《最优化技术》实验报告

一、实验目的

理解掌握一维搜索算法中的黄金分割法，牛顿法，并用于实际问题的求解。

二、实验项目内容

1) 给定一个函数 $f(x)=8e^{1-x}+7\log(x)$，利用黄金分割法把区间压缩到长度只有 0.23，需给出所有中间结果。

2) 给定一个函数 $f(x)=60-10x_1-4x_2+x_1^2+x_2^2-x_1x_2$，利用牛顿法求解该函数的最小值，需给出中间结果。

注意：所有程序请用 python 语言实现。只提交本电子文档，注意本文件末尾的文件命名要求；源程序一节请用代码备注的方式说明你的算法和思路；实验结果一节需要提供测试结果截图并给出结果分析。

三、实验过程或算法（源程序）

1）黄金分割法求函数最小值

```python
'''一维搜索算法—黄金分割法
求解思路:黄金分割法确定最小值,给定初始值 a 和 b,比较 f(a)和 f(b),
    若 f1>f2,则最小值必在[a1,b]中,另 a=a1,继续迭代
    若 f1<f2,则最小值必在[a,a2]中,另 b=a2,继续迭代
    终止条件是 b-a<e
'''
import math
import numpy as np
from matplotlib import pyplot as plt

def function(x):         #原函数待寻找最小值,注意此处 x>0
    return 8*math.exp(1-x)+7*math.log(x)

fig0=plt.figure(figsize=(7,5))
X=np.linspace(0.1,3,2000)
Y=[]
for i in range(len(X)):
    Y.append(function(X[i]))
plt.plot(X,np.array(Y))               #绘出图像
plt.title("Golden Section Method")
plt.legend(["f(x)=8e^(1-x)+7log(x)"])
```

```

def Golden_section(a,b,e):
#黄金分割法求最小值,注意输入满足 a<b
    a1=a+0.382*(b-a)
    a2=a+0.618*(b-a)
    f1=function(a1)
    f2=function(a2)
    while(b-a>e):
    #e 为精度,当|a-b|<e 时搜索停止
        print('Internal value:[a:%s  b:%s]'%(a,b))
        #输出中间结果
        plt.scatter([a,b],[function(a),function(b)])
        #在图上绘制中间结果
        if(f1>f2):
        #若 f1>f2,则最小值必在[a1,b]中,另 a=a1,继续迭代
            a=a1
            a1=a2
            a2=a+0.618*(b-a)
            f1=f2
            f2=function(a2)
        else:
        #若 f1<f2,则最小值必在[a,a2]中,另 b=a2,继续迭代
            b=a2
            a2=a1
            a1=a+0.382*(b-a)
            f2=f1
            f1=function(a1)
    plt.scatter([a,b],[function(a),function(b)])
    return a,b,(a+b)/2.0

a=float(input())
b=float(input())
e=0.23
a,b,x=Golden_section(a,b,e)
print('Final answer:[a:%s  b:%s]'%(a,b))
print('min:%f'%function(x))
plt.show()
```

