西安电子科技大学
XIDIAN UNIVERSITY

最优化上机报告

一维搜索法、常用无约束最优化方法

任俊杰
1702052
17170120015

最优化（一） 一维搜索法

一、一维搜索法简介

对于无约束优化问题：

$$\min_{X\in R^n} f(x)$$

当已知迭代点 X_k 和下降方向 P_k 时，要确定适当的步长 t_k 使 $f(X_{k+1}) = f(X_k + t_k P_k)$ 比 $f(X_k)$有所下降，即相当于对于参变量 t 的函数：

$$\varphi(t) = f(X_k + tP_k)$$

要在区间 $[0,+\infty)$ 上选取 $t = t_k$ 使$f(X_{k+1}) < f(X_k)$，即：

$$\varphi(t_k) = f(X_k + tP_k) < f(X_k) = \varphi(0)$$

由于这种从已知点 X_k 出发，沿某一下降的探索方向 P_k来确定步长 t_k的问题，实质上是单变量函数 $\varphi(t)$ 关于变量 t 的一维搜索选取问题，故通常叫做一维搜索。按这种方法确定的步长 t_k 称为最优步长，该方法的优点是：使目标函数值在搜索方向上下降的最多。

这种在给定方向上确定最优步长的过程，在多维优化过程中是多次反复进行的，所以说一维搜索是解多维优化问题的基础。

二、实验问题描述

对于以下优化问题：

$$minf(x) = 3x^4 - 4x^3 - 12x^2$$

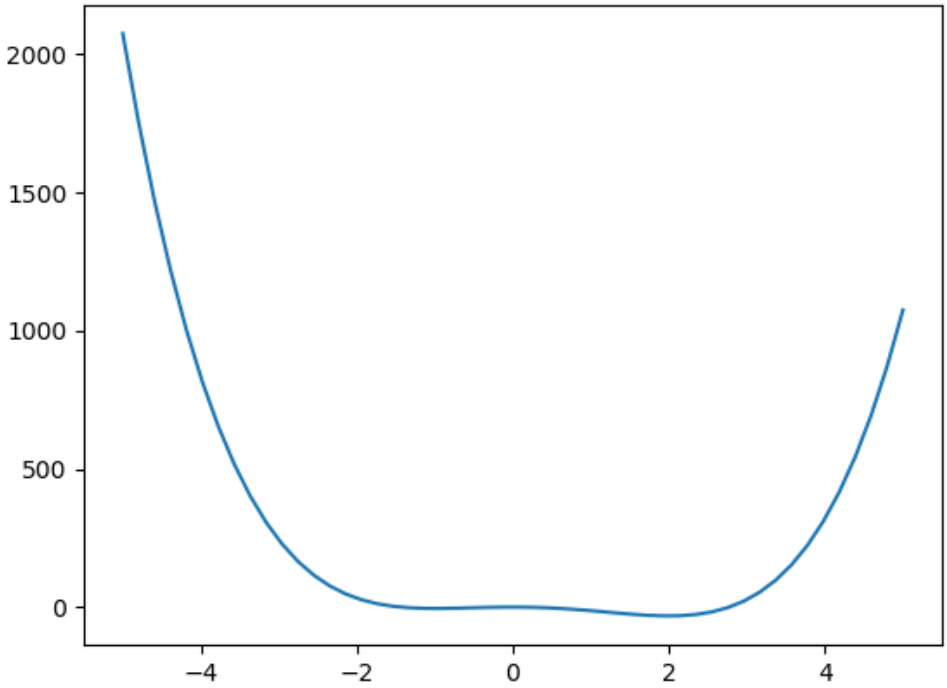
分别用黄金分割法、牛顿法、和二次插值法求解。

（1） 黄金分割法分别取初始搜索区间[-2,0]和[0,3]；

（2） 牛顿法分别取初始点$x_0 = -1.2$和$x_0 = 2.5$；

（3） 二次插值法分别取初始点$x_1 = -1.2$，$x_2 = -1.1$，$x_3 = -0.8$ 和$x_1 = 1.5$，$x_2 = 1.7$，$x_3 = 2.5$ 。

绘制出了需要求解的目标函数图像，以便于选择初始搜索区间：



本次实验对上述三种方法均编程实现，下面具体说明：

三、Newton 切线法

3.1 选择 Newton 切线法原因

第一，牛顿切线法实现的基本思想较为简单，虽然涉及到二阶导数，但 python 中 sympy 可以很方便的解决求导问题，故编程实现起来较为容易，代码行数很少。

第二，牛顿切线法收敛速度较快，在实际测试中，只需要三四次迭代就可以达到较高的精度。但是算法对初始点的选取要求过于严格，对于初始搜索

区间不确定的问题很不方便。

3.2 Newton 切线法算法介绍

设 $\varphi: R^1 \to R^1$ 在已获得搜索区间 $[a,b]$ 内具有连续二阶导数，求：

$$\min_{a \le t \le b} \varphi(t)$$

因为 $\varphi(t)$ 在 $[a,b]$ 上可微，故 $\varphi(t)$ 在 $[a,b]$ 上由最小值，令$\varphi'(t)=0.$

下面设在区间 $[a,b]$ 中经过 k 次迭代已求得方程 $\varphi'(t)=0$ 的一个近似根 t_k。过$(t_k, \varphi'(t_k))$ 做曲线 $y=\varphi'(t)$ 的切线，其方程是：

