基于最速下降法的平面选址问题应用研究

研究背景：

平面选址问题是运筹学中的一个经典问题。最早的选址问题是由经济学家 Alfred Weber 于 1909 年提出的,他所考虑的选址问题是确定一个仓库位置,从而使仓库与各处客户之间总的运输距离最短,这就是著名的 Weber 问题。选址问题在现实生活中有着广泛的应用背景,系统工程、现代物流、金融经济、甚至军事中都有着非常广泛的应用,如银行、超市、急救中心、消防站、垃圾处理中心、物流中心、导弹仓库的选址等。选址是最重要的长期决策之一,选址的好坏直接影响到企业的成本,人民生活的便利程度,战争的成败等;好的选址可以为企业降低服务成本,提高服务质量、服务效率,扩大利润和市场份额等,进而影响到企业利润和市场竞争力,甚至决定了企业的命运;差的选址往往会带来很大的不便和损失,甚至是灾难,所以,选址问题的研究有着重大的经济、社会和军事意义。

研究内容：

本文仅研究最常见的一种无约束平面选址问题,即二维空间的极值最优化问题:在平面上给定 n 个位置点 $P_i(x_i, y_i)(i=1,2,\cdots n)$，现要确定选址位置点 $P(x, y)$，使点 $P(x, y)$ 到平面上 n 个点的距离之和最小,即：

$$\min D(x,y)=\sum_{i=1}^{n}\sqrt{(x-x_i)^2+(y-y_i)^2}$$

对于无约束优化问题 $\min D(x,y)$。

将文章中的实例的数据带入到上述模型中，即可得到如下优化模型：

$$\min D(x,y)=\sqrt{(x-10)^2+(y-4)^2}+\sqrt{(x-2)^2+(y-15)^2}$$

$$+\sqrt{(x+4)^2+(y-12)^2}+\sqrt{(x+5)^2+y^2}+\sqrt{(x-6)^2+(y+3)^2}$$

由数学知识可知：求解 $y=\sqrt{x^2}$ 的极值问题，可以转化为先求解 $y=x^2$ 的极值问题（二者具有相同的极值点），之后再将该极值点带入到 $y=\sqrt{x^2}$ 中，就可以得到极值。同理，上述优化模型的等价模型可以转化为：

$$\min D(x,y)=(x-10)^2+(y-4)^2+(x-2)^2+(y-15)^2$$

$$+(x+4)^2+(y-12)^2+(x+5)^2+y^2+(x-6)^2+(y+3)^2$$

（1） 用最速下降法求解此模型的程序（此处仅列举一维搜索为二分法的情形）

首先，将下列程序保存，并命名为 bisection_search.py，作为线性搜索的子函数。

```python
import numpy as np
import sympy
import matplotlib.pyplot as plt
def secant_search(f,d,x):
    def Alpha(alpha):
        return f(x+alpha*d)
    def Alpha_d(a):
        alpha=sympy.Symbol('alpha',real=True)
        return sympy.diff(f(x+alpha*d),alpha,1).doit().subs(alpha,a)

    def eff(a, b, Theta_error):
        stepNum = 0
        while abs(b - a) > Theta_error:
            a1 = (a + b) / 2
            f1 = Alpha_d(a1)
            #print(a, b, f1)
            if f1 > 0:
                b = a1
            elif f1 < 0:
                a = a1
            else:
                print(a1, Alpha(a1))
            stepNum = stepNum + 1
        return ((b + a)/ 2)
    Theta_error = 0.3  # 初始值
    alpha= eff(0,5, Theta_error)
    return alpha
```

