运筹学课程实验实验报告

一、最小成本循环流的强多项式算法

二、动态规划算法

三、非精确一维线搜索Wolfe-Powell算法

（一）实验要求

1. 实现基于 Wolfe-Powell 准则的非精确一维步长搜索算法.
2. 基于非精确一维步长搜索，从牛顿法、拟牛顿类方法 (DFP/BFGS)、共轭梯度法中选择一种算法，手动实现.
3. 本作业需至少构造一个函数 (例如 Rosenbrock 函数)，应用算法在 不同初始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。

Rosenbrock函数：

$$f(x)=\sum_{i=1}^{d-1}\left[100\left(x_{i+1}-x_i^2\right)^2+\left(x_i-1\right)^2\right], \quad x \in \mathbb{R}^d$$

（二）问题描述

本实验选取牛顿法作为非精确一维步长搜索算法的背景算法。

非精确一维步长搜索算法旨在通过有限的计算获得恰当的步长因子，结合牛顿法给出的迭代搜索方向，对于最优化问题进行迭代求解。

牛顿法的一般算法如下：

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:

(0)初始化：选取适当的初始点$x^{(0)}$,令$k:=0$.

(1)计算搜索方向：$d^{(k)}=-\nabla^2 f\left(x^{(k)}\right)^{-1} \nabla f\left(x^{(k)}\right)$.

(2)确定步长因子：采用非精确的一维搜索确定步长因子α_k.

(3)更新迭代点：令$x^{(k+1)}=x^{(k)}+\alpha_k d^{(k)}$.置$k:=k+1$,返回第$(1)$步.

（三）算法原理

1. 理论说明

1. Wolfe-Powell条件

非精确一维搜索：找出满足某些适当条件的粗略近似解作为步长，提升算法的整体计算效率。
Wolfe-Powell conditions:

$$\varphi(\alpha) \leq \varphi(0)+\rho\alpha\varphi'(0)$$
$$\varphi'(\alpha) \geq \sigma\varphi'(0)$$

其中 $\rho \in (0,1/2), \sigma \in (\rho,1)$ 是固定参数。

2. Wolfe-Powell条件的实现

由于只通过上述约束条件，仍无法直接准确获得满足Wolfe-Powell条件的α值。所以仍然通过简单的迭代方法来获得满足Wolfe-Powell条件的值。

算法[1]如下：

(0) 给定初始一维搜索区间$[0,\bar{\alpha}]$，以及$\rho \in \left(0,\frac{1}{2}\right), \sigma \in (\rho,1)$. 计算$\varphi_0=\varphi(0)=f\left(x^{(k)}\right), \varphi_0'=\varphi'(0)=\nabla f\left(x^{(k)}\right)^T d^{(k)}$，并令 $a_1=0$， $a_2=\bar{\alpha}$，$\varphi_1=\varphi_0, \varphi_1'=\varphi_0'$. 选取适当的$\alpha \in (a_1,a_2)$.

(1) 计算$\varphi=\varphi(\alpha)=f\left(x^{(k)}+\alpha d^{(k)}\right)$. 若$\varphi(\alpha) \leq \varphi(0)+\rho\alpha\varphi'(0)$, 则转到 第(2)步。否则, 由$\varphi_1, \varphi_1', \varphi$构造二次插值多项式$p^{(1)}(t)$, 并得其极小 点$\hat{\alpha}$. 令$a_2=\alpha$, $\alpha=\hat{\alpha}$, 重复第(1)步.

(2) 计算$\varphi'=\varphi'(\alpha)=\nabla f\left(x^k+\alpha d^{(k)}\right)^T d^{(k)}$. 若$\varphi'(\alpha) \geq \sigma\varphi'(0)$, 则输出$\alpha_k=\alpha$, 并停止搜索. 否则由$\varphi$， φ', φ_1'构造两点二次插值多项式$p^{(2)}(t)$, 并求得极小点$\hat{\alpha}$. 令$a_1=\alpha$， $\alpha=\hat{\alpha}$, 返回第(1)步.