$$y-\varphi'(t_k)=\varphi''(t_k)(t-t_k)$$

然后用这条切线与横轴交点的横坐标 t_{k+1} 作为根的新的近似，它可由上述方程令 $y=0$ 时解出来，即：

$$t_{k+1}=t_k-\frac{\varphi'(t_k)}{\varphi''(t_k)}$$

这就是牛顿切线法迭代公式。

3.3 Newton 切线法迭代步骤

已知 $\varphi(t), \varphi'(t)$ 表达式，终止限 ε。

(1) 确定初始搜索区间 $[a,b]$，要求 $\varphi'(a)<0, \varphi'(b)>0$。

(2) 选定 t_0。

(3) 计算 $t=t_0-\varphi'(t_0)/\varphi''(t_0)$。

(4) 若 $|t-t_0| \ge \varepsilon$，则 $t_0=t$，转（3）；否则转（5）。

(5) 打印 t，$\varphi(t)$，结束。

3.4 Newton 切线法优缺点

优点：这种方法如果初始点选得适当，收敛速度很快，通常经过几次迭代

就可以得到满足一般精度要求的结果。

缺点：第一，需要求二阶导数。如果在多维最优化问题的一维搜索中使用这种方法，就要涉及 Hesse 矩阵，一般是难于求出的。第二，当曲线 $y = \varphi'(t)$ 在 $[a, b]$ 上有较复杂的弯曲时，这种方法也往往失效。

3.5 Python 实现代码及结果展示

代码：

```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
def f(x):
    return 3*x**4-4*x**3-12*x**2
def  f_diff_1(x_value):
    x = sp.Symbol('x')
    fx = sp.diff(f(x),x)
    return fx.evalf(subs = {x:x_value})
def  f_diff_2(x_value):
    x = sp.Symbol('x')
    fx = sp.diff(f(x),x)
    fx = sp.diff(fx,x)
    return fx.evalf(subs = {x:x_value})
# 牛顿切线法
def Newton_Tangent(a,b,t_0,stopValue = 0.0001):    #要求 a、b 两点处的一阶导数值相反
    t = t_0+1   #随便取值
    while abs(t-t_0) >= stopValue:
        print("t_0:",t_0)
        t_0 = t
        t = t_0 - f_diff_1(t_0)/f_diff_2(t_0)
    print("t_0:",t_0)
    t_out = t_0
    f_out = f(t_0)
    return t_out,f_out
if __name__ == "__main__":
    print("*************** 牛顿切线法 ***************************")
    t_out,f_out = Newton_Tangent(-5,5,-1.2)
    print("起始点：-1.2，  minf(x) = f(%f) = %f" % (t_out,f_out),'\n')
    t_out,f_out = Newton_Tangent(-5,5,2.5)
    print("起始点：2.5，  minf(x) = f(%f) = %f\n" % (t_out,f_out))
```

结果：

```
****************************** 牛顿切线法 ******************************
t_0: -1.2
t_0: -0.19999999999999996
t_0: 0.0378378378378378
t_0: 0.000638876784617548
t_0: 2.03690998006804e-7
起始点: -1.2,  minf(x) = f(0.000000) = -0.000000

t_0: 2.5
t_0: 3.5
t_0: 2.64864864864865
t_0: 2.19271295522491
t_0: 2.02488080704001
t_0: 2.00050011924040
t_0: 2.00000020830076
起始点: 2.5,  minf(x) = f(2.000000) = -32.000000
```

上图结果可以看到，当ε设为 0.0001 时，起始点在-1.2 处时，迭代 4 次搜索到的最小点为 0.0，搜索失败（实际为-1、2）。起始点在 2.5 处时，迭代 6 次搜索到的最小点为 2.0，搜索成功。

上述结果说明，牛顿切线法对初始点的选取非常严格，初始点选择的不好，可能就不收敛。并且初始搜索区间的确定也很严格，上述区间是从图像中观察得出的[-5,5]，在实际使用中非常不便。但牛顿法的收敛速度快，只用很少的迭代次数就可以达到很高的精度。

四、黄金分割法

4.1 选择黄金分割法原因

第一，相比于 Newton 切线法，黄金分割法不需要对初始搜索区间有要求，初始搜索区间选取方便灵活。

第二，这种试探法，通过试探点函数值的比较，不使用导数的搜索，因此不需要计算二阶导数，计算量小。

第三，黄金分割法适用于单峰区间上的任何函数，甚至是不连续函数，适

用范围广。

4.2 黄金分割法算法介绍

黄金分割：是指将整体一分为二，较大部分与整体部分的比值等于较小部分与较 大部分的比值，其比值约为 0.618。

在单谷区间 $[a,b]$ 内适当插入两点t_1，t_2，由此把区间 $[a,b]$ 分为三段，然后再通过比较这两点函数值的大小，就可以确定删去最左段还是最右段，或者同时删去左右两段保留中间段。如此继续下去可将单谷区间无限缩小。

