大作业：优化算法实验

【专业】计算机科学与技术

【姓名】谢宇桐

【学号】22336259

一、问题描述

考虑一个 10 节点的分布式系统。节点 i 有线性测量 $\boldsymbol{b}_i = A_i\boldsymbol{x} + \boldsymbol{e}_i$，其中 $\boldsymbol{b}_i$ 为 5 维的测量值，A_i 为 5×200 维的测量矩阵，$\boldsymbol{x}$ 为 200 维的未知稀疏向量且稀疏度为 5，e_i 为 5 维的测量噪声。从所有 $\boldsymbol{b}_i$ 与 A_i 中恢复 $\boldsymbol{x}$ 的一范数正则化最小二乘模型如下：

$$\min_{\boldsymbol{x}} \quad \frac{1}{2}\|A_1\boldsymbol{x} - b_1\|_2^2 + \cdots + \frac{1}{2}\|A_{10}\boldsymbol{x} - \boldsymbol{b}_{10}\|_2^2 + \lambda\|\boldsymbol{x}\|_1,$$

其中 $\lambda > 0$ 为正则化参数。请设计下述算法求解该问题：

1. **邻近梯度法；**
2. **交替方向乘子法；**
3. **次梯度法；**

在实验中，设 $\boldsymbol{x}$ 的真值中的非零元素服从均值为 0 方差为 1 的高斯分布，A_i 中的元素服从均值为 0 方差为 1 的高斯分布，$\boldsymbol{e}_i$ 中的元素服从均值为 0 方差为 0.1 的高斯分布。对于每种算法，请给出每步计算结果与真值的距离以及每步计算结果与最优解的距离。此外，请讨论正则化参数 λ 对计算结果的影响。

在这里，我使用python实现算法。

二、算法设计

1. f0 函数

f0函数用于计算目标函数的值，它是一个带有L1正则化的最小化问题的目标函数。具体定义如下：

$$f(x) = \frac{1}{2}\sum_{i=1}^{n}\|A_i x - b_i\|_2^2 + \lambda\|x\|_1$$

- **第一项**：$\frac{1}{2}\sum_{i=1}^{n}\|A_i x - b_i\|_2^2$，表示数据拟合项，衡量模型预测值 $A_i x$ 与真实值 b_i 之间的平方误差。
- **第二项**：$\lambda\|x\|_1$，表示L1正则化项，用于引入稀疏性，其中 λ 是正则化参数 。

代码实现如下：

```python
def f0(A, b, x, lamda):
    """
    计算目标函数值：
    f(x) = 0.5 * sum( ||A[i] @ x - b[i]||_2^2 ) + lamda * ||x||_1
    """
    residuals = A @ x - b  # 计算残差矩阵，形状为 (n, d1)
    data_term = 0.5 * np.sum(residuals ** 2)  # 计算平方误差的总和
    regularization_term = lamda * np.linalg.norm(x, ord=1)  # 计算L1正则化项
    return data_term + regularization_term
```

2. 软门限算法（Soft Thresholding）

软门限算法是一种用于处理L1正则化问题的工具，常用于邻近点梯度法中。

对于每个元素 xi，软门限操作定义为：

$$\mathcal{T}_\omega(x_i) = \begin{cases} x_i - \omega, & \text{if } x_i > \omega \\ 0, & \text{if } |x_i| \leq \omega \\ x_i + \omega, & \text{if } x_i < -\omega \end{cases}$$

其中，ω 是阈值参数。

如果 |xi|>ω，则 xi的值会被减去或加上 ω，具体取决于 xi的符号。

如果 |xi|≤ω，则 xi被设置为0。这有助于实现稀疏性，即减少非零元素的数量。

```python
def soft_thresholding(x, threshold):
    """
    软门限函数
    :param x: 输入向量
    :param threshold: 阈值
    :return: 软门限处理后的向量
    """
    return np.sign(x) * np.maximum(np.abs(x) - threshold, 0)
```