接着，在 pycharm 中新建一个名为 gradient_bisection.py 的文件，放入下面代码并运行。

```python
import numpy as np
import matplotlib.pyplot as plt
import bisection_search
from bisection_search import  secant_search
import math
```

```
def F1(x):

y=math.sqrt((x[0]-10)**2+(x[1]-4)**2)+math.sqrt((x[0]-2)**2+(x[1]-15)**2)+math.sqrt((x[0]+4)**2+(x[1]-12)**2)+math.sqrt((x[0]+5)**2+(x[1])**2)+math.sqrt((x[0]-6)**2+(x[1]+3)**2)
    return y
def F(x):

y=(x[0]-10)**2+(x[1]-4)**2+(x[0]-2)**2+(x[1]-15)**2+(x[0]+4)**2+(x[1]-12)**2+(x[0]+5)**2+x[1]**2+(x[0]-6)**2+(x[1]+3)**2
    return y

def g(x):
    return np.array([10*x[0]- 18,10*x[1]-56])

def steepest(x0):
    print('初始点为:')
    print(x0)
    imax =10
    W=np.zeros((2,imax))
    W_d=np.zeros((2,imax))
    W[:,0] = x0
    i = 1
    x = x0
    grad = g(x)
    W_d[:,0] = grad
    delta = sum(grad**2)  # 初始误差
    while i<imax and delta>10**(-5):
        p=-g(x)
        x0=x
        alpha=secant_search(F,p,x)
        #print(alpha)
        x=x+alpha*p
        #print(x)
```

```
        W[:, i]=x
        grad=g(x)
        W_d[:,  i]=grad
        #print(grad)
        delta=sum(grad**2)
        i=i+1
    print("迭代次数为:",i-1)
    print("近似最优解为:")
    print(x)
    W=W[:,0:i]   # 记录迭代点
    W_d= W_d[:,  0:i]   #记录迭代梯度
    return W,W_d,x

x0 = np.array([0,0])
W,W_d,x=steepest(x0)
print('选址中心位置坐标到各个居民区的总距离为:',F1(x))
```

实验结果：

```
D:\Python\Python36\python.exe C:/Users/lenovo/PycharmProjects/untitled3/gradient_bisection.py
初始点为:
[0 0]
迭代次数为: 7
近似最优解为:
[1.79995686 5.59986578]
选址中心位置坐标到各个居民区的总距离为: 44.773773043717014
```

之后，将梯度下降法和不同的一维搜索方法结合。具体程序放在文件夹里。下面是不同一维搜索方法的结果：

表 1 不同一维搜索方法的结果

方法	迭代次数	最优解	目标函数值
梯度+二分法	7	[1.79995686 5.59986578]	44.7737730
梯度+黄金分割	1	[1.79997265 5.59991491]	44.7737723
梯度+修正斐波那契法	6	[1.79991404 5.59973256]	44.7737750
梯度+牛顿	1	[1.80000000 5.60000000]	44.7737711
梯度+割线	1	[1.80000000 5.60000000]	44.7737711

（2） 用牛顿法求解此模型的程序

第一种程序：（此程序从书上例题和梯度下降法的程序启发而来）

```
import numpy as np
import matplotlib.pyplot as plt
import math
from mpl_toolkits.mplot3d import Axes3D as ax3
def f1(x):

y=math.sqrt((x[0]-10)**2+(x[1]-4)**2)+math.sqrt((x[0]-2)**2+(x[1]-15)**2)+math.sqrt((x[0]+4)**2+(x[1]-12)**2)+math.sqrt((x[0]+5)**2+(x[1])**2)+math.sqrt((x[0]-6)**2+(x[1]+3)**2)
    return y
def f11(x,y):

z=np.sqrt((x-10)**2+(y-4)**2)+np.sqrt((x-2)**2+(y-15)**2)+np.sqrt((x+4)**2+(y-12)**2)+np.sqrt((x+5)**2+(y)**2)+np.sqrt((x-6)**2+(y+3)**2)
    return z
def f(x):
    return
(x[0]-10)**2+(x[1]-4)**2+(x[0]-2)**2+(x[1]-15)**2+(x[0]+4)**2+(x[1]-12)**2+(x[0]+5)**2+x[1]**2+(x[0]-6)**2+(x[1]+3)**2
def jacobian(x):
    return np.array([10*x[0]- 18,10*x[1]-56])
def hessian(x):
    return np.array([[10,0],[0,10]])
def newton(x0):
    print('初始点为:')
    print(x0)
    W=np.zeros((len(x0),10**3))
    i=1
    imax=1000
    W[:,0] = x0
    x=x0
    delta= 1
    while i<imax and delta>10**(-6):
        p = -np.dot(np.linalg.inv(hessian(x)),jacobian(x))
        x0 = x
        x = x+p
        W[:,i] = x
        delta = sum((x-x0)**2)
        print('第',i,'次迭代结果:')
        print(x)
        print(f1(x))
        i=i+1
    W=W[:,0:i]  # 记录迭代点
```

