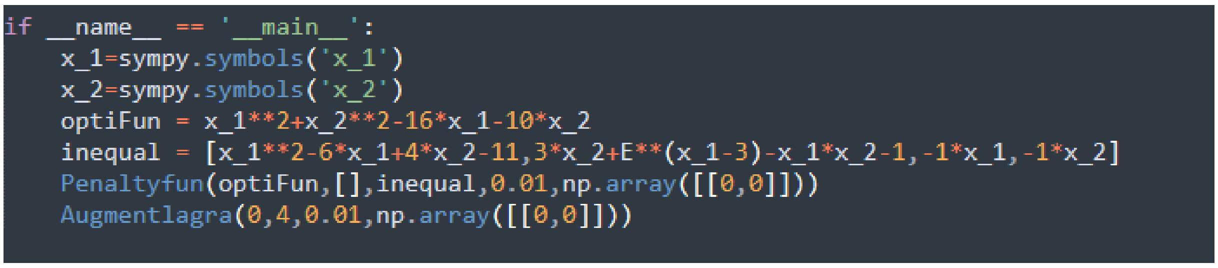
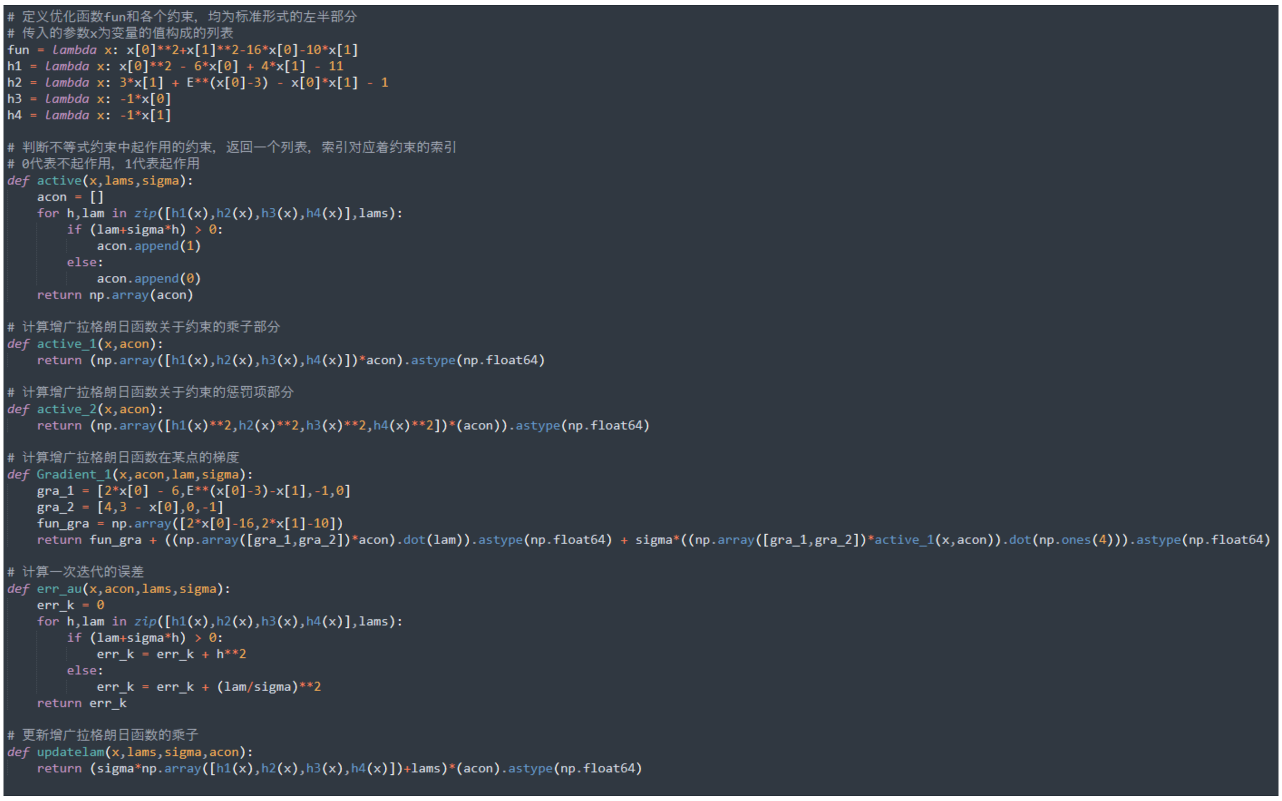
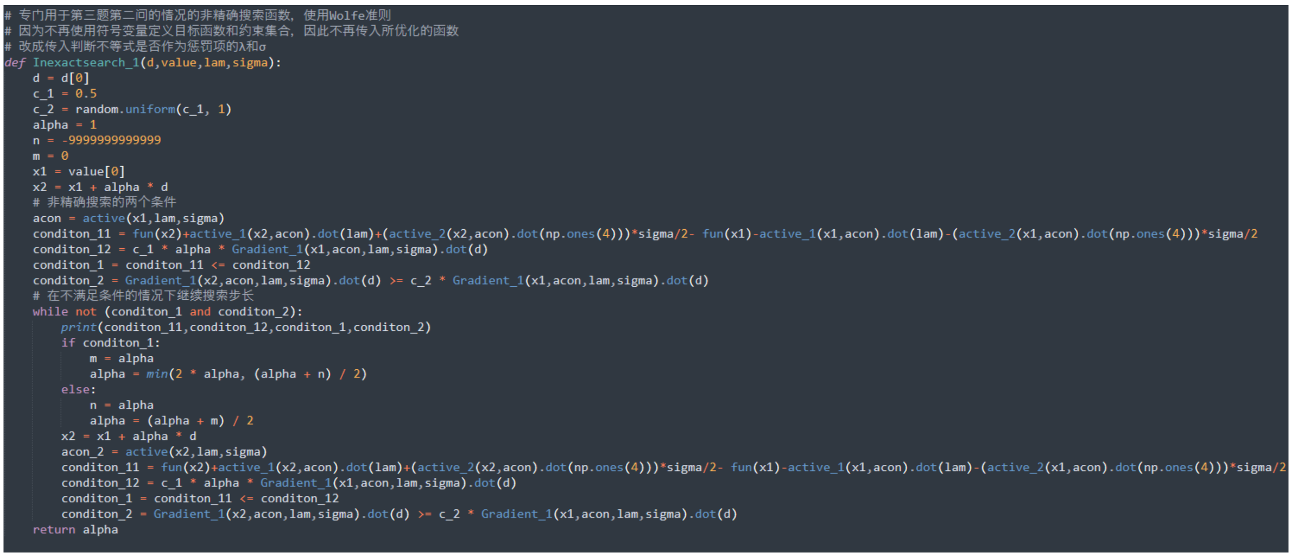
学号：22111076　姓名：卢常建



```python
if __name__ == '__main__':
    x_1=sympy.symbols('x_1')
    x_2=sympy.symbols('x_2')
    optiFun = x_1**2+x_2**2-16*x_1-10*x_2
    inequal = [x_1**2-6*x_1+4*x_2-11,3*x_2+E**(x_1-3)-x_1*x_2-1,-1*x_1,-1*x_2]
    Penaltyfun(optiFun,[],inequal,0.01,np.array([[0,0]]))
    Augmentlagra(0,4,0.01,np.array([[0,0]]))
```

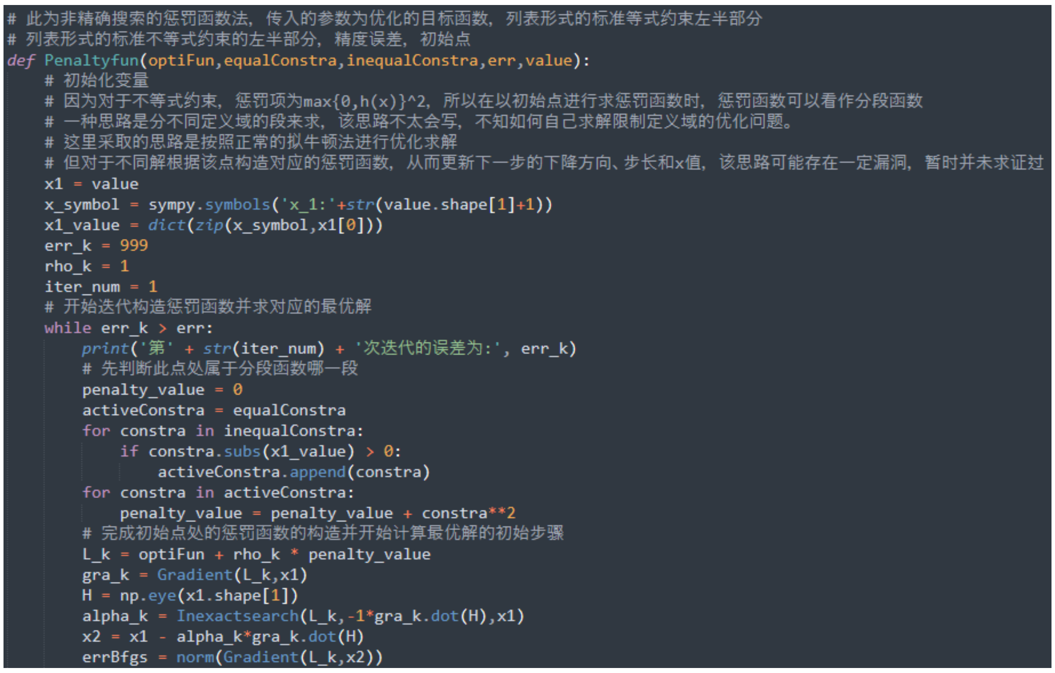


```python
# 定义优化函数fun和各个约束，均为标准形式的左半部分
# 传入的参数x为变量的值构成的列表
fun = lambda x: x[0]**2+x[1]**2-16*x[0]-10*x[1]
h1 = lambda x: x[0]**2 - 6*x[0] + 4*x[1] - 11
h2 = lambda x: 3*x[1] + E**(x[0]-3) - x[0]*x[1] - 1
h3 = lambda x: -1*x[0]
h4 = lambda x: -1*x[1]

# 判断不等式约束中起作用的约束，返回一个列表，索引对应着约束的索引
# 0代表不起作用，1代表起作用
def active(x,lams,sigma):
    acon = []
    for h,lam in zip([h1(x),h2(x),h3(x),h4(x)],lams):
        if (lam+sigma*h) > 0:
            acon.append(1)
        else:
            acon.append(0)
    return np.array(acon)

# 计算增广拉格朗日函数关于约束的乘子部分
def active_1(x,acon):
    return (np.array([h1(x),h2(x),h3(x),h4(x)])*acon).astype(np.float64)