2）牛顿法

```
'''一维搜索算法—牛顿法
求解思路:根据牛顿法公式进行迭代,当|ak+1-ak|<e 时迭代终止
    对一维牛顿法进行扩展,利用黑塞矩阵求解多维函数最小值
'''
import math
```

```python
import numpy as np
from matplotlib import pyplot as plt

def function(x1,x2):          #原函数
    return 60-10*x1-4*x2+x1*x1+x2*x2-x1*x2

def derivative1_1(x1,x2):    #fx
    return -10+2*x1-x2
def derivative1_2(x1,x2):    #fy
    return -4+2*x2-x1

def derivative2_11(x1,x2):   #fxx
    return 2
def derivative2_12(x1,x2):   #fxy
    return -1
def derivative2_21(x1,x2):   #fyx
    return -1
def derivative2_22(x1,x2):   #fyy
    return 2

x0=float(input())            #迭代初值
y0=float(input())
e=1                          #判断终止条件
eps=0.001                    #阈值
k=0                          #迭代次数
print('初值:(',"{0:.1f}".format(x0),',  ', "{0:.1f}".format(y0),')开始迭代')

while e>eps:
    x1=x0+(derivative1_1(x0,y0)*derivative2_22(x0,y0)-derivative1_2(x0,y0)*derivative2_12(x0,y0))/(derivative2_21(x0,y0)*derivative2_12(x0,y0)-derivative2_11(x0,y0)*derivative2_22(x0,y0))
    y1=y0+(derivative1_2(x0,y0)*derivative2_11(x0,y0)-derivative1_1(x0,y0)*derivative2_21(x0,y0))/(derivative2_21(x0,y0)*derivative2_12(x0,y0)-derivative2_11(x0,y0)*derivative2_22(x0,y0))
    e=max(abs(x1-x0),abs(y1-y0))     #取较大的|ak+1-ak|作为e
    x0=x1
    y0=y1
    k=k+1
    print('迭代次数:',"{0:.0f}".format(k),',方程根的近似值为x=', "{0:.10f}".format(x1),',y=',"{0:.10f}".format(y1))
print('函数的极小值点:(', "{0:.10f}".format(x1), ',', "{0:.10f}".format(y1), ')', "极小值:", "{0:.10f}".format(function(x1, y1)))
```

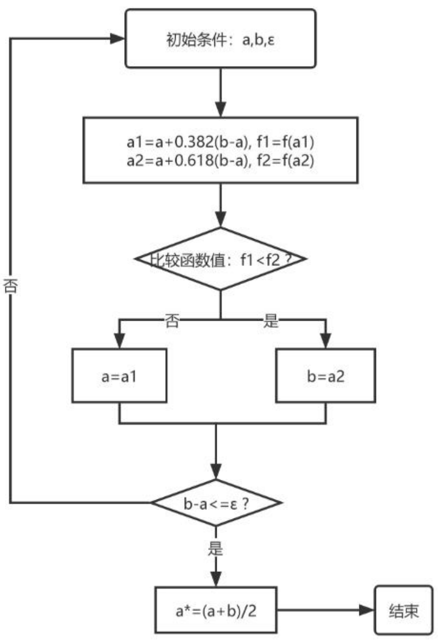
```python
fig=plt.figure()                #绘图
ax1=plt.axes(projection='3d')
xx=np.arange(-10,10,0.1)
yy=np.arange(-10,10,0.1)
x,y=np.meshgrid(xx,yy)
z=np.array(60-10*x-4*y+x*x+y*y-x*y)
ax1.plot_surface(x,y,z,rstride=1,cstride=1,cmap='rainbow')
ax1.contour(x,y,z,stride=0.05,zdim='z',offset=-3,cmap='rainbow')
plt.show()
```