4.3 黄金分割法迭代步骤

已知 $\varphi(t)$，常数 $\beta = 0.382$，终止限 ε。

（1） 确定 $\varphi(t)$ 的初始搜索区间 $[a,b]$。

（2） 计算$t_2 = a + \beta(b-a)$，$\varphi_2 = \varphi(t_2)$。

（3） 计算$t_1 = a + b - t_2$，$\varphi_1 = \varphi(t_1)$。

（4） $|t_1 - t_2| < \varepsilon$，则打印$t^* = \frac{t_1+t_2}{2}$，结束；否则转（5）

（5） 判断是否满足 $\varphi_1 < \varphi_2$：

若满足，则置 $a = t_2$，$t_2 = t_1$，$\varphi_2 = \varphi_1$，然后转（3）；

否则，置$b = t_1$，$t_1 = t_2$，$\varphi_1 = \varphi_2$，$t_2 = a + \beta(b-a)$，$\varphi_2 = \varphi(t_2)$，然后转（4）

4.4 黄金分割法优缺点

优点：黄金分割法是通过所选试点的函数值而逐步缩短单谷区间来搜索最优点 t。该方法适用于单谷区间[a,b]上的任何函数，甚至可以是不连续函数，因此这种算法属于直接法，适用相当广泛

缺点：收敛速度慢，由于每次迭代搜索区间的收缩率是 0.618，故黄金分

割法只是线性收敛的，即这一方法的计算效率并不高。

4.5 Python 实现代码及结果展示

代码：（库和相关函数在上面已经给出，不再重复）

```
# 黄金分割法
def Golden_Section(a,b,stopValue = 0.01):
    t_1 = a
    t_2 = b
    beta = 1-(np.sqrt(5)-1)/2   #t1 取在 0.618 位置，#t2 取在 0.382 处

    while abs(t_1-t_2) >= stopValue:
        t_2 = a + beta*(b-a)
        value_2 = f(t_2)
        t_1 = a + b - t_2
        value_1 = f(t_1)
        print("t1 : f(%0.3f) = %0.6f" % (t_1,value_1))
        print("t2 : f(%0.3f) = %0.6f" % (t_2,value_2))

        if value_1 < value_2:
            a = t_2
            print("丢弃 a -- t2   [a,b]:[%0.3f,%0.3f]\n" % (a,b))
        else:
            b = t_1
            print("丢弃 t1 -- b   [a,b]:[%0.3f,%0.3f]\n" % (a,b))

    t_out = (t_1+t_2)/2
    f_out = f(t_out)
    return t_out,f_out
if __name__ == "__main__":
    print("***************** 黄金分割法 ******************")
    t_out,f_out = Golden_Section(-2,0)
    print("起始区间：[-2,0]，  minf(x) = f(%f) = %f" % (t_out,f_out))
    t_out,f_out = Golden_Section(0,3)
    print("起始区间：[0，3]，  minf(x) = f(%f) = %f\n" % (t_out,f_out))
```

结果：（第一张图只为部分过程，第二张为全过程）

```
t2 : f(-0.997) = -4.999827
丢弃t1 -- b   [a,b]:[-1.013,-0.987]

t1 : f(-0.997) = -4.999827
t2 : f(-1.003) = -4.999826
丢弃a -- t2   [a,b]:[-1.003,-0.987]

起始区间：[-2,0]，  minf(x) = f(-1.000000) = -5.000000
```

```
t1 : f(1.854) = -31.294449
t2 : f(1.146) = -16.603066
丢弃a -- t2   [a,b]:[1.146,3.000]

t1 : f(2.292) = -28.416134
t2 : f(1.854) = -31.294449
丢弃t1 -- b   [a,b]:[1.146,2.292]

t1 : f(1.854) = -31.294449
t2 : f(1.584) = -27.111629
丢弃a -- t2   [a,b]:[1.584,2.292]

t1 : f(2.021) = -31.983495
t2 : f(1.854) = -31.294449
丢弃a -- t2   [a,b]:[1.854,2.292]

t1 : f(2.125) = -31.401565
t2 : f(2.021) = -31.983495
丢弃t1 -- b   [a,b]:[1.854,2.125]

t1 : f(2.021) = -31.983495
t2 : f(1.957) = -31.936286
丢弃a -- t2   [a,b]:[1.957,2.125]

t1 : f(2.061) = -31.862601
t2 : f(2.021) = -31.983495
丢弃t1 -- b   [a,b]:[1.957,2.061]

t1 : f(2.021) = -31.983495
t2 : f(1.997) = -31.999653
丢弃t1 -- b   [a,b]:[1.957,2.021]

t1 : f(1.997) = -31.999653
t2 : f(1.982) = -31.988221
丢弃a -- t2   [a,b]:[1.982,2.021]

t1 : f(2.006) = -31.998606
t2 : f(1.997) = -31.999653
丢弃t1 -- b   [a,b]:[1.982,2.006]

起始区间: [0, 3],  minf(x) = f(2.001553) = -31.999913
```