2. 编程实现

1. Newton法实现

```
def newton(y, x0):
    # 用牛顿法求解无约束问题
    # x0是初始点，fun，gfun和hess分别是目标函数值，梯度，海森矩阵的函数
    maxk = 500
    k = 0
    epsilon = 1e-5

```

```
    W = np.zeros((d, maxk))

    while k < maxk:
        W[:, k] = x0[:, 0]
        gk = get_gfun(y,x0)
        Gk = get_hess(y,x0)
        dk = -1.0 * np.linalg.solve(Gk, gk)
        if np.linalg.norm(dk) < epsilon:
            break

        alpha_k = wolfe_powell(y,x0,dk,2)
        x0 = x0 + alpha_k*dk
        k += 1

    W = W[:, 0:k + 1]  # 记录迭代点
    return x0, get_fun(y,x0), k, W
```

- 说明：
  - y是sympy格式的算式，x0是np.ndarray格式的$(d,1)$的array。
  - get_gfun，get_hess分别是已知y求x0点处的梯度向量和海森矩阵的函数，其分别在另外两个文件中定义。
  - 牛顿法的精髓在于搜索方向d_k为14行，满足$G_k \cdot d_k = g_k$的搜索方向。调用Wolfe-Powell函数确定步长因子。更新迭代点。
  - 迭代截止条件为15行，如果迭代方向的范数过于小，则停止迭代。

2. Wolfe-Powell条件实现

```
def judge(fai, fai0, fai1, p_fai0, p_fai1, xk, dk, y, rho, a1, sigma, a2, alpha):
    while fai > fai0 + rho*alpha*p_fai0:

        alpha_head = a1+0.5*(a1-alpha)**2*p_fai1/((fai1-fai)-(a1-alpha)*p_fai1)
        a2 = alpha
        alpha = alpha_head
        fai = get_fun(y,xk + alpha * dk)

    p_fai = np.dot(get_gfun(y,xk+alpha*dk).T, dk)[0][0]

    if p_fai< sigma*p_fai0:
        alpha_head = alpha - (a1-alpha)*p_fai/(p_fai1-p_fai)
        a1 = alpha
        alpha = alpha_head
        fai1=fai
        p_fai1 = p_fai
```

```
        return judge(fai, fai0, fai1, p_fai0, p_fai1, xk, dk, y, rho, a1, sigma, a2, alpha)
    else:
        return alpha

def wolfe_powell(y,xk,dk,max_value,rho = 0.1,sigma = 0.4):
    a1 = 0
    a2 = max_value
    fai0 = get_fun(y,xk)
    fai1 = fai0
    p_fai0 = np.dot(get_gfun(y,xk).T,dk)[0][0]
    p_fai1 = p_fai0

    alpha = (a1 + a2) / 2
    fai = get_fun(y,xk + alpha * dk)

    return judge(fai, fai0, fai1, p_fai0, p_fai1, xk, dk, y, rho, a1, sigma, a2, alpha)

```

- 说明：
  - 主函数是下方的wolfe_powell函数。其调用上方的judge函数。
  - wolfe_powell函数进行初始化，第一步选取的$\alpha = \frac{\alpha_0+\hat{\alpha}}{2}$,之后调用judge函数获取满足wolfe_powell条件的$\hat{\alpha}$
  - 由于该基于二阶插值极小点迭代的算法需要在第一步和第二步之间来回跳转，所以需要进行迭代求解，利用judge函数进行迭代。
  - judge函数：
    - while部分对应算法[1]的（1），if部分对应算法[1]的（2）
    - 4行构造$p^{(1)}(t)$的极小点$\hat{\alpha} = a_1 + \frac{1}{2}\frac{(a_1-\alpha)^2\varphi_1'}{(\varphi_1-\varphi)-(a_1-\alpha)\varphi_1'}$
    - 12行构造$p^{(2)}(t)$的极小点$\hat{\alpha} = \alpha - \frac{(a_1-\alpha)\varphi'}{\varphi_1'-\varphi'}$
    - 其余部分进行更新或者跳转