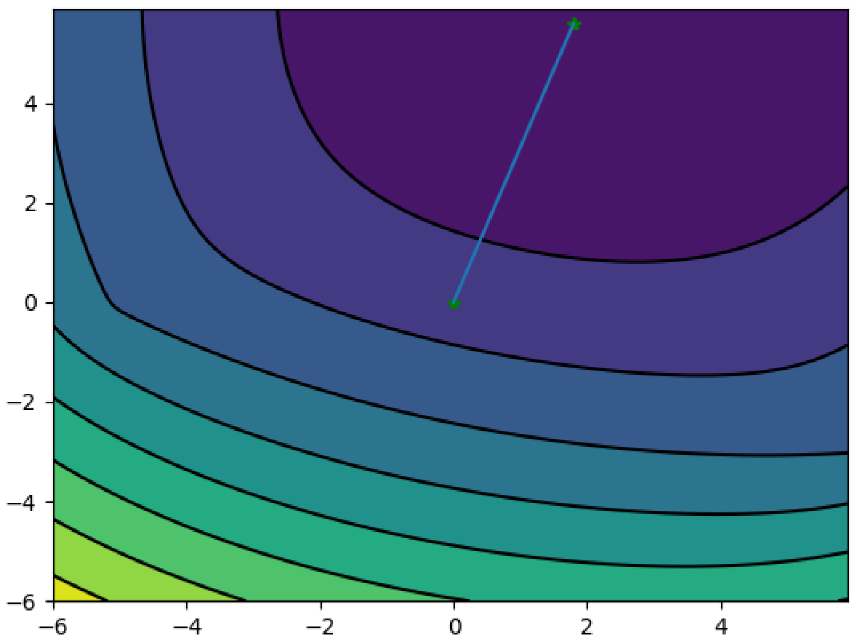
```
    return W

x0 = np.array([0,0])
W=newton(x0)

x=np.arange(-6,6,0.1)
y=x
X,Y=np.meshgrid(x, y)
C=plt.contour(X,Y,f11(X,Y),8,colors='black')  #生成等值线图
plt.contourf(X,Y,f11(X,Y),8)
plt.plot(W[0,:],W[1,:],'g*',W[0,:],W[1,:]) # 画出迭代点收敛的轨迹
plt.show()
```

实验结果：

```
初始点为:
[0 0]
第 1 次迭代结果:
[1.8 5.6]
44.77377111061479
第 2 次迭代结果:
[1.8 5.6]
44.77377111061479
```



第二种程序：

```
import numpy
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D as ax3
#Newton 寻优，二维实验
def NE(x0,y0,N,E):
    X1=[];X2=[];Y=[];Y_d=[]
    n = 1
    ee = g(x0,y0)
    e=(ee[0,0]**2+ee[1,0]**2)**0.5
    X1.append(x0)
    X2.append(y0)
    Y.append(f(x0,y0))
    Y_d.append(e)
    print('第%d 次迭代：e=%s' % (n, e))
    while n<N and e>E:
        n=n+1
        d=-numpy.linalg.solve(G(x0,y0),g(x0,y0))
        #d=-numpy.dot(numpy.linalg.pinv(G(x0,y0)),g(x0,y0))
        x0=x0+d[0,0]
        y0=y0+d[1,0]
        ee = g(x0, y0)
        e = (ee[0,0] ** 2 + ee[1,0] ** 2) ** 0.5
        X1.append(x0)
        X2.append(y0)
        Y.append(f(x0, y0))
        Y_d.append(e)
        print('第%d 次迭代：e=%s'%(n,e))
    return X1,X2,Y,Y_d
if __name__=='__main__':
    f1= lambda x,y:
math.sqrt((x-10)**2+(y-4)**2)+math.sqrt((x-2)**2+(y-15)**2)+math.sqrt
((x+4)**2+(y-12)**2)+math.sqrt((x+5)**2+(y)**2)+math.sqrt((x-6)**2+(y
```

```
+3)**2)
    f11=lambda x, y:
numpy.sqrt((x-10)**2+(y-4)**2)+numpy.sqrt((x-2)**2+(y-15)**2)+numpy.sqrt((x+4)**2+(y-12)**2)+numpy.sqrt((x+5)**2+(y)**2)+numpy.sqrt((x-6)**2+(y+3)**2)
    f = lambda x,y:
(x-10)**2+(y-4)**2+(x-2)**2+(y-15)**2+(x+4)**2+(y-12)**2+(x+5)**2+y**2+(x-6)**2+(y+3)**2 #原函数
    g = lambda x,y: numpy.array([[10*x- 18],[10*y-56]])    #一阶导函数向量
    G = lambda x,y: numpy.array([[10,0],[0,10]])     #二阶导函数矩阵
    x0=0;y0=0
    N=10;E=10**(-6)
    X1,X2,Y,Y_d=NE(x0,y0,N,E)
    figure1 = plt.figure('3D')
    x = numpy.arange(-6, 6, 0.1)
    y = x
    [xx, yy] = numpy.meshgrid(x, y)
    zz = numpy.zeros(xx.shape)
    n = xx.shape[0]
    for i in range(n):
        for j in range(n):
            zz[i, j] = f(xx[i, j], yy[i, j])
    ax = ax3(figure1)
    ax.contour3D(xx, yy, zz, 15)
    ax.plot3D(X1, X2, Y, 'ro--')
    figure2 = plt.figure('2D')
    x = numpy.arange(-6, 6, 0.1)
    y = x
    X, Y = numpy.meshgrid(x, y)
    C = plt.contour(X, Y, f11(X, Y), 8, colors='black')  # 生成等值线图
    plt.contourf(X, Y, f11(X, Y), 8)
    plt.plot(X1, X2, 'g*', X1, X2)  # 画出迭代点收敛的轨迹
    plt.show()
```