# 计算增广拉格朗日函数关于约束的惩罚项部分
def active_2(x,acon):
    return (np.array([h1(x)**2,h2(x)**2,h3(x)**2,h4(x)**2])*(acon)).astype(np.float64)

# 计算增广拉格朗日函数在某点的梯度
def Gradient_1(x,acon,lam,sigma):
    gra_1 = [2*x[0] - 6,E**(x[0]-3)-x[1],-1,0]
    gra_2 = [4,3 - x[0],0,-1]
    fun_gra = np.array([2*x[0]-16,2*x[1]-10])
    return fun_gra + ((np.array([gra_1,gra_2])*acon).dot(lam)).astype(np.float64) + sigma*((np.array([gra_1,gra_2])*active_1(x,acon)).dot(np.ones(4))).astype(np.float64)

# 计算一次迭代的误差
def err_au(x,acon,lams,sigma):
    err_k = 0
    for h,lam in zip([h1(x),h2(x),h3(x),h4(x)],lams):
        if (lam+sigma*h) > 0:
            err_k = err_k + h**2
        else:
            err_k = err_k + (lam/sigma)**2
    return err_k

# 更新增广拉格朗日函数的乘子
def updatelam(x,lams,sigma,acon):
    return (sigma*np.array([h1(x),h2(x),h3(x),h4(x)])+lams)*(acon).astype(np.float64)
```

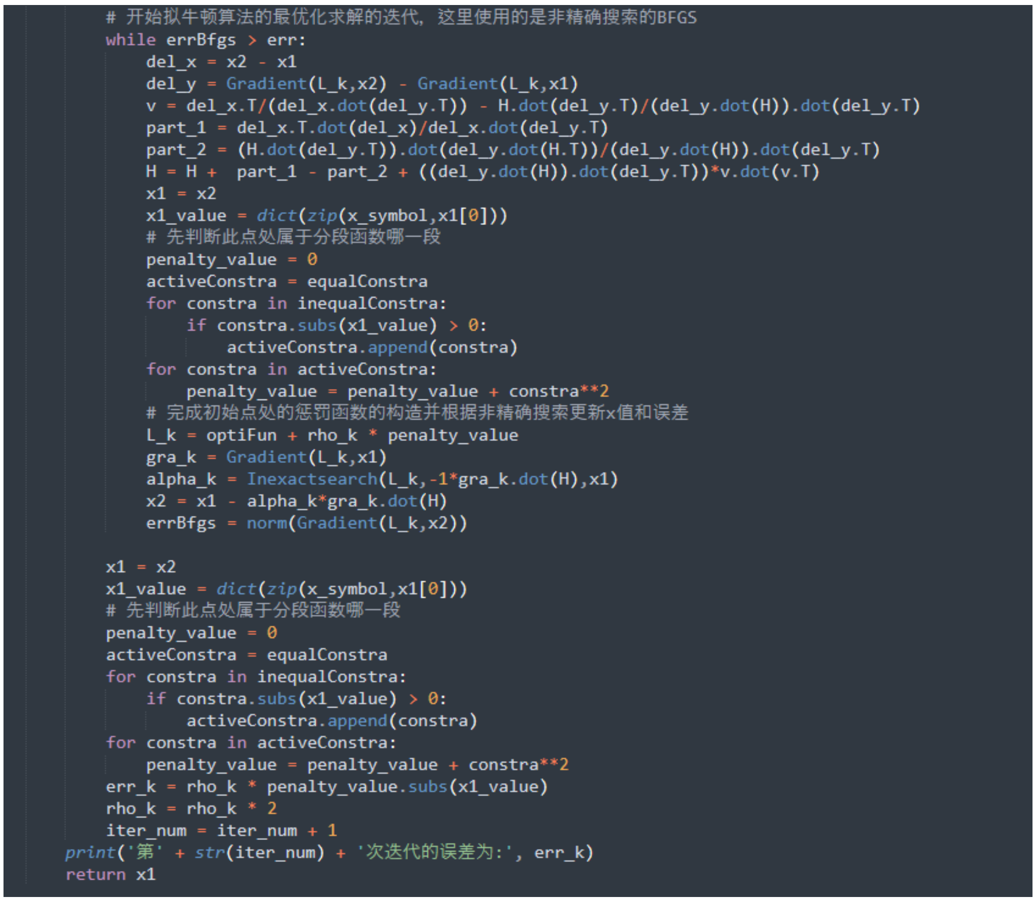


```python
# 专门用于第三题第二问的情况的非精确搜索函数，使用Wolfe准则
# 因为不再使用符号变量定义目标函数和约束集合，因此不再传入所优化的函数
# 改成传入判断不等式是否作为惩罚项的λ和σ
def Inexactsearch_1(d,value,lam,sigma):
    d = d[0]
    c_1 = 0.5
    c_2 = random.uniform(c_1, 1)
    alpha = 1