四、实验结果及分析和（或）源程序调试过程

1. 结果分析

1）黄金分割法

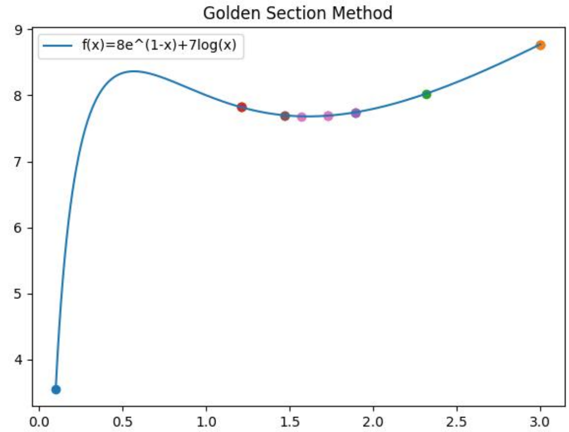
黄金分割法思路如下：



黄金分割法求解函数极小值，并将迭代过程中出现的结果绘在图中并打印至命令行。下面给出 5 个不同情况下的样例并进行分析。

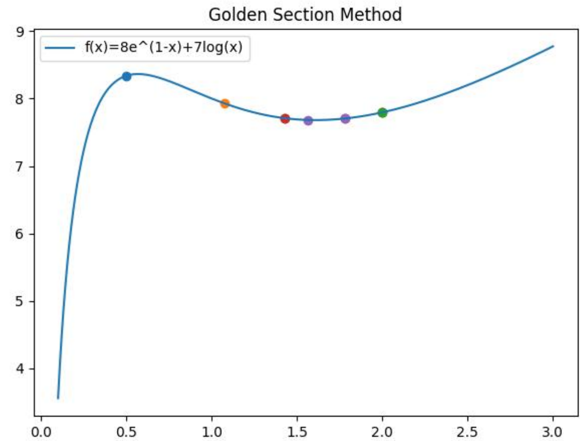
样例 1：给定搜索区间

```
0.1
3
Internal value:[a:0.1  b:3.0]
Internal value:[a:1.2078  b:3.0]
Internal value:[a:1.2078  b:2.3153796]
Internal value:[a:1.2078  b:1.8922]
Internal value:[a:1.4692408000000001  b:1.8922]
Internal value:[a:1.4692408000000001  b:1.7306295856]
Final answer:[a:1.569091316092002  b:1.7306295856]
min:7.681784
```



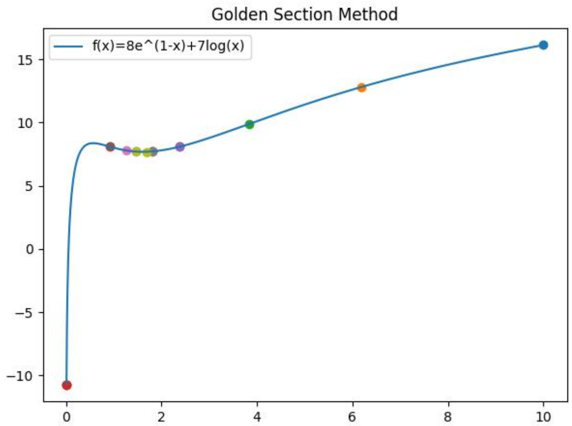
样例 2：给定搜索区间

```
0.5
2
Internal value:[a:0.5  b:2.0]
Internal value:[a:1.073  b:2.0]
Internal value:[a:1.427  b:2.0]
Internal value:[a:1.427  b:1.781114]
Final answer:[a:1.562271548  b:1.781114]
min:7.683602
```



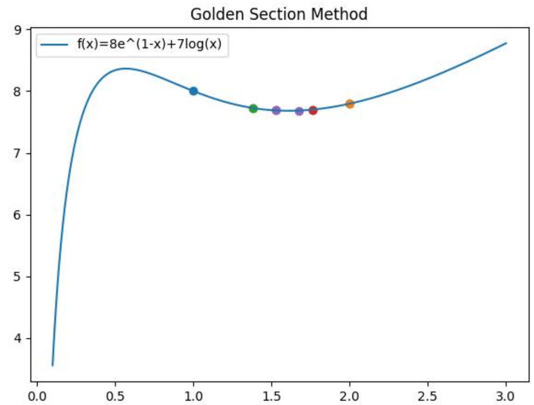
样例 3：给定搜索区间较大

```
0.01
10
Internal value:[a:0.01  b:10.0]
Internal value:[a:0.01  b:6.18382]
Internal value:[a:0.01  b:3.82618]
Internal value:[a:0.01  b:2.36839924]
Internal value:[a:0.9109085096800001  b:2.36839924]
Internal value:[a:0.9109085096800001  b:1.81163778101776]
Internal value:[a:1.2549870913310244  b:1.81163778101776]
Internal value:[a:1.4677807600000001  b:1.81163778101776]
Final answer:[a:1.4677807600000001  b:1.6802843989889757]
min:7.681482
```



