STAT 4830: Numerical optimization for data science and ML

Lecture 3: Linear Regression - Gradient Descent

Professor Damek Davis

The Memory Wall

Consider genomic prediction: 1000 patients, 100,000 genetic markers

```python
n_samples = 1000
n_markers = 100_000
memory_needed = (n_markers * n_markers * 8) / (1024**3)  # in GB
print(f"Memory needed for X^TX: {memory_needed:.1f} GB")  # 80.0 GB
```

Just forming $X^\top X$ would exceed most workstations' memory!

Even Worse: Medical Imaging

MRI reconstruction with 256^3 voxels:

- Matrix size: $256^3 \times 256^3$
- Memory for $X^\top X$: 2.2 petabytes
- That's 0.2% of world's total data center storage in 2023!

These aren't edge cases - they're routine analysis tasks.

Why Direct Methods Fail

Direct methods solve normal equations $X^\top X w = X^\top y$:

```
# Direct method (fails for large p)
XtX = X.T @ X                 # Form p × p matrix
Xty = X.T @ y                # Form p × 1 vector
w = solve(XtX, Xty)          # Solve p × p system
```

Costs:

1. Forming $X^\top X$: $O(np^2)$ operations, $O(p^2)$ memory
2. Forming $X^\top y$: $O(np)$ operations, $O(p)$ memory
3. Solving system: $O(p^3)$ operations

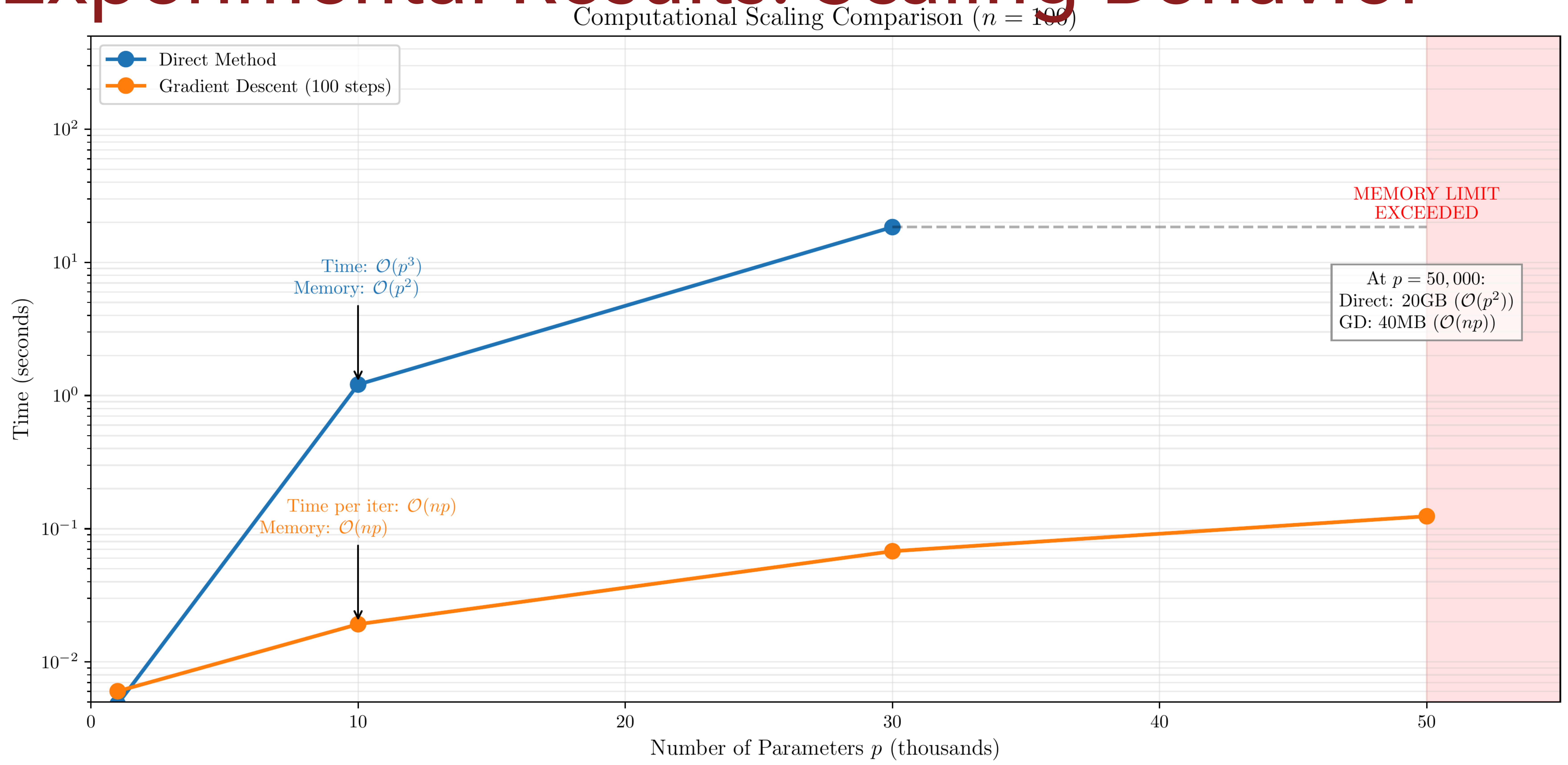
Experimental Results: Memory Wall

Results on MacBook M1 Pro (64GB RAM):

```
Size (p)    Memory for X^TX    Time       Status
1,000       8MB                 0.005s    Fast, fits in fast memory
5,000       200MB              0.182s    Fits in RAM
20,000      3.2GB              5.209s    RAM stressed
50,000      20GB               FAILS     Out of memory
```

Memory becomes bottleneck before computation time!