    n = -9999999999999
    m = 0
    x1 = value[0]
    x2 = x1 + alpha * d
    # 非精确搜索的两个条件
    acon = active(x1,lam,sigma)
    conditon_11 = fun(x2)+active_1(x2,acon).dot(lam)+(active_2(x2,acon).dot(np.ones(4)))*sigma/2- fun(x1)-active_1(x1,acon).dot(lam)-(active_2(x1,acon).dot(np.ones(4)))*sigma/2
    conditon_12 = c_1 * alpha * Gradient_1(x1,acon,lam,sigma).dot(d)
    conditon_1 = conditon_11 <= conditon_12
    conditon_2 = Gradient_1(x2,acon,lam,sigma).dot(d) >= c_2 * Gradient_1(x1,acon,lam,sigma).dot(d)
    # 在不满足条件的情况下继续搜索步长
    while not (conditon_1 and conditon_2):
        print(conditon_11,conditon_12,conditon_1,conditon_2)
        if conditon_1:
            m = alpha
            alpha = min(2 * alpha, (alpha + n) / 2)
        else:
            n = alpha
            alpha = (alpha + m) / 2
        x2 = x1 + alpha * d
        acon_2 = active(x2,lam,sigma)
        conditon_11 = fun(x2)+active_1(x2,acon).dot(lam)+(active_2(x2,acon).dot(np.ones(4)))*sigma/2- fun(x1)-active_1(x1,acon).dot(lam)-(active_2(x1,acon).dot(np.ones(4)))*sigma/2
        conditon_12 = c_1 * alpha * Gradient_1(x1,acon,lam,sigma).dot(d)
