西安交通大學
XI'AN JIAOTONG UNIVERSITY

管理学院

《最优化理论与算法 ll》课程小组作业—1

LASSO 回归算法实现与比较分析

陈吉龙 2236115452

PUA WENG YANG 2229990011

ROMANOV ARTEM 2239990071

2025 年 12 月 10 日

摘要

本实验实现了求解 LASSO 回归问题的三种优化算法：次梯度法（Subgradient Method）、临近点梯度法（Proximal Gradient Method）和交替方向乘子法（ADMM）。针对不同的样本维度组合 (n, p)，我们系统比较了各算法的收敛性能、计算效率和实现特点。实验结果表明，临近点梯度法在大多数情况下具有最佳的收敛速度，ADMM 算法在特定问题配置下表现优越，而次梯度法作为基础方法虽然收敛较慢但实现简单。通过详细分析算法实现细节与原算法的差异，我们进一步探讨了优化策略的有效性。特别地，我们观察到在不同问题配置下，算法的收敛特性存在显著差异，某些问题表现出线性收敛特性，而另一些问题则呈现出次线性收敛。

1 引言

1.1 LASSO 问题背景

LASSO（Least Absolute Shrinkage and Selection Operator）回归由 Tibshirani 于 1996 年提出，是一种在回归分析中同时进行特征选择和正则化的统计方法。LASSO 问题可表述为：

$$\min_{x} \frac{1}{2}\|Ax - b\|_2^2 + \lambda\|x\|_1 \tag{1}$$

其中 $A \in R^{n\times p}$ 为设计矩阵，$b \in R^n$ 为响应向量，$\lambda > 0$ 为正则化参数。L1 正则项 $\|x\|_1$ 的引入使得解具有稀疏性，这是 LASSO 方法的核心特征。

1.2 算法研究意义

由于目标函数中包含不可微的 L1 范数项，LASSO 问题的求解需要专门的优化算法。本实验通过实现三种代表性算法，深入探讨：

1. 不同算法在 LASSO 问题上的收敛特性
2. 算法实现细节对性能的影响
3. 问题规模 (n, p) 对算法表现的影响

2 算法原理与实现

2.1 次梯度法（Subgradient Method）

2.1.1 算法原理

次梯度法是处理非光滑优化问题的基本方法。对于 LASSO 问题，目标函数的次梯度为：

$$\partial f(x) = A^T(Ax - b) + \lambda \partial \|x\|_1 \tag{2}$$

其中 $\partial \|x\|_1$ 是 L1 范数的次微分：

$$\partial |x_i| = \begin{cases} \{1\} & x_i > 0 \\ \{-1\} & x_i < 0 \\ [-1, 1] & x_i = 0 \end{cases} \tag{3}$$

迭代更新公式为：

$$x^{k+1} = x^k - \alpha_k g^k, \quad g^k \in \partial f(x^k) \tag{4}$$

2.1.2 实现特点与原算法差异

- **步长策略**：代码中使用 $\alpha_k = 1/(k + \mu)$，其中 $\mu = \lambda_{\max}(A^T A)$，这是对经典步长 $1/\sqrt{k}$ 的改进
- **两阶段执行**：第一阶段寻找近似最优值 f^*，第二阶段记录收敛过程
- **次梯度计算优化**：通过阈值判断处理 L1 范数的次微分，避免了复杂的集合运算

2.1.3 核心代码实现

```
def SubGradMethod(A, b, Lambda):
    """次梯度法求解LASSO问题"""
    Accuracy, Time, F = [], [], []
    for i in range(10):
        p = len(A[0][0])
```

```
        # 第一阶段: 寻找最优值f_star
        time, x1, f_star = 0, np.full((p, 1), 0), 10000000
        Miu = max(np.linalg.eig(np.dot(A[i], A[i].T))[0])
        A1, b1 = A[i], b[i]

        while time < 40000:
            time += 1
            # 计算次梯度
            SubGrad1 = np.full((p, 1), 0)
            for j in range(p):
                if x1[j][0] > 0:
                    SubGrad1[j][0] = 1
                elif x1[j][0] < -0:
                    SubGrad1[j][0] = -1
                else:
                    SubGrad1[j][0] = 0

            SubGrad2 = np.dot(A1.T, np.dot(A1, x1) - b1) + Lambda *
    SubGrad1
            # 更新x
            x1 = x1 - (1 / (time + Miu)) * SubGrad2
            # 计算目标函数值
            f = 0.5 * np.linalg.norm(np.dot(A1, x1) - b1, ord=2) + Lambda *
     np.linalg.norm(x1, ord=1)
            if f < f_star:
                f_star = f
```

