ORF522 – Linear and Nonlinear Optimization

6. Numerical linear algebra and simplex implementation

Bartolomeo Stellato — Fall 2024

Recap

An iteration of the simplex method

Initialization

- a basic feasible solution x
- a basis matrix $A_B = \begin{bmatrix} A_{B(1)} & \ldots, A_{B(m)} \end{bmatrix}$

Iteration steps

1. Compute the reduced costs $\bar{c}$
    - Solve $A_B^T p = c_B$
    - $\bar{c} = c - A^T p$
2. If $\bar{c} \geq 0$, x **optimal. break**
3. Choose j such that $\bar{c}_j < 0$
4. Compute search direction d with $d_j = 1$ and $A_B d_B = -A_j$
5. If $d_B \geq 0$, the problem is **unbounded** and the optimal value is $-\infty$. **break**
6. Compute step length $\theta^\star = \min_{\{i \in B | d_i < 0\}} \left(-\frac{x_i}{d_i} \right)$
7. Define y such that $y = x + \theta^\star d$
8. Get new basis $\bar{B}$ (i exits and j enters)

Today's agenda

[Chapter 3, LO] [Chapter 13, NO] [Chapter 8, LP]

- Numerical linear algebra
- Realistic simplex implementation
- Example
- Empirical complexity

Numerical linear algebra

Deeper look at complexity

Flop count

floating-point operations: one addition, subtraction, multiplication, division

Estimate complexity of an algorithm

- Express number of flops as a **function of problem dimensions**
- Simplify and keep only leading terms

Remarks

- Not accurate in modern computers (multicore, GPU, etc.)
- Still rough and widely-used estimate of complexity

Complexity

Basic examples

Vector operations ($x, y \in \mathbf{R}^n$)

- Inner product $x^T y$: $2n - 1$ flops
- Sum $x + y$ or scalar multiplication αx: n flops

Matrix-vector product ($y = Ax$ with $A \in \mathbf{R}^{m \times n}$)

- $m(2n - 1)$ flops
- $2N$ if A is sparse with N nonzero elements

Matrix-matrix product ($C = AB$ with $A \in \mathbf{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$)

- $pm(2n - 1)$ flops
- Less if A and/or B are sparse

How do we solve linear systems in practice?

$$Ax = b$$

Idea

- compute A^{-1}
- multiply $A^{-1}b$

Example

5000×5000 matrix A and a 5000-vector b

- Solve by computing A^{-1}
- Solve with `numpy.linalg.solve`

What's happening inside?

Complexity

Solving linear system

Execution time (cost) of solving $Ax = b$ with $A \in \mathbf{R}^{n \times n}$

General case $O(n^3)$

Much less if A structured (sparse, banded, Toepliz, etc.)

You (almost) **never compute** A^{-1} explicitly!

- Numerically unstable (divisions)
- You lose structure

Easy linear systems

Diagonal matrix

$$\begin{bmatrix} A_{11} & & \\ & \ddots & \\ & & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \longrightarrow \begin{aligned} A_{11}x_1 &= b_1 \\ A_{22}x_2 &= b_2 \\ &\vdots \\ A_{nn}x_n &= b_n \end{aligned}$$

Solution

$$x = A^{-1}b = (b_1/A_{11}, \dots, b_n/A_{nn})$$

Complexity

n flops

Easy linear systems

Lower triangular matrix

$$\begin{bmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & & \ddots & \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \longrightarrow \begin{aligned} A_{11}x_1 &= b_1 \\ A_{21}x_1 + A_{22}x_2 &= b_2 \\ &\vdots \\ A_{n1}x_1 + A_{n2}x_2 + \dots A_{nn}x_n &= b_n \end{aligned}$$

Solution: "forward substitution"

- First equation: $x_1 = b_1/A_{11}$
- Second equation: $x_2 = (b_2 - A_{21}x_1)/A_{22}$
- Repeat to get $x_3, \dots, x_n$

Complexity

- First equation: 1 flop (division)
- Second equation: 3 flops
- ith step needs $2i - 1$ flops

$1 + 3 + \cdots + (2n - 1) = n^2$ flops

Easy linear systems

Permutation matrices

$\pi = (\pi_1, \ldots, \pi_n)$ is a permutation of $(1, 2, \ldots, n)$

A $n \times n$ permutation matrix P,
permutes the vector x

$$Px = (x_{\pi_1}, \ldots, x_{\pi_n})$$

Properties

- $P_{ij} = \begin{cases} 1 & j = \pi_i \\ 0 & \text{otherwise} \end{cases}$
- $P^{-1} = P^T$ (inverse permutation)

Complexity

Solve $Px = b$: 0 flops (no operations)

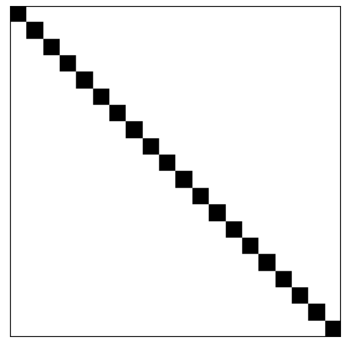
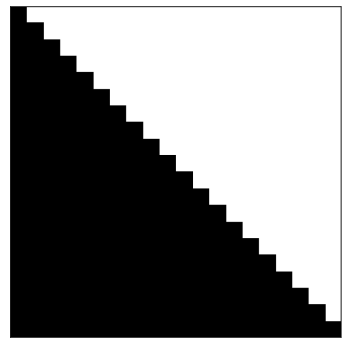
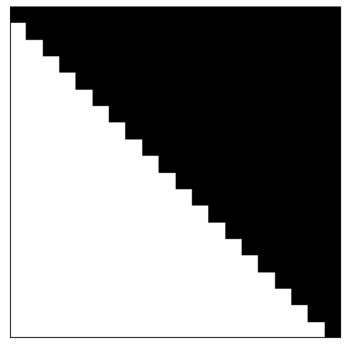
example

$$\pi = (2, 3, 1)$$

$$\overset{P}{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ x_1 \end{bmatrix}$$

$$\overset{P^{-1}}{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}} \begin{bmatrix} x_2 \\ x_3 \\ x_1 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