Experimental Results: Scaling Behavior



The pattern is clear: memory becomes the bottleneck long before computation time.

A Memory-Efficient Alternative

One memory-efficient alternative is gradient descent:

```
# This forms a huge p × p matrix (bad)
XtX = X.T @ X            # Need O(p²) memory
result = XtX @ w         # Matrix-vector product
# Gradient descent uses operations like these:
Xw = X @ w              # Need O(p) memory
result = X.T @ Xw       # Another O(p) operation
```

Both compute $(X^\top X)w$, but gradient descent:

- Never forms the $p \times p$ matrix
- Uses $O(np)$ operations (same as first approach)
- Only needs $O(p)$ extra memory for vectors

The Algorithm

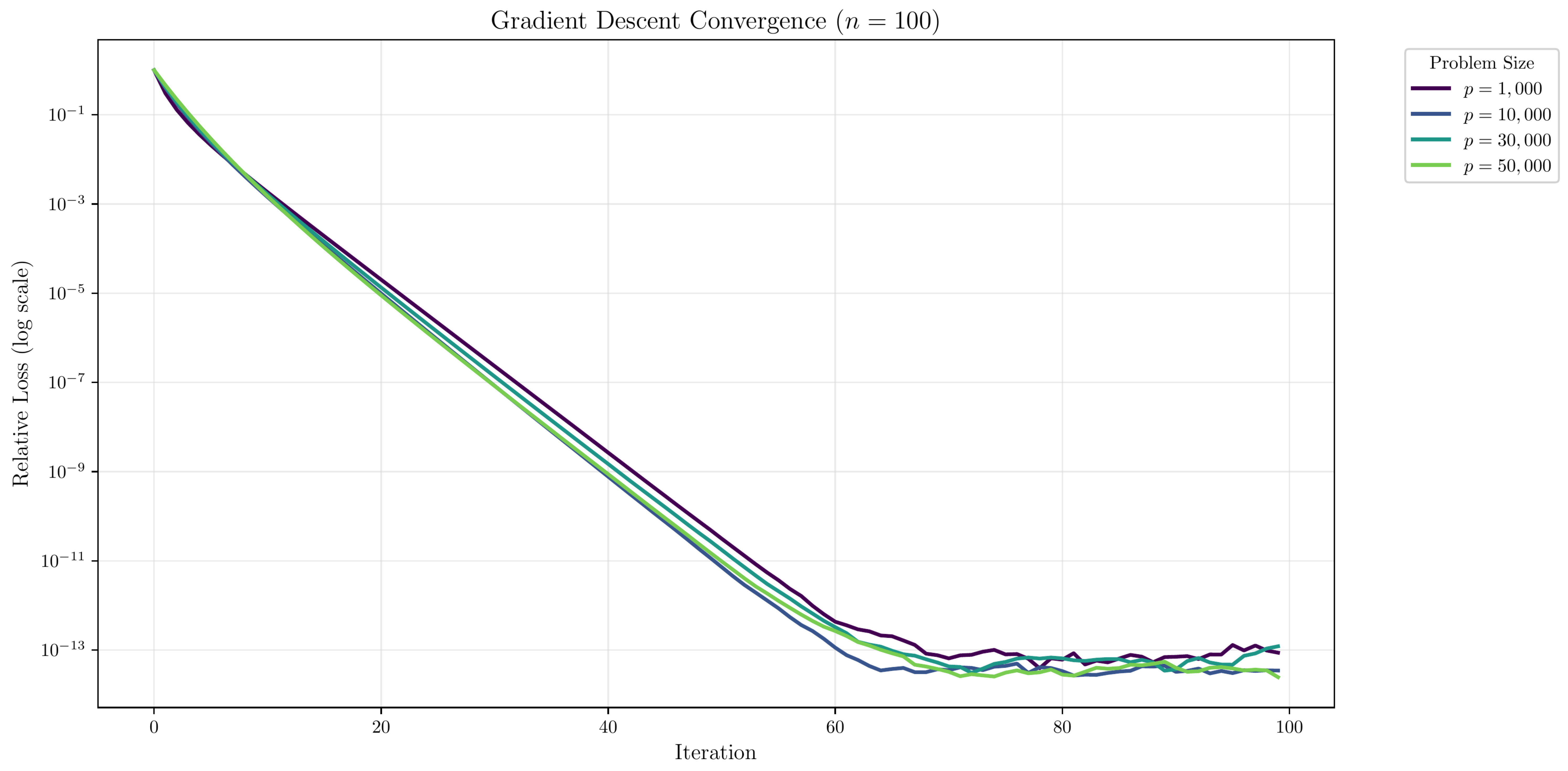
```
# Gradient descent with matrix-vector products
w = torch.zeros(p)            # Initial guess
for k in range(max_iters):
    Xw = X @ w               # Forward pass: O(np)
    grad = X.T @ (Xw - y)    # Backward pass: O(np)
    w -= step_size * grad    # Update: O(p)
```