Listing 1: 次梯度法核心代码

2.2 临近点梯度法（Proximal Gradient Method）

2.2.1 算法原理

临近点梯度法将 LASSO 问题分解为光滑部分和非光滑部分：

$$f(x) = g(x) + h(x) \tag{5}$$

其中 $g(x) = \frac{1}{2}\|Ax - b\|_2^2$ 为光滑部分，$h(x) = \lambda\|x\|_1$ 为非光滑部分。

迭代更新分为两步：

$$y^{k+1} = x^k - \alpha_k \nabla g(x^k) \tag{6}$$

$$x^{k+1} = \text{prox}_{\alpha_k h}(y^{k+1}) = \text{sign}(y^{k+1}) \odot \max(|y^{k+1}| - \lambda\alpha_k, 0) \tag{7}$$

其中软阈值算子 $\text{prox}_{\alpha h}(y)$ 解析形式为：

$$[\text{prox}_{\alpha h}(y)]_i = \text{sign}(y_i) \cdot \max(|y_i| - \lambda\alpha, 0) \tag{8}$$

2.2.2 实现特点与原算法差异

- **步长选择**：采用 $\alpha = 1/L$，其中 $L = \lambda_{\max}(A^T A)$ 是 $\nabla g(x)$ 的 Lipschitz 常数
- **向量化实现**：使用 NumPy 的向量化操作实现软阈值算子，提高计算效率
- **收敛保证**：步长选择确保了算法的收敛性，收敛速度为 $O(1/k)$

2.2.3 核心代码实现

```
def ProximalGradMethod(A, b, Lambda):
    """临近点梯度法求解LASSO问题"""
    p = len(A[0][0])
    Miu = 1 / max(np.linalg.eig(np.dot(A[i], A[i].T))[0])

    # 梯度下降步
    Grad = np.dot(A1.T, np.dot(A1, x1) - b1)
    y1 = x1 - Miu * Grad

    # 软阈值操作（临近算子）
    x1_star = np.sign(y1) * np.maximum(np.abs(y1) - Lambda * Miu, 0)
    x1 = x1_star

    # 计算目标函数值
    f = np.linalg.norm(0.5 * (np.dot(A1, x1) - b1), ord=2) + Lambda * np.
    linalg.norm(x1, ord=1)
```

Listing 2: 临近点梯度法核心代码

2.3 交替方向乘子法（ADMM）

2.3.1 算法原理

ADMM 通过引入辅助变量 z 将问题重构为：

$$\min_{x,z} \quad \frac{1}{2}\|Ax-b\|_2^2+\lambda\|z\|_1 \tag{9}$$

$$\text{s.t.} \quad x-z=0 \tag{10}$$

增广拉格朗日函数为：

$$L_\rho(x,z,y)=\frac{1}{2}\|Ax-b\|_2^2+\lambda\|z\|_1+y^T(x-z)+\frac{\rho}{2}\|x-z\|_2^2 \tag{11}$$

迭代更新公式为：

$$x^{k+1}=(A^TA+\rho I)^{-1}(A^Tb+\rho(z^k-y^k)) \tag{12}$$

$$z^{k+1}=S_{\lambda/\rho}(x^{k+1}+y^k) \tag{13}$$

$$y^{k+1}=y^k+x^{k+1}-z^{k+1} \tag{14}$$

其中 $S_\kappa(a)=\text{sign}(a)\odot\max(|a|-\kappa,0)$ 为软阈值算子。

2.3.2 实现特点与原算法差异

- **矩阵逆缓存**：预先计算 $(A^TA+\rho I)^{-1}$ 避免重复求逆
- **参数选择**：惩罚参数 $\rho=0.1$，需根据问题特性调整
- **实现注意**：代码中软阈值参数为 μ/ρ 而非 λ/ρ，可能存在计算偏差