        conditon_1 = conditon_11 <= conditon_12
        conditon_2 = Gradient_1(x2,acon,lam,sigma).dot(d) >= c_2 * Gradient_1(x1,acon,lam,sigma).dot(d)
    return alpha
```

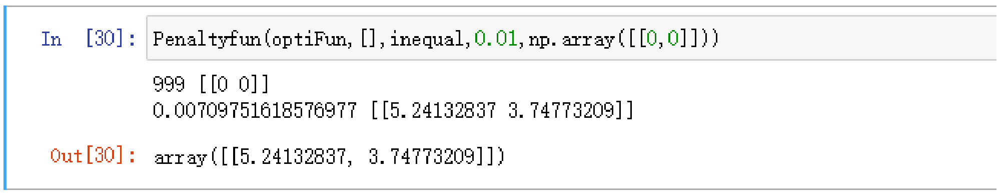
1、罚函数法



```python
# 此为非精确搜索的惩罚函数法，传入的参数为优化的目标函数，列表形式的标准等式约束左半部分
# 列表形式的标准不等式约束的左半部分，精度误差，初始点
def Penaltyfun(optiFun,equalConstra,inequalConstra,err,value):
    # 初始化变量
    # 因为对于不等式约束，惩罚项为max{0,h(x)}^2，所以在以初始点进行求惩罚函数时，惩罚函数可以看作分段函数
    # 一种思路是分不同定义域的段来求，该思路不太会写，不知如何自己求解限制定义域的优化问题。
    # 这里采取的思路是按照正常的拟牛顿法进行优化求解
    # 但对于不同解根据该点构造对应的惩罚函数，从而更新下一步的下降方向、步长和x值，该思路可能存在一定漏洞，暂时并未求证过
    x1 = value
    x_symbol = sympy.symbols('x_1:'+str(value.shape[1]+1))
    x1_value = dict(zip(x_symbol,x1[0]))
    err_k = 999
    rho_k = 1
    iter_num = 1
    # 开始迭代构造惩罚函数并求对应的最优解
    while err_k > err:
        print('第' + str(iter_num) + '次迭代的误差为:', err_k)
        # 先判断此点处属于分段函数哪一段
        penalty_value = 0
        activeConstra = equalConstra
        for constra in inequalConstra:
            if constra.subs(x1_value) > 0:
                activeConstra.append(constra)
        for constra in activeConstra:
            penalty_value = penalty_value + constra**2
