Algorithms for Large-Scale Optimization

Georgios Paschos

Draft Edition — February 8, 2026

Contents

Chapter 1

Introduction to Mathematical Programming

1.1 What is Mathematical Programming?

Mathematical programming studies the problem of optimizing a function over a set defined by constraints. In its most general form, a mathematical program can be written as

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{X}, \end{aligned} \tag{1.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is an objective function and $\mathcal{X} \subseteq \mathbb{R}^n$ is a feasible set encoding constraints.

Despite its simple appearance, this formulation encompasses a wide range of problems arising in engineering, data science, economics, and operations research. The difficulty of a mathematical program depends critically on the structure of both the objective function and the feasible set.

1.2 Components of an Optimization Problem

Every optimization problem consists of three fundamental ingredients: decision variables, an objective function, and constraints.

The **decision variables** $x_1, x_2, \ldots, x_n$ represent the quantities we seek to determine. These are arranged in a column vector $x \in \mathbb{R}^n$, where n is called the *dimension* of the problem. Depending on the problem class, variables may be continuous (taking any real value), integer-valued, or binary.

The **objective function** $f : \mathbb{R}^n \to \mathbb{R}$ assigns a scalar value to each candidate solution x, representing the cost, loss, or utility we wish to minimize or maximize. Maximizing $f(x)$ is equivalent to minimizing $-f(x)$, so we adopt minimization as our standard form without loss of generality.

The **constraint set** (or feasible set) $\mathcal{X} \subseteq \mathbb{R}^n$ restricts the values that decision variables may take. Constraints arise from physical limitations, resource budgets, or logical requirements. We typically express

the constraint set through inequality and equality constraint functions:

$$\mathcal{X} = \{x \in \mathbb{R}^n : g_i(x) \leq 0,\ i = 1, \ldots, m, \quad h_j(x) = 0,\ j = 1, \ldots, p\}.$$

Three cases arise: $\mathcal{X} = \emptyset$ means the problem is **infeasible**; $\mathcal{X}$ unbounded may lead to an **unbounded** objective; and $\mathcal{X}$ bounded and nonempty guarantees an optimal solution exists under mild conditions.

A vector $x \in \mathbb{R}^n$ is called a **feasible solution** if $x \in \mathcal{X}$. Among all feasible solutions, we seek those that optimize the objective. A feasible solution $x^* \in \mathcal{X}$ is a **global optimum** if $f(x^*) \leq f(x)$ for all $x \in \mathcal{X}$. It is a **local optimum** if $f(x^*) \leq f(x)$ for all feasible x in some neighborhood of x^*. Every global optimum is a local optimum, but the converse is not true in general. This distinction profoundly affects algorithm design: for some problem classes every local optimum is global, while for others the optimization landscape contains many suboptimal local minima.

1.3 Problem Structure and Classification

The structure of f and $\mathcal{X}$ determines the problem class and, consequently, which algorithms are applicable. We distinguish three major classes that form the backbone of this book.

Convex Optimization. A set $\mathcal{X}$ is **convex** if for any $x, y \in \mathcal{X}$ and $\theta \in [0, 1]$, the point $\theta x + (1-\theta)y \in \mathcal{X}$. A function f is **convex** if $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$ for any x, y and $\theta \in [0, 1]$. An optimization problem is a **convex program** if both f and $\mathcal{X}$ are convex. Convex optimization occupies a privileged position: *every local optimum of a convex program is a global optimum.* This property eliminates the danger of converging to suboptimal solutions and enables algorithms with strong theoretical guarantees.

Linear Programming. A **linear program (LP)** has a linear objective $f(x) = c^T x$ and a feasible set defined by linear inequalities $\mathcal{X} = \{x : Ax \leq b\}$. Linear programs are a special case of convex programs. Their polyhedral structure leads to a remarkable property: if an optimal solution exists, at least one occurs at an *extreme point* (vertex) of the feasible region. This geometric insight underlies the simplex method.

Integer Programming. An **integer program (IP)** restricts some or all variables to integer values. When variables are binary $\{0, 1\}$, we have a **0-1 integer program**; when some variables are continuous and others integer, we have a **mixed-integer program (MIP)**. Integrality fundamentally changes the problem: the feasible set is no longer convex, and the problem becomes combinatorial. Integer programming is $\mathcal{NP}$-hard in general, meaning no polynomial-time algorithm is known.

These problem classes form a hierarchy of increasing difficulty, as depicted in Figure 1.1. A key insight is that *relaxation*—replacing a hard problem with an easier one—provides bounds and algorithmic leverage. For instance, dropping integrality constraints from an integer program yields an LP relaxation.

The central theme of this book is that *optimization algorithms exploit structure*:

- **Smoothness** enables gradient-based methods that use local derivative information.

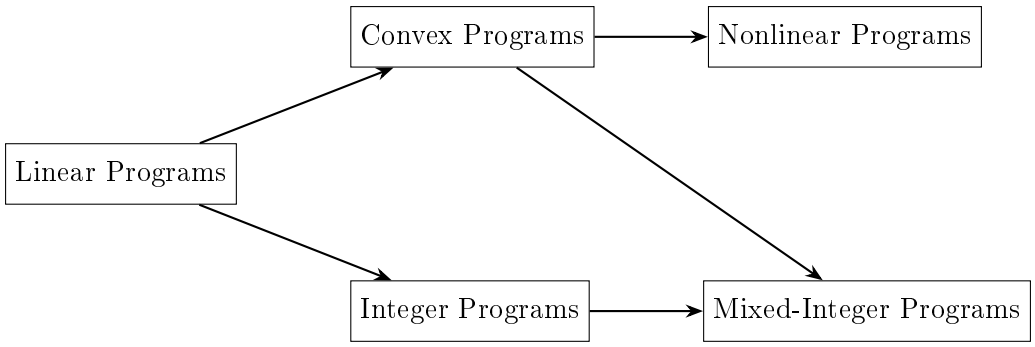

Figure 1.1: Hierarchy of optimization problem classes. Arrows indicate increasing generality and typically increasing computational difficulty.

- **Convexity** guarantees that local search finds global optima.
- **Linearity** yields polyhedral geometry and finite vertex-enumeration algorithms.
- **Integrality** introduces combinatorial complexity requiring enumeration and relaxation.

Understanding which structures are present is often more important than problem size: a large convex program may be easier to solve than a small integer program.

1.4 Industry Perspective: Optimization in Practice

To appreciate the scope of mathematical programming, consider representative applications from industry. These examples illustrate both the diversity of optimization problems and where each problem class arises in practice.

Convex Optimization. Training neural networks involves minimizing a loss function over billions of parameters. Despite nonconvexity of the overall landscape, the smooth structure enables stochastic gradient methods to find good solutions efficiently. Portfolio optimization, where investors balance expected return against risk, leads to convex quadratic programs. Statistical estimation problems—least squares, logistic regression, maximum likelihood—are naturally convex and solved routinely at massive scale.

Linear Programming. When a navigation app computes driving directions, it solves shortest path problems on graphs with hundreds of millions of nodes—these are LPs with special network structure admitting near-linear-time algorithms. Revenue management systems at airlines and hotels solve LPs to set prices dynamically. Telecommunication networks route data by solving multicommodity flow problems, which are LPs that can have millions of variables.

Integer Programming. Retailers and manufacturers optimize supply chains by deciding where to locate warehouses, how much inventory to hold, and which routes trucks should follow. These combine continuous quantities with discrete facility and routing decisions, yielding large mixed-integer programs. Vehicle

routing for delivery services, crew scheduling for airlines, and resource allocation in cloud computing are all integer programs where combinatorial structure demands sophisticated algorithms.

Table 1.1 summarizes how applications map to problem classes. A recurring theme is that *real problems combine multiple structures*: a supply chain problem may involve convex costs, linear constraints, and integer variables simultaneously. Effective solutions decompose such problems to exploit each component's structure.

Table 1.1: Representative applications and their optimization problem classes.

Application	**Problem Class**	**Typical Scale**
Neural network training	Nonconvex (smooth)	10^9 variables
Portfolio optimization	Convex QP	10^3–10^4 assets
Shortest paths / navigation	LP / Network flow	10^8 nodes
Revenue management	Linear program	10^5 variables
Multicommodity routing	Linear program	10^7 variables
Supply chain design	Mixed-integer LP	10^6 variables
Vehicle routing	Integer program	10^4 requests
Crew scheduling	Set covering IP	10^6 columns

1.5 Large-Scale Optimization

The scale evident in Table 1.1—millions or billions of variables—is what makes these problems genuinely challenging. Modern optimization problems are characterized by:

- **High dimensionality**: variables proliferate from fine discretization, rich data, or large networks.
- **Sparse structure**: most variables interact with only a few constraints, creating exploitable patterns.
- **Resource constraints**: memory, time, and communication are limited, especially in distributed systems.
- **Approximate solutions**: a good solution found quickly often beats an optimal solution found slowly.

These considerations demand algorithms designed for scale. First-order methods using only gradients replace second-order methods requiring expensive matrix factorizations. Decomposition techniques break monolithic problems into coordinated subproblems that can be solved in parallel, or by multiple agents. Randomization and stochastic approximation reduce per-iteration costs. For integer programs, heuristics and approximation algorithms provide practical solutions when exact methods are too slow.

Rather than exhaustive theory, this book emphasizes:

- Conceptual understanding of why optimization algorithms work
- The relationship between problem structure and algorithm design
- Methods that scale to large instances encountered in practice

What This Book Does Not Cover. This book focuses on *deterministic, static optimization*: we assume the problem data is known, and we seek a single decision that optimizes a fixed objective. Real-world problems are often more complex, and we briefly acknowledge several important extensions that lie beyond our scope.

- **Uncertainty.** Data may be missing, noisy, or based on forecasts. *Stochastic programming* models uncertainty through probability distributions over scenarios, while *robust optimization* hedges against worst-case realizations.
- **Dynamics.** Many problems unfold over time, with decisions affecting future states. *Dynamic programming* and *Markov decision processes* provide frameworks for sequential decision-making, while *online optimization* addresses settings where data arrives incrementally.
- **Learning.** When the underlying model is unknown or evolving, *reinforcement learning* and *bandit algorithms* learn good policies through interaction with the environment.
- **Multiple agents or objectives.** *Game theory* studies settings with competing decision-makers, while *multi-objective optimization* balances conflicting goals without a single scalar objective.

The vast majority of real-world problems involve some combination of these elements, and are often nonlinear with integer components besides. Why, then, focus on deterministic static optimization?

The answer is pragmatic. Based on the author's experience, the overwhelming majority of deployed industrial solutions rely on exactly the tools presented in this book—linear programs, convex optimization, and mixed-integer programming solved via branch-and-bound. Several factors explain this apparent paradox:

1. **Tractability.** Complex models incorporating uncertainty, dynamics, and learning are often intractable at industrial scale. Practitioners simplify, decompose, and approximate—frequently arriving at sequences of static optimization problems.
2. **Scalability.** Methods for stochastic and dynamic settings often struggle with the problem sizes routinely handled by LP and MIP solvers.
3. **Maintainability.** Static optimization models are easier to debug, interpret, and modify as business requirements evolve. A well-structured MIP can be adjusted by adding constraints; a reinforcement learning policy may require complete retraining, and a game theory strategy will require re-design from scratch.
4. **Diminishing returns.** The leap from ad-hoc heuristics to principled optimization typically yields the largest gains. Further sophistication offers incremental improvement at substantial complexity cost.

This is not to say that advanced modeling techniques are unhelpful—they can be essential in the right context. We merely observe that mastering the fundamentals of static optimization equips the practitioner to solve, or meaningfully approximate, a remarkably broad class of real problems.

1.6 How to Use This Book

This book is designed for both self-study and classroom use. Beyond the theoretical development, two features support active learning.

Code Examples. Optimization is a computational discipline: the goal is not just to formulate problems but to solve them. Throughout the book, short code examples in Python illustrate key algorithms and modeling techniques. These snippets use standard packages—`cvxpy` for convex optimization, `numpy` and `scipy` for numerical computation, and solver interfaces for linear and integer programming.

A companion repository contains complete, runnable notebooks with additional examples, visualizations, and computational experiments. Readers are encouraged to modify and extend these examples.

To give a flavor of what follows, consider a simple resource allocation problem. A company has budget b to invest across n projects, where project i has expected return r_i and requires investment $x_i \geq 0$. Maximizing total return subject to the budget constraint gives:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{maximize}} \quad & \sum_{i=1}^{n} r_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} x_i \leq b, \\ & x_i \geq 0, \quad i = 1, \ldots, n. \end{aligned} \tag{1.2}$$

This linear program can be formulated and solved in a few lines (see `ch0_resource_allocation.py` in the companion repository): https://github.com/gpasxos/large-scale-optimization/.

```
import cvxpy as cp
import numpy as np

n, b = 5, 100
r = np.array([0.12, 0.10, 0.07, 0.05, 0.03])

x = cp.Variable(n)
objective = cp.Maximize(r @ x)
constraints = [cp.sum(x) <= b, x >= 0]
problem = cp.Problem(objective, constraints)
problem.solve()

print(f"Optimal value: {problem.value:.2f}")
print(f"Optimal allocation: {x.value}")
```

The optimal solution invests the entire budget in project 1 (highest return). In later chapters, we will see how adding constraints—risk limits, diversification requirements, or integer restrictions—transforms both the problem structure and the solution approach.

Exercises and Checkpoints. Each chapter includes two types of exercises:

- **Conceptual checkpoints** appear inline throughout the text, testing understanding of definitions and key ideas. These are meant to be answered quickly before proceeding.
- **End-of-chapter exercises** range from short computations to modeling problems inspired by real applications. Selected solutions appear in Appendix **??**.

Check Your Understanding 1.1.

In the resource allocation example above, what happens to the optimal solution if we add the constraint $x_i \leq 30$ *for all* i*? What if we require* $x_i \in \{0, 50\}$ *(invest fully or not at all)?*

The first modification keeps the problem linear; the second makes it an integer program. This contrast—how small changes in problem structure dramatically affect algorithms and complexity—is a recurring theme of the book.

1.7 Organization of the Book

The book is organized around the problem hierarchy, progressing from problems with the most exploitable structure to those with the least.

Part I: Convex Optimization and First-Order Methods. We begin with convex optimization, where theory is most complete. After establishing foundations of convexity, duality, and optimality conditions, we develop gradient methods, proximal algorithms, and decomposition techniques for problems too large for second-order methods.

Part II: Linear Programming. We specialize to linear programming, studying the simplex method, interior-point methods, and large-scale techniques including column generation. LP provides exact polynomial-time algorithms and serves as the computational foundation for integer programming.

Part III: Integer Programming and Combinatorial Optimization. We address integer programming, where integrality introduces computational hardness. We study branch-and-bound, cutting planes, approximation algorithms, and heuristics. Network flow problems receive attention as integer programs solvable in polynomial time.

Throughout the book, applications from operations research, machine learning, and network optimization illustrate the concepts. The goal is to equip the reader with a coherent framework for understanding optimization algorithms and adapting them to new problem classes.

Part I

Foundations: Convexity, Duality, Optimality Conditions

Chapter 2

Introduction to Convex Programming

This chapter introduces *convex programming*: we seek a real-valued vector $x \in \mathbb{R}^n$ that minimizes a convex objective function over a convex feasible set. Convexity is the single most important structural property in optimization—it guarantees that local optima are global and enables efficient algorithms with provable guarantees. As a result, a lot of practical applications of optimization are built on the back of the convex programming theory, algorithms, or concepts.

2.1 Convex Sets

Before we introduce convex programs, we begin with the necessary definitions of convex sets and functions. We will go through all mathematical conventions that are necessary for understanding and deploying our algorithmic solutions.

Definition 2.1 (Convex set)**.** A set $\mathcal{X} \subseteq \mathbb{R}^n$ is **convex** if for any $x, y \in \mathcal{X}$ and any scalar $\theta \in [0, 1]$:

$$\theta x + (1 - \theta)y \in \mathcal{X}.$$

Geometrically, a set is convex if the line segment connecting any two points in the set lies entirely within the set. Figure 2.1 illustrates convex and non-convex sets in two dimensions.

A stronger notion is an **affine set**: $\mathcal{X}$ is affine if $\theta x + (1-\theta)y \in \mathcal{X}$ for all $\theta \in \mathbb{R}$ (the entire line through x and y, not just the segment). Every affine set is convex, but not vice versa.

Example 2.2 (Common convex sets)**.** Table 2.1 lists several important convex sets that appear frequently in optimization.

Operations preserving convexity. Complex convex sets can be built from simpler ones using operations that preserve convexity:

- **Intersection**: If $\mathcal{X}_1, \mathcal{X}_2$ are convex, then $\mathcal{X}_1 \cap \mathcal{X}_2$ is convex.
- **Affine transformation**: If $\mathcal{X}$ is convex and $f(x) = Ax + b$, then both $f(\mathcal{X}) = \{Ax + b : x \in \mathcal{X}\}$ and $f^{-1}(\mathcal{Y}) = \{x : Ax + b \in \mathcal{Y}\}$ are convex.

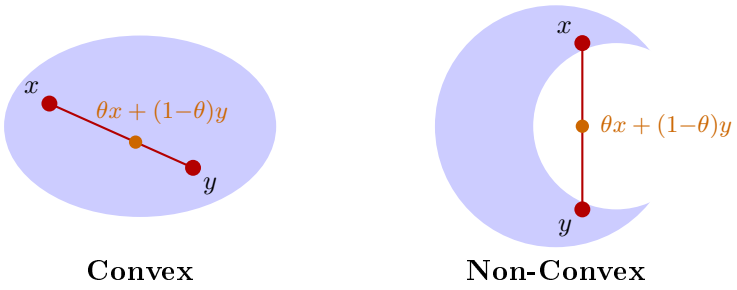

Figure 2.1: Examples of convex and non-convex sets for $n = 2$. **Left:** A convex set—the line segment between any two points x, y lies entirely within the set. **Right:** A non-convex set—there exist points x, y whose connecting segment passes outside the set.

Table 2.1: Common convex sets and their definitions.

Set	**Definition**	**Parameters**
Hyperplane	$\{x : a^T x = b\}$	$a \neq 0$, $b \in \mathbb{R}$
Halfspace	$\{x : a^T x \leq b\}$	$a \neq 0$, $b \in \mathbb{R}$
Polyhedron	$\{x : Ax \leq b\}$	$A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$
Euclidean ball	$\{x : \|x - x_0\| \leq r\}$	center x_0, radius $r > 0$
Ellipsoid	$\{x : (x - x_0)^T P^{-1}(x - x_0) \leq 1\}$	center x_0, $P \succ 0$
Second-order cone	$\{(x, t) : \|x\| \leq t\}$	$x \in \mathbb{R}^{n-1}$, $t \in \mathbb{R}$

For instance, a polyhedron is convex because it is the intersection of halfspaces. See [**?**] for a comprehensive treatment.

Check Your Understanding 2.1.

Which of the following sets are convex? Use the definition or operations above to justify your answer.

(a) $\{x \in \mathbb{R}^2 : x_1^2 + x_2^2 \leq 1\}$

(b) $\{x \in \mathbb{R}^2 : x_1^2 + x_2^2 \geq 1\}$

(c) $\{x \in \mathbb{R}^2 : x_1 \geq 0, x_2 \geq 0, x_1 + x_2 \leq 1\}$

(d) $\{x \in \mathbb{R}^2 : x_1 x_2 \geq 1, x_1 > 0, x_2 > 0\}$

2.2 Convex Functions

With convex sets in hand, we now turn to convex functions—the other essential ingredient for convex optimization.

Definition 2.3 (Convex function)**.** A function $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ is convex, is **convex** if for any $x, y \in \mathcal{X}$ and $\theta \in [0, 1]$:

$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta) f(y).$$

The function is **strictly convex** if the inequality is strict for $\theta \in (0,1)$ and $x \neq y$.

The defining inequality has a simple geometric interpretation: the function value at any point on the line segment lies below (or on) the chord connecting the endpoints. Equivalently, the **epigraph** $\{(x,t) : f(x) \leq t\}$—the region above the graph—is a convex set.

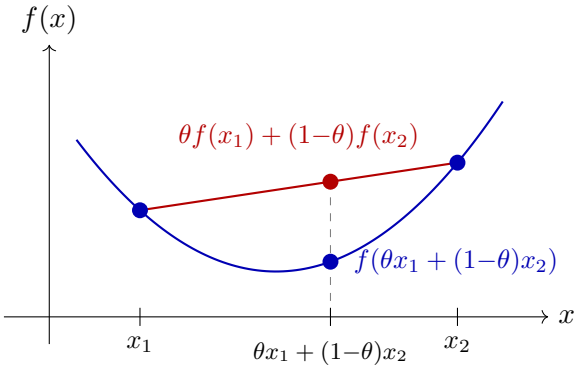

Figure 2.2: A convex function lies below the chord connecting any two points. The chord represents the linear interpolation $\theta f(x_1) + (1-\theta) f(x_2)$, while the function value $f(\theta x_1 + (1-\theta) x_2)$ is always at or below this chord.

If f is convex, then $-f$ is **concave**. Maximizing a concave function is equivalent to minimizing a convex function, so the theory of convex minimization covers both cases.

Checking convexity numerically. While analytical verification is preferable, we can check convexity numerically by sampling random points and testing the defining inequality. The following code tests whether a function $f : \mathbb{R}^n \to \mathbb{R}$ appears convex:

```
import numpy as np

def check_convexity(f, dim=2, n_samples=1000, bounds=(-5, 5)):
    """
    Numerically check if f: R^dim -> R appears convex.
    Returns (is_convex, number_of_violations).
    """
    violations = 0

    for _ in range(n_samples):
        # Generate two random points in R^dim
        x = np.random.uniform(bounds[0], bounds[1], dim)
        y = np.random.uniform(bounds[0], bounds[1], dim)
        theta = np.random.uniform(0, 1)

        # Convexity inequality: f(theta*x + (1-theta)*y) <= theta*f(x) + (1-theta)*f(y)
        z = theta * x + (1 - theta) * y    # Interpolated point
```

```
        lhs = f(z)                          # f evaluated at interpolation
        rhs = theta * f(x) + (1 - theta) * f(y)  # Interpolation of f values

        if lhs > rhs + 1e-9:  # Small tolerance for numerical error
            violations += 1

    return violations == 0, violations

# Define test functions (R^n -> R)
def f_quadratic(x):
    """f(x) = x_1^2 + x_2^2 + ... + x_n^2 (convex)"""
    return np.sum(x**2)

def f_norm(x):
    """f(x) = ||x||_2 (convex)"""
    return np.linalg.norm(x)

def f_nonconvex(x):
    """f(x) = sin(x_1) + x_2 (neither convex nor concave)"""
    return np.sin(x[0]) + x[1]

# Test in 2 dimensions
print("Testing f(x) = ||x||^2:")
is_convex, violations = check_convexity(f_quadratic, dim=2)
print(f"  Convex: {is_convex}, Violations: {violations}/1000")

print("Testing f(x) = ||x||_2:")
is_convex, violations = check_convexity(f_norm, dim=2)
print(f"  Convex: {is_convex}, Violations: {violations}/1000")

print("Testing f(x) = sin(x_1) + x_2:")
is_convex, violations = check_convexity(f_nonconvex, dim=2)
print(f"  Convex: {is_convex}, Violations: {violations}/1000")
```

This numerical test cannot *prove* convexity (we only check finitely many points), but violations indicate the function is not convex. For rigorous verification, we use the analytical conditions in the next section.

Example 2.4 (Common convex and concave functions)**.** Table 2.2 lists important convex and concave functions. These serve as building blocks for more complex objectives.

Operations preserving convexity. Just as with sets, we can build complex convex functions from simpler ones:

- **Nonnegative weighted sum**: If $f_1, \ldots, f_k$ are convex and $\alpha_i \geq 0$, then $\sum_i \alpha_i f_i$ is convex.

Table 2.2: Common convex and concave functions.

Function	Domain	Convexity	Notes
e^{ax}	$\mathbb{R}$	Convex	Any $a \in \mathbb{R}$
x^a	$\mathbb{R}_+$	Convex if $a \leq 0$ or $a \geq 1$	Concave if $0 \leq a \leq 1$
$\log x$	$\mathbb{R}_{++}$	Concave	
$x \log x$	$\mathbb{R}_+$	Convex	Entropy (negated)
$\|x\|_p$	$\mathbb{R}^n$	Convex	Any $p \geq 1$
$\max\{x_1, \ldots, x_n\}$	$\mathbb{R}^n$	Convex	Piecewise linear
$\log \sum_i e^{x_i}$	$\mathbb{R}^n$	Convex	Log-sum-exp
$(\prod_i x_i)^{1/n}$	$\mathbb{R}^n_+$	Concave	Geometric mean

- **Pointwise maximum**: If $f_1, \ldots, f_k$ are convex, then $\max_i f_i(x)$ is convex.
- **Composition with affine**: If f is convex, then $g(x) = f(Ax + b)$ is convex.
- **Pointwise supremum**: If $f(x, y)$ is convex in x for each y, then $g(x) = \sup_y f(x, y)$ is convex.

Check Your Understanding 2.2.

Determine whether each function is convex, concave, or neither, using Table 2.2 and the preservation rules above:

(a) $f(x) = 3e^x + 2e^{-x}$ *on* $\mathbb{R}$

(b) $f(x) = \sqrt{x}$ *on* $\mathbb{R}_+$

(c) $f(x_1, x_2) = \|x\|_1 + \|x\|_2$ *on* $\mathbb{R}^2$

(d) $f(x_1, x_2) = \max\{x_1, x_2\} - \log(x_1 + x_2)$ *on* $\mathbb{R}^2_{++}$

2.3 Derivative Conditions for Convexity

The definition of convexity can be difficult to verify directly. For differentiable functions, derivative-based conditions provide more practical tests.

2.3.1 First-Order Condition

Theorem 2.5 (First-order condition for convexity)**.** *Let $f : \mathcal{X} \to \mathbb{R}$ be differentiable, where $\mathcal{X} \subseteq \mathbb{R}^n$ is convex. Then f is convex if and only if for all $x, y \in \mathcal{X}$:*

$$f(y) \geq f(x) + \nabla f(x)^T (y - x). \tag{2.1}$$

Geometrically, this states that a convex function lies above all of its tangent hyperplanes. The right-hand side is the first-order Taylor approximation of f at x, so convexity means the function is always underestimated by its linear approximation. Figure 2.3 illustrates this property.

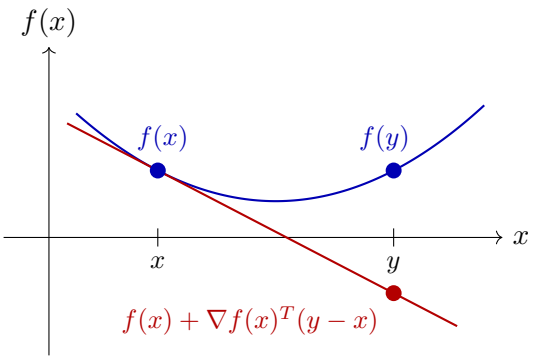

Figure 2.3: A convex function lies above its tangent line at every point. The tangent at x gives a global lower bound: $f(y) \geq f(x) + \nabla f(x)^T(y - x)$ for all y.

Proof. ($\Rightarrow$) Assume f is convex. For any $x, y \in \mathcal{X}$ and $\theta \in (0, 1]$, let $z = \theta y + (1 - \theta)x = x + \theta(y - x)$. By convexity:

$$f(z) = f(x + \theta(y - x)) \leq \theta f(y) + (1 - \theta)f(x).$$

Rearranging:

$$\frac{f(x + \theta(y - x)) - f(x)}{\theta} \leq f(y) - f(x).$$

Taking $\theta \to 0^+$, the left side becomes $\nabla f(x)^T(y - x)$, giving $f(y) \geq f(x) + \nabla f(x)^T(y - x)$.

($\Leftarrow$) Assume (2.1) holds. For any $x, y \in \mathcal{X}$ and $\theta \in [0, 1]$, let $z = \theta x + (1 - \theta)y$. Applying (2.1) twice:

$$f(x) \geq f(z) + \nabla f(z)^T(x - z) = f(z) + (1 - \theta)\nabla f(z)^T(x - y),$$
$$f(y) \geq f(z) + \nabla f(z)^T(y - z) = f(z) - \theta\nabla f(z)^T(x - y).$$

Multiplying by θ and $(1 - \theta)$ respectively and adding:

$$\theta f(x) + (1 - \theta)f(y) \geq f(z) = f(\theta x + (1 - \theta)y).$$

□

2.3.2 Second-Order Condition

For twice-differentiable functions, convexity can be characterized through the Hessian matrix.

Definition 2.6 (Hessian matrix)**.** For a twice-differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the **Hessian** at x is the

$n \times n$ matrix of second partial derivatives:

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Remark 2.7 (Symmetry of the Hessian). By Schwarz's theorem (also called Clairaut's theorem), if the second partial derivatives are continuous, then the mixed partials are equal: $\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$. This makes the Hessian symmetric. Throughout this book, we assume sufficient regularity for symmetry to hold.

Theorem 2.8 (Second-order condition for convexity)**.** *Let $f : \mathcal{X} \to \mathbb{R}$ be twice differentiable on a convex set $\mathcal{X} \subseteq \mathbb{R}^n$. Then f is convex if and only if for all $x \in \mathcal{X}$:*

$$\nabla^2 f(x) \succeq 0,$$

*where $A \succeq 0$ means A is **positive semidefinite**: $u^T A u \geq 0$ for all $u \in \mathbb{R}^n$.*

Equivalently, all eigenvalues of $\nabla^2 f(x)$ must be nonnegative. For strict convexity, we require $\nabla^2 f(x) \succ 0$ (positive definite—all eigenvalues strictly positive).

Remark 2.9. The Hessian captures the curvature of f. The directional second derivative along direction u is:

$$\frac{d^2}{dt^2} f(x + tu)\Big|_{t=0} = u^T \nabla^2 f(x)\, u.$$

Positive semidefiniteness means the function curves upward (or is flat) in every direction—the hallmark of convexity.

Example 2.10 (Quadratic functions)**.** Consider $f(x) = \frac{1}{2} x^T Q x + b^T x + c$ where Q is symmetric. Then:

$$\nabla f(x) = Qx + b, \qquad \nabla^2 f(x) = Q.$$

The function f is convex if and only if $Q \succeq 0$.

Check Your Understanding 2.3.

For each function, compute the Hessian and determine if the function is convex:

(a) $f(x_1, x_2) = x_1^2 + 4x_2^2 + 2x_1 x_2$

(b) $f(x_1, x_2) = e^{x_1 + x_2}$

(c) $f(x_1, x_2) = x_1^2 - x_2^2$

(d) $f(x_1, x_2) = \log(e^{x_1} + e^{x_2})$

Verifying convexity via Hessian eigenvalues. For functions of moderate dimension, we can check convexity numerically by computing Hessian eigenvalues at sample points:

```
import numpy as np

def numerical_hessian(f, x, eps=1e-5):
    """Compute Hessian of f at x using finite differences."""
    n = len(x)
    H = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            # Second partial derivative via central differences
            ei, ej = np.zeros(n), np.zeros(n)
            ei[i], ej[j] = eps, eps

            H[i, j] = (f(x + ei + ej) - f(x + ei - ej)
                       - f(x - ei + ej) + f(x - ei - ej)) / (4 * eps**2)

    return (H + H.T) / 2  # Ensure symmetry

def check_convexity_hessian(f, dim=2, n_points=100, bounds=(-5, 5)):
    """Check if Hessian is PSD at random points."""
    min_eigenvalue = float('inf')

    for _ in range(n_points):
        x = np.random.uniform(bounds[0], bounds[1], dim)
        H = numerical_hessian(f, x)
        eigvals = np.linalg.eigvalsh(H)  # Eigenvalues of symmetric matrix
        min_eigenvalue = min(min_eigenvalue, eigvals.min())

    is_convex = min_eigenvalue >= -1e-8  # Tolerance for numerical error
    return is_convex, min_eigenvalue

# Test functions
def f_convex(x):
    return x[0]**2 + 4*x[1]**2 + 2*x[0]*x[1]  # Quiz (a)

def f_nonconvex(x):
    return x[0]**2 - x[1]**2  # Quiz (c)

print("f(x) = x1^2 + 4*x2^2 + 2*x1*x2:")
is_cvx, min_eig = check_convexity_hessian(f_convex)
print(f"  Convex: {is_cvx}, Min eigenvalue: {min_eig:.4f}")
```

```
print("f(x) = x1^2 - x2^2:")
is_cvx, min_eig = check_convexity_hessian(f_nonconvex)
print(f"  Convex: {is_cvx}, Min eigenvalue: {min_eig:.4f}")
```

2.4 Convex Programs

We now formally define the class of optimization problems that will occupy us throughout part I.

Definition 2.11 (Convex program)**.** A **convex program** is an optimization problem of the form:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & Ax = b, \end{aligned} \tag{2.2}$$

where f and $g_1, \ldots, g_m$ are convex functions, and $A \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^p$ define linear equality constraints.

The feasible set $\mathcal{X} = \{x : g_i(x) \leq 0,\ Ax = b\}$ is convex (intersection of sublevel sets of convex functions and an affine set). Combined with a convex objective, this structure guarantees favorable properties explored in the next section. We often write a convex program more compactly as:

$$\underset{x \in \mathcal{X}}{\text{minimize}} \quad f(x), \tag{2.3}$$

where $\mathcal{X}$ is understood to be a convex set.

The power of the convex framework. Definition 2.11 may appear abstract, but it encompasses a remarkable variety of practical problems. By choosing different convex objectives and constraints, we recover:

- **Linear programming**: $f(x) = c^T x$, $g_i(x) = a_i^T x - b_i$ (supply chains, scheduling, resource allocation)
- **Quadratic programming**: $f(x) = \frac{1}{2} x^T Q x + c^T x$ with $Q \succeq 0$ (portfolio optimization, model predictive control)
- **Second-order cone programming**: constraints of the form $\|Ax + b\| \leq c^T x + d$ (robust optimization, antenna array design)
- **Semidefinite programming**: matrix variable $X \succeq 0$ with linear objective (sensor network localization, combinatorial relaxations)
- **Geometric programming**: products and ratios of monomials (circuit design, chemical engineering)

We will explore several concrete examples in Section 2.7. The key insight is that recognizing a problem as convex immediately unlocks efficient algorithms with guaranteed convergence—a luxury not available for general nonconvex optimization.

2.5 Function Properties for Algorithm Analysis

The performance of optimization algorithms depends critically on properties of the objective function beyond mere convexity. Understanding these properties allows us to select appropriate algorithms: we might use momentum when the function is smooth, apply acceleration for strongly convex objectives, or be cautious about convergence rates when the condition number is large. We now introduce three key properties that govern convergence rates.

2.5.1 Lipschitz Continuity

Definition 2.12 (Lipschitz continuity)**.** A function $f : \mathcal{X} \to \mathbb{R}$ is G**-Lipschitz continuous** if for all $x, y \in \mathcal{X}$:

$$|f(x) - f(y)| \leq G\|x - y\|.$$

The constant $G \geq 0$ is called the **Lipschitz constant**.

Lipschitz continuity bounds how fast the function value can change. If f is differentiable, then G-Lipschitz continuity is equivalent to:

$$\|\nabla f(x)\| \leq G \quad \text{for all } x \in \mathcal{X}.$$

Lipschitz continuity ensures that the norm of the gradient is bounded, which will be our blanket assumption for proving convergence of the gradient descent algorithm.

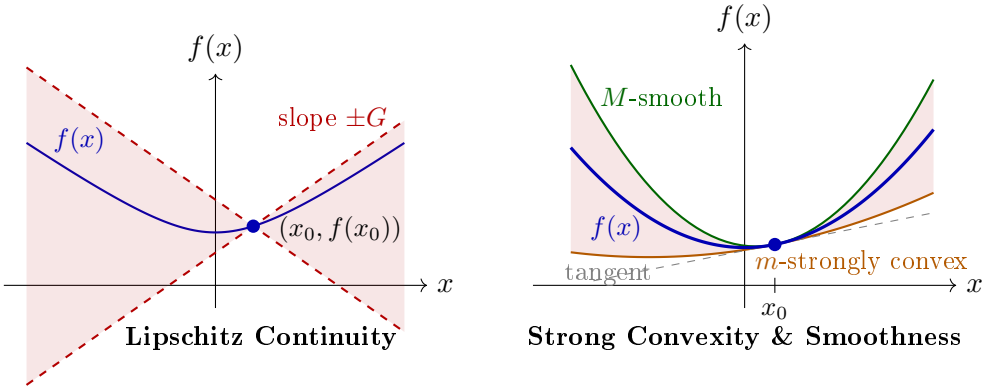

Figure 2.4: Function properties. **Left:** A G-Lipschitz function stays within a cone of slope $\pm G$ centered at any point. **Right:** A function that is both m-strongly convex and M-smooth is sandwiched between two parabolas touching the tangent line at any point.

Example 2.13.

- Linear functions $f(x) = c^T x$ are $\|c\|$-Lipschitz.
- The function $f(x) = \|x\|$ is 1-Lipschitz.

- Quadratic functions $f(x) = x^T Qx$ are *not* globally Lipschitz (gradients grow unbounded), but are Lipschitz on any bounded set.

2.5.2 Smoothness

Definition 2.14 (Smoothness)**.** A differentiable function $f : \mathcal{X} \to \mathbb{R}$ is M**-smooth** if for all $x, y \in \mathcal{X}$:

$$f(y) \leq f(x) + \nabla f(x)^T (y - x) + \frac{M}{2}\|y - x\|^2. \tag{2.4}$$

While convexity provides a lower bound via the first-order condition, smoothness provides an upper bound: the function grows no faster than a quadratic. For twice-differentiable functions, M-smoothness is equivalent to:

$$\nabla^2 f(x) \preceq MI \quad \text{for all } x \in \mathcal{X},$$

or equivalently, all eigenvalues of the Hessian are at most M:

$$\lambda_{\max}(\nabla^2 f(x)) \leq M.$$

Smoothness controls how well gradient steps predict function decrease—crucial for analyzing gradient descent.

2.5.3 Strong Convexity

Strong convexity strengthens ordinary convexity by requiring the function to curve upward by at least a fixed amount. To motivate the definition, recall that convexity means f lies above its tangent planes. Strong convexity demands more: f must lie above a *parabola* touching the tangent plane.

Definition 2.15 (Strong convexity)**.** A differentiable function $f : \mathcal{X} \to \mathbb{R}$ is m**-strongly convex** (with $m > 0$) if for all $x, y \in \mathcal{X}$:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{m}{2}\|y - x\|^2. \tag{2.5}$$

Comparing with the first-order convexity condition (2.1), we see that strong convexity adds the term $\frac{m}{2}\|y - x\|^2$. This extra curvature has powerful consequences.

For twice-differentiable functions, m-strong convexity is equivalent to:

$$\nabla^2 f(x) \succeq mI \quad \text{for all } x \in \mathcal{X},$$

meaning all Hessian eigenvalues are at least m:

$$\lambda_{\min}(\nabla^2 f(x)) \geq m.$$

Strong convexity implies a unique minimizer x^*. To see why, suppose there were two minimizers x^*

and y^* with $f(x^*) = f(y^*) = f^*$. Applying (2.5) with $x = x^*$ and $y = y^*$:

$$f(y^*) \geq f(x^*) + \nabla f(x^*)^T(y^* - x^*) + \frac{m}{2}\|y^* - x^*\|^2.$$

Since x^* is a minimizer, $\nabla f(x^*) = 0$, so $f^* \geq f^* + \frac{m}{2}\|y^* - x^*\|^2$, which forces $y^* = x^*$.

Strong convexity also provides a useful stopping criterion. Applying (2.5) with $y = x^*$:

$$f(x^*) \geq f(x) + \nabla f(x)^T(x^* - x) + \frac{m}{2}\|x^* - x\|^2.$$

Using $f(x^*) \leq f(x)$ and rearranging:

$$0 \geq \nabla f(x)^T(x^* - x) + \frac{m}{2}\|x^* - x\|^2.$$

By Cauchy-Schwarz, $\nabla f(x)^T(x^* - x) \geq -\|\nabla f(x)\|\|x^* - x\|$, so:

$$\|\nabla f(x)\|\|x^* - x\| \geq \frac{m}{2}\|x^* - x\|^2 \quad \Rightarrow \quad \|x^* - x\| \leq \frac{2}{m}\|\nabla f(x)\|.$$

Combined with the first-order bound $f(x) - f(x^*) \leq \nabla f(x)^T(x - x^*) \leq \|\nabla f(x)\|\|x - x^*\|$:

$$f(x) - f^* \leq \frac{2}{m}\|\nabla f(x)\|^2. \tag{2.6}$$

This provides a practical stopping criterion: if $\|\nabla f(x)\| \leq \sqrt{m\epsilon/2}$, then $f(x) - f^* \leq \epsilon$.

2.5.4 Condition Number

When a function is both m-strongly convex and M-smooth, the ratio $\kappa = M/m \geq 1$ is called the **condition number**. It measures how "stretched" the function's level sets are:

- $\kappa = 1$: perfectly conditioned (isotropic level sets, like $f(x) = \|x\|^2$)
- $\kappa \gg 1$: ill-conditioned (elongated level sets, slow convergence)

The condition number determines convergence rates for many algorithms. Gradient descent, for instance, converges linearly with rate $1 - 1/\kappa$.

Example 2.16 (Quadratic function)**.** For $f(x) = \frac{1}{2}x^TQx$ with $Q \succ 0$:

- $m = \lambda_{\min}(Q)$ (smallest eigenvalue)
- $M = \lambda_{\max}(Q)$ (largest eigenvalue)
- $\kappa = \lambda_{\max}(Q)/\lambda_{\min}(Q)$

Table 2.3: Function properties and their characterizations.

Property	Condition	Hessian	Implication
Convex	$f(y) \geq f(x) + \nabla f(x)^T(y-x)$	$\nabla^2 f \succeq 0$	Local = global
G-Lipschitz	$\lvert f(x) - f(y)\rvert \leq G\lVert x-y\rVert$	$\lVert\nabla f\rVert \leq G$	Bounded gradients
M-smooth	$f(y) \leq f(x) + \nabla f(x)^T(y-x) + \frac{M}{2}\lVert y-x\rVert^2$	$\nabla^2 f \preceq MI$	Upper curvature bound
m-strongly convex	$f(y) \geq f(x) + \nabla f(x)^T(y-x) + \frac{m}{2}\lVert y-x\rVert^2$	$\nabla^2 f \succeq mI$	Lower curvature bound

2.5.5 Summary of Function Properties

Table 2.3 summarizes the key properties and their implications.

Check Your Understanding 2.4.

For the function $f(x) = \frac{1}{2}x^TQx$ *where* $Q = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}$*:*

(a) What is the smoothness constant M*?*

(b) What is the strong convexity constant m*?*

(c) What is the condition number κ*?*

(d) If $\lVert\nabla f(x)\rVert = 0.1$*, what is an upper bound on* $f(x) - f^*$*?*

Computing function properties numerically. For quadratic functions, we can compute the smoothness and strong convexity constants exactly from the Hessian eigenvalues:

```
import numpy as np

def analyze_quadratic(Q):
    """Analyze properties of f(x) = 0.5 * x^T Q x."""
    eigenvalues = np.linalg.eigvalsh(Q)

    m = eigenvalues.min()  # Strong convexity constant
    M = eigenvalues.max()  # Smoothness constant
    kappa = M / m if m > 0 else float('inf')  # Condition number

    print(f"Eigenvalues: {eigenvalues}")
    print(f"Strong convexity (m): {m:.4f}")
    print(f"Smoothness (M): {M:.4f}")
    print(f"Condition number (kappa): {kappa:.4f}")

    return m, M, kappa

# Example: well-conditioned
```

```
print("Well-conditioned quadratic:")
Q1 = np.array([[2, 0], [0, 1]])
analyze_quadratic(Q1)

print("\nIll-conditioned quadratic:")
Q2 = np.array([[100, 0], [0, 1]])
analyze_quadratic(Q2)

print("\nQuiz example:")
Q3 = np.array([[4, 0], [0, 1]])
m, M, kappa = analyze_quadratic(Q3)
```

2.6 Optimality in Convex Optimization

Convex optimization enjoys a remarkable property: any locally optimal solution is globally optimal. This section establishes this result and derives practical optimality conditions.

2.6.1 Local Optima are Global

Recall the definition of global and local optimum from Chapter 1. Every global optimum is trivially a local optimum. The converse is false for general optimization problems—nonconvex functions can have many local minima that are not global. However, convexity eliminates this difficulty.

Theorem 2.17 (Local is global for convex optimization)**.** *Let $f : \mathcal{X} \to \mathbb{R}$ be convex, where $\mathcal{X}$ is convex. If x^* is a local optimum, then x^* is a global optimum.*

Proof. Suppose x^* is a local optimum but not global. Then there exists $y \in \mathcal{X}$ with $f(y) < f(x^*)$. Consider points on the line segment from x^* to y:

$$z_\theta = \theta y + (1-\theta)x^*, \quad \theta \in (0, 1].$$

By convexity of $\mathcal{X}$, we have $z_\theta \in \mathcal{X}$. By convexity of f:

$$f(z_\theta) \leq \theta f(y) + (1-\theta)f(x^*) < \theta f(x^*) + (1-\theta)f(x^*) = f(x^*).$$

For sufficiently small $\theta > 0$, the point z_θ is arbitrarily close to x^*:

$$\|z_\theta - x^*\| = \theta\|y - x^*\| < \epsilon$$

for any $\epsilon > 0$. Thus z_θ lies in any neighborhood of x^* and satisfies $f(z_\theta) < f(x^*)$, contradicting local optimality. □

This theorem is the foundation of convex optimization: we can use local search methods (like gradient descent) with confidence that any minimum found is global.

Check Your Understanding 2.5.
Consider minimizing $f(x) = |x|$ over $\mathcal{X} = \mathbb{R}$.

(a) Is f convex?

(b) Is $x^* = 0$ a local optimum?

(c) Is $x^* = 0$ a global optimum?

(d) Does the theorem apply even though f is not differentiable at $x^* = 0$?

2.6.2 First-Order Optimality Conditions

For differentiable convex functions, we can characterize optimal points through the gradient.

Theorem 2.18 (First-order optimality condition)**.** *Let $f : \mathcal{X} \to \mathbb{R}$ be convex and differentiable, where $\mathcal{X}$ is convex. Then $x^* \in \mathcal{X}$ is a global optimum if and only if:*

$$\nabla f(x^*)^T (y - x^*) \geq 0 \quad \textit{for all } y \in \mathcal{X}. \tag{2.7}$$

Proof. ($\Rightarrow$) Suppose x^* is optimal but (2.7) fails: there exists $y \in \mathcal{X}$ with $\nabla f(x^*)^T (y - x^*) < 0$. Consider $z(t) = x^* + t(y - x^*)$ for small $t > 0$. By convexity of $\mathcal{X}$, $z(t) \in \mathcal{X}$. The directional derivative is:

$$\frac{d}{dt} f(z(t))\Big|_{t=0} = \nabla f(x^*)^T (y - x^*) < 0.$$

Thus for small $t > 0$, $f(z(t)) < f(x^*)$, contradicting optimality.

($\Leftarrow$) Suppose (2.7) holds. By the first-order convexity condition:

$$f(y) \geq f(x^*) + \nabla f(x^*)^T (y - x^*) \geq f(x^*)$$

for all $y \in \mathcal{X}$. Hence x^* is optimal. □

Geometrically, condition (2.7) states that the negative gradient $-\nabla f(x^*)$ does not point into the interior of $\mathcal{X}$. Equivalently, there is no feasible descent direction.

2.6.3 Unconstrained Optimality

For unconstrained problems where $\mathcal{X} = \mathbb{R}^n$, the optimality condition simplifies dramatically.

