

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF
CHINA



OPERATIONS RESEARCH

001251

Course Project Report

Author:

Jiazhi HE: PB19151772

December 24, 2021

Contents

| | | |
|-----|------------------------------|---|
| 1 | 问题描述 | 2 |
| 2 | 算法原理 | 2 |
| 2.1 | 基于 Wolfe-Powell 准则的非精确一维搜索算法 | 2 |
| 2.2 | 牛顿法 | 3 |
| 3 | 数据集说明 | 3 |
| 4 | 程序输入输出说明 | 4 |
| 5 | 程序测试结果 | 6 |
| 5.1 | 测试函数 1 | 6 |
| 5.2 | 测试函数 2 | 7 |
| 5.3 | 测试函数 3 | 8 |
| 5.4 | 测试函数 4 | 8 |
| 6 | 分析总结 | 9 |

1 问题描述

在这项作业中, 我主要依据杨老师讲义 [1] 上的算法完成了以下几项工作:

- 实现基于 Wolfe-Powell 准则的非精确一维步长搜索算法.
- 基于非精确一维步长搜索, 手动实现牛顿法.
- 构造了多个函数, 例如 Rosenbrock 函数, 应用算法在不同初值下求解无约束最优化问题, 并分析不同初值点对结果的影响.

2 算法原理

2.1 基于 Wolfe-Powell 准则的非精确一维搜索算法

1. 给定初始一维搜索区间 $[0, \bar{\alpha}]$, 以及 $\rho \in (0, 1/2), \sigma \in (\rho, 1)$, 计算 $\varphi_0 = \varphi(0) = f(x^{(k)}), \varphi'_0 = \varphi'(0) = \nabla f(x^{(k)})^T d^{(k)}$. 并令 $a_1 = 0, a_2 = \bar{\alpha}, \varphi_1 = \varphi_0, \varphi'_1 = \varphi'_0$. 选取适当的 $\alpha \in (a_1, a_2)$.
2. 计算 $\varphi = \varphi(\alpha) = f(x^{(k)} + \alpha d^{(k)})$. 若 $\varphi(\alpha) \leq \varphi(0) + \rho\alpha\varphi'(0)$, 则转到第 2 步. 否则, 由 $\varphi_1, \varphi'_1, \varphi$ 构造两点二次插值多项式 $p^{(1)}(t)$, 并得其极小点

$$\hat{\alpha} = a_1 + \frac{1}{2} \frac{(a_1 - \alpha)^2 \varphi'_1}{(\varphi_1 - \varphi) - (a_1 - \alpha)\varphi'_1}$$

于是置 $a_2 = \alpha, \alpha = \hat{\alpha}$, 重复第 1 步.

3. 计算 $\varphi' = \varphi'(\alpha) = \nabla f(x^{(k)} + \alpha d^{(k)})^T d^{(k)}$. 若 $\varphi'(\alpha) \geq \sigma\varphi'(0)$, 则输出 $\alpha_k = \alpha$, 并停止搜索. 否则, 由 $\varphi, \varphi', \varphi'_1$ 构造两点二次插值多项式 $p^{(2)}(t)$, 并得其极小点

$$\hat{\alpha} = \alpha - \frac{(a_1 - \alpha)\varphi'}{\varphi'_1 - \varphi'}$$

于是置 $a_1 = \alpha, \alpha = \hat{\alpha}, \varphi_1 = \varphi, \varphi'_1 = \varphi'$, 返回第 1 步.