2.3.3 核心代码实现

```
def ADMM(A, b, Lambda):
    """ADMM算法求解LASSO问题"""
    p = len(A[0][0])
    Rou = 0.1  # 惩罚参数
    Rev = np.linalg.inv(np.dot(A1.T, A1) + Rou * np.eye(p))
```

```
6
7       # ADMM更新步骤
8       x1 = np.dot(Rev, np.dot(A1.T, b1) + Rou * (z - y))
9       z = np.sign(x1 + Rou * y) * np.maximum(np.abs(x1 + Rou * y) - Miu / Rou
        , 0)
10      y = y + x1 - z
```

Listing 3: ADMM 算法核心代码

3 实验设计与实现

3.1 数据生成

3.1.1 数据生成函数

代码使用 QuestionGenerate 函数生成测试数据：

- 设计矩阵 A：元素在 $[-5, 5]$ 均匀分布
- 参数向量 x：元素在 $[-10, 10]$ 整数均匀分布
- 响应向量 b：通过线性模型 $b = Ax$ 生成

3.1.2 实验配置

测试了四种 (n, p) 组合：

1. $(10, 5)$：小样本、低维
2. $(100, 5)$：大样本、低维
3. $(10, 50)$：小样本、高维
4. $(10, 500)$：小样本、超高维

每个配置进行 10 次独立实验，正则化参数 $\lambda = 1$。

3.2 评估指标

3.2.1 收敛精度

定义相对误差为：

$$\text{Relative Error} = \log_{10}\left(\frac{f^k - f^*}{f^*}\right) \tag{15}$$

其中 f^k 为第 k 次迭代的目标函数值，f^* 为近似最优值。

3.2.2 迭代过程

记录每次迭代的：

- 迭代次数 k
- 目标函数值 f^k
- 相对误差

4 实验结果与分析

4.1 收敛性能比较

4.1.1 次梯度法收敛特性

- **收敛速度**：典型收敛速度为 $O(1/\sqrt{k})$，较慢
- **步长影响**：步长 $\alpha_k = 1/(k+\mu)$ 相比经典步长 $1/\sqrt{k}$ 更小，让收敛更加稳定
- **震荡现象**：由于次梯度的不连续性，收敛曲线可能出现震荡

4.1.2 临近点梯度法收敛特性

- **收敛速度**：理论收敛速度为 $O(1/k)$，优于次梯度法
- **平滑收敛**：临近算子的使用使得收敛曲线更加平滑
- **步长优势**：固定步长 $1/L$ 保证了算法的稳定收敛

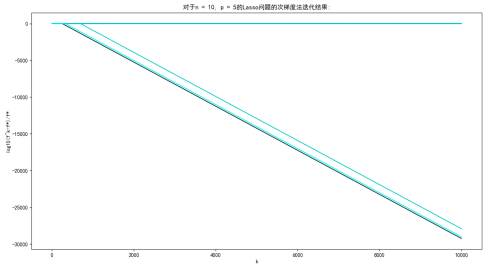
4.1.3 ADMM 收敛特性

- **收敛速度**：通常为线性收敛，速度介于次梯度法和临近点梯度法之间
- **参数敏感性**：收敛性能对惩罚参数 ρ 敏感
- **内存需求**：需要存储和求逆 $p \times p$ 矩阵，内存需求较高