3. 邻近点梯度法（Proximal Gradient Method）

邻近点梯度法是一种用于求解带有正则化项的优化问题的迭代算法。它结合了梯度下降法和软门限操作，特别适用于L1正则化问题。

算法原理：

- 梯度计算：计算目标函数的梯度 ∇f(x)。
- 梯度下降：更新 x 为 x−α∇f(x)，其中 α 是学习率。
- 软门限操作：对更新后的 x 应用软门限操作，以处理L1正则化项。
- 收敛判断：当 ‖x new − x old‖2 <tol 时停止迭代。

```
```

```python
def proximal_gradient_method(A, b, lamda, alpha=0.0001, max_iter=5000,
tol=1e-5, if_draw=True):
    start_time = time.time()

    n, d1, d2 = A.shape
    x = np.zeros(d2)  # 初始解
    iterates = []  # 记录每步的解

    for _ in range(max_iter):
        x_old = x.copy()

        # 计算梯度
        gradient = np.sum([A[i].T @ (A[i] @ x - b[i]) for i in range(n)],
axis=0)

        # 软门限法算argmin
        x = soft_thresholding(x - alpha * gradient, lamda * alpha)

        iterates.append(x)  # 记录解

        # 判断收敛
        if np.linalg.norm(x - x_old, ord=2) < tol:
            break

    end_time = time.time()
    diff_time = end_time - start_time

    # 计算每步解与真实解之间以及最优解之间的距离
    distances_to_true = [np.linalg.norm(iterate - x_true, ord=2) for iterate
in iterates]
    distances_to_opt = [np.linalg.norm(iterate - x, ord=2) for iterate in
iterates]

    if if_draw:
        plt.plot(distances_to_true, label='distance to true')
        plt.plot(distances_to_opt, label='distance to optimal')
        plt.title('proximal gradient method')
        plt.xlabel('iteration')
        plt.ylabel('distance')
        plt.grid()
        plt.legend()
        plt.show()

    print(f'proximal gradient using time(alpha={alpha}, lambda={lamda}):
{diff_time} s')
    print(f'distance of proximal gradient x_opt and x_true(alpha={alpha},
lambda={lamda}): {np.linalg.norm(x - x_true)}')

    return x, distances_to_true, distances_to_opt
```

4. 交替方向乘子法（ADMM）

原理：

ADMM是一种用于求解带有约束的优化问题的算法。它通过引入辅助变量和对偶变量，将原问题分解为更简单的子问题，交替更新这些变量。

在每次迭代中：

- 更新 x：求解线性方程组。
- 更新辅助变量 y：使用软阈值操作。
- 更新对偶变量 v。

```
def admm(A, b, lamda, C=1, max_iter=1000, tol=1e-5, if_draw=True):
    start_time = time.time()

    n, d1, d2 = A.shape
    x = np.zeros(d2)  # 初始解
    y = np.zeros(d2)  # 辅助变量
    v = np.zeros(d2)  # 对偶变量

    iterates = []  # 记录每步的解

    # 预计算矩阵和向量
    A_T_A = np.sum([A[i].T @ A[i] for i in range(n)], axis=0)
    A_T_b = np.sum([A[i].T @ b[i] for i in range(n)], axis=0)
    inv_matrix = np.linalg.inv(A_T_A + C * np.eye(d2))

    for _ in range(max_iter):
        x_old = x.copy()

        # 更新 x
        x = inv_matrix @ (A_T_b + C * y - v)

        # 更新 y
        y = soft_thresholding(x + v / C, lamda / C)

        # 更新 v
        v += C * (x - y)

        iterates.append(x)

        # 判断收敛
        if np.linalg.norm(x - x_old, ord=2) < tol:
            break

    end_time = time.time()
    diff_time = end_time - start_time

    # 计算每步解与真实解之间以及最优解之间的距离
    distances_to_true = [np.linalg.norm(iterate - x_true, ord=2) for iterate
in iterates]
    distances_to_opt = [np.linalg.norm(iterate - x, ord=2) for iterate in
iterates]

    if if_draw:
```

```
        plt.plot(distances_to_true, label='distance to true')
        plt.plot(distances_to_opt, label='distance to optimal')
        plt.title('alternating direction method of multipliers')
        plt.xlabel('iteration')
        plt.ylabel('distance')
        plt.grid()
        plt.legend()
        plt.show()

    print(f"ADMM using time (C={C}, lambda={lamda}): {diff_time} s")
    print(f"Distance of ADMM x_opt and x_true (C={C}, lambda={lamda}):
{np.linalg.norm(x - x_true)}")

    return x, distances_to_true, distances_to_opt
```

其中：

- 预计算矩阵：`inv_matrix = np.linalg.inv(A_T_A + C * np.eye(d2))`，这一步可以大大提高计算时间
- 更新 x：`x = inv_matrix @ (A_T_b + C * y - v)`
- 更新 y：`y = soft_thresholding(x + v / C, lamda / C)`
- 更新 v：`v += C * (x - y)`