Corollary 2.19 (Unconstrained optimality)**.** *For an unconstrained convex problem $\min_{x \in \mathbb{R}^n} f(x)$ with f differentiable, x^* is optimal if and only if:*

$$\nabla f(x^*) = 0. \tag{2.8}$$

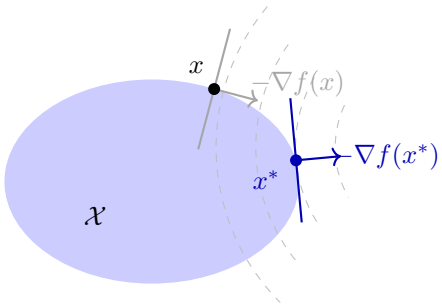

Figure 2.5: First-order optimality condition. At a suboptimal point x, the half-space indicated by the negative gradient $-\nabla f(x)$ includes feasible points—improvement is possible. At the optimal point x^*, the supporting hyperplane separates the feasible set from the descent direction: all points in the halfspace defined by $-\nabla f(x^*)$ are infeasible, so $\nabla f(x^*)^T(y - x^*) \geq 0$ for all feasible y.

Proof. Since $\mathcal{X} = \mathbb{R}^n$, for any x^* we can choose $y = x^* - \nabla f(x^*)$, which is feasible. Condition (2.7) requires:

$$\nabla f(x^*)^T(y - x^*) = \nabla f(x^*)^T(-\nabla f(x^*)) = -\|\nabla f(x^*)\|^2 \geq 0.$$

This holds only if $\|\nabla f(x^*)\| = 0$, i.e., $\nabla f(x^*) = 0$. □

This reduces optimization to solving the system of equations $\nabla f(x^*) = 0$—finding where the gradient vanishes. For convex f, any such point is a global minimum.

Remark 2.20 (Nonconvex functions). For nonconvex functions, $\nabla f(x^*) = 0$ characterizes *stationary points*, which include local minima, local maxima, and saddle points. Only convexity guarantees that stationary points are global minima.

Example 2.21 (Quadratic minimization)**.** Consider $f(x) = \frac{1}{2}x^TQx - b^Tx$ with $Q \succ 0$. Then $\nabla f(x) = Qx - b$, and the optimality condition $\nabla f(x^*) = 0$ yields:

$$Qx^* = b \quad \Rightarrow \quad x^* = Q^{-1}b.$$

This linear system can be solved in $O(n^3)$ time, or faster with structure.

Check Your Understanding 2.6.

Consider the constrained problem:

$$\begin{aligned} \underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & \frac{1}{2}(x_1^2 + x_2^2) \\ \text{subject to} \quad & x_1 + x_2 \geq 1. \end{aligned}$$

(a) Is this a convex program?

(b) Does the unconstrained optimum $x = (0, 0)$ satisfy the constraint?

(c) Using geometric intuition, where is the constrained optimum?

(d) Verify your answer satisfies the first-order condition (2.7).

Visualizing optimality conditions. The following code visualizes the first-order optimality condition for a constrained problem:

```
import numpy as np
import matplotlib.pyplot as plt

# Objective: f(x) = 0.5 * ||x||^2
def f(x):
    return 0.5 * np.sum(x**2)

def grad_f(x):
    return x

# Feasible set: x1 + x2 >= 1 (region above the line)
# Constrained optimum: closest point to origin on the line x1 + x2 = 1

# Create figure
fig, ax = plt.subplots(figsize=(8, 8))

# Plot contours of objective
x1 = np.linspace(-0.5, 2, 100)
x2 = np.linspace(-0.5, 2, 100)
X1, X2 = np.meshgrid(x1, x2)
Z = 0.5 * (X1**2 + X2**2)
ax.contour(X1, X2, Z, levels=10, cmap='Blues', alpha=0.7)

# Plot constraint boundary
ax.plot(x1, 1 - x1, 'r-', linewidth=2, label='$x_1 + x_2 = 1$')
ax.fill_between(x1, 1 - x1, 2, alpha=0.2, color='green', label='Feasible region')

# Optimal point
x_star = np.array([0.5, 0.5])
ax.plot(*x_star, 'ko', markersize=10, label=f'$x^* = (0.5, 0.5)$')

# Gradient at optimal point
grad = grad_f(x_star)
ax.arrow(x_star[0], x_star[1], grad[0]*0.5, grad[1]*0.5,
         head_width=0.05, head_length=0.03, fc='black', ec='black')
ax.text(x_star[0] + 0.35, x_star[1] + 0.35, r'$\nabla f(x^*)$', fontsize=12)

ax.set_xlim(-0.5, 2)
ax.set_ylim(-0.5, 2)
```

```
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_aspect('equal')
ax.legend()
ax.set_title('First-Order Optimality: Gradient perpendicular to constraint')
plt.savefig('optimality_condition.pdf', bbox_inches='tight')
plt.show()
```

2.7 Examples of Convex Programs

We now present several important examples of convex optimization problems arising in practice. These examples illustrate how real problems map to the convex framework and provide concrete instances for the algorithms developed in later chapters.

2.7.1 Portfolio Optimization

An investor allocates capital among n assets to balance expected return against risk. Let $x_i \geq 0$ denote the fraction of wealth invested in asset i, and let R_i be the (random) return of asset i. The portfolio return is $R = \sum_i x_i R_i$.

- **Expected return**: $\mathbb{E}[R] = x^T \mu$, where $\mu_i = \mathbb{E}[R_i]$.
- **Risk (variance)**: $\mathrm{Var}(R) = x^T \Sigma x$, where $\Sigma_{ij} = \mathrm{Cov}(R_i, R_j)$.

The covariance matrix Σ is symmetric positive semidefinite, so $x^T \Sigma x$ is convex. The classical Markowitz mean-variance optimization [**?**] minimizes risk for a target return:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & x^T \Sigma x \\ \text{subject to} \quad & x^T \mu \geq r \quad \text{(minimum return)}, \\ & \mathbf{1}^T x = 1 \quad \text{(budget)}, \\ & x \geq 0. \quad \text{(no short-selling)} \end{aligned} \tag{2.9}$$

This is a convex quadratic program (QP). Varying the target return r traces out the *efficient frontier*—the optimal risk-return tradeoff.

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

# Problem data
np.random.seed(42)
n = 5  # Number of assets
```

```
mu = np.array([0.12, 0.10, 0.07, 0.05, 0.03])  # Expected returns

# Generate random positive semidefinite covariance matrix
A = np.random.randn(n, n) * 0.1
Sigma = A @ A.T + 0.01 * np.eye(n)

# Solve for different target returns to trace efficient frontier
target_returns = np.linspace(0.03, 0.12, 50)
risks = []
portfolios = []

for r in target_returns:
    x = cp.Variable(n)
    objective = cp.Minimize(cp.quad_form(x, Sigma))
    constraints = [
        x @ mu >= r,          # Minimum return
        cp.sum(x) == 1,       # Budget constraint (weights sum to 1)
        x >= 0                # No short-selling
    ]
    problem = cp.Problem(objective, constraints)
    problem.solve()

    if problem.status == 'optimal':
        risks.append(np.sqrt(problem.value))  # Standard deviation
        portfolios.append(x.value)
    else:
        risks.append(np.nan)
        portfolios.append(None)

# Plot efficient frontier
plt.figure(figsize=(8, 5))
plt.plot(risks, target_returns, 'b-', linewidth=2)
plt.xlabel('Risk (Standard Deviation)')
plt.ylabel('Expected Return')
plt.title('Efficient Frontier')
plt.grid(True, alpha=0.3)
plt.savefig('efficient_frontier.pdf', bbox_inches='tight')
plt.show()
```

Check Your Understanding 2.7.

In the portfolio optimization problem:

(a) Why is $x^T \Sigma x$ convex?

(b) If we remove the constraint $x \geq 0$ (allow short-selling), is the problem still convex?

(c) If we add a constraint limiting the number of assets to at most k (cardinality constraint), is the problem still convex?

2.7.2 Linear Regression and Regularization

Given data points (a_i, b_i) for $i = 1, \ldots, m$, where $a_i \in \mathbb{R}^n$ are features and $b_i \in \mathbb{R}$ are labels, we seek a linear model $\hat{b} = a^T x$ that predicts labels from features.

Least squares. The ordinary least squares problem minimizes squared prediction error:

$$\begin{array}{ll} \text{minimize} \\ x \in \mathbb{R}^n \end{array} \quad \|Ax - b\|_2^2, \tag{2.10}$$

where $A \in \mathbb{R}^{m \times n}$ has rows a_i^T and $b \in \mathbb{R}^m$. This is an unconstrained convex problem with closed-form solution $x^* = (A^T A)^{-1} A^T b$ (assuming A has full column rank).

Ridge regression. Adding ℓ_2 regularization prevents overfitting and handles ill-conditioned $A^T A$:

$$\begin{array}{ll} \text{minimize} \\ x \in \mathbb{R}^n \end{array} \quad \|Ax - b\|_2^2 + \lambda \|x\|_2^2. \tag{2.11}$$

The solution is $x^* = (A^T A + \lambda I)^{-1} A^T b$, which is always well-defined for $\lambda > 0$.

Lasso regression. Using ℓ_1 regularization encourages sparse solutions [**?**]:

$$\begin{array}{ll} \text{minimize} \\ x \in \mathbb{R}^n \end{array} \quad \|Ax - b\|_2^2 + \lambda \|x\|_1. \tag{2.12}$$

The ℓ_1 norm $\|x\|_1 = \sum_i |x_i|$ is convex but not differentiable. The geometry of the ℓ_1 ball (a polytope with corners on the axes) encourages solutions where some $x_i = 0$—automatic feature selection.

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

# Generate synthetic data with sparse true coefficients
np.random.seed(42)
m, n = 100, 20  # 100 samples, 20 features
x_true = np.zeros(n)
x_true[:5] = [3, -2, 1.5, -1, 0.5]  # Only 5 nonzero coefficients

A = np.random.randn(m, n)
b = A @ x_true + 0.1 * np.random.randn(m)  # Add noise

# Solve for different regularization strengths
```

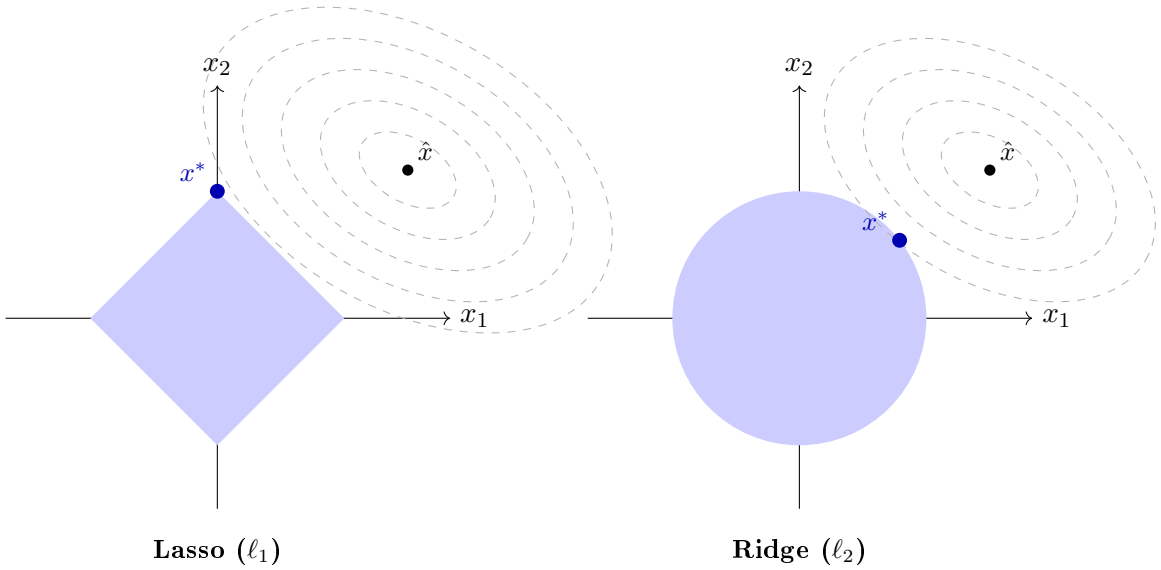

Figure 2.6: Geometry of ℓ_1 (Lasso) vs ℓ_2 (Ridge) regularization. The constraint region is a diamond for Lasso and a disk for Ridge. Ellipses show level sets of the least-squares objective centered at the unconstrained estimate $\hat{x}$. The Lasso solution tends to hit a corner of the diamond, yielding a sparse solution ($x_1 = 0$), while the Ridge solution lies on the smooth boundary with both coefficients nonzero. From [**?**].

```
lambdas = np.logspace(-3, 1, 50)

def solve_lasso(A, b, lam):
    x = cp.Variable(n)
    objective = cp.Minimize(cp.sum_squares(A @ x - b) + lam * cp.norm(x, 1))
    problem = cp.Problem(objective)
    problem.solve()
    return x.value

def solve_ridge(A, b, lam):
    x = cp.Variable(n)
    objective = cp.Minimize(cp.sum_squares(A @ x - b) + lam * cp.sum_squares(x))
    problem = cp.Problem(objective)
    problem.solve()
    return x.value

# Compare sparsity
lasso_solutions = [solve_lasso(A, b, lam) for lam in lambdas]
ridge_solutions = [solve_ridge(A, b, lam) for lam in lambdas]

lasso_nnz = [np.sum(np.abs(x) > 1e-4) for x in lasso_solutions]
ridge_nnz = [np.sum(np.abs(x) > 1e-4) for x in ridge_solutions]
```

```
plt.figure(figsize=(8, 5))
plt.semilogx(lambdas, lasso_nnz, 'b-', label='Lasso', linewidth=2)
plt.semilogx(lambdas, ridge_nnz, 'r--', label='Ridge', linewidth=2)
plt.xlabel('Regularization parameter $\\lambda$')
plt.ylabel('Number of nonzero coefficients')
plt.title('Sparsity: Lasso vs Ridge Regression')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('lasso_vs_ridge.pdf', bbox_inches='tight')
plt.show()
```

The plot reveals a fundamental difference between lasso and ridge regression. Ridge regression (red dashed line) keeps all 20 coefficients nonzero regardless of λ—the ℓ_2 penalty shrinks coefficients toward zero but never exactly to zero. In contrast, lasso (blue solid line) produces increasingly sparse solutions as λ grows: at small λ, it recovers all 20 features, but as regularization increases, coefficients are driven exactly to zero. For $\lambda \approx 0.1$, lasso correctly identifies approximately 5 nonzero coefficients, matching our ground truth. This sparsity-inducing property makes lasso valuable for feature selection and interpretable models.

2.7.3 Logistic Regression

For binary classification with labels $y_i \in \{0, 1\}$ and features $a_i \in \mathbb{R}^n$, we want to predict class membership from features. Unlike linear regression, directly predicting $y_i = w^T a_i + b$ is problematic: the output can be any real number, not a probability in $[0, 1]$.

Logistic regression addresses this by modeling the *probability* of class 1:

$$P(y_i = 1 \mid a_i) = \frac{1}{1 + e^{-(w^T a_i + b)}} = \sigma(w^T a_i + b),$$

where $\sigma(z) = 1/(1 + e^{-z})$ is the sigmoid function. The sigmoid maps any real number to $(0, 1)$, giving a valid probability. The linear combination $w^T a_i + b$ is called the *log-odds*: when it is large and positive, the probability is close to 1; when large and negative, close to 0.

To find the best parameters (w, b), we maximize the likelihood of observing our training data. For independent samples, the log-likelihood is:

$$\ell(w, b) = \sum_{i=1}^{m} \left[y_i \log \sigma(w^T a_i + b) + (1 - y_i) \log(1 - \sigma(w^T a_i + b)) \right].$$

Each term contributes $\log \sigma(\cdot)$ if $y_i = 1$ (we want high probability for class 1) and $\log(1 - \sigma(\cdot))$ if $y_i = 0$ (we want low probability for class 1).

Maximizing ℓ is equivalent to minimizing the negative log-likelihood, called the *logistic loss* or *cross-*

entropy loss:

$$\underset{w \in \mathbb{R}^n,\, b \in \mathbb{R}}{\text{minimize}} \quad \sum_{i=1}^{m} \log\left(1 + e^{-(2y_i - 1)(w^T a_i + b)}\right). \tag{2.13}$$

This is a convex optimization problem because the log-sum-exp function is convex. The convexity guarantees that gradient-based methods find the global optimum, and the resulting classifier is the maximum-likelihood linear separator.

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

# Generate synthetic classification data
np.random.seed(42)
n_per_class = 100

# Class 0: centered at (-1, -1)
X0 = np.random.randn(n_per_class, 2) * 0.8 + np.array([-1, -1])
# Class 1: centered at (1, 1)
X1 = np.random.randn(n_per_class, 2) * 0.8 + np.array([1, 1])

X = np.vstack([X0, X1])
y = np.hstack([np.zeros(n_per_class), np.ones(n_per_class)])

# Shuffle the data
perm = np.random.permutation(len(y))
X, y = X[perm], y[perm]

# Solve logistic regression using CVXPY
n_samples, n_features = X.shape
w = cp.Variable(n_features)
b = cp.Variable()

# Logistic loss: sum of log(1 + exp(-y_i * (w'x_i + b)))
# Convert y from {0,1} to {-1,+1}
y_signed = 2 * y - 1
loss = cp.sum(cp.logistic(-cp.multiply(y_signed, X @ w + b)))

problem = cp.Problem(cp.Minimize(loss))
problem.solve()

print(f"Optimal weights: {w.value}")
print(f"Optimal bias: {b.value}")

# Plot decision boundary
```

```
plt.figure(figsize=(8, 6))
plt.scatter(X[y==0, 0], X[y==0, 1], c='blue', label='Class 0', alpha=0.6)
plt.scatter(X[y==1, 0], X[y==1, 1], c='red', label='Class 1', alpha=0.6)

# Decision boundary: w'x + b = 0
x1_range = np.linspace(-4, 4, 100)
x2_boundary = -(w.value[0] * x1_range + b.value) / w.value[1]
plt.plot(x1_range, x2_boundary, 'k-', linewidth=2, label='Decision boundary')

plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.legend()
plt.title('Logistic Regression')
plt.grid(True, alpha=0.3)
plt.savefig('logistic_regression.pdf', bbox_inches='tight')
plt.show()
```

2.7.4 Support Vector Machines

Given labeled data (a_i, y_i) with $y_i \in \{-1, +1\}$, a support vector machine (SVM) finds the maximum-margin separating hyperplane. The hard-margin SVM solves:

$$\begin{array}{ll} \underset{w \in \mathbb{R}^n, b \in \mathbb{R}}{\text{minimize}} & \frac{1}{2}\|w\|_2^2 \\ \text{subject to} & y_i(w^T a_i - b) \geq 1, \quad i = 1, \ldots, m. \end{array} \tag{2.14}$$

The objective $\|w\|^2$ is convex, and the constraints are linear in (w, b). The margin width is $2/\|w\|$, so minimizing $\|w\|^2$ maximizes the margin.

For non-separable data, the soft-margin SVM introduces slack variables:

$$\begin{array}{ll} \underset{w, b, \xi}{\text{minimize}} & \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^{m} \xi_i \\ \text{subject to} & y_i(w^T a_i - b) \geq 1 - \xi_i, \quad i = 1, \ldots, m, \\ & \xi_i \geq 0, \qquad\quad i = 1, \ldots, m. \end{array} \tag{2.15}$$

The parameter $C > 0$ trades off margin width against classification errors.

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
```

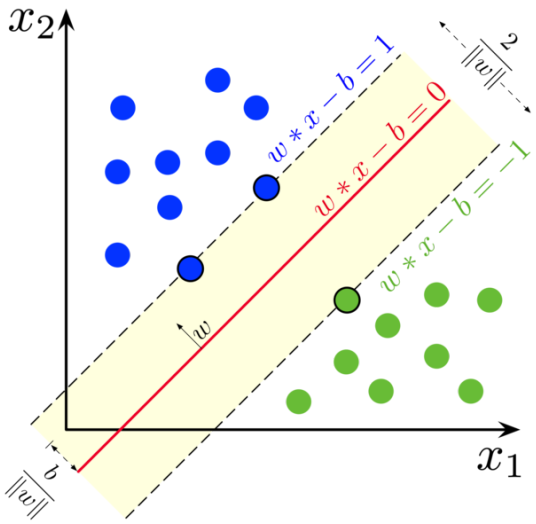

Figure 2.7: Support vector machine: the optimal hyperplane maximizes the margin between classes. Support vectors lie on the margin boundaries.

```python
# Generate linearly separable data
np.random.seed(42)
n_samples = 50

# Class +1
X_pos = np.random.randn(n_samples // 2, 2) + np.array([2, 2])
# Class -1
X_neg = np.random.randn(n_samples // 2, 2) + np.array([-2, -2])

X = np.vstack([X_pos, X_neg])
y = np.hstack([np.ones(n_samples // 2), -np.ones(n_samples // 2)])

# Solve hard-margin SVM
n_features = 2
w = cp.Variable(n_features)
b = cp.Variable()

objective = cp.Minimize(0.5 * cp.sum_squares(w))
constraints = [cp.multiply(y, X @ w - b) >= 1]

problem = cp.Problem(objective, constraints)
problem.solve()

print(f"Optimal w: {w.value}")
print(f"Optimal b: {b.value}")
print(f"Margin width: {2 / np.linalg.norm(w.value):.4f}")
```

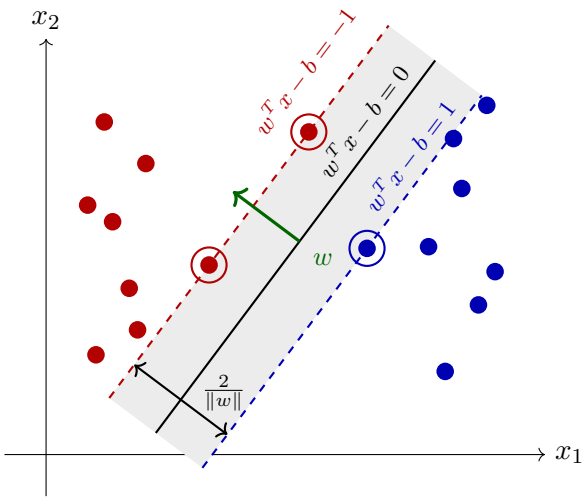

Figure 2.8: Support vector machine geometry. The optimal hyperplane $w^T x - b = 0$ (solid line) maximizes the margin between the two classes. The margin boundaries $w^T x - b = \pm 1$ (dashed lines) define a slab of width $2/\|w\|$. Support vectors (circled points) lie exactly on the margin boundaries and determine the solution. The vector w is perpendicular to the hyperplane.

```
# Find support vectors (points where constraint is tight)
margins = y * (X @ w.value - b.value)
support_vectors = np.abs(margins - 1) < 1e-4

# Plot
plt.figure(figsize=(8, 6))
plt.scatter(X[y==1, 0], X[y==1, 1], c='blue', label='Class +1', alpha=0.6)
plt.scatter(X[y==-1, 0], X[y==-1, 1], c='red', label='Class -1', alpha=0.6)
plt.scatter(X[support_vectors, 0], X[support_vectors, 1],
            s=200, facecolors='none', edgecolors='black', linewidths=2,
            label='Support vectors')

# Decision boundary and margins
x1_range = np.linspace(-5, 5, 100)
x2_boundary = (b.value - w.value[0] * x1_range) / w.value[1]
x2_margin_pos = (b.value + 1 - w.value[0] * x1_range) / w.value[1]
x2_margin_neg = (b.value - 1 - w.value[0] * x1_range) / w.value[1]

plt.plot(x1_range, x2_boundary, 'k-', linewidth=2, label='Decision boundary')
plt.plot(x1_range, x2_margin_pos, 'k--', linewidth=1)
plt.plot(x1_range, x2_margin_neg, 'k--', linewidth=1)

plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.xlabel('$x_1$')
```

```
plt.ylabel('$x_2$')
plt.legend()
plt.title('Support Vector Machine')
plt.savefig('svm.pdf', bbox_inches='tight')
plt.show()
```

Check Your Understanding 2.8.

In the SVM formulation:

(a) Why do we minimize $\|w\|^2$ instead of $\|w\|$?

(b) What is the geometric interpretation of a support vector?

(c) In the soft-margin formulation, what happens as $C \to \infty$?

(d) Is the soft-margin SVM still a convex program?

2.8 Exercises

1. **(Convex sets)** Prove that the intersection of any collection of convex sets is convex.

2. **(Convex functions)** Show that $f(x) = \log(e^{x_1} + \cdots + e^{x_n})$ is convex by verifying the second-order condition.

3. **(Strong convexity)** Let f be m-strongly convex with minimizer x^*. Prove that:

$$f(x) - f(x^*) \geq \frac{m}{2}\|x - x^*\|^2.$$

4. **(Portfolio optimization)** Implement the portfolio optimization example with real stock data. Download historical returns for 10 stocks, estimate μ and Σ, and plot the efficient frontier.

5. **(Regularization path)** For the Lasso problem, implement a path algorithm that solves for all values of λ from $\lambda_{\max}$ (where $x = 0$) down to 0. Plot the coefficients as functions of λ.

6. **(SVM duality)** The dual of the hard-margin SVM is:

$$\begin{aligned} \underset{\alpha \in \mathbb{R}^m}{\text{maximize}} \quad & \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j a_i^T a_j \\ \text{subject to} \quad & \alpha_i \geq 0, \quad i = 1, \ldots, m, \\ & \sum_i \alpha_i y_i = 0. \end{aligned} \tag{2.16}$$

   Implement both primal and dual formulations and verify they give the same solution.

Chapter 3

Duality and KKT Conditions

Every constrained optimization problem has a dual—a companion problem that approaches the same question from the opposite direction. When primal and dual agree, their meeting point yields the KKT conditions: a system of equations that characterizes optimality. This chapter develops Lagrange duality, Fenchel conjugates, and the saddle point perspective, revealing how dual variables serve as shadow prices for constraints. These tools draw form the theoretical backbone of constrained optimization and power algorithms throughout this book.

3.1 Lagrange Duality

In this section we formalize the Lagrangian dual of an optimization problem and provide various properties. We consider a prototypical (not necessarily convex) optimization problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0 \quad i = 1, \ldots, m, \\ & h_i(x) = 0 \quad i = 1, \ldots, p, \end{aligned} \tag{3.1}$$

with optimal value f^*. Our prototypical problem belongs to the category of *constrained optimization* due to the existence of the constraints. The section is adapted from [**?**], with some examples from [**?**].

Example 3.1 (Single inequality constraint)**.** Consider the problem illustrated in Figure 3.1:

$$\begin{aligned} \underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & x_1 + x_2 \\ \text{subject to} \quad & x_1^2 + x_2^2 - 2 \leq 0. \end{aligned} \tag{3.2}$$

The objective is linear with gradient $\nabla f(x) = [1 \quad 1]^T$, which points in the direction of increase as shown in the figure. The feasible region is a disk of radius $\sqrt{2}$ centered at the origin. Since the objective is linear (no curvature to create an interior minimum), the optimum must occur on the boundary of the disk, where a level set of the objective is tangent to the constraint.

To find the optimum, we use the method of Lagrange multipliers. At a boundary point where the constraint is active ($g(x) = 0$), the gradients of objective and constraint must be parallel:

$$\nabla f(x) = \lambda \nabla g(x) \quad \Rightarrow \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix}.$$

This gives $x_1 = x_2 = 1/(2\lambda)$. Substituting into the active constraint $x_1^2 + x_2^2 = 2$:

$$2 \cdot \frac{1}{4\lambda^2} = 2 \quad \Rightarrow \quad \lambda = \pm \frac{1}{2}.$$

Since we need $\lambda \geq 0$ for inequality constraints (as we will see), we have $\lambda = 1/2$, giving $x^* = (-1, -1)$ with optimal value $f^* = -2$.

Two important observations emerge from this example:

1. The unconstrained optimality condition $\nabla f(x) = 0$ does *not* hold for constrained problems.
2. At optimality, we must have $\nabla f(x^*) = \lambda \nabla g(x^*)$ for some $\lambda \geq 0$—the gradient of the objective and the gradient of the active constraint are parallel.

Why must the gradients be parallel? Suppose they weren't. Then $-\nabla f(x)$ (the descent direction) and $-\nabla g(x)$ (the direction into the feasible region) would span a cone of directions that both decrease the objective and remain feasible—contradicting optimality. At a true minimum, the only way to decrease f is to leave the feasible region: the descent direction points directly outward, making ∇f and ∇g parallel.

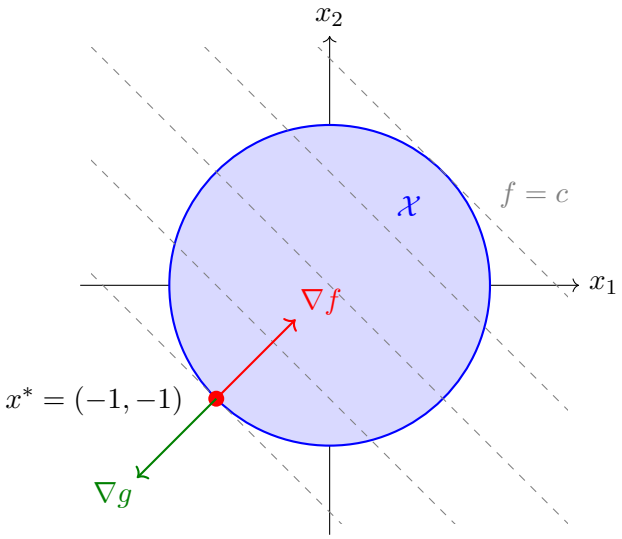

Figure 3.1: Minimizing $f(x) = x_1 + x_2$ over the disk $x_1^2 + x_2^2 \leq 2$. At the optimum $x^* = (-1, -1)$, the gradient of f is parallel to the outward-pointing gradient of the constraint.

Check Your Understanding 3.1.

In Example 3.1, what would happen if we changed the objective to $f(x) = x_1 - x_2$? Find the new optimal solution geometrically and verify using the Lagrange multiplier condition.

3.1.1 The Lagrangian Function

The optimality condition $\nabla f(x^*) + \lambda \nabla g(x^*) = 0$ has a striking interpretation: it is the stationarity condition for the function

$$L(x, \lambda) = f(x) + \lambda g(x).$$

Rather than enforcing the constraint $g(x) \leq 0$ explicitly, we *penalize* violations by adding $\lambda g(x)$ to the objective. The multiplier λ controls the penalty: when λ is large, constraint violations are expensive. At the right choice of λ, minimizing this combined function recovers the constrained optimum—no explicit constraint handling required. This function L is called the **Lagrangian**, and it transforms constrained optimization into a balancing act between objective and constraints.

For the general problem (3.1) with multiple inequality and equality constraints, we introduce the *Lagrangian function*:

$$L(x, \lambda, \mu) \triangleq f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \mu_j h_j(x).$$

The vectors $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p$ are called **Lagrange multipliers** (or **dual variables**)—λ_i for the i-th inequality constraint and μ_j for the j-th equality constraint. We use separate notation because, as we shall see, inequality and equality multipliers obey different sign conditions.

The Lagrangian encodes constraint violations through the constraint functions: $g_i(x) > 0$ indicates the i-th inequality is violated, while $h_j(x) \neq 0$ indicates the j-th equality is violated. Interpreting λ_i as the *price* of violating constraint i, the Lagrangian trades off the objective against constraint violations. For our single-constraint example, $\nabla_x L(x, \lambda) = \nabla f(x) + \lambda \nabla g(x)$, and the stationarity condition $\nabla_x L = 0$ recovers exactly the parallel gradient condition we derived geometrically.

Definition 3.2 (Lagrange dual function)**.** The **Lagrange dual function** $D : \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R} \cup \{-\infty\}$ is defined as:

$$D(\lambda, \mu) = \inf_{x \in \mathcal{X}} L(x, \lambda, \mu) = \inf_{x \in \mathcal{X}} \left[f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \mu_j h_j(x) \right].$$

The dual function has a crucial structural property: it is always concave, regardless of whether the original problem is convex.

Proposition 3.3 (Concavity of the dual function)**.** *The Lagrange dual function $D(\lambda, \mu)$ is concave.*

Proof. For any fixed x, the function $L(x, \lambda, \mu)$ is affine (hence concave) in (λ, μ). The dual function $D(\lambda, \mu) = \inf_x L(x, \lambda, \mu)$ is the pointwise infimum of a family of affine functions. Since the pointwise infimum of concave functions is concave (see [**?**], Section 3.2.3), D is concave. □

This concavity is significant: even if the primal problem (3.1) is nonconvex and difficult to solve, the dual problem of maximizing D is always a concave maximization problem.

3.1.2 Weak and Strong Duality

Theorem 3.4 (Weak duality)**.** *For any $\lambda \geq 0$ and any μ:*

$$D(\lambda, \mu) \leq f^*.$$

Proof. Let x^* be optimal for (3.1), so $g_i(x^*) \leq 0$ and $h_j(x^*) = 0$ for all i, j. Then:

$$\begin{aligned} D(\lambda, \mu) &= \inf_{x \in \mathcal{X}} L(x, \lambda, \mu) \leq L(x^*, \lambda, \mu) \\ &= f(x^*) + \sum_{i=1}^{m} \lambda_i g_i(x^*) + \sum_{j=1}^{p} \mu_j h_j(x^*) \leq f(x^*) = f^*, \end{aligned}$$

where the last inequality uses $\lambda_i \geq 0$, $g_i(x^*) \leq 0$, and $h_j(x^*) = 0$. □

Weak duality is surprisingly powerful: for *any* optimization problem—convex or not—evaluating the dual function at any (λ, μ) with $\lambda \geq 0$ yields a lower bound on f^*. This immediately raises a question: what is the *tightest* lower bound we can obtain? The answer defines the dual problem.

Definition 3.5 (Dual problem)**.** The **Lagrange dual problem** associated with (3.1) is:

$$\begin{aligned} \underset{\lambda, \mu}{\text{maximize}} \quad & D(\lambda, \mu) \\ \text{subject to} \quad & \lambda \geq 0. \end{aligned} \tag{3.3}$$

We denote its optimal value by d^*.

By weak duality, $d^* \leq f^*$ always holds. The difference $f^* - d^*$ is called the **duality gap**. Since D is concave (Proposition 3.3), the dual problem is a concave maximization—a tractable problem even when the primal is not.

Definition 3.6 (Strong duality)**.** We say **strong duality** holds if $d^* = f^*$, i.e., the duality gap is zero.

Strong duality does not hold in general. However, for convex problems satisfying certain regularity conditions—called **constraint qualifications**—strong duality is guaranteed. The most widely used is Slater's condition.

Definition 3.7 (Slater's condition)**.** Problem (3.1) satisfies **Slater's condition** if the problem is convex (i.e., f and g_i are convex, h_j are affine) and there exists a **strictly feasible** point $\tilde{x}$ such that:

$$g_i(\tilde{x}) < 0, \quad i = 1, \ldots, m, \qquad h_j(\tilde{x}) = 0, \quad j = 1, \ldots, p.$$

Slater's condition requires strict feasibility for the inequality constraints—we must satisfy them with room to spare—while the equality constraints need only be satisfied.[1] When there are no inequality constraints ($m = 0$), Slater's condition reduces to ordinary feasibility.

[1] When objective or constraint functions have restricted domains (e.g., $f(x) = -\log x$), a technical refinement requires $\tilde{x}$ to lie in the relative interior of the domain; see [**?**], Section 5.2.3.

Theorem 3.8 (Strong duality for convex problems)**.** *If problem* (3.1) *is convex and Slater's condition holds, then strong duality holds:*

$$d^* = \max_{\lambda \geq 0, \mu} D(\lambda, \mu) = f^*.$$

Moreover, the dual optimal value is attained.

The proof relies on separating hyperplane arguments; see [**?**], Section 5.2.3.

Remark 3.9 (Other constraint qualifications). Slater's condition is sufficient but not necessary for strong duality. The optimization literature has developed numerous alternative constraint qualifications, including the Linear Independence Constraint Qualification (LICQ), Mangasarian–Fromovitz Constraint Qualification (MFCQ), and various constant rank conditions. These become important for nonconvex problems and for understanding the sensitivity of solutions. For a comprehensive treatment, see Bertsekas [**?**], Chapter 3.

Example 3.10 (Quadratic program with linear constraints)**.** Consider the convex quadratic program:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2} x^T Q x + c^T x \\ \text{subject to} \quad & Ax \leq b, \end{aligned} \tag{3.4}$$

where $Q \succ 0$ (positive definite), $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. The Lagrangian is:

$$L(x, \lambda) = \frac{1}{2} x^T Q x + c^T x + \lambda^T (Ax - b).$$

To find the dual function, we minimize over x. Setting $\nabla_x L = Qx + c + A^T \lambda = 0$:

$$x^*(\lambda) = -Q^{-1}(c + A^T \lambda).$$

Substituting back:

$$D(\lambda) = -\frac{1}{2}(c + A^T \lambda)^T Q^{-1}(c + A^T \lambda) - b^T \lambda.$$

The dual problem is to maximize this concave quadratic over $\lambda \geq 0$. Note how the dual inherits structure from the primal: a quadratic primal yields a quadratic dual.

Check Your Understanding 3.2.

For Example 3.10 with $n = 1$, $Q = 2$, $c = -3$, $A = 1$, $b = 1$ (so the primal is $\min x^2 - 3x$ subject to $x \leq 1$):

(a) Solve the primal problem graphically or by calculus.

(b) Compute the dual function $D(\lambda)$ using the formula from Example 3.10.

(c) Maximize $D(\lambda)$ over $\lambda \geq 0$ to find λ^*.

(d) Verify strong duality: does $D(\lambda^*) = f^*$?

Visualizing duality. The following code computes and plots the dual function for a simple quadratic program:

```
import numpy as np
import matplotlib.pyplot as plt

# Primal: min 0.5*x^2 - 3*x  s.t. x <= 1
# Dual: max D(lambda) = -0.5*(A^T*lambda + c)^2/Q - b*lambda
#     = -0.5*(lambda - 3)^2 / 2 - lambda

def primal_objective(x):
    return 0.5 * x**2 - 3*x

def dual_function(lam):
    # D(lambda) = -0.5*(lambda - 3)^2 / 2 - lambda
    return -0.5 * (lam - 3)**2 / 2 - lam

# Solve primal by enumeration
x_vals = np.linspace(-2, 5, 1000)
x_feasible = x_vals[x_vals <= 1]
primal_vals = primal_objective(x_feasible)
x_opt = x_feasible[np.argmin(primal_vals)]
f_opt = primal_objective(x_opt)

print(f"Primal optimal: x* = {x_opt:.4f}, f* = {f_opt:.4f}")

# Solve dual by enumeration
lam_vals = np.linspace(0, 5, 1000)
dual_vals = dual_function(lam_vals)
lam_opt = lam_vals[np.argmax(dual_vals)]
d_opt = dual_function(lam_opt)

print(f"Dual optimal: lambda* = {lam_opt:.4f}, d* = {d_opt:.4f}")
print(f"Duality gap: {f_opt - d_opt:.6f}")

# Plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Primal
ax1.plot(x_vals, primal_objective(x_vals), 'b-', label='$f(x)$')
ax1.axvline(x=1, color='r', linestyle='--', label='Constraint $x \leq 1$')
ax1.fill_betweenx([-3, 3], -2, 1, alpha=0.2, color='green', label='Feasible')
ax1.plot(x_opt, f_opt, 'ko', markersize=10, label=f'$x^* = {x_opt:.2f}$')
ax1.set_xlabel('$x$')
ax1.set_ylabel('$f(x)$')
```

```
ax1.set_title('Primal Problem')
ax1.legend()
ax1.set_xlim(-2, 5)
ax1.set_ylim(-3, 3)
ax1.grid(True, alpha=0.3)

# Dual
ax2.plot(lam_vals, dual_vals, 'b-', label='$D(\\lambda)$')
ax2.axhline(y=f_opt, color='r', linestyle='--', label=f'$f^* = {f_opt:.2f}$')
ax2.plot(lam_opt, d_opt, 'ko', markersize=10, label=f'$\\lambda^* = {lam_opt:.2f}$')
ax2.set_xlabel('$\\lambda$')
ax2.set_ylabel('$D(\\lambda)$')
ax2.set_title('Dual Problem')
ax2.legend()
ax2.set_xlim(0, 5)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('duality_visualization.pdf', bbox_inches='tight')
plt.show()
```

3.1.3 Complementary Slackness

When strong duality holds, the optimal primal and dual solutions satisfy an important relationship.

Theorem 3.11 (Complementary slackness). *Assume strong duality holds, and let x^* and (λ^*, μ^*) be optimal for the primal and dual problems, respectively. Then:*

$$\lambda_i^* g_i(x^*) = 0, \quad i = 1, \ldots, m.$$

Equivalently:

$$\lambda_i^* > 0 \;\Rightarrow\; g_i(x^*) = 0, \qquad g_i(x^*) < 0 \;\Rightarrow\; \lambda_i^* = 0.$$

Proof. By strong duality:

$$\begin{aligned} f(x^*) = D(\lambda^*, \mu^*) &= \inf_x L(x, \lambda^*, \mu^*) \\ &\le L(x^*, \lambda^*, \mu^*) = f(x^*) + \sum_{i=1}^{m} \lambda_i^* g_i(x^*) + \sum_{j=1}^{p} \mu_j^* h_j(x^*) \\ &\le f(x^*), \end{aligned}$$

where the last inequality uses primal feasibility ($g_i(x^*) \le 0$, $h_j(x^*) = 0$) and dual feasibility ($\lambda^* \ge 0$).

Both inequalities must hold with equality, so:

$$\sum_{i=1}^{m} \lambda_i^* g_i(x^*) = 0.$$

Since each term $\lambda_i^* g_i(x^*) \leq 0$ (product of nonnegative and nonpositive), we must have $\lambda_i^* g_i(x^*) = 0$ for all i. □

Interpretation. Complementary slackness has a natural economic interpretation. Think of λ_i^* as the *shadow price* or *marginal value* of relaxing constraint i. If a constraint is not tight ($g_i(x^*) < 0$, meaning we have "slack"), then relaxing it further provides no benefit, so its price is zero ($\lambda_i^* = 0$). Conversely, if a constraint has positive price ($\lambda_i^* > 0$), it must be binding ($g_i(x^*) = 0$)—we are using this resource to its limit.

Check Your Understanding 3.3.

In Example 3.1, we found $x^* = (-1, -1)$ and $\lambda^* = 1/2$. Verify that complementary slackness holds.

3.1.4 The Karush–Kuhn–Tucker (KKT) Conditions

The KKT conditions, first published by Kuhn and Tucker in 1951 [**?**] and independently discovered in Karush's 1939 master's thesis [**?**], provide necessary and sufficient conditions for optimality in constrained optimization.

Definition 3.12 (KKT conditions)**.** A point (x^*, λ^*, μ^*) satisfies the **KKT conditions** for problem (3.1) if:

1. **Stationarity**: $\nabla_x L(x^*, \lambda^*, \mu^*) = \nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla g_i(x^*) + \sum_{j=1}^{p} \mu_j^* \nabla h_j(x^*) = 0$
2. **Complementary slackness**: $\lambda_i^* g_i(x^*) = 0$ for all $i = 1, \ldots, m$
3. **Primal feasibility**: $g_i(x^*) \leq 0$ for all i, and $h_j(x^*) = 0$ for all j
4. **Dual feasibility**: $\lambda_i^* \geq 0$ for all $i = 1, \ldots, m$

Theorem 3.13 (KKT conditions for convex problems)**.** *Consider a convex problem (3.1) where strong duality holds. Then x^* is primal optimal and (λ^*, μ^*) is dual optimal if and only if (x^*, λ^*, μ^*) satisfies the KKT conditions.*

Proof. **(Sufficiency)** Suppose (x^*, λ^*, μ^*) satisfies the KKT conditions. By stationarity, x^* minimizes $L(x, \lambda^*, \mu^*)$ over x (since L is convex in x and the gradient vanishes). Therefore:

$$\begin{aligned} D(\lambda^*, \mu^*) &= \inf_x L(x, \lambda^*, \mu^*) = L(x^*, \lambda^*, \mu^*) \\ &= f(x^*) + \sum_{i=1}^{m} \lambda_i^* g_i(x^*) + \sum_{j=1}^{p} \mu_j^* h_j(x^*) = f(x^*), \end{aligned}$$

where the last equality uses complementary slackness and primal feasibility. Since x^* is primal feasible, (λ^*, μ^*) is dual feasible, and their objective values coincide, both are optimal.

(Necessity) Suppose x^* is primal optimal and (λ^*, μ^*) is dual optimal. Primal and dual feasibility hold by definition. By strong duality:

$$f(x^*) = D(\lambda^*, \mu^*) = \inf_x L(x, \lambda^*, \mu^*) \leq L(x^*, \lambda^*, \mu^*) \leq f(x^*),$$

where the last inequality follows from feasibility. Equality throughout implies:

- x^* minimizes $L(x, \lambda^*, \mu^*)$, so $\nabla_x L(x^*, \lambda^*, \mu^*) = 0$ (stationarity).
- $\sum_i \lambda_i^* g_i(x^*) = 0$, which with $\lambda^* \geq 0$ and $g(x^*) \leq 0$ implies complementary slackness.

□

The power of the KKT conditions lies in reducing constrained optimization to solving a system of equations and inequalities. For convex problems, finding a KKT point is equivalent to finding the global optimum. In this sense, the KKT conditions play the same role for constrained optimization that the stationarity condition $\nabla f(x) = 0$ plays for unconstrained problems.

Example 3.14 (Minimum-norm solution)**.** Consider minimizing $f(x) = \frac{1}{2}\|x\|^2$ subject to $Ax = b$, where $A \in \mathbb{R}^{p \times n}$ has full row rank. The Lagrangian is:

$$L(x, \mu) = \frac{1}{2} x^T x + \mu^T (Ax - b).$$

The KKT conditions are:

$$\begin{aligned} \text{Stationarity:} \quad & x^* + A^T \mu^* = 0 \quad \Rightarrow \quad x^* = -A^T \mu^* \\ \text{Primal feasibility:} \quad & Ax^* = b \end{aligned}$$

Substituting the first into the second: $-AA^T \mu^* = b$, giving $\mu^* = -(AA^T)^{-1} b$ and:

$$x^* = A^T (AA^T)^{-1} b.$$

This is the minimum-norm solution to the underdetermined system $Ax = b$—the point in the affine subspace $\{x : Ax = b\}$ closest to the origin.

Numerical verification of KKT conditions. The following code solves a constrained optimization problem and verifies the KKT conditions:

```
import numpy as np
import cvxpy as cp

# Problem: min 0.5*||x||^2 - c'x  s.t.  Ax <= b
np.random.seed(42)
```

```
n, m = 3, 2
c = np.array([1.0, 2.0, 1.0])
A = np.array([[1, 1, 0], [0, 1, 1]])
b = np.array([1.0, 1.0])

# Solve with CVXPY
x = cp.Variable(n)
objective = cp.Minimize(0.5 * cp.sum_squares(x) - c @ x)
constraints = [A @ x <= b]
problem = cp.Problem(objective, constraints)
problem.solve()

x_opt = x.value
lambda_opt = constraints[0].dual_value

print("Optimal solution:")
print(f"  x* = {x_opt}")
print(f"  lambda* = {lambda_opt}")
print(f"  Optimal value: {problem.value:.6f}")

# Verify KKT conditions
print("\nKKT Verification:")

# 1. Stationarity: grad_f(x*) + A'*lambda* = 0
grad_f = x_opt - c  # gradient of 0.5*||x||^2 - c'x
stationarity_residual = grad_f + A.T @ lambda_opt
print(f"  Stationarity residual: {np.linalg.norm(stationarity_residual):.2e}")

# 2. Complementary slackness: lambda_i * (Ax - b)_i = 0
slack = A @ x_opt - b
comp_slack = lambda_opt * slack
print(f"  Complementary slackness: {comp_slack}")

# 3. Primal feasibility: Ax <= b
print(f"  Constraint values (Ax - b): {slack}")
print(f"  Primal feasible: {np.all(slack <= 1e-6)}")

# 4. Dual feasibility: lambda >= 0
print(f"  Dual feasible (lambda >= 0): {np.all(lambda_opt >= -1e-6)}")
```

3.1.5 Sensitivity Analysis and Shadow Prices

The dual variables have a natural interpretation as *sensitivity coefficients* or *shadow prices*. Consider perturbing the right-hand sides of the constraints:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \le u_i, \quad i = 1, \dots, m, \\ & h_j(x) = v_j, \quad j = 1, \dots, p, \end{aligned} \tag{3.5}$$

and let $f^*(u, v)$ denote the optimal value as a function of the perturbations, with $f^*(0, 0)$ being the optimal value of the original problem.