从上图可以明显看出，算法迭代了很多次才达到 0.01 的精度，可见黄金分割法的收敛速度非常慢，远不及牛顿切线法。但是两次初始区间的选取均得到了正确的结果，可见黄金分割法适用范围更广。

五、抛物线插值法

5.1 选择抛物线插值法原因

黄金分割法的收敛速度非常慢，计算精度不高，需要寻求一种在同样的搜索次数下，收敛速度较快，计算精度更高的算法，抛物线插值法作为多项式逼近法的一种，符合这些特性。

5.2 选择抛物线插值法算法介绍

抛物线插值法是多项式逼近法的一种。所谓多项式逼近，是利用目标函数在若干点的函数值或导数值等信息，构成一个与目标函数相接近的低次插值多项式，用该多项式的最优解作为目标函数的近似最优解。

在极小点附近，利用原目标函数 $\varphi(t)$ 上的三个插值点 $(t_1,\varphi(t_1)),(t_2,\varphi(t_2)),(t_3,\varphi(t_3))$，(其中 $\varphi(t)$ 是 $[t_1,t_2]$ 上的单谷函数，$t_1<t_0<t_2$，$\varphi(t_1)\geq\varphi(t_0)$，$\varphi(t_2)\geq\varphi(t_0)$ 构成一个二次插值多项式：

$$P(t)=a+bt+ct^2$$

在搜索区间中不断使用二次多项式去近似目标函数 $\varphi(t)$，并逐步用 $P(t)$ 的最优解 $\bar{t}$ 去逼近一维搜索问题

$$\min_{t_1\leq t\leq t_2}\varphi(t)=f(X_k+tP_k)$$

的极小点 t^*。

5.3 选择抛物线插值法迭代步骤

(1) 取三个点t_1，t_0，t_2，其中$t_1<t_0<t_2(t_0=0.5(t_1+t_2))$

(2) 计算三个点的原函数值$\varphi_1=\varphi(t_1)$，$\varphi_2=\varphi(t_2)$，$\varphi_3=\varphi(t_3)$；

(3) 过三点做一条二次曲线$P(t)=a+bt+ct^2$，并求解线性方程组；

(4) 计算二次曲线 $P(t)$ 的极小值 $\bar{t}$（推导过程略）；

$$\bar{t} = -\frac{b}{2c} = \frac{1}{2}\frac{(t_0^2 - t_2^2)\varphi_1 + (t_2^2 - t_1^2)\varphi_0 + (t_1^2 - t_0^2)\varphi_2}{(t_0 - t_2)\varphi_1 + (t_2 - t_1)\varphi_0 + (t_1 - t_0)\varphi_2}$$

（5）终止条件判断，若$|t_0 - \bar{t}| < \varepsilon$，则输出 $\bar{t}$、$\varphi(\bar{t})$ 作为近似最优解；否则缩短搜索空间（四种情况不再一一列举），构成新的三个点，继续上述过程，直至满足终止条件。

5.4 抛物线插值法优缺点

优点：在同样的搜索次数下，收敛速度较快，其计算精度更高，更适宜高维度优化问题。

缺点：程序略复杂，可靠性差些（但经过某些改进，方法的可靠度可大为增加）。

5.5 Python 实现代码及结果展示

代码：（库和相关函数在上面已经给出，不再重复）

```
# 二次插值法
def Quadratic_Interpolation(t_0,t_1,t_2,stopValue = 0.0001):    #要求
f(t_1)>f(t_0),f(t_2)>f(t_0)
    t_match = 0.5 \
        * ( (pow(t_0,2)-pow(t_2,2))*f(t_1) + (pow(t_2,2)-
pow(t_1,2))*f(t_0) + (pow(t_1,2)-pow(t_0,2))*f(t_2) ) \
        / ( (t_0-t_2)*f(t_1) + (t_2-t_1)*f(t_0) + (t_1-t_0)*f(t_2) )

    while abs(t_match-t_0) >= stopValue:
        if t_match > t_0:
            if f(t_match) <= f(t_0):
                t_2 = t_0
                t_0 = t_match
            else:
                t_1 = t_match
        else:
            if f(t_match) <= f(t_0):
                t_1 = t_0
                t_0 = t_match
            else:
                t_2 = t_match
```

```
        t_match = 0.5 \
            * ( (pow(t_0,2)-pow(t_2,2))*f(t_1) + (pow(t_2,2)-
pow(t_1,2))*f(t_0) + (pow(t_1,2)-pow(t_0,2))*f(t_2) ) \
            / ( (t_0-t_2)*f(t_1) + (t_2-t_1)*f(t_0) + (t_1-
t_0)*f(t_2) )

    t_out = t_match
    f_out = f(t_match)
    return t_out,f_out

if __name__ == "__main__":
    print("****************** 二次插值法 *****************")
    t_out,f_out = Quadratic_Interpolation(-1.2,-1.1,-0.8)
    print("起始点：[-1.2,-1.1,-0.8]，  minf(x) = f(%f) = %f"
    % (t_out,f_out))
    t_out,f_out = Quadratic_Interpolation(1.5,1.7,2.5)
    print("\n 起始点：[1.5, 1.7, 2.5]，  minf(x) = f(%f) = %f\n"
    % (t_out,f_out))
```

