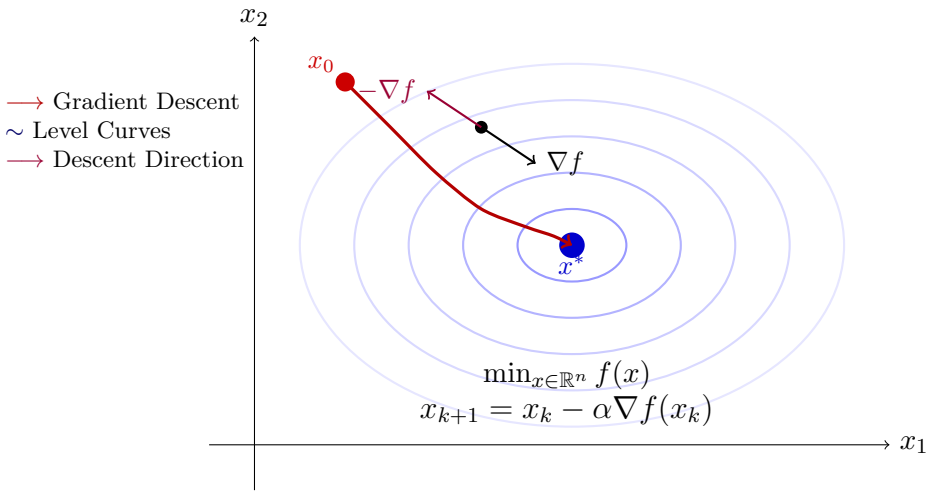
Supplementary of Mathematical Modeling

Unconstrained Optimization

Theory and Applications



Kenneth, Sok Kin Cheng

Last update: July 8, 2025

Elegant Solutions Through Mathematical Optimization

Contents

Chapter 1

Foundations of Unconstrained Optimization

1.1 Introduction

Unconstrained optimization forms the cornerstone of many mathematical modeling applications, from machine learning parameter tuning to engineering design optimization. This chapter establishes the theoretical foundations necessary for understanding optimization algorithms and their practical implementations.

Definition 1.1 (Unconstrained Optimization Problem)**.** An unconstrained optimization problem seeks to find:

$$\min_{x \in \mathbb{R}^n} f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, and $x \in \mathbb{R}^n$ is the decision variable vector.

1.2 Optimality Conditions

Understanding when a point is optimal is crucial for developing effective algorithms.

Theorem

First-Order Necessary Conditions (FONC)
Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable. If x^* is a local minimum of f, then:

$$\nabla f(x^*) = 0$$

Such points are called *stationary points* or *critical points*.

Theorem

Second-Order Necessary Conditions (SONC)
Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice continuously differentiable (in other sense, $f \in C^2$). If x^* is a local minimum of f, then:

1. $\nabla f(x^*) = 0$
2. $\nabla^2 f(x^*) \geq 0$ (Hessian is positive semidefinite)

Theorem

Second-Order Sufficient Conditions (SOSC)
Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable twice. If:

1. $\nabla f(x^*) = 0$
2. $\nabla^2 f(x^*) \succ 0$ (Hessian is positive definite)

then x^* is a strict local minimum of f.

1.3 Convexity and Global Optimality

Definition 1.2 (Convex Function)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for all $x, y \in \mathbb{R}^n$ and $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y)$$

Theorem

Global Optimality for Convex Functions
If f is convex and x^* is a stationary point (i.e., $\nabla f(x^*) = 0$), then x^* is a global minimum of f.

