

# 最优化上机作业

数学强基 22 DBQDSS

学号：1145141919

2024 年 4 月 29 日

## 目录

<b>1</b>	<b>第一道上机题目</b>	<b>2</b>
1.1	题目描述 . . . . .	2
1.2	迭代格式 . . . . .	2
1.3	运行结果 . . . . .	2
1.4	程序源码 . . . . .	2
<b>2</b>	<b>第二道上机题目</b>	<b>5</b>
2.1	题目描述 . . . . .	5
2.2	迭代格式 . . . . .	5
2.3	运行结果 . . . . .	6
2.4	程序源码 . . . . .	6

## 程序说明

执行平台	<b>PyCharm</b>
代码语言	<b>Python</b>
执行环境	<b>Anaconda 自定义环境</b>
所用库	<b>Numpy</b>
Python 版本	<b>Python 3.11.8</b>

## 1 第一道上机题目

### 1.1 题目描述

分别用最速下降法、牛顿法极小化 Rosenbrock 函数

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, x^{(0)} = (-1.2, 1), x^* = (1, 1), f(x^*) = 0$$

### 1.2 迭代格式

最速下降法

$$x_{k+1} = x_k - \alpha_k g_k$$

其中,  $g_k$  是目标函数在  $x_k$  处的梯度,  $\alpha_k$  是第  $k$  步的步长因子

牛顿法

$$x_{k+1} = x_k - G_k^{-1} g_k$$

其中,  $g_k$  是目标函数在  $x_k$  处的梯度,  $G_k$  是目标函数在  $x_k$  处的 Hesse 矩阵

### 1.3 运行结果

	最速下降法	牛顿法
目标函数最优解	[1. 1.]	[1. 1.]
目标函数最小值	1.5073383385145568e-19	0.0
迭代次数	1692	7

### 1.4 程序源码

最速下降法

```
1 import numpy as np
2
3 # 目标函数
```

```
4 def f(x):
5     return 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2
6
7 # 目标函数的梯度
8 def grad_f(x):
9     return np.array([- 400 * (x[1] - x[0] ** 2) * x[0] - 2 * (1 - x[0]),
10                      200 * (x[1] - x[0] ** 2)])
11
12 # Armijo算法
13 def armijo(x, dk, c1=0.2, beta=0.5):
14     alpha = 1
15     while f(x + alpha * dk) > f(x) + c1 * alpha * np.dot(grad_f(x), dk):
16         alpha *= beta
17     return alpha
18
19 # 最速下降法
20 def gradient_descent(x0, max_iter=10000, tol=1e-12):
21     """
22     x0: 初始点
23     max_iter: 最大迭代次数
24     tol: 精度
25     """
26     x = x0
27     k = 0 # 计数: 迭代次数
28     while k < max_iter:
29         g = grad_f(x)
30         d = - g
31         # 求步长因子alpha
32         alpha = armijo(x, d)
33         x_new = x + alpha * d
34         if np.linalg.norm(x_new - x) < tol:
35             break
36         x = x_new # 更新点
37         k += 1 # 更新迭代次数
38     return x, f(x), k
39
40 # 初始点
41 x0 = np.array([-1.2, 1])
42 x_star, f_min, k = gradient_descent(x0)
```

```
42 print("目标函数最优解: ", x_star)
43 print("目标函数最小值: ", f_min)
44 print("迭代次数: ", k)
```

## 牛顿法

```
1 import numpy as np
2
3 # 目标函数
4 def f(x):
5     return 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2
6
7 # 目标函数的梯度
8 def grad_f(x):
9     return np.array([- 400 * (x[1] - x[0] ** 2) * x[0] - 2 * (1 - x[0]),
10                      200 * (x[1] - x[0] ** 2)])
11
12 # 海森矩阵
13 def hessian(x):
14     x1 = x[0]
15     x2 = x[1]
16     return np.array([[ - 400 * (x2 - 3 * x1 ** 2) + 2, - 400 * x1],
17                      [- 400 * x1 , 200]])
18
19 # 牛顿法
20 def newton(x0, max_iter=10000, tol=1e-10):
21     """
22     x0: 初始点
23     max_iter: 最大迭代次数
24     tol: 精度
25     """
26     x = x0
27     k = 0 # 计数: 迭代次数
28     while k < max_iter:
29         g = grad_f(x)
30         G = hessian(x) # 海森矩阵
31         x_new = x - np.squeeze(np.linalg.inv(G) @ g)
32         if np.linalg.norm(g) < tol:
33             break
```

```

32     x = x_new # 更新点
33     k += 1 # 更新迭代次数
34     return x, f(x), k
35
36 # 初始点
37 x0 = np.array([-1.2, 1])
38 x_star, f_min, k = newton(x0)
39 print("目标函数最优解: ", x_star)
40 print("目标函数最小值: ", f_min)
41 print("迭代次数: ", k)

```

## 2 第二道上机题目

### 2.1 题目描述

分别用共轭梯度法和拟牛顿法极小化 Powell 奇异函数

$$f(x) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4$$

初始点  $x^{(0)} = (3, -1, 0, 1)$ , 解为  $x^* = (0, 0, 0, 0)$ ,  $f(x^*) = 0$ .