Theorem 3.15 (Local sensitivity)**.** *Assume strong duality holds for the unperturbed problem and let (λ^*, μ^*) be optimal dual variables. Then for small perturbations (u, v):*

$$f^*(u, v) \ge f^*(0, 0) - (\lambda^*)^T u - (\mu^*)^T v,$$

with equality when $(u, v) = (0, 0)$. If f^* is differentiable at $(0, 0)$, then:

$$\left.\frac{\partial f^*}{\partial u_i}\right|_{u=0,v=0} = -\lambda_i^*, \qquad \left.\frac{\partial f^*}{\partial v_j}\right|_{u=0,v=0} = -\mu_j^*.$$

The inequality always holds; the exact derivative interpretation requires f^* to be smooth, which is typical for problems where the active constraints don't change under small perturbations.

Interpretation. The theorem tells us:

- Relaxing inequality i by ϵ (setting $u_i = \epsilon > 0$) decreases the optimal value by approximately $\lambda_i^* \epsilon$.
- If $\lambda_i^* = 0$, the constraint has slack and small relaxations have no first-order effect.
- Large λ_i^* indicates the constraint is "expensive"—relaxing it significantly improves the objective.

This is why λ_i^* is called a *shadow price*: it measures the marginal value of relaxing constraint i.

Check Your Understanding 3.4.

A factory maximizes profit subject to three resource constraints. The optimal dual variables are $\lambda_1^* = 5$, $\lambda_2^* = 0$, $\lambda_3^* = 3$.

(a) Which resource is not fully utilized? How do you know?

(b) If you could purchase one additional unit of any resource, which would you choose?

(c) If resource 1 becomes 10% more available (say, from 100 to 110 units), approximately how much would profit increase?

3.2 Application: Water-Filling

We apply the KKT conditions to a classical problem from information theory. A base station transmits signals to n mobile users over channels with different qualities. From information theory, the achievable rate for user i is:

$$r_i = \log(1 + p_i/N_i),$$

where $p_i \geq 0$ is the transmission power allocated to user i and $N_i > 0$ is a channel quality parameter (noise level). The goal is to maximize total rate subject to a power budget:

$$\begin{aligned} \underset{p \in \mathbb{R}^n}{\text{maximize}} \quad & \sum_{i=1}^{n} \log(1 + p_i/N_i) \\ \text{subject to} \quad & p_i \geq 0, \quad i = 1, \ldots, n, \\ & \sum_{i=1}^{n} p_i = P, \end{aligned} \tag{3.6}$$

where $P > 0$ is the total available power. This is equivalent to the minimization problem:

$$\begin{aligned} \underset{p \in \mathbb{R}^n}{\text{minimize}} \quad & -\sum_{i=1}^{n} \log(1 + p_i/N_i) \\ \text{subject to} \quad & -p_i \leq 0, \quad i = 1, \ldots, n, \\ & \sum_{i=1}^{n} p_i - P = 0. \end{aligned} \tag{3.7}$$

The Lagrangian is:

$$L(p, \lambda, \mu) = -\sum_{i=1}^{n} \log(1 + p_i/N_i) - \sum_{i=1}^{n} \lambda_i p_i + \mu \left(\sum_{i=1}^{n} p_i - P \right).$$

KKT conditions.

1. **Stationarity**: $\frac{\partial L}{\partial p_i} = -\frac{1}{N_i + p_i} - \lambda_i + \mu = 0$, giving $\lambda_i = \mu - \frac{1}{N_i + p_i}$.
2. **Complementary slackness**: $\lambda_i p_i = 0$ for all i.
3. **Primal feasibility**: $p_i \geq 0$ and $\sum_i p_i = P$.
4. **Dual feasibility**: $\lambda_i \geq 0$ for all i.

Substituting stationarity into complementary slackness:

$$p_i \left(\mu - \frac{1}{N_i + p_i} \right) = 0, \quad i = 1, \ldots, n. \tag{3.8}$$

For each i, either $p_i = 0$ or $p_i = \frac{1}{\mu} - N_i$. We analyze two cases:

Case 1: If $\mu < 1/N_i$, then choosing $p_i = 0$ gives $\lambda_i = \mu - 1/N_i < 0$, violating dual feasibility. Hence $p_i = \frac{1}{\mu} - N_i > 0$.

Case 2: If $\mu \geq 1/N_i$, then choosing $p_i > 0$ would require $\mu = \frac{1}{N_i + p_i} < \frac{1}{N_i}$, a contradiction. Hence $p_i = 0$.

Combining both cases:

$$p_i^* = \max\left\{0, \frac{1}{\mu} - N_i\right\} = \left(\frac{1}{\mu} - N_i\right)^+, \tag{3.9}$$

where $(x)^+ = \max\{0, x\}$. The parameter μ (the "water level") is determined by the power constraint:

$$\sum_{i=1}^{n} \left(\frac{1}{\mu} - N_i\right)^+ = P.$$

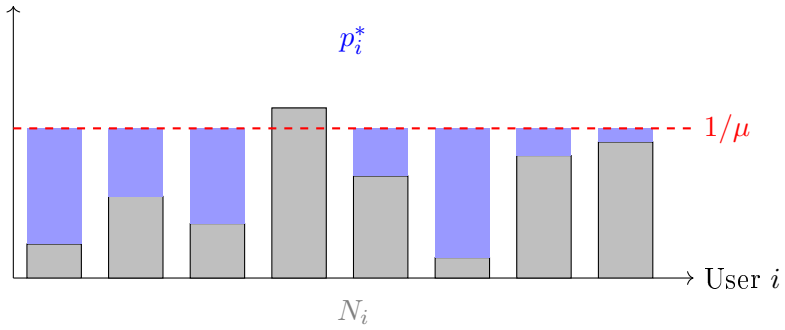

Figure 3.2: Water-filling solution. The noise levels N_i form a "terrain," and power is allocated like water filling to a common level $1/\mu$. Channels with high noise ($N_i > 1/\mu$) receive no power.

The solution has an intuitive "water-filling" interpretation (Figure 3.2): imagine the noise levels N_i as the bottom of containers, and pour water (power) until the total volume equals P. The water level $1/\mu$ is uniform across all channels that receive power, and poor channels (high N_i) may receive no power at all.

```
import numpy as np
import matplotlib.pyplot as plt

def water_filling(N, P):
    """
    Solve the water-filling problem.
    N: array of noise levels
    P: total power budget
    Returns: optimal power allocation
    """
    n = len(N)

    # Binary search on water level directly
    W_low = min(N)       # At this level, no power allocated
```

```
    W_high = max(N) + P  # Upper bound: enough to cover all channels

    for _ in range(100):  # Sufficient iterations for convergence
        W = (W_low + W_high) / 2
        p = np.maximum(0, W - N)
        total_power = np.sum(p)

        if total_power > P:
            W_high = W
        else:
            W_low = W

    water_level = W
    p_opt = np.maximum(0, water_level - N)

    # Small adjustment to exactly meet power constraint
    if np.sum(p_opt) > 0:
        p_opt = p_opt * P / np.sum(p_opt)

    return p_opt, water_level

# Example
np.random.seed(44)
n = 8
N = 0.5 * np.array([0.5, 1.2, 0.8, 2.5, 1.5, 0.3, 1.8, 2.0])
P = 5.0

p_opt, water_level = water_filling(N, P)

print("Noise levels:", N)
print("Optimal allocation:", np.round(p_opt, 3))
print(f"Water level: {water_level:.3f}")
print(f"Total power: {np.sum(p_opt):.3f}")
print(f"Total rate: {np.sum(np.log(1 + p_opt/N)):.3f}")

# Verify KKT conditions
print("\nKKT verification:")
for i in range(n):
    if p_opt[i] > 1e-6:
        print(f" Channel {i}: p={p_opt[i]:.3f}, N+p={N[i]+p_opt[i]:.3f}, " +
            f"should equal 1/mu={water_level:.3f}")
    else:
        print(f" Channel {i}: p=0, N={N[i]:.3f} > water_level={water_level:.3f}")
```

```
# Plot
fig, ax = plt.subplots(figsize=(10, 5))
x = np.arange(n)
width = 0.6

ax.bar(x, N, width, label='Noise $N_i$', color='gray', alpha=0.7)
ax.bar(x, p_opt, width, bottom=N, label='Power $p_i^*$', color='blue', alpha=0.7)
ax.axhline(y=water_level, color='red', linestyle='--', linewidth=2, label=f'Water level $1/\\mu
    = {water_level:.2f}$')

ax.set_xlabel('Channel $i$')
ax.set_ylabel('Level')
ax.set_title('Water-Filling Power Allocation')
ax.legend()
ax.set_xticks(x)

plt.tight_layout()
plt.savefig('water_filling.pdf', bbox_inches='tight')
plt.show()
```

Check Your Understanding 3.5.

In the water-filling problem:

(a) If all channels have the same noise level $N_i = N$, what is the optimal allocation?

(b) What happens to the water level $1/\mu$ as the power budget P increases?

(c) If we add a constraint $p_i \leq p_{\max}$ (maximum power per channel), how would you modify the KKT analysis?

3.3 Convex Conjugates and Fenchel Duality

Lagrange duality arises from a specific problem formulation. A more fundamental duality theory, based on convex conjugates, reveals deeper structure and leads to powerful algorithms. This section introduces Fenchel duality, which will be essential for understanding primal-dual algorithms in later chapters.

3.3.1 The Convex Conjugate

Definition 3.16 (Convex conjugate)**.** The **convex conjugate** (or **Fenchel conjugate**) of a function $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ is:

$$f^*(y) = \sup_{x \in \mathbb{R}^n} \{\langle y, x \rangle - f(x)\}.$$

The conjugate f^* maps a "slope" y to the maximum gap between the linear function $x \mapsto \langle y, x\rangle$ and $f(x)$. Figure 3.3 illustrates this geometrically: given a slope y, we seek the linear function $\ell(x) = \langle y, x\rangle - c$ that lies entirely below $f(x)$ with the largest possible intercept $-c$. The supremum is achieved when the line is tangent to f, touching at some point x^*. At this point:

$$f^*(y) = \langle y, x^*\rangle - f(x^*),$$

and $-f^*(y)$ is the y-intercept of this tangent line.

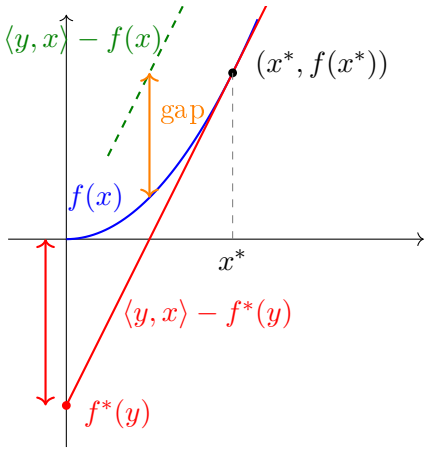

Figure 3.3: Geometric interpretation of the convex conjugate for $f(x) = \frac{1}{2}x^2$ and slope $y = 2$. The gap $\langle y, x\rangle - f(x)$ is the vertical distance from the curve to the dashed line. This gap is maximized at $x^* = 2$ where it equals $f^*(y) = 2$. The supporting hyperplane (red) lies below f everywhere, touching at x^*.

Before presenting examples, we introduce an important function that appears throughout optimization.

Definition 3.17 (Indicator function)**.** The **indicator function** of a set $\mathcal{C} \subseteq \mathbb{R}^n$ is:

$$\delta_{\mathcal{C}}(x) = \begin{cases} 0 & \text{if } x \in \mathcal{C} \\ +\infty & \text{if } x \notin \mathcal{C}. \end{cases}$$

The indicator function encodes constraints: minimizing $f(x) + \delta_{\mathcal{C}}(x)$ is equivalent to minimizing $f(x)$ subject to $x \in \mathcal{C}$. Points outside $\mathcal{C}$ have infinite cost and are effectively forbidden.

Example 3.18 (Conjugates of common functions)**.** Table 3.1 summarizes conjugates that appear frequently. We derive two important cases.

Quadratic. Let $f(x) = \frac{1}{2}x^TQx$ with $Q \succ 0$. To find $f^*(y)$, we maximize $\langle y, x\rangle - \frac{1}{2}x^TQx$ over x. Setting the gradient to zero:

$$y - Qx = 0 \quad \Rightarrow \quad x^* = Q^{-1}y.$$

Substituting back:

$$f^*(y) = \langle y, Q^{-1}y\rangle - \frac{1}{2}(Q^{-1}y)^TQ(Q^{-1}y) = y^TQ^{-1}y - \frac{1}{2}y^TQ^{-1}y = \frac{1}{2}y^TQ^{-1}y.$$

The conjugate of a quadratic with matrix Q is a quadratic with matrix Q^{-1}.

Indicator function. Let $f(x) = \delta_{\mathcal{C}}(x)$ for a convex set $\mathcal{C}$. Then:

$$f^*(y) = \sup_x \{\langle y, x\rangle - \delta_{\mathcal{C}}(x)\} = \sup_{x\in\mathcal{C}} \langle y, x\rangle \triangleq \sigma_{\mathcal{C}}(y).$$

The conjugate of an indicator function is the **support function** $\sigma_{\mathcal{C}}(y)$, which measures how far the set $\mathcal{C}$ extends in direction y.

Table 3.1: Convex conjugates of common functions.

Function $f(x)$	**Domain**	**Conjugate** $f^*(y)$
$\frac{1}{2}x^TQx, \quad Q \succ 0$	$\mathbb{R}^n$	$\frac{1}{2}y^TQ^{-1}y$
$\frac{1}{2}\|x\|_2^2$	$\mathbb{R}^n$	$\frac{1}{2}\|y\|_2^2$
$\|x\|$ (any norm)	$\mathbb{R}^n$	$\delta_{\{\|y\|_* \le 1\}}(y)$
$\|x\|_1$	$\mathbb{R}^n$	$\delta_{\{\|y\|_\infty \le 1\}}(y)$
$\delta_{\mathcal{C}}(x)$ (indicator)	$\mathcal{C}$	$\sigma_{\mathcal{C}}(y) = \sup_{x\in\mathcal{C}}\langle y, x\rangle$
$\delta_{\{b\}}(x)$ (singleton)	$\{b\}$	$\langle b, y\rangle$
$\delta_{\{x\le b\}}(x)$	$\{x : x \le b\}$	$\begin{cases}\langle b, y\rangle & y \ge 0\\ +\infty & \text{otherwise}\end{cases}$
e^x	$\mathbb{R}$	$\begin{cases} y\log y - y & y > 0\\ 0 & y = 0\\ +\infty & y < 0\end{cases}$
$-\log x$	$x > 0$	$-1 - \log(-y)$ for $y < 0$

The conjugate has several fundamental properties.

Proposition 3.19 (Properties of the conjugate)**.** *Let $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ be a function.*

1. ***Convexity**: f^* is always convex, even if f is not (as a supremum of affine functions).*
2. ***Fenchel's inequality**: $f(x) + f^*(y) \ge \langle y, x\rangle$ for all x, y.*
3. ***Double conjugate**: If f is convex and lower semicontinuous, then $f^{**} = f$.*
4. ***Subdifferential correspondence**: For convex f:*

$$y \in \partial f(x) \quad \Leftrightarrow \quad x \in \partial f^*(y) \quad \Leftrightarrow \quad f(x) + f^*(y) = \langle y, x\rangle.$$

Property 4 is particularly important for algorithms. For smooth functions, $\partial f(x) = \{\nabla f(x)\}$, and the correspondence becomes:

$$y = \nabla f(x) \quad \Leftrightarrow \quad x = \nabla f^*(y).$$

The gradient of the conjugate is the *inverse* of the gradient of f. This is why Q^{-1} appears in the conjugate of a quadratic with matrix Q: if $\nabla f(x) = Qx = y$, then $x = Q^{-1}y = \nabla f^*(y)$.

Remark 3.20 (Lower semicontinuity). A function is **lower semicontinuous** if its sublevel sets $\{x : f(x) \leq \alpha\}$ are closed for all α. This is a mild technical condition satisfied by all continuous functions and by indicator functions of closed sets. We assume it throughout without further mention.

Check Your Understanding 3.6.

Compute the convex conjugate of:

(a) $f(x) = |x|$ *(absolute value). Hint: what is the dual norm of* $\|\cdot\|_1$*?*

(b) $f(x) = \frac{1}{2}\|x\|_2^2 + \delta_{\{x:x\geq 0\}}(x)$ *(quadratic on the positive orthant).*

(c) $f(x) = -\log x$ *for* $x > 0$*. Hint: find where the supremum is attained by differentiation, then verify the constraint* $x > 0$ *is satisfied.*

3.3.2 Fenchel Duality

Fenchel duality provides a general framework for deriving dual problems, encompassing Lagrange duality as a special case.

Theorem 3.21 (Fenchel duality)**.** *Consider the primal problem:*

$$p^* = \inf_x \{f(x) + g(Ax)\}, \tag{3.10}$$

where $f : \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ *and* $g : \mathbb{R}^m \to \mathbb{R} \cup \{+\infty\}$ *are convex, and* $A \in \mathbb{R}^{m\times n}$*. The* ***Fenchel dual*** *is:*

$$d^* = \sup_y \left\{-f^*(-A^T y) - g^*(y)\right\}. \tag{3.11}$$

Weak duality $d^* \leq p^*$ *always holds. Strong duality* ($d^* = p^*$) *holds when there exists a point* x *in the domain of* f *such that* Ax *is in the domain of* g *where* g *is continuous.*[2]

Proof sketch. The key is to replace g with its double conjugate representation. For any z:

$$g(z) = \sup_y \{\langle y, z\rangle - g^*(y)\}.$$

Substituting $z = Ax$:

$$\begin{aligned} p^* &= \inf_x \{f(x) + g(Ax)\} \\ &= \inf_x \sup_y \{f(x) + \langle y, Ax\rangle - g^*(y)\} \\ &= \inf_x \sup_y \left\{f(x) + \langle A^T y, x\rangle - g^*(y)\right\}. \end{aligned}$$

[2]More precisely, strong duality holds if $\operatorname{dom} f \cap A^{-1}(\operatorname{dom} g) \neq \emptyset$ and g is continuous at some point in $A(\operatorname{dom} f)$. For indicator functions $g = \delta_{\mathcal{C}}$, this reduces to requiring Ax in the interior of $\mathcal{C}$—analogous to Slater's condition.

By weak duality (min-max $\geq$ max-min):

$$\begin{aligned} p^* &\geq \sup_y \inf_x \left\{ f(x) + \langle A^T y, x \rangle - g^*(y) \right\} \\ &= \sup_y \left\{ - \sup_x \left\{ \langle -A^T y, x \rangle - f(x) \right\} - g^*(y) \right\} \\ &= \sup_y \left\{ -f^*(-A^T y) - g^*(y) \right\} = d^*. \end{aligned}$$

The constraint qualification ensures the inequality is tight. □

Example 3.22 (Lagrange duality as Fenchel duality)**.** Consider the constrained problem $\min_x f(x)$ subject to $Ax \leq b$. We encode the constraint using an indicator function:

$$\min_x \left\{ f(x) + \delta_{\{z : z \leq b\}}(Ax) \right\}.$$

Here $g(z) = \delta_{\{z:z\leq b\}}(z)$. From Table 3.1:

$$g^*(y) = \begin{cases} \langle b, y \rangle & \text{if } y \geq 0 \\ +\infty & \text{otherwise.} \end{cases}$$

The Fenchel dual (3.11) becomes:

$$\sup_{y \geq 0} \left\{ -f^*(-A^T y) - b^T y \right\}.$$

Using the definition of f^*:

$$-f^*(-A^T y) = -\sup_x \left\{ \langle -A^T y, x \rangle - f(x) \right\} = \inf_x \left\{ f(x) + \langle A^T y, x \rangle \right\} = \inf_x L(x, y),$$

where $L(x, y) = f(x) + y^T(Ax - b)$ is the Lagrangian. Hence the Fenchel dual is:

$$\sup_{y \geq 0} \inf_x L(x, y) = d^*,$$

which is exactly the Lagrange dual.

3.3.3 Saddle Points

Duality is intimately connected to saddle point problems. This connection is fundamental for primal-dual algorithms.

Definition 3.23 (Saddle point)**.** A point (x^*, y^*) is a **saddle point** of $L : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ if:

$$L(x^*, y) \leq L(x^*, y^*) \leq L(x, y^*) \quad \text{for all } x \in \mathcal{X}, y \in \mathcal{Y}.$$

Equivalently, x^* minimizes $L(\cdot, y^*)$ and y^* maximizes $L(x^*, \cdot)$.

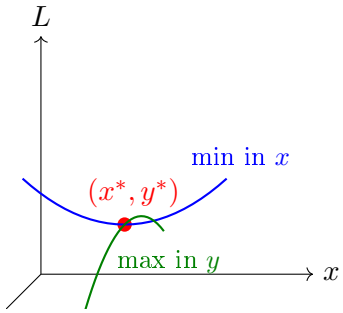

Figure 3.4: A saddle point: minimum along x (blue curve), maximum along y (green curve).

The saddle point characterization connects directly to strong duality.

Theorem 3.24 (Saddle points and strong duality)**.** *Consider* $\inf_x \sup_y L(x, y)$*. The following are equivalent:*

1. (x^*, y^*) *is a saddle point of* L*.*
2. *Strong duality holds:* $\inf_x \sup_y L(x, y) = L(x^*, y^*) = \sup_y \inf_x L(x, y)$*.*

Connection to constrained optimization. For the constrained problem (3.1), consider the Lagrangian $L(x, \lambda, \mu) = f(x) + \sum_i \lambda_i g_i(x) + \sum_j \mu_j h_j(x)$. What happens when we maximize over the dual variables?

For a feasible point x (where $g_i(x) \leq 0$ and $h_j(x) = 0$):

- The terms $\lambda_i g_i(x) \leq 0$ for $\lambda_i \geq 0$, maximized at $\lambda_i = 0$.
- The terms $\mu_j h_j(x) = 0$ regardless of μ_j.

So $\sup_{\lambda \geq 0, \mu} L(x, \lambda, \mu) = f(x)$.

For an infeasible point x (where some $g_i(x) > 0$ or $h_j(x) \neq 0$):

- If $g_i(x) > 0$, taking $\lambda_i \to +\infty$ makes $L \to +\infty$.
- If $h_j(x) \neq 0$, taking $\mu_j \to \pm\infty$ (with appropriate sign) makes $L \to +\infty$.

So $\sup_{\lambda \geq 0, \mu} L(x, \lambda, \mu) = +\infty$.

Combining:

$$\sup_{\lambda \geq 0, \mu} L(x, \lambda, \mu) = \begin{cases} f(x) & \text{if } x \text{ is feasible} \\ +\infty & \text{otherwise.} \end{cases}$$

Therefore:

$$\inf_x \sup_{\lambda \geq 0, \mu} L(x, \lambda, \mu) = \inf_{x \text{ feasible}} f(x) = f^*.$$

This is the primal problem. The dual problem is:

$$\sup_{\lambda \geq 0, \mu} \inf_x L(x, \lambda, \mu) = d^*.$$

Strong duality means these are equal—equivalently, that the Lagrangian has a saddle point.

3.3.4 Primal-Dual Formulation

The saddle point perspective leads directly to primal-dual algorithms. Consider the composite problem:

$$\min_x \{f(x) + g(Ax)\}. \tag{3.12}$$

This structure appears throughout optimization: f captures a regularizer or simple objective, g encodes constraints or a loss function, and A couples variables.

Using the conjugate representation $g(z) = \sup_y\{\langle y, z\rangle - g^*(y)\}$, we rewrite (5.5) as a saddle point problem:

$$\min_x \max_y \underbrace{f(x) + \langle Ax, y\rangle - g^*(y)}_{L(x,y)}. \tag{3.13}$$

This is the **primal-dual formulation**: minimize over the primal variable x, maximize over the dual variable y.

Optimality conditions. At a saddle point (x^*, y^*), the function L is minimized in x and maximized in y. For convex-concave L, this requires:

$$0 \in \partial_x L(x^*, y^*) = \partial f(x^*) + A^T y^*, \tag{3.14}$$

$$0 \in -\partial_y(-L(x^*, y^*)) = Ax^* - \partial g^*(y^*). \tag{3.15}$$

The first condition says $-A^T y^* \in \partial f(x^*)$. By the subdifferential correspondence (Proposition 3.19.4), this is equivalent to $x^* \in \partial f^*(-A^T y^*)$.

The second condition says $Ax^* \in \partial g^*(y^*)$. Again by the subdifferential correspondence, this is equivalent to $y^* \in \partial g(Ax^*)$.

These conditions have a symmetric structure that algorithms can exploit: update x using the gradient/subgradient of f and information from y; update y using the gradient/subgradient of g^* and information from x.

Example 3.25 (Equality-constrained optimization)**.** Consider $\min_x f(x)$ subject to $Ax = b$. We write this as:

$$\min_x \left\{f(x) + \delta_{\{b\}}(Ax)\right\},$$

where $g(z) = \delta_{\{b\}}(z)$ is the indicator of the singleton $\{b\}$. From Table 3.1, $g^*(y) = \langle b, y\rangle$. The primal-dual formulation (3.13) becomes:

$$\min_x \max_y \{f(x) + \langle Ax, y\rangle - \langle b, y\rangle\} = \min_x \max_y \left\{f(x) + y^T(Ax - b)\right\}.$$

This is exactly the Lagrangian for equality-constrained optimization, and y plays the role of the Lagrange multiplier μ.

The optimality conditions (3.14)–(3.15) become:

$$0 \in \partial f(x^*) + A^T y^* \quad \text{(stationarity)},$$
$$0 = Ax^* - b \quad \text{(primal feasibility)}.$$

The second condition uses $\partial g^*(y) = \partial(\langle b, y\rangle) = \{b\}$.

Example 3.26 (Inequality-constrained optimization)**.** Consider $\min_x f(x)$ subject to $Ax \le b$. Now $g(z) = \delta_{\{z \le b\}}(z)$, and from Table 3.1:

$$g^*(y) = \begin{cases} \langle b, y\rangle & y \ge 0 \\ +\infty & \text{otherwise.} \end{cases}$$

The primal-dual formulation is:

$$\min_x \max_{y \ge 0} \left\{ f(x) + y^T(Ax - b) \right\}.$$

The constraint $y \ge 0$ emerges naturally from the structure of g^*—it is not imposed externally but arises because $g^*(y) = +\infty$ for $y \not\ge 0$.

Why this structure matters. The primal-dual formulation (3.13) separates the problem into three components:

- $f(x)$: often simple (e.g., $\frac{1}{2}\|x\|^2$, $\|x\|_1$), with easy proximal operator.
- $g^*(y)$: the conjugate of the "complicated" part, often also simple.
- $\langle Ax, y\rangle$: bilinear coupling, handled by gradient steps.

Algorithms like Chambolle–Pock and PDHG alternate between:

$$x^{k+1} = \arg\min_x \left\{ f(x) + \langle Ax, y^k\rangle + \frac{1}{2\tau}\|x - x^k\|^2 \right\},$$
$$y^{k+1} = \arg\max_y \left\{ \langle Ax^{k+1}, y\rangle - g^*(y) - \frac{1}{2\sigma}\|y - y^k\|^2 \right\}.$$

When f and g^* have simple proximal operators, each step is cheap. This is the foundation of modern large-scale optimization methods, developed in Chapter 5.

Check Your Understanding 3.7.

For the saddle point problem $\min_x \max_y \{x^2 - y^2 + 2xy\}$*:*

(a) Find all stationary points by setting $\partial L/\partial x = 0$ *and* $\partial L/\partial y = 0$*.*

(b) Is the stationary point a saddle point? Check the definition directly.

(c) Is $L(x, y) = x^2 - y^2 + 2xy$ *convex in* x *for fixed* y*? Concave in* y *for fixed* x*? What does this imply about whether a saddle point exists?*

3.4 Summary

This chapter introduced duality in optimization: the Lagrangian function, weak and strong duality, KKT conditions, and the Fenchel conjugate. These concepts provide both theoretical insight and practical tools. The KKT conditions reduce constrained optimization to solving a system of equations—as demonstrated by the water-filling example, where we derived a closed-form solution. Dual variables serve as shadow prices that quantify the value of relaxing constraints, enabling sensitivity analysis. The saddle point perspective reformulates optimization as a min-max problem, opening the door to primal-dual algorithms.

These ideas are foundational for algorithms throughout this book. Gradient methods on the dual enable decomposition across processors. The saddle point structure underlies Chambolle-Pock and PDLP. Interior point methods follow paths defined by KKT conditions. Even branch-and-bound relies on LP duality for bounds. Duality is not just theory—it is the engine behind scalable optimization.

3.5 Exercises

1. **(Weak duality)** Prove that weak duality holds even when the primal problem is nonconvex. Give an example of a nonconvex problem where the duality gap is strictly positive.

2. **(KKT conditions)** Solve the following problem using KKT conditions:

$$\begin{aligned} \underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & \frac{1}{2}(x_1^2 + x_2^2) \\ \text{subject to} \quad & x_1 + 2x_2 \leq 1, \\ & x_1 - x_2 \leq 2, \\ & x_1, x_2 \geq 0. \end{aligned}$$

3. **(Slater's condition)** For each problem, determine whether Slater's condition holds:

   (a) $\min\{x_1 + x_2 : x_1^2 + x_2^2 \leq 1\}$

   (b) $\min\{x_1 + x_2 : x_1^2 + x_2^2 \leq 0\}$

   (c) $\min\{x_1 + x_2 : x_1^2 + x_2^2 = 1\}$

4. **(Dual of the dual)** Show that the dual of the Lagrange dual problem is the original primal problem (for convex problems with strong duality).

5. **(Conjugate computation)** Compute the convex conjugate of:

   (a) $f(x) = \|x\|_1$

   (b) $f(x) = \|x\|_\infty$

   (c) $f(x) = \max\{0, 1 - x\} + \max\{0, 1 + x\}$ (hinge loss)

   (d) $f(x) = x \log x - x$ for $x > 0$ (negative entropy)

6. **(Fenchel duality)** Consider the problem $\min_x\{\|x\|_1 + \frac{1}{2}\|Ax - b\|_2^2\}$.

   (a) Write this in the form $\min_x\{f(x) + g(Ax)\}$ and identify f and g.

   (b) Derive the Fenchel dual problem.

   (c) Show that strong duality holds.

7. **(Water-filling generalization)** Extend the water-filling analysis to include:

   (a) Individual power constraints $p_i \le p_{\max,i}$.

   (b) A minimum rate requirement $\log(1 + p_i/N_i) \ge r_{\min}$ for selected users.

8. **(Sensitivity analysis)** A company solves the LP:

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 10, \\ & 2x_1 + x_2 \le 16, \\ & x_1, x_2 \ge 0. \end{aligned}$$

   The optimal solution is $x^* = (6, 4)$ with optimal value 26, and dual variables $\lambda_1^* = 1$, $\lambda_2^* = 1$.

   (a) If the first constraint is relaxed to $x_1 + x_2 \le 11$, approximately what is the new optimal value?

   (b) For what range of right-hand side values does the dual solution remain optimal?

9. **(Implementation)** Implement a primal-dual algorithm for the saddle point problem:

$$\min_x \max_y \left\{ \frac{1}{2}\|x\|^2 + y^T(Ax - b) \right\}$$

   using alternating gradient descent-ascent:

$$\begin{aligned} x^{k+1} &= x^k - \alpha(x^k + A^T y^k) \\ y^{k+1} &= y^k + \alpha(Ax^{k+1} - b) \end{aligned}$$

   Test on a random system and compare convergence for different step sizes α.

10. **(Duality gap)** For the nonconvex problem $\min\{-x^2 : -1 \le x \le 1\}$:

    (a) Find the optimal value f^*.

    (b) Write the Lagrangian and dual function.

    (c) Find the dual optimal value d^*.

    (d) Compute the duality gap $f^* - d^*$.

Part II

Convex Algorithms

Chapter 4

First Order Methods for Unconstrained Optimization

This chapter develops algorithms for unconstrained convex optimization:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x), \tag{4.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is convex. We focus on **first-order methods**—algorithms that use only function values $f(x)$ and (sub)gradients $\nabla f(x)$ or $g \in \partial f(x)$. These methods are the workhorses of large-scale optimization for machine learning, powering everything from estimation, classification and regression to training neural networks. Furthermore, they form the basis upon which we build to design other algorithms for more complicated problems.

The chapter is organized as follows. We begin with gradient descent (Section 4.1), the simplest and most fundamental algorithm. Section 4.2 extends these ideas to non-smooth functions via subgradient methods. Section 4.3 establishes fundamental limits on convergence rates, revealing that gradient descent is suboptimal for smooth functions. This motivates the accelerated methods in Section 4.4, which achieve optimal rates by incorporating momentum. We close the chapter introducing the Stochastic Gradient Descent, which is often used to solve the empirical risk minimization problem, and we apply it to train a feed-forward neural network.

Iterative algorithms. All methods in this chapter are *iterative*: starting from an initial point $x^{(0)}$, we generate a sequence $x^{(1)}, x^{(2)}, \ldots$ that converges to the optimal solution x^*. At each iteration, we compute a search direction $d^{(k)}$ and take a step:

$$x^{(k+1)} = x^{(k)} + \eta^{(k)} d^{(k)},$$

where $\eta^{(k)} > 0$ is the **step size** (or **learning rate**). The key questions are:

- How should we choose the direction $d^{(k)}$?
- How should we choose the step size $\eta^{(k)}$?

- How many iterations K are needed to achieve $f(x^{(K)}) - f^* \leq \epsilon$?

4.1 Gradient Descent

The simplest first-order method is **gradient descent**, attributed to Cauchy (1847). The idea is intuitive: at each point, move in the direction of steepest decrease of f.

4.1.1 The Algorithm

Recall that the gradient $\nabla f(x)$ points in the direction of steepest *increase* of f at x. Therefore, $-\nabla f(x)$ points in the direction of steepest *decrease*. Gradient descent follows this direction:

$$x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla f(x^{(k)}), \quad k = 0, 1, 2, \ldots \tag{4.2}$$

The step size $\eta^{(k)} > 0$ controls how far we move. Note that the actual displacement is $\|\eta^{(k)} \nabla f(x^{(k)})\| = \eta^{(k)} \|\nabla f(x^{(k)})\|$, which depends on both the step size and the gradient magnitude.

Why the negative gradient? Any direction d satisfying $\nabla f(x)^T d < 0$ is a **descent direction**: moving infinitesimally along d decreases f. Among all unit vectors, the normalized negative gradient $-\nabla f(x)/\|\nabla f(x)\|$ achieves the most negative inner product with $\nabla f(x)$, making it the direction of steepest descent.

Motivation via local quadratic model. Gradient descent can be derived by minimizing a regularized first-order approximation. Consider:

$$\hat{f}(y) = f(x) + \nabla f(x)^T (y - x) + \frac{1}{2\eta} \|y - x\|^2. \tag{4.3}$$

The first two terms form the linear (Taylor) approximation of f at x. The third term is a *proximity penalty* (or a regularizer) that keeps y close to x—this is because we don't trust the linear approximation far from x. Minimizing over y:

$$\nabla_y \hat{f}(y) = \nabla f(x) + \frac{1}{\eta}(y - x) = 0 \quad \Rightarrow \quad y = x - \eta \nabla f(x),$$

which is exactly the gradient descent update. Thus, each step minimizes a local quadratic model of the objective.

Example 4.1 (Gradient descent on a univariate quadratic)**.** Consider $f(x) = 2x^2 + 1$ with minimum at $x^* = 0$. The gradient is $\nabla f(x) = 4x$, so gradient descent with constant step size η gives:

$$x^{(k+1)} = x^{(k)} - 4\eta x^{(k)} = (1 - 4\eta) x^{(k)}.$$

Unrolling: $x^{(k)} = (1 - 4\eta)^k x^{(0)}$. The behavior depends critically on η:

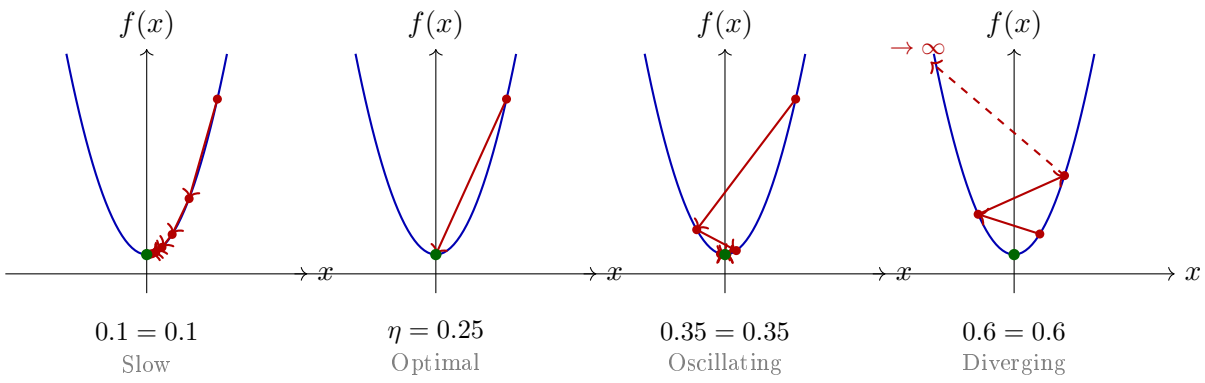

Figure 4.1: Gradient descent on $f(x) = 2x^2 + 1$ with different step sizes. The update $x^{(k+1)} = (1 - 4\eta)x^{(k)}$ converges when $|1 - 4\eta| < 1$, i.e., $\eta \in (0, 0.5)$. **Left:** Small step size converges slowly but monotonically. **Second:** Optimal $\eta = 1/4$ reaches the minimum in one step. **Third:** Larger step size still converges but oscillates around the minimum. **Right:** Step size too large causes divergence.

- $0 < \eta < 1/4$: $|1 - 4\eta| < 1$, so $x^{(k)} \to 0$ geometrically.
- $\eta = 1/4$: $1 - 4\eta = 0$, so $x^{(1)} = 0$ exactly—one-step convergence!
- $1/4 < \eta < 1/2$: $-1 < 1 - 4\eta < 0$, so $x^{(k)} \to 0$ but oscillates.
- $\eta \geq 1/2$: $|1 - 4\eta| \geq 1$, so $|x^{(k)}| \to \infty$—divergence.

This illustrates a fundamental principle: small steps converge surely but slowly. As we increase the step size, we converge faster, but we start overshooting the optimal point, resulting in oscillations. Finally, beyond a maximum step size, we diverge.

```
import numpy as np
import matplotlib.pyplot as plt

def gradient_descent_1d_demo():
    """Visualize GD behavior for different step sizes."""
    f = lambda x: 2*x**2 + 1
    grad_f = lambda x: 4*x
    x0 = 2.0

    fig, axes = plt.subplots(1, 4, figsize=(14, 3))
    step_sizes = [0.1, 0.25, 0.3, 0.6]
    titles = [r'$\eta=0.1$: Slow', r'$\eta=0.25$: Optimal',
              r'$\eta=0.3$: Oscillating', r'$\eta=0.6$: Diverging']

    x_plot = np.linspace(-3, 3, 200)

    for ax, eta, title in zip(axes, step_sizes, titles):
```

```
        # Plot function
        ax.plot(x_plot, f(x_plot), 'b-', linewidth=2, alpha=0.7)

        # Run GD
        x = x0
        trajectory = [(x, f(x))]
        for _ in range(15):
            x_new = x - eta * grad_f(x)
            trajectory.append((x_new, f(x_new)))
            if abs(x_new) > 5:
                break
            x = x_new

        # Plot trajectory
        traj = np.array(trajectory)
        ax.plot(traj[:, 0], traj[:, 1], 'ro-', markersize=4, alpha=0.8)
        ax.plot(x0, f(x0), 'go', markersize=8, label='Start')

        ax.set_xlim(-3, 3)
        ax.set_ylim(0, 10)
        ax.set_xlabel('$x$')
        ax.set_title(title)
        ax.grid(True, alpha=0.3)

    axes[0].set_ylabel('$f(x)$')
    plt.tight_layout()
    plt.savefig('gd_step_sizes.pdf', bbox_inches='tight')
    plt.show()

gradient_descent_1d_demo()
```

Check Your Understanding 4.1.

For the function $f(x) = \frac{1}{2}x^T Q x$ where $Q = diag(1, 100)$:

(a) What is the gradient $\nabla f(x)$?

(b) If we use step size η, write the gradient descent update.

(c) For what values of η does gradient descent converge?

(d) Even with a valid step size, why might convergence be slow?

4.1.2 Step Size Selection

The step size $\eta^{(k)}$ critically affects both convergence and speed. We present four common strategies.

1. Fixed step size. Use $\eta^{(k)} = \eta$ for all k. Simple to implement, but requires knowing properties of f (specifically, the smoothness constant M) to choose η appropriately. We analyze this choice in convergence theorems below.

2. Exact line search. At each iteration, solve the one-dimensional optimization:

$$\eta^{(k)} = \arg\min_{\eta \geq 0} f(x^{(k)} - \eta \nabla f(x^{(k)})).$$

This finds the optimal step along the gradient direction. Exact line search is rarely practical because the inner optimization can be expensive, but it provides a useful theoretical benchmark and is efficient for some special cases (e.g., quadratic objectives).

3. Backtracking line search (Armijo's rule). A practical middle ground: start with a large step and reduce it until sufficient decrease is achieved. Given parameters $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$:

Algorithm 1 Backtracking Line Search

Input: Current point x, gradient $\nabla f(x)$, parameters α, β
$\eta \leftarrow 1$
while $f(x - \eta \nabla f(x)) > f(x) - \alpha\eta \|\nabla f(x)\|^2$ **do** ▷ Armijo condition
    $\eta \leftarrow \beta\eta$
end while
return η

The **Armijo condition** requires that the actual decrease $f(x) - f(x - \eta\nabla f(x))$ is at least $\alpha\eta\|\nabla f(x)\|^2$, which is a fraction α of the decrease predicted by the linear model. Typical values are $\alpha \in [0.01, 0.3]$ and $\beta \in [0.5, 0.8]$.

Proposition 4.2 (Termination of backtracking)**.** *Backtracking line search always terminates with $\eta > 0$. Since the last term in the condition is negative, it also improves.*

Proof. By Taylor expansion, for small η:

$$f(x - \eta\nabla f(x)) = f(x) - \eta\|\nabla f(x)\|^2 + O(\eta^2) < f(x) - \alpha\eta\|\nabla f(x)\|^2$$

since $\alpha < 1$. Thus the Armijo condition is satisfied for sufficiently small η. □

4. Diminishing step sizes. Use step sizes satisfying:

$$\sum_{k=0}^{\infty} \eta^{(k)} = \infty \quad \text{and} \quad \sum_{k=0}^{\infty} (\eta^{(k)})^2 < \infty.$$

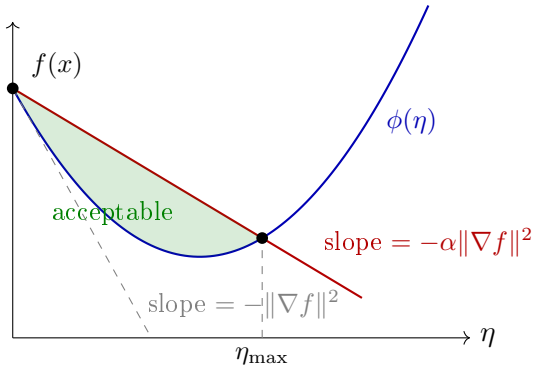

Figure 4.2: Backtracking line search accepts step sizes where $\phi(\eta) = f(x - \eta\nabla f(x))$ lies below the Armijo line with slope $-\alpha\|\nabla f\|^2$. Starting from $\eta = 1$, we shrink by factor β until this condition holds.

A common choice is $\eta^{(k)} = c/\sqrt{k+1}$ or $\eta^{(k)} = c/(k+1)$. The first condition ensures we can reach the optimum from any starting point; the second ensures stability. This strategy is essential for stochastic gradient descent (Section 4.6).

```python
import numpy as np

def backtracking_line_search(f, grad, x, alpha=0.3, beta=0.5):
    """
    Backtracking line search with Armijo condition.

    Parameters:
        f: objective function
        grad: gradient at x (precomputed)
        x: current point
        alpha: Armijo parameter (0.01-0.3 typical)
        beta: shrinkage factor (0.5-0.8 typical)

    Returns:
        eta: step size satisfying Armijo condition
    """
    eta = 1.0
    grad_norm_sq = np.dot(grad, grad)
    f_x = f(x)

    while f(x - eta * grad) > f_x - alpha * eta * grad_norm_sq:
        eta *= beta
        if eta < 1e-16:  # Numerical safeguard
            break

    return eta
```

```
def gradient_descent(f, grad_f, x0, method='backtracking',
                     eta=None, max_iter=1000, tol=1e-8):
    """
    Gradient descent with various step size strategies.

    Parameters:
        f: objective function
        grad_f: gradient function
        x0: initial point
        method: 'fixed', 'backtracking', or 'diminishing'
        eta: step size (for 'fixed') or initial scale (for 'diminishing')
        max_iter: maximum iterations
        tol: gradient norm tolerance for stopping

    Returns:
        x: final point
        history: dict with 'f', 'grad_norm', 'x' trajectories
    """
    x = np.array(x0, dtype=float)
    history = {'f': [f(x)], 'grad_norm': [], 'x': [x.copy()]}

    for k in range(max_iter):
        grad = grad_f(x)
        grad_norm = np.linalg.norm(grad)
        history['grad_norm'].append(grad_norm)

        if grad_norm < tol:
            break

        # Choose step size
        if method == 'fixed':
            step = eta
        elif method == 'backtracking':
            step = backtracking_line_search(f, grad, x)
        elif method == 'diminishing':
            step = eta / np.sqrt(k + 1)
        else:
            raise ValueError(f"Unknown method: {method}")

        # Update
        x = x - step * grad
        history['f'].append(f(x))
        history['x'].append(x.copy())
```

```
    return x, history
```

Check Your Understanding 4.2.

Consider gradient descent with backtracking on a strongly convex function.

(a) If $\alpha = 0.5$ (maximum allowed value), what does the Armijo condition require?

(b) If $\alpha \to 0$, what happens to the accepted step sizes?

(c) Why do we start with $\eta = 1$ rather than a larger value?

4.1.3 Convergence Analysis

The convergence rate of gradient descent depends critically on properties of f. Recall from Chapter 2:

- f is G-**Lipschitz** if $\|\nabla f(x)\| \leq G$ for all x
- f is M-**smooth** if $\nabla^2 f(x) \preceq MI$ for all x
- f is m-**strongly convex** if $\nabla^2 f(x) \succeq mI$ for all x

We present three convergence results of increasing strength.

Theorem 4.3 (Convergence of gradient descent)**.** *Let f be convex. Define $R = \|x^{(0)} - x^*\|$ (distance to optimum) and $f_k^{\text{best}} = \min_{i \leq k} f(x^{(i)})$ (best value found).*

1. ***Lipschitz case:*** *If f is G-Lipschitz, with step size $\eta^{(i)} = \frac{R}{G\sqrt{k}}$:*

$$f_k^{\text{best}} - f^* \leq \frac{RG}{\sqrt{k}}.$$

2. ***Smooth case:*** *If f is additionally M-smooth, with step size $\eta = \frac{1}{M}$:*

$$f_k^{\text{best}} - f^* \leq \frac{MR^2}{2k}.$$

3. ***Strongly convex case:*** *If f is additionally m-strongly convex, with step size $\eta = \frac{1}{M}$:*

$$f(x^{(k)}) - f^* \leq \left(1 - \frac{m}{M}\right)^k (f(x^{(0)}) - f^*).$$

We provide a complete proof for the Lipschitz case, as it illustrates the key techniques. The other cases follow similar patterns with additional structure.

Table 4.1: Summary of gradient descent convergence rates.