1.4 Practical Examples

Example 1.1 (Quadratic Function)**.** Consider the quadratic function:

$$f(x) = \frac{1}{2} x^T Q x + c^T x + d$$

where $Q \in \mathbb{R}^{n \times n}$ is symmetric, $c \in \mathbb{R}^n$, and $d \in \mathbb{R}$.
The gradient is: $\nabla f(x) = Qx + c$ The Hessian is: $\nabla^2 f(x) = Q$
If $Q \succ 0$, then f is strictly convex, and the unique global minimum is:

$$x^* = -Q^{-1} c$$

Real-World Application

Portfolio Optimization
In finance, the Markowitz portfolio optimization problem seeks to minimize risk while achieving a target return. The problem can be formulated as:

$$\min_w \frac{1}{2} w^T \Sigma w$$

subject to $\sum_{i=1}^n w_i = 1$ and $\mu^T w = r_{\text{target}}$
where w is the portfolio weight vector, Σ is the covariance matrix of asset returns, and μ is the expected return vector.

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

def quadratic_function(x, Q, c):
    """
    Evaluate quadratic function f(x) = 0.5 * x^T Q x + c^T x
    """
    return 0.5 * x.T @ Q @ x + c.T @ x

def quadratic_gradient(x, Q, c):
    """
    Compute gradient of quadratic function
    """
    return Q @ x + c

# Example: 2D quadratic function
Q = np.array([[2, 0.5], [0.5, 1]])
c = np.array([1, -2])

# Analytical solution
x_optimal = -np.linalg.solve(Q, c)
print(f"Optimal solution: {x_optimal}")
print(f"Optimal value: {quadratic_function(x_optimal, Q, c)}")

# Verify optimality conditions
grad_at_optimal = quadratic_gradient(x_optimal, Q, c)
print(f"Gradient at optimal: {grad_at_optimal}")
print(f"Hessian eigenvalues: {np.linalg.eigvals(Q)}")
```

Exploration

Visualization Exercise
Create a contour plot of the quadratic function above and verify that the gradient points in the direction of steepest ascent. Observe how the contour lines reveal the shape of the function and the location of the minimum.

Exercise 1.1. Prove that if $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable and convex, then $\nabla^2 f(x) \succeq 0$ for all $x \in \mathbb{R}^n$.

Exercise 1.2. Consider the function $f(x, y) = x^4 + y^4 - 4xy$. Find all stationary points and classify them using the second-order conditions.

Chapter 2

Gradient-Based Optimization Algorithms

2.1 The Gradient Descent Method

Gradient descent is the fundamental algorithm for unconstrained optimization, forming the basis for many advanced methods.

Definition 2.1 (Gradient Descent Algorithm)**.** Starting from an initial point x_0, the gradient descent method generates a sequence $\{x_k\}$ via:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

where $\alpha_k > 0$ is the step size (learning rate) at iteration k.

2.1.1 Step Size Selection

The choice of step size critically affects convergence behavior:

- **Fixed step size**: $\alpha_k = \alpha$ for all k
- **Exact line search**: $\alpha_k = \arg\min_{\alpha>0} f(x_k - \alpha \nabla f(x_k))$
- **Armijo backtracking**: Choose α_k to satisfy sufficient decrease condition

Theorem

Convergence of Gradient Descent
Suppose f is continuously differentiable and $\inf f > -\infty$. If the step sizes satisfy:

$$\sum_{k=0}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=0}^{\infty} \alpha_k^2 < \infty$$

then $\lim_{k\to\infty} \|\nabla f(x_k)\| = 0$.

2.2 Newton's Method

Newton's method uses second-order information to achieve faster convergence.

Definition 2.2 (Newton's Method)**.** The Newton iteration is:

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

provided that $\nabla^2 f(x_k)$ is positive definite.

Theorem

Quadratic Convergence of Newton's Method
Suppose f is twice continuously differentiable, x^* is a local minimum with $\nabla^2 f(x^*) \succ 0$, and x_0 is sufficiently close to x^*. Then Newton's method converges quadratically to x^*:

$$\|x_{k+1} - x^*\| \leq C\|x_k - x^*\|^2$$

for some constant $C > 0$.

2.3 Quasi-Newton Methods

Quasi-Newton methods approximate the Hessian to reduce computational cost while maintaining superlinear convergence.

2.3.1 The BFGS Algorithm

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is the most popular quasi-Newton algorithm.

Definition 2.3 (BFGS Update)**.** Starting with $B_0 = I$ (or another positive definite matrix), the BFGS method updates the Hessian approximation:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

where $s_k = x_{k+1} - x_k$ and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.

Real-World Application

Machine Learning Applications
Optimization algorithms are essential in training machine learning models:

- **Linear Regression**: Minimize $\|Ax - b\|_2^2$
- **Logistic Regression**: Minimize cross-entropy loss
- **Neural Networks**: Backpropagation with gradient descent variants

The Adam optimizer, widely used in deep learning, combines ideas from gradient descent with momentum and adaptive step sizes.

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

def rosenbrock(x):
    """The Rosenbrock function - a classic test function"""
    return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

def rosenbrock_grad(x):
    """Gradient of the Rosenbrock function"""
    grad = np.zeros_like(x)
    grad[0] = -400 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    grad[1] = 200 * (x[1] - x[0]**2)
    return grad

def gradient_descent(f, grad_f, x0, alpha=0.001, max_iter=1000, tol=1e-6):
    """
    Gradient descent implementation
    """
    x = x0.copy()
    trajectory = [x.copy()]

    for i in range(max_iter):
        grad = grad_f(x)
        if np.linalg.norm(grad) < tol:
            break
        x = x - alpha * grad
        trajectory.append(x.copy())

    return x, np.array(trajectory)

# Example: Optimize the Rosenbrock function
x0 = np.array([-1.0, 1.0])
x_opt, trajectory = gradient_descent(rosenbrock, rosenbrock_grad, x0)

print(f"Starting point: {x0}")
print(f"Optimal point: {x_opt}")
print(f"Optimal value: {rosenbrock(x_opt)}")
print(f"Number of iterations: {len(trajectory)}")

# Compare with scipy's implementation
result = minimize(rosenbrock, x0, method='BFGS', jac=rosenbrock_grad)
print(f"SciPy BFGS result: {result.x}")
print(f"SciPy BFGS value: {result.fun}")
```