The memory efficiency comes from iteratively updating our solution:

1. Start with an initial guess (even all zeros)
2. Compute the gradient using matrix-vector products
3. Take a small step in that direction
4. Repeat until convergence

Convergence Behavior: The Pattern

Our experiments with random matrices reveal a fascinating pattern:



Convergence Behavior: Key Insights

Linear Convergence

Error decreases exponentially, appearing as a straight line on log scale. This predictable rate of improvement lets us estimate progress.

Precision vs Time

Each doubling of iterations improves precision by ~10^4. This consistent behavior lets us plan computational resources.

Practical Impact

- 20 iterations: ~10^{-5} relative error
- 40 iterations: ~10^{-9} relative error
- 60 iterations: ~10^{-13} relative error

How to think about gradient descent

- Compute direction of steepest descent (how to compute?)
- Move in that direction (how far?)
- Repeat until convergence (how to measure?)

We'll answer these questions today.

The Least Squares Landscape

Our objective measures squared prediction error:

$$f(w) = \frac{1}{2}\|Xw - y\|_2^2 = \frac{1}{2}(Xw - y)^\top(Xw - y)$$

Expanding reveals the quadratic structure:

$$f(w) = \frac{1}{2}(w^\top X^\top Xw - 2y^\top Xw + y^\top y)$$

Each term has meaning:

- $w^\top X^\top Xw = \|Xw\|^2$: size of predictions
- $2y^\top Xw$: alignment with truth
- $y^\top y$: scale of target values

Computing the Gradient

The gradient has the form:

$$\frac{\partial f}{\partial w_j} = \sum_{i=1}^{n} (x_i^\top w - y_i) x_{ij}$$
$$\nabla f(w) = X^\top (Xw - y) = X^\top X w - X^\top y$$

This tells us:

- $Xw - y$ is prediction error in output space
- $X^\top$ projects error back to parameter space
- Direction tells us how to adjust each parameter

Finding the Direction of Steepest Descent

For our quadratic function, we can compute the exact change:

$$\begin{aligned} f(w+\epsilon v) &= \frac{1}{2}\|X(w+\epsilon v)-y\|_2^2 \\ &= f(w)+\epsilon(Xw-y)^\top Xv+\frac{\epsilon^2}{2}v^\top X^\top Xv \\ &= f(w)+\epsilon\nabla f(w)^\top v+\frac{\epsilon^2}{2}v^\top X^\top Xv \end{aligned}$$

For small ϵ, the ϵ dominates ϵ^2.

Linear Approximation

IDEA: At any point w, we can approximate f using its gradient:

$$f(w + \epsilon v) \approx f(w) + \epsilon \nabla f(w)^\top v$$

This so-called **first-order approximation**:

- Determines initial rate of descent
- Guides stepsize selection
- Explains convergence behavior

The Optimization Problem

At any point w, we want the direction v that decreases the first order approximation of f most rapidly:

$$\text{minimize} \quad \nabla f(w)^\top v \quad \text{subject to} \quad \|v\| = 1$$