        # 完成初始点处的惩罚函数的构造并开始计算最优解的初始步骤
        L_k = optiFun + rho_k * penalty_value
        gra_k = Gradient(L_k,x1)
        H = np.eye(x1.shape[1])
        alpha_k = Inexactsearch(L_k,-1*gra_k.dot(H),x1)
        x2 = x1 - alpha_k*gra_k.dot(H)
        errBfgs = norm(Gradient(L_k,x2))
```

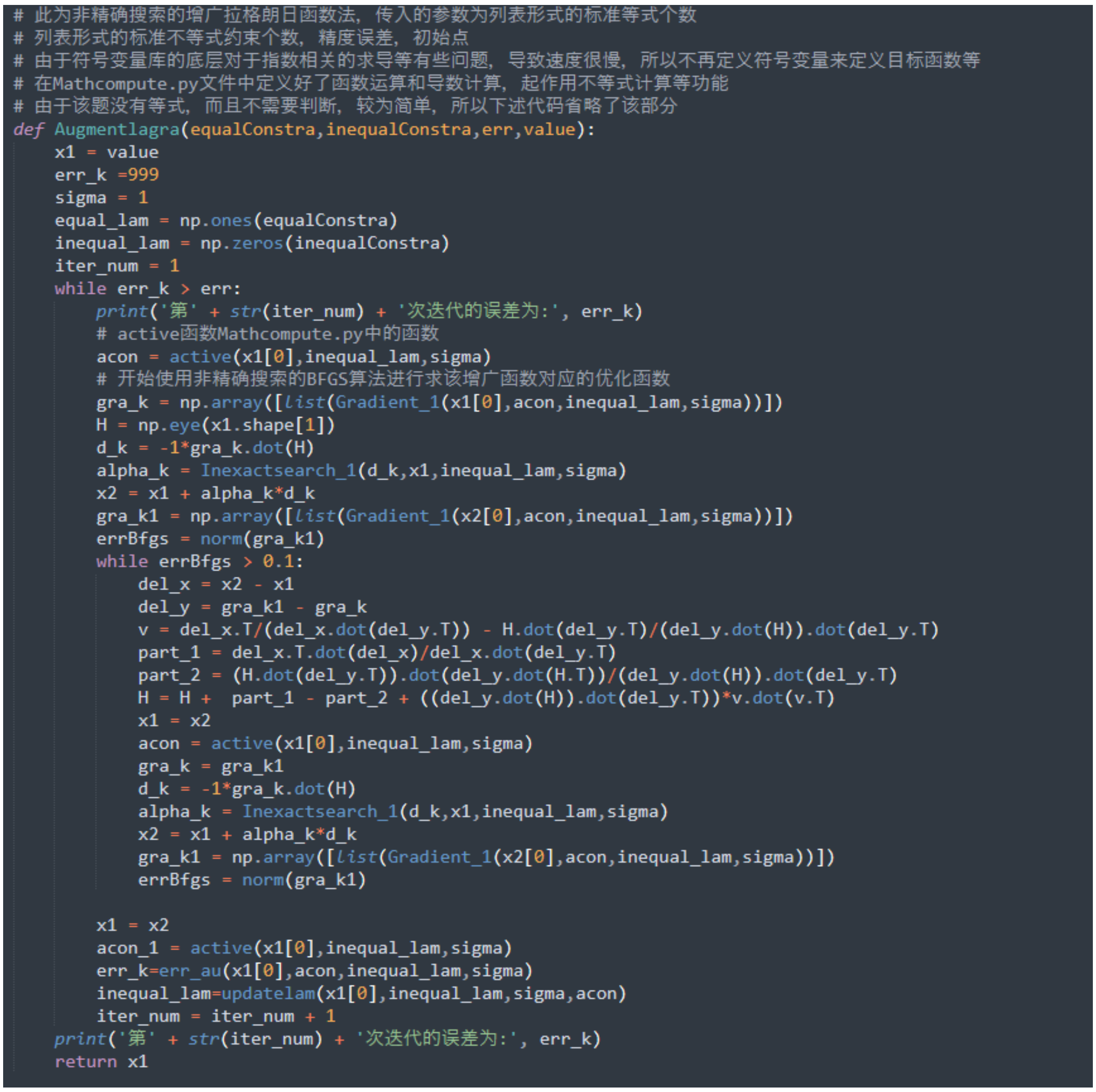


```python
        # 开始拟牛顿算法的最优化求解的迭代，这里使用的是非精确搜索的BFGS
        while errBfgs > err:
            del_x = x2 - x1
            del_y = Gradient(L_k,x2) - Gradient(L_k,x1)
            v = del_x.T/(del_x.dot(del_y.T)) - H.dot(del_y.T)/(del_y.dot(H)).dot(del_y.T)
            part_1 = del_x.T.dot(del_x)/del_x.dot(del_y.T)
            part_2 = (H.dot(del_y.T)).dot(del_y.dot(H.T))/(del_y.dot(H)).dot(del_y.T)
            H = H +  part_1 - part_2 + ((del_y.dot(H)).dot(del_y.T))*v.dot(v.T)
            x1 = x2
            x1_value = dict(zip(x_symbol,x1[0]))
            # 先判断此点处属于分段函数哪一段
            penalty_value = 0
            activeConstra = equalConstra
            for constra in inequalConstra:
                if constra.subs(x1_value) > 0:
                    activeConstra.append(constra)