Assumptions	**Rate**	**Iterations for ϵ**	**Step size**
Convex, G-Lipschitz	$O(1/\sqrt{k})$	$O(1/\epsilon^2)$	$\eta = R/(G\sqrt{k})$
Convex, M-smooth	$O(1/k)$	$O(1/\epsilon)$	$\eta = 1/M$
m-strongly convex, M-smooth	$O((1-m/M)^k)$	$O(\kappa \log(1/\epsilon))$	$\eta = 1/M$

Proof of Part 1 (Lipschitz case).

Step 1: Track distance to optimum. Starting from the gradient descent update $x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla f(x^{(k)})$:

$$\begin{aligned} \|x^{(k+1)} - x^*\|^2 &= \|x^{(k)} - \eta^{(k)} \nabla f(x^{(k)}) - x^*\|^2 \\ &= \|x^{(k)} - x^*\|^2 - 2\eta^{(k)} \nabla f(x^{(k)})^T (x^{(k)} - x^*) + (\eta^{(k)})^2 \|\nabla f(x^{(k)})\|^2. \end{aligned} \tag{4.4}$$

Step 2: Apply convexity. By the first-order condition for convexity (Theorem 2.5):

$$f(x^*) \geq f(x^{(k)}) + \nabla f(x^{(k)})^T (x^* - x^{(k)}),$$

which rearranges to:

$$\nabla f(x^{(k)})^T (x^{(k)} - x^*) \geq f(x^{(k)}) - f^*. \tag{4.5}$$

Substituting (4.5) into (4.4):

$$\|x^{(k+1)} - x^*\|^2 \leq \|x^{(k)} - x^*\|^2 - 2\eta^{(k)} (f(x^{(k)}) - f^*) + (\eta^{(k)})^2 \|\nabla f(x^{(k)})\|^2. \tag{4.6}$$

Step 3: Telescope over iterations. Sum (4.6) for $i = 0, 1, \ldots, k-1$:

$$\|x^{(k)} - x^*\|^2 \leq \|x^{(0)} - x^*\|^2 - 2\sum_{i=0}^{k-1} \eta^{(i)} (f(x^{(i)}) - f^*) + \sum_{i=0}^{k-1} (\eta^{(i)})^2 \|\nabla f(x^{(i)})\|^2.$$

Since $\|x^{(k)} - x^*\|^2 \geq 0$, we can drop the left side and rearrange:

$$\sum_{i=0}^{k-1} \eta^{(i)} (f(x^{(i)}) - f^*) \leq \frac{R^2 + G^2 \sum_{i=0}^{k-1} (\eta^{(i)})^2}{2}, \tag{4.7}$$

where we used $\|\nabla f(x)\| \leq G$ (Lipschitz property) and $R = \|x^{(0)} - x^*\|$.

Step 4: Extract convergence rate. Since $f_k^{\text{best}} \leq f(x^{(i)})$ for all i:

$$f_k^{\text{best}} - f^* \leq \frac{\sum_{i=0}^{k-1} \eta^{(i)} (f(x^{(i)}) - f^*)}{\sum_{i=0}^{k-1} \eta^{(i)}} \leq \frac{R^2 + G^2 \sum_{i=0}^{k-1} (\eta^{(i)})^2}{2 \sum_{i=0}^{k-1} \eta^{(i)}}. \tag{4.8}$$

For constant step size η:

$$f_k^{\text{best}} - f^* \leq \frac{R^2}{2k\eta} + \frac{\eta G^2}{2}.$$

Minimizing over η by setting the derivative to zero: $-\frac{R^2}{2k\eta^2} + \frac{G^2}{2} = 0$, giving $\eta^* = \frac{R}{G\sqrt{k}}$. Substituting:

$$f_k^{\text{best}} - f^* \leq \frac{RG}{\sqrt{k}}.$$

□

Proof sketch for Part 2 (Smooth case). Smoothness provides a tighter bound. From the smoothness inequality:

$$f(x^{(k+1)}) \leq f(x^{(k)}) + \nabla f(x^{(k)})^T (x^{(k+1)} - x^{(k)}) + \frac{M}{2}\|x^{(k+1)} - x^{(k)}\|^2.$$

With step size $\eta = 1/M$ and update $x^{(k+1)} = x^{(k)} - \frac{1}{M}\nabla f(x^{(k)})$:

$$f(x^{(k+1)}) \leq f(x^{(k)}) - \frac{1}{2M}\|\nabla f(x^{(k)})\|^2.$$

This shows the gradient "self-tunes": as we approach the optimum, $\|\nabla f(x^{(k)})\|$ shrinks, unlike the Lipschitz case where we bounded it by a constant G. Combining with convexity and telescoping yields the $O(1/k)$ rate. See [1] for details. □

Proof sketch for Part 3 (Strongly convex case). Strong convexity provides a lower bound: $\|\nabla f(x)\|^2 \geq 2m(f(x) - f^*)$. Combined with the smoothness result:

$$f(x^{(k+1)}) - f^* \leq f(x^{(k)}) - f^* - \frac{m}{M}(f(x^{(k)}) - f^*) = \left(1 - \frac{m}{M}\right)(f(x^{(k)}) - f^*).$$

This is a contraction: the error decreases by a constant factor each iteration. The rate $(1 - m/M)^k$ is called **linear convergence** because $\log(f(x^{(k)}) - f^*)$ decreases linearly in k. □

Remark 4.4 (Interpreting the rates).

- $O(1/\sqrt{k})$: Slow sublinear convergence. Halving the error requires quadrupling iterations.
- $O(1/k)$: Faster sublinear convergence. Halving error requires doubling iterations.
- $O(\rho^k)$ **with** $\rho < 1$: Linear (exponential) convergence. Error decreases by constant factor per iteration.

The **condition number** $\kappa = M/m$ governs convergence in the strongly convex case. Large κ (ill-conditioned problems) converge slowly because $1 - 1/\kappa \approx 1$.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def compare_convergence():
    """Compare GD convergence under different function properties."""
    np.random.seed(42)
    n = 20

    # Well-conditioned quadratic (kappa = 5)
    Q_good = np.diag(np.linspace(1, 5, n))

    # Ill-conditioned quadratic (kappa = 100)
    Q_bad = np.diag(np.linspace(1, 100, n))

    def run_gd(Q, n_iters=500):
        x = np.ones(n)
        M = np.max(np.diag(Q))
        eta = 1.0 / M

        errors = []
        for _ in range(n_iters):
            f_val = 0.5 * x @ Q @ x
            errors.append(f_val)  # f* = 0
            x = x - eta * (Q @ x)
        return errors

    errors_good = run_gd(Q_good)
    errors_bad = run_gd(Q_bad)

    # Plot
    fig, ax = plt.subplots(figsize=(8, 5))
    ax.semilogy(errors_good, 'b-', linewidth=2, label=r'$\kappa = 5$')
    ax.semilogy(errors_bad, 'r-', linewidth=2, label=r'$\kappa = 100$')

    # Theoretical rates
    k = np.arange(1, 501)
    ax.semilogy(errors_good[0] * (1 - 1/5)**k, 'b--', alpha=0.5)
    ax.semilogy(errors_bad[0] * (1 - 1/100)**k, 'r--', alpha=0.5)

    ax.set_xlabel('Iteration $k$')
    ax.set_ylabel('$f(x^{(k)}) - f^*$')
    ax.set_title('Effect of Condition Number on Convergence')
    ax.legend()
    ax.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig('condition_number.pdf', bbox_inches='tight')
```

```
    plt.show()

compare_convergence()
```

Check Your Understanding 4.3.

A function f has $M = 100$ and $m = 1$ (so $\kappa = 100$).

(a) What step size should gradient descent use?

(b) What is the convergence rate $(1 - m/M)$?

(c) How many iterations to reduce error by factor 10^{-6}?

(d) If we could improve the condition number to $\kappa = 10$, how many iterations?

4.2 Subgradient Methods

When f is convex but not differentiable everywhere, the gradient may not exist at some points. However, for convex functions $f : \mathbb{R}^n \to \mathbb{R}$, we can always define a non-empty **subdifferential**—a set of vectors called subgradients that generalize the gradient. The **subgradient method** extends gradient descent to this setting.

4.2.1 Subgradients and Subdifferentials

Definition 4.5 (Subgradient)**.** A vector $g \in \mathbb{R}^n$ is a **subgradient** of f at x if for all $y \in \mathbb{R}^n$:

$$f(y) \geq f(x) + g^T(y - x). \tag{4.9}$$

The set of all subgradients at x is the **subdifferential**, denoted $\partial f(x)$.

The defining inequality (4.9) says that g defines a supporting hyperplane to the epigraph of f at $(x, f(x))$. For differentiable functions, $\partial f(x) = \{\nabla f(x)\}$ contains only the gradient. At points of non-differentiability, $\partial f(x)$ can contain multiple vectors.

Example 4.6 (Subdifferential of absolute value)**.** For $f(x) = |x|$ on $\mathbb{R}$:

$$\partial f(x) = \begin{cases} \{-1\} & x < 0 \\ [-1, 1] & x = 0 \\ \{1\} & x > 0 \end{cases}$$

At the "kink" $x = 0$, any slope between -1 and 1 gives a line that lies below the graph of $|x|$.

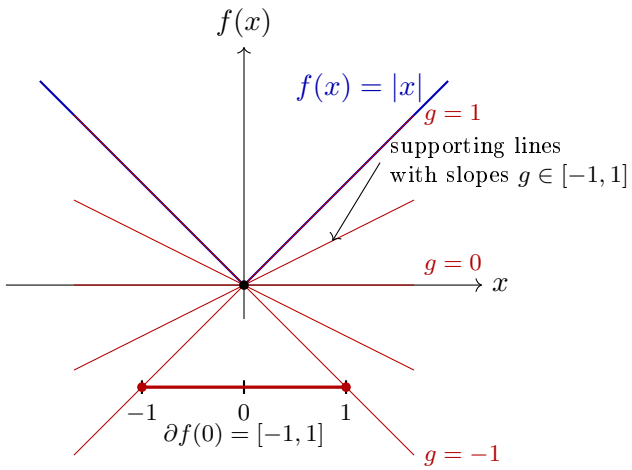

Figure 4.3: The subdifferential of $|x|$ at $x = 0$. Each value $g \in [-1, 1]$ is a subgradient (each subgradient is a scalar because f is one-dimensional), corresponding to the slope of a supporting line that lies below the graph. The subdifferential $\partial f(0) = [-1, 1]$ is shown on the number line below.

Example 4.7 (Subdifferential of ℓ_1 norm)**.** For $f(x) = \|x\|_1 = \sum_{i=1}^n |x_i|$, the subdifferential is:

$$\partial f(x) = \{g : g_i = \text{sign}(x_i) \text{ if } x_i \neq 0, \text{ and } g_i \in [-1, 1] \text{ if } x_i = 0\}.$$

The key property for optimization: x^* minimizes f if and only if $0 \in \partial f(x^*)$.

4.2.2 The Subgradient Method

The subgradient method iterates:

$$x^{(k+1)} = x^{(k)} - \eta^{(k)} g^{(k)}, \quad \text{where } g^{(k)} \in \partial f(x^{(k)}). \tag{4.10}$$

Algorithm 2 Subgradient Method

Input: Initial $x^{(0)}$, step sizes $\{\eta^{(k)}\}$, iterations K
$f^{\text{best}} \leftarrow f(x^{(0)})$, $x^{\text{best}} \leftarrow x^{(0)}$
for $k = 0, 1, \ldots, K-1$ **do**
    Compute any $g^{(k)} \in \partial f(x^{(k)})$
    $x^{(k+1)} \leftarrow x^{(k)} - \eta^{(k)} g^{(k)}$
    if $f(x^{(k+1)}) < f^{\text{best}}$ **then**
        $f^{\text{best}} \leftarrow f(x^{(k+1)})$, $x^{\text{best}} \leftarrow x^{(k+1)}$
    end if
end for
return x^{best}

Critical difference from gradient descent: A subgradient is *not* necessarily a descent direction!

The function value may increase: $f(x^{(k+1)}) > f(x^{(k)})$. This is why we track the best iterate x^{best}.

Theorem 4.8 (Convergence of subgradient method)**.** *Let f be convex with $\|g\| \leq G$ for all $g \in \partial f(x)$, and let $R = \|x^{(0)} - x^*\|$. With step size $\eta^{(k)} = R/(G\sqrt{k})$:*

$$f_k^{\text{best}} - f^* \leq \frac{RG}{\sqrt{k}}.$$

The proof is identical to the Lipschitz case for gradient descent, replacing the gradient with a subgradient and using (4.9) instead of the first-order convexity condition.

Remark 4.9. The $O(1/\sqrt{k})$ rate cannot be improved for non-smooth convex optimization—it is optimal (see Section 4.3). This is significantly slower than the $O(1/k)$ rate achievable for smooth functions.

```
import numpy as np
import matplotlib.pyplot as plt

def subgradient_l1(x):
    """Subgradient of ||x||_1."""
    g = np.sign(x)
    g[x == 0] = np.random.uniform(-1, 1, np.sum(x == 0))  # Any value in [-1,1]
    return g

def subgradient_method(f, subgrad_f, x0, R, G, max_iter=1000):
    """Subgradient method with step size eta_k = R/(G*sqrt(k))."""
    x = np.array(x0, dtype=float)
    f_best = f(x)
    x_best = x.copy()

    history = {'f': [f(x)], 'f_best': [f_best]}

    for k in range(1, max_iter + 1):
        g = subgrad_f(x)
        eta = R / (G * np.sqrt(k))
        x = x - eta * g

        f_val = f(x)
        history['f'].append(f_val)

        if f_val < f_best:
            f_best = f_val
            x_best = x.copy()
        history['f_best'].append(f_best)

    return x_best, history
```

```
# Example: minimize ||x||_1
np.random.seed(42)
n = 20
x0 = np.random.randn(n) * 3
R = np.linalg.norm(x0)  # Distance to x* = 0
G = np.sqrt(n)  # ||g||_2 <= sqrt(n) for l1 subgradient

x_opt, history = subgradient_method(
    lambda x: np.sum(np.abs(x)),
    subgradient_l1,
    x0, R, G, max_iter=2000
)

# Plot
fig, ax = plt.subplots(figsize=(8, 5))
ax.loglog(history['f'], 'r-', alpha=0.3, label='$f(x^{(k)})$')
ax.loglog(history['f_best'], 'b-', linewidth=2, label='$f^{\\mathrm{best}}_k$')

# Theoretical rate
k = np.arange(1, len(history['f_best']) + 1)
ax.loglog(k, R * G / np.sqrt(k), 'k--', label='$O(1/\\sqrt{k})$')

ax.set_xlabel('Iteration $k$')
ax.set_ylabel('Function value')
ax.set_title('Subgradient Method on $\\|x\\|_1$')
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('subgradient.pdf', bbox_inches='tight')
plt.show()
```

Check Your Understanding 4.4.

For $f(x) = \max\{x_1, x_2, x_3\}$ on $\mathbb{R}^3$:

(a) Is f convex? Differentiable everywhere?

(b) What is $\partial f(x)$ when $x_1 > x_2 > x_3$?

(c) What is $\partial f(x)$ when $x_1 = x_2 > x_3$?

(d) What is x^* and what condition does it satisfy?

4.3 Lower Bounds

How fast can *any* first-order method converge? Nesterov [2] established fundamental limits using the **first-order oracle** model: an algorithm can query $f(x)$ and $\nabla f(x)$ (or $g \in \partial f(x)$) at any point, but has no other information about f.

Theorem 4.10 (Lower bounds for first-order methods)**.** *Consider convex optimization using only first-order oracle queries. To achieve $f(x^{(k)}) - f^* \leq \epsilon$, any algorithm requires at least:*

1. ***Lipschitz convex:*** $k = \Omega(1/\epsilon^2)$ *iterations*
2. ***Smooth convex:*** $k = \Omega(1/\sqrt{\epsilon})$ *iterations*
3. ***Smooth + strongly convex:*** $k = \Omega(\sqrt{\kappa}\log(1/\epsilon))$ *iterations*

Table 4.2: Comparison of gradient descent rates with lower bounds.

Setting	GD Rate	Lower Bound	Optimal?
Lipschitz convex	$O(1/\epsilon^2)$	$\Omega(1/\epsilon^2)$	✓
Smooth convex	$O(1/\epsilon)$	$\Omega(1/\sqrt{\epsilon})$	×
Smooth + strongly convex	$O(\kappa\log(1/\epsilon))$	$\Omega(\sqrt{\kappa}\log(1/\epsilon))$	×

The gaps in Table 4.2 are significant:

- For smooth functions, gradient descent needs $O(1/\epsilon)$ iterations, but the lower bound is $\Omega(1/\sqrt{\epsilon})$—a quadratic gap!
- For strongly convex functions, the gap is in the dependence on condition number: κ vs $\sqrt{\kappa}$.

These gaps motivate **accelerated methods**, which we develop next.

4.4 Accelerated Methods

Can we close the gaps identified in Section 4.3? Yes! **Accelerated** (or **momentum**) methods achieve the optimal rates by incorporating information from previous iterates.

4.4.1 Intuition: The Problem with Gradient Descent

Consider minimizing $f(x) = \frac{1}{2}(x_1^2 + 100x_2^2)$, which has condition number $\kappa = 100$. The level sets are elongated ellipses. Gradient descent with step size $\eta = 1/100$ exhibits characteristic "zigzag" behavior:

The zigzag happens because the gradient points toward the nearest contour, not toward the optimum. Momentum methods address this by "averaging" recent directions, canceling out oscillations.

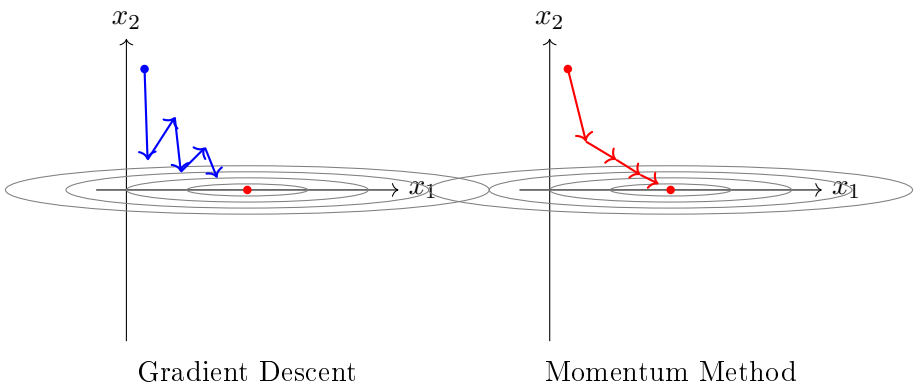

Figure 4.4: Gradient descent oscillates in narrow valleys (left). Momentum methods dampen oscillations and accelerate along consistent directions (right).

4.4.2 The Heavy Ball Method

Polyak's **heavy ball method** (1964) [3] adds a momentum term:

$$x^{(k+1)} = x^{(k)} - \eta \nabla f(x^{(k)}) + \beta(x^{(k)} - x^{(k-1)}), \tag{4.11}$$

where $\beta \in [0, 1)$ is the **momentum parameter**. The term $x^{(k)} - x^{(k-1)}$ is simply the previous step—the direction we were already moving—and the update incorporates a fraction β of this direction alongside the gradient.

Intuition: a ball with inertia. Imagine a heavy ball rolling down a hilly surface. Without momentum ($\beta = 0$), the ball has no inertia: at each instant it moves in the direction of steepest descent, causing it to oscillate back and forth in narrow valleys. With momentum ($\beta > 0$), the ball remembers where it was going. It builds up speed along consistent directions and rolls smoothly through valleys instead of bouncing between the walls.

Mathematically, this selective amplification and damping occurs because:

- **Consistent gradients** (e.g., along a valley floor): Successive steps point in similar directions, so momentum accumulates and accelerates progress.
- **Oscillating gradients** (e.g., across a narrow valley): Successive steps alternate in sign, so momentum terms partially cancel, damping the zigzag behavior.

Thus momentum automatically amplifies motion in consistent directions while suppressing oscillations—exactly what is needed for ill-conditioned problems.

Algorithm 3 Heavy Ball Method

Input: $x^{(0)}$, step size η, momentum β, iterations K
$x^{(-1)} \leftarrow x^{(0)}$ ▷ Initialize "previous" iterate
for $k = 0, 1, \ldots, K-1$ **do**
    $x^{(k+1)} \leftarrow x^{(k)} - \eta \nabla f(x^{(k)}) + \beta(x^{(k)} - x^{(k-1)})$
end for
return $x^{(K)}$

For quadratic $f(x) = \frac{1}{2}x^T Q x$ with eigenvalues in $[m, M]$, the optimal parameters are:

$$\eta^* = \frac{4}{(\sqrt{M} + \sqrt{m})^2}, \quad \beta^* = \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^2,$$

achieving linear convergence with rate $\rho = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$.

Remark 4.11. Compare the rates:

- Gradient descent: $\rho_{\text{GD}} = \frac{\kappa-1}{\kappa+1} \approx 1 - \frac{2}{\kappa}$ for large κ
- Heavy ball: $\rho_{\text{HB}} = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} \approx 1 - \frac{2}{\sqrt{\kappa}}$ for large κ

For $\kappa = 100$: GD needs $\approx 50\times$ more iterations than heavy ball for the same accuracy!

4.4.3 Nesterov's Accelerated Gradient

Nesterov's method (1983) [4] achieves optimal rates for smooth convex functions. The key innovation: evaluate the gradient at a *lookahead* point rather than the current iterate.

Algorithm 4 Nesterov's Accelerated Gradient Method

Input: $x^{(0)}$, step size $\eta = 1/M$, iterations K
$y^{(0)} \leftarrow x^{(0)}$
for $k = 0, 1, \ldots, K-1$ **do**
    $x^{(k+1)} \leftarrow y^{(k)} - \eta \nabla f(y^{(k)})$ ▷ Gradient step from lookahead
    $y^{(k+1)} \leftarrow x^{(k+1)} + \frac{k}{k+3}(x^{(k+1)} - x^{(k)})$ ▷ Momentum/extrapolation
end for
return $x^{(K)}$

The momentum coefficient $\frac{k}{k+3}$ grows from 0 toward 1 as iterations progress. Early on, we rely primarily on gradient information; as the algorithm proceeds, we increasingly trust the momentum direction built up from previous steps.

Theorem 4.12 (Convergence of Nesterov's method)**.** *Let f be convex and M-smooth. With $\eta = 1/M$:*

$$f(x^{(k)}) - f^* \leq \frac{2M|x^{(0)} - x^*|^2}{(k+1)^2}.$$

This is $O(1/k^2)$, requiring $O(1/\sqrt{\epsilon})$ iterations for ϵ-accuracy—matching the lower bound of Theorem 4.10.

For m-strongly convex functions, a variant with constant momentum $\beta = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ achieves:

$$f(x^{(k)}) - f^* \leq \left(1 - \frac{1}{\sqrt{\kappa}}\right)^k (f(x^{(0)}) - f^*).$$

Compared to gradient descent's rate of $(1 - 1/\kappa)^k$, the improvement is in the dependence on condition number: $\sqrt{\kappa}$ versus κ. This matches the lower bound of Theorem 4.10, confirming that Nesterov's method is optimal among first-order methods.

```
import numpy as np
import matplotlib.pyplot as plt

def gradient_descent(Q, x0, eta, n_iters):
    """Standard gradient descent on f(x) = 0.5 * x'Qx."""
    x = x0.copy()
    errors = [0.5 * x @ Q @ x]
    for _ in range(n_iters):
        x = x - eta * (Q @ x)
        errors.append(0.5 * x @ Q @ x)
    return errors

def heavy_ball(Q, x0, eta, beta, n_iters):
    """Heavy ball method."""
    x = x0.copy()
    x_prev = x0.copy()
    errors = [0.5 * x @ Q @ x]
    for _ in range(n_iters):
        x_new = x - eta * (Q @ x) + beta * (x - x_prev)
        x_prev = x
        x = x_new
        errors.append(0.5 * x @ Q @ x)
    return errors

def nesterov(Q, x0, eta, n_iters):
    """Nesterov's accelerated gradient."""
    x = x0.copy()
    y = x0.copy()
    errors = [0.5 * x @ Q @ x]
    for k in range(n_iters):
        x_new = y - eta * (Q @ y)
        momentum = k / (k + 3)
        y = x_new + momentum * (x_new - x)
        x = x_new
```

```
        errors.append(0.5 * x @ Q @ x)
    return errors

def compare_momentum_methods():
    """Compare GD, Heavy Ball, and Nesterov."""
    np.random.seed(42)
    n = 50

    # Ill-conditioned quadratic
    m, M = 1, 100
    kappa = M / m
    Q = np.diag(np.linspace(m, M, n))

    x0 = np.random.randn(n)
    n_iters = 300

    # Optimal parameters
    eta_gd = 1 / M
    eta_hb = 4 / (np.sqrt(M) + np.sqrt(m))**2
    beta_hb = ((np.sqrt(kappa) - 1) / (np.sqrt(kappa) + 1))**2
    eta_nest = 1 / M

    errors_gd = gradient_descent(Q, x0, eta_gd, n_iters)
    errors_hb = heavy_ball(Q, x0, eta_hb, beta_hb, n_iters)
    errors_nest = nesterov(Q, x0, eta_nest, n_iters)

    # Plot
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.semilogy(errors_gd, 'b-', linewidth=2, label='Gradient Descent')
    ax.semilogy(errors_hb, 'g-', linewidth=2, label='Heavy Ball')
    ax.semilogy(errors_nest, 'r-', linewidth=2, label='Nesterov')

    # Theoretical rates
    k = np.arange(len(errors_gd))
    ax.semilogy(errors_gd[0] * ((kappa-1)/(kappa+1))**k, 'b--', alpha=0.4)
    ax.semilogy(errors_hb[0] * ((np.sqrt(kappa)-1)/(np.sqrt(kappa)+1))**(2*k), 'g--', alpha
    =0.4)
    ax.semilogy(errors_nest[0] * 4 / (k+2)**2, 'r--', alpha=0.4)

    ax.set_xlabel('Iteration $k$')
    ax.set_ylabel('$f(x^{(k)}) - f^*$')
    ax.set_title(f'Momentum Methods Comparison ($\\kappa = {int(kappa)}$)')
    ax.legend()
    ax.grid(True, alpha=0.3)
```

```
    ax.set_ylim([1e-14, errors_gd[0] * 10])

    plt.tight_layout()
    plt.savefig('momentum_comparison.pdf', bbox_inches='tight')
    plt.show()

    # Print iteration counts
    target = 1e-8 * errors_gd[0]
    for name, errors in [('GD', errors_gd), ('Heavy Ball', errors_hb), ('Nesterov', errors_nest
    )]:
        iters = next((i for i, e in enumerate(errors) if e < target), len(errors))
        print(f"{name}: {iters} iterations to reach {target:.2e}")

compare_momentum_methods()
```

Check Your Understanding 4.5.

For a quadratic with condition number $\kappa = 10000$:

(a) Approximately how many GD iterations for 10^{-8} relative accuracy?

(b) Approximately how many Nesterov iterations?

(c) What is the optimal momentum parameter β for the strongly convex Nesterov variant?

4.5 Comparison of Deterministic Methods

Before turning to stochastic methods, we summarize the deterministic algorithms developed so far. Table 4.3 compares their convergence rates under various assumptions.

Table 4.3: Summary of first-order methods for unconstrained optimization. All methods assume f is convex; the table shows additional assumptions required for each rate.

Method	Additional Assumptions	Rate	Iterations	Optimal?
Subgradient	G-Lipschitz	$O(1/\sqrt{k})$	$O(1/\epsilon^2)$	✓
Gradient descent	G-Lipschitz	$O(1/\sqrt{k})$	$O(1/\epsilon^2)$	✓
Gradient descent	M-smooth	$O(1/k)$	$O(1/\epsilon)$	×
Gradient descent	m-SC, M-smooth	$O((1-1/\kappa)^k)$	$O(\kappa \log(1/\epsilon))$	×
Heavy ball	m-SC, M-smooth	$O((1-1/\sqrt{\kappa})^k)$	$O(\sqrt{\kappa}\log(1/\epsilon))$	✓
Nesterov	M-smooth	$O(1/k^2)$	$O(1/\sqrt{\epsilon})$	✓
Nesterov	m-SC, M-smooth	$O((1-1/\sqrt{\kappa})^k)$	$O(\sqrt{\kappa}\log(1/\epsilon))$	✓

Practical guidance.

- **Non-smooth problems**: Use subgradient method with diminishing step sizes.
- **Smooth problems**: Use Nesterov acceleration if you know M; otherwise, gradient descent with backtracking is robust.
- **Strongly convex + smooth**: Heavy ball or accelerated Nesterov variant. The speedup over GD can be dramatic for ill-conditioned problems.
- **Very large scale**: Consider stochastic methods (Section 4.6), which trade accuracy for cheaper iterations.

4.6 Stochastic Gradient Descent

Many machine learning problems have objectives that are averages over data:

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x), \tag{4.12}$$

where $f_i(x)$ is the loss on the i-th training example. For instance, in empirical risk minimization with n data points (a_i, b_i):

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} \ell(h_x(a_i), b_i),$$

where h_x is a model parameterized by x and ℓ is a loss function.

When n is large (millions or billions of samples), computing the full gradient

$$\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x)$$

requires n gradient evaluations per iteration—prohibitively expensive.

4.6.1 The SGD Algorithm

Stochastic gradient descent (SGD) replaces the full gradient with a cheap, unbiased estimate:

$$x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla f_{i_k}(x^{(k)}), \tag{4.13}$$

where i_k is sampled uniformly at random from $\{1, \ldots, n\}$.

Proposition 4.13 (Unbiased gradient estimate)**.** *The stochastic gradient is an unbiased estimator:*

$$\mathbb{E}_{i_k}[\nabla f_{i_k}(x)] = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x) = \nabla f(x).$$

The cost per iteration drops from $O(n)$ to $O(1)$—essential for large-scale learning.

Algorithm 5 Stochastic Gradient Descent

1: **Input:** Initial $x^{(0)}$, step sizes $\{\eta^{(k)}\}$, iterations K
2: **for** $k = 0, 1, \ldots, K-1$ **do**
3: Sample i_k uniformly from $\{1, \ldots, n\}$
4: $x^{(k+1)} \leftarrow x^{(k)} - \eta^{(k)} \nabla f_{i_k}(x^{(k)})$
5: **end for**
6: **return** $x^{(K)}$ or $\bar{x}^{(K)} = \frac{1}{K} \sum_{k=0}^{K-1} x^{(k)}$

4.6.2 Mini-batch SGD

Instead of a single sample, we can use a **mini-batch** of b samples:

$$x^{(k+1)} = x^{(k)} - \eta^{(k)} \frac{1}{b} \sum_{j \in B_k} \nabla f_j(x^{(k)}), \tag{4.14}$$

where $B_k \subset \{1, \ldots, n\}$ is a random subset of size b.

Why mini-batches?

- **Variance reduction**: The variance of the gradient estimate is reduced by factor b.
- **Parallelism**: The b gradients can be computed in parallel on GPUs.
- **Memory efficiency**: Modern hardware is optimized for batch operations.

Typical mini-batch sizes range from 32 to 512, balancing variance reduction against computation cost.

4.6.3 Convergence Analysis

SGD introduces variance from random sampling, fundamentally changing convergence behavior.

Theorem 4.14 (Convergence of SGD). *Let $f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x)$ where each f_i is convex and M-smooth. Assume bounded variance:*

$$\mathbb{E}[\|\nabla f_i(x) - \nabla f(x)\|^2] \leq \sigma^2 \quad \text{for all } x.$$

1. ***Convex case**: With step size $\eta^{(k)} = \frac{c}{\sqrt{K}}$ for appropriate c:*

$$\mathbb{E}[f(\bar{x}^{(K)})] - f^* = O\left(\frac{R^2 + \sigma^2}{\sqrt{K}}\right),$$

where $\bar{x}^{(K)} = \frac{1}{K} \sum_{k=0}^{K-1} x^{(k)}$ and $R = \|x^{(0)} - x^*\|$.

2. ***Strongly convex case**: With step size $\eta^{(k)} = \frac{2}{m(k+1)}$:*

$$\mathbb{E}[f(\bar{x}^{(K)})] - f^* = O\left(\frac{M\sigma^2}{mK}\right).$$

Proof sketch for convex case. The analysis parallels gradient descent but accounts for variance. Starting from:

$$\|x^{(k+1)} - x^*\|^2 = \|x^{(k)} - x^*\|^2 - 2\eta^{(k)}\nabla f_{i_k}(x^{(k)})^T(x^{(k)} - x^*) + (\eta^{(k)})^2\|\nabla f_{i_k}(x^{(k)})\|^2.$$

Taking expectations and using unbiasedness $\mathbb{E}[\nabla f_{i_k}(x)] = \nabla f(x)$:

$$\mathbb{E}[\|x^{(k+1)} - x^*\|^2] \leq \mathbb{E}[\|x^{(k)} - x^*\|^2] - 2\eta^{(k)}(f(x^{(k)}) - f^*) + (\eta^{(k)})^2\mathbb{E}[\|\nabla f_{i_k}(x^{(k)})\|^2].$$

The key difference from deterministic GD: $\mathbb{E}[\|\nabla f_{i_k}(x)\|^2] = \|\nabla f(x)\|^2 + \sigma^2$, introducing an extra σ^2 term that doesn't vanish near the optimum. This necessitates diminishing step sizes. □

Table 4.4: Comparison of full gradient descent vs SGD.

	Full GD	**SGD**	**Winner**
Cost per iteration	$O(n)$	$O(1)$	SGD
Iterations for ϵ (convex, smooth)	$O(1/\epsilon)$	$O(1/\epsilon^2)$	GD
Total cost for ϵ	$O(n/\epsilon)$	$O(1/\epsilon^2)$	Depends

When is SGD faster? SGD wins when $1/\epsilon^2 < n/\epsilon$, i.e., when $n > 1/\epsilon$. For modern datasets with millions of samples and moderate accuracy requirements, SGD is dramatically faster.

Example 4.15 (SGD vs GD cost)**.** Consider $n = 10^6$ samples and target accuracy $\epsilon = 10^{-3}$:

- Full GD: $O(n/\epsilon) = O(10^6/10^{-3}) = O(10^9)$ gradient evaluations
- SGD: $O(1/\epsilon^2) = O(10^6)$ gradient evaluations

SGD is 1000× faster!

4.6.4 Practical Considerations

Step size schedules. Common choices include:

- $1/\sqrt{k}$ **decay**: $\eta^{(k)} = \eta_0/\sqrt{k+1}$. Theoretically motivated.
- $1/k$ **decay**: $\eta^{(k)} = \eta_0/(k+1)$. Faster decay, good for strongly convex.
- **Step decay**: Reduce η by factor (e.g., 10) at fixed epochs. Common in deep learning.
- **Cosine annealing**: $\eta^{(k)} = \eta_{\min} + \frac{1}{2}(\eta_0 - \eta_{\min})(1 + \cos(\pi k/K))$.

Epochs and shuffling. One **epoch** = one pass through all n data points. In practice, we shuffle the data each epoch and cycle through mini-batches sequentially, rather than sampling with replacement. This reduces variance.

Momentum in SGD. Adding momentum helps SGD navigate noisy gradients:

$$v^{(k+1)} = \beta v^{(k)} + \nabla f_{i_k}(x^{(k)}) \tag{4.15}$$

$$x^{(k+1)} = x^{(k)} - \eta^{(k)} v^{(k+1)} \tag{4.16}$$

Typical $\beta = 0.9$. This is the basis for optimizers like SGD with momentum, Adam, and RMSprop used in deep learning.

```python
import numpy as np
import matplotlib.pyplot as plt

class SGDOptimizer:
    """SGD with optional momentum and various step size schedules."""

    def __init__(self, eta0=0.1, schedule='constant', momentum=0.0):
        self.eta0 = eta0
        self.schedule = schedule
        self.momentum = momentum
        self.velocity = None

    def get_lr(self, k, K):
        """Get learning rate at iteration k of K total."""
        if self.schedule == 'constant':
            return self.eta0
        elif self.schedule == 'sqrt':
            return self.eta0 / np.sqrt(k + 1)
        elif self.schedule == 'linear':
            return self.eta0 / (k + 1)
        elif self.schedule == 'cosine':
            return self.eta0 * 0.5 * (1 + np.cos(np.pi * k / K))
        else:
            raise ValueError(f"Unknown schedule: {self.schedule}")

    def step(self, x, grad, k, K):
        """Perform one SGD step."""
        eta = self.get_lr(k, K)

        if self.velocity is None:
            self.velocity = np.zeros_like(x)
```

```
        self.velocity = self.momentum * self.velocity + grad
        return x - eta * self.velocity


def sgd_experiment():
    """Compare SGD with full GD on logistic regression."""
    np.random.seed(42)

    # Generate data
    n, d = 10000, 50
    X = np.random.randn(n, d)
    w_true = np.random.randn(d)
    y = (1 / (1 + np.exp(-X @ w_true)) > 0.5).astype(float)

    # Logistic loss and gradients
    def sigmoid(z):
        return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

    def full_loss(w):
        p = sigmoid(X @ w)
        return -np.mean(y * np.log(p + 1e-10) + (1-y) * np.log(1-p + 1e-10))

    def full_grad(w):
        p = sigmoid(X @ w)
        return X.T @ (p - y) / n

    def stochastic_grad(w, idx):
        p = sigmoid(X[idx] @ w)
        return X[idx].reshape(-1, 1) @ np.atleast_2d(p - y[idx])

    # Run full GD
    w_gd = np.zeros(d)
    eta_gd = 1.0
    losses_gd = []
    grad_evals_gd = []
    total_grads = 0

    for epoch in range(50):
        losses_gd.append(full_loss(w_gd))
        grad_evals_gd.append(total_grads)
        w_gd = w_gd - eta_gd * full_grad(w_gd)
        total_grads += n

    # Run SGD
```

```
    w_sgd = np.zeros(d)
    losses_sgd = []
    grad_evals_sgd = []
    total_grads = 0
    batch_size = 32

    for epoch in range(50):
        losses_sgd.append(full_loss(w_sgd))
        grad_evals_sgd.append(total_grads)

        perm = np.random.permutation(n)
        for start in range(0, n, batch_size):
            idx = perm[start:start+batch_size]
            p = sigmoid(X[idx] @ w_sgd)
            grad = X[idx].T @ (p - y[idx]) / len(idx)

            k = epoch * (n // batch_size) + start // batch_size
            eta = 1.0 / np.sqrt(k + 1)
            w_sgd = w_sgd - eta * grad
            total_grads += len(idx)

    # Plot
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].semilogy(losses_gd, 'b-', linewidth=2, label='Full GD')
    axes[0].semilogy(losses_sgd, 'r-', linewidth=2, label='SGD')
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Loss')
    axes[0].set_title('Convergence per Epoch')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)

    axes[1].semilogy(grad_evals_gd, losses_gd, 'b-', linewidth=2, label='Full GD')
    axes[1].semilogy(grad_evals_sgd, losses_sgd, 'r-', linewidth=2, label='SGD')
    axes[1].set_xlabel('Gradient Evaluations')
    axes[1].set_ylabel('Loss')
    axes[1].set_title('Convergence per Computation')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('sgd_vs_fullgd.pdf', bbox_inches='tight')
    plt.show()
```

```
sgd_experiment()
```

Check Your Understanding 4.6.

You have $n = 10^7$ samples and want $\epsilon = 10^{-4}$ accuracy.

(a) How many gradient evaluations does full GD need?

(b) How many does SGD need?

(c) If each gradient evaluation takes $1\mu s$, how long does each method take?

(d) With mini-batch size $b = 100$, what is the variance reduction factor?

4.7 Application: Training Neural Networks

We've developed the theory of stochastic gradient descent for problems with finite-sum structure. But does this theory matter in practice? In this section, we'll see that it absolutely does—by applying SGD to train a neural network for image classification, one of the most important applications of optimization in modern machine learning.

Our goal is twofold: first, to see how the abstract framework of Section 4.6 translates into a concrete algorithm; and second, to build intuition for why neural network training is fundamentally an optimization problem, and why SGD has become the workhorse of deep learning.

4.7.1 Problem Setup: MNIST Classification

Let's start with a classic benchmark: the MNIST dataset. This collection contains 70,000 grayscale images of handwritten digits (0–9), each consisting of 28×28 pixels. The task is **multiclass classification**: given an image, predict which digit it represents.

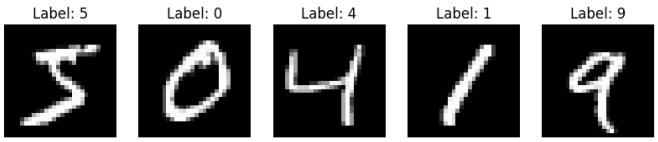

Figure 4.5: MNIST contains images of handwritten digits 0–9. Each image is $28 \times 28 = 784$ pixels. Despite its simplicity, MNIST captures the essential challenges of image classification.

Why MNIST? While modern deep learning tackles far more complex datasets, MNIST remains an excellent pedagogical example. It's small enough to train quickly on a laptop, yet rich enough to illustrate all the key ideas. Think of it as the "hello world" of machine learning.

Mathematical formulation. To turn this into an optimization problem, we need to be precise about what we're optimizing. Let $a \in \mathbb{R}^{784}$ be a flattened image (we unroll the 28×28 grid into a vector) and $y \in \{0, 1, \ldots, 9\}$ its true label. Our goal is to learn a function $h_\theta : \mathbb{R}^{784} \to \mathbb{R}^{10}$ parameterized by θ that takes an image as input and outputs a probability distribution over the 10 possible digit classes.

The key question is: what form should h_θ take, and how do we find good parameters θ?

4.7.2 The Multilayer Perceptron

The simplest neural network architecture that can learn complex patterns is the **multilayer perceptron (MLP)**. The core idea is elegant: stack together simple building blocks—linear transformations followed by nonlinear activations—to create a function that can approximate sophisticated input-output relationships.

A one-hidden-layer MLP processes an input a through the following sequence of operations:

$$z^{(1)} = W^{(1)}a + b^{(1)} \qquad \text{(linear transformation to hidden layer)} \tag{4.17}$$

$$h^{(1)} = \sigma(z^{(1)}) \qquad \text{(nonlinear activation)} \tag{4.18}$$

$$z^{(2)} = W^{(2)}h^{(1)} + b^{(2)} \qquad \text{(linear transformation to output)} \tag{4.19}$$

$$\hat{p} = \text{softmax}(z^{(2)}) \qquad \text{(convert to probabilities)} \tag{4.20}$$

Let's unpack each component:

- **First layer weights and biases:** $W^{(1)} \in \mathbb{R}^{H \times 784}$ and $b^{(1)} \in \mathbb{R}^H$, where H is the number of hidden units. This layer transforms the 784-dimensional input into an H-dimensional representation.
- **Second layer weights and biases:** $W^{(2)} \in \mathbb{R}^{10 \times H}$ and $b^{(2)} \in \mathbb{R}^{10}$. This layer maps the hidden representation to 10 output scores, one per digit class.
- **ReLU activation:** $\sigma(z) = \max(0, z)$, applied elementwise. This simple nonlinearity is crucial—without it, stacking linear layers would just give another linear function, limiting what the network can learn.
- **Softmax:** $\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}$ converts the raw output scores into a valid probability distribution (non-negative and summing to one).

The parameters we need to learn are $\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$. For a typical choice of $H = 128$ hidden units, this gives us $784 \times 128 + 128 + 128 \times 10 + 10 = 101{,}770$ parameters—a modest optimization problem by modern standards, but large enough to be interesting.

Remark 4.16 (Why nonlinearity matters). You might wonder: why bother with the ReLU activation? Here's the key insight: if we removed it, the composition of two linear transformations $W^{(2)}(W^{(1)}a + b^{(1)}) + b^{(2)}$ would just be another linear function $\tilde{W}a + \tilde{b}$. No matter how many layers we stack, we'd only be able to learn linear decision boundaries—far too restrictive for recognizing handwritten digits. The nonlinearity is what gives neural networks their expressive power.

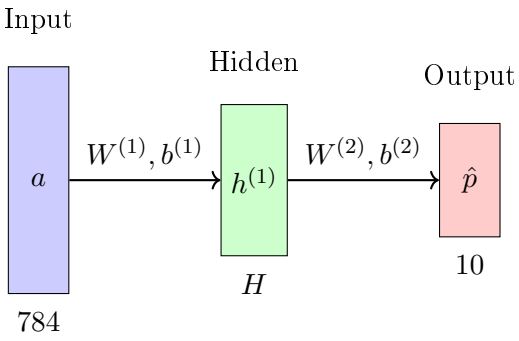

Figure 4.6: Architecture of a one-hidden-layer MLP for MNIST classification. Information flows left to right: the input image is transformed into a hidden representation, then into class probabilities.

4.7.3 The Loss Function: Cross-Entropy

Now that we've defined our model, we need a way to measure how well it's doing. This is where the **loss function** comes in—it quantifies the "badness" of our predictions, giving us something concrete to minimize.

For classification problems, the standard choice is **cross-entropy loss**. If the true label is y and our network outputs predicted probabilities $\hat{p} = (\hat{p}_0, \ldots, \hat{p}_9)$, the cross-entropy loss is:

$$\ell(\hat{p}, y) = -\log \hat{p}_y = -\log \frac{e^{z_y^{(2)}}}{\sum_{k=0}^{9} e^{z_k^{(2)}}}. \tag{4.21}$$

The intuition is simple: we want the predicted probability $\hat{p}_y$ for the correct class to be as high as possible. Since $-\log(\cdot)$ is a decreasing function:

- If $\hat{p}_y \approx 1$ (confident and correct), then $\ell \approx 0$—no penalty.
- If $\hat{p}_y \approx 0$ (confident but wrong), then $\ell \to \infty$—severe penalty.

This asymmetry is exactly what we want: the loss strongly discourages confident mistakes while being lenient with correct predictions.

The optimization problem. Given n training examples $\{(a_i, y_i)\}_{i=1}^n$, our goal is to minimize the average loss:

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(h_\theta(a_i), y_i). \tag{4.22}$$

Notice anything familiar? This is exactly the finite-sum structure from equation (4.12)! Each training example contributes one term $f_i(\theta) = \ell(h_\theta(a_i), y_i)$ to the total loss. This means all the SGD theory we developed earlier applies directly.

Remark 4.17 (Connecting to our theory). The function $f(\theta)$ is generally *non-convex* due to the nonlinear activations in the network. This means we cannot guarantee finding the global minimum. However, SGD

often finds solutions that generalize well in practice—one of the fascinating mysteries of deep learning that remains an active area of research.

4.7.4 Backpropagation: Computing Gradients Efficiently

To apply SGD, we need to compute $\nabla_\theta \ell$ for each training sample. With over 100,000 parameters, computing this gradient naively would be prohibitively expensive. Fortunately, **backpropagation** provides an elegant and efficient solution by systematically applying the chain rule.

The key insight is that neural networks have a layered structure that we can exploit. We first compute the output (the *forward pass*), then propagate gradient information backwards through the network (the *backward pass*).

Forward pass. Starting from the input, we compute and store all intermediate values:

$$a \xrightarrow{W^{(1)},b^{(1)}} z^{(1)} \xrightarrow{\sigma} h^{(1)} \xrightarrow{W^{(2)},b^{(2)}} z^{(2)} \xrightarrow{\text{softmax}} \hat{p} \xrightarrow{\ell} L$$

Storing these intermediate values is essential—we'll need them when computing gradients.

Backward pass. Now we work backwards, computing how the loss changes with respect to each quantity. The chain rule tells us how to "propagate" gradients through each operation.

Step 1: Output layer gradients. We start at the end: how does the loss change with respect to the output scores $z^{(2)}$? For cross-entropy combined with softmax, there's a remarkably clean formula:

$$\frac{\partial \ell}{\partial z_j^{(2)}} = \hat{p}_j - \mathbf{1}[j = y] = \begin{cases} \hat{p}_j - 1 & \text{if } j = y \\ \hat{p}_j & \text{if } j \neq y \end{cases}$$

In vector form, we write $\delta^{(2)} = \hat{p} - e_y$, where e_y is the one-hot encoding of the true label y. This gradient has an intuitive interpretation: it points in the direction that would increase the scores for incorrect classes and decrease the score for the correct class. SGD moves in the opposite direction.

Step 2: Second layer parameters. Using the chain rule, we obtain the gradients for $W^{(2)}$ and $b^{(2)}$:

$$\frac{\partial \ell}{\partial W^{(2)}} = \delta^{(2)} (h^{(1)})^T, \quad \frac{\partial \ell}{\partial b^{(2)}} = \delta^{(2)}$$

Step 3: Propagate through the hidden layer. To get gradients for the first layer, we need to backpropagate through the ReLU activation:

$$\delta^{(1)} = (W^{(2)})^T \delta^{(2)} \odot \sigma'(z^{(1)})$$

Here $\sigma'(z) = \mathbf{1}[z > 0]$ is the derivative of ReLU (1 where the input was positive, 0 otherwise), and $\odot$ denotes elementwise multiplication. The term $(W^{(2)})^T \delta^{(2)}$ propagates the error signal backwards through the linear transformation.

Step 4: First layer parameters. Finally, we compute:

$$\frac{\partial \ell}{\partial W^{(1)}} = \delta^{(1)} a^T, \quad \frac{\partial \ell}{\partial b^{(1)}} = \delta^{(1)}$$

The beauty of backpropagation is its efficiency: computing all these gradients takes roughly the same amount of work as a single forward pass. This makes training neural networks with millions of parameters tractable.

4.7.5 Putting It All Together: The Training Algorithm

We now have all the ingredients to train our network. Algorithm 6 shows the complete procedure, combining mini-batch SGD with backpropagation.

Algorithm 6 Training an MLP with Mini-Batch SGD

Input: Training data $\{(a_i, y_i)\}_{i=1}^n$, hidden size H, learning rate η, epochs E, batch size B
Initialize: $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ with small random values
for epoch $= 1, \ldots, E$ **do**
    Shuffle training data randomly
    for each mini-batch $\mathcal{B} = \{(a_j, y_j)\}_{j=1}^B$ **do**
        Initialize gradient accumulators: $g_{W^{(1)}}, g_{b^{(1)}}, g_{W^{(2)}}, g_{b^{(2)}} \leftarrow 0$
        for each $(a, y) \in \mathcal{B}$ **do**
            Forward pass: Compute $z^{(1)}, h^{(1)}, z^{(2)}, \hat{p}$
            Backward pass: Compute gradients via backpropagation
            Accumulate: $g_{W^{(\ell)}} \mathrel{+}= \frac{\partial \ell}{\partial W^{(\ell)}}$, $g_{b^{(\ell)}} \mathrel{+}= \frac{\partial \ell}{\partial b^{(\ell)}}$ for $\ell = 1, 2$
        end for
        SGD update: $W^{(\ell)} \leftarrow W^{(\ell)} - \frac{\eta}{B} g_{W^{(\ell)}}$, $b^{(\ell)} \leftarrow b^{(\ell)} - \frac{\eta}{B} g_{b^{(\ell)}}$ for $\ell = 1, 2$
    end for
    Evaluate and report training/validation accuracy
end for
Output: Trained parameters $\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$

A few practical notes on this algorithm:

- **Shuffling** ensures that consecutive mini-batches are independent, which is important for the SGD convergence guarantees we proved earlier.
- **Epochs** refer to complete passes through the training data. Training typically requires multiple epochs for the parameters to converge.
- **Initialization** matters: starting with values that are too large or too small can cause training to fail. A common choice is to draw entries from a Gaussian with standard deviation proportional to $1/\sqrt{\text{fan-in}}$.