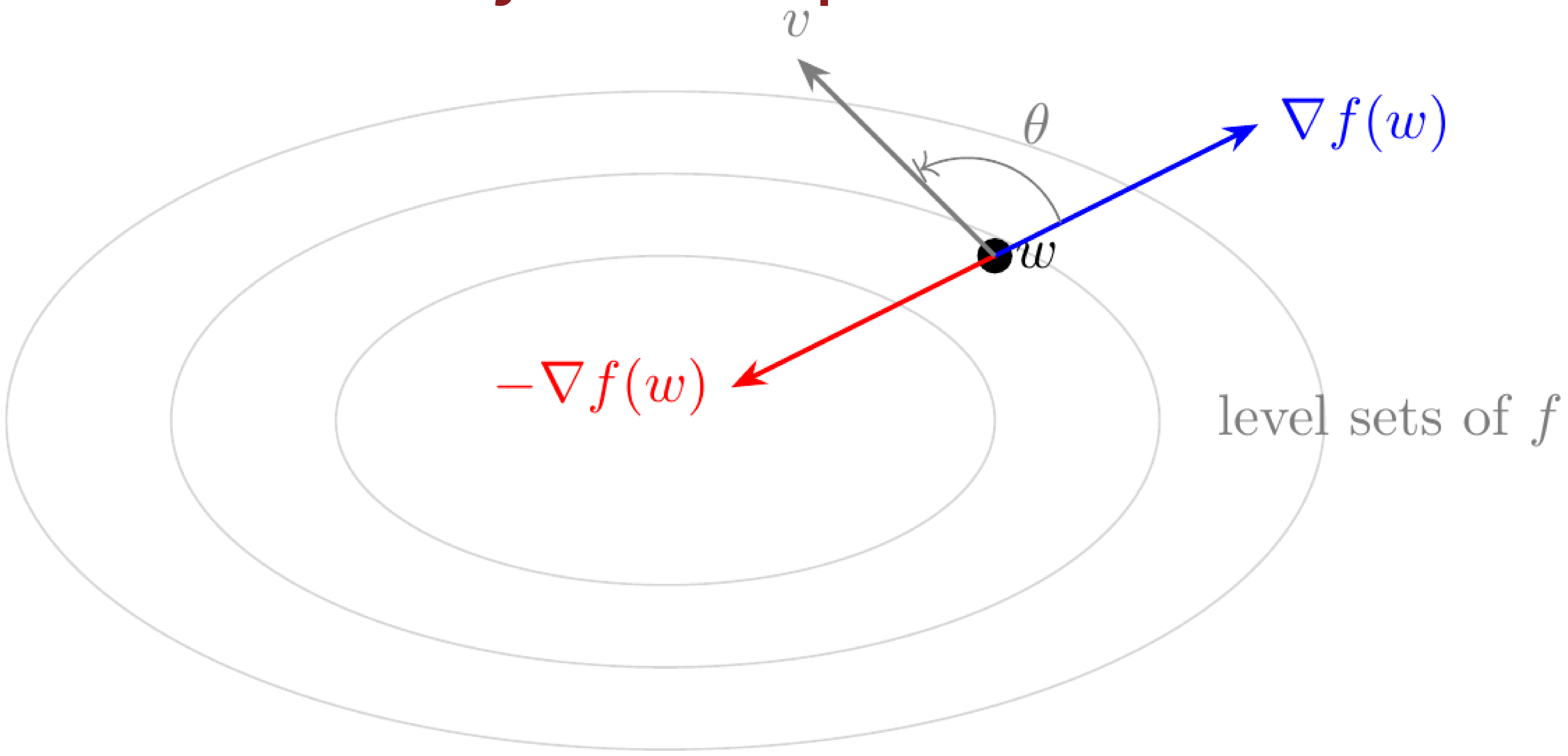
The solution is:

$$v_\star = -\frac{\nabla f(w)}{\|\nabla f(w)\|}$$

Indeed, by Cauchy-Schwarz inequality:

$$|\nabla f(w)^\top v| \leq \|\nabla f(w)\| \|v\| = \|\nabla f(w)\|$$

The Geometry of Steepest Descent



What if the Gradient is zero?

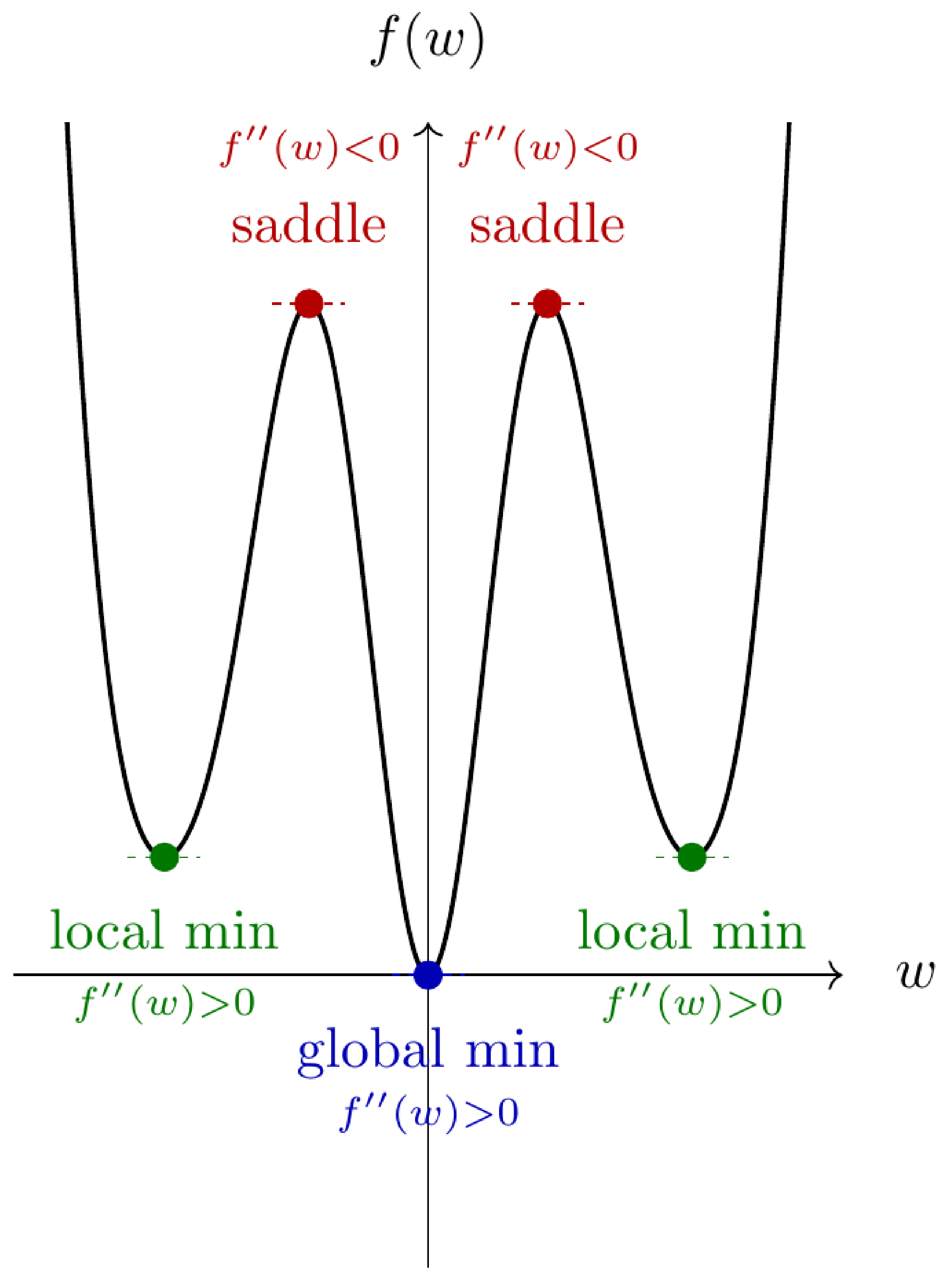
When $\nabla f(w) = 0$, we've found a critical point:

- **Local Minimum**: All directions curve upward
- **Local Maximum**: All directions curve downward
- **Saddle Point**: Some up, some down

For least squares: All critical points are global minima!

- This is due to **convexity** -- a property we'll study later.

What if the Gradient is zero?



The Algorithm: Overview

At each step:

1. Start at our current point w_k
2. Compute the gradient $g_k = X^\top X w_k - X^\top y$
3. Move in the negative gradient direction: $w_{k+1} = w_k - \alpha_k g_k$
4. Repeat until the gradient becomes small

Three key factors determine success:

- Stepsize selection
- Problem conditioning
- Initial guess quality

The Algorithm: Implementation

```
# Gradient descent with matrix-vector products
w = torch.zeros(p)              # Initial guess
for k in range(max_iters):
    Xw = X @ w                 # Forward pass: O(np)
    grad = X.T @ (Xw - y)      # Backward pass: O(np)
    w -= step_size * grad      # Update: O(p)
```

For least squares, starting at zero is reasonable:

- Gives zero predictions - a natural baseline
- Will eventually find minimum (thanks to convexity)
- Good initial guess reduces iterations needed

Stepsize Selection: The Theory

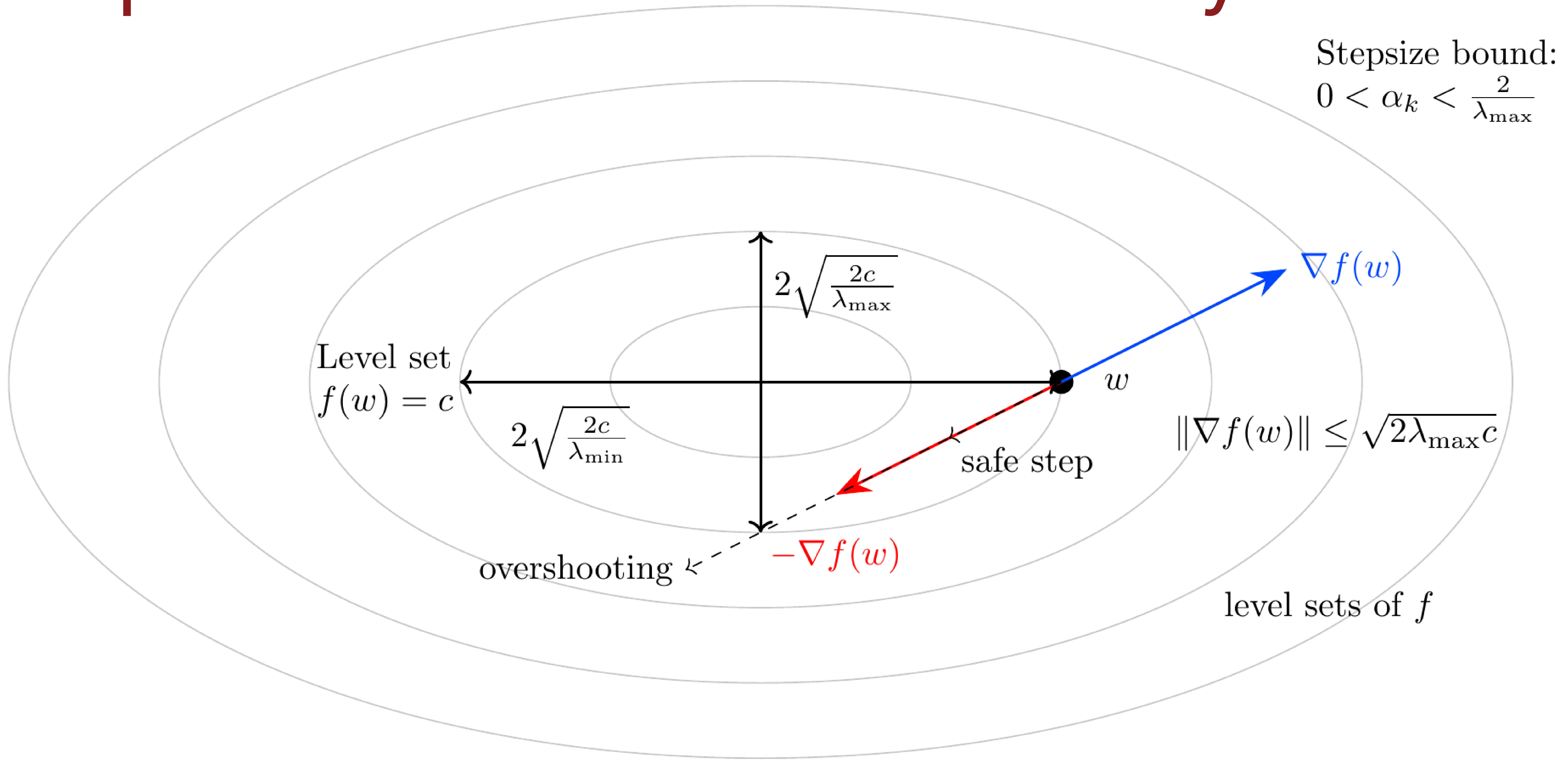
Convergence is guaranteed when:

$$0 < \alpha_k < \frac{2}{\lambda_{\max}(X^\top X)}$$

Why this bound?

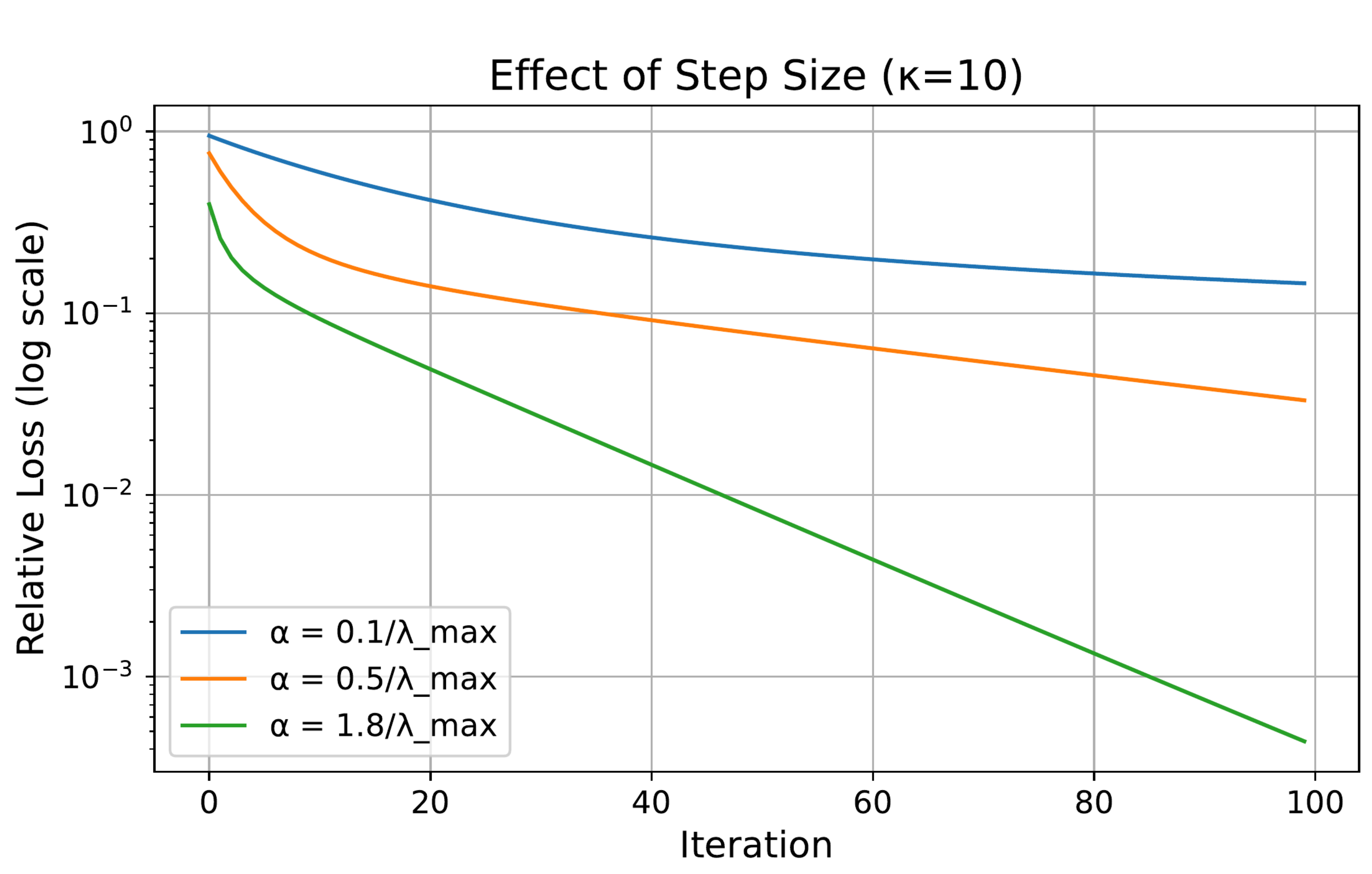
- Level sets become very narrow in some directions
- Width determined by eigenvalues of $X^\top X$
- Too large a step overshoots the minimum