            for constra in activeConstra:
                penalty_value = penalty_value + constra**2
            # 完成初始点处的惩罚函数的构造并根据非精确搜索更新x值和误差
            L_k = optiFun + rho_k * penalty_value
            gra_k = Gradient(L_k,x1)
            alpha_k = Inexactsearch(L_k,-1*gra_k.dot(H),x1)
            x2 = x1 - alpha_k*gra_k.dot(H)
            errBfgs = norm(Gradient(L_k,x2))

        x1 = x2
        x1_value = dict(zip(x_symbol,x1[0]))
        # 先判断此点处属于分段函数哪一段
        penalty_value = 0
        activeConstra = equalConstra
        for constra in inequalConstra:
            if constra.subs(x1_value) > 0:
                activeConstra.append(constra)
        for constra in activeConstra:
            penalty_value = penalty_value + constra**2
        err_k = rho_k * penalty_value.subs(x1_value)
        rho_k = rho_k * 2
        iter_num = iter_num + 1
    print('第' + str(iter_num) + '次迭代的误差为:', err_k)
    return x1
```

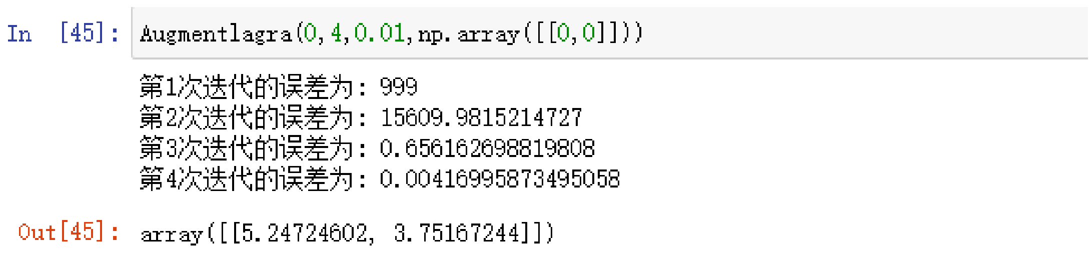


```
In  [30]:  Penaltyfun(optiFun,[],inequal,0.01,np.array([[0,0]]))

           999 [[0 0]]
           0.00709751618576977 [[5.24132837 3.74773209]]

