最优化上机作业

数学强基 22 DBQDSS
学号：1145141919

2024 年 4 月 29 日

目录

程序说明

执行平台	**PyCharm**
代码语言	**Python**
执行环境	**Anaconda 自定义环境**
所用库	**Numpy**
Python 版本	**Python 3.11.8**

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，α_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

```
import numpy as np

# 目标函数
```

```python
def f(x):
    return 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2

# 目标函数的梯度
def grad_f(x):
    return np.array([- 400 * (x[1] - x[0] ** 2) * x[0] - 2 * (1 - x[0]),
        200 * (x[1] - x[0] ** 2)])

# Armijo算法
def armijo(x, dk, c1=0.2, beta=0.5):
    alpha = 1
    while f(x + alpha * dk) > f(x) + c1 * alpha * np.dot(grad_f(x), dk):
        alpha *= beta
    return alpha

# 最速下降法
def gradient_descent(x0, max_iter=10000, tol=1e-12):
    """
    x0: 初始点
    max_iter: 最大迭代次数
    tol: 精度
    """
    x = x0
    k = 0 # 计数: 迭代次数
    while k < max_iter:
        g = grad_f(x)
        d = - g
        # 求步长因子alpha
        alpha = armijo(x, d)
        x_new = x + alpha * d
        if np.linalg.norm(x_new - x) < tol:
            break
        x = x_new # 更新点
        k += 1 # 更新迭代次数
    return x, f(x), k

# 初始点
x0 = np.array([-1.2, 1])
x_star, f_min, k = gradient_descent(x0)
```

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

牛顿法

```
import numpy as np

# 目标函数
def f(x):
    return 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2

# 目标函数的梯度
def grad_f(x):
    return np.array([- 400 * (x[1] - x[0] ** 2) * x[0] - 2 * (1 - x[0]),
        200 * (x[1] - x[0] ** 2)])

# 海森矩阵
def hessian(x):
    x1 = x[0]
    x2 = x[1]
    return np.array([[ - 400 * (x2 - 3 * x1 ** 2) + 2, - 400 * x1],
                [- 400 * x1 , 200]])
# 牛顿法
def newton(x0, max_iter=10000, tol=1e-10):
    """
    x0: 初始点
    max_iter: 最大迭代次数
    tol: 精度
    """
    x = x0
    k = 0 # 计数: 迭代次数
    while k < max_iter:
        g = grad_f(x)
        G = hessian(x) # 海森阵
        x_new = x - np.squeeze(np.linalg.inv(G) @ g)
        if np.linalg.norm(g) < tol:
            break
```

```
        x = x_new # 更新点
        k += 1 # 更新迭代次数
    return x, f(x), k

# 初始点
x0 = np.array([-1.2, 1])
x_star, f_min, k = newton(x0)
print("目标函数最优解: ", x_star)
print("目标函数最小值: ", f_min)
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

$$\begin{aligned} x_{k+1} &= x_k + \alpha_k d_k \\ d_k &= -g_k + \beta_{k-1} d_{k-1} \\ d_0 &= -g_0 \\ \beta_{k-1} &= \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}} \quad \text{(Fletcher-Reeves 公式)} \end{aligned}$$

其中，g_k 是目标函数在 x_k 处的梯度，α_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，α_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

```
import numpy as np

# 目标函数
def powell(x):
    return ((x[0] + 10 * x[1]) ** 2 + 5 * (x[2] - x[3]) ** 2
           + (x[1] - 2 * x[2]) ** 4 + 10 * (x[0] - x[3]) ** 4)

# 目标函数的梯度
def grad_powell(x):
    return np.array([2 * (x[0] + 10 * x[1]) + 40 * (x[0] - x[3]) ** 3,
                     20 * (x[0] + 10 * x[1]) + 4 * (x[1] - 2 * x[2]) ** 3,
                     10 * (x[2] - x[3]) - 8 * (x[1] - 2 * x[2]) ** 3,
                     -10 * (x[2] - x[3]) - 40 * (x[0] - x[3]) ** 3])