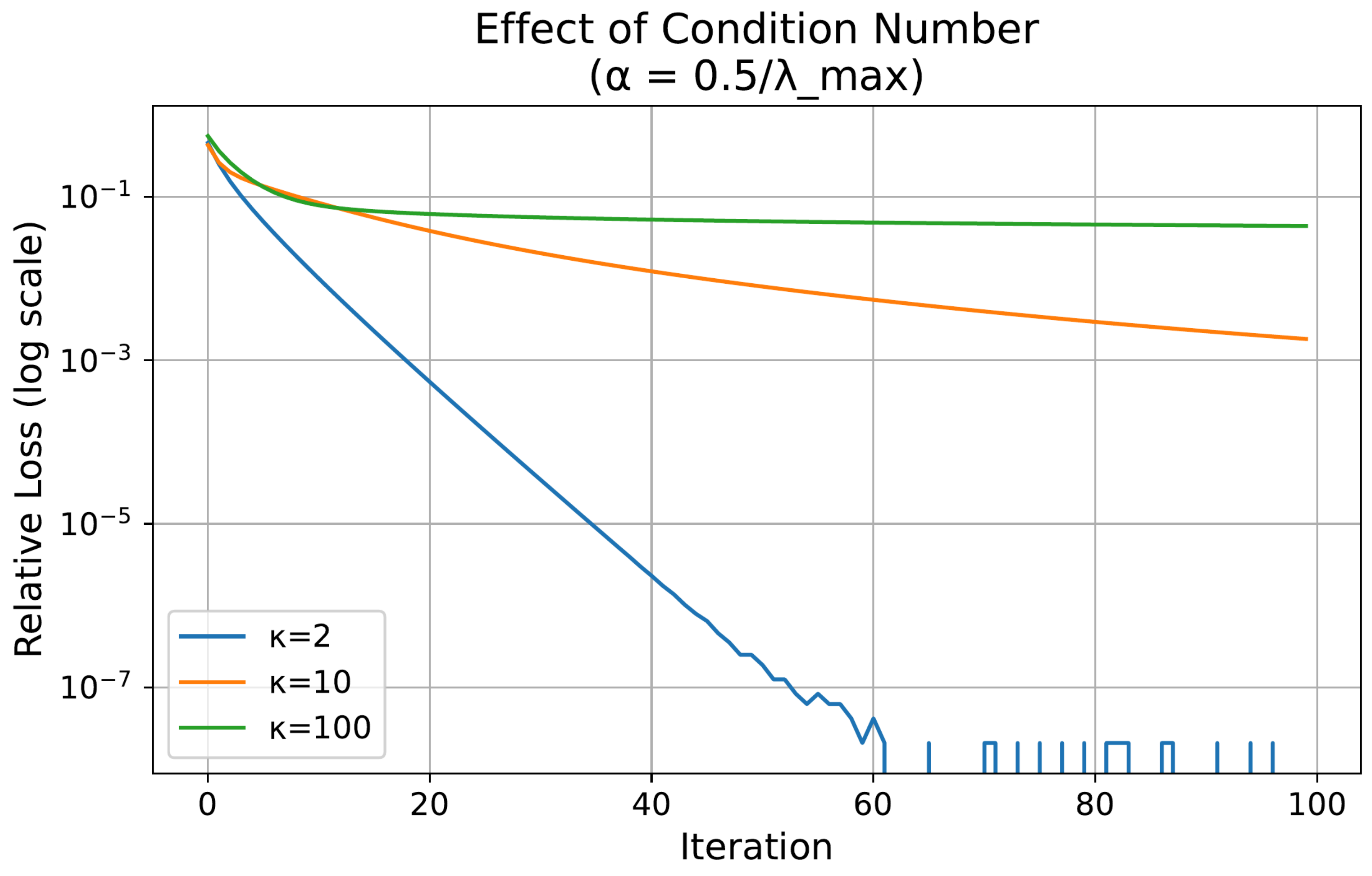
Stepsize Selection: The Geometry



Stepsize bound:
$0 < \alpha_k < \frac{2}{\lambda_{\max}}$
$\nabla f(w)$
$2\sqrt{\frac{2c}{\lambda_{\max}}}$
Level set
$f(w) = c$
w
$2\sqrt{\frac{2c}{\lambda_{\min}}}$
$\|\nabla f(w)\| \leq \sqrt{2\lambda_{\max} c}$
safe step
$-\nabla f(w)$
overshooting
level sets of f