2.4 Convergence Analysis

2.4.1 Linear Convergence

Definition 2.4 (Linear Convergence)**.** A sequence $\{x_k\}$ converges linearly to x^* if there exist constants $C > 0$ and $0 < r < 1$ such that:

$$\|x_{k+1} - x^*\| \leq r\|x_k - x^*\|$$

The constant r is called the convergence rate.

Theorem

Convergence Rate of Gradient Descent
For a strongly convex function with Lipschitz continuous gradient, gradient descent with appropriate step size achieves linear convergence with rate:

$$r = \frac{\kappa - 1}{\kappa + 1}$$

where $\kappa = \frac{L}{\mu}$ is the condition number (L is the Lipschitz constant, μ is the strong convexity parameter).

2.4.2 Superlinear and Quadratic Convergence

Definition 2.5 (Superlinear Convergence)**.** A sequence $\{x_k\}$ converges superlinearly to x^* if:

$$\lim_{k \to \infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|} = 0$$

Exploration

Numerical Experiment
Implement and compare the convergence behavior of:

1. Gradient descent with fixed step size
2. Gradient descent with line search
3. Newton's method
4. BFGS method

Test on functions with different condition numbers and observe how the convergence rate depends on problem characteristics.

Python Code

```
def visualize_optimization_path(f, trajectory, title="Optimization Path"):
    """
    Visualize the optimization trajectory on a contour plot
    """
    # Create a grid for contour plot
```

```
    x_range = np.linspace(-2, 2, 100)
    y_range = np.linspace(-1, 3, 100)
    X, Y = np.meshgrid(x_range, y_range)
    Z = np.zeros_like(X)

    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i, j] = f([X[i, j], Y[i, j]])

    plt.figure(figsize=(10, 8))
    plt.contour(X, Y, Z, levels=50, alpha=0.6)
    plt.colorbar(label='Function Value')

    # Plot optimization path
    plt.plot(trajectory[:, 0], trajectory[:, 1], 'ro-',
             markersize=3, linewidth=1, alpha=0.8)
    plt.plot(trajectory[0, 0], trajectory[0, 1], 'go',
             markersize=10, label='Start')
    plt.plot(trajectory[-1, 0], trajectory[-1, 1], 'ro',
             markersize=10, label='End')

    plt.xlabel(' x')
    plt.ylabel(' x')
    plt.title(title)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Visualize the trajectory
visualize_optimization_path(rosenbrock, trajectory,
                            "Gradient Descent on Rosenbrock Function")
```

Exercise 2.1. Implement the Armijo backtracking line search and compare its performance with fixed step size gradient descent on the Rosenbrock function.

Exercise 2.2. Prove that for a quadratic function $f(x) = \frac{1}{2}x^T Qx + c^T x$ with $Q \succ 0$, Newton's method converges in exactly one iteration regardless of the starting point.

Exercise 2.3. Derive the L-BFGS algorithm (limited-memory BFGS) and implement it for large-scale problems where storing the full Hessian approximation is impractical.

Chapter 3

Advanced Optimization Methods

3.1 Conjugate Gradient Methods

Conjugate gradient methods combine the simplicity of gradient descent with the fast convergence properties of Newton's method for quadratic functions.

Definition 3.1 (Conjugate Directions)**.** A set of vectors $\{d_0, d_1, \ldots, d_{k-1}\}$ is said to be conjugate with respect to a positive definite matrix A if:

$$d_i^T A d_j = 0 \quad \text{for all } i \neq j$$

Definition 3.2 (Conjugate Gradient Algorithm)**.** The conjugate gradient method generates a sequence $\{x_k\}$ using:

$$x_{k+1} = x_k + \alpha_k d_k \tag{3.1}$$
$$d_{k+1} = -\nabla f(x_{k+1}) + \beta_k d_k \tag{3.2}$$

where α_k is determined by line search and β_k is chosen to ensure conjugacy.

3.1.1 Fletcher-Reeves and Polak-Ribière Formulas

Different choices of β_k lead to different conjugate gradient variants:

- **Fletcher-Reeves**: $\beta_k^{FR} = \frac{\|\nabla f(x_{k+1})\|^2}{\|\nabla f(x_k)\|^2}$
- **Polak-Ribière**: $\beta_k^{PR} = \frac{\nabla f(x_{k+1})^T(\nabla f(x_{k+1}) - \nabla f(x_k))}{\|\nabla f(x_k)\|^2}$
- **Hestenes-Stiefel**: $\beta_k^{HS} = \frac{\nabla f(x_{k+1})^T(\nabla f(x_{k+1}) - \nabla f(x_k))}{d_k^T(\nabla f(x_{k+1}) - \nabla f(x_k))}$

Theorem

Finite Termination for Quadratic Functions
For a quadratic function $f(x) = \frac{1}{2}x^T Ax - b^T x$ with $A \succ 0$, the conjugate gradient method terminates at the exact solution in at most n steps, where n is the dimension of the problem.