5. 次梯度法（Subgradient Method）

原理：

次梯度法是一种用于求解非光滑优化问题的迭代算法。它通过计算目标函数的次梯度来更新变量。

在每次迭代中，计算次梯度 g，并更新 x。

```
def subgradient(A, b, lamda, alpha=0.0001, max_iter=5000, tol=1e-5,
if_draw=False):
    start_time = time.time()

    n, d1, d2 = A.shape
    x = np.zeros(d2)  # 初始解
    iterates = []  # 记录每步的解

    # 预计算梯度相关的矩阵和向量
    A_T_A = np.sum([A[i].T @ A[i] for i in range(n)], axis=0)
    A_T_b = np.sum([A[i].T @ b[i] for i in range(n)], axis=0)

    for _ in range(max_iter):
        x_old = x.copy()

        # 计算次梯度
        gradient = A_T_A @ x - A_T_b
        subgrad = np.sign(x)
        subgrad[x == 0] = np.random.uniform(-1, 1, size=np.sum(x == 0))  # 对
于 x=0 的情况，随机选择 [-1, 1] 之间的值
        g = gradient + lamda * subgrad

        # 更新 x
```

```
        x = x - alpha * g

        iterates.append(x)

        # 判断收敛
        if np.linalg.norm(x - x_old, ord=2) < tol:
            break

    end_time = time.time()
    diff_time = end_time - start_time

    # 计算每步解与真实解之间以及最优解之间的距离
    distances_to_true = [np.linalg.norm(iterate - x_true, ord=2) for iterate
in iterates]
    distances_to_opt = [np.linalg.norm(iterate - x, ord=2) for iterate in
iterates]

    if if_draw:
        plt.figure()
        plt.plot(distances_to_true, label='distance to true')
        plt.plot(distances_to_opt, label='distance to optimal')
        plt.title('subgradient')
        plt.xlabel('iteration')
        plt.ylabel('distance')
        plt.grid()
        plt.legend()
        plt.show()

    print(f"Subgradient using time (alpha={alpha}, lambda={lamda}):
{diff_time} s")
    print(f"Distance of Subgradient x_opt and x_true (alpha={alpha}, lambda=
{lamda}): {np.linalg.norm(x - x_true)}")

    return x, distances_to_true, distances_to_opt
```

6. 正则化调整参数

```
def adjust_lamda(A, b, lamdas, method):
    '''
    调整正则化参数，lamdas是参数列表，method决定用哪个优化算法，同时作为绘制图形的
suptitle，
    method只能取值'proximal gradient', 'admm' 或'subgradient'
    '''
    fig, axes = plt.subplots(int(sqrt(len(lamdas))), ceil(len(lamdas) / 2),
figsize=(12, 8))  # 创建多个子图
    # 画每一个参数值对应的子图
    for i, lamda in enumerate(lamdas):
        if method == 'proximal gradient':
            r1 = proximal_gradient_method(A, b, lamda, if_draw=False)
        elif method == 'admm':
            r1 = admm(A, b, lamda, if_draw=False)
        elif method == 'subgradient':
            r1 = subgradient(A, b, lamda, if_draw=False)
```

```
        row, col = divmod(i, ceil(len(lamdas) / 2))  # 计算子图位置
        axes[row, col].plot(r1[1], label='distance to true')
        axes[row, col].plot(r1[2], label='distance to opt')
        axes[row, col].set_title(r"$\lambda = $" + f"{lamda}")
        axes[row, col].set_xlabel('iteration')
        axes[row, col].set_ylabel('distance')
        axes[row, col].grid()
        axes[row, col].legend()

    plt.suptitle(method)
    plt.tight_layout()
    plt.show()
```