结果：

```
******************************** 二次插值法 ********************************
t_match: -0.98379746835443
t_match: -1.0056315659730515
t_match: -0.9991372577617613
t_match: -0.9999631987029904
起始点：[-1.2,-1.1,-0.8]，  minf(x) = f(-0.999998) = -5.000000

t_match: 1.9194068343004504
t_match: 2.093646418377661
t_match: 1.9949612698426484
t_match: 1.9978918769274374
t_match: 1.9998232391681423
t_match: 1.9999427967392762
起始点：[1.5, 1.7, 2.5]，  minf(x) = f(1.999994) = -32.000000
```

上图结果可以看到，当ε设为 0.0001 时，二次插值法使用了很少的迭代次数就完成了对区间的搜索，并且达到了很高的精度。

上述结果说明，相比比黄金分割法，二次插值法的收敛速度更快。

最优化（二） 常用无约束最优化方法

一、无约束最优化方法简介

对于多维无约束优化问题：

$$\min f(X)$$

其中 $f: R^n \to R^1$ 。这个问题的求解是指，在 R^n 中找到一点 X^* ，使得对于任意的 $X \in R^n$ 都有：

$$f(X^*) \leq f(X)$$

成立。点 X^* 就是问题 $\min f(X)$的全局最优点。但是，大多数最优化方法只能求解到局部最优点，即在 R^n 中找到一点 X^*，使得上式在 X^* 的某个邻域中成立。

无约束优化方法是优化技术中极为重要和基本的内容之一。它不仅可以直接用来解决无约束优化问题，而且很多约束优化问题也常将其转化为无约束优化问题，然后用无约束优化方法来求解。

无约束优化方法可以分为两大类：一类是仅用计算函数值所得到的信息来确定搜索方向，通常称它为直接搜索法，简称为直接法。直接法不涉及导数和 Hesse 矩阵，适应性强，但收敛速度一般较慢；另一类需要计算函数的一阶或二阶导数值所得到的信息来确定搜索方向，这一类方法称为间接法（或解析法）。间接法收敛速度一般较快，但需计算梯度，甚至需要计算 Hesse 矩阵。

一般的经验是，在可能求得目标函数导数的情况下还是尽可能使用间接方法；相反，在不可能求得目标函数的导数或根本不存在导数的情况下，当然就应该使用直接法

二、实验问题描述

对于以下优化问题：

$$minf(X) = x_1^2 + x_2^2 - x_1x_2 - 10x_1 - 4x_2 + 60$$

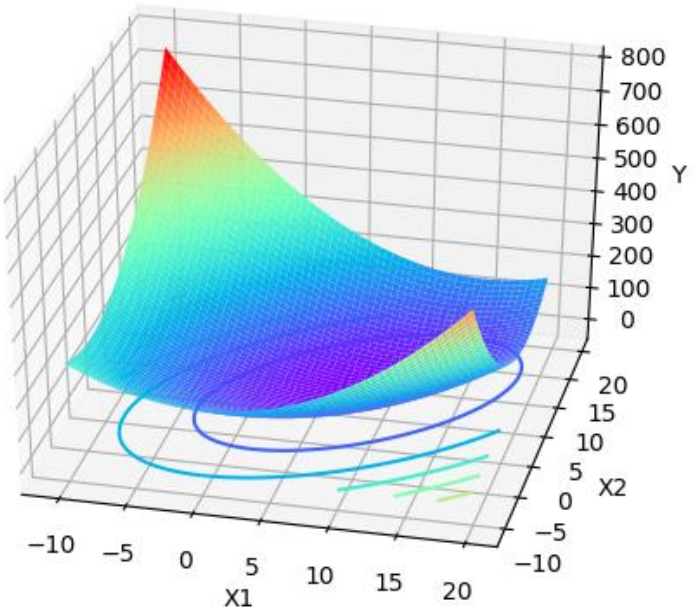
选取初始点$X_0 = [0\,,0]^T$,$\varepsilon = 10^{-2}$,分别用以下方法求解：

（1） 最速下降法；

（2） Newton 法或修正 Newton 法；

（3） 共轭梯度法。

绘制出了需要求解的目标函数图像，以便于对搜索结果进行对照：



本次实验因时间原因，仅编程实现最速下降法和牛顿法，下面具体说明：

三、最速下降法

3.1 选择最速下降法原因

最速下降法作为在无约束优化方法所学的第一种算法，算法思想不是很复杂，计算量相对不大，编程很容易实现，而且对初始点的选取不加以约束，灵

活方便。对这种方法的实现可以进一步加强对这一章所研究内容的理解，编程实现下还是很有必要的。

3.2 选择最速下降法算法介绍

对于无约束优化问题：

$$\min f(X)\ , X \in \boldsymbol{R}^n$$

其中函数 $f(X)$ 具有一阶连续偏导数。为了求其最优解，按最优化算法的基本思想是从一个给定的初始点 X_0 出发，通过基本迭代公式 $X_{k+1} = X_k + t_k P_k$，按照特定的算法产生一串点列 $\{X_K\}$，如果点列收敛，则该点列的极限点为最优解。