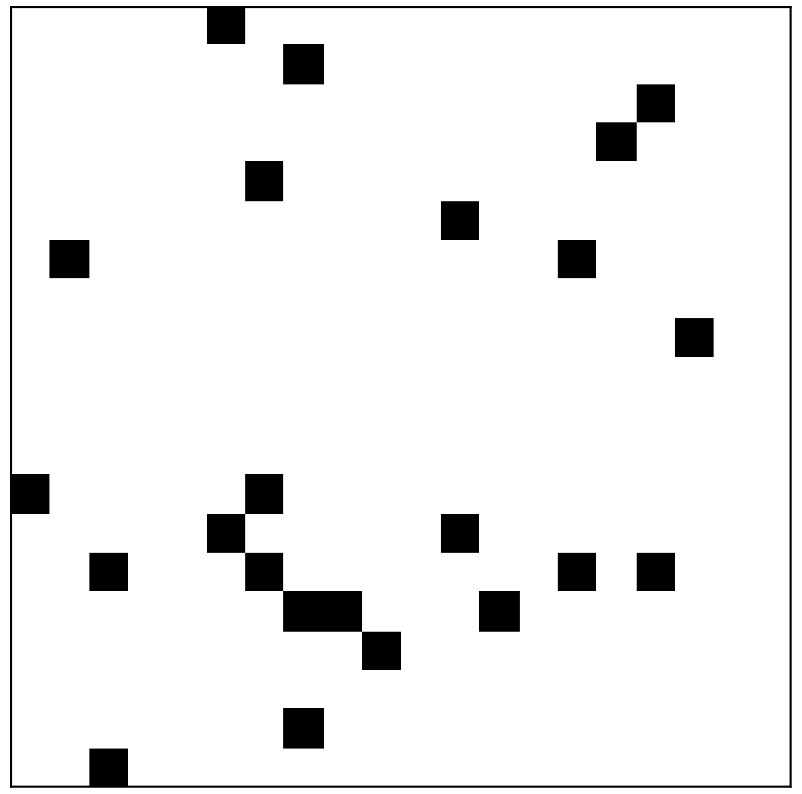
Summary of easy linear systems

		method	flops
	**diagonal** $A = \mathbf{diag}(a_1, \ldots, a_n)$	$x_i = b_i / a_i$	n
	**lower triangular** $A_{ij} = 0$ for $i < j$	forward substitution	n^2
	**upper triangular** $A_{ij} = 0$ for $i > j$	backward substitution	n^2
	permutation $P_{ij} = 1$ if $j = \pi_i$ else 0	inverse permutation	0

Sparse matrices

Most **real-world problems** are sparse

A matrix A is **sparse** if the majority of its elements is 0



typically $< 15\%$ nonzeros

Efficient representations

- Triplet format: (i, j, x_{ij})
- Compressed Sparse Column format: (i, x_{ij}) and p_j
- Compressed Sparse Row format: (j, x_{ij}) and p_i

How do we solve linear systems in practice?

$$Ax = b$$

Any idea?

We know how to solve special ones

Let’s use that!

The factor-solve method for solving $Ax = b$

1. **Factor** A as a product of simple matrices:

$$A = A_1 A_2 \cdots A_k, \longrightarrow A_1 A_2, \dots A_k x = b$$

(A_i diagonal, upper/lower triangular, permutation, etc)

2. **Compute** $x = A^{-1} b = A_k^{-1} \cdots A_1^{-1} b$ by solving k "easy" systems

$$\longrightarrow \begin{aligned} A_1 x_1 &= b \\ A_2 x_2 &= x_1 \\ &\vdots \\ A_k x &= x_{k-1} \end{aligned}$$

Note: step 2 is much cheaper than step 1

Multiple right-hand sides

You now have factored A and you want to solve d linear systems with different righ-hand side m-vectors b_i

$$Ax = b_1 \qquad Ax = b_2 \qquad \ldots \quad Ax = b_d$$

Factorization-caching procedure

1. Factor $A = A_1, \ldots, A_k$ **only once** (expensive)
2. Solve all linear systems using **the same factorization** (cheap)

Solve many "at the price of one"

(Sparse) LU factorization

Every nonsingular matrix A can be factored as

$$A = P_r L U P_c \longrightarrow P_r^T A P_c^T = LU$$

P_r, P_c permutation, L lower triangular, U upper triangular

Permutations

- Reorder rows P_r and columns P_c of A to (heuristically) get **sparser** L, U
- P_r, P_c depend on sparsity pattern and values of A

Cost

- If A dense, typically $O(n^3)$ but usually much less
- It depends on the number of nonzeros in A, sparsity pattern, etc.

(Sparse) LU solution

$$Ax = b, \quad \Rightarrow \quad P_r L U P_c x = b$$

Iterations

1. *Permutation*: Solve $P_r z_1 = b$ (0 flops)
2. *Forward substitution*: Solve $L z_2 = z_1$ (n^2 flops)
3. *Backward substitution*: Solve $U z_3 = z_2$ (n^2 flops)
4. *Permutation*: Solve $P_c x = z_3$ (0 flops)

Cost

Factor + Solve $\sim O(n^3)$

Just solve (prefactored) $\sim O(n^2)$

(Sparse) Cholesky factorization

Every positive definite matrix A can be factored as

$$A = PLL^T P^T \longrightarrow P^T AP = LL^T$$

P permutation, L lower triangular

Permutations

- Reorder rows/cols of A with P to (heuristically) get **sparser** L
- P depends only on sparsity pattern of A (unlike LU factorization)
- If A is dense, we can set $P = I$

Cost

- If A dense, typically $O(n^3)$ but usually much less
- It depends on the number of nonzeros in A, sparsity pattern, etc.
- Typically 50% faster than LU (need to find only one matrix)

(Sparse) Cholesky solution

$$Ax = b, \quad \Rightarrow \quad PLL^T P^T x = b$$

Iterations

1. *Permutation*: Solve $Pz_1 = b$ (0 flops)
2. *Forward substitution*: Solve $Lz_2 = z_1$ (n^2 flops)
3. *Backward substitution*: Solve $L^T z_3 = z_2$ (n^2 flops)
4. *Permutation*: Solve $P^T x = z_3$ (0 flops)

Cost

Factor + Solve $\sim O(n^3)$

Just solve (prefactored) $\sim O(n^2)$

“Realistic” simplex implementation

Complexity of a single simplex iteration

1. Compute the reduced costs $\bar{c}$
   - Solve $A_B^T p = c_B$
   - $\bar{c} = c - A^T p$
2. If $\bar{c} \geq 0$, x **optimal. break**
3. Choose j such that $\bar{c}_j < 0$
4. Compute search direction d with $d_j = 1$ and $A_B d_B = -A_j$
5. If $d_B \geq 0$, the problem is **unbounded** and the optimal value is $-\infty$. **break**
6. Compute step length $\theta^\star = \min_{\{i \in B \mid d_i < 0\}} \left(-\frac{x_i}{d_i}\right)$
7. Define y such that $y = x + \theta^\star d$
8. Get new basis $\bar{B}$ (i exits and j enters)

Bottleneck

"same" two linear systems

Linear system solutions

Very similar linear systems

$$A_B^T p = c_B$$
$$A_B d_B = -A_j$$

→

LU **factorization**

$O(n^3)$ flops

$$A_B = P_r L U P_c$$

→

Easy linear systems

$O(n^2)$ flops

$$P_c^T U^T L^T P_r^T p = c_B$$
$$P_r L U P_c d_B = -A_j c_B$$

Factorization is expensive

Do we need to recompute it at every iteration?