7. **随机生成数据**

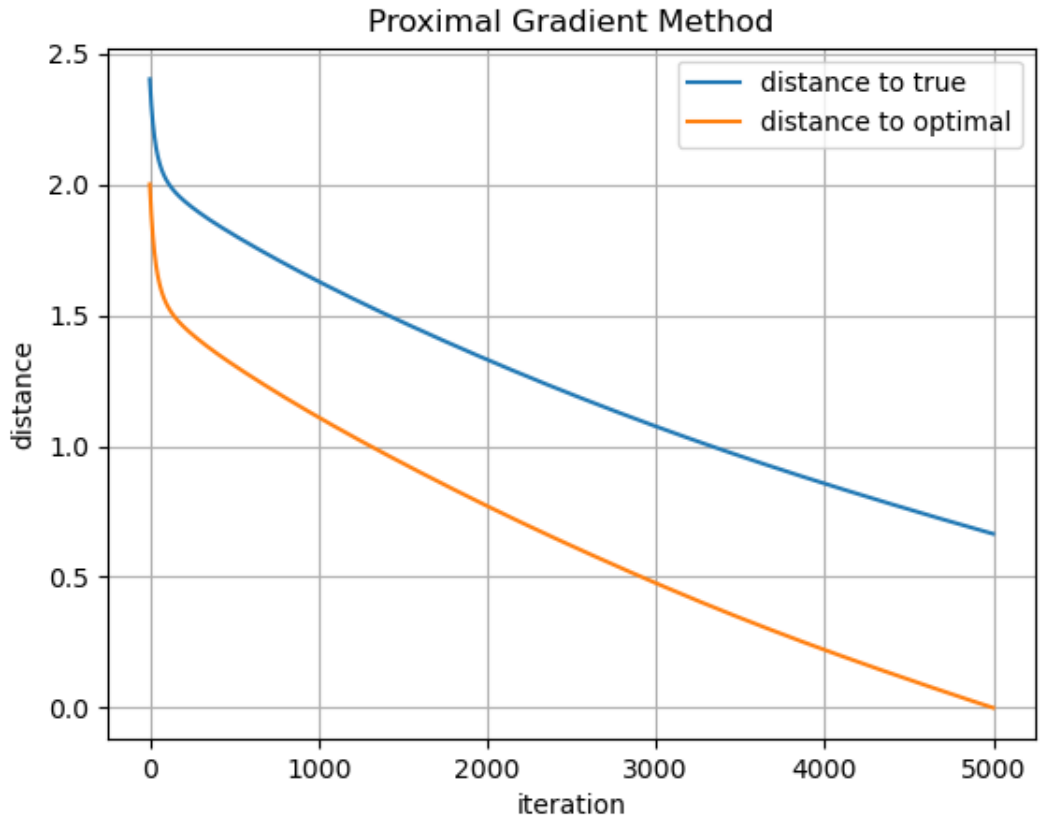
先随机生成已知值，包括Ai 、真值x 、bi 。代码如下：

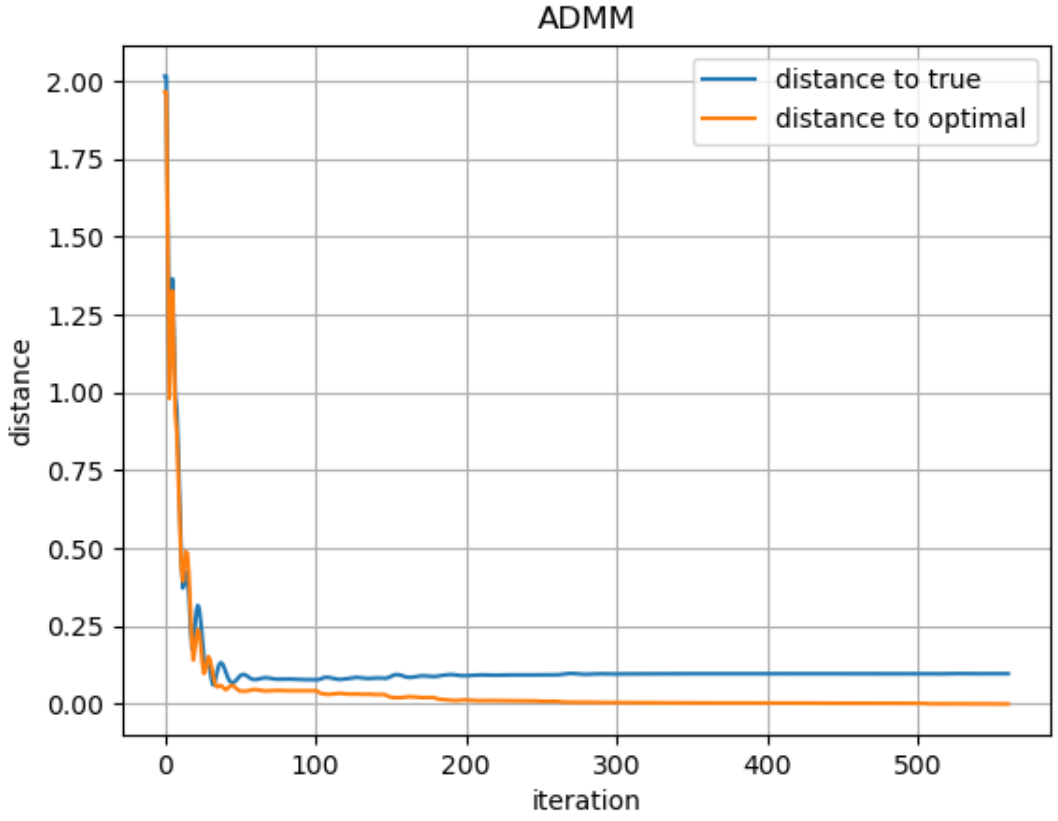
```
np.random.seed(0)
num = 10
d1 = 5
d2 = 200
lamda = 1

# 随机生成矩阵A、x的真值、和向量b
A = np.array([np.random.normal(0, 1, (d1, d2)) for _ in range(num)])
x_true = np.zeros(d2)
# 随机选择5个位置非0，其他位置为0
nonzero_indices = np.random.choice(d2, 5, replace=False)
x_true[nonzero_indices] = np.random.normal(0, 1, d1)
b = np.array([A[i].dot(x_true) + np.random.normal(0, 0.1, d1) for i in
range(num)])

# 三种算法求解
x_opt1 = proximal_gradient_method(A, b, lamda)[0]
x_opt2 = admm(A, b, lamda)[0]
x_opt3 = subgradient(A, b, lamda)[0]
```

