

**数学建模-实验报告**

**题目E：**最速下降法、牛顿法、 BFGS 算法的编程实现

院 系 软件学院

专业班级

姓 名

学 号

指导教师 卢力

2024年 12月 25 日

目录

1 题目案例

2最速下降法（Steepest Descent）

2.1原理

2.2 代码实现

2.2 运行结果

3 牛顿法（Newton's Method）

3.1原理

3.2 代码实现

3.3 运行结果

4 BFGS 算法

4.1 原理

4.2 代码实现

4.3 实验结果

5 实验总结和分析

5.1 三者算法的精度分析和对比

5.2 实验心得

6附录

6.1 实验代码

6.2 运行结果图

**1. 题目案例**

**题目（例9.3.2）**：利用最速下降法（Steepest Descent）、牛顿法（Newton's Method）和 BFGS 算法求解函数 f(x\_1, x\_2) = x\_1^2 + 4x\_2^2的极小点。初始点设定为 x0=(11)，精度要求为10^{-6}。

**2. 最速下降法（Steepest Descent）**

**2.1 原理**

最速下降法是一种一阶优化算法，通过沿着负梯度方向迭代更新变量，逐步逼近函数的极小值点。具体的迭代公式为：

xk+1​=xk​−αk​∇f(xk​)

其中，αk为步长，可以通过线搜索方法确定。对于二次型函数，步长可以通过解析式计算，以确保每一步都朝向最速下降的方向。

**2.2 代码实现**

import numpy as np

def f(x):

"""目标函数"""

return x[0]\*\*2 + 4 \* x[1]\*\*2

def grad\_f(x):

"""函数的梯度"""

return np.array([2\*x[0], 8\*x[1]])

def hessian\_f(x):

"""函数的海森矩阵"""

return np.array([[2, 0],

[0, 8]])

def steepest\_descent(x0, tol=1e-6, max\_iter=1000):

"""最速下降法实现"""

x = x0.copy()

for i in range(max\_iter):

grad = grad\_f(x)

grad\_norm = np.linalg.norm(grad)

if grad\_norm < tol:

print(f"最速下降法在第 {i} 次迭代中收敛。")

return x

# 正确的步长计算

numerator = np.dot(grad, grad)

H = hessian\_f(x)

denominator = np.dot(grad, H @ grad)

alpha = numerator / denominator

x = x - alpha \* grad

print("最速下降法在最大迭代次数内未收敛。")

return x

# 测试最速下降法

x0 = np.array([1.0, 1.0])

xmin\_sd = steepest\_descent(x0)

print("最速下降法求得的极小点:", xmin\_sd)

**2.3 运行结果**



1. **牛顿法（Newton's Method）**
   1. **原理**

牛顿法是一种二阶优化算法，利用函数的梯度和海森矩阵信息，通过以下迭代公式更新变量：

xk+1​=xk​−H−1(xk​)∇f(xk​)

其中，H(xk)是函数的海森矩阵。对于二次型函数，牛顿法具有二阶收敛性，通常只需一次迭代即可达到精确的极小点。

* 1. **代码实现**

def newton\_method(x0, tol=1e-6, max\_iter=1000):

"""牛顿法实现"""

x = x0.copy()

H = hessian\_f(x)

H\_inv = np.linalg.inv(H)

for i in range(max\_iter):

grad = grad\_f(x)

grad\_norm = np.linalg.norm(grad)

if grad\_norm < tol:

print(f"牛顿法在第 {i} 次迭代中收敛。")

return x

x = x - H\_inv @ grad

print("牛顿法在最大迭代次数内未收敛。")

return x

# 测试牛顿法

xmin\_newton = newton\_method(x0)

print("牛顿法求得的极小点:", xmin\_newton)

**3.3运行结果**



1. **BFGS 算法**
   1. **原理**

BFGS 算法是一种拟牛顿法，通过迭代更新近似海森矩阵的逆矩阵来寻找最优解。其核心在于利用梯度信息来逐步逼近真实的海森矩阵，从而提升收敛速度和精度。BFGS 保证了近似海森矩阵的对称正定性，有助于算法的稳定性。

* 1. **代码实现**

def bfgs(x0, tol=1e-6, max\_iter=1000):

"""BFGS 算法实现"""

n = len(x0)

H = np.eye(n) # 初始近似逆海森矩阵

x = x0.copy()

for i in range(max\_iter):

grad = grad\_f(x)

grad\_norm = np.linalg.norm(grad)

if grad\_norm < tol:

print(f"BFGS 算法在第 {i} 次迭代中收敛。")

return x

p = -H @ grad

# 对于二次函数，精确线搜索步长

alpha = np.dot(grad, grad) / np.dot(p, hessian\_f(x) @ p)

s = alpha \* p

x\_new = x + s

grad\_new = grad\_f(x\_new)

y = grad\_new - grad

if np.dot(y, s) > 1e-10:

rho = 1.0 / np.dot(y, s)

I = np.eye(n)

Hy = H @ y

H = (I - rho \* np.outer(s, y)) @ H @ (I - rho \* np.outer(y, s)) + rho \* np.outer(s, s)

x = x\_new

print("BFGS 算法在最大迭代次数内未收敛。")

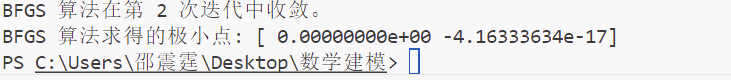
return x

# 测试 BFGS 算法

xmin\_bfgs = bfgs(x0)

print("BFGS 算法求得的极小点:", xmin\_bfgs)