实验结果：

```
D:\Python\Python36\python.exe C:/Users/lenovo/PycharmProjects/untitled3/nd.py
第1次迭代：e=58.82176467941097
第2次迭代：e=0.0
```

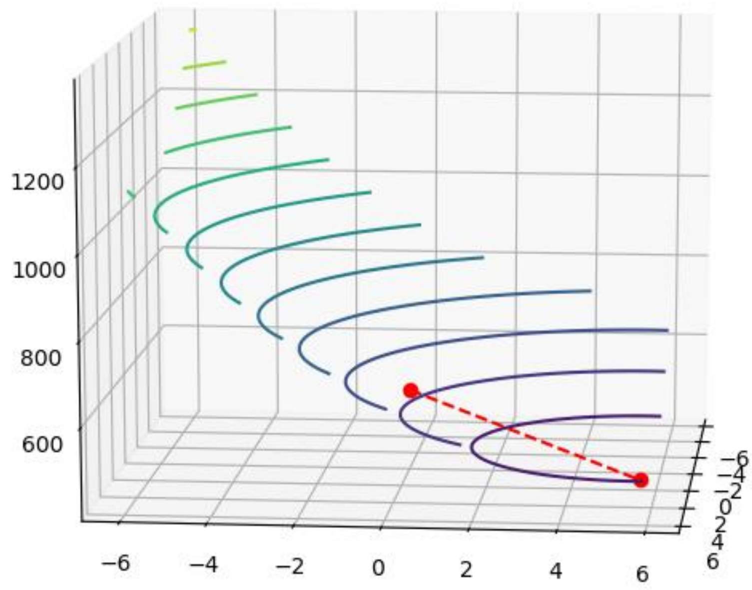


图 1 等价模型的 3D

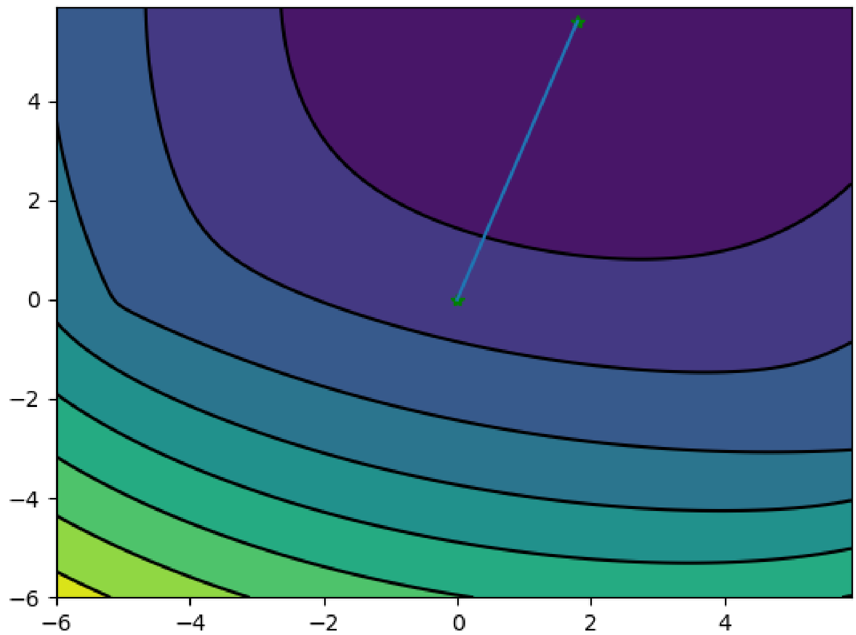


图 2 原模型的等值线图

以下是选择书上例题 10.3，对其分别使用梯度下降、牛顿法、共轭方向法（H-S 法、PR 法，FR 法）来计算目标函数的极小值。

目标函数为：

$$f(x_1,x_2,x_3)=\frac{3}{2}x_1^2+2x_2^2+\frac{3}{2}x_3^2+x_1x_3+2x_2x_3-3x_1-x_3$$

初始点为： $x^{(0)}=[0,0,0]^T$

将函数转化为二次型的形式为：

$$f(x)=\frac{1}{2}x^TQx^T-x^Tb$$

其中;

$$Q=\begin{bmatrix}3 & 0 & 1\\ 0 & 4 & 2\\ 1 & 2 & 4\end{bmatrix} \qquad b=\begin{bmatrix}3\\ 0\\ 1\end{bmatrix}$$