Out[30]:  array([[5.24132837, 3.74773209]])
```

2、 增广拉格朗日函数法



```
# 此为非精确搜索的增广拉格朗日函数法, 传入的参数为列表形式的标准等式个数
# 列表形式的标准不等式约束个数, 精度误差, 初始点
# 由于符号变量库的底层对于指数相关的求导等有些问题, 导致速度很慢, 所以不再定义符号变量来定义目标函数等
# 在Mathcompute.py文件中定义好了函数运算和导数计算, 起作用不等式计算等功能
# 由于该题没有等式, 而且不需要判断, 较为简单, 所以下述代码省略了该部分
def Augmentlagra(equalConstra,inequalConstra,err,value):
    x1 = value
    err_k =999
    sigma = 1
    equal_lam = np.ones(equalConstra)
    inequal_lam = np.zeros(inequalConstra)
    iter_num = 1
    while err_k > err:
        print('第' + str(iter_num) + '次迭代的误差为:', err_k)
        # active函数Mathcompute.py中的函数
        acon = active(x1[0],inequal_lam,sigma)
        # 开始使用非精确搜索的BFGS算法进行求该增广函数对应的优化函数
        gra_k = np.array([list(Gradient_1(x1[0],acon,inequal_lam,sigma))])
        H = np.eye(x1.shape[1])
        d_k = -1*gra_k.dot(H)
        alpha_k = Inexactsearch_1(d_k,x1,inequal_lam,sigma)
        x2 = x1 + alpha_k*d_k
        gra_k1 = np.array([list(Gradient_1(x2[0],acon,inequal_lam,sigma))])
        errBfgs = norm(gra_k1)
        while errBfgs > 0.1:
            del_x = x2 - x1
            del_y = gra_k1 - gra_k
            v = del_x.T/(del_x.dot(del_y.T)) - H.dot(del_y.T)/(del_y.dot(H)).dot(del_y.T)