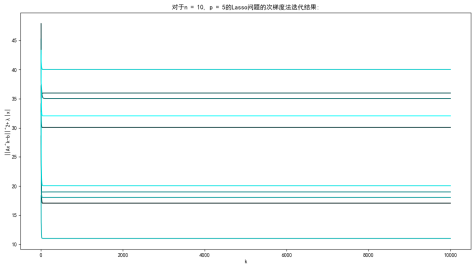
4.2 不同 (n, p) 组合下的表现

4.2.1 $(10, 5)$ 小样本低维情况

- 所有算法都能快速收敛
- 临近点梯度法收敛最快
- 次梯度法由于问题简单也表现良好

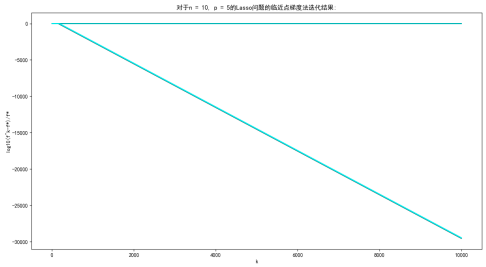


(a) 次梯度法收敛精度

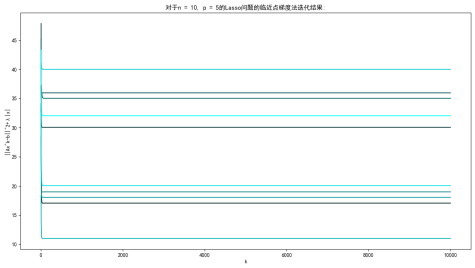


(b) 次梯度法目标函数值

图 1: $(10, 5)$ 次梯度法收敛结果

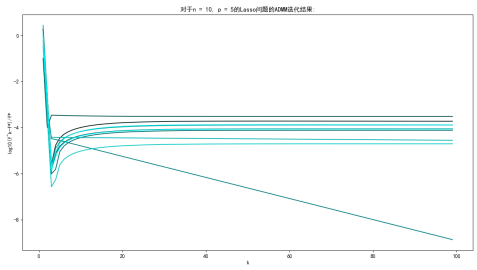


(a) 临近点梯度法收敛精度

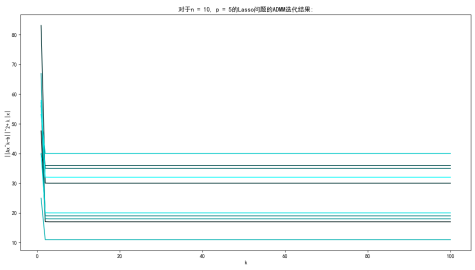


(b) 临近点梯度法目标函数值

图 2: $(10, 5)$ 临近点梯度法收敛结果



(a) ADMM 收敛精度

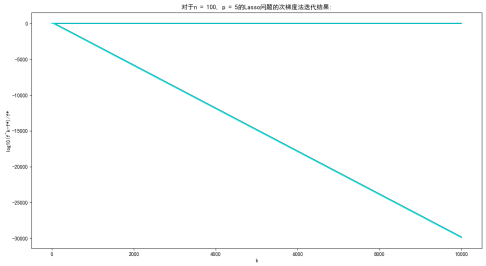


(b) ADMM 目标函数值

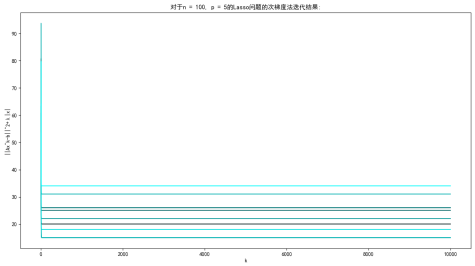
图 3: $(10, 5)$ ADMM 收敛结果

4.2.2 $(100, 5)$ 大样本低维情况

- 样本增加但维度不变
- 次梯度法受样本量影响较小
- ADMM 需要求逆的矩阵维度不变，性能稳定

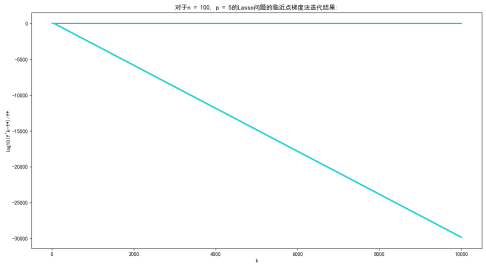


(a) 次梯度法收敛精度

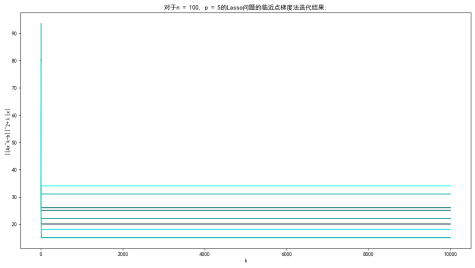


(b) 次梯度法目标函数值