用不同方法求解该函数的程序如下（以下程序中涉及到的以为搜索选用的时黄金分割法）：

```
import sympy, numpy
import math
import matplotlib.pyplot as plt
import goldenOpt_search
from goldenOpt_search import  secant_search
# 共轭梯度法 FR、PRP, HS 三种格式
def CG_FR(x0, N, E, f, f_d):
    X = x0
    Y = []
    Y_d = []
    n = 1
    ee =f_d(x0)
    e = (ee[0] ** 2 + ee[1] ** 2) ** 0.5
    d = -f_d(x0)
    Y.append(f(x0)[0, 0])
    Y_d.append(e)
    a = sympy.Symbol('a', real=True)
    print('第%2s 次迭代：e=%f' % (n, e))
    while n < N and e > E:
        n = n + 1
        g1 = f_d(x0)
```

```
        a0 = secant_search(f, d, x0)
        print(a0)
        x0 = x0+d * a0
        X = numpy.c_[X, x0]
        Y.append(f(x0)[0, 0])
        ee = f_d(x0)
        e = math.pow(math.pow(ee[0, 0], 2) + math.pow(ee[1, 0], 2), 0.5)
        Y_d.append(e)
        g2 = f_d(x0)
        beta = (numpy.dot(g2.T, g2)) / numpy.dot(g1.T, g1)
        d =-g2 + beta * d
        print('第%2s 次迭代：e=%f' % (n, e))
    return X, Y, Y_d
def CG_PRP(x0, N, E, f, f_d):
    X = x0
    Y = []
    Y_d = []
    n = 1
    ee = f_d(x0)
    e = (ee[0] ** 2 + ee[1] ** 2) ** 0.5
    d = -f_d(x0)
    Y.append(f(x0)[0, 0])
    Y_d.append(e)
    a = sympy.Symbol('a', real=True)
    print('第%2s 次迭代：e=%f' % (n, e))
    while n < N and e > E:
        n = n + 1
        g1 = f_d(x0)
        a0 = secant_search(f, d, x0)
        print(a0)
        x0 = x0+d * a0
        X = numpy.c_[X, x0]
        Y.append(f(x0)[0, 0])
        ee = f_d(x0)
```

```
        e = math.pow(math.pow(ee[0, 0], 2) + math.pow(ee[1, 0], 2), 0.5)
        Y_d.append(e)
        g2 = f_d(x0)
        beta = (numpy.dot(g2.T, g2 - g1)) / numpy.dot(g1.T, g1)
        d = -f_d(x0) + beta * d
        print('第%2s 次迭代：e=%f' % (n, e))
    return X, Y, Y_d
def CG_HS(x0, N, E, f, f_d):
    X = x0
    Y = []
    Y_d = []
    n = 1
    ee = f_d(x0)
    e = (ee[0] ** 2 + ee[1] ** 2) ** 0.5
    d = -f_d(x0)
    Y.append(f(x0)[0, 0])
    Y_d.append(e)
    a = sympy.Symbol('a', real=True)
    print('第%2s 次迭代：e=%f' % (n, e))
    while n < N and e > E:
        n = n + 1
        g1 = f_d(x0)
        a0 = secant_search(f, d, x0)
        print(a0)
        x0 = x0+d * a0
        X = numpy.c_[X, x0]
        Y.append(f(x0)[0, 0])
        ee = f_d(x0)
        e = math.pow(math.pow(ee[0, 0], 2) + math.pow(ee[1, 0], 2), 0.5)
        Y_d.append(e)
        g2 = f_d(x0)
        beta = (numpy.dot(g2.T, g2 - g1)) / numpy.dot(d.T,g2-g1)
        d = -f_d(x0) + beta * d
        print('第%2s 次迭代：e=%f' % (n, e))
```

```
    return X, Y, Y_d
def SD(x0,Q,b,c,N,E):
    f = lambda x: 0.5 * (numpy.dot(numpy.dot(x.T, Q), x)) - numpy.dot(b.T, x) + c
    f_d = lambda x: numpy.dot(Q, x) - b
    X=x0;Y=[];Y_d=[]
    xx=sympy.symarray('xx',(2,1))
    n = 1
    ee = f_d(x0)
    e=math.pow(math.pow(ee[0,0],2)+math.pow(ee[1,0],2),0.5)