```python
import numpy as np
import matplotlib.pyplot as plt

# ============================================================
# Load MNIST data (simplified - using sklearn's fetch_openml)
# ============================================================
def load_mnist():
    """Load MNIST dataset."""
    try:
        from sklearn.datasets import fetch_openml
        mnist = fetch_openml('mnist_784', version=1, as_frame=False)
        X, y = mnist.data, mnist.target.astype(int)
    except:
        # Fallback: generate synthetic data for demonstration
        print("Generating synthetic data (MNIST not available)")
        np.random.seed(42)
        n_samples = 10000
        X = np.random.randn(n_samples, 784)
        y = np.random.randint(0, 10, n_samples)

    # Normalize to [0, 1]
    X = X / 255.0

    # Train/test split
    n_train = 60000 if len(X) >= 70000 else int(0.85 * len(X))
    X_train, X_test = X[:n_train], X[n_train:]
    y_train, y_test = y[:n_train], y[n_train:]

    return X_train, y_train, X_test, y_test

# ============================================================
# Neural Network Implementation
# ============================================================
class MLP:
    """Multilayer Perceptron with one hidden layer."""

    def __init__(self, input_dim=784, hidden_dim=128, output_dim=10):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Xavier initialization
        self.W1 = np.random.randn(hidden_dim, input_dim) * np.sqrt(2.0 / input_dim)
        self.b1 = np.zeros(hidden_dim)
```

```
        self.W2 = np.random.randn(output_dim, hidden_dim) * np.sqrt(2.0 / hidden_dim)
        self.b2 = np.zeros(output_dim)

        # For storing intermediate values during forward pass
        self.cache = {}

    def relu(self, z):
        return np.maximum(0, z)

    def relu_derivative(self, z):
        return (z > 0).astype(float)

    def softmax(self, z):
        # Numerically stable softmax
        z_shifted = z - np.max(z, axis=-1, keepdims=True)
        exp_z = np.exp(z_shifted)
        return exp_z / np.sum(exp_z, axis=-1, keepdims=True)

    def forward(self, X):
        """Forward pass. X shape: (batch_size, 784)"""
        # First layer
        z1 = X @ self.W1.T + self.b1  # (batch, hidden)
        h1 = self.relu(z1)

        # Second layer
        z2 = h1 @ self.W2.T + self.b2  # (batch, 10)
        p = self.softmax(z2)

        # Cache for backward pass
        self.cache = {'X': X, 'z1': z1, 'h1': h1, 'z2': z2, 'p': p}

        return p

    def compute_loss(self, p, y):
        """Cross-entropy loss."""
        batch_size = len(y)
        # Clip for numerical stability
        p_clipped = np.clip(p, 1e-10, 1 - 1e-10)
        loss = -np.mean(np.log(p_clipped[np.arange(batch_size), y]))
        return loss

    def backward(self, y):
        """Backward pass (backpropagation)."""
        batch_size = len(y)
```

```
        X, z1, h1, z2, p = (self.cache['X'], self.cache['z1'],
                            self.cache['h1'], self.cache['z2'], self.cache['p'])

        # Output layer gradient: d_loss/d_z2 = p - one_hot(y)
        one_hot_y = np.zeros_like(p)
        one_hot_y[np.arange(batch_size), y] = 1
        delta2 = (p - one_hot_y) / batch_size  # (batch, 10)

        # Gradients for W2, b2
        grad_W2 = delta2.T @ h1  # (10, hidden)
        grad_b2 = np.sum(delta2, axis=0)  # (10,)

        # Backpropagate to hidden layer
        delta1 = (delta2 @ self.W2) * self.relu_derivative(z1)  # (batch, hidden)

        # Gradients for W1, b1
        grad_W1 = delta1.T @ X  # (hidden, 784)
        grad_b1 = np.sum(delta1, axis=0)  # (hidden,)

        return {'W1': grad_W1, 'b1': grad_b1, 'W2': grad_W2, 'b2': grad_b2}

    def update(self, grads, lr):
        """SGD update."""
        self.W1 -= lr * grads['W1']
        self.b1 -= lr * grads['b1']
        self.W2 -= lr * grads['W2']
        self.b2 -= lr * grads['b2']

    def predict(self, X):
        """Predict class labels."""
        p = self.forward(X)
        return np.argmax(p, axis=1)

    def accuracy(self, X, y):
        """Compute classification accuracy."""
        predictions = self.predict(X)
        return np.mean(predictions == y)


def train_mlp(X_train, y_train, X_test, y_test,
              hidden_dim=128, lr=0.1, batch_size=64, epochs=20):
    """Train MLP with SGD."""

    model = MLP(hidden_dim=hidden_dim)
```

```
    n_train = len(X_train)

    history = {
        'train_loss': [], 'train_acc': [],
        'test_acc': [], 'epoch': []
    }

    for epoch in range(epochs):
        # Shuffle training data
        perm = np.random.permutation(n_train)
        X_shuffled = X_train[perm]
        y_shuffled = y_train[perm]

        epoch_loss = 0
        n_batches = 0

        # Mini-batch SGD
        for start in range(0, n_train, batch_size):
            end = min(start + batch_size, n_train)
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]

            # Forward pass
            p = model.forward(X_batch)
            loss = model.compute_loss(p, y_batch)
            epoch_loss += loss
            n_batches += 1

            # Backward pass
            grads = model.backward(y_batch)

            # SGD update
            model.update(grads, lr)

        # Record metrics
        train_loss = epoch_loss / n_batches
        train_acc = model.accuracy(X_train[:5000], y_train[:5000])  # Subsample for speed
        test_acc = model.accuracy(X_test, y_test)

        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)
        history['test_acc'].append(test_acc)
        history['epoch'].append(epoch + 1)
```

```
        print(f"Epoch {epoch+1:2d}: Loss={train_loss:.4f}, "
              f"Train Acc={train_acc:.4f}, Test Acc={test_acc:.4f}")

    return model, history


def visualize_training(history):
    """Plot training curves."""
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    # Loss
    axes[0].plot(history['epoch'], history['train_loss'], 'b-', linewidth=2)
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Training Loss')
    axes[0].set_title('Cross-Entropy Loss')
    axes[0].grid(True, alpha=0.3)

    # Accuracy
    axes[1].plot(history['epoch'], history['train_acc'], 'b-', linewidth=2, label='Train')
    axes[1].plot(history['epoch'], history['test_acc'], 'r-', linewidth=2, label='Test')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('Accuracy')
    axes[1].set_title('Classification Accuracy')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('mnist_training.pdf', bbox_inches='tight')
    plt.show()


# Main execution
if __name__ == "__main__":
    print("Loading MNIST...")
    X_train, y_train, X_test, y_test = load_mnist()
    print(f"Training samples: {len(X_train)}, Test samples: {len(X_test)}")

    print("\nTraining MLP with SGD...")
    model, history = train_mlp(X_train, y_train, X_test, y_test,
                               hidden_dim=128, lr=0.5, batch_size=64, epochs=20)

    visualize_training(history)

    print(f"\nFinal test accuracy: {history['test_acc'][-1]:.4f}")
```

4.7.6 Observations and Insights

Running the code above typically achieves $\approx 97\%$ test accuracy after 20 epochs. Several observations connect to our theory:

1. **SGD works despite non-convexity.** Neural network loss functions are highly non-convex, yet SGD finds good solutions. The theory in this chapter provides intuition (descent along negative gradient), even if convergence guarantees don't directly apply.

2. **Step size matters.** Too large $\to$ divergence or oscillation. Too small $\to$ slow convergence. In practice, learning rate tuning is crucial.

3. **Mini-batches balance variance and speed.** Batch size 64 gives reasonable gradient estimates while enabling efficient GPU computation.

4. **Momentum helps.** Adding momentum (not shown in basic code) typically improves convergence, consistent with Section 4.4.

Check Your Understanding 4.7.

Consider the MLP architecture with $H = 128$ hidden units.

(a) How many total parameters does the network have?

(b) For a mini-batch of size 64, what is the shape of the gradient $\nabla_{W^{(1)}} \ell$?

(c) Why do we use softmax rather than just taking $\arg\max$ *of $z^{(2)}$?*

(d) What would happen if we used $\sigma(z) = z$ (identity) instead of ReLU?

4.8 Exercises

1. **(Gradient computation)** Consider the logistic loss for a single sample, $f(x) = \log(1+e^{a^T x})$, where $a \in \mathbb{R}^n$ is fixed.

   (a) Compute the gradient $\nabla f(x)$.

   (b) Compute the Hessian $\nabla^2 f(x)$ and show that f is convex.

   (c) Find the smoothness constant M in terms of $\|a\|$.

2. **(Convergence analysis)** Consider gradient descent on $f(x) = \frac{1}{2}(x_1^2 + \gamma x_2^2)$ with $\gamma > 1$.

   (a) Determine the strong convexity constant m, smoothness constant M, and condition number κ.

   (b) With step size $\eta = 1/\gamma$, write the update $x^{(k+1)}$ in terms of $x^{(k)}$.

   (c) Express $x^{(k)}$ explicitly in terms of $x^{(0)}$.

(d) For $\gamma = 100$ and $\|x^{(0)}\| = 1$, how many iterations are needed to achieve $\|x^{(k)}\| \leq 10^{-6}$?

3. **(Backtracking line search)** Prove that for an M-smooth function, backtracking line search with parameters $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$ always accepts a step size $\eta \geq \beta/M$.

4. **(Exact line search)** Consider the quadratic $f(x) = \frac{1}{2}x^T Q x$ with $Q \succ 0$.

   (a) Show that exact line search yields the step size

   $$\eta^{(k)} = \frac{\|\nabla f(x^{(k)})\|^2}{\nabla f(x^{(k)})^T Q \nabla f(x^{(k)})}.$$

   (b) Implement gradient descent with exact line search and compare its convergence to gradient descent with the fixed step size $\eta = 1/\lambda_{\max}(Q)$.

5. **(Subdifferential of a maximum)** Let $f(x) = \max_{i=1,\ldots,m}(a_i^T x + b_i)$ for vectors $a_i \in \mathbb{R}^n$ and scalars b_i.

   (a) Prove that f is convex.

   (b) Find $\partial f(x)$ when the maximum is achieved by a unique index i^*.

   (c) Find $\partial f(x)$ when the maximum is achieved by the set of indices $I(x) = \{i : a_i^T x + b_i = f(x)\}$.

   (d) Characterize the minimizer x^* and verify that $0 \in \partial f(x^*)$.

6. **(Subgradient method implementation)** Implement the subgradient method for the ℓ_1 regression problem:

   $$\min_{x \in \mathbb{R}^n} \|Ax - b\|_1.$$

   Compare the following step size rules:

   (a) $\eta^{(k)} = c/\sqrt{k}$

   (b) $\eta^{(k)} = c/k$

   (c) Polyak step: $\eta^{(k)} = (f(x^{(k)}) - f^*)/\|g^{(k)}\|^2$ (requires knowing f^*)

   For each rule, plot the convergence of f_k^{best} and discuss your findings.

7. **(Nesterov's method)** Implement Nesterov's accelerated gradient method and verify the $O(1/k^2)$ convergence rate empirically on:

   (a) A quadratic $f(x) = \frac{1}{2}x^T Q x$ with condition number $\kappa = 1000$.

   (b) Logistic regression: $f(x) = \frac{1}{n}\sum_{i=1}^n \log(1 + e^{-y_i a_i^T x})$.

   In each case, compare with standard gradient descent and plot $f(x^{(k)}) - f^*$ versus k on a log-log scale.

8. **(Variance of stochastic gradients)** Consider the least squares objective $f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$ where $f_i(x) = \frac{1}{2}(a_i^T x - b_i)^2$.

   (a) Compute $\nabla f_i(x)$ and $\nabla f(x)$.

   (b) Compute the variance $\mathbb{E}_i[\|\nabla f_i(x) - \nabla f(x)\|^2]$, where the expectation is over a uniformly random index i.

   (c) How does the variance of the mini-batch gradient estimator scale with batch size b?

9. **(SGD convergence proof)** Prove Theorem 4.14 for the convex case in full detail. Clearly identify where the bounded variance assumption is used.

10. **(Learning rate schedules)** Implement SGD with the following learning rate schedules and compare their performance on logistic regression:

    (a) Constant: $\eta^{(k)} = \eta_0$

    (b) Inverse square root: $\eta^{(k)} = \eta_0/\sqrt{k+1}$

    (c) Step decay: $\eta^{(k)} = \eta_0 \cdot 0.1^{\lfloor k/T \rfloor}$ for period T

    (d) Cosine annealing: $\eta^{(k)} = \frac{\eta_0}{2}\left(1 + \cos\left(\frac{\pi k}{K}\right)\right)$

    Which schedule performs best? Does the answer depend on the total number of epochs?

11. **(Backpropagation with two hidden layers)** Derive the backpropagation formulas for a network with two hidden layers:

$$a \xrightarrow{W^{(1)},b^{(1)}} z^{(1)} \xrightarrow{\sigma} h^{(1)} \xrightarrow{W^{(2)},b^{(2)}} z^{(2)} \xrightarrow{\sigma} h^{(2)} \xrightarrow{W^{(3)},b^{(3)}} z^{(3)} \xrightarrow{\text{softmax}} \hat{p}.$$

    Write out the gradients $\frac{\partial \ell}{\partial W^{(\ell)}}$ and $\frac{\partial \ell}{\partial b^{(\ell)}}$ for $\ell = 1, 2, 3$.

12. **(Activation functions)** Modify the MLP implementation from Section 4.7 to use each of the following activation functions:

    (a) Sigmoid: $\sigma(z) = 1/(1+e^{-z})$

    (b) Tanh: $\sigma(z) = \tanh(z)$

    (c) Leaky ReLU: $\sigma(z) = \max(0.01z, z)$

    Compare training dynamics (loss curves) and final test accuracy. Which activation trains fastest? Which achieves the best accuracy?

13. **(Weight decay)** Add ℓ_2 regularization (weight decay) to the MLP objective:

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(h_\theta(a_i), y_i) + \frac{\lambda}{2}\left(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2\right).$$

(a) Derive the modified gradients $\frac{\partial f}{\partial W^{(1)}}$ and $\frac{\partial f}{\partial W^{(2)}}$.

(b) Implement weight decay and train the network for several values of $\lambda \in \{0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$.

(c) Plot training accuracy and test accuracy versus λ. What value of λ gives the best test accuracy?

14. **(Batch normalization)** Research batch normalization [**?**] and implement it for the hidden layer of the MLP. Compare convergence speed and final accuracy with and without batch normalization. How does batch normalization affect the sensitivity to learning rate?

15. **(Accelerated stochastic methods)** Standard momentum does not improve the convergence rate of SGD due to gradient noise. Research variance-reduced methods such as SVRG or accelerated variants like Katyusha.

(a) Explain why naively adding momentum to SGD fails to achieve acceleration.

(b) Describe the key idea behind variance reduction that enables acceleration.

(c) What convergence rate does Katyusha achieve for smooth, strongly convex finite-sum problems?

Chapter 5

First Order Methods for Constrained Optimization

The previous chapter developed gradient methods for unconstrained problems. In practice, most optimization problems include constraints—budget limits, physical bounds, probability requirements, or structural properties like sparsity. This chapter extends first-order methods to handle constraints efficiently.

We consider problems of the form:

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) + g(x) \\ \text{subject to} & x \in \mathcal{X}, \end{array} \tag{5.1}$$

where f is smooth and convex, g is convex but possibly non-smooth (e.g., $\|x\|_1$), and $\mathcal{X}$ is a closed convex set. Different problem structures call for different algorithms:

- **Projected gradient descent** (Section 5.1): When $g = 0$ and projection onto $\mathcal{X}$ is cheap.
- **Proximal gradient methods** (Section 5.2): When $\mathcal{X} = \mathbb{R}^n$ but g is non-smooth.
- **Dual and ADMM methods** (Section 5.3): When the problem decomposes across variables or constraints.
- **Primal-dual methods** (Section 5.5): For saddle-point formulations and general linear constraints.

5.1 Projected Gradient Descent

We begin with the simplest constrained setting:

$$\underset{x \in \mathcal{X}}{\text{minimize}}\, f(x), \tag{5.2}$$

where f is convex and differentiable, and $\mathcal{X} \subseteq \mathbb{R}^n$ is a closed convex set.

5.1.1 The Projection Operator

The key tool is the **Euclidean projection** onto $\mathcal{X}$:

$$\Pi_{\mathcal{X}}(y) = \arg\min_{x \in \mathcal{X}} \|x - y\|^2. \tag{5.3}$$

Proposition 5.1 (Properties of projection)**.** *Let $\mathcal{X}$ be closed and convex. Then:*

1. ***Existence and uniqueness:*** *$\Pi_{\mathcal{X}}(y)$ exists and is unique for all $y \in \mathbb{R}^n$.*
2. ***Characterization:*** *$x^* = \Pi_{\mathcal{X}}(y)$ if and only if $(y - x^*)^T(x - x^*) \leq 0$ for all $x \in \mathcal{X}$.*
3. ***Non-expansiveness:*** *$\|\Pi_{\mathcal{X}}(y_1) - \Pi_{\mathcal{X}}(y_2)\| \leq \|y_1 - y_2\|$ for all y_1, y_2.*
4. ***Fixed points:*** *$\Pi_{\mathcal{X}}(x) = x$ if and only if $x \in \mathcal{X}$.*

Proof. We prove property 2. By first-order optimality for the convex problem (5.3):

$$0 \in \partial \left[\|x - y\|^2 + \delta_{\mathcal{X}}(x) \right] = 2(x^* - y) + N_{\mathcal{X}}(x^*),$$

where $N_{\mathcal{X}}(x^*) = \{v : v^T(x - x^*) \leq 0, \forall x \in \mathcal{X}\}$ is the normal cone. This gives $(y - x^*)^T(x - x^*) \leq 0$ for all $x \in \mathcal{X}$. □

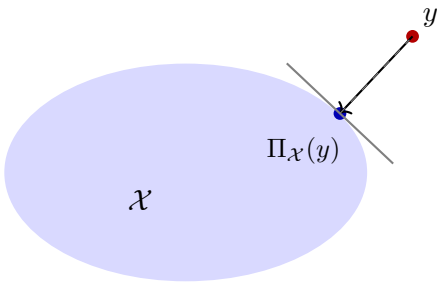

Figure 5.1: The projection $\Pi_{\mathcal{X}}(y)$ is the closest point in $\mathcal{X}$ to y. The vector $y - \Pi_{\mathcal{X}}(y)$ is perpendicular to the supporting hyperplane (gray line) at the projection point.

5.1.2 Common Projections

The practicality of projected gradient descent depends on computing projections efficiently. Table 5.1 lists common cases with closed-form solutions.

Example 5.2 (Box projection)**.** For $\mathcal{X} = \{x : 0 \leq x \leq 1\}^n$ (the unit hypercube):

$$[\Pi_{\mathcal{X}}(y)]_i = \begin{cases} 0 & y_i < 0 \\ y_i & 0 \leq y_i \leq 1 \\ 1 & y_i > 1 \end{cases}$$

Table 5.1: Projections onto common convex sets.

Set $\mathcal{X}$	**Definition**	**Projection $\Pi_{\mathcal{X}}(y)$**
Box	$\{x : l \le x \le u\}$	$[\Pi_{\mathcal{X}}(y)]_i = \min(\max(y_i, l_i), u_i)$
ℓ_2-ball	$\{x : \|x\| \le r\}$	$\begin{cases} y & \|y\| \le r \\ r \cdot y/\|y\| & \|y\| > r \end{cases}$
Halfspace	$\{x : a^T x \le b\}$	$y - \frac{(a^T y - b)^+}{\|a\|^2} a$
Affine	$\{x : Ax = b\}$	$y - A^T(AA^T)^{-1}(Ay - b)$
Simplex	$\{x : x \ge 0, \mathbf{1}^T x = 1\}$	See Algorithm 7

This is simply clipping each coordinate to $[0, 1]$.

Example 5.3 (Simplex projection)**.** The probability simplex $\Delta_n = \{x \in \mathbb{R}^n : x \ge 0, \sum_i x_i = 1\}$ arises in probability distributions, portfolio weights, and attention mechanisms. The projection requires finding a threshold τ such that $[\Pi_{\Delta_n}(y)]_i = (y_i - \tau)^+$ sums to 1.

Algorithm 7 Projection onto the Simplex

1: **Input:** Vector $y \in \mathbb{R}^n$
2: Sort y to get $y_{(1)} \ge y_{(2)} \ge \cdots \ge y_{(n)}$
3: Find $\rho = \max\left\{j : y_{(j)} - \frac{1}{j}\left(\sum_{i=1}^{j} y_{(i)} - 1\right) > 0\right\}$
4: Set $\tau = \frac{1}{\rho}\left(\sum_{i=1}^{\rho} y_{(i)} - 1\right)$
5: **return** x with $x_i = \max(y_i - \tau, 0)$

```python
import numpy as np

def project_box(y, lower, upper):
    """Project onto box constraints [lower, upper]."""
    return np.clip(y, lower, upper)

def project_l2_ball(y, radius=1.0):
    """Project onto l2 ball of given radius."""
    norm_y = np.linalg.norm(y)
    if norm_y <= radius:
        return y
    return radius * y / norm_y

def project_simplex(y):
    """Project onto probability simplex {x >= 0, sum(x) = 1}."""
    n = len(y)
    # Sort in descending order
    y_sorted = np.sort(y)[::-1]
```

```
    # Find rho
    cumsum = np.cumsum(y_sorted)
    rho = np.where(y_sorted - (cumsum - 1) / np.arange(1, n+1) > 0)[0][-1] + 1

    # Compute threshold
    tau = (cumsum[rho-1] - 1) / rho

    return np.maximum(y - tau, 0)

def project_affine(y, A, b):
    """Project onto affine set {x : Ax = b}."""
    # x = y - A^T (A A^T)^{-1} (Ay - b)
    residual = A @ y - b
    correction = np.linalg.solve(A @ A.T, residual)
    return y - A.T @ correction

# Test projections
np.random.seed(42)
y = np.array([0.5, 1.2, -0.3, 0.8])

print("Original y:", y)
print("Box [0,1]:", project_box(y, 0, 1))
print("L2 ball r=1:", project_l2_ball(y))
print("Simplex:", project_simplex(y))
print("Sum of simplex projection:", project_simplex(y).sum())
```

5.1.3 The Algorithm

Projected gradient descent combines a gradient step with projection:

$$x^{(k+1)} = \Pi_{\mathcal{X}}\left(x^{(k)} - \eta^{(k)}\nabla f(x^{(k)})\right). \tag{5.4}$$

Algorithm 8 Projected Gradient Descent

1: **Input:** Initial $x^{(0)} \in \mathcal{X}$, step sizes $\{\eta^{(k)}\}$, iterations K
2: **for** $k = 0, 1, \ldots, K-1$ **do**
3: $\quad y^{(k)} \leftarrow x^{(k)} - \eta^{(k)}\nabla f(x^{(k)})$ ▷ Gradient step
4: $\quad x^{(k+1)} \leftarrow \Pi_{\mathcal{X}}(y^{(k)})$ ▷ Projection step
5: **end for**
6: **return** $x^{(K)}$

Interpretation. The algorithm alternates between:

1. Moving in the negative gradient direction (as in unconstrained GD)
2. "Repairing" feasibility by projecting back onto $\mathcal{X}$

Equivalently, projected GD minimizes a local quadratic model subject to constraints:

$$x^{(k+1)} = \arg\min_{x \in \mathcal{X}} \left\{ f(x^{(k)}) + \nabla f(x^{(k)})^T (x - x^{(k)}) + \frac{1}{2\eta^{(k)}} \|x - x^{(k)}\|^2 \right\}.$$

5.1.4 Convergence Analysis

The convergence rates mirror the unconstrained case, with projection adding no penalty (thanks to non-expansiveness).

Theorem 5.4 (Convergence of projected gradient descent)**.** *Let f be convex and M-smooth, and let $\mathcal{X}$ be closed and convex. With step size $\eta = 1/M$:*

1. ***Convex case:***
$$f(x^{(k)}) - f^* \leq \frac{M\|x^{(0)} - x^*\|^2}{2k}.$$
2. ***Strongly convex case:*** *If f is additionally m-strongly convex:*
$$\|x^{(k)} - x^*\|^2 \leq \left(1 - \frac{m}{M}\right)^k \|x^{(0)} - x^*\|^2.$$

Proof sketch. The key insight is that projection onto a convex set is non-expansive. For any $x^* \in \mathcal{X}$:

$$\begin{aligned} \|x^{(k+1)} - x^*\|^2 &= \|\Pi_{\mathcal{X}}(x^{(k)} - \eta \nabla f(x^{(k)})) - \Pi_{\mathcal{X}}(x^*)\|^2 \\ &\leq \|x^{(k)} - \eta \nabla f(x^{(k)}) - x^*\|^2, \end{aligned}$$

where we used that $x^* \in \mathcal{X}$ implies $\Pi_{\mathcal{X}}(x^*) = x^*$. The rest follows as in unconstrained gradient descent (Theorem 4.3). □

```python
import numpy as np
import matplotlib.pyplot as plt

def projected_gradient_descent(f, grad_f, project, x0, eta, max_iter=1000, tol=1e-8):
    """
    Projected gradient descent.

    Parameters:
        f: objective function
        grad_f: gradient function
        project: projection function onto constraint set
```

```
        x0: initial point (should be feasible)
        eta: step size (or function of iteration)
        max_iter: maximum iterations
        tol: tolerance for stopping
    """
    x = project(x0)  # Ensure feasibility
    history = {'f': [f(x)], 'x': [x.copy()]}

    for k in range(max_iter):
        grad = grad_f(x)

        # Step size (constant or callable)
        step = eta if not callable(eta) else eta(k)

        # Gradient step then projection
        y = x - step * grad
        x_new = project(y)

        history['f'].append(f(x_new))
        history['x'].append(x_new.copy())

        # Check convergence
        if np.linalg.norm(x_new - x) < tol:
            break

        x = x_new

    return x, history

# Example: minimize quadratic over simplex
def demo_projected_gd():
    np.random.seed(42)
    n = 10

    # f(x) = 0.5 * x'Qx + c'x
    Q = np.eye(n) + 0.5 * np.random.randn(n, n)
    Q = Q @ Q.T  # Make positive definite
    c = np.random.randn(n)

    f = lambda x: 0.5 * x @ Q @ x + c @ x
    grad_f = lambda x: Q @ x + c

    # Constraint: probability simplex
    project = project_simplex
```

```
    # Initial point
    x0 = np.ones(n) / n  # Uniform distribution

    # Run projected GD
    M = np.max(np.linalg.eigvalsh(Q))
    eta = 1.0 / M

    x_opt, history = projected_gradient_descent(f, grad_f, project, x0, eta, max_iter=500)

    # Plot convergence
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    f_star = min(history['f'])
    axes[0].semilogy([fval - f_star + 1e-16 for fval in history['f']], 'b-', linewidth=2)
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('$f(x^{(k)}) - f^*$')
    axes[0].set_title('Convergence')
    axes[0].grid(True, alpha=0.3)

    # Plot trajectory of first two coordinates
    trajectory = np.array(history['x'])
    axes[1].plot(trajectory[:, 0], trajectory[:, 1], 'b.-', alpha=0.7)
    axes[1].plot(trajectory[0, 0], trajectory[0, 1], 'go', markersize=10, label='Start')
    axes[1].plot(trajectory[-1, 0], trajectory[-1, 1], 'r*', markersize=15, label='End')
    axes[1].set_xlabel('$x_1$')
    axes[1].set_ylabel('$x_2$')
    axes[1].set_title('Trajectory (first 2 coordinates)')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('projected_gd.pdf', bbox_inches='tight')
    plt.show()

    print(f"Optimal x (should sum to 1): {x_opt}")
    print(f"Sum: {x_opt.sum():.6f}, Min: {x_opt.min():.6f}")

demo_projected_gd()
```

Check Your Understanding 5.1.

Consider minimizing $f(x) = \frac{1}{2}\|x - c\|^2$ over the unit ℓ_2-ball $\mathcal{X} = \{x : \|x\| \leq 1\}$.

(a) What is the unconstrained minimizer? When is it feasible?

(b) What is the constrained minimizer when $\|c\| > 1$?

(c) Write one iteration of projected gradient descent with step size η.

(d) For what step sizes does projected GD converge in one iteration?

5.2 Proximal Operators and Proximal Gradient Methods

Many problems have **composite** structure:

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\, f(x) + g(x), \tag{5.5}$$

where f is convex and smooth, but g is convex and possibly *non-smooth*. The classic example is LASSO:

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\, \frac{1}{2}\|Ax - b\|^2 + \lambda\|x\|_1,$$

where $f(x) = \frac{1}{2}\|Ax - b\|^2$ is smooth and $g(x) = \lambda\|x\|_1$ is non-smooth.

Gradient descent cannot be directly applied because g is not differentiable. The **proximal gradient method** elegantly handles this structure.

5.2.1 The Proximal Operator

Definition 5.5 (Proximal operator)**.** The **proximal operator** of a convex function g with parameter $\eta > 0$ is:

$$\text{prox}_{\eta g}(y) = \arg\min_{x\in\mathbb{R}^n} \left\{ g(x) + \frac{1}{2\eta}\|x - y\|^2 \right\}. \tag{5.6}$$

The proximal operator balances two objectives:

- Minimize $g(x)$ — push toward where g is small
- Stay close to y — don't move too far from the current point

The parameter η controls the trade-off: larger η allows moving further to reduce g.

Proposition 5.6 (Projection as proximal operator)**.** *For the indicator function* $\delta_{\mathcal{X}}(x) = \begin{cases} 0 & x \in \mathcal{X} \\ +\infty & x \notin \mathcal{X} \end{cases}$*:*

$$\text{prox}_{\eta\delta_{\mathcal{X}}}(y) = \Pi_{\mathcal{X}}(y).$$

Proof. The problem $\min_x\{\delta_{\mathcal{X}}(x) + \frac{1}{2\eta}\|x - y\|^2\}$ requires $x \in \mathcal{X}$ (otherwise objective is $+\infty$), and among feasible points minimizes $\|x - y\|^2$. This is exactly the projection. □

Thus, projection is a special case of the proximal operator. Table 5.2 lists proximal operators for common functions.

Table 5.2: Common proximal operators.

Function $g(x)$	**Proximal** $\text{prox}_{\eta g}(y)$	**Name**
$\delta_{\mathcal{X}}(x)$	$\Pi_{\mathcal{X}}(y)$	Projection
$\lambda\|x\|_1$	$\text{sign}(y) \odot (\|y\| - \eta\lambda)^+$	Soft-thresholding
$\frac{\lambda}{2}\|x\|^2$	$\frac{y}{1+\eta\lambda}$	Shrinkage
$\lambda\|x\|_2$	$\left(1 - \frac{\eta\lambda}{\|y\|}\right)^+ y$	Block soft-threshold
$\delta_{\{x:\|x\|\leq r\}}$	$\frac{r \cdot y}{\max(r, \|y\|)}$	Ball projection

Example 5.7 (Soft-thresholding)**.** For $g(x) = \lambda\|x\|_1 = \lambda \sum_i |x_i|$, the proximal operator is:

$$[\text{prox}_{\eta g}(y)]_i = \text{sign}(y_i) \cdot \max(|y_i| - \eta\lambda, 0) = \mathcal{S}_{\eta\lambda}(y_i),$$

called **soft-thresholding**. It shrinks each coordinate toward zero, setting small values exactly to zero.

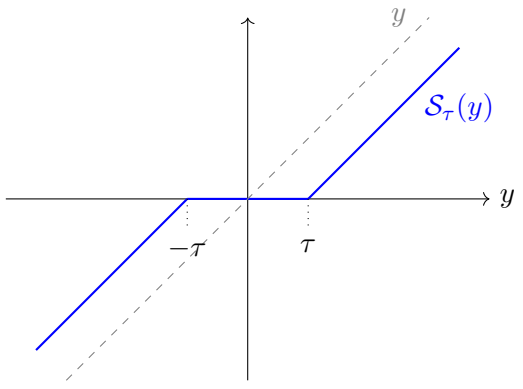

Figure 5.2: Soft-thresholding $\mathcal{S}_\tau(y) = \text{sign}(y)(|y| - \tau)^+$. Values within $[-\tau, \tau]$ are set to zero; others are shrunk by τ.

```python
import numpy as np

def prox_l1(y, eta_lambda):
    """Proximal operator for lambda * ||x||_1 (soft-thresholding)."""
    return np.sign(y) * np.maximum(np.abs(y) - eta_lambda, 0)

def prox_l2_squared(y, eta_lambda):
    """Proximal operator for (lambda/2) * ||x||^2."""
    return y / (1 + eta_lambda)

def prox_l2(y, eta_lambda):
    """Proximal operator for lambda * ||x||_2 (group soft-threshold)."""
    norm_y = np.linalg.norm(y)
```

```
    if norm_y == 0:
        return y
    return np.maximum(1 - eta_lambda / norm_y, 0) * y

# Visualize soft-thresholding
import matplotlib.pyplot as plt

y = np.linspace(-3, 3, 1000)
tau = 1.0

plt.figure(figsize=(8, 5))
plt.plot(y, y, 'k--', alpha=0.5, label='Identity')
plt.plot(y, prox_l1(y, tau), 'b-', linewidth=2, label=f'Soft-threshold ($\\tau={tau}$)')
plt.axhline(0, color='gray', linewidth=0.5)
plt.axvline(0, color='gray', linewidth=0.5)
plt.xlabel('$y$')
plt.ylabel('$\\mathcal{S}_\\tau(y)$')
plt.title('Soft-Thresholding (Proximal of $\\ell_1$ norm)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('soft_threshold.pdf', bbox_inches='tight')
plt.show()
```

5.2.2 The Proximal Gradient Method (ISTA)

The **proximal gradient method** (also called ISTA—Iterative Shrinkage-Thresholding Algorithm) solves composite problems (5.5):

$$x^{(k+1)} = \text{prox}_{\eta g}\left(x^{(k)} - \eta \nabla f(x^{(k)})\right). \tag{5.7}$$

Algorithm 9 Proximal Gradient Method (ISTA)

1: **Input:** Initial $x^{(0)}$, step size $\eta = 1/M$ where f is M-smooth
2: **for** $k = 0, 1, \ldots, K-1$ **do**
3: $\quad y^{(k)} \leftarrow x^{(k)} - \eta \nabla f(x^{(k)})$ ▷ Gradient step on smooth part
4: $\quad x^{(k+1)} \leftarrow \text{prox}_{\eta g}(y^{(k)})$ ▷ Proximal step on non-smooth part
5: **end for**
6: **return** $x^{(K)}$

Interpretation. The proximal gradient step minimizes a local model:

$$x^{(k+1)} = \arg\min_x \left\{ f(x^{(k)}) + \nabla f(x^{(k)})^T (x - x^{(k)}) + \frac{1}{2\eta}\|x - x^{(k)}\|^2 + g(x) \right\}.$$

The first three terms are the quadratic approximation of f; we add $g(x)$ exactly.

Theorem 5.8 (Convergence of proximal gradient)**.** *Let f be convex and M-smooth, and g be convex. With step size $\eta = 1/M$:*

$$f(x^{(k)}) + g(x^{(k)}) - f^* - g^* \leq \frac{M\|x^{(0)} - x^*\|^2}{2k}.$$

This is $O(1/k)$, matching gradient descent for smooth problems.

Remark 5.9 (FISTA achieves $O(1/k^2)$). The **Fast Iterative Shrinkage-Thresholding Algorithm (FISTA)** applies Nesterov acceleration to the proximal gradient method, achieving the optimal $O(1/k^2)$ rate. See Exercise 5.

5.2.3 Application: LASSO

The LASSO (Least Absolute Shrinkage and Selection Operator) problem is:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \, \frac{1}{2}\|Ax - b\|^2 + \lambda\|x\|_1. \tag{5.8}$$

Here $f(x) = \frac{1}{2}\|Ax - b\|^2$ with $\nabla f(x) = A^T(Ax - b)$ and $M = \|A^TA\|$ (largest eigenvalue), and $g(x) = \lambda\|x\|_1$ with proximal operator being soft-thresholding.

```
import numpy as np
import matplotlib.pyplot as plt

def lasso_ista(A, b, lambda_reg, max_iter=1000, tol=1e-6):
    """
    Solve LASSO via ISTA (proximal gradient).

    min_x 0.5 * ||Ax - b||^2 + lambda * ||x||_1
    """
    m, n = A.shape
    x = np.zeros(n)

    # Smoothness constant
    M = np.linalg.norm(A.T @ A, 2)  # Spectral norm
    eta = 1.0 / M

    # Precompute
    AtA = A.T @ A
    Atb = A.T @ b

    history = {'objective': [], 'sparsity': []}

    for k in range(max_iter):
        # Compute objective
```

```
        residual = A @ x - b
        obj = 0.5 * np.sum(residual**2) + lambda_reg * np.sum(np.abs(x))
        history['objective'].append(obj)
        history['sparsity'].append(np.sum(np.abs(x) > 1e-8))

        # Gradient step
        grad = AtA @ x - Atb
        y = x - eta * grad

        # Proximal step (soft-thresholding)
        x_new = prox_l1(y, eta * lambda_reg)

        # Check convergence
        if np.linalg.norm(x_new - x) < tol:
            x = x_new
            break

        x = x_new

    return x, history

def demo_lasso():
    """Demonstrate LASSO for sparse recovery."""
    np.random.seed(42)

    # Problem setup: recover sparse signal
    n = 100  # Signal dimension
    m = 50   # Number of measurements
    k = 10   # Sparsity (number of nonzeros)

    # True sparse signal
    x_true = np.zeros(n)
    support = np.random.choice(n, k, replace=False)
    x_true[support] = np.random.randn(k) * 3

    # Measurement matrix and observations
    A = np.random.randn(m, n) / np.sqrt(m)
    b = A @ x_true + 0.1 * np.random.randn(m)  # Noisy measurements

    # Solve LASSO for different lambda values
    lambdas = [0.01, 0.1, 0.5]

    fig, axes = plt.subplots(2, 3, figsize=(14, 8))
```

```
    for i, lam in enumerate(lambdas):
        x_est, history = lasso_ista(A, b, lam, max_iter=2000)

        # Plot recovered signal
        axes[0, i].stem(x_true, linefmt='b-', markerfmt='bo', basefmt='k-', label='True')
        axes[0, i].stem(x_est, linefmt='r-', markerfmt='r^', basefmt='k-', label='Recovered')
        axes[0, i].set_title(f'$\\lambda = {lam}$, nnz = {np.sum(np.abs(x_est) > 1e-4)}')
        axes[0, i].legend()
        axes[0, i].set_xlabel('Index')
        axes[0, i].set_ylabel('Value')

        # Plot convergence
        axes[1, i].semilogy(history['objective'], 'b-')
        axes[1, i].set_xlabel('Iteration')
        axes[1, i].set_ylabel('Objective')
        axes[1, i].set_title('Convergence')
        axes[1, i].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('lasso_demo.pdf', bbox_inches='tight')
    plt.show()

    print(f"True sparsity: {k}")
    print(f"Recovery error (lambda=0.1): {np.linalg.norm(x_est - x_true):.4f}")

demo_lasso()
```

Check Your Understanding 5.2.

(a) Show that $\text{prox}_{\eta g}(y) = y - \eta \nabla g(y)$ *when* g *is differentiable with* $\nabla^2 g = 0$ *(i.e.,* g *is linear).*

(b) For $g(x) = \delta_{\{c\}}(x)$ *(indicator of singleton* $\{c\}$*), what is* $\text{prox}_{\eta g}(y)$*?*

(c) Verify that soft-thresholding satisfies the optimality condition $0 \in \partial[g(x) + \frac{1}{2\eta}\|x - y\|^2]$*.*

(d) What happens to the LASSO solution as $\lambda \to 0$*? As* $\lambda \to \infty$*?*

5.3 Lagrangian Methods and Decomposition

The methods so far handle constraints via projection or proximal operators. When constraints couple variables across a distributed system, or when we want to decompose a large problem, **Lagrangian methods** provide a powerful alternative.

5.3.1 Dual Ascent

Consider the equality-constrained problem:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & Ax = b. \end{aligned} \tag{5.9}$$

The Lagrangian is $L(x, \lambda) = f(x) + \lambda^T(Ax - b)$, and the dual function is:

$$D(\lambda) = \inf_x L(x, \lambda) = \inf_x \left\{ f(x) + \lambda^T(Ax - b) \right\}.$$

If f is strictly convex, the infimum is achieved at a unique $x(\lambda)$, and the dual function is concave (though possibly non-differentiable). **Dual ascent** maximizes the dual via gradient ascent:

Algorithm 10 Dual Ascent

1: **Input:** Initial $\lambda^{(0)}$, step sizes $\{\eta^{(k)}\}$
2: **for** $k = 0, 1, \ldots$ **do**
3: $\quad x^{(k+1)} \leftarrow \arg\min_x L(x, \lambda^{(k)})$ ▷ Minimize Lagrangian
4: $\quad \lambda^{(k+1)} \leftarrow \lambda^{(k)} + \eta^{(k)}(Ax^{(k+1)} - b)$ ▷ Dual update
5: **end for**

The dual update uses the (sub)gradient of $D(\lambda)$, which is $Ax(\lambda) - b$ (the constraint residual).

Lemma 5.10 (Subgradient of dual function)**.** *If $x^* \in \arg\min_x L(x, \lambda)$, then $g = Ax^* - b$ is a subgradient of D at λ:*

$$D(\mu) \geq D(\lambda) + g^T(\mu - \lambda) \quad \textit{for all } \mu.$$

Proof. Let $x_\lambda \in \arg\min_x L(x, \lambda)$. Then:

$$\begin{aligned} D(\mu) &= \inf_x L(x, \mu) \geq L(x_\lambda, \mu) = f(x_\lambda) + \mu^T(Ax_\lambda - b) \\ &= f(x_\lambda) + \lambda^T(Ax_\lambda - b) + (\mu - \lambda)^T(Ax_\lambda - b) \\ &= D(\lambda) + (Ax_\lambda - b)^T(\mu - \lambda). \end{aligned}$$

□

When does dual ascent work? Dual ascent converges when:

1. f is strictly convex (ensures unique $x(\lambda)$)
2. Strong duality holds (ensures $D(\lambda^*) = f^*$)

However, even when it converges in the dual, the primal iterates $x^{(k)}$ may not converge or may not be feasible.

5.3.2 Application: Distributed Internet Congestion Control

A beautiful application of dual ascent is the distributed solution of network congestion control, which underlies how TCP works. Consider a network with links $\ell = 1, \dots, L$ and routes $r = 1, \dots, R$. Each route r carries traffic $x_r \geq 0$ and has utility $U_r(x_r)$ (concave, increasing). Each link ℓ has capacity b_ℓ. The problem is:

$$\begin{aligned} \underset{x \geq 0}{\text{maximize}} \quad & \sum_{r=1}^{R} U_r(x_r) \\ \text{subject to} \quad & \sum_{r:\ell \in r} x_r \leq b_\ell \quad \forall \ell = 1, \dots, L \end{aligned} \tag{5.10}$$

The Lagrangian is:

$$L(x, \lambda) = \sum_r U_r(x_r) - \sum_\ell \lambda_\ell \left(\sum_{r:\ell \in r} x_r - b_\ell \right).$$

Key observation: Decomposability. Rearranging:

$$L(x, \lambda) = \sum_r \left[U_r(x_r) - x_r \sum_{\ell \in r} \lambda_\ell \right] + \sum_\ell \lambda_\ell b_\ell.$$

For fixed λ, maximizing over x **decomposes by route**:

$$x_r^*(\lambda) = \arg\max_{x_r \geq 0} \left\{ U_r(x_r) - x_r q_r \right\},$$

where $q_r = \sum_{\ell \in r} \lambda_\ell$ is the total "price" of route r. If U_r is differentiable and strictly concave:

$$x_r^*(\lambda) = (U_r')^{-1}(q_r).$$

Algorithm 11 Distributed Congestion Control via Dual Ascent

1: **Input:** Step sizes $\{\eta^{(k)}\}$, initial $\lambda^{(0)} = 0$
2: **for** $k = 0, 1, \dots$ **do**
3: **Network** $\to$ **Users**: Send $q_r^{(k)} = \sum_{\ell \in r} \lambda_\ell^{(k)}$ to each route
4: **Users**: Each route r computes $x_r^{(k+1)} = (U_r')^{-1}(q_r^{(k)})$
5: **Users** $\to$ **Network**: Report rates $x_r^{(k+1)}$
6: **Network**: Update prices $\lambda_\ell^{(k+1)} = \left[\lambda_\ell^{(k)} + \eta^{(k)} \left(\sum_{r:\ell \in r} x_r^{(k+1)} - b_\ell \right) \right]^+$
7: **end for**

This is exactly how TCP congestion control works:

- Links signal congestion through prices (packet loss, delay)
- Users adjust rates based on perceived congestion

- The system converges to optimal resource allocation