Python Code

```python
import numpy as np

def conjugate_gradient(A, b, x0, tol=1e-6, max_iter=None):
    """
    Conjugate Gradient method for solving Ax = b
    """
    n = len(b)
    if max_iter is None:
        max_iter = n

    x = x0.copy()
    r = A @ x - b  # residual
    d = -r         # search direction

    trajectory = [x.copy()]

    for k in range(max_iter):
        if np.linalg.norm(r) < tol:
            break

        # Line search for optimal step size
        Ad = A @ d
        alpha = (r.T @ r) / (d.T @ Ad)

        # Update solution
        x = x + alpha * d
        r_new = r + alpha * Ad

        # Compute beta using Fletcher-Reeves formula
        beta = (r_new.T @ r_new) / (r.T @ r)

        # Update search direction
        d = -r_new + beta * d
        r = r_new

        trajectory.append(x.copy())

    return x, np.array(trajectory)

def cg_nonlinear(f, grad_f, x0, tol=1e-6, max_iter=1000):
    """
    Nonlinear Conjugate Gradient with Polak-Ribière formula
    """
    x = x0.copy()
    g = grad_f(x)
    d = -g

    trajectory = [x.copy()]

    for k in range(max_iter):
        if np.linalg.norm(g) < tol:
```

```
            break

        # Line search (simple backtracking)
        alpha = 1.0
        c = 0.5
        rho = 0.5

        while f(x + alpha * d) > f(x) + c * alpha * (g.T @ d):
            alpha *= rho
            if alpha < 1e-10:
                break

        # Update
        x_new = x + alpha * d
        g_new = grad_f(x_new)

        # Polak-Ribière formula
        beta = max(0, (g_new.T @ (g_new - g)) / (g.T @ g))

        d = -g_new + beta * d

        x = x_new
        g = g_new
        trajectory.append(x.copy())

    return x, np.array(trajectory)

# Example: Solve a linear system using CG
A = np.array([[4, 1], [1, 3]])
b = np.array([1, 2])
x0 = np.array([0.0, 0.0])

x_solution, cg_trajectory = conjugate_gradient(A, b, x0)
print(f"CG Solution: {x_solution}")
print(f"True solution: {np.linalg.solve(A, b)}")
print(f"CG iterations: {len(cg_trajectory)}")
```

3.2 Trust Region Methods

Trust region methods provide a robust alternative to line search methods by constraining the step size based on a local model's trustworthiness.

Definition 3.3 (Trust Region Subproblem)**.** At each iteration, trust region methods solve:

$$\min_d m_k(d) = f(x_k) + \nabla f(x_k)^T d + \frac{1}{2} d^T B_k d$$

subject to $\|d\| \leq \Delta_k$, where $\Delta_k > 0$ is the trust region radius and B_k approximates $\nabla^2 f(x_k)$.

3.2.1 Trust Region Radius Update

The trust region radius is updated based on the agreement between the model and the actual function:

$$\rho_k = \frac{f(x_k) - f(x_k + d_k)}{m_k(0) - m_k(d_k)}$$

The radius update rule is:

$$\Delta_{k+1} = \begin{cases} \gamma_1 \Delta_k & \text{if } \rho_k < \eta_1 \\ \Delta_k & \text{if } \eta_1 \leq \rho_k < \eta_2 \\ \min(\gamma_2 \Delta_k, \Delta_{\max}) & \text{if } \rho_k \geq \eta_2 \end{cases}$$

where typically $0 < \gamma_1 < 1 < \gamma_2$, $0 < \eta_1 < \eta_2 < 1$.

Theorem

Global Convergence of Trust Region Methods
Under mild conditions on f and B_k, trust region methods have global convergence properties: every limit point of the sequence $\{x_k\}$ is a stationary point of f.