Basis update

Index update

- j enters (x_j becomes $\theta^\star$)
- $i = B(\ell)$ exists (x_i becomes 0)

$\longrightarrow$

Basis matrix change

$$A_{\bar{B}} = A_B + (A_j - A_i)e_\ell^T$$

Example

$$A = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$B = \{4, 1, 6\} \quad \rightarrow \quad \bar{B} = \{4, 1, 2\}$$

- 2 enters
- $6 = B(3)$ exists

$$A_{\bar{B}} = \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 2 & 1 \end{bmatrix}}_{A_B} + \underbrace{\begin{bmatrix} 0 & 0 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \end{bmatrix}}_{A_2 e_3^T} - \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{A_6 e_3^T} = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 2 & 1 \\ 0 & 2 & 2 \end{bmatrix}$$

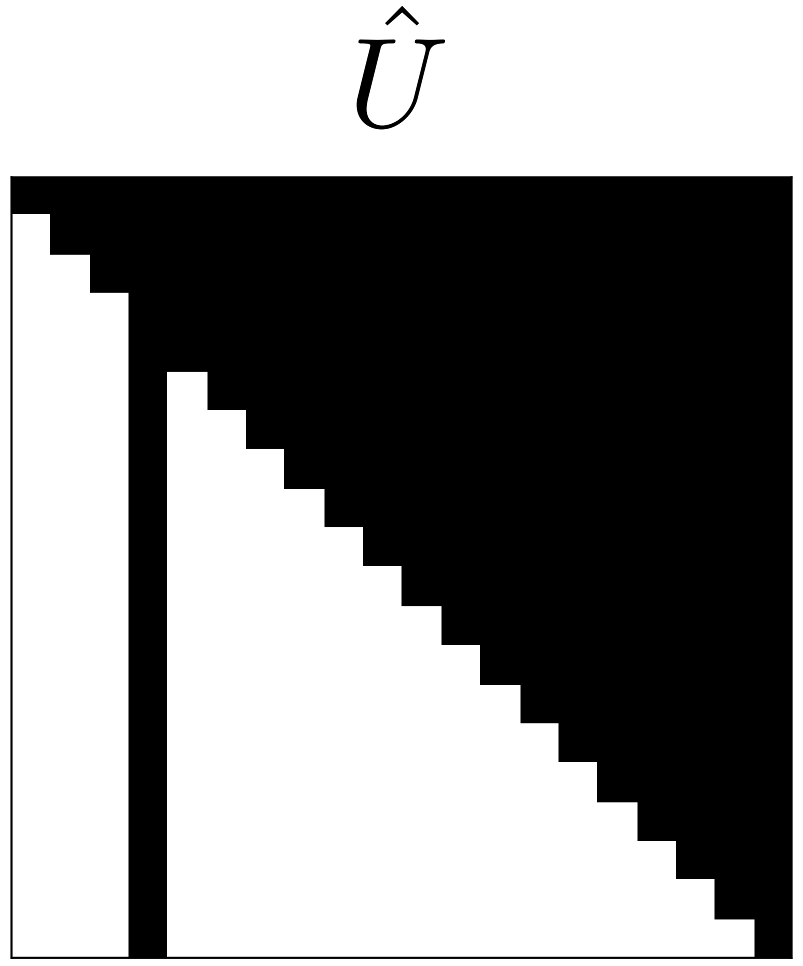
Basis update

Rank-1 update

$$A_{\bar{B}} = A_B + (A_j - A_i)e_\ell^T$$

Forrest-Tomlin update $O(m^2)$

- **Given:** $A_B = LU$
- **Goal:** compute $A_{\bar{B}} = LR\bar{U}$ (same L, lower tri. R, upper tri. $\bar{U}$)
    1. $L^{-1}A_{\bar{B}} = U + (L^{-1}A_j - Ue_\ell)e_\ell^T = \hat{U}$
    2. LU factorization $\hat{U} = R\bar{U}$ via **elimination** ($O(m^2)$)



Remarks

- Implemented in modern sparse solvers
- Accumulates errors (we need to refactor B from scratch once in a while)
- Many more algorithms: Block-LU, Bartels-Golub-Reid, etc.

Realistic (revised) simplex method

Initialization

- a basic feasible solution x
- a basis matrix $A_B = \begin{bmatrix} A_{B(1)} & \dots, A_{B(m)} \end{bmatrix}$

Iteration steps

Per-iteration cost $O(m^2)$

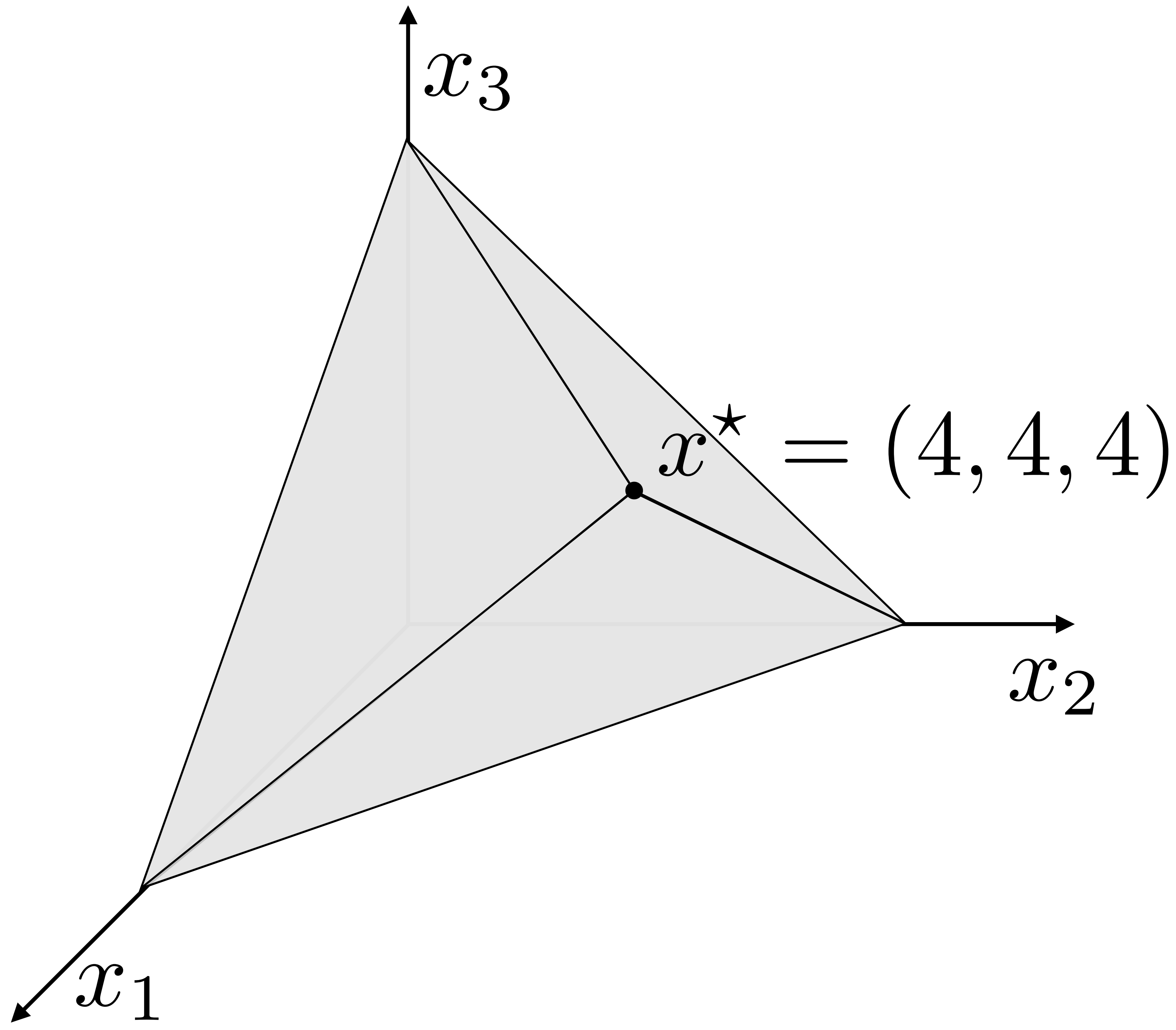
1. Compute the reduced costs $\bar{c}$
   - Solve $A_B^T p = c_B$ ($O(m^2)$)
   - $\bar{c} = c - A^T p$
2. If $\bar{c} \geq 0$, x **optimal. break**
3. Choose j such that $\bar{c}_j < 0$
4. Compute search direction. d with $d_j = 1$ and $A_B d_B = -A_j$ ($O(m^2)$)
5. If $d_B \geq 0$, the problem is **unbounded** and the optimal value is $-\infty$. **break**
6. Compute step length $\theta^\star = \min_{\{i \in B | d_i < 0\}} \left(-\frac{x_i}{d_i} \right)$
7. Define y such that $y = x + \theta^\star d$
8. Get new basis $A_{\bar{B}} = A_B + (A_j - A_i)e_\ell^T$
   rank-1 factor update (i exits and j enters) (($O(m^2)$)