Wolfe(1968)-Powell(1976) conditions:

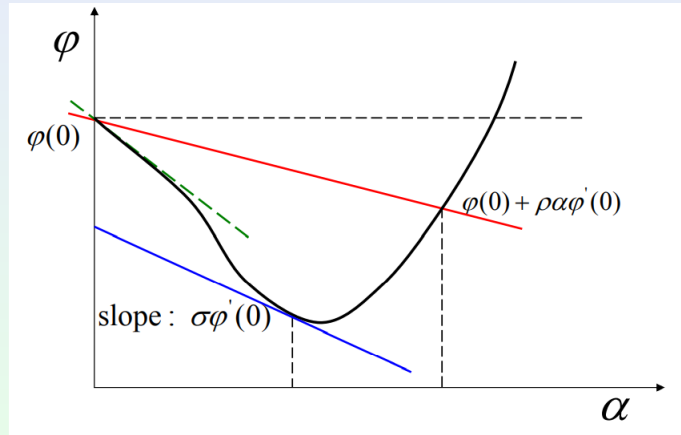


Figure 1: Wolfe-Powell

2.2 牛顿法

对于如下的无约束优化问题

$$\min_x f(x)$$

牛顿法的一般迭代格式

1. 初始化: 选取适当的初始点 $x^{(0)}$, 令 $k := 0$
2. 计算搜索方向: $d^{(k)} = -\nabla^2 f(x^{(k)})^{-1} \nabla f(x^{(k)})$
3. 确定步长因子: 采用非精确的一维搜索确定步长因子 α_k
4. 更新迭代点: 令 $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$. 置 $k := k + 1$, 返回第 2 步.

详细源代码见文件 [main.py](#).

3 数据集说明

选取一些测试函数和初值点进行测试.

4 程序输入输出说明

手动输入测试函数的表达式, 初值点, 只需对原程序稍作修改即可. 例如我的程序中选取了 4 个测试函数

测试函数 1:

$$100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 100(x_3 - x_2^2)^2 \quad (1)$$

测试函数 2:

$$(1 - x_1)^2 + 100(x_2 - x_1^2)^2 \quad (2)$$

测试函数 3:

$$1 + (1 - x_1)^2 \quad (3)$$

测试函数 4:

$$1 + (1 - \sin(x_1))^2 \quad (4)$$

定义测试函数如下:

```
def f_test1(init):  
    return 100*np.power(init[1]-np.power(init[0], 2), 2)+np.power(1-init[0]  
                                                                    ], 2)+100*np.power(init[2]-np.  
                                                                    power(init[1], 2), 2)  
  
def f_test2(init):  
    return 100*np.power(init[1]-np.power(init[0], 2), 2)+np.power(1-init[0]  
                                                                    ], 2)  
  
def f_test3(init):  
    return 1+np.power(1-init[0], 2)  
  
def f_test4(init):  
    return 1+np.power(np.array([1])-np.sin(init[0]), 2)
```

其中函数的参数 `init` 表示未知数向量, 存储在 `numpy.array` 中, 返回的即是函数的表达式, 后面可以用 `autograd` 包中的 `jacobian` 对其求梯度.

定义好函数后, 为了后面求 Hessian 矩阵方便, 还需要对每一个测试函数定义一个函数 `hassiantest` 用来求它们的 Hessian 矩阵. 其实这些 `hassiantest` 函数差别不大, 只需修改测试函数的表达式和相应变量的维度即可. 这里是使用了 `sympy` 包, 但是它求得的 Hessian 阵接受的是不可迭代型数据, 需要使用 `lambdify` 将其转化为可迭代数据, 因此这里的操作比较繁琐, 短时间内我没有想到好的优化办法. 具体定义 `hassiantest` 函数的方式如下:

```
def hessian_test1(vec):
    x_1, x_2, x_3 = sp.symbols('x_1, x_2, x_3')
    f = (1-x_1)**2 + 100*(x_2-x_1**2)**2+100*(x_3-x_2**2)**2
    v = list(ordered(f.free_symbols))
    hess = hessian(f, v)
    lam_f = lambdify(v, hess, 'numpy')
    return lam_f(vec[0], vec[1], vec[2])

def hessian_test2(vec):
    x_1, x_2 = sp.symbols('x_1, x_2')
    f = (1-x_1)**2+100*(x_2-x_1**2)**2
    v = list(ordered(f.free_symbols))
    hess = hessian(f, v)
    lam_f = lambdify(v, hess, 'numpy')
    return lam_f(vec[0], vec[1])

def hessian_test3(vec):
    x_1 = sp.symbols('x_1')
    f = 1+(1-x_1)**2
    v = list(ordered(f.free_symbols))
    hess = hessian(f, v)
    lam_f = lambdify(v, hess, 'numpy')
    return lam_f(vec[0])

def hessian_test4(vec):
    x_1 = sp.symbols('x_1')
    f = 1+(1-sp.sin(x_1))**2
    v = list(ordered(f.free_symbols))
    hess = hessian(f, v)
    lam_f = lambdify(v, hess, 'numpy')
```

```
return lam_f(vec[0])
```