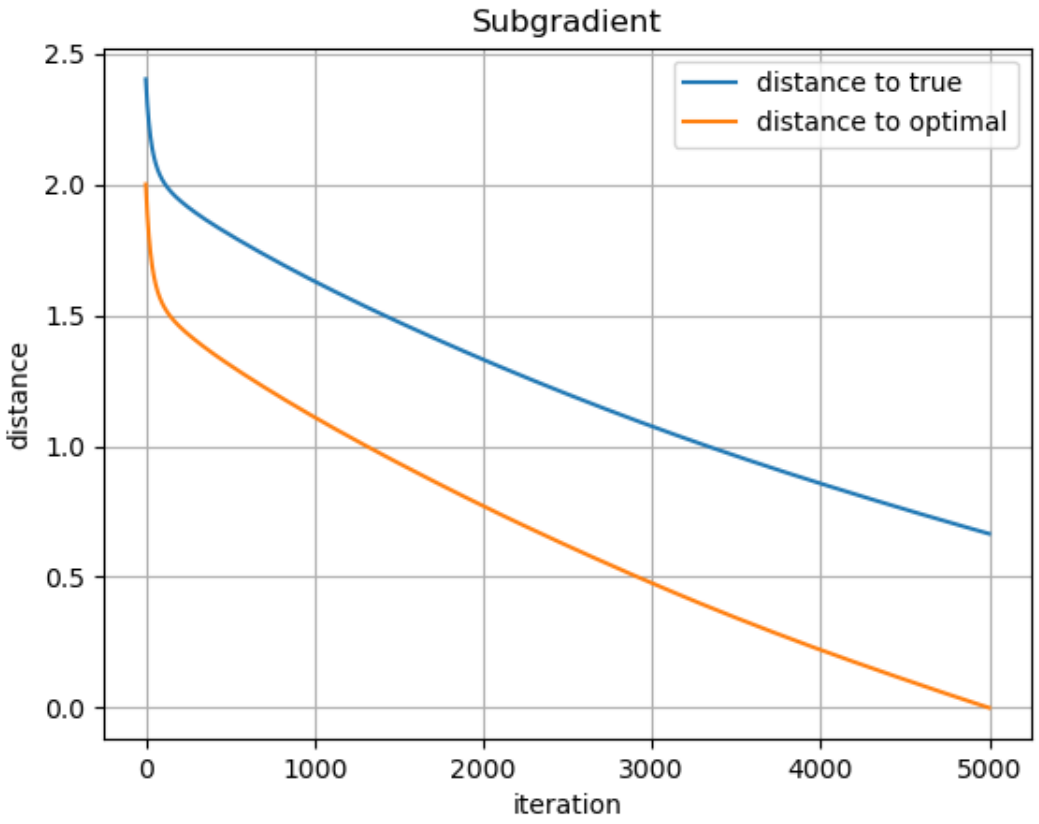
三、数值实验与结果分析

不同方法对比：

我们令lamda=1来进行求解。绘制每步解与真值之间以及最优解之间的距离曲线。


Proximal Gradient Method
distance to true
distance to optimal
distance
iteration
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
5000