Python Code

```python
import numpy as np
from scipy.linalg import solve

def solve_trust_region_dogleg(g, B, Delta):
    """
    Solve trust region subproblem using dogleg method
    """
    # Cauchy point
    gTg = g.T @ g
    gTBg = g.T @ B @ g

    if gTBg <= 0:
        tau = 1.0
    else:
        tau = min(1.0, gTg**1.5 / (Delta * gTBg))

    p_cauchy = -tau * Delta * g / np.linalg.norm(g)

    # Newton point (if B is positive definite)
    try:
        p_newton = -solve(B, g)
        if np.linalg.norm(p_newton) <= Delta:
            return p_newton
    except np.linalg.LinAlgError:
        return p_cauchy

    # Dogleg path
    p_diff = p_newton - p_cauchy
    a = np.dot(p_diff, p_diff)
    b = 2 * np.dot(p_cauchy, p_diff)
    c = np.dot(p_cauchy, p_cauchy) - Delta**2

    discriminant = b**2 - 4*a*c
```

```
    if discriminant < 0:
        return p_cauchy

    tau2 = (-b + np.sqrt(discriminant)) / (2*a)
    return p_cauchy + tau2 * p_diff

def trust_region_dogleg(f, grad_f, hess_f, x0, Delta0=1.0,
                        tol=1e-6, max_iter=1000):
    """
    Trust Region method with Dogleg step
    """
    x = x0.copy()
    Delta = Delta0

    eta1, eta2 = 0.25, 0.75
    gamma1, gamma2 = 0.5, 2.0

    trajectory = [x.copy()]

    for k in range(max_iter):
        g = grad_f(x)
        if np.linalg.norm(g) < tol:
            break

        B = hess_f(x)

        # Solve trust region subproblem
        d = solve_trust_region_dogleg(g, B, Delta)

        # Evaluate reduction ratio
        actual_reduction = f(x) - f(x + d)
        predicted_reduction = -(g.T @ d + 0.5 * d.T @ B @ d)

        if predicted_reduction <= 0:
            rho = -1
        else:
            rho = actual_reduction / predicted_reduction

        # Update trust region radius
        if rho < eta1:
            Delta *= gamma1
        elif rho >= eta2:
            Delta = min(gamma2 * Delta, 10.0)

        # Accept or reject step
        if rho > eta1:
            x = x + d
            trajectory.append(x.copy())

    return x, np.array(trajectory)

# Example usage with Rosenbrock function
def rosenbrock_hess(x):
```

```
    """Hessian of Rosenbrock function"""
    H = np.zeros((2, 2))
    H[0, 0] = 1200*x[0]**2 - 400*x[1] + 2
    H[0, 1] = H[1, 0] = -400*x[0]
    H[1, 1] = 200
    return H

x0 = np.array([-1.0, 1.0])
x_opt, tr_trajectory = trust_region_dogleg(rosenbrock, rosenbrock_grad,
                                           rosenbrock_hess, x0)
print(f"Trust Region result: {x_opt}")
print(f"Function value: {rosenbrock(x_opt)}")
print(f"Iterations: {len(tr_trajectory)}")
```

3.3 Accelerated Gradient Methods

Accelerated methods use momentum to achieve better convergence rates than standard gradient descent.

3.3.1 Heavy Ball Method

The heavy ball method adds momentum to gradient descent:

Definition 3.4 (Heavy Ball Method)**.**

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \beta(x_k - x_{k-1})$$

where $\alpha > 0$ is the step size and $\beta \geq 0$ is the momentum parameter.

3.3.2 Nesterov's Accelerated Gradient

Nesterov's method evaluates the gradient at a "look-ahead" point:

Definition 3.5 (Nesterov's Accelerated Gradient)**.**

$$y_k = x_k + \frac{k-1}{k+2}(x_k - x_{k-1}) \tag{3.3}$$

$$x_{k+1} = y_k - \alpha \nabla f(y_k) \tag{3.4}$$

Theorem

Optimal Convergence Rate
For convex functions with Lipschitz continuous gradients, Nesterov's accelerated gradient achieves the optimal convergence rate of $O(1/k^2)$ compared to $O(1/k)$ for standard gradient descent.