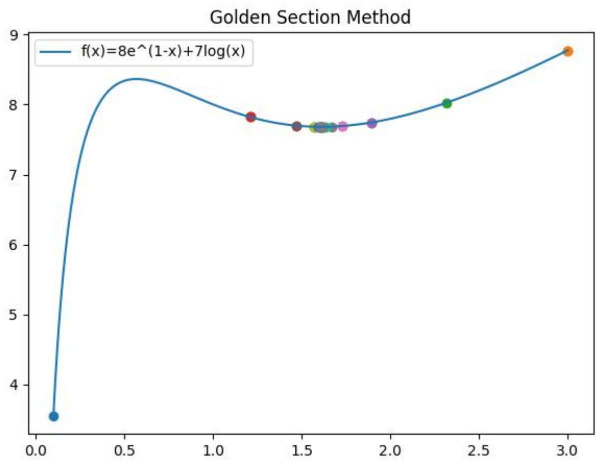
样例 4：给定搜索区间较小

```
1
2
Internal value:[a:1.0  b:2.0]
Internal value:[a:1.3820000000000001  b:2.0]
Internal value:[a:1.3820000000000001  b:1.763924]
Internal value:[a:1.527894968  b:1.763924]
Final answer:[a:1.527894968  b:1.673760909776]
min:7.680506
```



样例 5：增强精度限制，将 e 设为 0.01

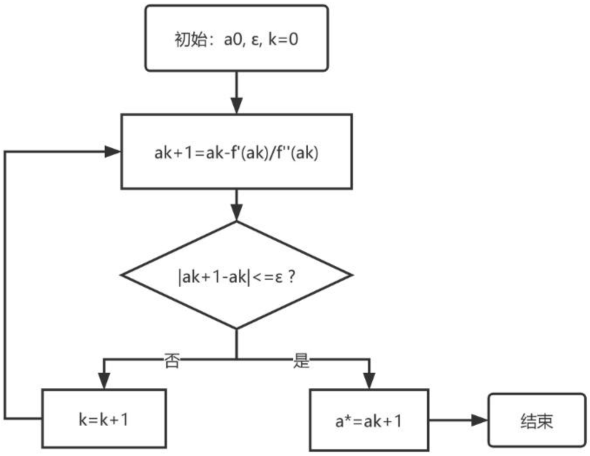
```
0.1
3
Internal value:[a:0.1  b:3.0]
Internal value:[a:1.2078  b:3.0]
Internal value:[a:1.2078  b:2.3153796]
Internal value:[a:1.2078  b:1.8922]
Internal value:[a:1.4692408000000001  b:1.8922]
Internal value:[a:1.4692408000000001  b:1.7306295856]
Internal value:[a:1.5690913160992002  b:1.7306295856]
Internal value:[a:1.5690913160992002  b:1.6689219666506945]
Internal value:[a:1.5690913160992002  b:1.6308954072]
Internal value:[a:1.5927004788997057  b:1.6308954072]
Internal value:[a:1.5927004788997057  b:1.6163049445892874]
Internal value:[a:1.6017173847931259  b:1.6163049445892874]
Internal value:[a:1.607226624609871  b:1.6163049445892874]
Internal value:[a:1.607226624609871  b:1.6128370263571503]
Internal value:[a:1.607226624609871  b:1.6107324967471537]
Internal value:[a:1.608565867766313  b:1.6107324967471537]
Internal value:[a:1.608565867766313  b:1.6099048444764725]
Final answer:[a:1.609077356869594  b:1.6099048444764725]
min:7.680446
```



分析：用黄金分割法确定函数极小值时,对给定初始值 a 和 b,比较 f(a)和 f(b), 并不断削减区间进行迭代，直至区间长度小于阈值，迭代结束得到答案。本次给出 5 个样例，其中样例 1、2 给出**常规输入**得到预期值：**极小值点 x∈[1.56,1.73]**，**极小值 7.68**；样例 3、4 分别给定**较大**和**较小的搜索区间**，发现结果差不多，**迭代次数明显增多（减少）**；样例 5 **调整迭代结束阈值**，为 0.01，发现迭代次数明显增多，此种情况下**结果更加精确，计算也更加繁琐**。

