# 数学建模大作业报告

| 姓名 | 班级 | 学号 |
| --- | --- | --- |
| 郑德凯 | 软件2202班 | U202217216 |

## 题目选择

本次大作业选择 **问题E**，结合具体案例对 **最速下降法**、**牛顿法**、**DFP算法** 和 **FR算法** 四个优化算法进行编程实现和比较。

---

## 一、案例描述

寻找函数

$$f(x, y, z) = x^2 + 2y^2 + 3z^2$$

的极小点。
该函数是一个二次型函数，具有凸性，其极小点位于 $(0, 0, 0)$。选择该函数作为案例是因为其简单且易于分析，同时能够清晰地展示不同优化算法的性能。

设初始点为 $(1, 1, 1)$，收敛阈值为 $\epsilon = 0.01$。初始点的选择远离极小点，便于观察算法的收敛过程；收敛阈值的设置是为了在保证精度的同时控制计算量。

---

## 二、算法实现

### 1. 最速下降法

**原理：**
最速下降法通过沿着目标函数梯度的负方向更新参数寻找极小点。算法核心步骤如下：

1. 计算当前点 $\mathbf{x}_k$ 的梯度 $\nabla f(\mathbf{x}_k)$。
2. 确定步长 $\alpha_k$，满足一定准则（例如 Wolfe 条件）。
3. 更新点 $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)$。
4. 重复上述步骤，直到满足收敛条件 $\|\nabla f(\mathbf{x}_k)\| < \epsilon$。

**代码实现：**

```python
import numpy as np

# 定义目标函数
def f(x):
    return x[0]**2 + 2 * x[1]**2 + 3 * x[2]**2

# 定义梯度函数
def grad_f(x):
    return np.array([2 * x[0], 4 * x[1], 6 * x[2]])

# 最速下降法
def gradient_descent(initial_x, epsilon, step_size):
    x = initial_x
    history = [x]
    while np.linalg.norm(grad_f(x)) >= epsilon:
        x = x - step_size * grad_f(x)
        history.append(x)
    return x, history

# 参数设置
initial_x = np.array([1, 1, 1])
epsilon = 0.01
step_size = 0.1  # 固定步长

# 执行算法
optimal_x, history = gradient_descent(initial_x, epsilon, step_size)
print("极小点:", optimal_x)
print("迭代次数:", len(history))
```

**运行结果：**

- 极小点：$[4.72236648 \times 10^{-03}, 4.73838134 \times 10^{-06}, 2.81474977 \times 10^{-10}]$
- 迭代次数：25

**分析：**
最速下降法实现简单，但收敛速度较慢，且对步长的选择敏感。固定步长可能导致收敛速度慢或不收敛，实际应用中通常结合线搜索方法动态调整步长。

---

## 2. 牛顿法

**原理：**
牛顿法使用目标函数的梯度和 Hessian 矩阵寻找极值点。公式如下：

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}^{-1} \nabla f(\mathbf{x}_k)$$

其中，$\mathbf{H}$ 为 Hessian 矩阵。

**代码实现：**

```
# 牛顿法
def newton_method(initial_x):
    H_inv = np.linalg.inv(np.array([[2, 0, 0], [0, 4, 0], [0, 0, 6]]))  # Hessian 矩阵的逆
    x_newton = initial_x - H_inv.dot(grad_f(initial_x))
    return x_newton

# 执行算法
optimal_x = newton_method(initial_x)
print("极小点:", optimal_x)
```

**运行结果：**

- 极小点：$[0, 0, 0]$

**分析：**
牛顿法在二次函数上表现极佳，仅需一次迭代即可收敛到极小点。但对于非二次函数或高维问题，Hessian 矩阵的计算和求逆可能成为瓶颈，且 Hessian 矩阵可能不可逆。

---

## 3. DFP算法

**原理：**
DFP 算法通过更新逆 Hessian 矩阵的估计来寻找极小点，核心更新公式如下：

$$\mathbf{H}_{k+1} = \mathbf{H}_k + \frac{\mathbf{s}_k \mathbf{s}_k^T}{\mathbf{s}_k^T \mathbf{y}_k} - \frac{\mathbf{H}_k \mathbf{y}_k \mathbf{y}_k^T \mathbf{H}_k}{\mathbf{y}_k^T \mathbf{H}_k \mathbf{y}_k}$$

其中，$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$，$\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$。

**代码实现：**

```
# DFP算法
def dfp_method(f, grad_f, initial_x, epsilon, max_iter=1000):
    x = initial_x
    H = np.eye(len(x))  # 初始逆 Hessian 矩阵
    g = grad_f(x)
    history = [x]
    for _ in range(max_iter):
        if np.linalg.norm(g) < epsilon:
            break
        p = -H.dot(g)
        alpha = 0.1  # 固定步长
        x_new = x + alpha * p
        g_new = grad_f(x_new)
```

```
        s = x_new - x
        y = g_new - g
        sy = s.dot(y) + 1e-10  # 避免除零
        H = H + np.outer(s, s) / sy - H.dot(np.outer(y, y)).dot(H) /
(y.dot(H).dot(y) + 1e-10)
        x, g = x_new, g_new
        history.append(x)
    return x, history

# 执行算法
optimal_x, history = dfp_method(f, grad_f, initial_x, epsilon)
print("极小点:", optimal_x)
print("迭代次数:", len(history))
```

