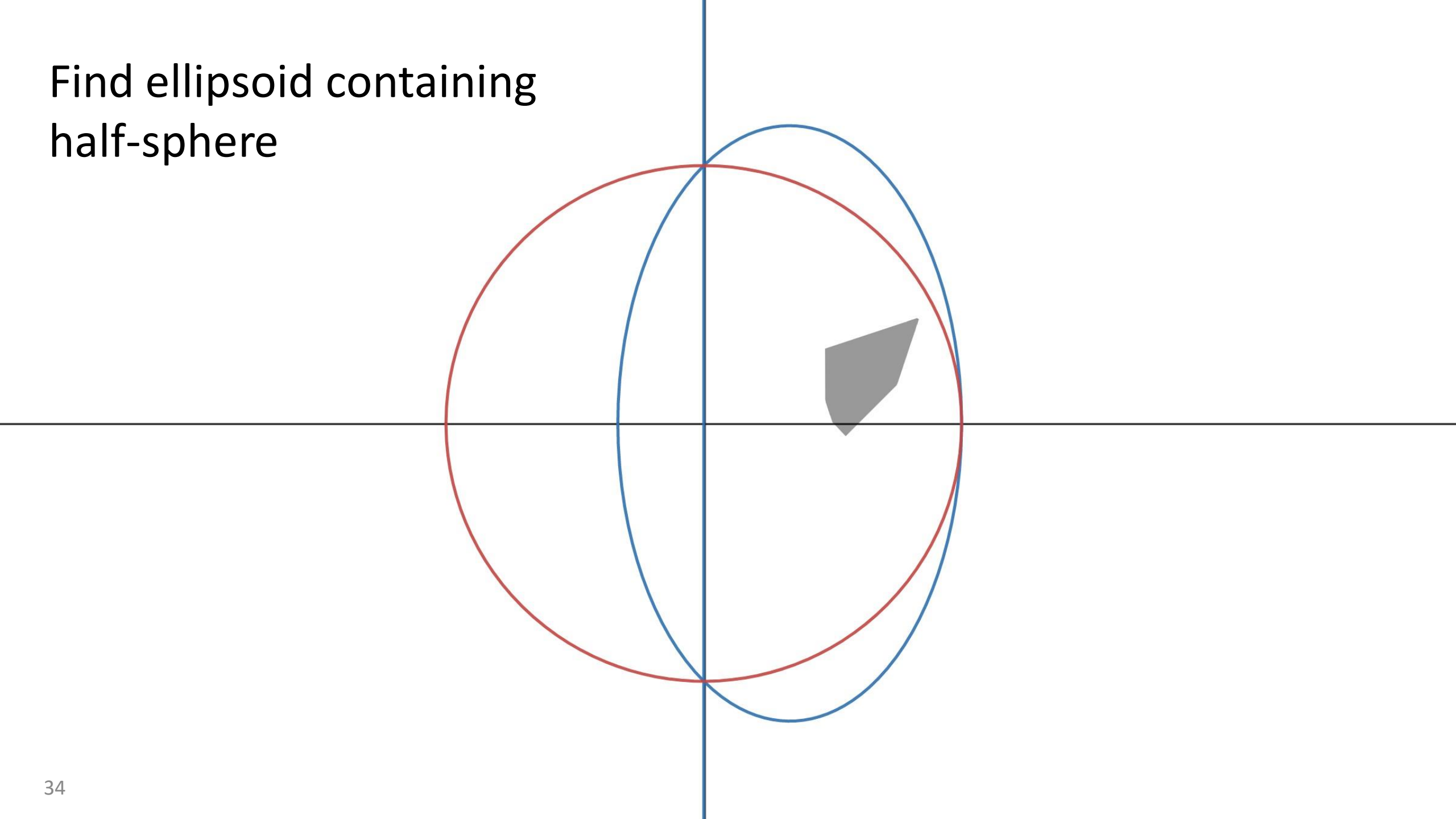
Find violated inequality



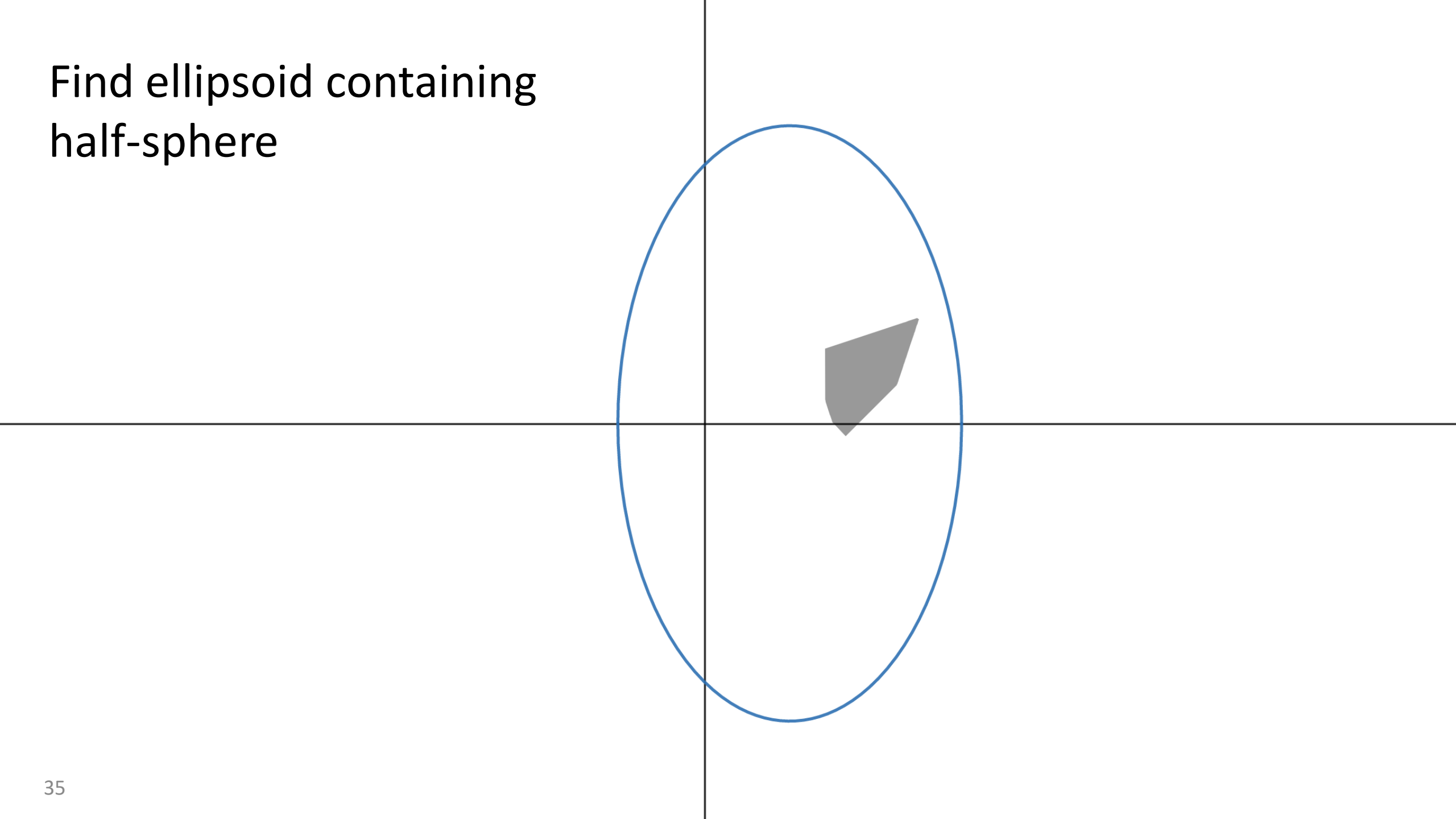
Shift inequality to origin



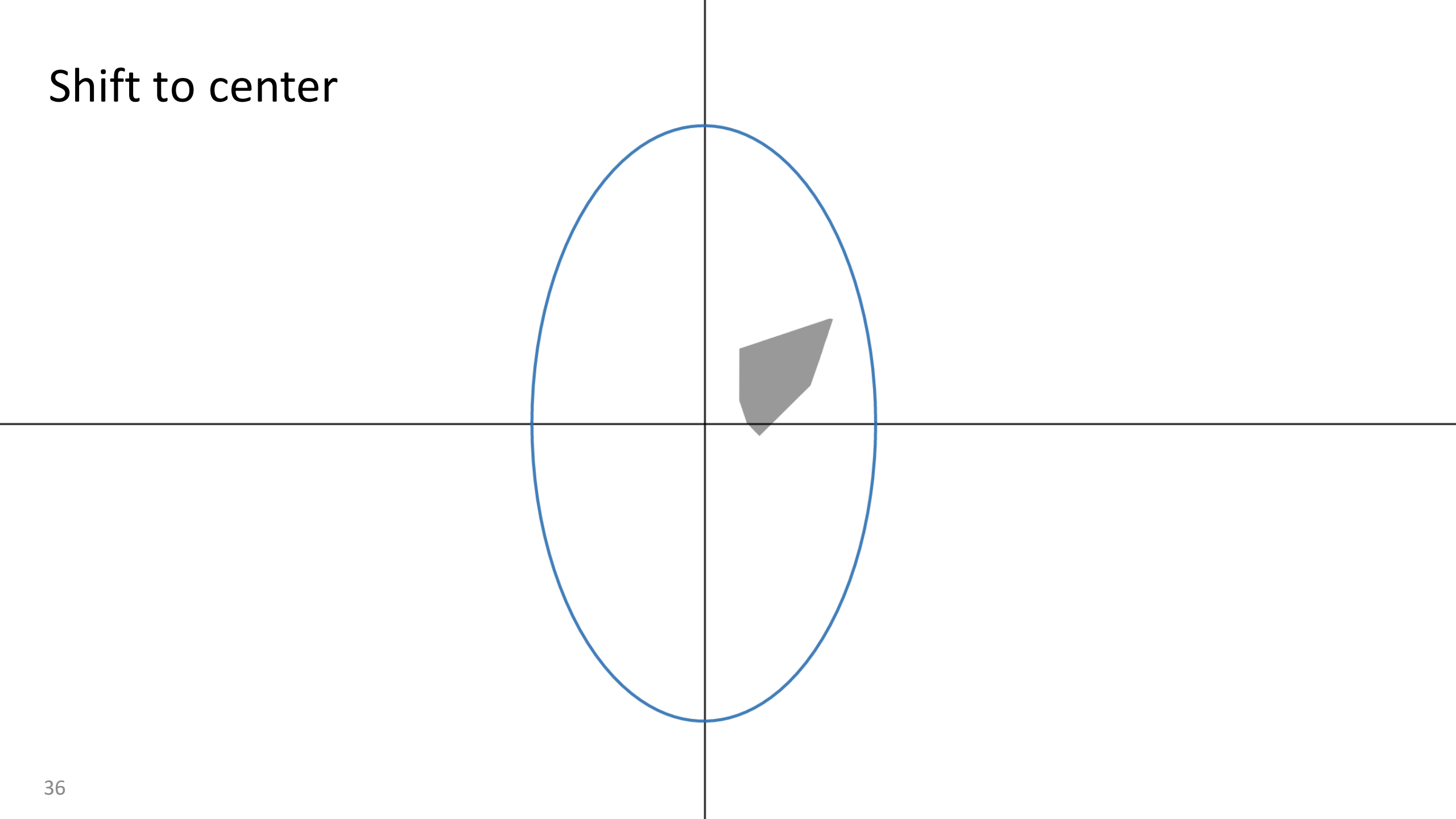
Find ellipsoid containing half-sphere



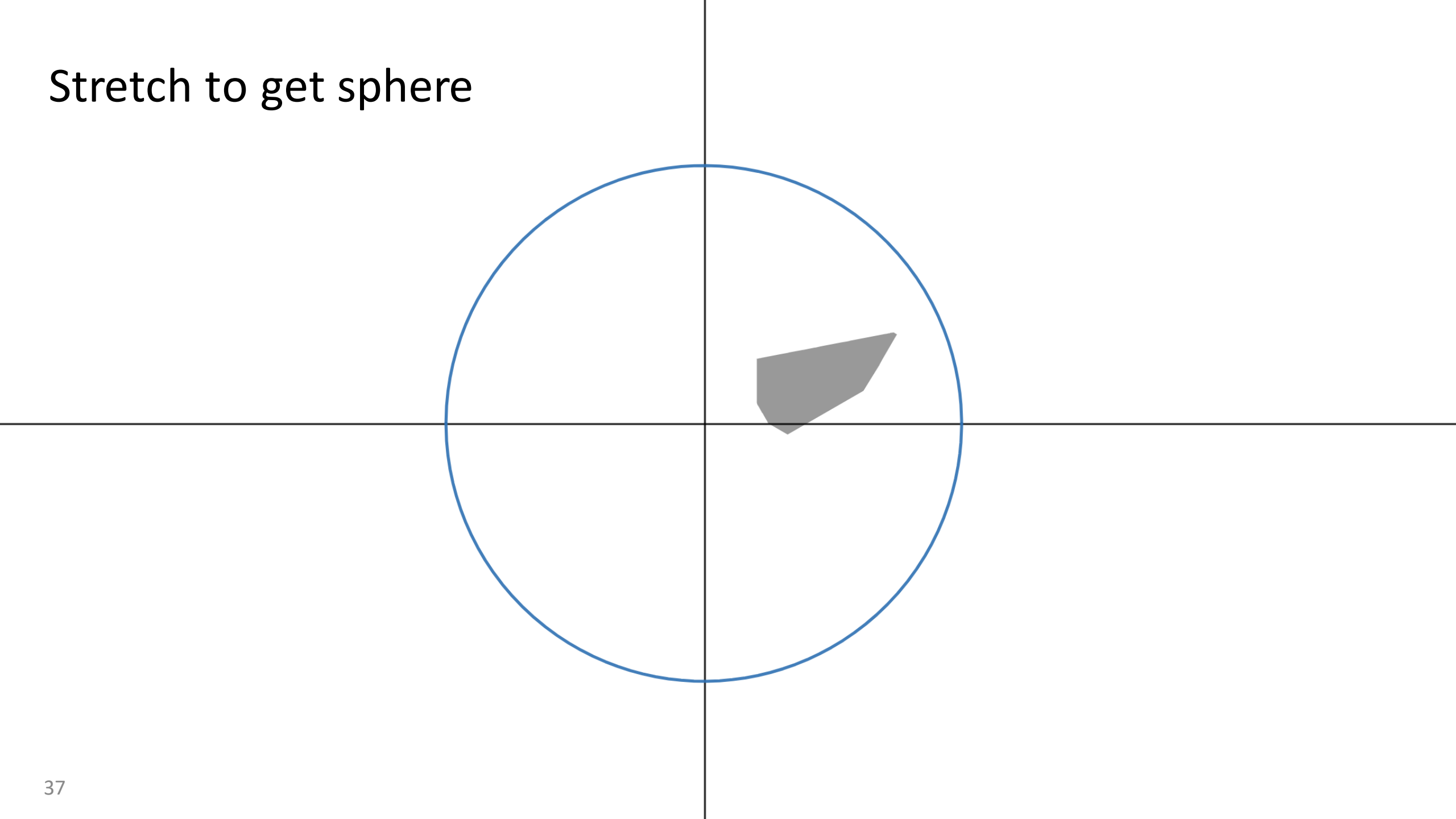
Find ellipsoid containing half-sphere



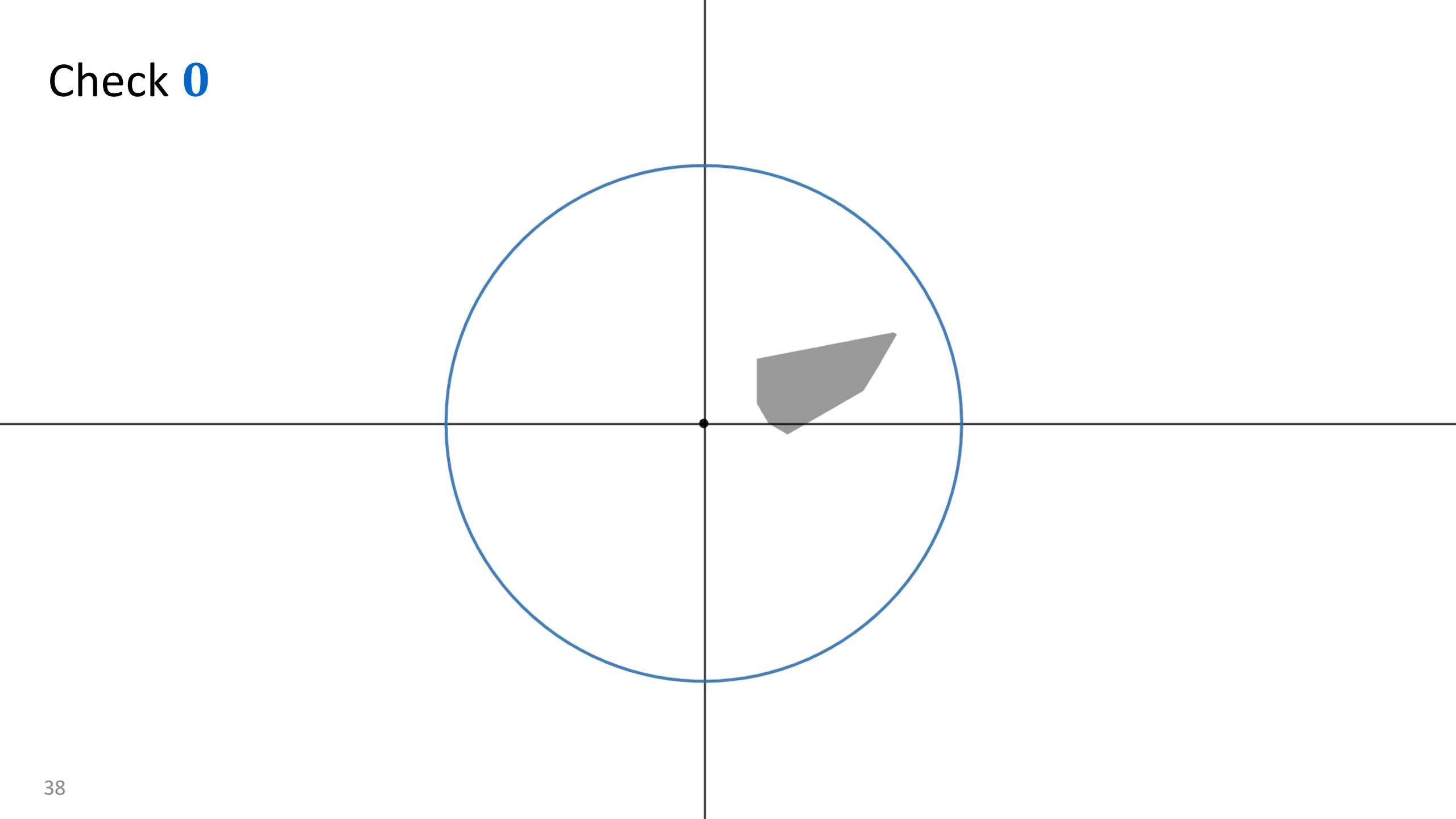