2）牛顿法

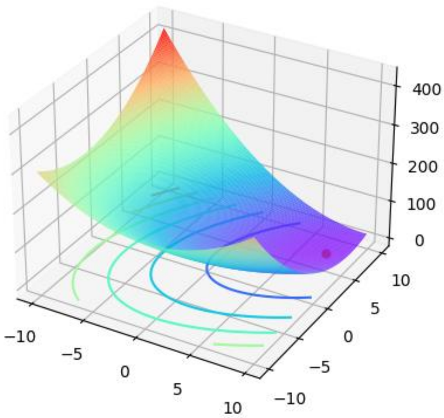
牛顿法思路如下：



牛顿法求解函数极小值，并将迭代过程中出现的结果绘在图中并打印至命令行。下面给出 4 个不同情况下的样例并进行分析。

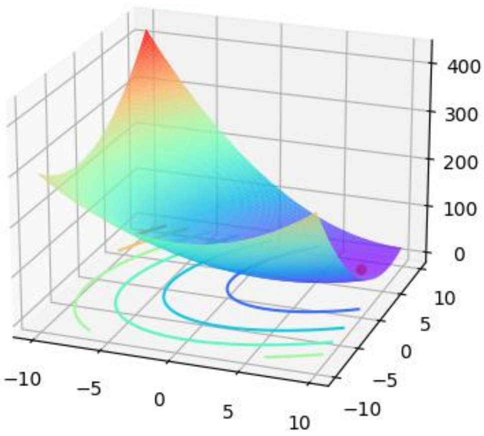
样例 1：给定初始值

```
0
0
初值:( 0.0 ,  0.0 )开始迭代
迭代次数: 1 ,方程根的近似值为x= 8.0000000000 ,y= 6.0000000000
迭代次数: 2 ,方程根的近似值为x= 8.0000000000 ,y= 6.0000000000
函数的极小值点:( 8.0000000000 , 6.0000000000 ) 极小值: 8.0000000000
```



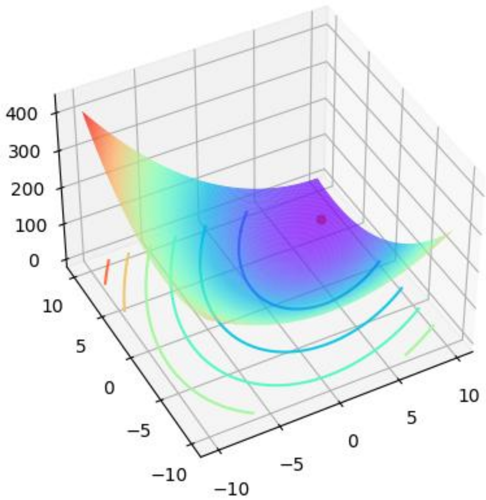
样例 2：给定靠近极小值点的初始值

```
5
9
初值:( 5.0 ,  9.0 )开始迭代
迭代次数: 1 ,方程根的近似值为x= 8.0000000000 ,y= 6.0000000000
迭代次数: 2 ,方程根的近似值为x= 8.0000000000 ,y= 6.0000000000
函数的极小值点:( 8.0000000000 , 6.0000000000 ) 极小值: 8.0000000000
```



样例 3：给定远离极小值点的初始值

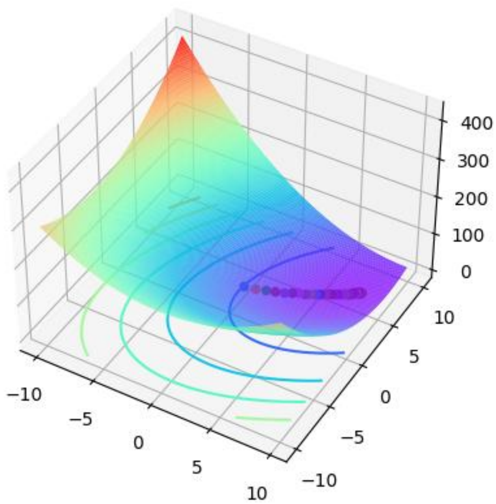
```
-5
-10
初值:( -5.0 ,  -10.0 )开始迭代
迭代次数: 1 ,方程根的近似值为x= 8.0000000000 ,y= 6.0000000000
迭代次数: 2 ,方程根的近似值为x= 8.0000000000 ,y= 6.0000000000
函数的极小值点:( 8.0000000000 , 6.0000000000 ) 极小值: 8.0000000000
```