Python Code

```
def nesterov_accelerated_gradient(f, grad_f, x0, alpha=0.01,
                                  max_iter=1000, tol=1e-6):
```

```
    """
    Nesterov's Accelerated Gradient Method
    """
    x = x0.copy()
    x_prev = x0.copy()

    trajectory = [x.copy()]

    for k in range(1, max_iter + 1):
        # Momentum coefficient
        theta = (k - 1) / (k + 2)

        # Look-ahead point
        y = x + theta * (x - x_prev)

        # Gradient at look-ahead point
        grad = grad_f(y)

        if np.linalg.norm(grad) < tol:
            break

        # Update
        x_new = y - alpha * grad

        x_prev = x
        x = x_new
        trajectory.append(x.copy())

    return x, np.array(trajectory)

def heavy_ball_method(f, grad_f, x0, alpha=0.01, beta=0.9,
                      max_iter=1000, tol=1e-6):
    """
    Heavy Ball Method with momentum
    """
    x = x0.copy()
    x_prev = x0.copy()

    trajectory = [x.copy()]

    for k in range(max_iter):
        grad = grad_f(x)

        if np.linalg.norm(grad) < tol:
            break

        # Heavy ball update
        x_new = x - alpha * grad + beta * (x - x_prev)

        x_prev = x
        x = x_new
        trajectory.append(x.copy())
```

```
    return x, np.array(trajectory)

# Compare methods on Rosenbrock function
x0 = np.array([-1.0, 1.0])

# Standard gradient descent
x_gd, traj_gd = gradient_descent(rosenbrock, rosenbrock_grad, x0, alpha=0.001)

# Nesterov accelerated gradient
x_nag, traj_nag = nesterov_accelerated_gradient(rosenbrock, rosenbrock_grad,
                                                x0, alpha=0.001)

# Heavy ball method
x_hb, traj_hb = heavy_ball_method(rosenbrock, rosenbrock_grad, x0,
                                  alpha=0.001, beta=0.5)

print("Method Comparison:")
print(f"Gradient Descent: {len(traj_gd)} iterations")
print(f"Nesterov AGD: {len(traj_nag)} iterations")
print(f"Heavy Ball: {len(traj_hb)} iterations")
```

3.4 Modern Adaptive Methods

Adaptive methods automatically adjust the learning rate for each parameter, making them particularly effective for machine learning applications.

3.4.1 AdaGrad

AdaGrad adapts the learning rate based on historical gradients:

Definition 3.6 (AdaGrad Algorithm)**.**

$$G_k = G_{k-1} + \nabla f(x_k) \odot \nabla f(x_k) \tag{3.5}$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_k + \epsilon}} \odot \nabla f(x_k) \tag{3.6}$$

where $\odot$ denotes element-wise multiplication and $\epsilon > 0$ is a small constant.

3.4.2 RMSprop

RMSprop uses exponential moving averages to prevent the learning rate from decreasing too rapidly:

Definition 3.7 (RMSprop Algorithm)**.**

$$v_k = \rho v_{k-1} + (1-\rho)\nabla f(x_k) \odot \nabla f(x_k) \tag{3.7}$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{v_k + \epsilon}} \odot \nabla f(x_k) \tag{3.8}$$

where $\rho \in (0, 1)$ is the decay rate.

3.4.3 Adam Optimizer

Adam combines the benefits of AdaGrad and RMSprop with momentum:

Definition 3.8 (Adam Algorithm)**.**

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla f(x_k) \tag{3.9}$$

$$v_k = \beta_2 v_{k-1} + (1-\beta_2)\nabla f(x_k) \odot \nabla f(x_k) \tag{3.10}$$

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k} \tag{3.11}$$

$$\hat{v}_k = \frac{v_k}{1-\beta_2^k} \tag{3.12}$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{\hat{v}_k + \epsilon}} \odot \hat{m}_k \tag{3.13}$$

where $\beta_1, \beta_2 \in (0,1)$ are exponential decay rates.