### 2.2 迭代格式

共轭梯度法

$$x_{k+1} = x_k + \alpha_k d_k$$

$$d_k = -g_k + \beta_{k-1} d_{k-1}$$

$$d_0 = -g_0$$

$$\beta_{k-1} = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}} \quad (\text{Fletcher-Reeves 公式})$$

其中,  $g_k$  是目标函数在  $x_k$  处的梯度,  $\alpha_k$  是第  $k$  步的步长因子

拟牛顿法

$$x_{k+1} = x_k - \alpha_k B_k^{-1} g_k$$

其中,  $g_k$  是目标函数在  $x_k$  处的梯度,  $\alpha_k$  是第  $k$  步的步长因子,  $B_k$  为 Hesse 矩阵的第  $k$  步近似

## 2.3 运行结果

	共轭梯度法
目标函数最优解	[-9.75925474e-08, 9.75925474e-09, -4.30262309e-08, -4.30262309e-08]
目标函数最小值	1.7292433324185916e-28
迭代次数	2682

	拟牛顿法
目标函数最优解	[ 3.51691746e-08, -3.51691746e-09, -6.78848416e-08, -6.78848416e-08]
目标函数最小值	1.43379954e-27
迭代次数	374

## 2.4 程序源码

### 共轭梯度法

```

1 import numpy as np
2
3 # 目标函数
4 def powell(x):
5     return ((x[0] + 10 * x[1]) ** 2 + 5 * (x[2] - x[3]) ** 2
6             + (x[1] - 2 * x[2]) ** 4 + 10 * (x[0] - x[3]) ** 4)
7
8 # 目标函数的梯度
9 def grad_powell(x):
10    return np.array([2 * (x[0] + 10 * x[1]) + 40 * (x[0] - x[3]) ** 3,
11                    20 * (x[0] + 10 * x[1]) + 4 * (x[1] - 2 * x[2]) ** 3,
12                    10 * (x[2] - x[3]) - 8 * (x[1] - 2 * x[2]) ** 3,
13                    -10 * (x[2] - x[3]) - 40 * (x[0] - x[3]) ** 3])
14
15 # Armijo算法
16 def armijo(x, dk, c1=0.2, beta=0.5):

```

```
17     alpha = 1
18     while powell(x + alpha * dk) > powell(x) + c1 * alpha *
19         np.dot(grad_powell(x), dk):
20         alpha *= beta
21     return alpha
22
23 # FR-CG算法
24 def FRCG(x0, max_iter=10000, tol=1e-19):
25     """
26     x0: 初始点
27     max_iter: 最大迭代次数
28     tol: 精度
29     """
30     x = x0
31     k = 0
32     d = - grad_powell(x)
33
34     while k < max_iter:
35         if np.linalg.norm(d) < tol:
36             return x, powell(x), k
37
38         # 求步长因子
39         alpha = armijo(x, d)
40
41         # 计算下一个迭代点
42         x_new = x + alpha * d
43         beta = (grad_powell(x_new).T @ grad_powell(x_new)) /
44             (grad_powell(x).T @ grad_powell(x))
45         d = -grad_powell(x_new) + beta * d
46
47         # 更新点
48         x = x_new
49         k += 1 # 更新迭代次数
50     return x, powell(x), k
51
52 # 初始点
53 x0 = np.array([3.0, -1, 0, 1])
54 x_star, f_min, k = FRCG(x0)
55 print("目标函数最优解: ", x_star)
```

```
54 print("目标函数最小值: ", f_min)
55 print("迭代次数: ", k)
```

### 拟牛顿法

```
1 import numpy as np
2
3 def powell(x):
4     return ((x[0] + 10 * x[1]) ** 2 + 5 * (x[2] - x[3]) ** 2
5             + (x[1] - 2 * x[2]) ** 4 + 10 * (x[0] - x[3]) ** 4)
6
7 def grad_powell(x):
8     return np.array([2 * (x[0] + 10 * x[1]) + 40 * (x[0] - x[3]) ** 3,
9                      2 * (x[0] + 10 * x[1]) + 4 * (x[1] - 2 * x[2]) ** 3,
10                     10 * (x[2] - x[3]) - 8 * (x[1] - 2 * x[2]) ** 3,
11                     -10 * (x[2] - x[3]) - 40 * (x[0] - x[3])
12                     ** 3]).reshape(1,-1).T
13
14 # BFGS算法
15 def bfgs_update(B, s, y):
16     return B + (y @ y.T) / (y.T @ s) - (B @ s @ s.T @ B) / (s.T @ B @ s)
17
18 # Armijo算法
19 def armijo(x, dk, c1=0.2, beta=0.5):
20     alpha = 1
21     while powell(x + alpha * dk) > powell(x) + c1 * alpha *
22         np.dot(grad_powell(x).T, dk):
23         alpha *= beta
24     return alpha
25
26 def quasi_newton_method(x0, max_iter=10000, tol=1e-19):
27     """
28     x0: 初始点
29     tol: 收敛阈值
30     max_iter: 最大迭代次数
31     """
32     # 转为列向量
33     x = x0.reshape(1,-1).T
```



```
32 B = np.eye(4) # 初始海森矩阵近似为单位矩阵
33 g = grad_powell(x)
34
35 k = 0 # 计数
36 while k < max_iter:
37     # 检查收敛性
38     if np.linalg.norm(g) < tol:
39         break
40
41     # 计算搜索方向
42     d = - np.linalg.inv(B) @ g
43
44     # 求步长因子 alpha
45     alpha = armijo(x, d)
46     x_new = x + alpha * d
47     g_new = grad_powell(x_new)
48
49     # 计算s和y
50     s = x_new - x
51     y = g_new - g
52
53     # BFGS校正更新B
54     B = bfgs_update(B, s, y)
55
56     # 更新x和g
57     x = x_new
58     g = g_new
59
60     k += 1
61     return x, powell(x), k
62
63 # 初始点
64 x0 = np.array([3,-1,0,1])
65 # 使用拟牛顿法求解
66 x_star, f_min, k= quasi_newton_method(x0)
67 # 转为行向量
68 x_star = x_star.T.reshape(4)
69 print("目标函数最优解: ", x_star)
70 print("目标函数最小值: ", f_min)
```

71 `print("迭代次数: ", k)`

---