ADMM
distance to true
distance to optimal
distance
iteration
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
100
200
300
400
500



由曲线我们可以看到：

它们的解都是走势单调下降的。邻近点梯度法和次梯度法的曲线图几乎一致，因为它们本质是一样的，尤其是固定步长的时候。区别在于邻近点梯度法使用的是软门限算法，次梯度法随机选取一个次梯度进行梯度下降。而ADMM在正则化强度比较小的情况下也能很好的解出真值，效果很好。而且收敛速度很快。

```
Proximal Gradient Time(lambda=1): 0.21726608276367188 s
Distance(lambda=1): 0.665112123667698

ADMM Time (lambda=1): 0.05924081802368164 s
Distance (lambda=1): 0.09709518002336946

Subgradient Time (lambda=1): 0.27386951446533203 s
Distance (lambda=1): 0.6653472627253224
```

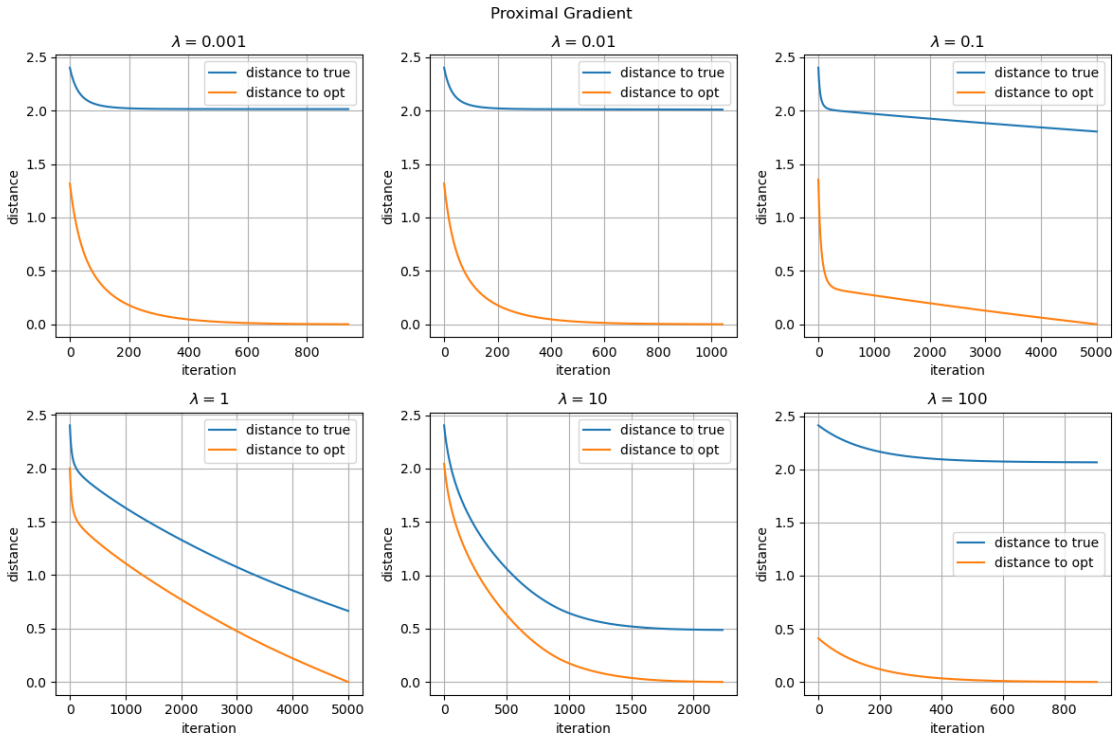
由二范数误差（最优解和真实解的距离）可以看到，在正则化强度比较小的时候，交替方向乘子法的解是最好的，但是交替方向乘子法速度相对慢，如果按照数学进行重复迭代矩阵求逆将会更慢，我在这里进行了矩阵和向量的预运算进行优化，将不会这么慢。但相比其它两个方法还是略慢一些。