```
import numpy as np
import matplotlib.pyplot as plt

def congestion_control_dual_ascent():
    """
    Distributed congestion control via dual ascent.
    Utility: U_r(x) = w_r * log(x) (proportional fairness)
    """
    np.random.seed(42)

    # Network topology
    n_links = 5
    n_routes = 8

    # Routing matrix: A[l,r] = 1 if route r uses link l
    A = np.array([
        [1, 1, 0, 0, 1, 0, 0, 1],  # Link 1
        [1, 0, 1, 0, 0, 1, 0, 0],  # Link 2
        [0, 1, 1, 1, 0, 0, 1, 0],  # Link 3
        [0, 0, 0, 1, 1, 1, 0, 1],  # Link 4
        [0, 0, 0, 0, 0, 0, 1, 1],  # Link 5
    ], dtype=float)

    # Link capacities
    b = np.array([10, 8, 12, 10, 6], dtype=float)

    # Route weights (utility = w * log(x))
    w = np.ones(n_routes)

    # For U(x) = w*log(x), U'(x) = w/x, so x* = w/q
    def compute_rates(q, w):
        return w / np.maximum(q, 1e-8)

    # Dual ascent
    lam = np.zeros(n_links)  # Lagrange multipliers (prices)
    eta = 0.1  # Step size

    history = {'lambda': [lam.copy()], 'x': [], 'constraint_violation': []}

    for k in range(200):
        # Route prices
        q = A.T @ lam  # q[r] = sum of lambda[l] for l in route r
```

```
        # User rate computation (distributed!)
        x = compute_rates(q, w)

        # Constraint violation
        violation = A @ x - b

        # Dual update (projected to non-negative)
        lam = np.maximum(lam + eta * violation, 0)

        history['lambda'].append(lam.copy())
        history['x'].append(x.copy())
        history['constraint_violation'].append(violation.copy())

    # Plot results
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # Lagrange multipliers (prices)
    lam_hist = np.array(history['lambda'])
    for l in range(n_links):
        axes[0, 0].plot(lam_hist[:, l], label=f'Link {l+1}')
    axes[0, 0].set_xlabel('Iteration')
    axes[0, 0].set_ylabel('$\\lambda_\\ell$ (price)')
    axes[0, 0].set_title('Lagrange Multipliers')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # Rates
    x_hist = np.array(history['x'])
    for r in range(min(4, n_routes)):  # Plot first 4 routes
        axes[0, 1].plot(x_hist[:, r], label=f'Route {r+1}')
    axes[0, 1].set_xlabel('Iteration')
    axes[0, 1].set_ylabel('$x_r$ (rate)')
    axes[0, 1].set_title('Route Rates')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # Constraint violation
    viol_hist = np.array(history['constraint_violation'])
    for l in range(n_links):
        axes[1, 0].plot(viol_hist[:, l], label=f'Link {l+1}')
    axes[1, 0].axhline(0, color='k', linestyle='--')
    axes[1, 0].set_xlabel('Iteration')
    axes[1, 0].set_ylabel('$Ax - b$ (violation)')
```

```
    axes[1, 0].set_title('Constraint Violation')
    axes[1, 0].legend()
    axes[1, 0].grid(True, alpha=0.3)

    # Final allocation
    x_final = history['x'][-1]
    axes[1, 1].bar(range(1, n_routes+1), x_final)
    axes[1, 1].set_xlabel('Route')
    axes[1, 1].set_ylabel('Rate')
    axes[1, 1].set_title('Final Rate Allocation')
    axes[1, 1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('congestion_control.pdf', bbox_inches='tight')
    plt.show()

    # Check final constraints
    print("Final link utilization:")
    for l in range(n_links):
        usage = A[l] @ x_final
        print(f"  Link {l+1}: {usage:.2f} / {b[l]:.1f} ({100*usage/b[l]:.1f}%)")

congestion_control_dual_ascent()
```

Check Your Understanding 5.3.

In the congestion control problem with $U_r(x) = \log(x)$:

(a) What is the optimal rate x_r^* as a function of the price q_r?

(b) If route r uses only one link ℓ with $\lambda_\ell = 2$, what is x_r^*?

(c) What is the economic interpretation of λ_ℓ?

(d) Why do we project λ onto the non-negative orthant?

5.3.3 Augmented Lagrangian Method

Dual ascent has limitations: it may not converge when f is not strictly convex, and even when it does, primal iterates may not be feasible. The **augmented Lagrangian method** addresses these issues by adding a quadratic penalty.

For the problem $\min_x f(x)$ s.t. $Ax = b$, define the **augmented Lagrangian**:

$$L_\rho(x, \lambda) = f(x) + \lambda^T(Ax - b) + \frac{\rho}{2}\|Ax - b\|^2, \tag{5.11}$$

where $\rho > 0$ is the **penalty parameter**.

The augmented Lagrangian adds a quadratic penalty for constraint violation. This has two benefits:

1. Makes the dual function $D_\rho(\lambda) = \inf_x L_\rho(x, \lambda)$ differentiable (under mild conditions)
2. Encourages primal feasibility even during iterations

Algorithm 12 Augmented Lagrangian Method (Method of Multipliers)

Input: Initial $\lambda^{(0)}$, penalty $\rho > 0$
for $k = 0, 1, \ldots$ **do**
  $x^{(k+1)} \leftarrow \arg\min_x L_\rho(x, \lambda^{(k)})$ ▷ Minimize augmented Lagrangian
  $\lambda^{(k+1)} \leftarrow \lambda^{(k)} + \rho(Ax^{(k+1)} - b)$ ▷ Dual update
end for

Remark 5.11. The dual update $\lambda^{(k+1)} = \lambda^{(k)} + \rho(Ax^{(k+1)} - b)$ is gradient ascent on D_ρ with step size ρ. Unlike dual ascent, the augmented Lagrangian method converges under weaker conditions and drives primal iterates toward feasibility.

Drawback. The quadratic term $\frac{\rho}{2}\|Ax - b\|^2$ couples all variables in x, destroying the decomposability that made dual ascent attractive for distributed optimization. ADMM addresses this.

5.3.4 ADMM: Alternating Direction Method of Multipliers

ADMM combines the decomposability of dual ascent with the convergence properties of the augmented Lagrangian. It applies to problems with separable structure:

$$\begin{aligned} \underset{x,\,z}{\text{minimize}} \quad & f(x) + g(z) \\ \text{subject to} \quad & Ax + Bz = c. \end{aligned} \tag{5.12}$$

The augmented Lagrangian is:

$$L_\rho(x, z, \lambda) = f(x) + g(z) + \lambda^T(Ax + Bz - c) + \frac{\rho}{2}\|Ax + Bz - c\|^2.$$

ADMM alternates between minimizing over x and z separately:

Algorithm 13 ADMM

Input: Initial $z^{(0)}$, $\lambda^{(0)}$, penalty $\rho > 0$
for $k = 0, 1, \ldots$ **do**
  $x^{(k+1)} \leftarrow \arg\min_x L_\rho(x, z^{(k)}, \lambda^{(k)})$ ▷ x-update
  $z^{(k+1)} \leftarrow \arg\min_z L_\rho(x^{(k+1)}, z, \lambda^{(k)})$ ▷ z-update
  $\lambda^{(k+1)} \leftarrow \lambda^{(k)} + \rho(Ax^{(k+1)} + Bz^{(k+1)} - c)$ ▷ Dual update
end for

Scaled form. Defining the scaled dual variable $u = \lambda/\rho$ and residual $r = Ax + Bz - c$, ADMM simplifies to:

$$x^{(k+1)} = \arg\min_x \left\{ f(x) + \frac{\rho}{2}\|Ax + Bz^{(k)} - c + u^{(k)}\|^2 \right\} \tag{5.13}$$

$$z^{(k+1)} = \arg\min_z \left\{ g(z) + \frac{\rho}{2}\|Ax^{(k+1)} + Bz - c + u^{(k)}\|^2 \right\} \tag{5.14}$$

$$u^{(k+1)} = u^{(k)} + Ax^{(k+1)} + Bz^{(k+1)} - c \tag{5.15}$$

Theorem 5.12 (ADMM convergence [**?**]). *Assume f and g are closed, proper, and convex, and the unaugmented Lagrangian has a saddle point. Then:*

1. *Residual convergence:* $Ax^{(k)} + Bz^{(k)} - c \to 0$
2. *Objective convergence:* $f(x^{(k)}) + g(z^{(k)}) \to f^* + g^*$
3. *Dual convergence:* $\lambda^{(k)} \to \lambda^*$

5.3.5 Application: LASSO via ADMM

The LASSO problem (5.8) can be written in ADMM form:

$$\begin{aligned} \underset{x,\,z}{\text{minimize}} \quad & \frac{1}{2}\|Ax - b\|^2 + \lambda\|z\|_1 \\ \text{subject to} \quad & x - z = 0. \end{aligned} \tag{5.16}$$

Here $f(x) = \frac{1}{2}\|Ax - b\|^2$ and $g(z) = \lambda\|z\|_1$.

- **x-update**: Solve $\min_x \frac{1}{2}\|Ax - b\|^2 + \frac{\rho}{2}\|x - z^{(k)} + u^{(k)}\|^2$

  This is a ridge regression problem with closed-form solution:

$$x^{(k+1)} = (A^TA + \rho I)^{-1}(A^Tb + \rho(z^{(k)} - u^{(k)}))$$

- **z-update**: $\min_z \lambda\|z\|_1 + \frac{\rho}{2}\|x^{(k+1)} - z + u^{(k)}\|^2$

  This is soft-thresholding:

$$z^{(k+1)} = \mathcal{S}_{\lambda/\rho}(x^{(k+1)} + u^{(k)})$$

```
import numpy as np
import matplotlib.pyplot as plt

def lasso_admm(A, b, lambda_reg, rho=1.0, max_iter=1000, tol=1e-6):
    """
    Solve LASSO via ADMM.
```

```
    min_x 0.5 * ||Ax - b||^2 + lambda * ||x||_1

    ADMM form: min_{x,z} 0.5*||Ax-b||^2 + lambda*||z||_1  s.t. x = z
    """
    m, n = A.shape

    # Precompute
    AtA = A.T @ A
    Atb = A.T @ b
    L = np.linalg.cholesky(AtA + rho * np.eye(n))  # For fast solves

    # Initialize
    x = np.zeros(n)
    z = np.zeros(n)
    u = np.zeros(n)  # Scaled dual variable

    history = {'objective': [], 'primal_residual': [], 'dual_residual': []}

    for k in range(max_iter):
        # x-update: solve (A'A + rho*I)x = A'b + rho*(z - u)
        x_new = np.linalg.solve(L.T, np.linalg.solve(L, Atb + rho * (z - u)))

        # z-update: soft-thresholding
        z_old = z.copy()
        z = prox_l1(x_new + u, lambda_reg / rho)

        # u-update
        u = u + x_new - z

        # Compute residuals
        primal_res = np.linalg.norm(x_new - z)
        dual_res = np.linalg.norm(rho * (z - z_old))

        # Objective
        obj = 0.5 * np.sum((A @ x_new - b)**2) + lambda_reg * np.sum(np.abs(z))

        history['objective'].append(obj)
        history['primal_residual'].append(primal_res)
        history['dual_residual'].append(dual_res)

        x = x_new

        # Check convergence
        if primal_res < tol and dual_res < tol:
```

```
            break

    return z, history  # Return z as it satisfies the sparsity constraint

def compare_ista_admm():
    """Compare ISTA and ADMM for LASSO."""
    np.random.seed(42)

    n, m = 200, 100
    k = 20  # Sparsity

    # Generate problem
    A = np.random.randn(m, n) / np.sqrt(m)
    x_true = np.zeros(n)
    x_true[np.random.choice(n, k, replace=False)] = np.random.randn(k) * 3
    b = A @ x_true + 0.1 * np.random.randn(m)

    lambda_reg = 0.1

    # Solve with both methods
    x_ista, hist_ista = lasso_ista(A, b, lambda_reg, max_iter=500)
    x_admm, hist_admm = lasso_admm(A, b, lambda_reg, rho=1.0, max_iter=500)

    # Plot comparison
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    # Objective
    axes[0].semilogy(hist_ista['objective'], 'b-', label='ISTA', linewidth=2)
    axes[0].semilogy(hist_admm['objective'], 'r-', label='ADMM', linewidth=2)
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('Objective')
    axes[0].set_title('Convergence Comparison')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)

    # Residuals (ADMM)
    axes[1].semilogy(hist_admm['primal_residual'], 'b-', label='Primal residual', linewidth=2)
    axes[1].semilogy(hist_admm['dual_residual'], 'r-', label='Dual residual', linewidth=2)
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('Residual')
    axes[1].set_title('ADMM Residuals')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)
```

```
    plt.tight_layout()
    plt.savefig('ista_vs_admm.pdf', bbox_inches='tight')
    plt.show()

    print(f"ISTA final objective: {hist_ista['objective'][-1]:.6f}")
    print(f"ADMM final objective: {hist_admm['objective'][-1]:.6f}")
    print(f"Recovery error ISTA: {np.linalg.norm(x_ista - x_true):.4f}")
    print(f"Recovery error ADMM: {np.linalg.norm(x_admm - x_true):.4f}")

compare_ista_admm()
```

5.3.6 Application: Consensus Optimization

Consider minimizing a sum of functions across B agents:

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \sum_{i=1}^{B} f_i(x), \tag{5.17}$$

where agent i has access only to f_i. This arises in distributed machine learning: f_i is the loss on agent i's local data.

Reformulation. Give each agent its own copy x_i and enforce consensus:

$$\begin{aligned} \underset{x_1,\dots,x_B,\,z}{\text{minimize}} \quad & \sum_{i=1}^{B} f_i(x_i) \\ \text{subject to} \quad & x_i = z \quad i = 1,\dots,B \end{aligned} \tag{5.18}$$

The global variable z enforces agreement. The ADMM updates are:

$$x_i^{(k+1)} = \arg\min_{x_i} \left\{ f_i(x_i) + \frac{\rho}{2}\|x_i - z^{(k)} + u_i^{(k)}\|^2 \right\} \quad \text{(local, parallel)} \tag{5.19}$$

$$z^{(k+1)} = \frac{1}{B}\sum_{i=1}^{B}(x_i^{(k+1)} + u_i^{(k)}) \quad \text{(averaging)} \tag{5.20}$$

$$u_i^{(k+1)} = u_i^{(k)} + x_i^{(k+1)} - z^{(k+1)} \quad \text{(local)} \tag{5.21}$$

Each agent solves a **local** optimization problem (embarrassingly parallel!), then they **average** their solutions to get the consensus variable z.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def consensus_admm():
    """
    Consensus optimization via ADMM.

    min sum_i f_i(x)  where f_i(x) = 0.5 * ||A_i x - b_i||^2
    """
    np.random.seed(42)

    n = 20  # Dimension
    B = 5   # Number of agents
    m_per_agent = 30  # Samples per agent

    # Generate distributed data
    x_true = np.random.randn(n)
    data = []
    for i in range(B):
        A_i = np.random.randn(m_per_agent, n)
        b_i = A_i @ x_true + 0.5 * np.random.randn(m_per_agent)
        data.append((A_i, b_i))

    # ADMM for consensus
    rho = 1.0

    # Initialize
    x = [np.zeros(n) for _ in range(B)]  # Local variables
    z = np.zeros(n)  # Global consensus variable
    u = [np.zeros(n) for _ in range(B)]  # Scaled duals

    history = {'objective': [], 'consensus_error': []}

    for k in range(100):
        # Local x-updates (parallel)
        for i in range(B):
            A_i, b_i = data[i]
            # Solve: min 0.5*||A_i x - b_i||^2 + (rho/2)*||x - z + u_i||^2
            AtA = A_i.T @ A_i
            Atb = A_i.T @ b_i
            x[i] = np.linalg.solve(AtA + rho * np.eye(n), Atb + rho * (z - u[i]))

        # z-update (averaging)
        z = np.mean([x[i] + u[i] for i in range(B)], axis=0)

        # u-updates
        for i in range(B):
```

```
            u[i] = u[i] + x[i] - z

        # Compute metrics
        total_obj = sum(0.5 * np.sum((data[i][0] @ x[i] - data[i][1])**2) for i in range(B))
        consensus_err = np.sqrt(sum(np.sum((x[i] - z)**2) for i in range(B)))

        history['objective'].append(total_obj)
        history['consensus_error'].append(consensus_err)

    # Plot
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].plot(history['objective'], 'b-', linewidth=2)
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('Total Objective')
    axes[0].set_title('Objective Convergence')
    axes[0].grid(True, alpha=0.3)

    axes[1].semilogy(history['consensus_error'], 'r-', linewidth=2)
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('$\\sum_i \\|x_i - z\\|$')
    axes[1].set_title('Consensus Error')
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('consensus_admm.pdf', bbox_inches='tight')
    plt.show()

    print(f"Recovery error: {np.linalg.norm(z - x_true):.4f}")
    print(f"Final consensus error: {history['consensus_error'][-1]:.6f}")

consensus_admm()
```

Check Your Understanding 5.4.

(a) In ADMM, what role does the penalty parameter ρ play? What happens if ρ is very small? Very large?

(b) For the LASSO ADMM formulation, why do we introduce the auxiliary variable z instead of directly minimizing over x?

(c) In consensus ADMM, what communication is required between agents and a central coordinator?

(d) How would you modify consensus ADMM for a fully decentralized setting (no central coordinator)?

5.4 Mirror Descent

Projected gradient descent uses Euclidean distance to measure progress. But is this always the best choice? For certain constraint sets—particularly the simplex—a different geometry leads to dramatically better performance. **Mirror descent** generalizes gradient descent by replacing Euclidean distance with a **Bregman divergence**.

5.4.1 Motivation: The Simplex

Consider minimizing a linear function over the probability simplex:

$$\underset{x\in\Delta_n}{\text{minimize}}\, c^T x, \quad \text{where } \Delta_n = \{x \in \mathbb{R}^n : x \geq 0, \sum_i x_i = 1\}.$$

Projected gradient descent requires projecting onto Δ_n at each iteration (Algorithm 7), which costs $O(n \log n)$ due to sorting. More fundamentally, the convergence rate depends on the *diameter* of the feasible set in the chosen geometry:

- **Euclidean diameter**: $\max_{x,y\in\Delta_n} \|x - y\|_2 = \sqrt{2}$
- **Euclidean radius**: $\max_{x\in\Delta_n} \|x - x_0\|_2$ for $x_0 = \mathbf{1}/n$ is $\sqrt{1 - 1/n} \approx 1$

But the simplex has n vertices, and moving between them requires $O(n)$ gradient steps in Euclidean geometry. Can we do better?

5.4.2 Bregman Divergence

Definition 5.13 (Bregman divergence)**.** Let $\phi : \mathcal{X} \to \mathbb{R}$ be a strictly convex, differentiable function. The **Bregman divergence** associated with ϕ is:

$$D_\phi(x, y) = \phi(x) - \phi(y) - \nabla\phi(y)^T(x - y). \tag{5.22}$$

The Bregman divergence measures the gap between $\phi(x)$ and its linear approximation at y. By convexity, $D_\phi(x, y) \geq 0$ with equality iff $x = y$.

Example 5.14 (Common Bregman divergences)**.**

1. **Euclidean**: $\phi(x) = \frac{1}{2}\|x\|_2^2$ gives $D_\phi(x, y) = \frac{1}{2}\|x - y\|_2^2$.
2. **Negative entropy**: $\phi(x) = \sum_i x_i \log x_i$ (for $x > 0$) gives the **KL divergence**:

$$D_\phi(x, y) = \sum_i x_i \log \frac{x_i}{y_i} - \sum_i x_i + \sum_i y_i = \mathrm{KL}(x\|y).$$

   For $x, y \in \Delta_n$: $D_\phi(x, y) = \sum_i x_i \log(x_i/y_i)$.

3. **Burg entropy**: $\phi(x) = -\sum_i \log x_i$ gives the **Itakura-Saito divergence**.

5.4.3 The Mirror Descent Algorithm

Mirror descent replaces the Euclidean projection with a **Bregman projection**:

$$\Pi_{\mathcal{X}}^{\phi}(y) = \arg\min_{x \in \mathcal{X}} D_\phi(x, y). \tag{5.23}$$

Algorithm 14 Mirror Descent

Input: Initial $x^{(0)} \in \mathcal{X}$, step sizes $\{\eta^{(k)}\}$, mirror map ϕ
for $k = 0, 1, \ldots, K-1$ **do**
  $y^{(k)} \leftarrow \nabla\phi(x^{(k)}) - \eta^{(k)} \nabla f(x^{(k)})$ ▷ Dual update
  $x^{(k+1)} \leftarrow \arg\min_{x \in \mathcal{X}} \left\{ D_\phi(x, x^{(k)}) + \eta^{(k)} \nabla f(x^{(k)})^T x \right\}$ ▷ Bregman projection
end for
return $x^{(K)}$

Equivalently, mirror descent can be written as:

1. **Mirror step**: Map to dual space: $\theta^{(k)} = \nabla\phi(x^{(k)})$
2. **Gradient step**: Update in dual space: $\theta^{(k+1/2)} = \theta^{(k)} - \eta^{(k)} \nabla f(x^{(k)})$
3. **Inverse mirror**: Map back: $x^{(k+1)} = \arg\min_{x \in \mathcal{X}} \{\phi(x) - (\theta^{(k+1/2)})^T x\}$

5.4.4 Entropic Mirror Descent on the Simplex

For the simplex Δ_n with negative entropy $\phi(x) = \sum_i x_i \log x_i$:

Proposition 5.15 (Entropic mirror descent update)**.** *The mirror descent update with $\phi(x) = \sum_i x_i \log x_i$ on Δ_n is:*

$$x_i^{(k+1)} = \frac{x_i^{(k)} \exp(-\eta^{(k)} [\nabla f(x^{(k)})]_i)}{\sum_j x_j^{(k)} \exp(-\eta^{(k)} [\nabla f(x^{(k)})]_j)}. \tag{5.24}$$

*This is called **exponentiated gradient descent** or **multiplicative weights update**.*

Proof. The mirror descent step solves:

$$x^{(k+1)} = \arg\min_{x \in \Delta_n} \left\{ \sum_i x_i \log \frac{x_i}{x_i^{(k)}} + \eta^{(k)} \nabla f(x^{(k)})^T x \right\}.$$

The Lagrangian for the constraint $\sum_i x_i = 1$ is:

$$L = \sum_i x_i \log \frac{x_i}{x_i^{(k)}} + \eta^{(k)} \sum_i g_i x_i + \lambda(\sum_i x_i - 1),$$

where $g = \nabla f(x^{(k)})$. Setting $\partial L/\partial x_i = 0$:

$$\log \frac{x_i}{x_i^{(k)}} + 1 + \eta^{(k)} g_i + \lambda = 0 \implies x_i = x_i^{(k)} e^{-1-\lambda-\eta^{(k)} g_i}.$$

Using $\sum_i x_i = 1$ to solve for $e^{-1-\lambda}$ gives (5.24). □

Key advantage: No projection needed! The update (5.24) automatically satisfies $x^{(k+1)} \in \Delta_n$ if $x^{(k)} \in \Delta_n$—no explicit projection required. The cost is $O(n)$ per iteration versus $O(n \log n)$ for projected gradient descent.

5.4.5 Convergence of Mirror Descent

Theorem 5.16 (Convergence of mirror descent). *Let f be convex with $\|\nabla f(x)\|_* \le G$ (dual norm), and let ϕ be σ-strongly convex w.r.t. norm $\|\cdot\|$. With step size $\eta^{(k)} = \eta$:*

$$f\left(\frac{1}{K}\sum_{k=0}^{K-1} x^{(k)}\right) - f^* \le \frac{D_\phi(x^*, x^{(0)}) + \eta G^2 K/(2\sigma)}{\eta K}.$$

With $\eta = \sqrt{2\sigma D_\phi(x^*, x^{(0)})/(G^2 K)}$:

$$f(\bar{x}^{(K)}) - f^* \le G\sqrt{\frac{2 D_\phi(x^*, x^{(0)})}{\sigma K}}.$$

Why entropy is better for the simplex.

- **Euclidean**: $D_\phi(x^*, x^{(0)}) = \frac{1}{2}\|x^* - x^{(0)}\|_2^2 \le 1$. Gradient bound in ℓ_2: $\|\nabla f\|_2 \le G$.
- **Entropy**: $D_\phi(x^*, x^{(0)}) = \mathrm{KL}(x^* \| x^{(0)}) \le \log n$ (for $x^{(0)} = \mathbf{1}/n$). Gradient bound in ℓ_∞: $\|\nabla f\|_\infty \le G$.

The entropy setup has $D_\phi = O(\log n)$ but uses the ℓ_∞ norm for gradients. For many problems on the simplex, $\|\nabla f\|_\infty$ is much smaller than $\|\nabla f\|_2$, leading to $O(\sqrt{\log n/K})$ convergence versus $O(1/\sqrt{K})$—a significant improvement in high dimensions!

```
import numpy as np
import matplotlib.pyplot as plt

def projected_gd_simplex(grad_f, x0, eta, max_iter):
    """Projected gradient descent on simplex."""
    x = x0.copy()
    history = [x.copy()]

    for k in range(max_iter):
```

```
        g = grad_f(x)
        y = x - eta * g
        x = project_simplex(y)
        history.append(x.copy())

    return x, np.array(history)

def mirror_descent_simplex(grad_f, x0, eta, max_iter):
    """Mirror descent with entropy on simplex (exponentiated gradient)."""
    x = x0.copy()
    history = [x.copy()]

    for k in range(max_iter):
        g = grad_f(x)
        # Exponentiated gradient update
        x_unnorm = x * np.exp(-eta * g)
        x = x_unnorm / np.sum(x_unnorm)
        history.append(x.copy())

    return x, np.array(history)

def compare_pgd_mirror():
    """Compare projected GD and mirror descent on simplex."""
    np.random.seed(42)
    n = 100  # Dimension

    # Problem: min c'x over simplex (optimum at vertex with smallest c_i)
    c = np.random.randn(n)
    c = c - c.min() + 0.1  # Make all positive, with one small
    opt_idx = np.argmin(c)
    x_opt = np.zeros(n)
    x_opt[opt_idx] = 1.0
    f_opt = c[opt_idx]

    f = lambda x: c @ x
    grad_f = lambda x: c

    # Initial point: uniform distribution
    x0 = np.ones(n) / n

    max_iter = 500

    # Run both methods
    # For PGD, use step size 1/L where L is Lipschitz constant
```

```
    # For linear f, any reasonable step size works
    eta_pgd = 0.1
    eta_mirror = 1.0  # Can be larger for mirror descent

    x_pgd, hist_pgd = projected_gd_simplex(grad_f, x0, eta_pgd, max_iter)
    x_mirror, hist_mirror = mirror_descent_simplex(grad_f, x0, eta_mirror, max_iter)

    # Compute objectives
    obj_pgd = [c @ x for x in hist_pgd]
    obj_mirror = [c @ x for x in hist_mirror]

    # Plot
    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    # Objective gap
    axes[0].semilogy([o - f_opt for o in obj_pgd], 'b-', linewidth=2, label='Projected GD')
    axes[0].semilogy([o - f_opt for o in obj_mirror], 'r-', linewidth=2, label='Mirror Descent'
    )
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('$f(x) - f^*$')
    axes[0].set_title('Objective Gap')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)

    # Entropy of iterates (measure of "concentration")
    def entropy(x):
        x_pos = x[x > 1e-10]
        return -np.sum(x_pos * np.log(x_pos))

    ent_pgd = [entropy(x) for x in hist_pgd]
    ent_mirror = [entropy(x) for x in hist_mirror]

    axes[1].plot(ent_pgd, 'b-', linewidth=2, label='Projected GD')
    axes[1].plot(ent_mirror, 'r-', linewidth=2, label='Mirror Descent')
    axes[1].axhline(0, color='k', linestyle='--', alpha=0.5, label='Vertex (entropy=0)')
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('Entropy')
    axes[1].set_title('Entropy of Iterate')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    # Final distribution
    width = 0.35
    indices = np.arange(min(20, n))  # Show first 20 components
```

```
    axes[2].bar(indices - width/2, hist_pgd[-1][:20], width, label='PGD', alpha=0.7)
    axes[2].bar(indices + width/2, hist_mirror[-1][:20], width, label='Mirror', alpha=0.7)
    axes[2].axvline(opt_idx if opt_idx < 20 else -1, color='g', linestyle='--',
                    label=f'Optimum (idx={opt_idx})')
    axes[2].set_xlabel('Component')
    axes[2].set_ylabel('Value')
    axes[2].set_title('Final Iterate (first 20 components)')
    axes[2].legend()

    plt.tight_layout()
    plt.savefig('mirror_vs_pgd.pdf', bbox_inches='tight')
    plt.show()

    print(f"Dimension: {n}")
    print(f"Optimal index: {opt_idx}")
    print(f"PGD final gap: {obj_pgd[-1] - f_opt:.6f}")
    print(f"Mirror final gap: {obj_mirror[-1] - f_opt:.6f}")
    print(f"PGD mass on optimum: {hist_pgd[-1][opt_idx]:.4f}")
    print(f"Mirror mass on optimum: {hist_mirror[-1][opt_idx]:.4f}")

compare_pgd_mirror()
```

Check Your Understanding 5.5.

(a) For $\phi(x) = \frac{1}{2}\|x\|_2^2$, show that mirror descent reduces to projected gradient descent.

(b) Why does the exponentiated gradient update (5.24) automatically satisfy $x^{(k+1)} \geq 0$?

(c) What happens in exponentiated gradient if $x_i^{(0)} = 0$ for some i?

(d) For what types of problems is mirror descent with entropy preferable to projected GD?

5.5 Primal-Dual Methods

Many constrained optimization problems can be formulated as **saddle-point problems**:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} \mathcal{L}(x, y), \tag{5.25}$$

where we minimize over primal variables x and maximize over dual variables y. **Primal-dual methods** update both simultaneously, converging to a saddle point.

5.5.1 Saddle-Point Formulation

Consider the composite problem with linear constraints:

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\, g(x) + h(Kx), \tag{5.26}$$

where g and h are convex (possibly non-smooth), and $K \in \mathbb{R}^{m\times n}$.

Using the Fenchel conjugate $h^*(y) = \sup_z\{y^T z - h(z)\}$, we have $h(z) = \sup_y\{y^T z - h^*(y)\}$, so:

$$g(x) + h(Kx) = g(x) + \max_y\{y^T Kx - h^*(y)\} = \max_y\{g(x) + y^T Kx - h^*(y)\}.$$

This gives the saddle-point formulation:

$$\min_x \max_y \left\{\mathcal{L}(x,y) = g(x) + \langle Kx, y\rangle - h^*(y)\right\}. \tag{5.27}$$

Example 5.17 (Constrained optimization as saddle point)**.** The problem $\min_x g(x)$ s.t. $Kx = b$ can be written as:

$$\min_x g(x) + \delta_{\{b\}}(Kx) = \min_x \max_y\{g(x) + y^T(Kx - b)\},$$

using $h(z) = \delta_{\{b\}}(z)$ with conjugate $h^*(y) = b^T y$.

Example 5.18 (Total variation denoising)**.** Given a noisy image $b \in \mathbb{R}^n$, recover a clean image x by solving:

$$\min_x \frac{1}{2}\|x - b\|^2 + \lambda\|\nabla x\|_1,$$

where ∇ is the discrete gradient operator and $\|\cdot\|_1$ is the anisotropic total variation. Here $g(x) = \frac{1}{2}\|x-b\|^2$, $K = \nabla$, and $h(z) = \lambda\|z\|_1$.

5.5.2 The Chambolle-Pock Algorithm

The **Chambolle-Pock algorithm** (also called **Primal-Dual Hybrid Gradient**, PDHG) solves (5.27) by alternating proximal steps on x and y:

Algorithm 15 Chambolle-Pock (Primal-Dual Hybrid Gradient)

Input: Initial $x^{(0)}$, $y^{(0)}$, step sizes $\tau, \sigma > 0$, $\theta \in [0,1]$
$\bar{x}^{(0)} \leftarrow x^{(0)}$
for $k = 0, 1, \ldots, K-1$ **do**
  $y^{(k+1)} \leftarrow \text{prox}_{\sigma h^*}(y^{(k)} + \sigma K\bar{x}^{(k)})$ ▷ Dual update
  $x^{(k+1)} \leftarrow \text{prox}_{\tau g}(x^{(k)} - \tau K^T y^{(k+1)})$ ▷ Primal update
  $\bar{x}^{(k+1)} \leftarrow x^{(k+1)} + \theta(x^{(k+1)} - x^{(k)})$ ▷ Extrapolation
end for
return $x^{(K)}, y^{(K)}$

The extrapolation step $\bar{x}^{(k+1)} = x^{(k+1)} + \theta(x^{(k+1)} - x^{(k)})$ with $\theta = 1$ is crucial for convergence.

Key insight: Moreau decomposition. Computing $\text{prox}_{\sigma h^*}$ can be done via prox_h using:

$$\text{prox}_{\sigma h^*}(y) = y - \sigma \, \text{prox}_{h/\sigma}(y/\sigma). \tag{5.28}$$

For $h(z) = \lambda\|z\|_1$: $\text{prox}_h(z) = \mathcal{S}_\lambda(z)$ (soft-thresholding), so:

$$\text{prox}_{\sigma h^*}(y) = y - \sigma \mathcal{S}_{\lambda/\sigma}(y/\sigma) = \text{proj}_{\|z\|_\infty \le \lambda}(y).$$

Theorem 5.19 (Convergence of Chambolle-Pock). *Let g and h^* be proper, convex, lower semicontinuous. If $\tau\sigma\|K\|^2 < 1$ and $\theta = 1$, then:*

1. *$(x^{(k)}, y^{(k)}) \to (x^*, y^*)$ where (x^*, y^*) is a saddle point.*
2. *For the ergodic average $(\bar{x}^{(K)}, \bar{y}^{(K)}) = \frac{1}{K}\sum_{k=1}^{K}(x^{(k)}, y^{(k)})$:*

$$\mathcal{L}(\bar{x}^{(K)}, y) - \mathcal{L}(x, \bar{y}^{(K)}) \le \frac{C}{K}$$

for all (x, y) in a bounded set.

5.5.3 Application: Constrained Least Squares

Consider the equality-constrained least squares:

$$\underset{x}{\text{minimize}} \, \frac{1}{2}\|Ax - b\|^2 \quad \text{s.t.} \quad Cx = d. \tag{5.29}$$

Saddle-point form:

$$\min_x \max_y \left\{ \frac{1}{2}\|Ax - b\|^2 + y^T(Cx - d) \right\}.$$

Here $g(x) = \frac{1}{2}\|Ax - b\|^2$, $K = C$, $h(z) = \delta_{\{d\}}(z)$, $h^*(y) = d^T y$.

- $\text{prox}_{\tau g}(v) = (A^T A + \frac{1}{\tau} I)^{-1}(A^T b + \frac{1}{\tau} v)$
- $\text{prox}_{\sigma h^*}(u) = u - \sigma d$ (since h^* is linear)

```
import numpy as np
import matplotlib.pyplot as plt

def chambolle_pock(prox_g, prox_h_conj, K, x0, y0, tau, sigma, theta=1.0, max_iter=1000):
    """
    Chambolle-Pock (PDHG) algorithm.

    Solves: min_x max_y { g(x) + <Kx, y> - h*(y) }
```

```
    Parameters:
        prox_g: proximal operator of tau*g
        prox_h_conj: proximal operator of sigma*h*
        K: linear operator (matrix or function)
        x0, y0: initial points
        tau, sigma: step sizes (must satisfy tau*sigma*||K||^2 < 1)
        theta: extrapolation parameter (typically 1)
    """
    x = x0.copy()
    y = y0.copy()
    x_bar = x.copy()

    # Handle K as matrix or operator
    if callable(K):
        Kop = K
        Ktop = lambda v: K(v, adjoint=True)
    else:
        Kop = lambda v: K @ v
        Ktop = lambda v: K.T @ v

    history = {'x': [x.copy()], 'y': [y.copy()]}

    for k in range(max_iter):
        # Dual update
        y_new = prox_h_conj(y + sigma * Kop(x_bar), sigma)

        # Primal update
        x_new = prox_g(x - tau * Ktop(y_new), tau)

        # Extrapolation
        x_bar = x_new + theta * (x_new - x)

        x, y = x_new, y_new
        history['x'].append(x.copy())
        history['y'].append(y.copy())

    return x, y, history

def demo_constrained_least_squares():
    """Solve constrained least squares with Chambolle-Pock."""
    np.random.seed(42)

    n = 50  # Variables
```

```
    m = 30  # Measurements
    p = 5   # Constraints

    # Problem: min ||Ax - b||^2 s.t. Cx = d
    A = np.random.randn(m, n)
    x_true = np.random.randn(n)
    b = A @ x_true + 0.1 * np.random.randn(m)

    C = np.random.randn(p, n)
    d = C @ x_true  # Constraints satisfied by true solution

    # Proximal operators
    # prox_{tau * g}(v) where g(x) = 0.5*||Ax - b||^2
    AtA = A.T @ A
    Atb = A.T @ b

    def prox_g(v, tau):
        # Solve (A'A + (1/tau)I)x = A'b + (1/tau)v
        return np.linalg.solve(AtA + (1/tau) * np.eye(n), Atb + (1/tau) * v)

    # prox_{sigma * h*}(u) where h*(y) = d'y
    def prox_h_conj(u, sigma):
        return u - sigma * d

    # Step sizes: need tau * sigma * ||C||^2 < 1
    norm_C = np.linalg.norm(C, 2)
    tau = 0.9 / norm_C
    sigma = 0.9 / norm_C

    # Initial points
    x0 = np.zeros(n)
    y0 = np.zeros(p)

    # Run Chambolle-Pock
    x_cp, y_cp, history = chambolle_pock(prox_g, prox_h_conj, C, x0, y0,
                                         tau, sigma, theta=1.0, max_iter=200)

    # Compute metrics
    x_hist = np.array(history['x'])
    primal_obj = [0.5 * np.sum((A @ x - b)**2) for x in x_hist]
    constraint_viol = [np.linalg.norm(C @ x - d) for x in x_hist]

    # Compare with direct solution
    # KKT: [A'A  C'] [x]   [A'b]
```

```
    #      [C    0 ] [nu] = [d  ]
    KKT = np.block([[AtA, C.T], [C, np.zeros((p, p))]])
    rhs = np.concatenate([Atb, d])
    sol = np.linalg.solve(KKT, rhs)
    x_direct = sol[:n]

    # Plot
    fig, axes = plt.subplots(1, 3, figsize=(14, 4))

    axes[0].plot(primal_obj, 'b-', linewidth=2)
    axes[0].axhline(0.5 * np.sum((A @ x_direct - b)**2), color='r', linestyle='--',
                    label='Direct solution')
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('$\\frac{1}{2}\\|Ax - b\\|^2$')
    axes[0].set_title('Primal Objective')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)

    axes[1].semilogy(constraint_viol, 'b-', linewidth=2)
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('$\\|Cx - d\\|$')
    axes[1].set_title('Constraint Violation')
    axes[1].grid(True, alpha=0.3)

    axes[2].plot(x_direct, 'ro', markersize=8, label='Direct', alpha=0.7)
    axes[2].plot(x_cp, 'b^', markersize=6, label='Chambolle-Pock', alpha=0.7)
    axes[2].set_xlabel('Component')
    axes[2].set_ylabel('Value')
    axes[2].set_title('Solution Comparison')
    axes[2].legend()
    axes[2].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('chambolle_pock_cls.pdf', bbox_inches='tight')
    plt.show()

    print(f"Direct solution objective: {0.5 * np.sum((A @ x_direct - b)**2):.6f}")
    print(f"CP solution objective: {0.5 * np.sum((A @ x_cp - b)**2):.6f}")
    print(f"Solution difference: {np.linalg.norm(x_cp - x_direct):.6f}")
    print(f"Constraint violation: {np.linalg.norm(C @ x_cp - d):.6f}")

demo_constrained_least_squares()
```

5.5.4 Application: Total Variation Denoising

Total variation (TV) denoising removes noise while preserving edges:

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\, \frac{1}{2}\|x-b\|^2 + \lambda\|Dx\|_1, \tag{5.30}$$

where D is the discrete gradient operator and $\|Dx\|_1 = \sum_i |[Dx]_i|$ is the anisotropic TV.

Saddle-point form:

$$\min_x \max_y \left\{ \frac{1}{2}\|x-b\|^2 + y^T Dx - \delta_{\|y\|_\infty \le \lambda}(y) \right\}.$$

Here:

- $g(x) = \frac{1}{2}\|x-b\|^2$ with $\text{prox}_{\tau g}(v) = \frac{v+\tau b}{1+\tau}$
- $h(z) = \lambda\|z\|_1$ with $h^*(y) = \delta_{\|y\|_\infty \le \lambda}(y)$
- $\text{prox}_{\sigma h^*}(u) = \text{proj}_{\|y\|_\infty \le \lambda}(u) = \text{clip}(u, -\lambda, \lambda)$

```
import numpy as np
import matplotlib.pyplot as plt

def tv_denoising_1d(b, lambda_tv, max_iter=500):
    """
    1D Total Variation denoising via Chambolle-Pock.

    min_x 0.5*||x - b||^2 + lambda * ||Dx||_1

    where D is the forward difference operator.
    """
    n = len(b)

    # Forward difference operator D (size (n-1) x n)
    # [Dx]_i = x_{i+1} - x_i
    def D_op(x):
        return x[1:] - x[:-1]

    def Dt_op(y):
        # Adjoint: D^T y
        result = np.zeros(n)
        result[:-1] -= y
        result[1:] += y
        return result

    # Proximal operators
    def prox_g(v, tau):
```

```
        # prox of (tau/2)||x - b||^2
        return (v + tau * b) / (1 + tau)

    def prox_h_conj(u, sigma):
        # prox of indicator ||y||_inf <= lambda
        return np.clip(u, -lambda_tv, lambda_tv)

    # Step sizes
    # ||D||_2 <= 2 for 1D finite differences
    L = 2.0
    tau = 1.0 / L
    sigma = 1.0 / L

    # Initialize
    x = b.copy()
    y = np.zeros(n - 1)
    x_bar = x.copy()

    for k in range(max_iter):
        # Dual update
        y = prox_h_conj(y + sigma * D_op(x_bar), sigma)

        # Primal update
        x_new = prox_g(x - tau * Dt_op(y), tau)

        # Extrapolation
        x_bar = 2 * x_new - x
        x = x_new

    return x

def demo_tv_denoising():
    """Demonstrate TV denoising on 1D signal."""
    np.random.seed(42)

    # Create piecewise constant signal
    n = 200
    x_true = np.zeros(n)
    x_true[30:70] = 1.0
    x_true[100:130] = -0.5
    x_true[150:180] = 0.8

    # Add noise
    noise_level = 0.3
```

```
    b = x_true + noise_level * np.random.randn(n)

    # Denoise with different lambda values
    lambdas = [0.1, 0.5, 1.0]

    fig, axes = plt.subplots(2, 2, figsize=(12, 8))

    # Original and noisy
    axes[0, 0].plot(x_true, 'g-', linewidth=2, label='Original')
    axes[0, 0].plot(b, 'b.', alpha=0.5, markersize=3, label='Noisy')
    axes[0, 0].set_title('Original and Noisy Signal')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    colors = ['r', 'orange', 'purple']
    for i, (lam, color) in enumerate(zip(lambdas, colors)):
        x_denoised = tv_denoising_1d(b, lam, max_iter=500)

        ax = axes[(i+1) // 2, (i+1) % 2]
        ax.plot(x_true, 'g-', linewidth=2, label='Original', alpha=0.7)
        ax.plot(b, 'b.', alpha=0.3, markersize=2)
        ax.plot(x_denoised, color=color, linewidth=2, label=f'TV ($\\lambda={lam}$)')
        ax.set_title(f'$\\lambda = {lam}$')
        ax.legend()
        ax.grid(True, alpha=0.3)

        mse = np.mean((x_denoised - x_true)**2)
        print(f"lambda={lam}: MSE={mse:.4f}")

    plt.tight_layout()
    plt.savefig('tv_denoising.pdf', bbox_inches='tight')
    plt.show()

demo_tv_denoising()
```

Check Your Understanding 5.6.

(a) In the Chambolle-Pock algorithm, why do we need the extrapolation step $\bar{x}^{(k+1)} = x^{(k+1)} + \theta(x^{(k+1)} - x^{(k)})$?

(b) What is the condition on step sizes τ, σ for convergence? Why does it involve $\|K\|$?

(c) For TV denoising, what happens as $\lambda \to 0$? As $\lambda \to \infty$?

(d) How does Chambolle-Pock relate to ADMM? (Hint: consider the specific structure of each.)

5.6 Summary and Method Selection

Table 5.3 summarizes the methods in this chapter.

Table 5.3: Summary of constrained first-order methods.

Method	**Problem Structure**	**Key Operation**	**Rate**
Projected GD	$\min_{x\in\mathcal{X}} f(x)$	Projection $\Pi_{\mathcal{X}}$	$O(1/k)$
Proximal GD	$\min_x f(x)+g(x)$	Prox of g	$O(1/k)$
Mirror descent	$\min_{x\in\mathcal{X}} f(x)$	Bregman projection	$O(1/\sqrt{k})$
Dual ascent	$\min_x f(x)$ s.t. $Ax=b$	x-minimization	Varies
Aug. Lagrangian	$\min_x f(x)$ s.t. $Ax=b$	x-minimization	$O(1/k)$
ADMM	$\min_{x,z} f(x)+g(z)$, $Ax+Bz=c$	x, z subproblems	$O(1/k)$
Chambolle-Pock	$\min_x g(x)+h(Kx)$	Prox of g, h^*	$O(1/k)$

Choosing the right algorithm depends on problem structure, constraint type, and computational considerations. The following tables provide guidance.

5.6.1 Problem Structure Compatibility

Table 5.4 shows which methods apply to which problem structures. A ✓ indicates the method is designed for this structure; a ○ indicates it can be adapted; a × indicates it does not apply or is inefficient.

Table 5.4: Method applicability by problem structure.

	$\min_{x\in\mathcal{X}} f(x)$	$\min_x f(x)+g(x)$	$\min_x f(x)$ s.t. $Ax=b$	$\min_x f(x)$ s.t. $Ax\leq b$	$\min_{x,z} f(x)+g(z)$, $Ax+Bz=c$	$\min_x g(x)+h(Kx)$	Distributed/separable
Projected GD	✓	○	○	○	×	×	×
Proximal GD	○	✓	×	×	×	○	×
FISTA	○	✓	×	×	×	○	×
Mirror descent	✓	×	×	×	×	×	×
Dual ascent	×	×	✓	○	○	×	✓
Aug. Lagrangian	×	×	✓	✓	○	×	×
ADMM	○	✓	✓	✓	✓	○	✓
Chambolle-Pock	○	✓	✓	✓	○	✓	○

5.6.2 Convergence Rates by Function Properties

Table 5.5 shows convergence rates for different combinations of objective properties. We use the notation: M = smoothness constant, m = strong convexity constant, $\kappa = M/m$ = condition number, G = Lipschitz constant.

Table 5.5: Convergence rates (iterations to achieve ϵ-accuracy).

Method	**Convex, Lipschitz**	**Convex, Smooth**	**Strongly Cvx, Smooth**	**Non-smooth g**
Projected GD	$O(1/\epsilon^2)$	$O(1/\epsilon)$	$O(\kappa \log(1/\epsilon))$	N/A
Proximal GD	$O(1/\epsilon^2)$	$O(1/\epsilon)$	$O(\kappa \log(1/\epsilon))$	$O(1/\epsilon)$[a]
FISTA	$O(1/\epsilon^2)$	$O(1/\sqrt{\epsilon})$	$O(\sqrt{\kappa} \log(1/\epsilon))$	$O(1/\sqrt{\epsilon})$[a]
Mirror descent	$O(1/\epsilon^2)$	$O(1/\epsilon^2)$[b]	$O(1/\epsilon^2)$[b]	N/A
Dual ascent	$O(1/\epsilon^2)$	$O(1/\epsilon)$[c]	$O(\kappa \log(1/\epsilon))$[c]	$O(1/\epsilon^2)$
Aug. Lagrangian	—	$O(1/\epsilon)$	$O(\log(1/\epsilon))$	—
ADMM	—	$O(1/\epsilon)$	$O(\log(1/\epsilon))$[d]	$O(1/\epsilon)$
Chambolle-Pock	—	$O(1/\epsilon)$	$O(\log(1/\epsilon))$[d]	$O(1/\epsilon)$

[a] When f is smooth and g is non-smooth (composite).
[b] Mirror descent doesn't exploit smoothness; use for geometry benefits only.
[c] Convergence of dual; primal recovery may require additional assumptions.
[d] Linear rate requires strong convexity or additional structure.