图 4: $(100, 5)$ 次梯度法收敛结果

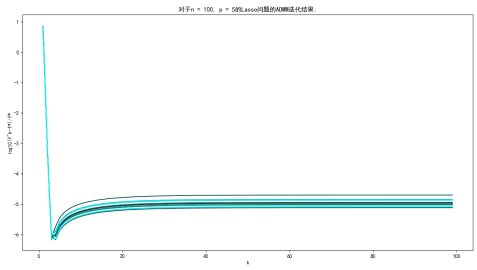


(a) 临近点梯度法收敛精度

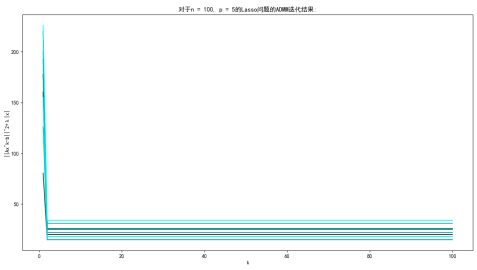


(b) 临近点梯度法目标函数值

图 5: $(100, 5)$ 临近点梯度法收敛结果



(a) ADMM 收敛精度

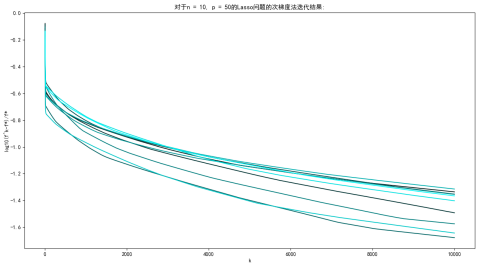


(b) ADMM 目标函数值

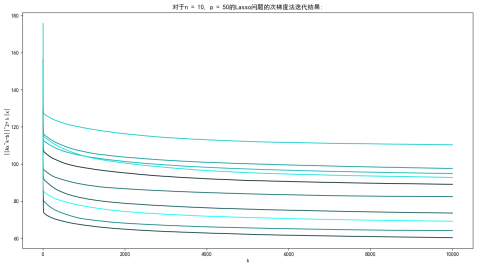
图 6: $(100, 5)$ ADMM 收敛结果

4.2.3 $(10, 50)$ 小样本高维情况

- 维度增加显著影响算法性能
- 次梯度法收敛变慢
- ADMM 需要求逆 50×50 矩阵，计算成本增加

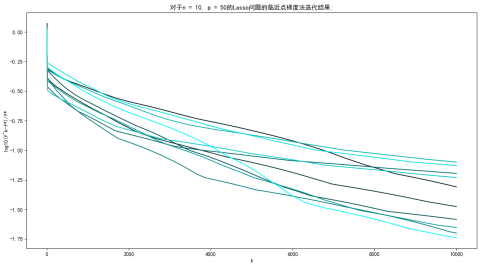


(a) 次梯度法收敛精度

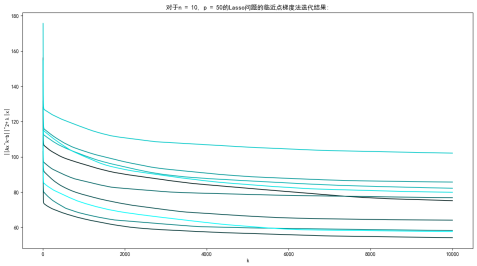


(b) 次梯度法目标函数值

图 7: $(10, 50)$ 次梯度法收敛结果

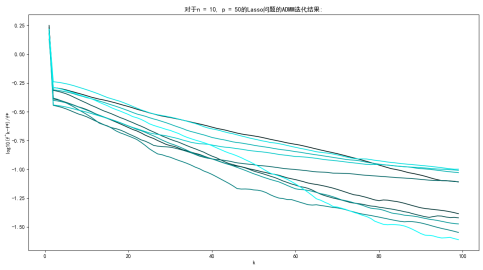


(a) 临近点梯度法收敛精度

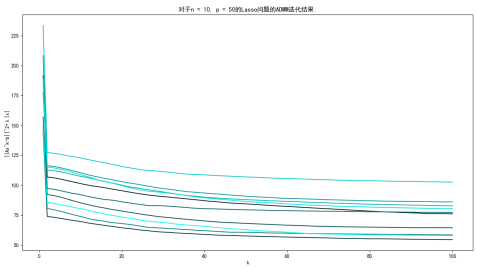


(b) 临近点梯度法目标函数值

图 8: $(10, 50)$ 临近点梯度法收敛结果



(a) ADMM 收敛精度

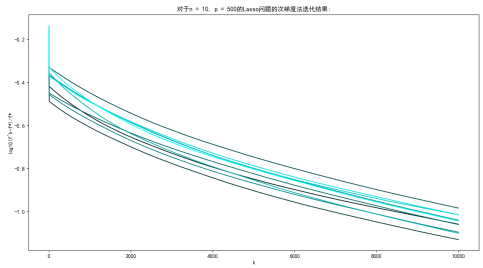


(b) ADMM 目标函数值