在基本迭代公式 $X_{k+1} = X_k + t_k P_k$ 中，每次迭代搜索方向 P_k 取为目标函数 $f(X)$的负梯度方向，即 $P_k = -\nabla f(X_k)$，而每次迭代的不步长 t_k 取为最优步长，由此确定的算法称为最速下降法。

其中步长因子 t_k 按下式确定：

$$f\left(X_k - t_k \nabla f(X_k)\right) = \min_t f(X_k - t\nabla f(X_k))$$

将最速下降法应用于正定二次函数：

$$f(X) = \frac{1}{2} X^T A X + b^T X + c$$

由上式可以推出最速下降法用于二次函数的显式迭代公式(推导过程略)：

$(g_k = g(X_k) = \nabla f(X))$

$$t_k = \frac{g_k^T g_k}{g_k^T A g_k} \text{ 即 } X_{k+1} = X_k - \frac{g_k^T g_k}{g_k^T A g_k} g_k$$

3.3 选择最速下降法迭代步骤

已知目标函数 $f(X)$ 及其梯度 $g(X)$,终止限 ε_1 , ε_2 , ε_3 。

(1) 选定初始点 X_0，计算$f_0 = f(X_0), g_0 = g(X_0)$；置 $k = 0$。

(2) 作直线搜索：$X_{k+1} = ls(X_k, -g_k)$；计算

$$f_{k+1} = f(X_{k+1}), g_{k+1} = g(X_{k+1})$$

(3) 用终止准则检验是否满足：若满足，则打印最优解 $(X_{k+1}, f(X_{k+1}))$，结束；否则，置 $k = k + 1$，转 (2)。

3.4 最速下降法优缺点

优点：最速下降法的优点是算法简单，每次迭代计算量小，占用内存量小，即使从一个不好的初始点出发，往往也能收敛到局部极小点。

缺点：收敛速度慢。沿负梯度方向函数值下降很快的说法，容易使人们产生一种错觉，认为这一定是最理想的搜索方向，沿该方向搜索时收敛速度应该很快，然而事实证明，梯度法的收敛速度并不快。特别是对于等值线（面）具有狭长深谷形状的函数，收敛速度更慢。其原因是由于每次迭代后下一次搜索方向总是与前一次搜索方向相互垂直，如此继续下去就产生所谓的锯齿现象。即从直观上看，在远离极小点的地方每次迭代可能使目标函数有较大的下降，但是在接近极小点的地方，由于锯齿现象，从而导致每次迭代行进距离缩短，因而收敛速度不快。

3.5 Python 实现代码及结果展示

代码：

```
import numpy as np
import sympy as sp
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#函数
def f(X):
    x1 = X[0][0]
```

```
    x2 = X[1][0]
    y = x1**2+x2**2-x1*x2-10*x1-4*x2+60
    return y
A = np.mat([[2,-1],[-1,2]])  #需要手动给出
#求梯度矩阵
def g(X):
    x1_value = np.array(X)[0][0]
    x2_value = np.array(X)[1][0]

    x1 = sp.Symbol('x1')
    x2 = sp.Symbol('x2')
    #求偏导
    f_x1 = sp.diff(f([[x1],[x2]]),x1)
    f_x2 = sp.diff(f([[x1],[x2]]),x2)

    return np.mat([f_x1.evalf(subs={x1:x1_value,x2:x2_value}),
                   f_x2.evalf(subs={x1:x1_value,x2:x2_value})]).T

#最速下降法
def Steepest_Descent(X_0,stop_value = 0.01):
    Iteration_times = 0
    X_0 = np.mat(X_0)

    f_0 = f(X_0)
    g_0 = g(X_0)

    while 1:
        Iteration_times += 1
        print("************* ", Iteration_times , "****************")
        #计算 K+1 个点的位置
        X_1 = X_0 - float( (g_0.T*g_0) / (g_0.T*A*g_0) )*g_0
        f_1 = f(X_1)
        g_1 = g(X_1)
        print("X_0 = \n",np.array(X_0),'\nf_0 = \n',f_0,'\ng_0 = \n',g
_0,"\n")
        print("X_1 = \n",np.array(X_1),'\nf_1 = \n',f_1,'\ng_1 = \n',g
_1,"\n")
        #终止条件
        if abs(f_1 - f_0) < stop_value:
            break
        #继续迭代，更新点位置
        X_0 = X_1
        f_0 = f_1
        g_0 = g_1
```

```
    return X_1,f_1,Iteration_times-1

if __name__ == "__main__":
    plt.figure()  #定义新的三维坐标轴
    ax = plt.axes(projection='3d')
    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('Y')
    #定义三维数据
    x1 = np.arange(-10,20,0.5)
    x2 = np.arange(-10,20,0.5)
    X1, X2 = np.meshgrid(x1, x2)
    Y = f([[X1],[X2]])
    #作图
    ax.plot_surface(X1,X2,Y,rstride = 1, cstride = 1,cmap='rainbow')
    ax.contour(X1, X2, Y, offset=0, cmap='rainbow')   #等高线图

    print("****************** 最速下降法 ******************")
    X_out,f_out,times = Steepest_Descent([[0],[0]],stop_value = 0.01)
    print("起始点：[0,0]，经过",times,"次迭代，找出最优解：\n 当 X = (",
    float(X_out[0][0]),",",float(X_out[1][0]),") 时，minf(X) = ",
    float(f_out))

    ax.scatter(float(X_out[0][0]),float(X_out[1][0]),float(f_out),c =
'r')
    plt.show()
```