Example

Example

Inequality form

$$
\begin{array}{ll}
\text{minimize} & -10x_1 - 12x_2 - 12x_3 \\
\text{subject to} & x_1 + 2x_2 + 2x_3 \leq 20 \\
& 2x_1 + x_2 + 2x_3 \leq 20 \\
& 2x_1 + 2x_2 + x_3 \leq 20 \\
& x_1, x_2, x_3 \geq 0
\end{array}
$$



Standard form

$$
\begin{array}{ll}
\text{minimize} & -10x_1 - 12x_2 - 12x_3 \\
\text{subject to} & \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 20 \\ 20 \\ 20 \end{bmatrix} \\
& x \geq 0
\end{array}
$$

Example

Start

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

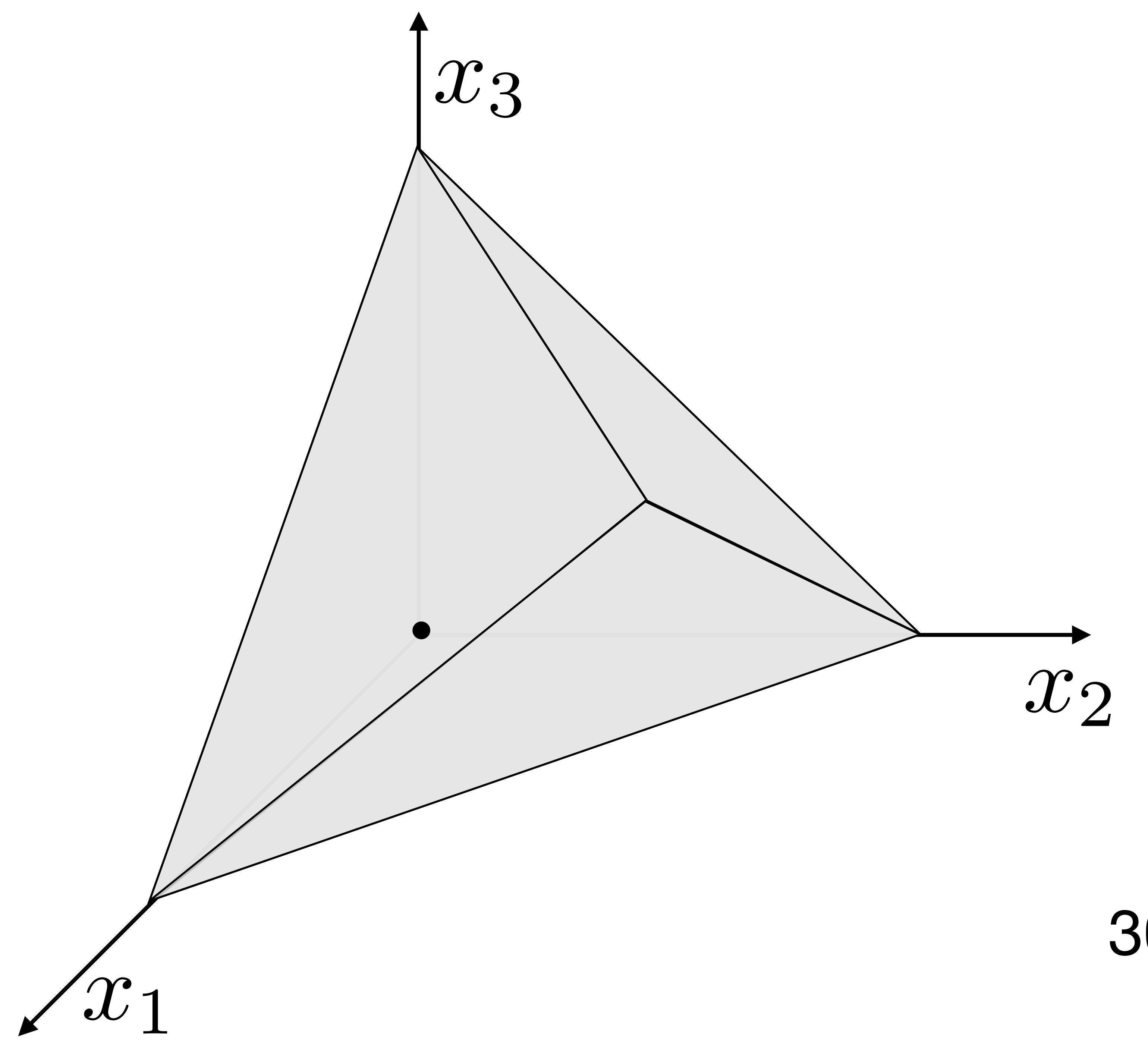
$$c = (-10, -12, -12, 0, 0, 0)$$

$$A = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$b = (20, 20, 20)$$

Initialize

$$x = (0, 0, 0, 20, 20, 20) \qquad A_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Example