Python Code

```
def adam_optimizer(f, grad_f, x0, alpha=0.001, beta1=0.9, beta2=0.999,
                   epsilon=1e-8, max_iter=1000, tol=1e-6):
    """
    Adam Optimizer
    """
    x = x0.copy()
    m = np.zeros_like(x0)  # First moment
    v = np.zeros_like(x0)  # Second moment

    trajectory = [x.copy()]

    for k in range(1, max_iter + 1):
        grad = grad_f(x)

        if np.linalg.norm(grad) < tol:
            break

        # Update biased first moment estimate
        m = beta1 * m + (1 - beta1) * grad

        # Update biased second moment estimate
        v = beta2 * v + (1 - beta2) * (grad * grad)

        # Compute bias-corrected first moment estimate
        m_hat = m / (1 - beta1**k)

        # Compute bias-corrected second moment estimate
        v_hat = v / (1 - beta2**k)

        # Update parameters
        x = x - alpha * m_hat / (np.sqrt(v_hat) + epsilon)
        trajectory.append(x.copy())

    return x, np.array(trajectory)
```

```
def rmsprop_optimizer(f, grad_f, x0, alpha=0.001, rho=0.9,
                      epsilon=1e-8, max_iter=1000, tol=1e-6):
    """
    RMSprop Optimizer
    """
    x = x0.copy()
    v = np.zeros_like(x0)  # Moving average of squared gradients

    trajectory = [x.copy()]

    for k in range(max_iter):
        grad = grad_f(x)

        if np.linalg.norm(grad) < tol:
            break

        # Update moving average of squared gradients
        v = rho * v + (1 - rho) * (grad * grad)

        # Update parameters
        x = x - alpha * grad / (np.sqrt(v) + epsilon)
        trajectory.append(x.copy())

    return x, np.array(trajectory)

def adagrad_optimizer(f, grad_f, x0, alpha=0.01, epsilon=1e-8,
                      max_iter=1000, tol=1e-6):
    """
    AdaGrad Optimizer
    """
    x = x0.copy()
    G = np.zeros_like(x0)  # Sum of squared gradients

    trajectory = [x.copy()]

    for k in range(max_iter):
        grad = grad_f(x)

        if np.linalg.norm(grad) < tol:
            break

        # Accumulate squared gradients
        G = G + grad * grad

        # Update parameters
        x = x - alpha * grad / (np.sqrt(G) + epsilon)
        trajectory.append(x.copy())

    return x, np.array(trajectory)

# Compare adaptive methods
x0 = np.array([-1.0, 1.0])
```

```
methods = {
    'Adam': lambda: adam_optimizer(rosenbrock, rosenbrock_grad, x0),
    'RMSprop': lambda: rmsprop_optimizer(rosenbrock, rosenbrock_grad, x0),
    'AdaGrad': lambda: adagrad_optimizer(rosenbrock, rosenbrock_grad, x0)
}

print("Adaptive Methods Comparison:")
for name, method in methods.items():
    x_opt, trajectory = method()
    print(f"{name}: {len(trajectory)} iterations, "
          f"f(x) = {rosenbrock(x_opt):.6f}")
```

Real-World Application

Deep Learning Applications
Modern adaptive optimizers are crucial in training deep neural networks:

- **Computer Vision**: CNNs for image classification often use Adam or RMSprop
- **Natural Language Processing**: Transformer models typically employ Adam with learning rate schedules
- **Reinforcement Learning**: Policy gradient methods benefit from adaptive step sizes
- **Generative Models**: GANs and VAEs require careful optimizer tuning for stable training

The choice of optimizer can significantly impact training speed and final model performance.

Exploration

Optimizer Comparison Study
Design experiments to compare different optimization methods:

1. Test on various function types (convex, non-convex, ill-conditioned)
2. Analyze convergence rates and computational costs
3. Study sensitivity to hyperparameters
4. Investigate performance on high-dimensional problems
5. Compare wall-clock time vs. iteration count

This will provide insights into when to use each method in practice.

Exercise 3.1. Implement the AdaMax optimizer (a variant of Adam) and compare its performance with Adam on the Rosenbrock function.

Exercise 3.2. Prove that for strongly convex functions, Nesterov's accelerated gradient achieves

$O(1/k^2)$ convergence rate.

Exercise 3.3. Design a hybrid optimization algorithm that automatically switches between different methods based on convergence behavior.