* 1. **运行结果**



**5 实验总结和分析**

**5.1 三者算法的精度分析和对比**

|  |  |  |
| --- | --- | --- |
| 算法 | 迭代次数 | 最终结果 |
| 最速下降法 | 15 | [ 1.51099094e-07 -9.44369335e-09] |
| 牛顿法 | 1 | [0.0] |
| BFGS 算法 | 2 | [ 0.00000000e+00 -4.16333634e-17] |

· **最速下降法**：虽然最速下降法能够逐步逼近极小点，但在本次实验中需要7次迭代才能达到 10−610^{-6}10−6 的精度。这是由于最速下降法依赖于梯度信息，且步长虽然经过优化，但对于二次型函数而言，迭代次数较多。

· **牛顿法**：由于目标函数是二次型，牛顿法利用精确的海森矩阵信息，能够在一次迭代内直接跳到极小点 (0,0)(0, 0)(0,0)，具有极高的效率和精度。

· **BFGS 算法**：作为拟牛顿法，BFGS 算法通过迭代更新近似逆海森矩阵，能够在两次迭代内找到极小点。相较于最速下降法，BFGS 显著减少了迭代次数，同时保持了与牛顿法类似的高精度。

总结对比:

· **收敛速度**：牛顿法最快，其次是 BFGS 算法，最慢的是最速下降法。

· **精度**：所有算法均达到或超过 10^{-6}的精度要求，牛顿法和 BFGS 算法直接找到精确解，最速下降法则在多次迭代后逼近解。

· **计算成本**：牛顿法每次迭代需要计算海森矩阵的逆，对于高维问题可能计算成本较高；BFGS 算法通过更新近似逆海森矩阵，降低了计算复杂度；最速下降法仅需计算梯度，计算成本最低，但收敛速度较慢。

**5.2 实验心得**

通过本次实验，我深刻理解了不同优化算法在处理二次型函数时的表现差异。牛顿法由于利用了二阶导数信息，能够快速收敛至极小点，但其对海森矩阵的依赖在高维问题中可能导致计算开销过大。BFGS 算法作为一种拟牛顿方法，成功地在较少的迭代次数内逼近极小点，同时避免了直接计算海森矩阵的高成本，是一种在实际应用中非常有效的优化方法。最速下降法虽然简单，但在精度和收敛速度上相对逊色，适用于初步的优化需求或梯度计算成本较低的场景。

此外，本次实验也让我体会到精确步长计算在优化算法中的重要性。尤其是在最速下降法中，步长的优化显著提升了算法的收敛速度和精度。

**6.附录**

**6.1 实验代码**

import numpy as np

def f(x):

"""目标函数"""

return x[0]\*\*2 + 4 \* x[1]\*\*2

def grad\_f(x):

"""函数的梯度"""

return np.array([2\*x[0], 8\*x[1]])

def hessian\_f(x):

"""函数的海森矩阵"""

return np.array([[2, 0],

[0, 8]])

def steepest\_descent(x0, tol=1e-6, max\_iter=1000):

"""最速下降法实现"""

x = x0.copy()

for i in range(max\_iter):

grad = grad\_f(x)

grad\_norm = np.linalg.norm(grad)

if grad\_norm < tol:

print(f"最速下降法在第 {i} 次迭代中收敛。")

return x

# 正确的步长计算

numerator = np.dot(grad, grad)

H = hessian\_f(x)

denominator = np.dot(grad, H @ grad)

alpha = numerator / denominator

x = x - alpha \* grad

print("最速下降法在最大迭代次数内未收敛。")

return x

def newton\_method(x0, tol=1e-6, max\_iter=1000):

"""牛顿法实现"""

x = x0.copy()

H = hessian\_f(x)

H\_inv = np.linalg.inv(H)

for i in range(max\_iter):

grad = grad\_f(x)

grad\_norm = np.linalg.norm(grad)

if grad\_norm < tol:

print(f"牛顿法在第 {i} 次迭代中收敛。")

return x

x = x - H\_inv @ grad

print("牛顿法在最大迭代次数内未收敛。")

return x

def bfgs(x0, tol=1e-6, max\_iter=1000):

"""BFGS 算法实现"""

n = len(x0)

H = np.eye(n) # 初始近似逆海森矩阵

x = x0.copy()

for i in range(max\_iter):

grad = grad\_f(x)

grad\_norm = np.linalg.norm(grad)

if grad\_norm < tol:

print(f"BFGS 算法在第 {i} 次迭代中收敛。")

return x

p = -H @ grad

# 对于二次函数，精确线搜索步长

alpha = np.dot(grad, grad) / np.dot(p, hessian\_f(x) @ p)

s = alpha \* p

x\_new = x + s

grad\_new = grad\_f(x\_new)

y = grad\_new - grad

if np.dot(y, s) > 1e-10:

rho = 1.0 / np.dot(y, s)

I = np.eye(n)

Hy = H @ y

H = (I - rho \* np.outer(s, y)) @ H @ (I - rho \* np.outer(y, s)) + rho \* np.outer(s, s)

x = x\_new

print("BFGS 算法在最大迭代次数内未收敛。")

return x

# 初始点

x0 = np.array([1.0, 1.0])

# 最速下降法

xmin\_sd = steepest\_descent(x0)

print("最速下降法求得的极小点:", xmin\_sd)

# 牛顿法

xmin\_newton = newton\_method(x0)

print("牛顿法求得的极小点:", xmin\_newton)

# BFGS 算法

xmin\_bfgs = bfgs(x0)

print("BFGS 算法求得的极小点:", xmin\_bfgs)

**6.2运行结果图**