Iteration 1

Current point

$x = (0, 0, 0, 20, 20, 20)$
$c^T x = 0$

Basis: $\{4, 5, 6\}$

$$A_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$c = (-10, -12, -12, 0, 0, 0)$

$$A = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$b = (20, 20, 20)$

Reduced costs $\bar{c} = c$

Solve $A_B^T p = c_B \quad \Rightarrow \quad p = c_B = 0$

$\bar{c} = c - A^T p = c$

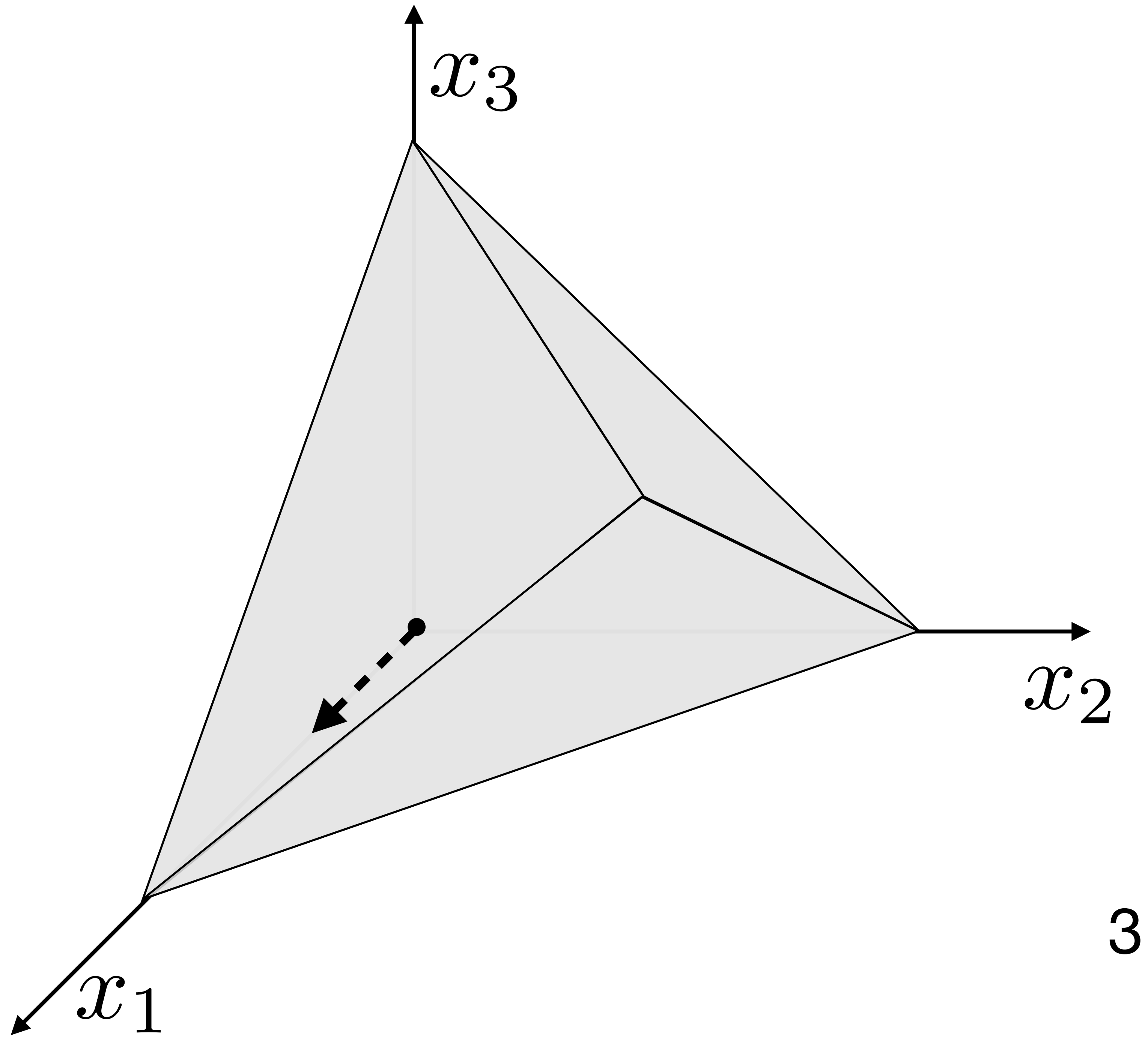
Direction $d = (1, 0, 0, -1, -2, -2), \quad j = 1$

Solve $A_B d_B = -A_j \quad \Rightarrow \quad d_B = (-1, -2, -2)$

Step $\theta^\star = 10, \quad i = 5$

$\theta^\star = \min_{\{i | d_i < 0\}} (-x_i / d_i) = \min\{20, 10, 10\}$

New $x \leftarrow x + \theta^\star d = (10, 0, 0, 10, 0, 0)$



Example
Iteration 2

Current point

$x = (10, 0, 0, 10, 0, 0)$
$c^T x = -100$
Basis: $\{4, 1, 6\}$

$$A_B = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

$c = (-10, -12, -12, 0, 0, 0)$

$$A = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$b = (20, 20, 20)$

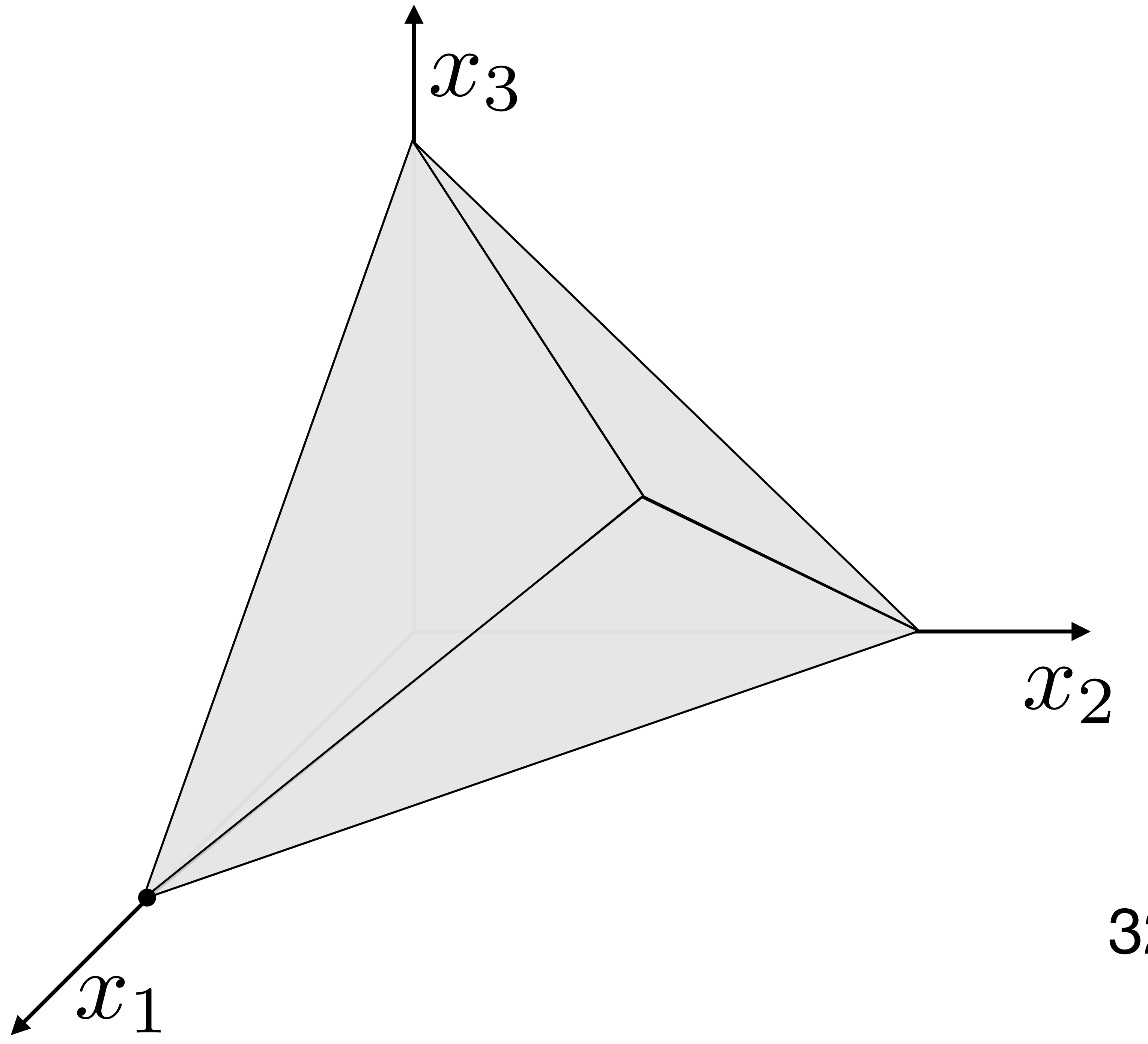
Reduced costs $\bar{c} = (0, -7, -2, 0, 5, 0)$
Solve $A_B^T p = c_B \quad \Rightarrow \quad p = (0, -5, 0)$
$\bar{c} = c - A^T p = (0, -7, -2, 0, 5, 0)$