Shift to center



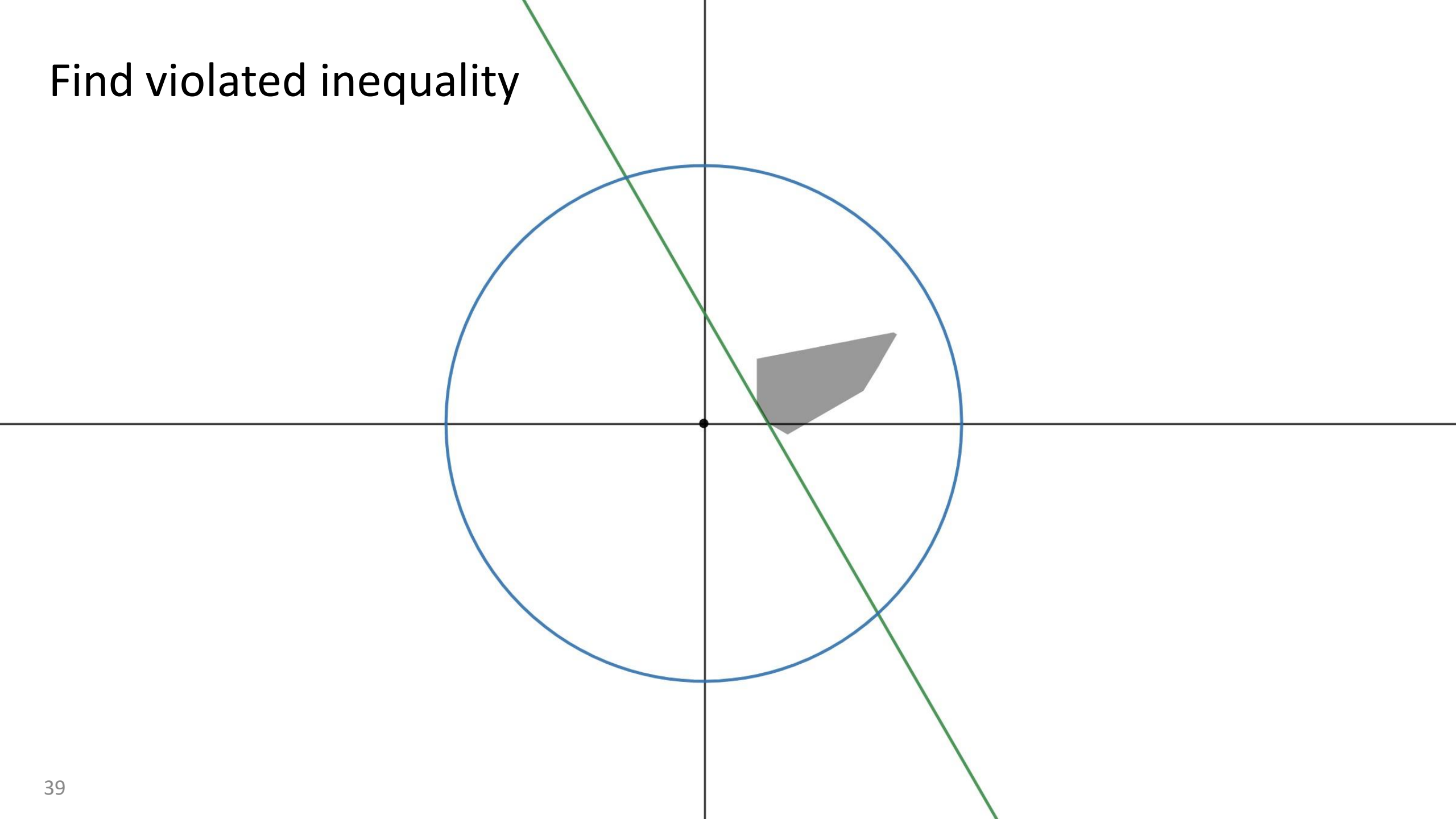
Stretch to get sphere



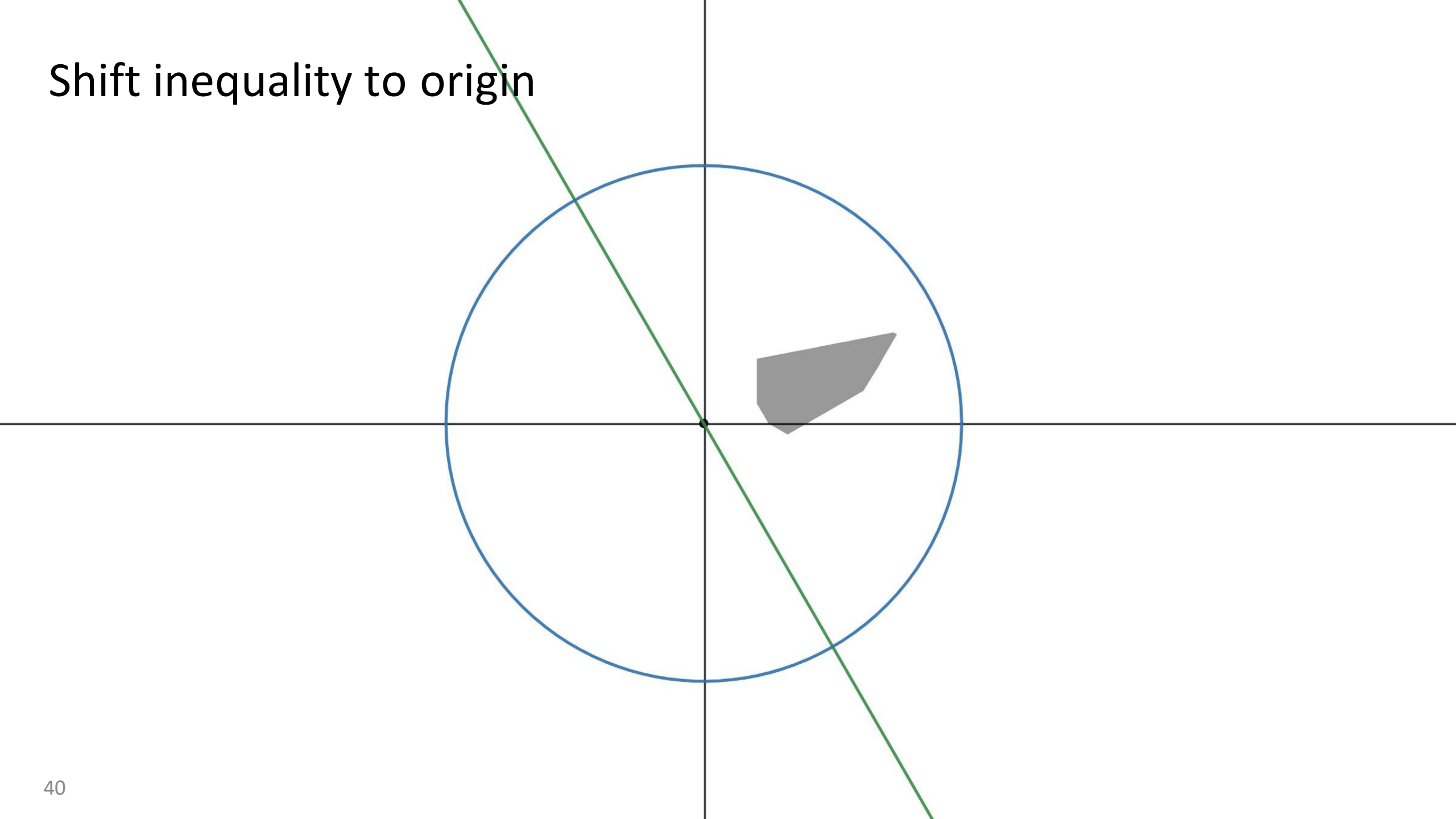
Check **0**



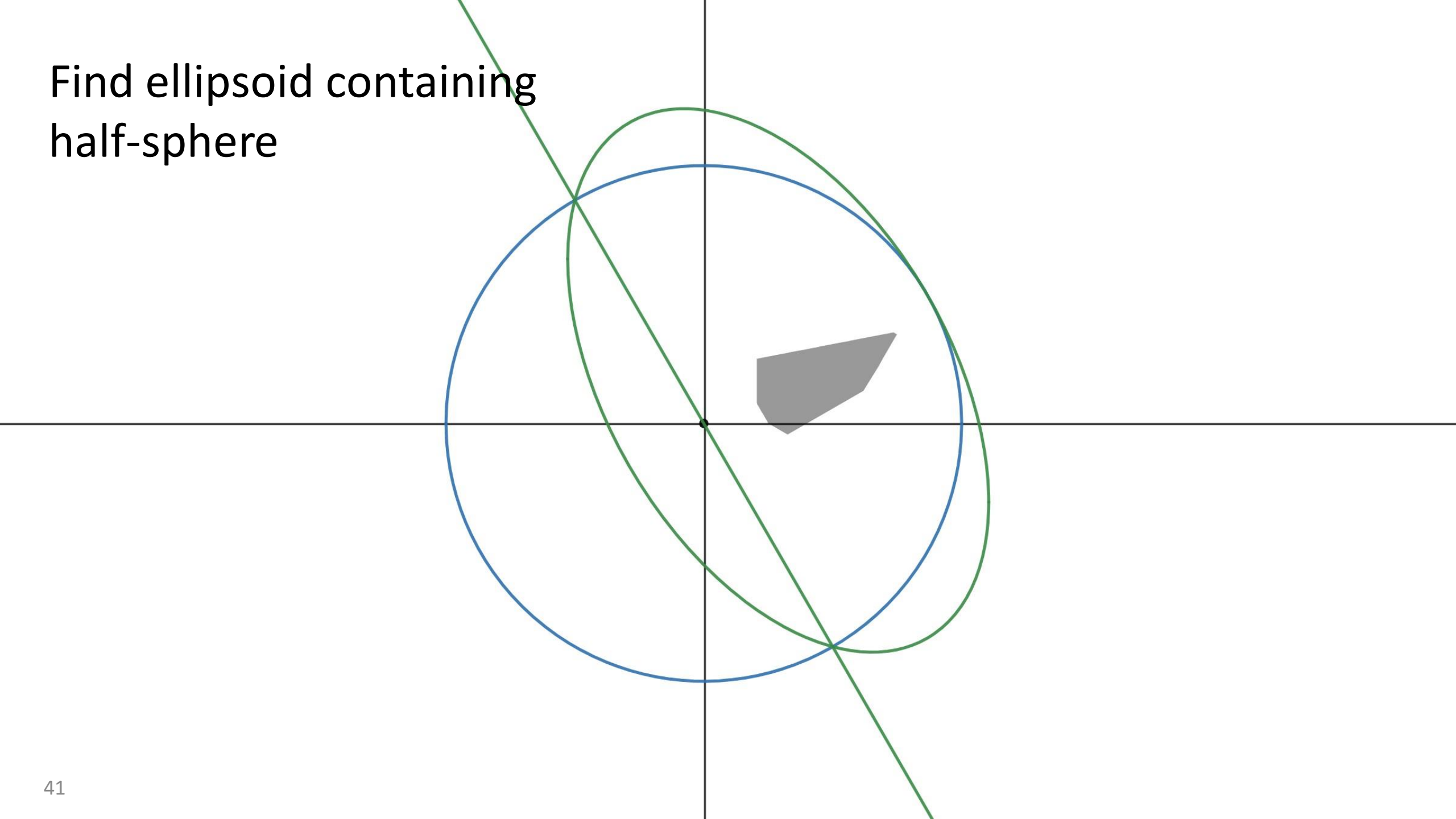
Find violated inequality



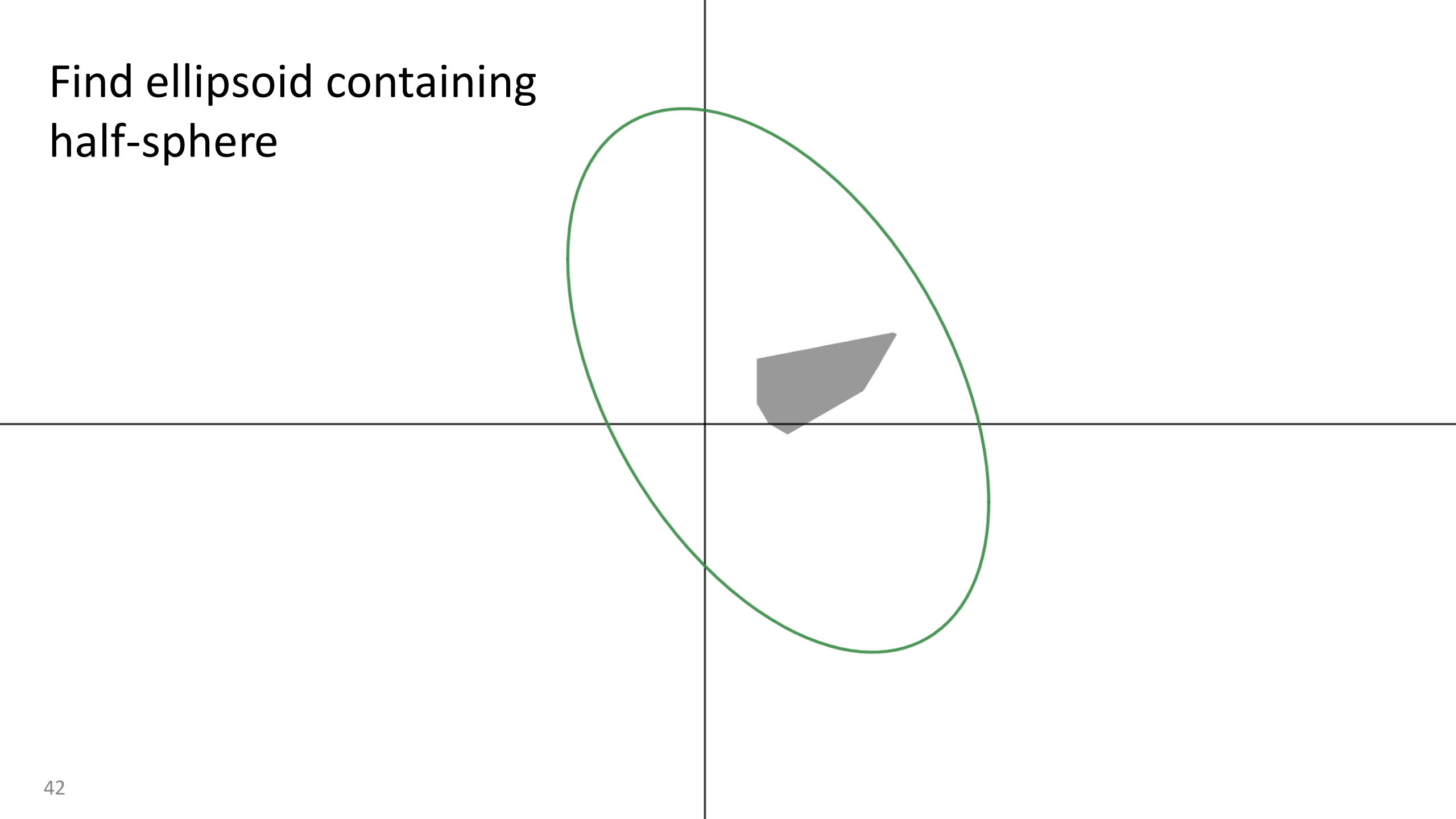
Shift inequality to origin



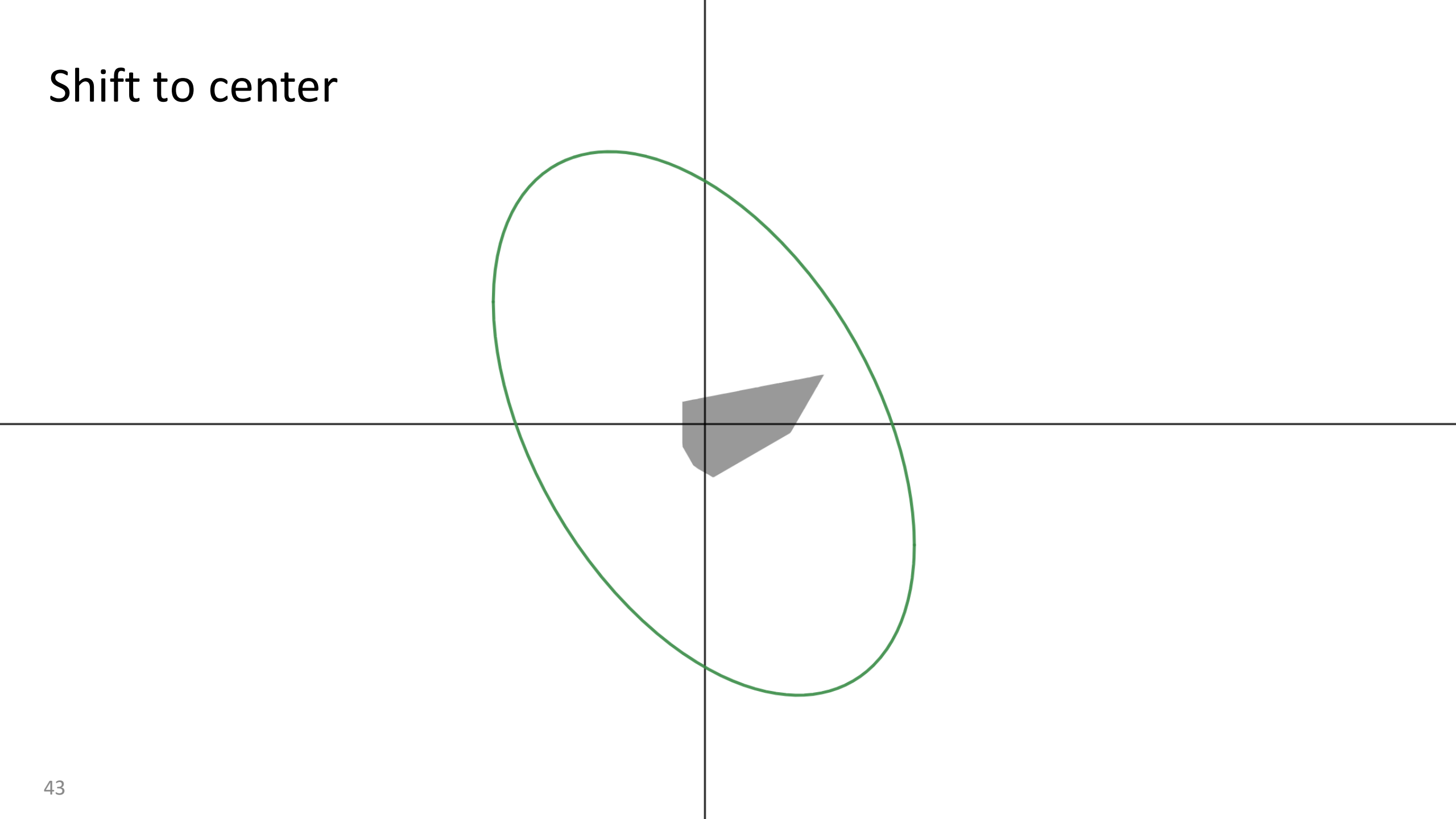
Find ellipsoid containing half-sphere