            part_1 = del_x.T.dot(del_x)/del_x.dot(del_y.T)
            part_2 = (H.dot(del_y.T)).dot(del_y.dot(H.T))/(del_y.dot(H)).dot(del_y.T)
            H = H +  part_1 - part_2 + ((del_y.dot(H)).dot(del_y.T))*v.dot(v.T)
            x1 = x2
            acon = active(x1[0],inequal_lam,sigma)
            gra_k = gra_k1
            d_k = -1*gra_k.dot(H)
            alpha_k = Inexactsearch_1(d_k,x1,inequal_lam,sigma)
            x2 = x1 + alpha_k*d_k
            gra_k1 = np.array([list(Gradient_1(x2[0],acon,inequal_lam,sigma))])
            errBfgs = norm(gra_k1)

        x1 = x2
        acon_1 = active(x1[0],inequal_lam,sigma)
        err_k=err_au(x1[0],acon,inequal_lam,sigma)
        inequal_lam=updatelam(x1[0],inequal_lam,sigma,acon)
        iter_num = iter_num + 1
    print('第' + str(iter_num) + '次迭代的误差为:', err_k)
    return x1
```



```
In  [45]: Augmentlagra(0,4,0.01,np.array([[0,0]]))

          第1次迭代的误差为: 999
          第2次迭代的误差为: 15609.9815214727
          第3次迭代的误差为: 0.656162698819808
          第4次迭代的误差为: 0.004169958734950 58

Out[45]:  array([[5.24724602,  3.75167244]])
```