说明交替方向乘子法效果最好，但是速度最慢，邻近点梯度法和次梯度法效果基本相同。

不同参数对比：

1. **邻近点梯度法**

曲线图如下：



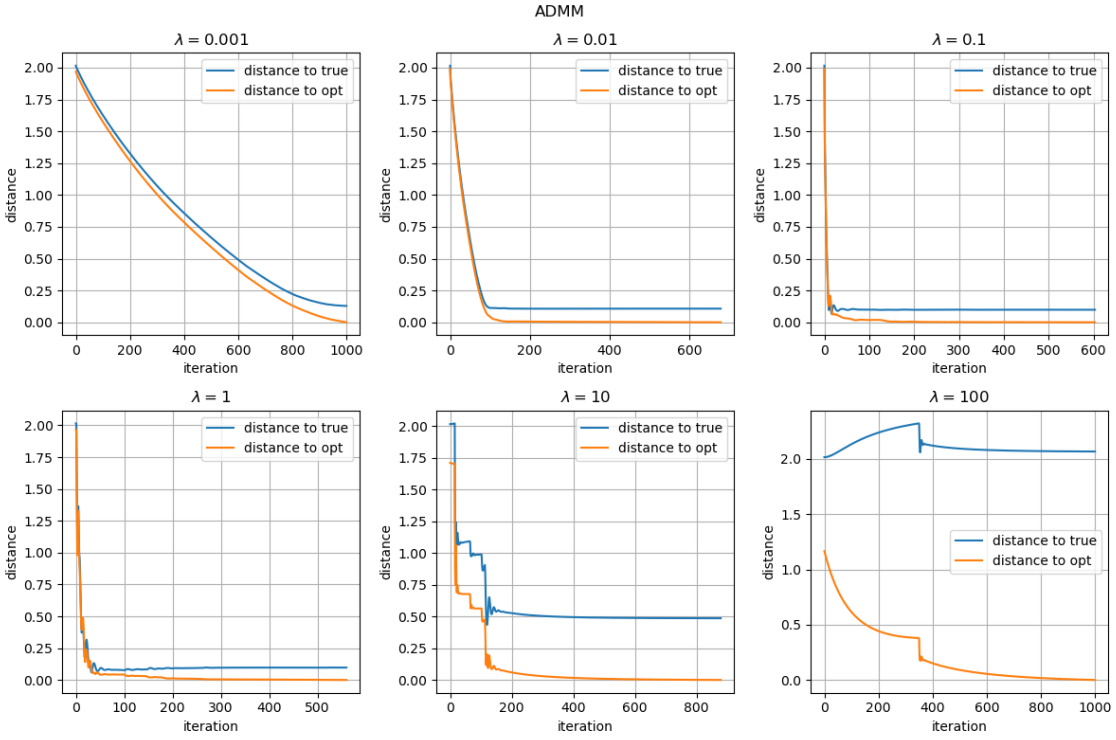

```
Proximal Gradient Time(lambda=0.001): 0.040038347244262695 s
Distance(lambda=0.001): 2.01528183345559

Proximal Gradient Time(lambda=0.01): 0.04206395149230957 s
Distance(lambda=0.01): 2.0108056602625766

Proximal Gradient Time(lambda=0.1): 0.20898818969726562 s
Distance(lambda=0.1): 1.8047782670044041

Proximal Gradient Time(lambda=1): 0.20668506622314453 s
Distance(lambda=1): 0.665112123667698

Proximal Gradient Time(lambda=10): 0.09207344055175781 s
Distance(lambda=10): 0.4873572562679913

Proximal Gradient Time(lambda=100): 0.03612017631530762 s
Distance(lambda=100): 2.0661212455528224
```

可以看到，对于邻近点梯度法并不是越大正则化效果越好，随着lamda增 大，最优解的正确性先变好再变差，即最优解和真值先接近后远离。

lamda过小，解不够稀疏，而真值是一个稀疏向量，故和真值有一些距离； 取值合适，解的稀疏性适
中，和真值比较接近； 过大，解过于稀疏，都很接近于0，拟合程度很低，故和真值有所偏离。