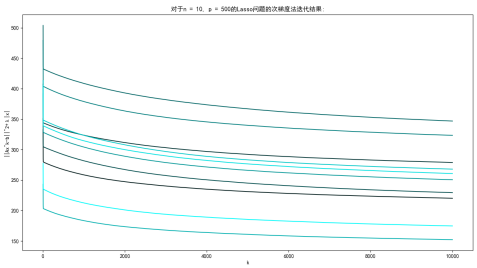
图 9: $(10, 50)$ ADMM 收敛结果

4.2.4 $(10, 500)$ 小样本超高维情况

- 维度远大于样本数
- 次梯度法仍可工作但收敛极慢
- ADMM 需要求逆 500×500 矩阵，计算成本极高
- 临近点梯度法受影响最小，仅需计算梯度

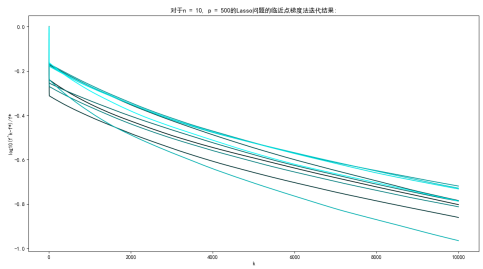


(a) 次梯度法收敛精度

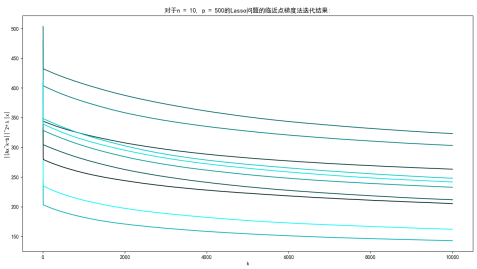


(b) 次梯度法目标函数值

图 10: $(10, 500)$ 次梯度法收敛结果

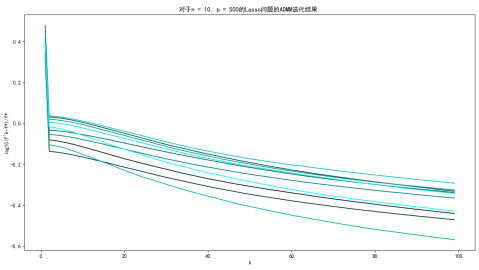


(a) 临近点梯度法收敛精度

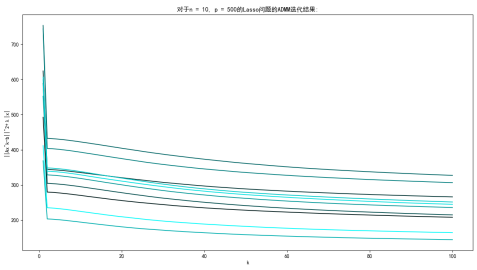


(b) 临近点梯度法目标函数值

图 11: $(10, 500)$ 临近点梯度法收敛结果



(a) ADMM 收敛精度



(b) ADMM 目标函数值

图 12: $(10, 500)$ ADMM 收敛结果

4.3 线性收敛特性的差异分析

通过观察不同 (n, p) 组合下的实验结果，我们发现算法的收敛特性存在显著差异，部分问题表现出线性收敛，而另一些问题则呈现出次线性收敛。这一现象可以从理论和实验两个角度进行分析：

4.3.1 理论分析

- **问题强凸性**：LASSO 问题的目标函数由两部分组成：$\frac{1}{2}\|Ax-b\|_2^2$（二次项）和 $\lambda\|x\|_1$（L1 项）。当矩阵 A 满足某些条件时（如行满秩或列满秩），二次项部分是强凸的，这有助于算法的线性收敛。