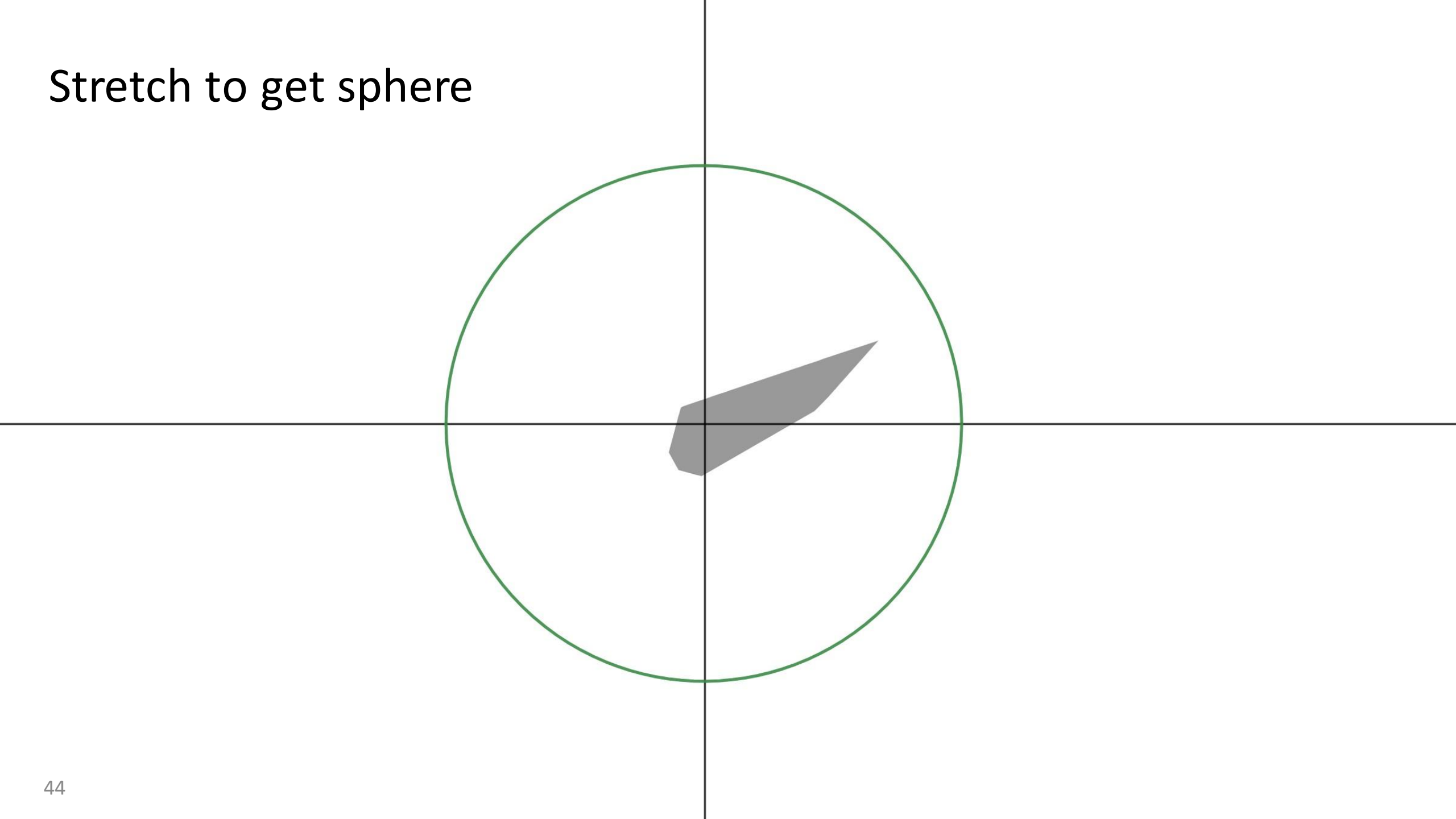
Find ellipsoid containing half-sphere



Shift to center



Stretch to get sphere



Where we have been…

Problems and major solution paradigms

- Stable Matching
- Graph Traversal
- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
- Network Flow
- Linear Programming