结果：

```
****************************** 最速下降法 ******************************
********************  1 ************************
X_0 =
 [[0]
 [0]]
f_0 =
 [[60]]
g_0 =
 [[-10.0000000000000]
 [-4.00000000000000]]

X_1 =
 [[7.63157894736842]
 [3.05263157894737]]
f_1 =
 [[15.7368421052632]]
g_1 =
 [[2.21052631578947]
 [-5.52631578947368]]
```

……

```
********************  6 ************************
X_0 =
 [[7.99184422531274]
 [5.93475380250196]]
f_0 =
 [[8.00379144966270]]
g_0 =
 [[0.0489346481235300]
 [-0.122336620308825]]

X_1 =
 [[7.97365057408733]
 [5.98023793056550]]
f_1 =
 [[8.00056411244982]]
g_1 =
 [[-0.0329367823908377]
 [-0.0131747129563351]]

起始点：[0,0]，经过 5 次迭代，找出最优解：
当X = ( 7.97365057408733 , 5.980237930565497 ) 时，minf(X) =  8.000564112449815
```

从上述结果可以看出：当初始点选择在[0，0]时，经过了 5 次迭代，才能达到 $\varepsilon = 0.01$ 的精度，可见最速下降法的收敛速度非常慢，与下面的牛顿法相比更为明显。简单分析可以看出：从上面绘制的函数图像来看，该函数的等值线具有狭长深谷形状，刚好达到最差的情况，用负梯度方向搜索，产生了上述提到的锯齿现象，因此收敛速度很不可观。

四、Newton 法

4.1 选择 Newton 法原因

由于最速下降法的收敛速度过慢，所以还想实现一种收敛速度快的方法，因此继续编程实现 Newton 法，以计算量的提升换取收敛速度，而且 Newton 法的思路并不是很复杂，只涉及到梯度和 Hesse 矩阵的计算，在 Python 中均可以很方便的实现，实现这一方法来和最速下降法做一对比。

4.2 选择 Newton 法介绍

如果目标函数 $f(X)$ 在 R^n 上具有连续的二阶偏导数，其 Hesse 矩阵 $G(X)$ 正定并且可以以表达成为显式（今后为了简便起见，记 $G(X)=\nabla^2 f(X)$），那么可以使用 Newton 法。这种方法一旦好用，收敛速度是很快的，它是一维搜索 Newton 切线法的推广。

为寻求收敛速度快的算法，我们考虑在应用基本迭代公式 $X_{k+1}=X_k+t_kP_k$ 中，每轮迭代在迭代的起始点 X_k 处用一个适当的二次函数来近似该点处的目标函数，由此用点 X_k 指向近似二次函数极小点的方向来构造搜索方向P_k。

设最优化问题 $minf(X)$，其中 $f:\boldsymbol{R}^n\to\boldsymbol{R}^1$ 二阶可导，**Hesse 矩阵 $\nabla^2 f(X)$ 正定。**

不妨设经过 k 次迭代已获点 X_k，现将 $f(X)$ 在 $X=X_k$ 处展成二阶泰勒公式，于是有：

$$f(X)\approx Q(X)=f(X_k)+\nabla f(X_k)^T(X-X_k)+\frac{1}{2}(X-X_k)^T\nabla^2 f(X_k)^T(X-X_k)$$