Direction $d = (-0.5, 1, 0, -1.5, 0, -1), \quad j = 2$
Solve $A_B d_B = -A_j \quad \Rightarrow \quad d_B = (-1.5, -0.5, -1)$

Step $\theta^\star = 0, \quad i = 6$

$$\theta^\star = \min_{\{i | d_i < 0\}} (-x_i / d_i) = \min\{6.66, 20, 0\}$$

New $x \leftarrow x + \theta^\star d = (10, 0, 0, 10, 0, 0)$



Example
Iteration 3

Current point

$x = (10, 0, 0, 10, 0, 0)$
$c^T x = -100$

Basis: $\{4, 1, 2\}$

$$A_B = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 2 & 1 \\ 0 & 2 & 2 \end{bmatrix}$$

$c = (-10, -12, -12, 0, 0, 0)$

$$A = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$b = (20, 20, 20)$

Reduced costs $\bar{c} = (0, 0, -9, 0, -2, 7)$

Solve $A_B^T p = c_B \quad \Rightarrow \quad p = (0, 2, -7)$

$\bar{c} = c - A^T p = (0, 0, -9, 0, -2, 7)$

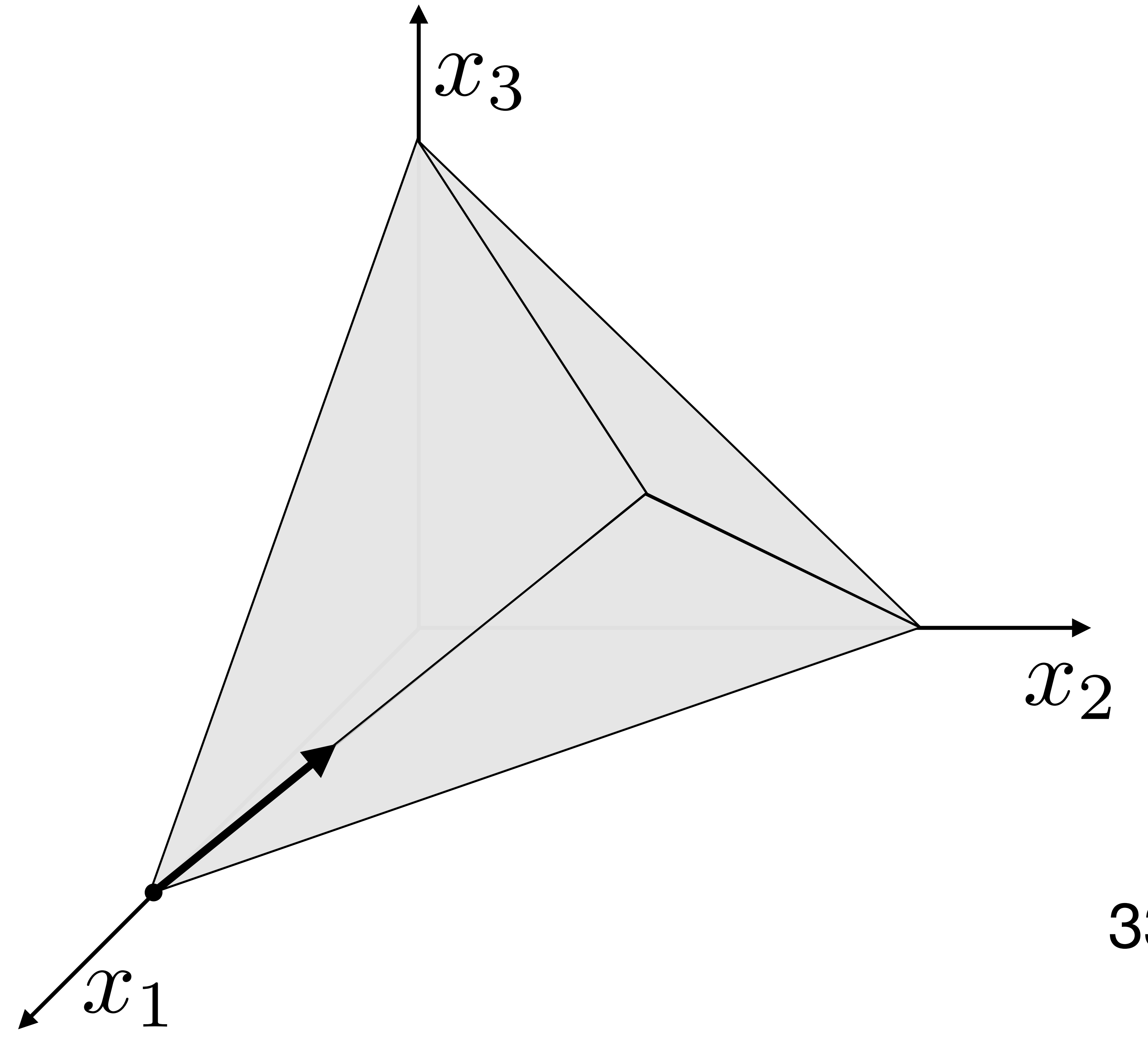
Direction $d = (-1.5, 1, 1, -2.5, 0, 0), \quad j = 3$