2. **交替方向乘子法**

曲线图如下：



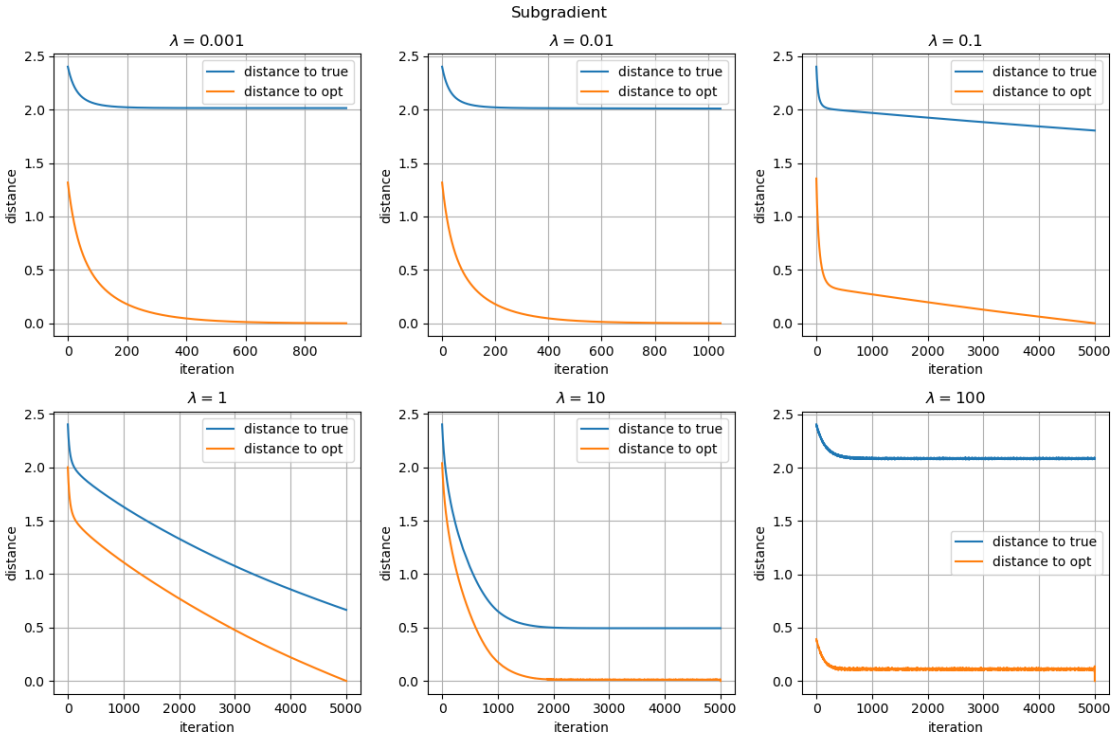
```
ADMM Time (lambda=0.001): 0.062325954437255 86 s
Distance (lambda=0.001): 0.128194573017646 53

ADMM Time (lambda=0.01): 0.042856931686401 37 s
Distance (lambda=0.01): 0.107093375911674 81

ADMM Time (lambda=0.1): 0.041861295700073 24 s
Distance (lambda=0.1): 0.098283436431573 22

ADMM Time (lambda=1): 0.038372278213500 98 s
Distance (lambda=1): 0.097095180023369 46

ADMM Time (lambda=10): 0.054810762405395 51 s
Distance (lambda=10): 0.486018413553223 6

ADMM Time (lambda=100): 0.076483011245727 54 s
Distance (lambda=100): 2.06696978351091 4
```

交替方向乘子法影响和邻近点梯度法类似，但是可以发现，当取值比较小的时候，交替方向乘子法效果还是很好，这说明交替方向乘子法的解本身就具有稀疏性。

交替方向乘子法的最优lamda大概在1左右。

3. **次梯度法**

结果如下：




```
Subgradient Time (lambda=0.001): 0.058762311935424805 s
Distance (lambda=0.001): 2.0152823741625028

Subgradient Time (lambda=0.01): 0.05336642265319824 s
Distance (lambda=0.01): 2.01078253966013

Subgradient Time (lambda=0.1): 0.2718369960784912 s
Distance (lambda=0.1): 1.8048146574998711

Subgradient Time (lambda=1): 0.24629545211791992 s
Distance (lambda=1): 0.6653935545308185

Subgradient Time (lambda=10): 0.24298405647277832 s
Distance (lambda=10): 0.49386766902105517

Subgradient Time (lambda=100): 0.24025535583496094 s
Distance (lambda=100): 2.0898702509719063
```

前面也提到过，次梯度法和邻近点梯度法本质是一样的，所以结果几乎一致， 对其影响也和邻近点梯度法一致。

四、实验结语

通过本次实验，我们对比了邻近点梯度法、交替方向乘子法（ADMM）和次梯度法在求解带有L1正则化的优化问题中的表现。实验结果表明，ADMM在正则化参数较小时能够快速收敛并获得较好的解，尽管其计算速度相对较慢，但优化效果最佳。邻近点梯度法和次梯度法在效果上较为接近，但对正则化参数较为敏感。实验还发现，正则化参数的选择对优化结果有显著影响，过小或过大的参数均会导致解的偏离。总体而言，三种方法各有优劣，适用于不同的应用场景。未来可进一步探索参数调整策略和算法优化，以提高求解效率和准确性。