  - Focus on encoding as LPs as opposed to how to solve them.
- NP-completeness
- Approximation algorithms & other ways of side-stepping NP-hardness

How to use these ideas

- 1st : See if your problem is a special case/close to/reminds you of one of the problems we have considered in the class
- 2nd : Try these:
  - Graph traversal
  - Greedy algorithms
    - Be **skeptical**! Your first greedy idea is probably wrong; maybe all greedy approaches are wrong. Proving correctness is critical.
- 3rd: Try to solve it recursively w/ smaller subproblems of the same type
  - If subproblems are a constant ratio smaller: Divide and Conquer
  - If same subproblems show up repeatedly: Dynamic Programming
    - See how the pattern of recursion/subproblems matches example patterns you already know. Maybe try a new one.

How to use these ideas

- 4th: See if you can express your problem as a Flow/Cut/Matching.
- 5th : Try Linear Programming
- 6th : Maybe your problem is NP-hard. Check out lists of NP-hard problems to see if yours is there, or try to show it directly.
- 7th : If your problem is NP-hard, try to side-step that hardness.

There are other methods we have barely touched on and

getting some polynomial-time algorithm isn't the end of the story…

… we have barely touched on the subject.

What comes next?

- CSE 431 (complexity theory)
  - What can't you do? (in polynomial time, at all, or in limited memory)
- CSE 422 Toolkit for modern algorithms
  - Algorithmic principles behind modern stats and ML
- CSE 426 [490C] Cryptography
  - a mix of math, algorithms, and complexity
- CSE 521 and 525
  - Graduate level courses in algorithms and randomized algorithms
- Look ahead! These courses usually only run once-per-year.