Solve $A_B d_B = -A_j \quad \Rightarrow \quad d_B = (-2.5, -1.5, 1)$

Step $\theta^\star = 4, \quad i = 4$

$$\theta^\star = \min_{\{i | d_i < 0\}} (-x_i / d_i) = \min\{4, 6.67\}$$

New $x \leftarrow x + \theta^\star d = (4, 4, 4, 0, 0, 0)$



Example

Iteration 4

Current point

$x = (4, 4, 4, 0, 0, 0)$

$c^T x = -136$

Basis: $\{3, 1, 2\}$

$$A_B = \begin{bmatrix} 2 & 1 & 2 \\ 2 & 2 & 1 \\ 1 & 2 & 2 \end{bmatrix}$$

$c = (-10, -12, -12, 0, 0, 0)$

$$A = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$b = (20, 20, 20)$

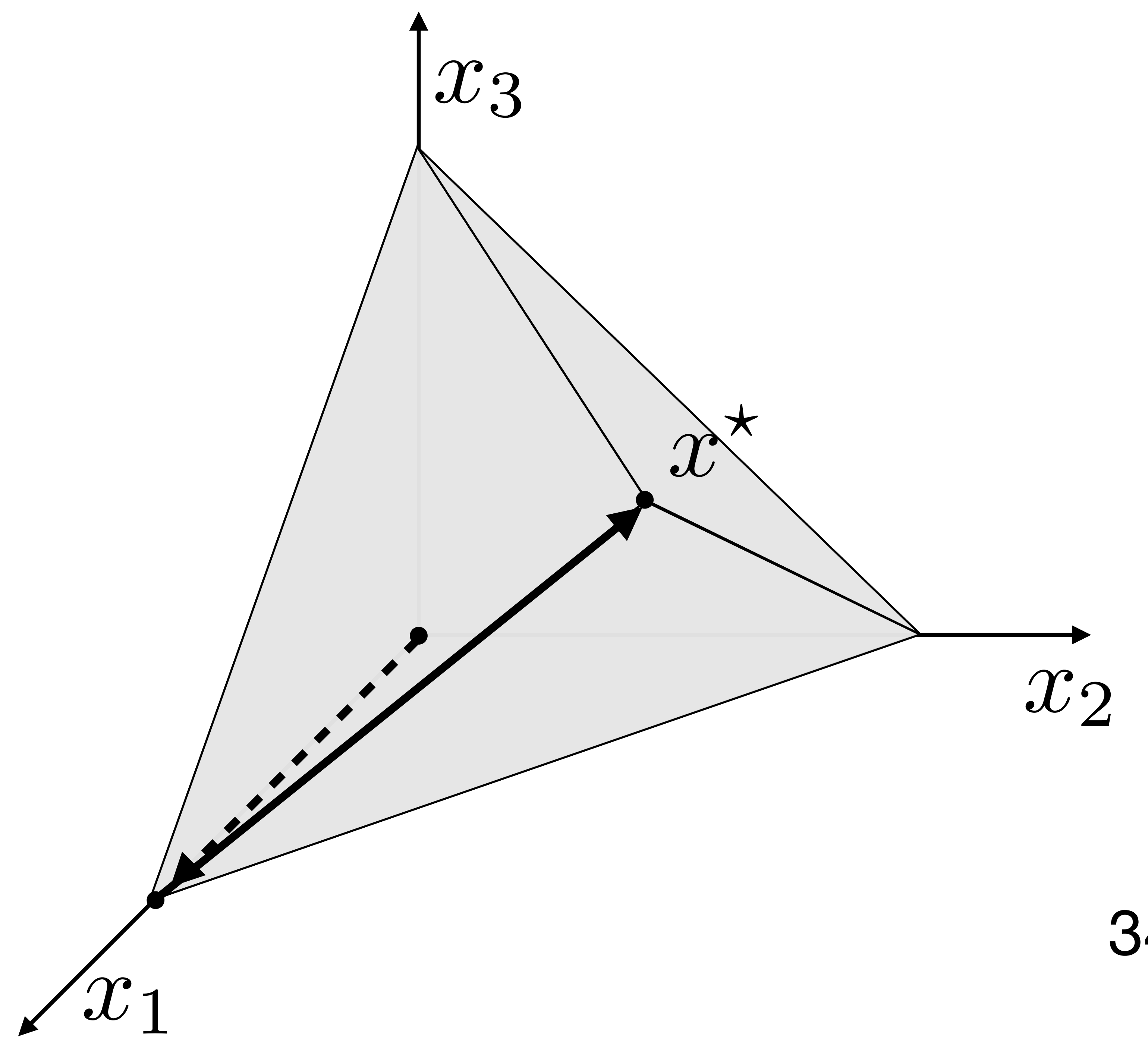
Reduced costs $\bar{c} = (0, 0, 0, 3.6, 1.6, 1.6)$

Solve $A_B^T p = c_B \quad \Rightarrow \quad p = (-3.6, -1.6, -1.6)$

$\bar{c} = c - A^T p = (0, 0, 0, 3.6, 1.6, 1.6)$

Optimal

$\bar{c} \geq 0 \longrightarrow x^\star = (4, 4, 4, 0, 0, 0)$



Simplex tableau implementation

Can we solve LPs by hand?

Minus cost →	$-c_B^T x_B$	$\bar{c}_1$	$\dots$	$\bar{c}_n$	← Reduced costs
Basic variables →	$x_B(1)$ $\vdots$ $x_B(1)$	$\vert$ $A_B^{-1} A_1$ $\vert$	$\dots$	$\vert$ $A_B^{-1} A_n$ $\vert$	

People did it **before computers were invented**!

Nobody does it anymore…

Empirical complexity

Example with real solver

GLPK (open-source)

Code

```
import numpy as np
import cvxpy as cp

c = np.array([-10, -12, -12])
A = np.array([[1, 2, 2],
              [2, 1, 2],
              [2, 2, 1]])
b = np.array([20, 20, 20])
n = len(c)

x = cp.Variable(n)
problem = cp.Problem(cp.Minimize(c @ x),
                     [A @ x <= b, x >= 0])
problem.solve(solver=cp.GLPK, verbose=True)
```