显然 $Q(X)$ 是二次函数，特别由假设知 $Q(X)$ 还是正定二次函数，所以 $Q(X)$ 是凸函数且存在唯一全局极小点。为求此极小点，令：

$$\nabla Q(X)=\nabla f(X_k)+\nabla^2 f(X_k)^T(X-X_k)=0$$

即可解得：

$$X=X_k-[\nabla^2 f(X_k)]^{-1}\nabla f(X_k)$$

亦即：

$$X_{k+1}=X_k-[\nabla^2 f(X_k)]^{-1}\nabla f(X_k)$$

对照基本迭代公式：

$$X_{k+1}=X_k+t_kP_k$$

易知，式中的搜索方向：

$$P_k = -[\nabla^2 f(X_k)]^{-1}\nabla f(X_k)$$

并取步长因子 $t_k = 1$，上式即为从 X_k 出发的 Newton 方向。

从初始点开始，每一轮从当前迭代点出发，沿 Newton 方向并取步长 $t_k = 1$ 的算法称为 Newton 法。

4.3 选择 Newton 法迭代步骤

已知目标函数 $f(X)$ 及其梯度 $g(X)$，Hesse 矩阵 $G(X)$，终止限 ε。

（1）选定初始点 X_0，计算$f_0 = f(X_0)$，$g_0 = g(X_0)$；置 $k = 0$。

（2）计算 $G_k = \nabla^2 f(X_k)$。

（3）由方程 $G_k P_k = -g_k$。

（4）计算 $X_{k+1} = X_k + P_k$，$f_{k+1} = f(X_{k+1})$，$g_{k+1} = g(X_{k+1})$

（5）用终止准则检验是否满足：若满足，则打印最优解 $(X_{k+1}, f(X_{k+1}))$，结束；否则，置 $k = k + 1$，转（2）。

4.4 Newton 法优缺点

优点：Newton 法收敛速度非常快，具有二次收敛的特性，用 Newton 法解目标函数为正定二次函数的无约束优化问题，只需一次迭代就可得最优解。

缺点：

（1）尽管每次迭代不会使目标函数 $f(X)$ 上升，但仍不能保证 $f(X)$ 下降。当 Hesse 矩阵非正定时，Newton 法的搜索将会失败。

（2）对初始点要求严格。一般要求比较接近或有利于接近极值点，而这在实际使用中是比较难办的。

（3）在进行某次迭代时可能求不出搜索方向。由于搜索方向为：

$$P_k = -[\nabla^2 f(X_k)]^{-1}\nabla f(X_k)$$

若目标函数在 X_k 点 Hesse 矩阵是非奇异的，则 $\left[\nabla^2 f(X_k)\right]^{-1}$ 不存在，因而不能构成 Newton 方向，从而使迭代无法进行。

（4）Newton 方向构造困难，计算相当复杂，除了求梯度以外还需要计算 Hesse 矩阵及其逆矩阵，占用机器内存相当大。

4.5 Python 实现代码及结果展示

代码：（库和一些相关函数在上面已经给出，不再重复）

```
#求 Hesse 矩阵
def G(X):
    x1_value = np.array(X)[0][0]
    x2_value = np.array(X)[1][0]
    x1 = sp.Symbol('x1')
    x2 = sp.Symbol('x2')
    #求偏导
    f_x1 = sp.diff(f([[x1],[x2]]),x1)
    f_x2 = sp.diff(f([[x1],[x2]]),x2)

    f_x1_x1 = sp.diff(f_x1,x1)
    f_x1_x2 = sp.diff(f_x1,x2)
    f_x2_x1 = sp.diff(f_x2,x1)
    f_x2_x2 = sp.diff(f_x2,x2)
    return np.mat([[float(f_x1_x1.evalf(subs={x1:x1_value,x2:x2_value}
)),
                    float(f_x1_x2.evalf(subs={x1:x1_value,x2:x2_value})
)],
                    [float(f_x2_x1.evalf(subs={x1:x1_value,x2:x2_value}
)),
                    float(f_x2_x2.evalf(subs={x1:x1_value,x2:x2_value})
)]])
#牛顿法
def Newton(X_0,stop_value = 0.01):
    Iteration_times = 0
    X_0 = np.mat(X_0)
    f_0 = f(X_0)
    g_0 = g(X_0)
```

```
    while 1:
        Iteration_times += 1
        print("************* ", Iteration_times , "***************")
        #计算 K+1 个点的位置
        X_1 = X_0 + G(X_0).I * (-1*g_0)   #t=1
        f_1 = f(X_1)
        g_1 = g(X_1)
        print("X_0 = \n",np.array(X_0),'\nf_0 = \n',f_0,"\n")
        print("X_1 = \n",np.array(X_1),'\nf_1 = \n',f_1,"\n")
        #终止条件
        if abs(f_1 - f_0) < stop_value:
            break
        #继续迭代，更新点位置
        X_0 = X_1
        f_0 = f_1
        g_0 = g_1
    return X_1,f_1,Iteration_times-1
```

结果：

```
****************************** 牛顿法 ******************************
********************  1 ************************
X_0 =
 [[0]
 [0]]
f_0 =
 [[60]]

X_1 =
 [[8.0000000000000]
 [6.0000000000000]]
f_1 =
 [[8.0000000000001]]

********************  2 ************************
X_0 =
 [[8.0000000000000]
 [6.0000000000000]]
f_0 =
 [[8.0000000000001]]

X_1 =
 [[8.0000000000000]
 [6.0000000000000]]
f_1 =
 [[8.0000000000000]]

起始点：[0,0]，经过 1 次迭代，找出最优解：
当X = ( 8.0 , 6.0 ) 时，minf(X) =  8.0
PS C:\Users\RJJ\Desktop\最优化\大作业> python .\chapter5.py
```

从上图可以明显看出：仅经过一次迭代，算法就能找出最优解，而且精度非常的高，效率比最速下降法高了很多。这也正因为目标函数是正定二次函数，所以 Newton 方向就是指向极小点的方向，一次迭代就能找到最优解。但从程序中也可以发现，虽然用了库，但计算 Hesse 矩阵还是很耗费时间和空间的。因此，Newton 法相当于牺牲了计算量来换取收敛速度的加快，在实验的函数中表现良好，但适用范围很小，需要靠修正 Newton 法、共轭梯度法等改进。

由于精力原因，第四章实现所有方法，第五章仅实习前两种算法，以上所有公式均可编辑，代码均可正常运行，谢谢老师。