这些定义好以后, 可以单独对每一个函数进行测试, 也可以同时测试. 同时测试的方法如下

```
f_test = [f_test1, f_test2, f_test3, f_test4]
hessian_test = [hessian_test1, hessian_test2, hessian_test3,
                hessian_test4]
inp = [np.array([1.0001, 1.0001, 1.000001]), np.array([1.2, 1.2]), np.
        array([1.2]), np.array([1.2])]
f_str = ['(1-x1)^2+100*(x2-x1^2)^2+100*(x3-x2^2)^2', '(1-x1)^2+100*(x2
        -x1^2)^2', '1+(1-x1)^2', '1+(1-
        sin(x1))^2']

for i in range(4):
    print('function:' + f_str[i])
    print('start:' + str(inp[i]))
    print('f(start):' + str(f_test[i](inp[i])))
    best = newton(f_test[i], inp[i], hessian_test3)
    print('optimal point:' + str(best[0]))
    print('f(optimal):' + str(best[1]))
```

利用 for 循环进行同时测试, 也可以不用循环, 取列表中每一个测试函数单独测试.

5 程序测试结果

5.1 测试函数 1

首先对三维的 Rosenbrock 函数进行测试

$$100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 100(x_3 - x_2^2)^2$$

取初值 (1.0001, 1.0001, 1.0001), 初值点很接近最优值点 (1, 1, 1), 因此很快达到要求误差范围内

```
Run: main x
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function: (1-x1)^2+100*(x2-x1^2)^2+100*(x3-x2^2)^2
start: [1.0001 1.0001 1.0001]
f(start): 2.010400019997126e-06
term: 0
optimal point: [1.00000006 1.00000007 1.00000007]
f(optimal): 6.411688829595411e-13
Process finished with exit code 0
```

下面再取初值为 (1.2, 1.2, 1.2), 得到的结果相比上面的结果稍微差一些, 但也能很快的接近最优值

```
Run: main x
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function: (1-x1)^2+100*(x2-x1^2)^2+100*(x3-x2^2)^2
start: [1.2 1.2 1.2]
f(start): 11.559999999999999
term: 0
optimal point: [1.00000266 1.0000053 1.00001054]
f(optimal): 7.570913695446552e-12
Process finished with exit code 0
```

当初值点选取的很远时, 例如 (3, 3, 3), 程序就不能正常运行了, 不能正常收敛. 因此初值点的选取很关键.

```
Run: main x
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function: (1-x1)^2+100*(x2-x1^2)^2+100*(x3-x2^2)^2
start: [3 3 3]
f(start): 7204
Traceback (most recent call last):
  File "C:\Users\34659\anaconda3\envs\ORproject2\lib\site-packages\autograd\tracer.py", line 118, in new_box
    return box_type_mappings[type(value)](value, trace, node)
KeyError: <class 'numpy.int32'>
```

5.2 测试函数 2

再对二维的 Rosenbrock 函数进行测试

$$(1 - x_1)^2 + 100(x_2 - x_1^2)^2$$

类似取初值为 (1.2, 1.2), 得到结果如下

```
Run: main x
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function: (1-x1)^2+100*(x2-x1^2)^2
start: [1.2 1.2]
f(start): 5.8
term: 0
optimal point: [1.00000191 1.00000375]
f(optimal): 3.995594210472431e-12
```