样例 4：设置一定学习率 a=0.1

```
0
0
初值:( 0.0 ,  0.0 )开始迭代
迭代次数: 1 ,方程根的近似值为x= 0.8000000000 ,y= 0.6000000000
迭代次数: 2 ,方程根的近似值为x= 1.5200000000 ,y= 1.1400000000
迭代次数: 3 ,方程根的近似值为x= 2.1680000000 ,y= 1.6260000000
迭代次数: 4 ,方程根的近似值为x= 2.7512000000 ,y= 2.0634000000
迭代次数: 5 ,方程根的近似值为x= 3.2760800000 ,y= 2.4570600000
迭代次数: 6 ,方程根的近似值为x= 3.7484720000 ,y= 2.8113540000
迭代次数: 7 ,方程根的近似值为x= 4.1736248000 ,y= 3.1302186000
迭代次数: 8 ,方程根的近似值为x= 4.5562623200 ,y= 3.4171967400
迭代次数: 9 ,方程根的近似值为x= 4.9006360880 ,y= 3.6754770660
迭代次数: 10 ,方程根的近似值为x= 5.2105724792 ,y= 3.9079293594
迭代次数: 11 ,方程根的近似值为x= 5.4895152313 ,y= 4.1171364235
迭代次数: 12 ,方程根的近似值为x= 5.7405637082 ,y= 4.3054227811
迭代次数: 13 ,方程根的近似值为x= 5.9665073373 ,y= 4.4748805030
迭代次数: 14 ,方程根的近似值为x= 6.1698566036 ,y= 4.6273924527
迭代次数: 15 ,方程根的近似值为x= 6.3528709432 ,y= 4.7646532074
迭代次数: 16 ,方程根的近似值为x= 6.5175838489 ,y= 4.8881878867
迭代次数: 17 ,方程根的近似值为x= 6.6658254640 ,y= 4.9993690980
迭代次数: 18 ,方程根的近似值为x= 6.7992429176 ,y= 5.0994321882
迭代次数: 19 ,方程根的近似值为x= 6.9193186259 ,y= 5.1894889694
迭代次数: 20 ,方程根的近似值为x= 7.0273867633 ,y= 5.2705400725
迭代次数: 21 ,方程根的近似值为x= 7.1246480869 ,y= 5.3434860652
迭代次数: 22 ,方程根的近似值为x= 7.2121832783 ,y= 5.4091374587
迭代次数: 23 ,方程根的近似值为x= 7.2909649504 ,y= 5.4682237128
迭代次数: 24 ,方程根的近似值为x= 7.3618684554 ,y= 5.5214013415
迭代次数: 25 ,方程根的近似值为x= 7.4256816098 ,y= 5.5692612074
迭代次数: 26 ,方程根的近似值为x= 7.4831134489 ,y= 5.6123350866
迭代次数: 27 ,方程根的近似值为x= 7.5348021040 ,y= 5.6511015780
迭代次数: 28 ,方程根的近似值为x= 7.5813218936 ,y= 5.6859914202
迭代次数: 29 ,方程根的近似值为x= 7.6231897042 ,y= 5.7173922782
迭代次数: 30 ,方程根的近似值为x= 7.6608707338 ,y= 5.7456530503
```

```
迭代次数: 31 ,方程根的近似值为x= 7.6947836604 ,y= 5.7710877453
迭代次数: 32 ,方程根的近似值为x= 7.7253052944 ,y= 5.7939789708
迭代次数: 33 ,方程根的近似值为x= 7.7527747649 ,y= 5.8145810737