5.6.3 Constraint-Specific Recommendations

Table 5.6 provides recommendations based on the constraint structure.

Table 5.6: Method recommendations by constraint type.

Constraint Type	**Best Method**	**Alternative**	**Notes**
Box: $l \le x \le u$	Projected GD	Proximal GD	Projection is $O(n)$ clipping
ℓ_2-ball: $\|x\| \le r$	Projected GD	—	Projection is $O(n)$ scaling
Simplex: $x \ge 0, \mathbf{1}^T x = 1$	Mirror descent	Projected GD	Mirror: $O(n)$; PGD: $O(n \log n)$ sort
ℓ_1-ball: $\|x\|_1 \le r$	Projected GD	Mirror descent	Projection: $O(n \log n)$
Affine: $Ax = b$	ADMM, C-P	Aug. Lagrangian	Dual ascent if f strictly convex
Polyhedron: $Ax \le b$	ADMM	Aug. Lagrangian	Projection onto polyhedron expensive
ℓ_1 regularization	Proximal GD	ADMM	Prox is soft-thresholding $O(n)$
TV regularization	Chambolle-Pock	ADMM	Natural saddle-point structure
Nuclear norm	Proximal GD	Frank-Wolfe	Prox requires SVD; FW uses top singular vector
Distributed/consensus	ADMM	Dual ascent	ADMM handles non-smooth; dual needs strict cvx

5.6.4 Decision Flowchart

The following questions guide method selection:

1. **Is the problem unconstrained or with simple constraints?**
   - Unconstrained → Gradient descent (Chapter **??**)
   - Simple set constraint (box, ball) → Projected GD
   - Simplex constraint → Mirror descent (entropy)
2. **Does the objective have a non-smooth regularizer?**
   - ℓ_1, nuclear norm, TV → Proximal GD or FISTA
   - Need acceleration? → FISTA
3. **Are there linear equality/inequality constraints?**
   - Equality $Ax = b$ with smooth f → Dual ascent (if strictly convex) or ADMM
   - General linear constraints → ADMM or Chambolle-Pock
4. **Is the problem separable or distributed?**
   - Separable objective → ADMM (consensus)
   - Coupled through linear operator → Chambolle-Pock
5. **Is projection/prox expensive?**
   - Expensive projection → Consider Frank-Wolfe (if linear minimization is easy)
   - Expensive prox → Consider ADMM splitting

5.6.5 Common Pitfalls and Remedies

Table 5.7 summarizes common issues and solutions.

5.7 Exercises

Projected Gradient Descent

1. **(Projection onto intersection)** Let $\mathcal{X} = \mathcal{X}_1 \cap \mathcal{X}_2$ where projections onto $\mathcal{X}_1$ and $\mathcal{X}_2$ are easy.
   (a) Is $\Pi_{\mathcal{X}}(y) = \Pi_{\mathcal{X}_1}(\Pi_{\mathcal{X}_2}(y))$ in general? Give a counterexample.
   (b) Describe **Dykstra's algorithm** for projecting onto intersections.
   (c) Implement alternating projections and Dykstra's algorithm for $\mathcal{X}_1 = \{x : \|x\|_2 \leq 1\}$ and $\mathcal{X}_2 = \{x : \mathbf{1}^T x = 1\}$.

Table 5.7: Common pitfalls and remedies.

Symptom	**Likely Cause**	**Remedy**
Slow convergence, oscillation	Ill-conditioning ($\kappa \gg 1$)	Use acceleration (FISTA, Nesterov); preconditioning
Divergence	Step size too large	Reduce η; use backtracking; ensure $\eta < 1/M$
Primal infeasibility (dual methods)	f not strictly convex	Use augmented Lagrangian or ADMM instead of dual ascent
ADMM slow convergence	Poor choice of ρ	Adaptive ρ: increase if primal residual $\gg$ dual, decrease if opposite
Chambolle-Pock diverges	$\tau\sigma\|K\|^2 \geq 1$	Reduce step sizes; estimate $\|K\|$ via power iteration
Mirror descent stuck	Started with $x_i = 0$	Initialize with $x^{(0)} = \mathbf{1}/n$ (uniform) for simplex
Proximal GD slow on ℓ_1	Large λ kills all signal	Reduce λ; use warm-starting from larger λ

2. **(Projection complexity)**

   (a) Prove that projecting onto an affine subspace $\{x : Ax = b\}$ requires $O(mn^2)$ for the first projection and $O(mn)$ for subsequent ones (with precomputation).

   (b) Prove that projecting onto the simplex requires $O(n \log n)$ due to sorting.

   (c) Can you do better than $O(n \log n)$ for simplex projection? Research median-finding approaches.

3. **(Non-expansiveness)** Prove that the projection onto a closed convex set is non-expansive:

$$\|\Pi_{\mathcal{X}}(y_1) - \Pi_{\mathcal{X}}(y_2)\| \leq \|y_1 - y_2\|.$$

   (Hint: Use the characterization of projection via normal cones.)

Proximal Operators

4. **(Proximal calculus)**

   (a) Prove that $\text{prox}_{\eta g}(y) = y - \eta \,\text{prox}_{g^*/\eta}(y/\eta)$ (Moreau decomposition).

   (b) If $g(x) = g_1(x_1) + g_2(x_2)$ is separable, show that $\text{prox}_g(y) = (\text{prox}_{g_1}(y_1), \text{prox}_{g_2}(y_2))$.

   (c) Compute $\text{prox}_{\eta g}$ for $g(x) = \|x\|_\infty$.

5. **(FISTA)** Implement the Fast Iterative Shrinkage-Thresholding Algorithm:

$$y^{(k)} = x^{(k)} + \frac{t_{k-1} - 1}{t_k}(x^{(k)} - x^{(k-1)})$$
$$x^{(k+1)} = \text{prox}_{\eta g}(y^{(k)} - \eta \nabla f(y^{(k)}))$$
$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2}$$

with $t_0 = 1$. Compare with ISTA on LASSO and verify the $O(1/k^2)$ rate.

6. **(Elastic net)** The elastic net combines ℓ_1 and ℓ_2 penalties:

$$\min_x \frac{1}{2}\|Ax - b\|^2 + \lambda_1 \|x\|_1 + \frac{\lambda_2}{2}\|x\|_2^2.$$

   (a) Derive the proximal operator for $g(x) = \lambda_1 \|x\|_1 + \frac{\lambda_2}{2}\|x\|_2^2$.

   (b) Implement proximal gradient for elastic net.

   (c) Compare sparsity patterns of LASSO and elastic net solutions.

Mirror Descent

7. **(Bregman divergence properties)**

   (a) Prove that $D_\phi(x, y) \geq 0$ with equality iff $x = y$.

   (b) Is $D_\phi(x, y) = D_\phi(y, x)$ in general? Give a counterexample.

   (c) Prove the three-point identity: $D_\phi(x, z) = D_\phi(x, y) + D_\phi(y, z) + (\nabla\phi(y) - \nabla\phi(z))^T(x - y)$.

8. **(Mirror descent for ℓ_1 ball)** Derive the mirror descent update for $\mathcal{X} = \{x : \|x\|_1 \leq 1, x \geq 0\}$ using:

   (a) Euclidean mirror map $\phi(x) = \frac{1}{2}\|x\|_2^2$

   (b) Entropy mirror map $\phi(x) = \sum_i x_i \log x_i$

   Compare the per-iteration cost and convergence.

9. **(Online learning)** Mirror descent is fundamental for online learning. Consider the online linear optimization setting:

   (a) At round t, player chooses $x_t \in \Delta_n$, adversary reveals $\ell_t \in \mathbb{R}^n$, player incurs loss $\ell_t^T x_t$.

   (b) Show that exponentiated gradient achieves regret $O(\sqrt{T \log n})$.

   (c) Compare with projected gradient descent regret $O(\sqrt{T})$.

ADMM and Dual Methods

10. **(ADMM for basis pursuit)** Basis pursuit finds the sparsest solution to an underdetermined system:
$$\min_x \|x\|_1 \quad \text{s.t.} \quad Ax = b.$$
    (a) Reformulate as ADMM: $\min_{x,z} \|z\|_1$ s.t. $x = z$, $Ax = b$.
    (b) Derive the ADMM updates.
    (c) Implement and test on a compressed sensing problem.

11. **(Choosing ρ in ADMM)** The penalty parameter ρ affects convergence speed.
    (a) Run ADMM for LASSO with $\rho \in \{0.1, 1, 10, 100\}$ and compare convergence.
    (b) Implement adaptive ρ selection: increase ρ if primal residual $\gg$ dual residual, decrease if opposite.
    (c) Compare fixed vs adaptive ρ.

12. **(Decentralized ADMM)** Extend consensus ADMM to a decentralized setting where agents can only communicate with neighbors in a graph.
    (a) Formulate the consensus problem with edge-based constraints.
    (b) Derive decentralized ADMM updates.
    (c) Implement for a ring topology and compare with centralized consensus.

Primal-Dual Methods

13. **(Chambolle-Pock for LASSO)** Derive and implement Chambolle-Pock for LASSO by writing it as:
$$\min_x \max_y \left\{ \frac{1}{2}\|Ax - b\|^2 + y^T x - \delta_{\|y\|_\infty \le \lambda}(y) \right\}.$$
    Compare with ISTA and ADMM.

14. **(2D Total Variation)** Extend TV denoising to 2D images:
$$\min_X \frac{1}{2}\|X - B\|_F^2 + \lambda(\|D_h X\|_1 + \|D_v X\|_1),$$
    where D_h, D_v are horizontal and vertical difference operators.
    (a) Derive the Chambolle-Pock updates.
    (b) Implement for a noisy image.
    (c) Compare anisotropic TV ($\|D_h X\|_1 + \|D_v X\|_1$) with isotropic TV ($\sum_{i,j} \sqrt{(D_h X)_{ij}^2 + (D_v X)_{ij}^2}$).

15. **(Primal-dual gap)** For the saddle-point problem $\min_x \max_y \mathcal{L}(x, y)$, the primal-dual gap at (x, y) is:

$$\text{Gap}(x, y) = \max_{y'} \mathcal{L}(x, y') - \min_{x'} \mathcal{L}(x', y).$$

    (a) Show that $\text{Gap}(x, y) \geq 0$ and equals zero iff (x, y) is a saddle point.

    (b) For constrained least squares, derive a computable expression for the gap.

    (c) Use the gap as a convergence criterion in Chambolle-Pock.

Theory

16. **(Proximal gradient convergence)** Prove Theorem 5.8: for M-smooth f and convex g, proximal gradient with $\eta = 1/M$ achieves $f(x^{(k)}) + g(x^{(k)}) - f^* - g^* \leq \frac{M\|x^{(0)} - x^*\|^2}{2k}$.

17. **(ADMM convergence rate)** The $O(1/k)$ rate for ADMM is known. Research and explain:

    (a) Under what conditions does ADMM achieve linear convergence?

    (b) What is the role of strong convexity?

    (c) Can ADMM be accelerated?

18. **(Operator splitting)** Chambolle-Pock can be viewed as operator splitting. Research:

    (a) Forward-backward splitting

    (b) Douglas-Rachford splitting

    (c) How does Chambolle-Pock relate to these?

Chapter 6

Newton Method

This chapter develops **second-order methods**—algorithms that exploit curvature information through the Hessian matrix $\nabla^2 f(x)$. While first-order methods use only gradient information and achieve at best linear convergence, Newton's method achieves **quadratic convergence**: near the optimum, the number of correct digits roughly doubles each iteration. This dramatic speedup makes Newton's method the algorithm of choice when the problem size permits computing and inverting the Hessian.

We begin with unconstrained Newton's method (Section 6.1), deriving the algorithm from local quadratic approximation and analyzing its two-phase convergence. Section 6.2 introduces quasi-Newton methods that approximate the Hessian, enabling second-order-like convergence for larger problems. Section 6.3 extends Newton's method to equality-constrained problems. Finally, Section 6.4 develops interior point methods for inequality constraints, connecting barrier functions to the KKT conditions through a elegant relaxation of complementary slackness.

6.1 Unconstrained Newton's Method

Consider the unconstrained problem:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}}\, f(x), \tag{6.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable and convex.

6.1.1 Derivation from Quadratic Approximation

Gradient descent uses a first-order (linear) approximation of f at the current point. Newton's method uses a second-order (quadratic) approximation, which captures the local curvature.

The Taylor expansion of f around x is:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^T \Delta x + \frac{1}{2}(\Delta x)^T \nabla^2 f(x)(\Delta x). \tag{6.2}$$

The right-hand side is a quadratic function of Δx. For convex f, the Hessian $\nabla^2 f(x) \succeq 0$, so this quadratic

is convex in Δx. Minimizing over Δx by setting the gradient to zero:

$$\nabla f(x) + \nabla^2 f(x)\Delta x = 0.$$

If $\nabla^2 f(x) \succ 0$ (positive definite), this has a unique solution:

$$\Delta x_{\text{nt}} = -(\nabla^2 f(x))^{-1}\nabla f(x). \tag{6.3}$$

This is the **Newton step**. The **pure Newton's method** iterates:

$$x^{(k+1)} = x^{(k)} + \Delta x_{\text{nt}}^{(k)} = x^{(k)} - (\nabla^2 f(x^{(k)}))^{-1}\nabla f(x^{(k)}). \tag{6.4}$$

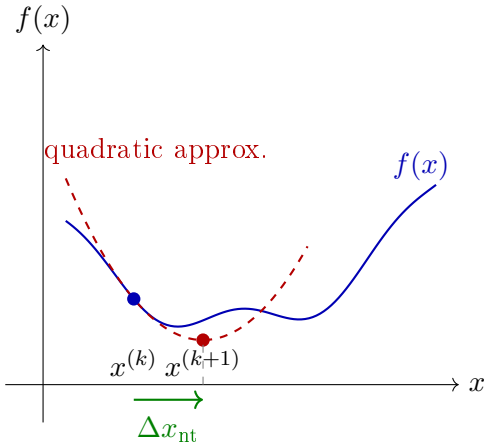

Figure 6.1: Newton's method minimizes a local quadratic approximation (dashed red) of the objective function (solid blue). The Newton step Δx_{nt} moves to the minimizer of this quadratic model.

Comparison with gradient descent. Recall that gradient descent with step size η updates:

$$x^{(k+1)} = x^{(k)} - \eta\nabla f(x^{(k)}).$$

Newton's method can be viewed as gradient descent with a *variable, matrix-valued step size* $(\nabla^2 f(x))^{-1}$. This matrix rescales the gradient to account for curvature: in directions where f curves sharply (large eigenvalues of $\nabla^2 f$), Newton takes smaller steps; in flat directions, it takes larger steps.

While Newton's method superficially resembles gradient descent with the Hessian inverse replacing the scalar step size, this analogy is misleading. The matrix $(\nabla^2 f(x))^{-1}$ does not merely scale the gradient—it fundamentally changes its direction.

The gradient $-\nabla f(x)$ points in the direction of steepest descent, which is generally not toward the optimum. The Newton direction $-(\nabla^2 f(x))^{-1}\nabla f(x)$ instead points toward the minimizer of the local quadratic model. To see the difference, consider $f(x) = x_1^2 + 100x_2^2$ at $x = (1, 1)^T$: the gradient points mostly along x_2 (since $\partial f/\partial x_2 = 200$), while Newton points directly at the origin.

In the eigenbasis of the Hessian, Newton independently rescales each component of the gradient by the inverse eigenvalue—shrinking steps in high-curvature directions and stretching them in flat directions. This coordinate-wise correction both reorients and rescales the step, eliminating the condition number dependence that plagues gradient descent.

Example 6.1 (Newton on a quadratic)**.** Consider $f(x) = \frac{1}{2}x^TQx + b^Tx + c$ with $Q \succ 0$. Then $\nabla f(x) = Qx + b$ and $\nabla^2 f(x) = Q$. The Newton step is:

$$\Delta x_{\text{nt}} = -Q^{-1}(Qx + b) = -x - Q^{-1}b.$$

One iteration gives:

$$x^{(1)} = x^{(0)} + \Delta x_{\text{nt}} = x^{(0)} - x^{(0)} - Q^{-1}b = -Q^{-1}b = x^*.$$

Newton's method finds the exact minimizer of a quadratic in **one step**, regardless of the condition number! Compare this to gradient descent, which requires $O(\kappa \log(1/\epsilon))$ iterations where κ is the condition number of Q.

6.1.2 The Newton Decrement

The **Newton decrement** provides a measure of how far we are from optimality.

Definition 6.2 (Newton decrement)**.** The **Newton decrement** at x is:

$$\lambda(x) = \sqrt{\nabla f(x)^T(\nabla^2 f(x))^{-1}\nabla f(x)} = \|\Delta x_{\text{nt}}\|_{\nabla^2 f(x)}, \tag{6.5}$$

where $\|v\|_H = \sqrt{v^THv}$ is the norm induced by positive definite matrix H.

Proposition 6.3 (Properties of Newton decrement)**.** *The Newton decrement satisfies:*

1. $\lambda(x)^2/2$ *equals the decrease in* f *predicted by the quadratic model:*

$$f(x) - \min_{\Delta x}\left\{f(x) + \nabla f(x)^T\Delta x + \frac{1}{2}(\Delta x)^T\nabla^2 f(x)(\Delta x)\right\} = \frac{\lambda(x)^2}{2}.$$

2. $\lambda(x) = 0$ *if and only if* $x = x^*$ *(the optimal point).*

3. $\lambda(x)$ *is affine invariant: if* $\tilde{x} = Ax$ *for invertible* A*, then* $\lambda_{\tilde{f}}(\tilde{x}) = \lambda_f(x)$.

The Newton decrement provides a natural stopping criterion: terminate when $\lambda(x)^2/2 \leq \epsilon$.

6.1.3 Damped Newton's Method

The pure Newton's method (6.4) may not decrease f when far from the optimum—the quadratic approximation may be poor. The **damped Newton's method** adds a step size $\eta^{(k)} \in (0, 1]$:

$$x^{(k+1)} = x^{(k)} + \eta^{(k)}\Delta x_{\text{nt}}^{(k)}. \tag{6.6}$$

We select $\eta^{(k)}$ via backtracking line search to ensure sufficient decrease.

Algorithm 16 Newton's Method with Backtracking

Input: Initial $x^{(0)}$, tolerance $\epsilon > 0$, parameters $\alpha \in (0, 0.5)$, $\beta \in (0, 1)$
for $k = 0, 1, 2, \ldots$ **do**
  Compute gradient $g \leftarrow \nabla f(x^{(k)})$ and Hessian $H \leftarrow \nabla^2 f(x^{(k)})$
  Solve $H\Delta x = -g$ for Newton step Δx_{nt} ▷ $O(n^3)$ via Cholesky
  Compute Newton decrement $\lambda \leftarrow \sqrt{g^T(-\Delta x_{\text{nt}})}$
  if $\lambda^2/2 \leq \epsilon$ **then** ▷ Stopping criterion
    return $x^{(k)}$
  end if
  Backtracking line search:
  $\eta \leftarrow 1$
  while $f(x^{(k)} + \eta\Delta x_{\text{nt}}) > f(x^{(k)}) - \alpha\eta\lambda^2$ **do**
    $\eta \leftarrow \beta\eta$
  end while
  $x^{(k+1)} \leftarrow x^{(k)} + \eta\Delta x_{\text{nt}}$
end for

Remark 6.4 (Computational cost). Each Newton iteration requires:

- Computing the gradient: $O(n)$ to $O(n^2)$ depending on structure
- Computing the Hessian: $O(n^2)$ storage, $O(n^2)$ to $O(n^3)$ computation
- Solving the linear system $H\Delta x = -g$: $O(n^3)$ for dense systems via Cholesky factorization

The $O(n^3)$ cost per iteration is the main limitation of Newton's method. For n in the thousands, this is manageable; for n in the millions, first-order methods are necessary.

6.1.4 Convergence Analysis

Newton's method exhibits two distinct phases of convergence.

Theorem 6.5 (Convergence of Newton's method)**.** *Suppose f is twice continuously differentiable, m-strongly convex, M-smooth, with L-Lipschitz Hessian:*

$$\|\nabla^2 f(x) - \nabla^2 f(y)\|_2 \leq L\|x - y\| \quad \textit{for all } x, y.$$

Let $\eta_0 = \min\{1, 3(1-2\alpha)/(L/m^2)\}$ where $\alpha \in (0, 1/2)$ is the backtracking parameter. Define:

$$\gamma = \alpha\beta\eta_0 m \quad \textit{(minimum decrease per iteration in Phase 1).}$$

Newton's method with backtracking satisfies:

1. ***Phase 1 (Damped phase):*** *While* $\lambda(x^{(k)}) > \eta_0 m^2/L$*:*

$$f(x^{(k+1)}) - f(x^{(k)}) \leq -\gamma.$$

This phase lasts at most $(f(x^{(0)}) - f^*)/\gamma$ *iterations.*

2. ***Phase 2 (Quadratic convergence):*** *Once* $\lambda(x^{(k)}) \leq \eta_0 m^2/L$*, backtracking always accepts* $\eta = 1$*, and:*

$$\lambda(x^{(k+1)}) \leq \frac{L}{2m^2}\lambda(x^{(k)})^2.$$

The quadratic convergence in Phase 2 is remarkable. If $\lambda(x^{(k)}) = 10^{-2}$, then $\lambda(x^{(k+1)}) \approx 10^{-4}$, $\lambda(x^{(k+2)}) \approx 10^{-8}$, and $\lambda(x^{(k+3)}) \approx 10^{-16}$. The number of correct digits roughly doubles each iteration!

Proof of quadratic convergence (Phase 2). We prove the key inequality $\|x^{(k+1)} - x^*\| \leq \frac{L}{2m}\|x^{(k)} - x^*\|^2$.

Step 1: Integral representation of gradient. By the fundamental theorem of calculus:

$$\nabla f(x) - \nabla f(x^*) = \int_0^1 \nabla^2 f(x^* + t(x - x^*))(x - x^*)\, dt.$$

Since $\nabla f(x^*) = 0$:

$$\nabla f(x) = \int_0^1 \nabla^2 f(x^* + t(x - x^*))(x - x^*)\, dt. \tag{6.7}$$

Step 2: One Newton iteration. For pure Newton ($\eta = 1$):

$$\begin{aligned} x^{(k+1)} - x^* &= x^{(k)} - (\nabla^2 f(x^{(k)}))^{-1}\nabla f(x^{(k)}) - x^* \\ &= (\nabla^2 f(x^{(k)}))^{-1}\left[\nabla^2 f(x^{(k)})(x^{(k)} - x^*) - \nabla f(x^{(k)})\right]. \end{aligned}$$

Step 3: Substitute integral representation. Using (6.7) with $x = x^{(k)}$:

$$x^{(k+1)} - x^* = (\nabla^2 f(x^{(k)}))^{-1}\int_0^1 \left[\nabla^2 f(x^{(k)}) - \nabla^2 f(x^* + t(x^{(k)} - x^*))\right](x^{(k)} - x^*)\, dt.$$

Step 4: Bound the norm. Taking norms and using submultiplicativity:

$$\begin{aligned} \|x^{(k+1)} - x^*\| &\leq \|(\nabla^2 f(x^{(k)}))^{-1}\| \int_0^1 \|\nabla^2 f(x^{(k)}) - \nabla^2 f(x^* + t(x^{(k)} - x^*))\|\|x^{(k)} - x^*\|\, dt \\ &\leq \frac{1}{m}\int_0^1 L(1-t)\|x^{(k)} - x^*\|\, dt \cdot \|x^{(k)} - x^*\| \\ &= \frac{L}{m}\cdot\frac{1}{2}\|x^{(k)} - x^*\|^2 = \frac{L}{2m}\|x^{(k)} - x^*\|^2. \end{aligned}$$

Here we used: $\|(\nabla^2 f)^{-1}\| \leq 1/m$ (from strong convexity), and the Lipschitz Hessian condition with the observation that $x^{(k)} - (x^* + t(x^{(k)} - x^*)) = (1-t)(x^{(k)} - x^*)$. □

Corollary 6.6 (Total iteration count). *Newton's method achieves $f(x) - f^* \leq \epsilon$ in at most:*

$$\frac{f(x^{(0)}) - f^*}{\gamma} + O(\log\log(1/\epsilon))$$

iterations, where the first term bounds Phase 1 and the second bounds Phase 2.

The $\log\log(1/\epsilon)$ term is essentially constant for practical purposes: achieving $\epsilon = 10^{-100}$ requires only about 7 Phase 2 iterations.

6.1.5 Affine Invariance

A remarkable property of Newton's method is its invariance under affine transformations.

Proposition 6.7 (Affine invariance). *Let $T : \mathbb{R}^n \to \mathbb{R}^n$ be an invertible affine transformation $T(x) = Ax + b$. Define $\tilde{f}(\tilde{x}) = f(T^{-1}(\tilde{x})) = f(A^{-1}(\tilde{x} - b))$. Then Newton's method on $\tilde{f}$ starting from $\tilde{x}^{(0)} = T(x^{(0)})$ produces iterates $\tilde{x}^{(k)} = T(x^{(k)})$, where $x^{(k)}$ are the Newton iterates on f.*

Proof. Let $\tilde{x} = Ax + b$. Then:

$$\begin{aligned}\nabla\tilde{f}(\tilde{x}) &= A^{-T}\nabla f(x),\\ \nabla^2\tilde{f}(\tilde{x}) &= A^{-T}\nabla^2 f(x)A^{-1}.\end{aligned}$$

The Newton step for $\tilde{f}$ is:

$$\begin{aligned}\Delta\tilde{x}_{\text{nt}} &= -(\nabla^2\tilde{f}(\tilde{x}))^{-1}\nabla\tilde{f}(\tilde{x})\\ &= -A(\nabla^2 f(x))^{-1}A^T \cdot A^{-T}\nabla f(x)\\ &= -A(\nabla^2 f(x))^{-1}\nabla f(x) = A\Delta x_{\text{nt}}.\end{aligned}$$

Thus $\tilde{x}^{(k+1)} = \tilde{x}^{(k)} + \Delta\tilde{x}_{\text{nt}} = Ax^{(k)} + b + A\Delta x_{\text{nt}} = A(x^{(k)} + \Delta x_{\text{nt}}) + b = T(x^{(k+1)})$. □

Practical implication. Affine invariance means Newton's method is immune to poor scaling or conditioning of the problem. Gradient descent, in contrast, is highly sensitive: an ill-conditioned Hessian (large $\kappa = M/m$) leads to slow convergence. Newton's method automatically adapts to the local geometry through the Hessian inverse.

6.1.6 Implementation

```
import numpy as np
from scipy.linalg import cho_factor, cho_solve

def newton_method(f, grad_f, hess_f, x0, tol=1e-10, max_iter=100,
                  alpha=0.25, beta=0.5, verbose=True):
```

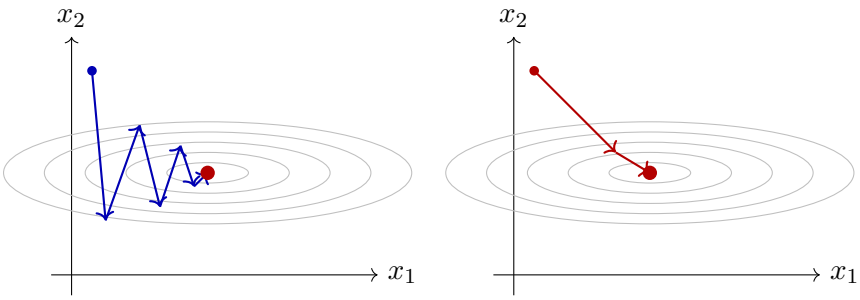

Figure 6.2: Comparison of gradient descent (left) and Newton's method (right) on an ill-conditioned quadratic. Gradient descent zigzags due to the elongated level sets, while Newton's method accounts for curvature and converges rapidly. The Hessian inverse "circularizes" the level sets from Newton's perspective.

```
    """
    Newton's method with backtracking line search.

    Parameters:
        f: objective function
        grad_f: gradient function
        hess_f: Hessian function (returns n x n matrix)
        x0: initial point
        tol: tolerance for Newton decrement squared / 2
        alpha: Armijo parameter (0 < alpha < 0.5)
        beta: backtracking shrinkage factor (0 < beta < 1)
        verbose: print progress

    Returns:
        x: optimal point
        history: dict with convergence information
    """
    x = np.array(x0, dtype=float)
    n = len(x)

    history = {
        'f': [f(x)],
        'newton_decrement': [],
        'step_size': [],
        'x': [x.copy()]
    }

    for k in range(max_iter):
        # Compute gradient and Hessian
```

```
        g = grad_f(x)
        H = hess_f(x)

        # Solve H @ dx = -g for Newton step using Cholesky factorization
        try:
            c, lower = cho_factor(H)
            dx = cho_solve((c, lower), -g)
        except np.linalg.LinAlgError:
            print("Hessian not positive definite, adding regularization")
            H_reg = H + 1e-6 * np.eye(n)
            c, lower = cho_factor(H_reg)
            dx = cho_solve((c, lower), -g)

        # Compute Newton decrement
        lambda_sq = -g @ dx  # = g^T H^{-1} g
        newton_decrement = np.sqrt(lambda_sq)
        history['newton_decrement'].append(newton_decrement)

        # Check stopping criterion
        if lambda_sq / 2 <= tol:
            if verbose:
                print(f"Converged at iteration {k}: lambda^2/2 = {lambda_sq/2:.2e}")
            break

        # Backtracking line search
        eta = 1.0
        f_x = f(x)
        while f(x + eta * dx) > f_x - alpha * eta * lambda_sq:
            eta *= beta
            if eta < 1e-16:
                print("Warning: step size too small")
                break

        history['step_size'].append(eta)

        # Update
        x = x + eta * dx
        history['f'].append(f(x))
        history['x'].append(x.copy())

        if verbose:
            print(f"Iter {k:3d}: f = {f(x):.6e}, lambda = {newton_decrement:.2e}, eta = {eta:.4
    f}")
```

```
    return x, history


def demo_newton():
    """Demonstrate Newton's method on an ill-conditioned quadratic."""
    np.random.seed(42)
    n = 20

    # Create ill-conditioned quadratic: f(x) = 0.5 * x^T Q x + b^T x
    kappa = 1000  # Condition number
    eigenvalues = np.linspace(1, kappa, n)
    Q_diag = np.diag(eigenvalues)

    # Random rotation
    U, _ = np.linalg.qr(np.random.randn(n, n))
    Q = U @ Q_diag @ U.T
    b = np.random.randn(n)

    # Optimal solution
    x_star = np.linalg.solve(Q, -b)
    f_star = 0.5 * x_star @ Q @ x_star + b @ x_star

    f = lambda x: 0.5 * x @ Q @ x + b @ x
    grad_f = lambda x: Q @ x + b
    hess_f = lambda x: Q

    # Initial point
    x0 = np.random.randn(n) * 10

    print("="*60)
    print("Newton's Method on Ill-Conditioned Quadratic")
    print(f"Dimension: {n}, Condition number: {kappa}")
    print("="*60)

    x_newton, history = newton_method(f, grad_f, hess_f, x0, verbose=True)

    print(f"\nFinal error: ||x - x*|| = {np.linalg.norm(x_newton - x_star):.2e}")
    print(f"Final objective gap: f(x) - f* = {f(x_newton) - f_star:.2e}")

    # Compare with gradient descent
    print("\n" + "="*60)
    print("Gradient Descent for comparison")
    print("="*60)
```

```
    x_gd = x0.copy()
    eta_gd = 1.0 / kappa  # Step size = 1/L
    gd_errors = [np.linalg.norm(x_gd - x_star)]

    for k in range(1000):
        x_gd = x_gd - eta_gd * grad_f(x_gd)
        gd_errors.append(np.linalg.norm(x_gd - x_star))
        if gd_errors[-1] < 1e-10:
            break

    print(f"GD converged in {len(gd_errors)-1} iterations")
    print(f"Newton converged in {len(history['f'])-1} iterations")

    return history, gd_errors

# Run demonstration
history, gd_errors = demo_newton()
```

Check Your Understanding 6.1.

Consider $f(x) = e^x + e^{-x}$ on $\mathbb{R}$.

(a) Compute $\nabla f(x)$, $\nabla^2 f(x)$, and the Newton step Δx_{nt} at $x = 1$.

(b) What is the Newton decrement at $x = 1$?

(c) Verify that $x^* = 0$ and compute the Newton step at $x = 0$.

(d) Starting from $x^{(0)} = 1$, compute $x^{(1)}$ using pure Newton.

6.2 Quasi-Newton Methods

Newton's method requires computing and factorizing the $n \times n$ Hessian at each iteration—$O(n^2)$ storage and $O(n^3)$ computation. For large-scale problems, this is prohibitive. **Quasi-Newton methods** approximate the Hessian (or its inverse) using only gradient information, achieving superlinear convergence without explicit Hessian computation.

6.2.1 The Quasi-Newton Framework

Instead of computing $H_k = \nabla^2 f(x^{(k)})$, quasi-Newton methods maintain an approximation $B_k \approx H_k$ (or $W_k \approx H_k^{-1}$) and update it each iteration using gradient differences.

Secant condition. At consecutive iterates $x^{(k)}$ and $x^{(k+1)}$, define:

$$s_k = x^{(k+1)} - x^{(k)} \quad \text{(step)} \tag{6.8}$$

$$y_k = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)}) \quad \text{(gradient difference)} \tag{6.9}$$

For a quadratic $f(x) = \frac{1}{2}x^T Hx + b^T x$, we have $y_k = Hs_k$. Quasi-Newton methods require the approximation B_{k+1} to satisfy the **secant condition**:

$$B_{k+1}s_k = y_k. \tag{6.10}$$

6.2.2 The BFGS Algorithm

The **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** method is the most successful quasi-Newton algorithm. It maintains an approximation $W_k \approx H_k^{-1}$ and updates it to satisfy the secant condition while remaining positive definite.

Algorithm 17 BFGS Method

1: **Input:** Initial $x^{(0)}$, initial $W_0 = I$ (or scaled identity)
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Compute search direction: $d_k = -W_k \nabla f(x^{(k)})$
4: Line search: find η_k satisfying Wolfe conditions
5: Update: $x^{(k+1)} = x^{(k)} + \eta_k d_k$
6: Compute $s_k = x^{(k+1)} - x^{(k)}$ and $y_k = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$
7: Compute $\rho_k = 1/(y_k^T s_k)$
8: Update inverse Hessian approximation:

$$W_{k+1} = (I - \rho_k s_k y_k^T) W_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$$

9: **end for**

The BFGS update has several important properties:

- If $W_k \succ 0$ and $s_k^T y_k > 0$ (curvature condition), then $W_{k+1} \succ 0$.
- The update requires only $O(n^2)$ operations (matrix-vector products).
- For quadratic functions, BFGS converges in at most n iterations.

6.2.3 Limited-Memory BFGS (L-BFGS)

For very large n, even storing an $n \times n$ matrix is infeasible. **L-BFGS** stores only the last m pairs (s_k, y_k) (typically $m = 5$ to 20) and computes $W_k \nabla f(x^{(k)})$ implicitly.

Algorithm 18 L-BFGS Two-Loop Recursion

Input: Current gradient $g = \nabla f(x^{(k)})$, pairs $\{(s_i, y_i)\}_{i=k-m}^{k-1}$
$q \leftarrow g$
for $i = k-1, k-2, \ldots, k-m$ **do**
    $\alpha_i \leftarrow \rho_i s_i^T q$
    $q \leftarrow q - \alpha_i y_i$
end for
$r \leftarrow H_k^0 q$ ▷ $H_k^0 = \gamma_k I$ with $\gamma_k = s_{k-1}^T y_{k-1} / (y_{k-1}^T y_{k-1})$
for $i = k-m, k-m+1, \ldots, k-1$ **do**
    $\beta \leftarrow \rho_i y_i^T r$
    $r \leftarrow r + (\alpha_i - \beta) s_i$
end for
return $r = W_k g$

L-BFGS requires only $O(mn)$ storage and $O(mn)$ computation per iteration—linear in n! This makes it the method of choice for large-scale smooth unconstrained optimization, including training neural networks before the advent of adaptive methods like Adam.

```
import numpy as np
from collections import deque

class LBFGS:
    """Limited-memory BFGS optimizer."""

    def __init__(self, m=10):
        """
        Parameters:
            m: number of correction pairs to store
        """
        self.m = m
        self.s_history = deque(maxlen=m)
        self.y_history = deque(maxlen=m)
        self.rho_history = deque(maxlen=m)

    def compute_direction(self, g):
        """
        Compute search direction using two-loop recursion.

        Parameters:
            g: current gradient

        Returns:
            d: search direction (approximation to -H^{-1} g)
```

```
        """
        q = g.copy()
        n = len(g)
        k = len(self.s_history)

        if k == 0:
            # No history yet, use steepest descent
            return -g

        alpha = np.zeros(k)

        # First loop (backward)
        for i in range(k - 1, -1, -1):
            alpha[i] = self.rho_history[i] * np.dot(self.s_history[i], q)
            q = q - alpha[i] * self.y_history[i]

        # Initial Hessian approximation: gamma * I
        gamma = (np.dot(self.s_history[-1], self.y_history[-1]) /
                 np.dot(self.y_history[-1], self.y_history[-1]))
        r = gamma * q

        # Second loop (forward)
        for i in range(k):
            beta = self.rho_history[i] * np.dot(self.y_history[i], r)
            r = r + (alpha[i] - beta) * self.s_history[i]

        return -r

    def update(self, s, y):
        """
        Update the history with new correction pair.

        Parameters:
            s: step (x_{k+1} - x_k)
            y: gradient difference (grad_{k+1} - grad_k)
        """
        rho = 1.0 / np.dot(y, s)
        if rho > 0:  # Curvature condition
            self.s_history.append(s)
            self.y_history.append(y)
            self.rho_history.append(rho)


def lbfgs_optimize(f, grad_f, x0, m=10, max_iter=1000, tol=1e-8,
```

```
                c1=1e-4, c2=0.9, verbose=True):
    """
    L-BFGS optimization with Wolfe line search.
    """
    x = np.array(x0, dtype=float)
    optimizer = LBFGS(m=m)

    history = {'f': [f(x)], 'grad_norm': []}

    g = grad_f(x)

    for k in range(max_iter):
        grad_norm = np.linalg.norm(g)
        history['grad_norm'].append(grad_norm)

        if grad_norm < tol:
            if verbose:
                print(f"Converged at iteration {k}")
            break

        # Compute search direction
        d = optimizer.compute_direction(g)

        # Wolfe line search (simplified backtracking for demo)
        eta = 1.0
        f_x = f(x)
        g_d = np.dot(g, d)

        for _ in range(50):
            x_new = x + eta * d
            if f(x_new) <= f_x + c1 * eta * g_d:
                g_new = grad_f(x_new)
                if np.dot(g_new, d) >= c2 * g_d:
                    break
            eta *= 0.5

        # Update
        s = x_new - x
        y = g_new - g

        optimizer.update(s, y)

        x = x_new
        g = g_new
```

```
        history['f'].append(f(x))

        if verbose and k % 10 == 0:
            print(f"Iter {k:4d}: f = {f(x):.6e}, ||grad|| = {grad_norm:.2e}")

    return x, history
```

Check Your Understanding 6.2.

(a) Why does BFGS update the inverse Hessian approximation rather than the Hessian approximation?

(b) What is the storage requirement of BFGS vs L-BFGS for a problem with $n = 10^6$ variables?

(c) The curvature condition $s_k^T y_k > 0$ is needed for BFGS. When might this fail?

(d) For a strongly convex quadratic, how many iterations does BFGS need?

6.3 Newton's Method for Equality-Constrained Problems

Consider the equality-constrained problem:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & Ax = b, \end{aligned} \tag{6.11}$$

where f is twice differentiable and convex, $A \in \mathbb{R}^{p \times n}$ has full row rank, and $b \in \mathbb{R}^p$.

6.3.1 KKT Conditions and Newton Step

The Lagrangian is:

$$L(x, \nu) = f(x) + \nu^T (Ax - b),$$

where $\nu \in \mathbb{R}^p$ are the Lagrange multipliers. The KKT conditions for optimality are:

$$\nabla f(x^*) + A^T \nu^* = 0, \tag{6.12}$$

$$Ax^* = b. \tag{6.13}$$

To derive Newton's method, we seek a step $(\Delta x, \nu)$ from current point x such that the linearized KKT conditions hold at $x + \Delta x$:

$$\begin{aligned} \nabla f(x + \Delta x) + A^T \nu \approx \nabla f(x) + \nabla^2 f(x) \Delta x + A^T \nu = 0, \\ A(x + \Delta x) = b. \end{aligned}$$

This gives the **KKT system**:

$$\begin{bmatrix} \nabla^2 f(x) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \nu \end{bmatrix} = \begin{bmatrix} -\nabla f(x) \\ b - Ax \end{bmatrix}. \tag{6.14}$$

The matrix on the left is called the **KKT matrix**. It is symmetric but indefinite (has both positive and negative eigenvalues).

Proposition 6.8 (Solvability of KKT system)**.** *If $\nabla^2 f(x) \succ 0$ and A has full row rank, the KKT matrix is nonsingular.*

Algorithm 19 Newton's Method for Equality-Constrained Optimization

Input: Initial $x^{(0)}$ (feasible: $Ax^{(0)} = b$), tolerance ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Compute gradient $g = \nabla f(x^{(k)})$ and Hessian $H = \nabla^2 f(x^{(k)})$
  Solve the KKT system:

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \nu \end{bmatrix} = \begin{bmatrix} -g \\ 0 \end{bmatrix}$$

  Compute Newton decrement $\lambda = \sqrt{-g^T \Delta x}$
  if $\lambda^2/2 \le \epsilon$ **then**
    return $x^{(k)}$
  end if
  Backtracking line search for η
  $x^{(k+1)} \leftarrow x^{(k)} + \eta \Delta x$
end for

Remark 6.9 (Feasibility preservation). If $x^{(k)}$ is feasible ($Ax^{(k)} = b$), then for any step size η:

$$A(x^{(k)} + \eta \Delta x) = Ax^{(k)} + \eta A \Delta x = b + 0 = b.$$

The Newton step preserves feasibility because $A\Delta x = 0$ (from the KKT system with zero right-hand side in the second block).

Remark 6.10 (Infeasible start). If $x^{(0)}$ is not feasible, the right-hand side becomes $[-g; b - Ax]^T$, and the Newton step simultaneously reduces the objective and the constraint violation. This is the **infeasible-start Newton method**.

```
import numpy as np
from scipy.linalg import lu_factor, lu_solve

def newton_equality_constrained(f, grad_f, hess_f, A, b, x0,
                                tol=1e-10, max_iter=100, alpha=0.25, beta=0.5):
```

```
    """
    Newton's method for equality-constrained optimization.

    min f(x)  s.t.  Ax = b
    """
    x = np.array(x0, dtype=float)
    n = len(x)
    p = len(b)

    # Check initial feasibility
    residual = A @ x - b
    if np.linalg.norm(residual) > 1e-10:
        print("Warning: initial point not feasible, using infeasible-start Newton")

    history = {'f': [f(x)], 'newton_decrement': [], 'constraint_violation': [np.linalg.norm(
    residual)]}

    for k in range(max_iter):
        g = grad_f(x)
        H = hess_f(x)
        residual = A @ x - b

        # Form and solve KKT system
        # [H   A^T] [dx]   [-g        ]
        # [A   0  ] [nu] = [b - Ax    ]
        KKT = np.block([
            [H, A.T],
            [A, np.zeros((p, p))]
        ])
        rhs = np.concatenate([-g, -residual])

        try:
            lu, piv = lu_factor(KKT)
            sol = lu_solve((lu, piv), rhs)
        except np.linalg.LinAlgError:
            print("KKT system singular")
            break

        dx = sol[:n]
        nu = sol[n:]

        # Newton decrement
        lambda_sq = -g @ dx
        if lambda_sq < 0:
```

```
            lambda_sq = abs(lambda_sq)  # Numerical issue
        newton_decrement = np.sqrt(lambda_sq)
        history['newton_decrement'].append(newton_decrement)

        # Check convergence
        if lambda_sq / 2 <= tol and np.linalg.norm(residual) < tol:
            print(f"Converged at iteration {k}")
            break

        # Backtracking line search
        eta = 1.0
        f_x = f(x)

        # Merit function: f(x) + penalty * ||Ax - b||
        penalty = 1.0
        merit = lambda z: f(z) + penalty * np.linalg.norm(A @ z - b)
        merit_x = merit(x)

        while merit(x + eta * dx) > merit_x - alpha * eta * lambda_sq:
            eta *= beta
            if eta < 1e-16:
                break

        x = x + eta * dx
        history['f'].append(f(x))
        history['constraint_violation'].append(np.linalg.norm(A @ x - b))

        print(f"Iter {k}: f = {f(x):.6e}, lambda = {newton_decrement:.2e}, "
              f"||Ax-b|| = {np.linalg.norm(A @ x - b):.2e}, eta = {eta:.4f}")

    return x, nu, history


def demo_equality_constrained():
    """Demo: minimize quadratic subject to linear equality constraints."""
    np.random.seed(42)
    n, p = 10, 3

    # Objective: f(x) = 0.5 * x^T Q x + c^T x
    Q = np.eye(n) + 0.5 * np.random.randn(n, n)
    Q = Q @ Q.T  # Make positive definite
    c = np.random.randn(n)

    f = lambda x: 0.5 * x @ Q @ x + c @ x
```

```
    grad_f = lambda x: Q @ x + c
    hess_f = lambda x: Q

    # Constraints: Ax = b
    A = np.random.randn(p, n)
    x_feas = np.random.randn(n)
    b = A @ x_feas  # Ensure feasibility is possible

    # Initial point (may not be feasible)
    x0 = np.zeros(n)

    print("="*60)
    print("Newton for Equality-Constrained Optimization")
    print("="*60)

    x_opt, nu_opt, history = newton_equality_constrained(
        f, grad_f, hess_f, A, b, x0
    )

    print(f"\nOptimal x (first 5): {x_opt[:5]}")
    print(f"Optimal multipliers: {nu_opt}")
    print(f"Final constraint violation: {np.linalg.norm(A @ x_opt - b):.2e}")

    # Verify KKT conditions
    kkt_residual = grad_f(x_opt) + A.T @ nu_opt
    print(f"KKT residual ||grad f + A^T nu||: {np.linalg.norm(kkt_residual):.2e}")

demo_equality_constrained()
```

6.4 Interior Point Methods

We now address the general convex program with inequality constraints:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & Ax = b, \end{aligned} \tag{6.15}$$

where f and $g_1, \ldots, g_m$ are convex and twice differentiable, and $A \in \mathbb{R}^{p \times n}$.

6.4.1 The Barrier Method

The key idea is to replace the inequality constraints with a **barrier function** that goes to infinity at the constraint boundary.

Definition 6.11 (Logarithmic barrier)**.** The **logarithmic barrier** for the constraints $g_i(x) \leq 0$ is:

$$\phi(x) = -\sum_{i=1}^{m} \log(-g_i(x)). \tag{6.16}$$

The domain of ϕ is the strictly feasible set $\{x : g_i(x) < 0,\ i = 1, \ldots, m\}$.

The barrier function $\phi(x) \to +\infty$ as any constraint approaches its boundary ($g_i(x) \to 0^-$), effectively preventing the iterates from leaving the feasible region.

For a parameter $t > 0$, we solve the **barrier problem**:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & tf(x) + \phi(x) \\ \text{subject to} \quad & Ax = b. \end{aligned} \tag{6.17}$$

As $t \to \infty$, the barrier term becomes negligible and the solution approaches the solution of the original problem.