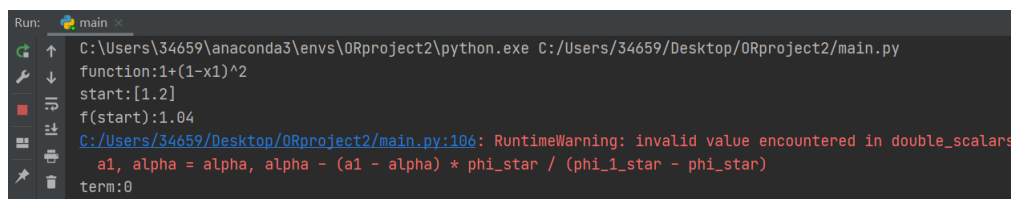

初值点对结果的影响与测试函数 1 类似.

5.3 测试函数 3

测试函数 3 为

$$1 + (1 - x_1)^2$$

首先取初值为 1.2, 程序报错



```
Run: main x
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function:1+(1-x1)^2
start:[1.2]
f(start):1.04
C:/Users/34659/Desktop/ORproject2/main.py:106: RuntimeWarning: invalid value encountered in double_scalars
  a1, alpha = alpha, alpha - (a1 - alpha) * phi_star / (phi_1_star - phi_star)
term:0
```

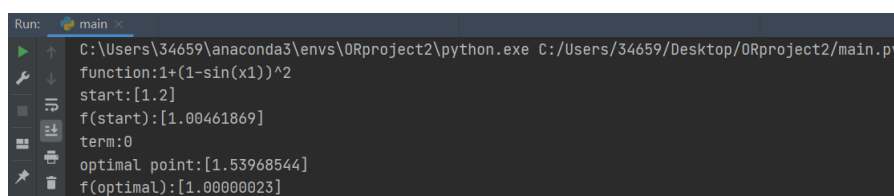
应该是在一维搜索的时候, 插值点出现了分子分母全为 0 的情况? 这可能是 Wolfe-Powell 搜索需要优化的地方.

5.4 测试函数 4

测试函数 4 为

$$1 + (1 - \sin(x_1))^2$$

首先取初值为 1.2, 得到结果如下



```
Run: main x
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function:1+(1-sin(x1))^2
start:[1.2]
f(start):[1.00461869]
term:0
optimal point:[1.53968544]
f(optimal):[1.00000023]
```

能很快接近最优点.

将初值点改为 2 时, 程序报错

```
Run: main ×
C:\Users\34659\anaconda3\envs\ORproject2\python.exe C:/Users/34659/Desktop/ORproject2/main.py
function:1+(1-sin(x1))^2
start:[2]
f(start):[1.00822696]
Traceback (most recent call last):
  File "C:\Users\34659\anaconda3\envs\ORproject2\lib\site-packages\autograd\tracer.py", line 118, in new_box
    return box_type_mappings[type(value)](value, trace, node)
KeyError: <class 'numpy.int32'>
```

6 分析总结

由上面的测试结果可以看出, 牛顿法收敛速度较快, 对于正定二次函数一步迭代即达最优解. 但牛顿法也有很多缺点

- 每一步都需要求解目标函数的 Hessian 矩阵的逆矩阵, 计算比较复杂
- 牛顿法是局部收敛的, 当初始点选择不当时, 往往导致不收敛
- 二阶 Hessian 矩阵必须可逆, 否则算法进行困难

在 Hessian 矩阵非正定, 甚至奇异的情况可能需要考虑拟牛顿法.

本次实验让我认识到了, 算法在实际应用中会遇到各种各样的问题, 如何选取初值点, 在什么情况下算法可以收敛等等, 这些都是值得思考的问题. 在未来的学习中, 需要不断深入思考这些问题, 不断尝试代码的优化和改进.

References

- [1] 杨周旺: 运筹学讲义;