（四）数据集说明

1. Rosenbrock函数的实现

利用sympy库对该数学公式进行计算，其实现的结果保存在Rosenbrock.py文件中。

代码如下：

```
# 可自由定义
d=5

x_list=[]
for i in range(1,d+1):
    exec("x{} = symbols('x{}')".format(i,i),globals())
    exec("x_list.append(x{})".format(i))

y=0
for i in range(1,d):
    exec("y += (x{}-1)**2".format(i),globals())
    exec("y += 100*(x{}-x{}**2)**2".format(i+1,i),globals())
```

- 说明：
    - 由于d的数量未知，故采用exec格式来确定变量的个数和最后Rosenbrock函数的实现。

2. 辅助函数get_gfun()和get_hess()函数的实现

辅助函数get_gfun()和get_hess()函数，对于输入的形式化函数y和指定点x0进行操作，最终获得梯度向量$w \in \mathcal{R}^{d\times 1}$和海森矩阵$G \in \mathcal{R}^{d\times d}$.

代码如下：

```
def get_gfun(expression, position):
    length = len(position)
    position = position.flatten()
    gradient = np.zeros(length)
    # get the variables
    variables = list(expression.free_symbols)
    # construct the evaluation list
    value = [(variables[k], position[k]) for k in range(length)]
    for k in range(length):
        gradient[k] = expression.diff(variables[k]).subs(value)
    return np.array(gradient).reshape(-1,1)

def get_hess(expression, position):
    length = len(position)
    position = position.flatten()
    hess = np.zeros((length,length))
    # get the variables
    variables = list(expression.free_symbols)
    # construct the evaluation list
    value = [(variables[k], position[k]) for k in range(length)]
    for k in range(length):
        for j in range(length):
```

```
            hess[k][j] = expression.diff(variables[k],variables[j]).subs(value)
    return np.matrix(hess)
```

- 说明：
  - flatten操作用于展平数组，获得一个点值列表。
  - value数组确定变量和取值的一一对应关系。
  - diff函数用来求差分，一个参数对应梯度，两个参数对应海森矩阵。