**运行结果：**

- 极小点：$[1.88384 \times 10^{-03}, 1.40315 \times 10^{-03}, 1.17085 \times 10^{-03}]$
- 迭代次数：58

**分析：**
DFP 算法不需要显式计算 Hessian 矩阵，适合大规模优化问题。但其收敛速度受初始条件和步长选择的影响，且在高维问题中可能存在数值不稳定性。

---

## 4. FR（Fletcher-Reeves）算法实现

**原理：**
FR 算法是一种共轭梯度法，通过利用前一次的搜索方向信息来构造新的搜索方向，从而加速收敛。其核心步骤如下：

1. 初始化：选择初始点 $\mathbf{x}_0$，计算初始梯度 $\mathbf{g}_0 = \nabla f(\mathbf{x}_0)$，设置初始搜索方向 $\mathbf{d}_0 = -\mathbf{g}_0$。
2. 迭代更新：
   - 计算步长 $\alpha_k$，通常通过线搜索确定。
   - 更新点 $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$。
   - 计算新梯度 $\mathbf{g}_{k+1} = \nabla f(\mathbf{x}_{k+1})$。
   - 计算共轭系数 $\beta_k = \frac{\mathbf{g}_{k+1}^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}$。
   - 更新搜索方向 $\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k$。
3. 重复上述步骤，直到满足收敛条件 $\|\mathbf{g}_k\| < \epsilon$。

**代码实现：**

```
# FR（Fletcher-Reeves）算法
def fr_method(f, grad_f, initial_x, epsilon, max_iter=1000):
    x = initial_x
```

```
    g = grad_f(x)
    d = -g
    history = [x]
    for _ in range(max_iter):
        if np.linalg.norm(g) < epsilon:
            break
        # 线搜索确定步长（这里使用固定步长）
        alpha = 0.1
        x_new = x + alpha * d
        g_new = grad_f(x_new)
        beta = np.dot(g_new, g_new) / (np.dot(g, g) + 1e-10)  # 避免除零
        d = -g_new + beta * d
        x, g = x_new, g_new
        history.append(x)
    return x, history

# 执行算法
optimal_x, history = fr_method(f, grad_f, initial_x, epsilon)
print("极小点:", optimal_x)
print("迭代次数:", len(history))
```

**运行结果：**

- 极小点：$[3.41954590 \times 10^{-03}, 8.57379014 \times 10^{-05}, -3.63416172 \times 10^{-04}]$
- 迭代次数：12

**分析：**
FR 算法通过利用前一次的搜索方向信息，能够在一定程度上加速收敛。与最速下降法相比，FR 算法在收敛速度上有显著提升，尤其适合大规模优化问题。然而，FR 算法的性能仍然受到初始条件和步长选择的影响。

---

# 三、算法比较

| 算法 | 迭代次数 | 极小点 | 特点 |
| --- | --- | --- | --- |
| 最速下降法 | 25 | $[4.72236648 \times 10^{-03}, 4.73838134 \times 10^{-06}, 2.81474977 \times 10^{-10}]$ | 实现简单，收敛较慢，适合初学者实现。 |
| 牛顿法 | 1 | $[0, 0, 0]$ | 对二次函数高效，但 Hessian |

| 算法 | 迭代次数 | 极小点 | 特点 |
| --- | --- | --- | --- |
| | | | 矩阵计算复杂，适合小规模问题。 |
| DFP 算法 | 58 | $[1.88384 \times 10^{-03}, 1.40315 \times 10^{-03}, 1.17085 \times 10^{-03}]$ | 不需显式计算 Hessian 矩阵，适合大规模问题，但收敛速度受初始条件影响。 |
| FR算法 | 12 | $[3.41954590 \times 10^{-03}, 8.57379014 \times 10^{-05}, -3.63416172 \times 10^{-04}]$ | 利用共轭梯度加速收敛，适合大规模问题，但收敛速度受初始条件影响。 |

---

# 四、总结

通过对最速下降法、牛顿法、DFP 算法和 FR 算法的实现与比较，我们发现：

1. **牛顿法** 在二次函数上表现最优，但计算复杂度较高，适合小规模问题。
2. **最速下降法** 实现简单，但收敛速度较慢，适合初学者学习和实现。
3. **DFP 算法** 和 **FR 算法** 在不需要显式计算 Hessian 矩阵的情况下，仍能保持较快的收敛速度，适合大规模优化问题。其中，FR 算法通过利用共轭梯度信息，能够在一定程度上加速收敛。