# Armijo算法
def armijo(x, dk, c1=0.2, beta=0.5):
```

```
    alpha = 1
    while powell(x + alpha * dk) > powell(x) + c1 * alpha *
        np.dot(grad_powell(x), dk):
        alpha *= beta
    return alpha

# FR-CG算法
def FRCG(x0, max_iter=10000, tol=1e-19):
    """
    x0: 初始点
    max_iter: 最大迭代次数
    tol: 精度
    """
    x = x0
    k = 0
    d = - grad_powell(x)

    while k < max_iter:
        if np.linalg.norm(d) < tol:
            return x, powell(x), k

        # 求步长因子
        alpha = armijo(x, d)

        # 计算下一个迭代点
        x_new = x + alpha * d
        beta = (grad_powell(x_new).T @ grad_powell(x_new)) /
            (grad_powell(x).T @ grad_powell(x))
        d = -grad_powell(x_new) + beta * d

        # 更新点
        x = x_new
        k += 1 # 更新迭代次数
    return x, powell(x), k

# 初始点
x0 = np.array([3.0, -1, 0, 1])
x_star, f_min, k = FRCG(x0)
print("目标函数最优解: ", x_star)
```

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

拟牛顿法

```
import numpy as np

def powell(x):
    return ((x[0] + 10 * x[1]) ** 2 + 5 * (x[2] - x[3]) ** 2
           + (x[1] - 2 * x[2]) ** 4 + 10 * (x[0] - x[3]) ** 4)

def grad_powell(x):
    return np.array([2 * (x[0] + 10 * x[1]) + 40 * (x[0] - x[3]) **3,
                     2 * (x[0] + 10 * x[1]) + 4 * (x[1] - 2 * x[2]) **3,
                     10 * (x[2] - x[3]) - 8 * (x[1] - 2 * x[2]) **3,
                     -10 * (x[2] - x[3]) - 40 * (x[0] - x[3])
                         **3]).reshape(1,-1).T

# BFGS算法
def bfgs_update(B, s, y):
    return B + (y @ y.T) / (y.T @ s) - (B @ s @ s.T @ B) / (s.T @ B @ s)

# Armijo算法
def armijo(x, dk, c1=0.2, beta=0.5):
    alpha = 1
    while powell(x + alpha * dk) > powell(x) + c1 * alpha *
        np.dot(grad_powell(x).T, dk):
       alpha *= beta
    return alpha

def quasi_newton_method(x0, max_iter=10000, tol=1e-19):
    """
    x0: 初始点
    tol: 收敛阈值
    max_iter: 最大迭代次数
    """
    # 转为列向量
    x = x0.reshape(1,-1).T
```

```
    B = np.eye(4) # 初始海森矩阵近似为单位矩阵
    g = grad_powell(x)

    k = 0 # 计数
    while k < max_iter:
        # 检查收敛性
        if np.linalg.norm(g) < tol:
            break

        # 计算搜索方向
        d = - np.linalg.inv(B) @ g

        # 求步长因子 alpha
        alpha = armijo(x, d)
        x_new = x + alpha * d
        g_new = grad_powell(x_new)

        # 计算s和y
        s = x_new - x
        y = g_new - g

        # BFGS校正更新B
        B = bfgs_update(B, s, y)

        # 更新x和g
        x = x_new
        g = g_new

        k += 1
    return x, powell(x), k

# 初始点
x0 = np.array([3,-1,0,1])
# 使用拟牛顿法求解
x_star, f_min, k= quasi_newton_method(x0)
# 转为行向量
x_star = x_star.T.reshape(4)
print("目标函数最优解: ", x_star)
print("目标函数最小值: ", f_min)
```

```
71 print("迭代次数: ", k)
```