    Y.append(f(x0)[0,0]);Y_d.append(e)
    a=sympy.Symbol('a',real=True)
    print('第%d 次迭代：e=%d' % (n, e))
    while n<N and e>E:
        n=n+1
        d=-f_d(x0)
        a0 = secant_search(f, d, x0)
        print(a0)
        x0=x0-a0*f_d(x0)
        X=numpy.c_[X,x0]
        Y.append(f(x0)[0,0])
        ee = f_d(x0)
        e = math.pow(math.pow(ee[0,0],2)+math.pow(ee[1,0],2),0.5)
        Y_d.append(e)
        print('第%d 次迭代：e=%s'%(n,e))
    return X,Y,Y_d
def NDF(x0, N, E, f, f_d):
    X = x0;Y = [];Y_d = [];n = 1
    ee = f_d(x0)
    e = (ee[0] ** 2 + ee[1] ** 2) ** 0.5
    g= -f_d(x0)
    Y.append(f(x0)[0, 0])
    Y_d.append(e)
    print('第%2s 次迭代：e=%f' % (n, e))
```

```
    while n < N and e > E:
        n = n + 1
        g1 = f_d(x0)
        d =-numpy.dot(numpy.linalg.inv(G),g1)
        x0 = x0+d
        X = numpy.c_[X, x0]
        Y.append(f(x0)[0, 0])
        ee = f_d(x0)
        e = math.pow(math.pow(ee[0, 0], 2) + math.pow(ee[1, 0], 2), 0.5)
        Y_d.append(e)
        g2 = f_d(x0)
        beta = (numpy.dot(g2.T, g2 - g1)) / numpy.dot(g1.T, g1)
        d = -f_d(x0) + beta * d
        print('第%2s 次迭代：e=%f' % (n, e))
    return X, Y, Y_d
if __name__ == '__main__':
    G =numpy.array([[3, 0, 1],[0, 4, 2],[1, 2, 3]])
    b = numpy.array([[3], [0], [1]])
    x0 = numpy.array([[0], [0], [0]])
    c=0
    f = lambda x: 0.5 * (numpy.dot(numpy.dot(x.T, G), x)) - numpy.dot(b.T,
x)+c
    f_d = lambda x: numpy.dot(G, x) - b
    #x0 =x0+ numpy.random.rand(len(x0), 1) * 100
    x0 = x0
    N = 100;E = 10 ** (-2)
    print('共轭梯度 FR')
    X1, Y1, Y_d1 = CG_FR(x0, N, E, f, f_d)
    print(X1, Y1, Y_d1)
    print('共轭梯度 PR')
    X2, Y2, Y_d2 = CG_PRP(x0, N, E, f, f_d)
    print(X2, Y2, Y_d2)
    figure1 = plt.figure('trend')
    print('梯度下降法')
```

```
    X3, Y3, Y_d3 = SD(x0,G,b,c, N, E)
    print(X3, Y3, Y_d3)
    print('共轭梯度 HS')
    X4, Y4, Y_d4 = CG_HS(x0, N, E, f, f_d)
    print(X4, Y4, Y_d4)
    print('牛顿法')
    X5, Y5, Y_d5 = NDF(x0, N, E, f, f_d)
    print(X5, Y5, Y_d5)
    n1 = len(Y1)
    x1 = numpy.arange(1, n1 + 1)
    n2 = len(Y2)
    x2 = numpy.arange(1, n2 + 1)
    n3 = len(Y3)
    x3 = range(1, n3 + 1)
    n4 = len(Y4)
    x4 = range(1, n4 + 1)
    n5 = len(Y5)
    x5 = range(1, n5 + 1)
    plt.plot(x1[0:n1-1], Y1[0:n1-1], 'r^', markersize=15,
label='CG-FR:' + str(n1-1))
    plt.plot(x2[0:n2-1], Y2[0:n2-1], 'b*', markersize=10,
label='CG-PP:' + str(n2-1))
    plt.plot(x3[0:n3-1], Y3[0:n3-1], 'gv', markersize=20, label='SD:' +
str(n3-1))
    plt.plot(x4[0:n4-1], Y4[0:n4-1], 'cp', markersize=5, label='HS:' +
str(n4-1))
    plt.plot(x5[0:n5-1], Y5[0:n5-1], 'yo', markersize=5, label='ND:' +
str(n5 - 1))
    plt.legend()
    # 图像显示了不同的方法各自迭代的次数与最优值变化情况，共轭梯度方
法是明显优于最速下降法的
    plt.xlabel('n')
    plt.ylabel('f(x)')
    plt.show()
```