Convergence Speed vs Condition Number





Left: Effect of stepsize ($\kappa = 10$)

Right: Effect of condition number (fixed stepsize)

Effect of Condition Number: Analysis

The path to the minimum depends on problem conditioning:

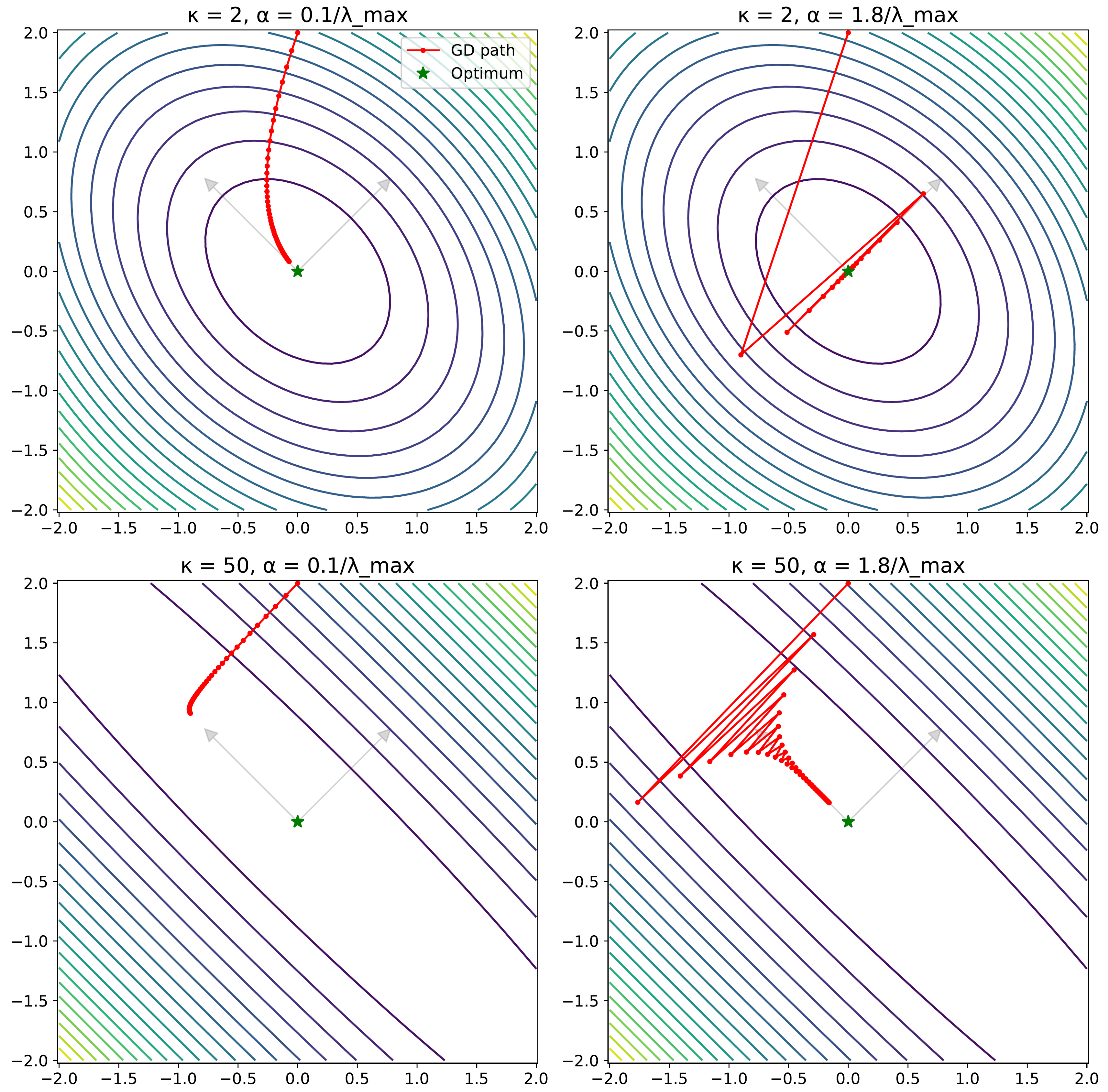
Well-Conditioned ($\kappa = 2$)

- Direct path to minimum
- Fast, steady progress
- Efficient use of computation

Poorly-Conditioned ($\kappa = 50$)

- Zigzag path to minimum
- Slow overall progress
- Many wasted steps

Effect of Condition Number



Limitations and Next Steps

Gradient descent also has limitations:

- For large n: Computing full gradient expensive
- For large p: Memory still scales with problem size
- Poor conditioning: Slow convergence

Solutions we'll cover later:

1. Stochastic methods for large n
2. Coordinate descent for large p
3. Momentum and adaptive methods for conditioning

Summary

1. **Memory Wall**: Direct methods fail for large problems
2. **Gradient Descent**: Memory-efficient iterative solution
3. **Convergence**: Linear rate with predictable behavior
4. **Geometry**: Follows steepest descent direction
5. **Implementation**: Simple, scalable algorithm
6. **Limitations**: Sets up need for advanced methods

Next lecture: problems beyond least squares.