迭代次数: 34 ,方程根的近似值为x= 7.7774972884 ,y= 5.8331229663
迭代次数: 35 ,方程根的近似值为x= 7.7997475596 ,y= 5.8498106697
迭代次数: 36 ,方程根的近似值为x= 7.8197728036 ,y= 5.8648296027
迭代次数: 37 ,方程根的近似值为x= 7.8377955233 ,y= 5.8783466425
迭代次数: 38 ,方程根的近似值为x= 7.8540159709 ,y= 5.8905119782
迭代次数: 39 ,方程根的近似值为x= 7.8686143739 ,y= 5.9014607804
迭代次数: 40 ,方程根的近似值为x= 7.8817529365 ,y= 5.9113147024
迭代次数: 41 ,方程根的近似值为x= 7.8935776428 ,y= 5.9201832321
迭代次数: 42 ,方程根的近似值为x= 7.9042198785 ,y= 5.9281649089
迭代次数: 43 ,方程根的近似值为x= 7.9137978907 ,y= 5.9353484180
迭代次数: 44 ,方程根的近似值为x= 7.9224181016 ,y= 5.9418135762
迭代次数: 45 ,方程根的近似值为x= 7.9301762915 ,y= 5.9476322186
迭代次数: 46 ,方程根的近似值为x= 7.9371586623 ,y= 5.9528689967
迭代次数: 47 ,方程根的近似值为x= 7.9434427961 ,y= 5.9575820971
迭代次数: 48 ,方程根的近似值为x= 7.9490985165 ,y= 5.9618238874
迭代次数: 49 ,方程根的近似值为x= 7.9541886648 ,y= 5.9656414986
迭代次数: 50 ,方程根的近似值为x= 7.9587697983 ,y= 5.9690773488
迭代次数: 51 ,方程根的近似值为x= 7.9628928185 ,y= 5.9721696139
迭代次数: 52 ,方程根的近似值为x= 7.9666035367 ,y= 5.9749526525
迭代次数: 53 ,方程根的近似值为x= 7.9699431830 ,y= 5.9774573872
迭代次数: 54 ,方程根的近似值为x= 7.9729488647 ,y= 5.9797116485
迭代次数: 55 ,方程根的近似值为x= 7.9756539782 ,y= 5.9817404837
迭代次数: 56 ,方程根的近似值为x= 7.9780885804 ,y= 5.9835664353
迭代次数: 57 ,方程根的近似值为x= 7.9802797224 ,y= 5.9852097918
迭代次数: 58 ,方程根的近似值为x= 7.9822517501 ,y= 5.9866888126
迭代次数: 59 ,方程根的近似值为x= 7.9840265751 ,y= 5.9880199313
迭代次数: 60 ,方程根的近似值为x= 7.9856239176 ,y= 5.9892179382
迭代次数: 61 ,方程根的近似值为x= 7.9870615258 ,y= 5.9902961444
迭代次数: 62 ,方程根的近似值为x= 7.9883553733 ,y= 5.9912665299
迭代次数: 63 ,方程根的近似值为x= 7.9895198359 ,y= 5.9921398769
迭代次数: 64 ,方程根的近似值为x= 7.9905678523 ,y= 5.9929258893
迭代次数: 65 ,方程根的近似值为x= 7.9915110671 ,y= 5.9936333003
函数的极小值点:( 7.9915110671 , 5.9936333003 ) 极小值: 8.0000585504
```