（五）数据输入、输出

1. 程序输入

对于Rosenbrock函数，只需修改Rosenbrock.py配置文件中d的值即可。

2. 程序输出

程序会输出最终的迭代点和最后的迭代结果。根据不同的要求，可对newton函数返回的结果加以利用并进行测试。

（六）程序测试结果

1. 简单数值测试

取$d=5$，简单测试结果如下：

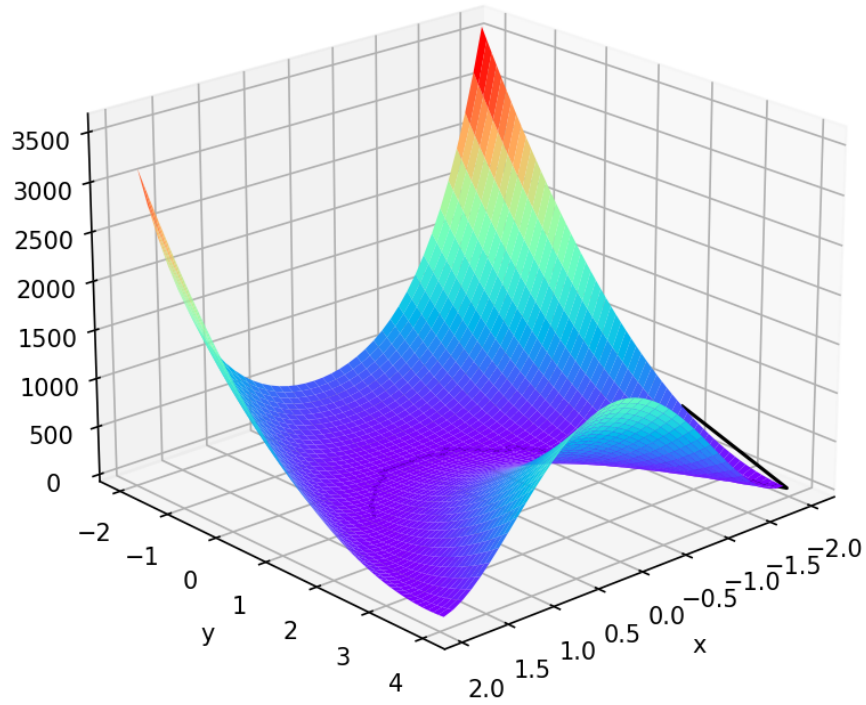
```
----------
初始值为[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
最优解的值为[[0.999999997570125]
 [0.999999999414002]
 [0.999999998812579]
 [0.999999999709196]
 [0.999999994934602]]
最小值为1.22998717800235E-17
迭代次数为17
----------
初始值为[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

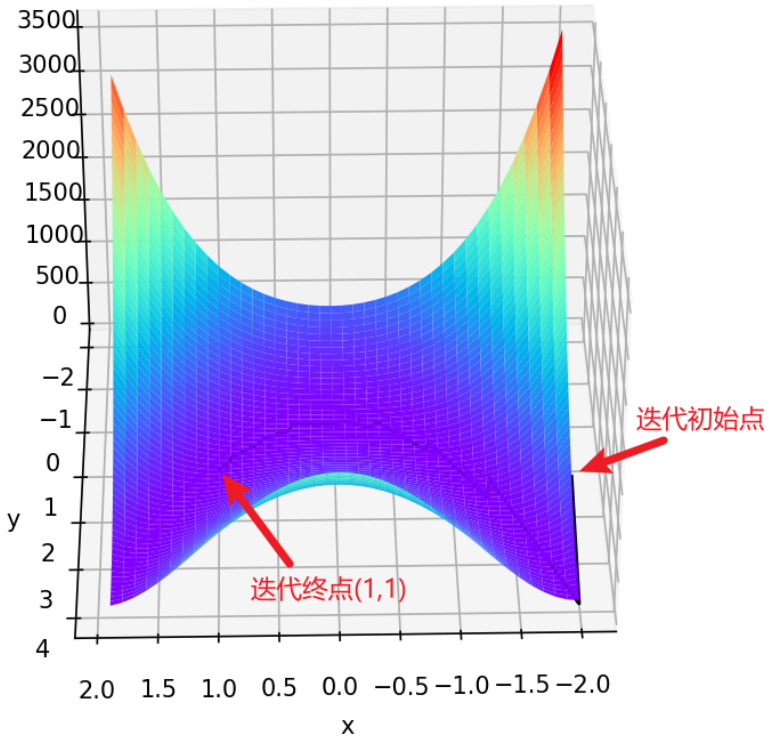
```
最优解的值为[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
最小值为0
迭代次数为0
----------
初始值为[[-2]
 [-1]
 [ 0]
 [ 1]
 [ 2]]
最优解的值为[[0.999999788826667]
 [0.999999950009533]
 [0.999999898058807]
 [0.999999975272156]
 [0.999999549641053]]
最小值为1.42294400935099E-13
迭代次数为24
----------
初始值为[[-20]
 [ -1]
 [  0]
 [  1]
 [  2]]
最优解的值为[[0.999999997166308]
 [0.999999999319434]
 [0.999999994062276]
 [0.999999999662508]
 [0.999999998619015]]
最小值为1.83792721065816E-17
迭代次数为28
```

运算结果相同，但显然运算时间和迭代次数有较大差异。

2. 数据可视化

仅以二维的情况为例(有两个角度)：





图中的线条即为迭代过程中迭代点的变化情况。

（七）分析总结

关于wolfe_powell算法，有以下分析：

1. 鉴于牛顿法不能保证海森矩阵的正定性，所以在进行一些较大的值的测试时，会发生梯度爆炸的情况，算法终止，或者迭代达到最深，算法被迫中止；
2. 上述缺陷可以通过拟牛顿法或共轭梯度法的相关措施来进行弥补，但由于已经写完整个实验，所以不在进行重写；
3. Wolfe-Powell算法的精髓在于确定步长因子的两个约束条件进行限制，所以通过二次插值对于小于或者大于约束范围的可能的步长因子进行变换，从而获得合适的步长因子；
4. 由于计算步长因子的可能性与超参数ρ,σ有关，且迭代的步数与数据和函数有很大相关性，其算法复杂度很难分析得到，但可以大致确定为多项式时间的复杂度。

[1] 运筹学课程实验PPT20页。 ↩ ↩ ↩