实验结果：

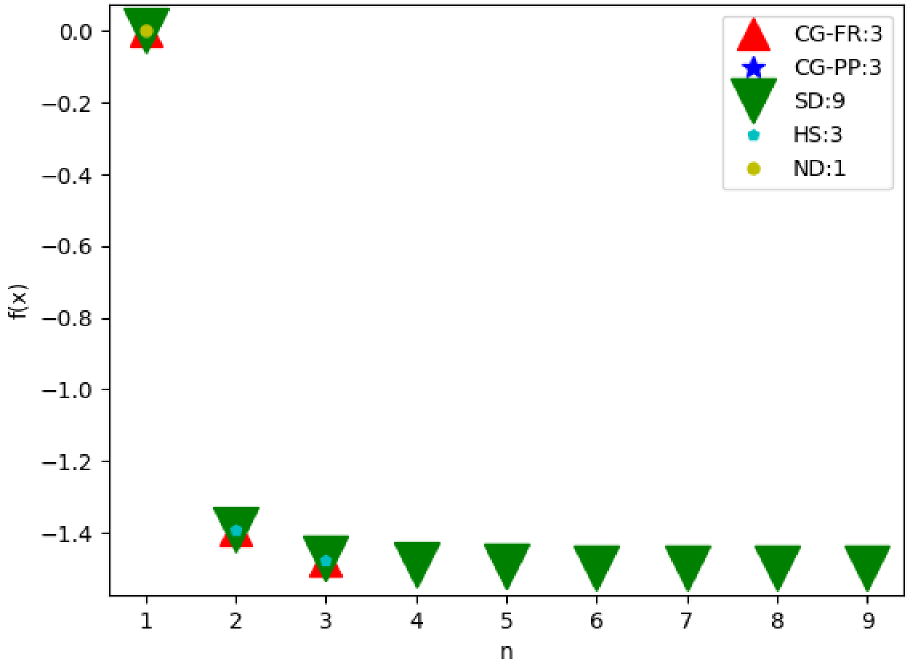


表 2 不同方法的迭代结果

方法	迭代次数	极小值点	极小值
共轭梯度 FR	3	（0.999998563，-2.75281773e-06，-3.74824528e-06）	-1.5000
共轭梯度 PR	3	（0.999991495，1.12941318e-06，-9.27877747e-07）	-1.5000
梯度下降法	9	（0.99699966，-0.00284756，0.00839324）	-1.4999
共轭梯度 HS	3	（0.999992091，7.84070664e-07，-1.18988092e-06）	-1.5000
牛顿法	1	（1，0，0）	-1.5000