Definition 6.12 (Central path)**.** The **central path** is the set of solutions $\{x^*(t) : t > 0\}$ where $x^*(t)$ solves the barrier problem for parameter t. As $t \to \infty$, $x^*(t) \to x^*$ (the solution of the original problem).

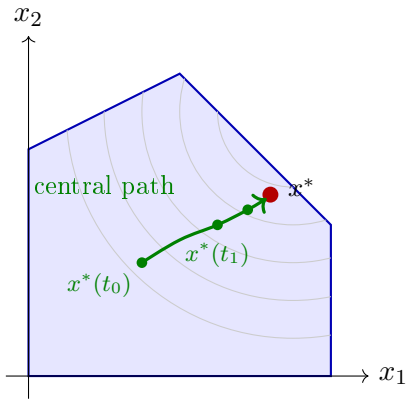

Figure 6.3: The central path traces solutions of barrier problems for increasing t. As $t \to \infty$, the path converges to the optimal point x^* on the boundary of the feasible region.

6.4.2 Connection to KKT via Relaxed Complementary Slackness

The barrier method has an elegant interpretation through the KKT conditions. The Lagrangian for (6.15) is:

$$L(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \nu^T (Ax - b).$$

The KKT conditions are:

$$\nabla f(x) + \sum_{i=1}^{m} \lambda_i \nabla g_i(x) + A^T \nu = 0, \tag{6.18}$$

$$\lambda_i g_i(x) = 0, \quad i = 1, \ldots, m, \tag{6.19}$$

$$g_i(x) \leq 0, \quad i = 1, \ldots, m, \tag{6.20}$$

$$\lambda_i \geq 0, \quad i = 1, \ldots, m, \tag{6.21}$$

$$Ax = b. \tag{6.22}$$

The complementary slackness condition (6.19) is problematic: it's a system of nonlinear equations. The key insight is to **relax** complementary slackness:

Definition 6.13 (Modified KKT conditions)**.** For $t > 0$, the **modified KKT conditions** replace (6.19) with:

$$\lambda_i g_i(x) = -1/t, \quad i = 1, \ldots, m. \tag{6.23}$$

Since we require $g_i(x) < 0$ (strict feasibility) and $\lambda_i > 0$, we can solve for:

$$\lambda_i = -\frac{1}{t g_i(x)}, \quad i = 1, \ldots, m. \tag{6.24}$$

Substituting into (6.18):

$$\nabla f(x) - \sum_{i=1}^{m} \frac{1}{t g_i(x)} \nabla g_i(x) + A^T \nu = 0. \tag{6.25}$$

Proposition 6.14 (Barrier gradient interpretation)**.** *The condition* (6.25) *is exactly the stationarity condition for:*

$$\underset{x}{\text{minimize}} \; t f(x) + \phi(x) \quad s.t. \quad Ax = b,$$

where $\phi(x) = -\sum_{i=1}^{m} \log(-g_i(x))$ *is the log barrier.*

Proof. Compute the gradient of the barrier:

$$\nabla \phi(x) = -\sum_{i=1}^{m} \frac{\nabla g_i(x)}{g_i(x)} = \sum_{i=1}^{m} \frac{-\nabla g_i(x)}{g_i(x)}.$$

The stationarity condition for the barrier problem is:

$$t \nabla f(x) + \nabla \phi(x) + A^T \nu = 0,$$

which gives:

$$t \nabla f(x) - \sum_{i=1}^{m} \frac{\nabla g_i(x)}{g_i(x)} + A^T \nu = 0.$$

Dividing by t yields (6.25) with $\tilde{\nu} = \nu / t$. □

This connection reveals that solving the barrier problem is equivalent to solving the KKT conditions with relaxed complementary slackness. As $t \to \infty$, the relaxation $\lambda_i g_i(x) = -1/t \to 0$ approaches exact complementary slackness.

6.4.3 The Barrier Method Algorithm

Algorithm 20 Barrier Method

1: **Input:** Strictly feasible $x^{(0)}$, $t^{(0)} > 0$, $\mu > 1$, tolerance $\epsilon > 0$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: **Centering step:** Solve (via Newton) starting from $x^{(k)}$:

$$x^{(k+1)} = \arg\min_{Ax=b} \left\{ t^{(k)} f(x) + \phi(x) \right\}$$

4: **Duality gap check:** If $m/t^{(k)} < \epsilon$, **return** $x^{(k+1)}$
5: **Increase** t**:** $t^{(k+1)} = \mu \cdot t^{(k)}$
6: **end for**

Theorem 6.15 (Convergence of barrier method)**.** *Let $x^*(t)$ denote the solution of the barrier problem with parameter t. Then:*

$$f(x^*(t)) - f^* \leq \frac{m}{t}, \tag{6.26}$$

where m is the number of inequality constraints and f^* is the optimal value.

Proof sketch. The dual variables from (6.24) satisfy $\lambda_i^*(t) = -1/(t g_i(x^*(t)))$. The duality gap is:

$$f(x^*(t)) - f^* \leq f(x^*(t)) - g(\lambda^*(t), \nu^*(t)) = \sum_{i=1}^{m} \lambda_i^*(t)(-g_i(x^*(t))) = \sum_{i=1}^{m} \frac{1}{t} = \frac{m}{t}.$$

□

Corollary 6.16 (Iteration complexity)**.** *To achieve $f(x) - f^* \leq \epsilon$, the barrier method requires:*

$$\left\lceil \frac{\log(m/(\epsilon t^{(0)}))}{\log \mu} \right\rceil$$

outer iterations, each involving a Newton solve on the barrier problem.

6.4.4 Newton Step for the Barrier Problem

The centering step requires solving:

$$\underset{x}{\text{minimize}} \; t f(x) + \phi(x) \quad \text{s.t.} \quad Ax = b.$$

The gradient and Hessian of the objective are:

$$\nabla(tf + \phi) = t\nabla f(x) + \sum_{i=1}^{m} \frac{-\nabla g_i(x)}{g_i(x)}, \tag{6.27}$$

$$\nabla^2(tf + \phi) = t\nabla^2 f(x) + \sum_{i=1}^{m} \frac{\nabla g_i(x)\nabla g_i(x)^T}{g_i(x)^2} + \sum_{i=1}^{m} \frac{-\nabla^2 g_i(x)}{g_i(x)}. \tag{6.28}$$

The Newton step is found by solving the KKT system:

$$\begin{bmatrix} \nabla^2(tf + \phi) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \nu \end{bmatrix} = \begin{bmatrix} -\nabla(tf + \phi) \\ 0 \end{bmatrix}. \tag{6.29}$$

6.4.5 Implementation of the Barrier Method

```
import numpy as np
from scipy.linalg import cho_factor, cho_solve

def barrier_method(f, grad_f, hess_f, g_list, grad_g_list, hess_g_list,
                   A, b, x0, t0=1.0, mu=10.0, tol=1e-8, max_outer=50,
                   max_inner=50, alpha=0.25, beta=0.5, verbose=True):
    """
    Barrier method for inequality-constrained optimization.

    min f(x)  s.t.  g_i(x) <= 0, i=1,...,m,  Ax = b

    Parameters:
        f, grad_f, hess_f: objective and derivatives
        g_list: list of constraint functions g_i(x)
        grad_g_list, hess_g_list: constraint derivatives
        A, b: equality constraint matrix and vector (can be None)
        x0: strictly feasible initial point
        t0: initial barrier parameter
        mu: barrier parameter growth factor (typically 10-20)
        tol: duality gap tolerance
    """
    x = np.array(x0, dtype=float)
    n = len(x)
    m = len(g_list)  # Number of inequality constraints
    t = t0

    # Handle equality constraints
    if A is not None:
        p = A.shape[0]
    else:
```

```
        p = 0
        A = np.zeros((0, n))
        b = np.zeros(0)

    history = {
        'x': [x.copy()],
        'f': [f(x)],
        't': [t],
        'duality_gap': [m / t],
        'newton_iters': []
    }

    def barrier(x):
        """Logarithmic barrier function."""
        val = 0.0
        for g_i in g_list:
            gi = g_i(x)
            if gi >= 0:
                return np.inf
            val -= np.log(-gi)
        return val

    def barrier_grad(x):
        """Gradient of barrier function."""
        grad = np.zeros(n)
        for g_i, grad_g_i in zip(g_list, grad_g_list):
            gi = g_i(x)
            grad -= grad_g_i(x) / gi
        return grad

    def barrier_hess(x):
        """Hessian of barrier function."""
        H = np.zeros((n, n))
        for g_i, grad_g_i, hess_g_i in zip(g_list, grad_g_list, hess_g_list):
            gi = g_i(x)
            grad_gi = grad_g_i(x)
            H += np.outer(grad_gi, grad_gi) / (gi ** 2)
            H -= hess_g_i(x) / gi
        return H

    for outer in range(max_outer):
        # Check duality gap
        gap = m / t
        if verbose:
```

```
            print(f"Outer {outer}: t = {t:.2e}, gap = {gap:.2e}, f = {f(x):.6e}")

        if gap < tol:
            if verbose:
                print(f"Converged: duality gap {gap:.2e} < {tol}")
            break

        # Centering: minimize t*f(x) + phi(x) subject to Ax = b
        # using Newton's method
        newton_iters = 0

        for inner in range(max_inner):
            # Gradient and Hessian of t*f + phi
            g_total = t * grad_f(x) + barrier_grad(x)
            H_total = t * hess_f(x) + barrier_hess(x)

            # Form KKT system
            if p > 0:
                KKT = np.block([
                    [H_total, A.T],
                    [A, np.zeros((p, p))]
                ])
                rhs = np.concatenate([-g_total, np.zeros(p)])
            else:
                KKT = H_total
                rhs = -g_total

            # Solve for Newton step
            try:
                if p > 0:
                    sol = np.linalg.solve(KKT, rhs)
                    dx = sol[:n]
                else:
                    c, lower = cho_factor(KKT)
                    dx = cho_solve((c, lower), rhs)
            except np.linalg.LinAlgError:
                print("Warning: KKT system singular")
                break

            # Newton decrement
            lambda_sq = -g_total @ dx

            if lambda_sq / 2 < 1e-10:
                break
```

```
            # Backtracking line search (ensure feasibility)
            eta = 1.0

            # First, find maximum step that maintains feasibility
            for g_i in g_list:
                # Binary search for feasibility
                while eta > 1e-10 and g_i(x + eta * dx) >= 0:
                    eta *= 0.5

            eta *= 0.99  # Stay strictly inside

            # Now backtrack for sufficient decrease
            f_curr = t * f(x) + barrier(x)
            while eta > 1e-16:
                x_new = x + eta * dx
                f_new = t * f(x_new) + barrier(x_new)
                if f_new <= f_curr - alpha * eta * lambda_sq:
                    break
                eta *= beta

            x = x + eta * dx
            newton_iters += 1

        history['x'].append(x.copy())
        history['f'].append(f(x))
        history['t'].append(t)
        history['duality_gap'].append(m / t)
        history['newton_iters'].append(newton_iters)

        # Increase t
        t *= mu

    return x, history


def demo_barrier_method():
    """Demo: quadratic program with inequality constraints."""
    np.random.seed(42)

    # min 0.5 * x'Qx + c'x
    # s.t. Gx <= h

    n = 5
```

```
m = 8  # Number of inequality constraints

# Objective
Q = np.eye(n) + 0.5 * np.random.randn(n, n)
Q = Q @ Q.T  # Positive definite
c = np.random.randn(n)

f = lambda x: 0.5 * x @ Q @ x + c @ x
grad_f = lambda x: Q @ x + c
hess_f = lambda x: Q

# Inequality constraints: Gx <= h (written as Gx - h <= 0)
G = np.random.randn(m, n)
h = np.abs(np.random.randn(m)) + 1  # Ensure feasible region is non-empty

g_list = [lambda x, i=i: G[i] @ x - h[i] for i in range(m)]
grad_g_list = [lambda x, i=i: G[i] for i in range(m)]
hess_g_list = [lambda x: np.zeros((n, n)) for _ in range(m)]

# Find strictly feasible initial point
# Solve: min ||x||^2 s.t. Gx <= h - epsilon
from scipy.optimize import minimize
eps = 0.1
res = minimize(lambda x: np.sum(x**2), np.zeros(n),
               constraints={'type': 'ineq', 'fun': lambda x: h - eps - G @ x})
x0 = res.x

print("="*60)
print("Barrier Method Demo: Quadratic Program")
print("="*60)
print(f"Variables: {n}, Inequality constraints: {m}")
print(f"Initial point feasibility: {all(G @ x0 - h < 0)}")

x_opt, history = barrier_method(
    f, grad_f, hess_f,
    g_list, grad_g_list, hess_g_list,
    A=None, b=None,
    x0=x0, t0=1.0, mu=15.0, tol=1e-8, verbose=True
)

print(f"\nOptimal value: {f(x_opt):.6f}")
print(f"Constraint satisfaction: {[g_i(x_opt) for g_i in g_list[:3]]}...")

# Plot convergence
```

```
    import matplotlib.pyplot as plt

    fig, axes = plt.subplots(1, 3, figsize=(14, 4))

    axes[0].semilogy(history['duality_gap'], 'b-o', linewidth=2, markersize=6)
    axes[0].set_xlabel('Outer Iteration')
    axes[0].set_ylabel('Duality Gap $m/t$')
    axes[0].set_title('Duality Gap Convergence')
    axes[0].grid(True, alpha=0.3)

    axes[1].semilogy(history['t'], 'r-o', linewidth=2, markersize=6)
    axes[1].set_xlabel('Outer Iteration')
    axes[1].set_ylabel('Barrier Parameter $t$')
    axes[1].set_title('Barrier Parameter Growth')
    axes[1].grid(True, alpha=0.3)

    axes[2].bar(range(len(history['newton_iters'])), history['newton_iters'])
    axes[2].set_xlabel('Outer Iteration')
    axes[2].set_ylabel('Newton Iterations')
    axes[2].set_title('Centering Steps per Outer Iteration')
    axes[2].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('barrier_method.pdf', bbox_inches='tight')
    plt.show()

    return x_opt, history

demo_barrier_method()
```

Remark 6.17 (Choosing the barrier parameter growth factor μ). The growth factor μ controls the trade-off between outer and inner iterations:

- **Large** μ (e.g., $\mu = 50$): Fewer outer iterations, but each centering step requires more Newton iterations because the problem changes dramatically.
- **Small** μ (e.g., $\mu = 2$): More outer iterations, but each centering step is easier (warm start is effective).

In practice, $\mu \in [10, 20]$ often works well. The total Newton iteration count is relatively insensitive to μ within this range.

6.4.6 Primal-Dual Interior Point Methods

The barrier method requires solving a sequence of unconstrained (or equality-constrained) problems. **Primal-dual interior point methods** take a more direct approach: they solve the modified KKT conditions (6.23) directly using Newton's method on the combined primal-dual system.

The Primal-Dual System

Recall the modified KKT conditions for problem (6.15):

$$\nabla f(x) + \sum_{i=1}^{m} \lambda_i \nabla g_i(x) + A^T \nu = 0, \tag{6.30}$$

$$\lambda_i g_i(x) + 1/t = 0, \quad i = 1, \ldots, m, \tag{6.31}$$

$$Ax - b = 0. \tag{6.32}$$

Define the **residual** $r_t(x, \lambda, \nu)$:

$$r_t(x, \lambda, \nu) = \begin{bmatrix} r_{\text{dual}} \\ r_{\text{cent}} \\ r_{\text{pri}} \end{bmatrix} = \begin{bmatrix} \nabla f(x) + Dg(x)^T \lambda + A^T \nu \\ -\text{diag}(\lambda) g(x) - (1/t)\mathbf{1} \\ Ax - b \end{bmatrix}, \tag{6.33}$$

where $g(x) = (g_1(x), \ldots, g_m(x))^T$ and $Dg(x) \in \mathbb{R}^{m \times n}$ is the Jacobian.

The modified KKT conditions are $r_t(x, \lambda, \nu) = 0$.

Newton Step for the Primal-Dual System

Newton's method linearizes r_t and solves for the step $(\Delta x, \Delta\lambda, \Delta\nu)$:

$$\begin{bmatrix} \nabla^2 f + \sum_i \lambda_i \nabla^2 g_i & Dg^T & A^T \\ -\text{diag}(\lambda) Dg & -\text{diag}(g) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta\nu \end{bmatrix} = - \begin{bmatrix} r_{\text{dual}} \\ r_{\text{cent}} \\ r_{\text{pri}} \end{bmatrix}. \tag{6.34}$$

This is a system of $(n+m+p)$ equations. The matrix is not symmetric, but the system can be reduced to a smaller symmetric system.

Eliminating $\Delta\lambda$

From the second block equation:

$$-\text{diag}(\lambda) Dg\, \Delta x - \text{diag}(g)\, \Delta\lambda = -r_{\text{cent}}.$$

Solving for $\Delta\lambda$:

$$\Delta\lambda = -\text{diag}(g)^{-1} \left(\text{diag}(\lambda) Dg\, \Delta x + r_{\text{cent}} \right). \tag{6.35}$$

Substituting into the first block equation and simplifying yields the **reduced KKT system**:

$$\begin{bmatrix} H_{\text{pd}} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \nu \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}, \tag{6.36}$$

where:

$$H_{\text{pd}} = \nabla^2 f + \sum_i \lambda_i \nabla^2 g_i + \sum_i \frac{\lambda_i}{-g_i} \nabla g_i \nabla g_i^T, \tag{6.37}$$

$$r_1 = -r_{\text{dual}} - Dg^T \text{diag}(g)^{-1} r_{\text{cent}}, \tag{6.38}$$

$$r_2 = -r_{\text{pri}}. \tag{6.39}$$

The matrix H_{pd} is positive definite when (x, λ, ν) is near the central path with $\lambda > 0$.

Algorithm 21 Primal-Dual Interior Point Method

Input: Strictly feasible $(x^{(0)}, \lambda^{(0)}, \nu^{(0)})$ with $\lambda^{(0)} > 0$, $g(x^{(0)}) < 0$
Parameters: $\mu > 1$, $\alpha, \beta \in (0, 1)$, tolerance $\epsilon_{\text{feas}}, \epsilon$
for $k = 0, 1, 2, \ldots$ **do**
  Compute duality gap: $\eta = -g(x)^T \lambda$
  Set $t = \mu m / \eta$
  Compute residual $r_t(x, \lambda, \nu)$ via (6.33)
  Check convergence:
  if $\|r_{\text{pri}}\| \le \epsilon_{\text{feas}}$, $\|r_{\text{dual}}\| \le \epsilon_{\text{feas}}$, $\eta \le \epsilon$ **then**
    return (x, λ, ν)
  end if
  Solve (6.34) for $(\Delta x, \Delta \lambda, \Delta \nu)$
  Line search: Find largest $s \in (0, 1]$ such that
  - $\lambda + s\Delta\lambda > 0$ (dual feasibility)
  - $g(x + s\Delta x) < 0$ (primal feasibility)
  - $\|r_t(x + s\Delta x, \lambda + s\Delta\lambda, \nu + s\Delta\nu)\| \le (1 - \alpha s)\|r_t\|$
  Update: $(x, \lambda, \nu) \leftarrow (x + s\Delta x, \lambda + s\Delta\lambda, \nu + s\Delta\nu)$
end for

Theorem 6.18 (Convergence of primal-dual IPM)**.** *Under standard assumptions (strict feasibility, constraint qualification), the primal-dual interior point method converges to an optimal solution. Moreover:*

1. *The algorithm requires $O(\sqrt{m} \log(1/\epsilon))$ Newton iterations to achieve duality gap $\le \epsilon$.*
2. *Each Newton iteration costs $O(n^3 + n^2 m)$ for dense problems.*
3. *The bound $O(\sqrt{m})$ is tight: there exist problems requiring $\Omega(\sqrt{m})$ iterations.*

Remark 6.19 (Comparison: Barrier vs Primal-Dual).

- **Barrier method**: Conceptually simpler; each centering step is a standard equality-constrained Newton problem. Requires strictly feasible start.
- **Primal-dual**: Solves primal and dual simultaneously; can handle infeasible starts; often faster in practice due to aggressive step sizes.

Both achieve $O(\sqrt{m}\log(1/\epsilon))$ iteration complexity—a remarkable improvement over simplex-type methods that can require exponentially many iterations in the worst case.

```
import numpy as np
from scipy.linalg import solve

def primal_dual_ipm(f, grad_f, hess_f, g_list, grad_g_list, hess_g_list,
                    A, b, x0, lambda0, nu0, mu=10.0, tol=1e-8,
                    tol_feas=1e-8, max_iter=100, alpha=0.01, beta=0.5):
    """
    Primal-dual interior point method.

    min f(x)  s.t.  g_i(x) <= 0, Ax = b
    """
    x = np.array(x0, dtype=float)
    lam = np.array(lambda0, dtype=float)  # Dual for inequalities
    nu = np.array(nu0, dtype=float) if nu0 is not None else np.zeros(0)

    n = len(x)
    m = len(g_list)
    p = len(nu)

    history = {'x': [x.copy()], 'gap': [], 'residual': []}

    for iteration in range(max_iter):
        # Evaluate constraints
        g = np.array([g_i(x) for g_i in g_list])
        Dg = np.array([grad_g_i(x) for grad_g_i in grad_g_list])  # m x n

        # Surrogate duality gap
        eta = -g @ lam

        # Barrier parameter
        t = mu * m / eta

        # Residuals
        r_dual = grad_f(x) + Dg.T @ lam
        if p > 0:
            r_dual += A.T @ nu
```

```
        r_cent = -lam * g - 1.0 / t
        r_pri = A @ x - b if p > 0 else np.zeros(0)

        # Residual norm
        r_norm = np.sqrt(np.sum(r_dual**2) + np.sum(r_cent**2) + np.sum(r_pri**2))

        history['gap'].append(eta)
        history['residual'].append(r_norm)

        # Check convergence
        if (np.linalg.norm(r_pri) <= tol_feas and
            np.linalg.norm(r_dual) <= tol_feas and
            eta <= tol):
            print(f"Converged at iteration {iteration}")
            break

        # Build Hessian of Lagrangian
        H = hess_f(x).copy()
        for i in range(m):
            H += lam[i] * hess_g_list[i](x)

        # Add barrier contribution: sum_i (lambda_i / (-g_i)) * grad_g_i grad_g_i^T
        for i in range(m):
            grad_gi = Dg[i]
            H += (lam[i] / (-g[i])) * np.outer(grad_gi, grad_gi)

        # Modified RHS
        r1 = -r_dual - Dg.T @ (r_cent / (-g))
        r2 = -r_pri

        # Solve reduced KKT system
        if p > 0:
            KKT = np.block([[H, A.T], [A, np.zeros((p, p))]])
            rhs = np.concatenate([r1, r2])
            sol = solve(KKT, rhs)
            dx = sol[:n]
            dnu = sol[n:]
        else:
            dx = solve(H, r1)
            dnu = np.zeros(0)

        # Recover dlambda
        dlam = (r_cent - lam * (Dg @ dx)) / (-g)
```

```
        # Line search
        # Step 1: Find maximum step for lam + s*dlam > 0
        s_max = 1.0
        for i in range(m):
            if dlam[i] < 0:
                s_max = min(s_max, -lam[i] / dlam[i] * 0.99)

        # Step 2: Find maximum step for g(x + s*dx) < 0
        s = s_max
        for _ in range(50):
            x_new = x + s * dx
            g_new = np.array([g_i(x_new) for g_i in g_list])
            if np.all(g_new < 0):
                break
            s *= 0.5

        # Step 3: Backtrack for residual decrease
        for _ in range(50):
            x_new = x + s * dx
            lam_new = lam + s * dlam
            nu_new = nu + s * dnu if p > 0 else nu

            # Compute new residual
            g_new = np.array([g_i(x_new) for g_i in g_list])
            Dg_new = np.array([grad_g_i(x_new) for grad_g_i in grad_g_list])

            r_dual_new = grad_f(x_new) + Dg_new.T @ lam_new
            if p > 0:
                r_dual_new += A.T @ nu_new
            r_cent_new = -lam_new * g_new - 1.0 / t
            r_pri_new = A @ x_new - b if p > 0 else np.zeros(0)

            r_norm_new = np.sqrt(np.sum(r_dual_new**2) + np.sum(r_cent_new**2) +
                                 np.sum(r_pri_new**2))

            if r_norm_new <= (1 - alpha * s) * r_norm:
                break
            s *= beta

        # Update
        x = x + s * dx
        lam = lam + s * dlam
        if p > 0:
            nu = nu + s * dnu
```

```
        history['x'].append(x.copy())

        if iteration % 5 == 0:
            print(f"Iter {iteration}: gap = {eta:.2e}, ||r|| = {r_norm:.2e}, s = {s:.4f}")

    return x, lam, nu, history


def demo_primal_dual():
    """Compare barrier method and primal-dual IPM."""
    np.random.seed(42)

    # Simple QP: min 0.5*x'Qx + c'x s.t. x >= 0 (i.e., -x <= 0)
    n = 10
    Q = np.eye(n) + 0.3 * np.random.randn(n, n)
    Q = Q @ Q.T
    c = np.random.randn(n)

    f = lambda x: 0.5 * x @ Q @ x + c @ x
    grad_f = lambda x: Q @ x + c
    hess_f = lambda x: Q

    # Constraints: -x_i <= 0 (i.e., x >= 0)
    m = n
    g_list = [lambda x, i=i: -x[i] for i in range(m)]
    grad_g_list = [lambda x, i=i: -np.eye(n)[i] for i in range(m)]
    hess_g_list = [lambda x: np.zeros((n, n)) for _ in range(m)]

    # Initial point
    x0 = np.ones(n)  # Strictly feasible
    lambda0 = np.ones(m)  # Positive dual variables

    print("="*60)
    print("Primal-Dual Interior Point Method Demo")
    print("="*60)

    x_opt, lam_opt, nu_opt, history = primal_dual_ipm(
        f, grad_f, hess_f,
        g_list, grad_g_list, hess_g_list,
        A=None, b=None,
        x0=x0, lambda0=lambda0, nu0=None,
        mu=10.0, tol=1e-10
    )
```

```
    print(f"\nOptimal x: {x_opt}")
    print(f"Optimal value: {f(x_opt):.6f}")
    print(f"Constraint satisfaction (should be >= 0): {x_opt.min():.6f}")

    # Plot convergence
    import matplotlib.pyplot as plt

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].semilogy(history['gap'], 'b-o', linewidth=2)
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('Duality Gap')
    axes[0].set_title('Duality Gap Convergence')
    axes[0].grid(True, alpha=0.3)

    axes[1].semilogy(history['residual'], 'r-o', linewidth=2)
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('Residual Norm')
    axes[1].set_title('KKT Residual Convergence')
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('primal_dual_ipm.pdf', bbox_inches='tight')
    plt.show()

demo_primal_dual()
```

6.4.7 Interior Point Methods for Linear Programming

Interior point methods revolutionized linear programming. For the standard LP:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}}\, c^T x \quad \text{s.t.} \quad Ax = b,\ x \geq 0, \tag{6.40}$$

the barrier problem is:

$$\underset{x}{\text{minimize}}\ t \cdot c^T x - \sum_{i=1}^{n} \log x_i \quad \text{s.t.} \quad Ax = b.$$

The Newton step for the centering problem has special structure that can be exploited.

Proposition 6.20 (LP Newton system). *For the LP barrier problem, the Newton step satisfies:*

$$\begin{bmatrix} D^{-2} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \nu \end{bmatrix} = \begin{bmatrix} -tc + D^{-1}\mathbf{1} \\ 0 \end{bmatrix}, \tag{6.41}$$

where $D = diag(x)$. This can be reduced to the ***normal equations****:*

$$(AD^2A^T)\nu = AD^2(tc - D^{-1}\mathbf{1}), \quad \Delta x = D^2(A^T\nu - tc + D^{-1}\mathbf{1}). \tag{6.42}$$

The matrix AD^2A^T is $p \times p$ (number of equality constraints), which is often much smaller than n. Moreover, if A is sparse, AD^2A^T inherits sparsity structure.

Theorem 6.21 (LP interior point complexity)**.** *Interior point methods solve linear programs with n variables and m constraints in:*

$$O(\sqrt{n}\log(n/\epsilon))$$

iterations, each requiring $O(n^3)$ or $O(n \cdot nnz(A) + n^2)$ operations depending on sparsity. This gives polynomial-time complexity, in contrast to the simplex method's exponential worst-case.

```
import numpy as np
from scipy.linalg import cho_factor, cho_solve

def lp_interior_point(c, A, b, x0=None, tol=1e-8, mu=10.0, max_iter=100):
    """
    Interior point method for LP in standard form.

    min c'x  s.t.  Ax = b, x >= 0

    Uses barrier method with log barrier for x >= 0.
    """
    m, n = A.shape

    # Initialize with strictly feasible point
    if x0 is None:
        # Phase I: find feasible point (simplified)
        x = np.ones(n)
        # Scale to satisfy Ax = b approximately
        # This is a simplified initialization
        x = np.linalg.lstsq(A, b, rcond=None)[0]
        x = np.maximum(x, 0.1)  # Ensure positivity
    else:
        x = x0.copy()

    t = 1.0

    history = {'objective': [], 'gap': []}

    for outer in range(max_iter):
        gap = n / t
        history['objective'].append(c @ x)
```

```
        history['gap'].append(gap)

        if gap < tol:
            print(f"Converged at outer iteration {outer}")
            break

        # Centering: Newton's method
        for inner in range(50):
            # Gradient and Hessian of t*c'x - sum(log(x_i))
            grad = t * c - 1.0 / x
            H = np.diag(1.0 / x**2)

            # KKT system: [H A'; A 0] [dx; nu] = [-grad; 0]
            KKT = np.block([[H, A.T], [A, np.zeros((m, m))]])
            rhs = np.concatenate([-grad, np.zeros(m)])

            sol = np.linalg.solve(KKT, rhs)
            dx = sol[:n]

            # Newton decrement
            lambda_sq = -grad @ dx
            if lambda_sq / 2 < 1e-10:
                break

            # Line search (maintain x > 0)
            s = 1.0
            for i in range(n):
                if dx[i] < 0:
                    s = min(s, -x[i] / dx[i] * 0.99)

            # Backtrack
            f_curr = t * c @ x - np.sum(np.log(x))
            for _ in range(50):
                x_new = x + s * dx
                if np.all(x_new > 0):
                    f_new = t * c @ x_new - np.sum(np.log(x_new))
                    if f_new <= f_curr - 0.25 * s * lambda_sq:
                        break
                s *= 0.5

            x = x + s * dx

        t *= mu
```

```
    return x, history


def demo_lp_ipm():
    """Demo: solve a random LP with interior point method."""
    np.random.seed(42)

    n = 100  # Variables
    m = 30   # Equality constraints

    # Random LP: min c'x s.t. Ax = b, x >= 0
    c = np.random.rand(n)  # Non-negative costs
    A = np.random.randn(m, n)

    # Generate feasible b
    x_true = np.random.rand(n) + 0.1  # Strictly positive
    b = A @ x_true

    print("="*60)
    print("LP Interior Point Method Demo")
    print(f"Variables: {n}, Constraints: {m}")
    print("="*60)

    x_opt, history = lp_interior_point(c, A, b, tol=1e-10)

    print(f"Optimal value: {c @ x_opt:.6f}")
    print(f"Feasibility ||Ax - b||: {np.linalg.norm(A @ x_opt - b):.2e}")
    print(f"Minimum x_i: {x_opt.min():.6f}")

    # Plot
    import matplotlib.pyplot as plt

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].plot(history['objective'], 'b-', linewidth=2)
    axes[0].set_xlabel('Outer Iteration')
    axes[0].set_ylabel('Objective $c^T x$')
    axes[0].set_title('Objective Value')
    axes[0].grid(True, alpha=0.3)

    axes[1].semilogy(history['gap'], 'r-', linewidth=2)
    axes[1].set_xlabel('Outer Iteration')
    axes[1].set_ylabel('Duality Gap')
    axes[1].set_title('Convergence')
```

```
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('lp_ipm.pdf', bbox_inches='tight')
    plt.show()

demo_lp_ipm()
```

Check Your Understanding 6.3.

(a) Why does the barrier method require a strictly *feasible starting point?*

(b) In the primal-dual method, why do we need both $\lambda > 0$ and $g(x) < 0$ throughout?

(c) The iteration complexity $O(\sqrt{m})$ depends on the number of inequality constraints. Why doesn't it depend on n?

(d) For LP, the simplex method has exponential worst-case but often outperforms IPM on small problems. Why?

6.5 Summary and Method Selection

This chapter developed second-order methods that exploit curvature information. Table 6.1 summarizes the key methods.

Table 6.1: Summary of second-order methods.

Method	**Problem Type**	**Per-Iteration Cost**	**Convergence**
Newton	Unconstrained	$O(n^3)$ Hessian solve	Quadratic (local)
Damped Newton	Unconstrained	$O(n^3)$ + line search	Linear $\to$ Quadratic
BFGS	Unconstrained	$O(n^2)$ update	Superlinear
L-BFGS	Unconstrained (large)	$O(mn)$	Superlinear
Newton (equality)	$Ax = b$	$O((n+p)^3)$ KKT	Quadratic (local)
Barrier Method	Inequality constraints	$O(n^3)$ per Newton	$O(\sqrt{m}\log(1/\epsilon))$ outer
Primal-Dual IPM	Inequality constraints	$O(n^3 + n^2 m)$	$O(\sqrt{m}\log(1/\epsilon))$

6.5.1 When to Use Second-Order Methods

Use Newton's method when:

- Problem dimension n is moderate (hundreds to low thousands)
- High accuracy is required
- The Hessian is available and cheap to compute
- The problem is well-conditioned or ill-conditioned (Newton is affine-invariant)

Use quasi-Newton (BFGS/L-BFGS) when:

- Hessian is expensive or unavailable
- Problem is medium to large scale
- Superlinear convergence is desired but $O(n^3)$ is too expensive

Use interior point methods when:

- Problem has inequality constraints
- Moderate accuracy suffices (10^{-6} to 10^{-8})
- Problem structure allows efficient Newton solves (sparsity)

Prefer first-order methods when:

- Problem is very large scale ($n > 10^6$)
- Low to moderate accuracy suffices
- Memory is limited
- Problem has special structure (sparsity, separability)

6.5.2 Complexity Comparison

Table 6.2 compares iteration complexity and per-iteration cost.

Table 6.2: Complexity comparison for achieving ϵ-accuracy.

Method	Iterations	Per-Iteration	Total (dense)
Gradient Descent	$O(\kappa \log(1/\epsilon))$	$O(n)$	$O(\kappa n \log(1/\epsilon))$
Accelerated GD	$O(\sqrt{\kappa} \log(1/\epsilon))$	$O(n)$	$O(\sqrt{\kappa} n \log(1/\epsilon))$
Newton	$O(\log\log(1/\epsilon))$	$O(n^3)$	$O(n^3 \log\log(1/\epsilon))$
BFGS	$O(n)$ (quadratic)	$O(n^2)$	$O(n^3)$
L-BFGS	$O(n)$ (superlinear)	$O(mn)$	$O(mn^2)$
IPM	$O(\sqrt{m} \log(1/\epsilon))$	$O(n^3)$	$O(\sqrt{m} n^3 \log(1/\epsilon))$

Remark 6.22 (The $\log\log(1/\epsilon)$ term). Newton's quadratic convergence means the $\log\log(1/\epsilon)$ term is essentially constant for practical purposes. Achieving $\epsilon = 10^{-16}$ (machine precision) requires only about $\log_2(\log_2(10^{16})) \approx 6$ quadratic-convergence iterations after entering the local region.

6.6 Exercises

Unconstrained Newton

1. **(Newton on exponentials)** Consider $f(x) = e^x + e^{-x}$.
   (a) Find the optimal x^* analytically.
   (b) Compute the Newton step at $x = 1$ and at $x = 0.1$.
   (c) Implement Newton's method and verify quadratic convergence.
   (d) How many iterations to reach machine precision from $x^{(0)} = 5$?

2. **(Affine invariance)** Let $f(x) = \frac{1}{2}x^T Q x$ with $Q = \text{diag}(1, 100)$.
   (a) Run gradient descent from $x^{(0)} = (1, 1)^T$. How many iterations?
   (b) Run Newton's method from the same point. How many iterations?
   (c) Transform variables: $\tilde{x} = Dx$ where $D = \text{diag}(1, 10)$. Verify Newton gives identical iterates (after transformation).
   (d) Does gradient descent give identical iterates after transformation?

3. **(Self-concordant functions)** A function $f : \mathbb{R} \to \mathbb{R}$ is **self-concordant** if $|f'''(x)| \leq 2f''(x)^{3/2}$ for all x.
   (a) Show that $f(x) = -\log(x)$ is self-concordant on $x > 0$.
   (b) Show that $f(x) = -\log(1 - x^2)$ is self-concordant on $|x| < 1$.
   (c) Why is self-concordance important for barrier methods?

4. **(Modified Newton)** When the Hessian is not positive definite, pure Newton may not descend.
   (a) Give an example where Newton ascends (increases f).
   (b) Implement **modified Newton**: replace H with $H + \tau I$ where τ ensures positive definiteness.
   (c) How should τ be chosen adaptively?

Quasi-Newton Methods

5. **(BFGS derivation)** The BFGS update minimizes $\|W_{k+1} - W_k\|_F$ subject to the secant condition $W_{k+1} y_k = s_k$ and symmetry.
   (a) Verify that the BFGS formula satisfies the secant condition.
   (b) Prove that if $W_k \succ 0$ and $s_k^T y_k > 0$, then $W_{k+1} \succ 0$.
   (c) What happens if $s_k^T y_k \leq 0$? How is this handled in practice?

6. **(DFP method)** The Davidon-Fletcher-Powell (DFP) method updates $B_k \approx H_k$ (not the inverse):

$$B_{k+1} = (I - \rho_k y_k s_k^T) B_k (I - \rho_k s_k y_k^T) + \rho_k y_k y_k^T.$$

   (a) Show this satisfies $B_{k+1} s_k = y_k$.

   (b) Implement DFP and compare with BFGS on the Rosenbrock function.

   (c) Why is BFGS generally preferred over DFP?

7. **(L-BFGS memory)**

   (a) Implement L-BFGS with $m \in \{3, 5, 10, 20\}$ and compare convergence on a large quadratic.

   (b) For what value of m does L-BFGS match full BFGS performance?

   (c) What is the storage requirement of L-BFGS vs BFGS for $n = 10^6$?

Equality-Constrained Newton

8. **(KKT system structure)** For the equality-constrained problem with n variables and p constraints:

   (a) Show that the KKT matrix is invertible iff $H \succ 0$ on $\text{null}(A)$.

   (b) Derive the **Schur complement** method for solving the KKT system.

   (c) When is the Schur complement method more efficient than direct solution?

9. **(Infeasible-start Newton)** Implement infeasible-start Newton for:

$$\min_x \frac{1}{2}\|x\|^2 \quad \text{s.t.} \quad Ax = b.$$

   (a) Start from $x^{(0)} = 0$ (infeasible unless $b = 0$).

   (b) Track primal infeasibility $\|Ax^{(k)} - b\|$ and optimality.

   (c) How many iterations to reach feasibility? Optimality?

10. **(Projected Newton)** An alternative to solving the KKT system is **projected Newton**:

$$x^{(k+1)} = x^{(k)} - P_A H^{-1} P_A \nabla f(x^{(k)}),$$

    where $P_A = I - A^T(AA^T)^{-1}A$ projects onto $\text{null}(A)$.

   (a) Show that this preserves feasibility if $x^{(0)}$ is feasible.

   (b) When does this coincide with the KKT-based Newton step?

   (c) Implement and compare with KKT-based Newton.

Interior Point Methods

11. **(Barrier function properties)** For the log barrier $\phi(x) = -\sum_i \log(-g_i(x))$:
    (a) Show that ϕ is convex if each g_i is convex.
    (b) Compute $\nabla\phi$ and $\nabla^2\phi$ for $g_i(x) = a_i^T x - b_i$.
    (c) What is the condition number of $\nabla^2\phi$ as x approaches a constraint boundary?

12. **(Central path)** Consider the LP: $\min\{c^T x : Ax \leq b\}$.
    (a) Write the optimality conditions for points on the central path.
    (b) Show that the central path is parameterized by $t > 0$ and converges to an optimal vertex as $t \to \infty$.
    (c) Plot the central path for a 2D example.

13. **(Warm starting)** Interior point methods benefit from warm starts.
    (a) Solve a sequence of LPs $\min\{c^T x : Ax \leq b + \delta_k\}$ where $\delta_k \to 0$.
    (b) Compare cold start (from scratch) vs warm start (from previous solution).
    (c) How does the number of Newton iterations change with warm starting?

14. **(Primal-dual vs barrier)** Compare the barrier method and primal-dual IPM:
    (a) Implement both for a QP with box constraints.
    (b) Compare total Newton iterations to reach 10^{-8} accuracy.
    (c) Which method is more robust to the initial point?

15. **(Mehrotra predictor-corrector)** The **Mehrotra predictor-corrector** method is a practical enhancement:
    (a) Compute an **affine-scaling** direction (set $t = \infty$).
    (b) Use this to adaptively choose the centering parameter $\sigma \in [0, 1]$.
    (c) Add a **corrector** term to the Newton step.
    (d) Implement and compare with basic primal-dual IPM.

Applications

16. **(Logistic regression via Newton)** For logistic regression:

$$\min_w \sum_{i=1}^{m} \log(1 + e^{-y_i w^T x_i}) + \frac{\lambda}{2}\|w\|^2.$$

    (a) Derive ∇f and $\nabla^2 f$.

(b) Implement Newton's method and compare with gradient descent.

(c) How does the condition number of the Hessian change during optimization?

17. **(Portfolio optimization)** The Markowitz portfolio problem is:

$$\min_w w^T \Sigma w \quad \text{s.t.} \quad \mu^T w \geq r,\ \mathbf{1}^T w = 1,\ w \geq 0.$$

(a) Formulate as a QP and solve with interior point methods.

(b) Plot the efficient frontier (vary r and plot risk vs return).

(c) How does sparsity in Σ affect IPM performance?

18. **(Support vector machine)** The SVM dual is:

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \quad \text{s.t.} \quad 0 \leq \alpha_i \leq C,\ \sum_i \alpha_i y_i = 0.$$

(a) Reformulate as a QP and solve with IPM.

(b) Compare with SMO (Sequential Minimal Optimization).

(c) For what problem sizes is IPM competitive with SMO?

19. **(Semidefinite programming)** Interior point methods extend to SDPs:

$$\min_X \langle C, X \rangle \quad \text{s.t.} \quad \langle A_i, X \rangle = b_i,\ X \succeq 0.$$

(a) What is the appropriate barrier function for $X \succeq 0$?

(b) Derive the gradient and Hessian of the log-det barrier.

(c) Implement a basic SDP solver for small problems.

Theory

20. **(Local convergence of Newton)** Prove that if f is twice continuously differentiable with Lipschitz Hessian, and $x^{(0)}$ is sufficiently close to x^* with $\nabla^2 f(x^*) \succ 0$, then Newton's method converges quadratically.

21. **(Global convergence of damped Newton)** Prove that damped Newton with backtracking converges to a stationary point for any starting point when f is bounded below and has Lipschitz gradient.

22. **(IPM iteration bound)** Prove the $O(\sqrt{m})$ iteration bound for interior point methods:

(a) Show that after one Newton step, the duality gap decreases by a factor depending on the Newton decrement.

(b) Relate the Newton decrement to the barrier parameter t.

(c) Derive the overall bound by tracking t growth.

23. **(Self-concordant barriers)** The iteration complexity $O(\sqrt{m})$ comes from the log barrier being a **self-concordant barrier** with parameter m.

(a) Define self-concordant barriers formally.

(b) Show that the log barrier for $\{x : Ax \leq b\}$ has parameter m (number of constraints).

(c) What is the barrier parameter for the semidefinite cone $\{X : X \succeq 0\}$?

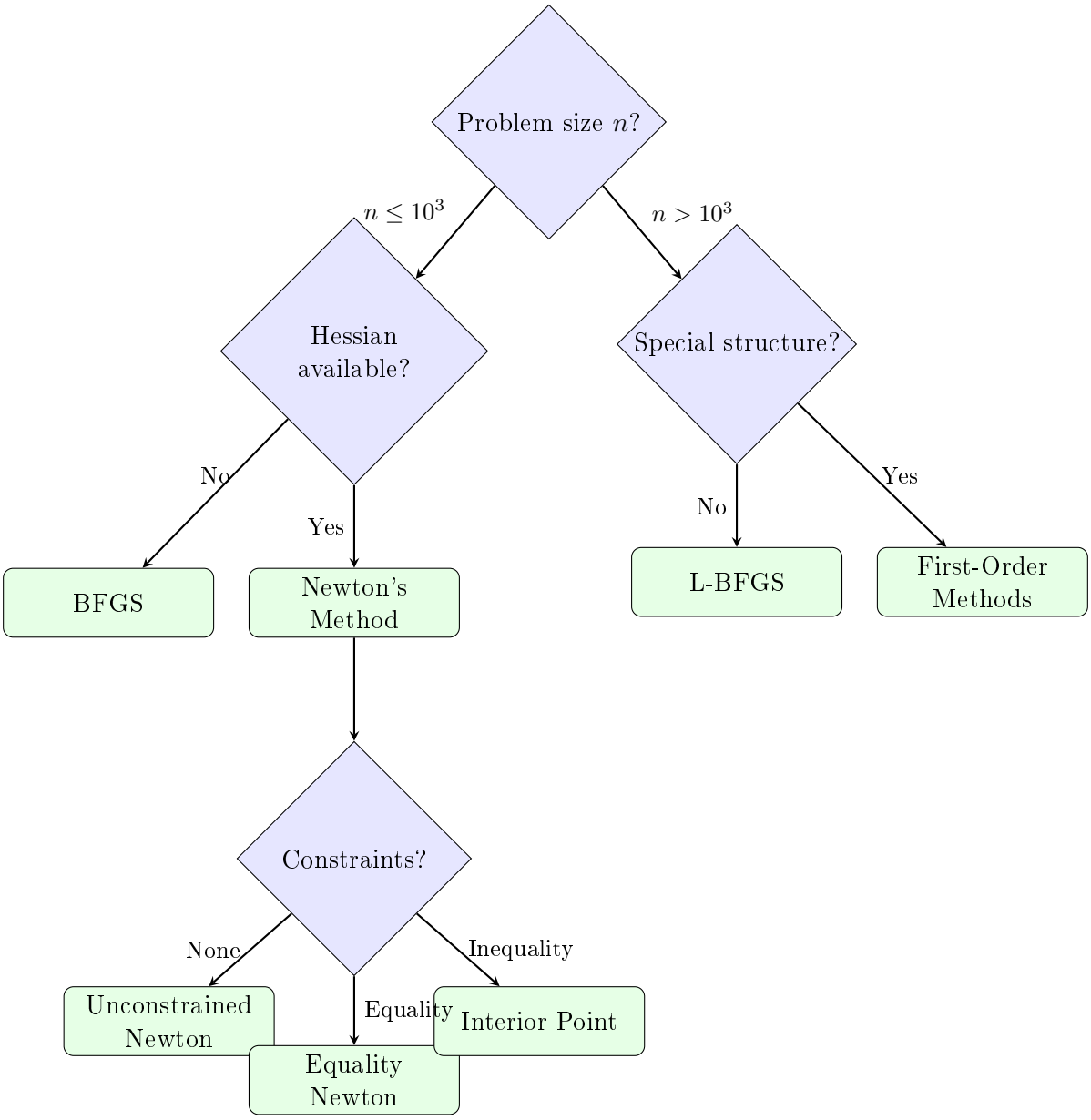

Figure 6.4: Decision flowchart for selecting second-order vs first-order methods.

Selected Solutions

Bibliography

[1] S. Boyd, L. Xiao, and A. Mutapcic, "Subgradient methods." https://web.stanford.edu/class/ee392o/subgrad_method.pdf, 2003. Stanford University, Notes for EE392o.

[2] Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*. Kluwer Academic Publishers, 2004.

[3] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[4] Y. E. Nesterov, "A method for solving the convex programming problem with convergence rate $O(1/k^2)$," in *Doklady Akademii Nauk SSSR*, vol. 269, pp. 543–547, 1983.