Output

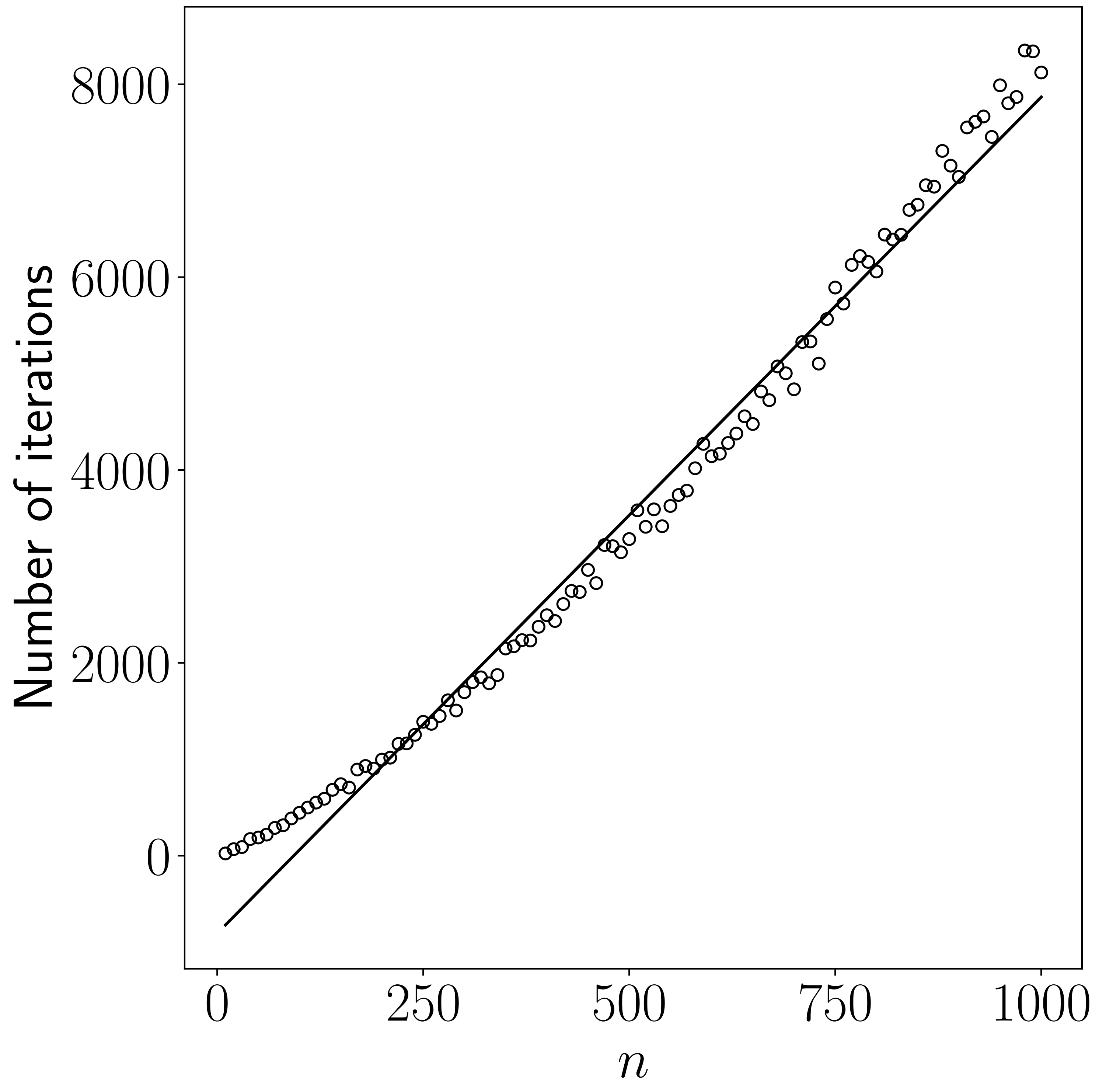
```
GLPK Simplex Optimizer, v4.65
6 rows, 3 columns, 12 non-zeros
*     0: obj =   0.000000000e+00 inf =   0.000e+00 (3)
*     3: obj =  -1.360000000e+02 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
```

Average simplex complexity

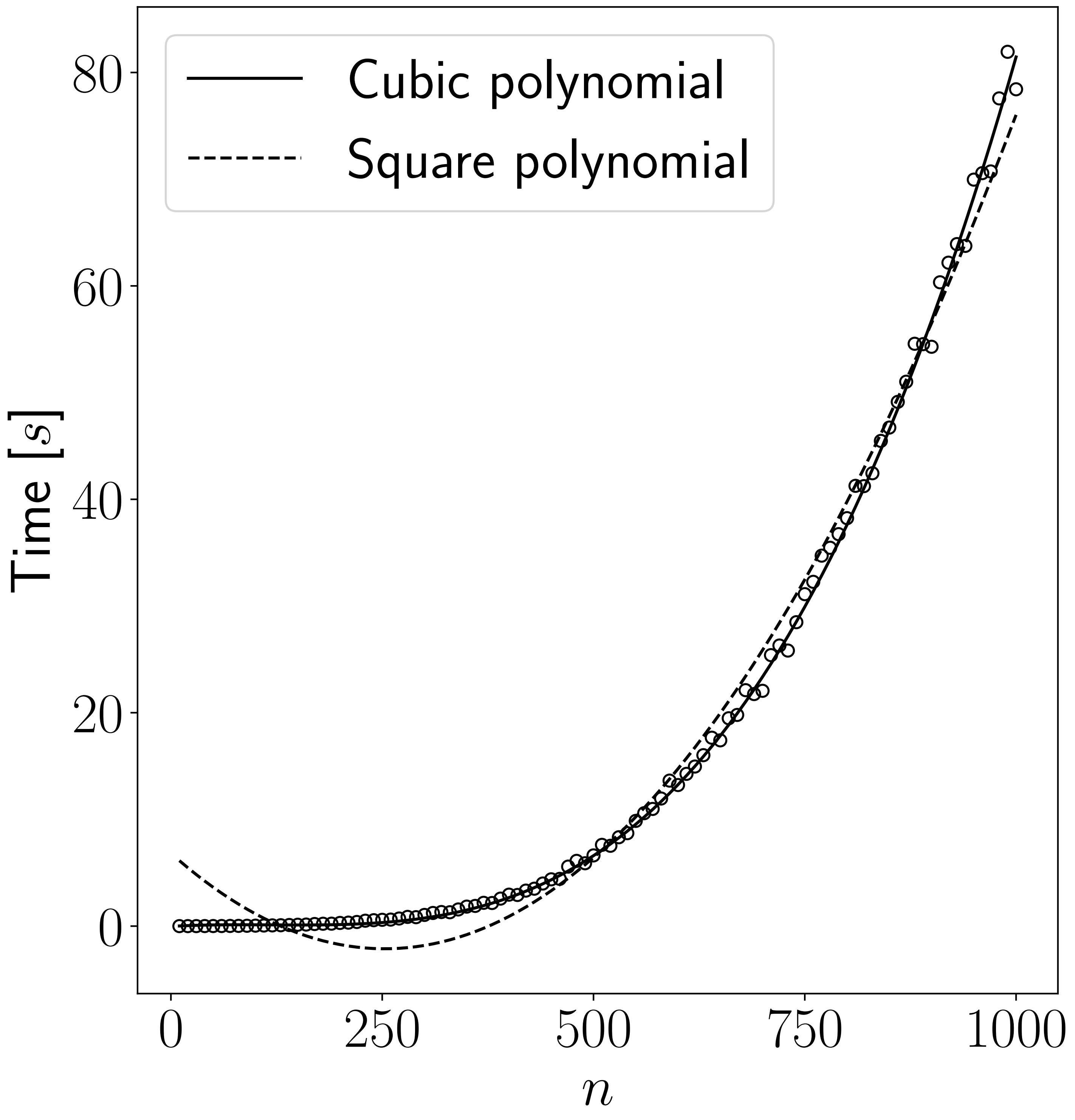
Random LPs

minimize	$c^T x$	n variables
subject to	$Ax \leq b$	$3n$ constraints

Iterations: $O(n)$



Time: $O(nn^2) = O(n^3)$



Numerical linear algebra and simplex implementation

Today, we learned to:

- **Identify** the pros and cons of different methods to solve a linear system
- **Derive** the computational complexity of the factor-solve method
- **Implement** a “realistic” version of the simplex method
- **Empirically** analyze the average complexity of the simplex method

Next lecture

- Linear optimization duality