分析：用牛顿法确定函数极小值时,对给定初始值 a 和 b,带入迭代公式，直至区间长度小于阈值，迭代结束得到答案。本次给出 4 个样例，其中样例 1 给出**常规输入**得到预期值：**极小值点 X=[8, 6]**，**极小值 8**；样例 2、3 分别给定离极小值点**较远**和**较近的搜索区间**，发现**迭代次数均为 2 次**；原因是本例中原函数的黑塞矩阵是正定矩阵，迭代两次就会下降到极小值点，属于巧合；样例 4 **给 X 的步长乘一个学习率**，发现迭代次数明显增多，此种情况下对于一般函数来说**可以更准确的到达极值，也可以人为控制迭代的速度**。

2. 调试过程

1）黄金分割法中注意函数定义域，一开始没注意 x>0 的限制，出现 Math error

```
  File "d:\PyLearn\1.py", line 9, in function
    return 8*math.exp(1-x)+7*math.log(x)
ValueError: math domain error
```

2）一开始误将迭代值 a1、a2 更新值赋值错误，导致结果错误
3）给程序输入不同初值，发现迭代次数明显不同，适应现实情况
4）对一维搜索牛顿法进行拓展，使其适用多维函数
5）牛顿法求解时由于函数特殊性，黑塞矩阵为正定矩阵，迭代次数较少
6）给牛顿法加上学习率可以控制迭代速度
7）上述黄金分割法代码对输入要求较高，要求数据包含一个极小值，即“大”-“小”-“大”的情况，可以用进退法加以改进，具体操作代码见附录。

3. 总结

黄金分割法和**牛顿法**均是**利用区间消去法原理将初始搜索区间不断缩短**，从而求得极小点的数值解。其中黄金分割法要求**试验点位置由给定的规律确定**，不考虑函数值的分布。而**牛顿法的试验点位置是按函数值近似分布的极小点确定**的。当函数具有较好的解析性质时，牛顿法效果更好。

另外，牛顿法的高维情况依然可以用牛顿迭代求解，但 **Hessian 矩阵引入的复杂性**，使得牛顿迭代求解的难度大大增加，但是已经有了解决这个问题的办法就是 Quasi-Newton method，不再直接计算 Hessian 矩阵，而是每一步的时候使用梯度向量更新 Hessian 矩阵的近似。Hessian 矩阵是一个多元函数的二阶偏导数构成的方阵，描述了函数的局部曲率。牛顿法是迭代算法，每一步需要求解目标函数的 Hessian 矩阵的逆矩阵，计算过程复杂。可以采用**拟牛顿法**通过正定矩阵近似 Hessian 矩阵的逆矩阵或 Hessian 矩阵，简化计算过程。

五、附录

对黄金分割法改进，加入进退法确定操作区间。代码如下：

```python
#进退法确定搜索区间—>黄金分割法确定最小值
def function(x):
    return pow(x,3)-3*pow(x,2)+2*x
# def function(x):
#     return x+1/x

def Interval_search(start,step):
    step=abs(step)                    #step>0
    a1=start
    a2=a1+step
    f1=function(a1)
    f2=function(a2)
    while(f1!=f2):
        if(f1<f2):                    #确定进 or 退
            tmp=a1
            a1=a2
            a2=tmp
            tmpf=f1
            f1=f2
```

```
                f2=tmpf
                step*=(-1)              #反向寻找

            a3=a2+step
            f3=function(a3)
            if(f2<=f3):
                return a1,a3
            else:
                step*=2
                a1=a2
                a2=a3
                f1=f2
                f2=f3

        return a1,a2

def Golden_section(a,b,e):              #a<b,e 为精度
        a1=a+0.382*(b-a)
        a2=a+0.618*(b-a)
        f1=function(a1)
        f2=function(a2)
        while(b-a>e):
            if(f1>f2):
                a=a1
                a1=a2
                a2=a+0.618*(b-a)
                f1=f2
                f2=function(a2)
            else:
                b=a2
                a2=a1
                a1=a+0.382*(b-a)
                f2=f1
                f1=function(a1)
        return (a+b)/2.0

a,b=Interval_search(1,0.1)
x=Golden_section(a,b,0.0001)
# a,b=Interval_search(0.01,0.1)
# x=Golden_section(a,b,0.0001)
print('a:%s  b:%s'%(a,b))
print('min:%f'%function(x))
```
运行结果:

样例 1:

$f(x)=x^3-3x^2+2x$

```
a:1.2000000000000002  b:1.8000000000000003
min:-0.384900
```

样例 2：

f(x)=x+1/x

```
a:0.41000000000000003  b:1.61
min:2.000000
```