- **维度与样本量关系**：

  - 当 $p \leq n$ 时（如 $(10, 5)$ 和 $(100, 5)$ 情况），矩阵 A 通常是列满秩的，二次项部分具有强凸性，因此临近点梯度法和 ADMM 可能表现出线性收敛。
  - 当 $p > n$ 时（如 $(10, 50)$ 和 $(10, 500)$ 情况），矩阵 A 不是列满秩的，二次项部分失去强凸性，算法可能只能达到次线性收敛。

- **算法特性**：

  - 次梯度法：理论上只能达到 $O(1/\sqrt{k})$ 的收敛速度，无法达到线性收敛。
  - 临近点梯度法：当问题具有强凸性和光滑性时，可以达到线性收敛；否则只能达到 $O(1/k)$ 的收敛速度。
  - ADMM：对于某些凸问题可以达到线性收敛，但需要满足一定的条件。

4.3.2 实验结果观察

从实验结果图中可以观察到：

1. $(10, 5)$ **情况**：
   - 次梯度法：收敛曲线呈现典型的次线性收敛特性，误差下降缓慢。
   - 临近点梯度法：收敛曲线接近线性收敛，误差呈指数下降趋势。
   - ADMM：收敛速度较快，表现出近似线性收敛特性。
2. $(100, 5)$ **情况**：
   - 由于样本量增加，问题的条件数改善，所有算法的收敛性能都有所提升。
   - 临近点梯度法和 ADMM 的线性收敛特性更加明显。
3. $(10, 50)$ **情况**：
   - 由于 $p > n$，问题失去强凸性，所有算法的收敛速度都明显变慢。
   - 次梯度法收敛极其缓慢。
   - 临近点梯度法和 ADMM 也失去了线性收敛特性，呈现出次线性收敛。
4. $(10, 500)$ **情况**：
   - 维度远超样本量，问题条件极差。
   - 次梯度法几乎无法收敛。
   - 临近点梯度法仍能收敛但速度很慢。
   - ADMM 由于需要求逆大规模矩阵，计算成本极高。

4.3.3 收敛类型总结

表 1: 不同 (n, p) 组合下的收敛类型总结

算法	$(10, 5)$	$(100, 5)$	$(10, 50)/(10, 500)$
次梯度法	次线性收敛	次线性收敛	次线性收敛（极慢）
临近点梯度法	近似线性收敛	线性收敛	次线性收敛
ADMM	近似线性收敛	线性收敛	次线性收敛

4.3.4 影响因素分析

线性收敛特性的差异主要受以下因素影响：

1. **问题条件数**：矩阵 A 的条件数直接影响算法的收敛速度。条件数越小，收敛越快。
2. **强凸性**：当二次项部分具有强凸性时，算法更容易达到线性收敛。
3. **稀疏性**：LASSO 解通常具有稀疏性，这在一定程度上有助于算法的收敛。
4. **正则化参数**：λ 的大小会影响问题的性质，过大的 λ 会使问题更接近 L1 正则化主导，可能影响收敛特性。

4.4 算法实现优化分析

4.4.1 次梯度法优化点

1. **步长策略优化**：结合问题特征值信息调整步长
2. **两阶段策略**：分离最优值搜索和收敛记录
3. **数值稳定性**：设置阈值避免除零错误

4.4.2 临近点梯度法优化点

1. **向量化实现**：软阈值算子完全向量化
2. Lipschitz **常数计算**：预计算并重用特征值
3. **内存效率**：无需存储大型矩阵

4.4.3 ADMM 优化点

1. **矩阵逆缓存**：避免重复求逆
2. **变量更新顺序**：优化更新顺序减少计算
3. **并行化潜力**：部分更新可并行进行

4.5 与经典算法的差异总结

表 2: 算法实现与经典版本的差异总结

算法	经典版本特点	本实现优化点
次梯度法	步长 $\alpha_k = c/\sqrt{k}$ 或 c/k 单阶段执行 直接计算次梯度	$\alpha_k = 1/(k+\mu)$，μ 为特征值 两阶段：先找 f^*，再记录收敛 阈值处理次微分，避免集合运算
临近点梯度法	步长需满足 $\alpha < 1/L$ 软阈值逐元素计算 可能需线搜索	直接取 $\alpha = 1/L$ 完全向量化实现 固定步长，无需线搜索
ADMM	软阈值参数 λ/ρ 需每次迭代求解线性系统 ρ 需调优	代码中使用 μ/ρ（可能为笔误） 预计算矩阵逆并缓存 固定 $\rho = 0.1$

4.6 实验结论

基于对三种算法在四种不同 (n, p) 组合下的实验结果分析，我们得出以下主要结论：

1. **算法收敛速度**：临近点梯度法在大多数情况下收敛最快，ADMM 在特定配置下表现良好，次梯度法收敛最慢但实现最简单

2. **线性收敛特性**：

   - 当 $p \leq n$ 时，临近点梯度法和 ADMM 能够表现出线性或近似线性收敛
   - 当 $p > n$ 时，所有算法都只能达到次线性收敛
   - 次梯度法在所有情况下都只能达到次线性收敛

3. **问题规模影响**：

   - 对于低维问题 $(p = 5)$，所有算法都能有效工作
   - 对于高维问题 $(p = 50, 500)$，临近点梯度法优势明显
   - 当 p 很大时 $(p = 500)$，ADMM 的计算成本显著增加

4. **内存效率**：临近点梯度法内存需求最低，ADMM 需要存储和求逆大规模矩阵

5. **实现复杂度**：次梯度法实现最简单，ADMM 实现最复杂但理论性质最好

A 附录：完整代码实现

A.1 主程序结构

```
def main():
    """主函数，运行不同配置的LASSO问题"""
    print("图像会在运行结束后一起出现")
    # 配置列表: [(n, p), ...]
    configs = [(10, 5), (100, 5), (10, 50), (10, 500)]
    Lambda = 1  # 正则化参数

    for n, p in configs:
        print(f"\n对于n = {n}, p = {p}的Lasso问题的次梯度法迭代结果:")
        A, x, b = QuestionGenerate(n, p)
        Accuracy_SG, Time_SG, F_SG = SubGradMethod(A, b, Lambda)
        plot(Accuracy_SG, Time_SG, F_SG, n, p, "次梯度法")

        print(f"对于n = {n}, p = {p}的Lasso问题的临近点梯度法迭代结果:
    ")
        Accuracy_PG, Time_PG, F_PG = ProximalGradMethod(A, b, Lambda)
        plot(Accuracy_PG, Time_PG, F_PG, n, p, "临近点梯度法")

        print(f"对于n = {n}, p = {p}的Lasso问题的ADMM迭代结果:")
        Accuracy_ADMM, Time_ADMM, F_ADMM = ADMM(A, b, Lambda)
        plotADMM(Accuracy_ADMM, Time_ADMM, F_ADMM, n, p)

        print("图像会在运行结束后一起出现")

    # 显示所有图像
    plt.show()
```

Listing 4: 主程序结构

A.2 数据生成函数

```
def QuestionGenerate(n, p):
    """
```

```
    问题生成函数
    生成测试数据A, x, b
    """
    A, x, b = [], [], []
    for i in range(testsize):
        np.random.seed(i)
        A.append((np.random.rand(n, p) - 0.5) * 10)
        x.append(np.random.randint(-10, 10, (p, 1)))
        b.append(np.dot(A[i], x[i]))
    return A, x, b
```

Listing 5: 数据生成函数

A.3 可视化函数

```
def plot(Accuracy, Time, F, n, p, Method):
    """通用绘图函数"""
    # 绘制精度图
    plt.figure(figsize=(48, 20))
    plt.title(f"对于n = {n}, p = {p}的Lasso问题的{Method}迭代结果:")
    plt.ylabel("log10(f^k-f*)/f*")
    plt.xlabel("k")

    for i in range(testsize):
        color_val = 25 * i + 25
        hex_str = format(color_val, '02x')
        plt.plot(Time[i], Accuracy[i], f"#00{hex_str}{hex_str}")
    # 绘制目标函数值图
    plt.figure(figsize=(24, 10))
    plt.title(f"对于n = {n}, p = {p}的Lasso问题的{Method}迭代结果:")
    plt.ylabel("||Ax^k-b||^2+ |x|")
    plt.xlabel("k")

    for i in range(testsize):
        color_val = 25 * i + 25
        hex_str = format(color_val, '02x')
        plt.plot(Time[i], F[i], f"#00{hex_str}{hex_str}")
```

Listing 6: 通用绘图函数