第六章 非线性方程求根与最优化问题

上一章主要介绍了线性问题的数值求解算法。然而，真实世界更普遍的是非线性系统，数值求解非线性方程是物理学中更为常见的计算问题[13]。本章将介绍非线性方程求解的数值方法，并介绍简单的问题优化算法。

6.1 非线性方程求根

方程求根的目标是找到方程$f(x)=0$在给定范围的根。常见的非线性方程求根算法有搜索法、二分法、牛顿-拉普逊法、以及弦割法等。本章将分别介绍以上算法。

搜索法是最直接的非线性求根算法，其基本思路是通过定向搜索并逐渐缩短搜索步长，从而找到满足给定精度的方程的根。图 6.1 的黑色实现表示函数$f(x)$，则求根过程可以转换为寻找曲线$f(x)$与 x 轴交点 x_0 的过程，具体算法如下：

步骤 1：设定搜索初始值$x=x^i$，使其数值小于待求方程根x_0，同时设定搜索步长初始值Δx和精度要求ε；

步骤 2：计算函数值$f(x^i)$；

步骤 3：以步长Δx更新x，即$x=x^{i+1}=x^i+\Delta x$，并计算$f(x^{i+1})$的数值。若$f(x^{i+1})<\varepsilon$；则x^{i+1}为满足精度要求的根，停止搜索，否则进行下一步。

步骤 4：判断$f(x^i)\,f(x^{i+1})>0$是否成立，如成立，则$x=x^{i+1}$为新的搜索值，并返回步骤 3，否则$x=x^i$，并将步长减半，即$\Delta x=\Delta x/2$，并返回步骤 3。

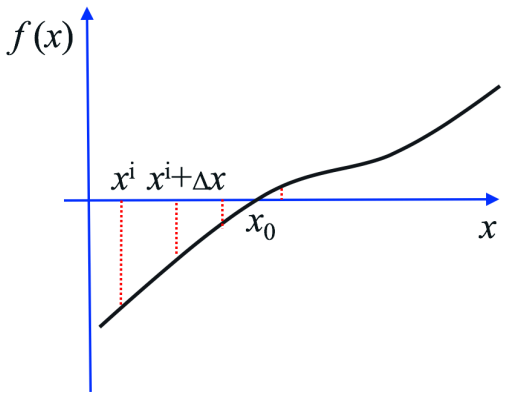

图 6.1 搜索法求根示意图。

Code-6-1 给出采用搜索法求解非线性方程 $f(x)=x^2-8=0$ 正根的过程。其中搜索初始值设为 1.0，搜索步长设为 1.0，精度要求为 10^{-6}。执行 code-6-1 得到的结果如图 6.1 所示。经过 34 步迭代搜索，得到满足精度要求的根为 2.828427.

```
# -*- coding: utf-8 -*-
# 搜索法
import pylab as pl
#  defining function form

def f(x):  #定义函数
    return x**2 - 8.0

eps = 0.000001 # 精度截断
dx = 1.0  # 初始步长
x=1     # 初始试探值

fold = f(x)
count = 0  #计数器
xroot = [] #空列表存储每步的根
iter = []  #空列表存储迭代次数

while abs(fold) > eps:  #搜索根
   count = count + 1
   x = x + dx
   fnew = f(x)
   if fnew*fold < 0:
      x = x - dx
      dx = dx/2
   fold = f(x)
   xroot.append(x)
   iter.append(count)
#----plotting ------------------
pl.plot(iter,xroot,'r-o',label='search')
pl.xlabel('iter',fontsize=15)
pl.ylabel('xroot',fontsize=15)
pl.xlim(0,35)
pl.ylim(1,3.5)
pl.legend(loc='lower right',fontsize=15)
pl.show()
print("found the root of x2-8=0 at x= %0.6f" %x)

```

Code-6-1

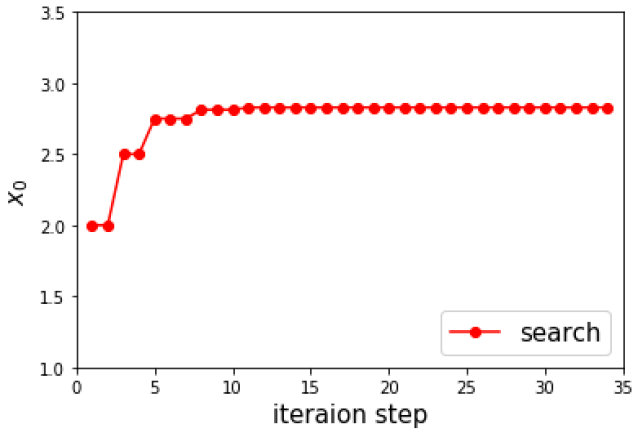

图 6.2 搜索法得到的方程$x^2-8=0$的正根x_0随搜索步数的变化。

另一个常见的求根算法为二分法，其基本思路是逐渐缩小包含方程根的区间，直至满足精度要求，实现步骤如下：

步骤 1：设定包含方程根的初始区间边界[a, b]，且$f(a)\,f(b)<0$，并设定精度要求ε；

步骤 2：计算新区间边界$x^i=(a+b)/2$，并计算$f(x^i)$的数值；

步骤 3：判断$f(x^i)<\varepsilon$是否成立。如果成立，则x^i为满足精度要求的根，停止迭代，否则进行下一步。

步骤 4：判断$f(x^i)\,f(a)<0$是否成立，如成立则$b=x^i$，否则$a=x^i$，返回步骤 2。

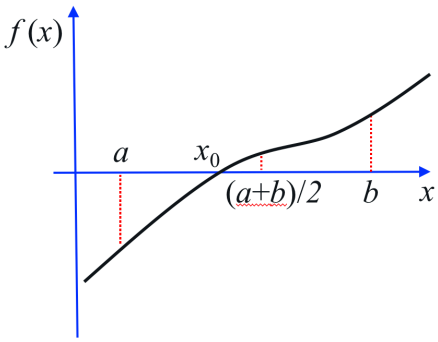

图 6.3　二分法求根示意图。

Code-6-2 给出了二分法求解非线性方程 $\boldsymbol{f(x)=x^2-8=0}$ 正根的过程。其中初始边界设为[0, 20]，精度要求为 10^{-6}。执行 code-6-2 得到的结果如图 6.2 所示。经过 24 步迭代搜索，得到满足精度要求的根为 2.828427。

```
# -*- coding: utf-8 -*-
# 二分法
import pylab as pl
def f(x): # 定义函数
    return x**2 - 8.0
eps = 0.000001 #精度截断
xa = 0.0  #初始区间
xb = 20.0  #初始区间
count = 0  #计数器
xroot = []
iter = []
N = 1000
for i in range(0, N):
    x = (xa + xb)/2
    fa = f(xa)
    fb = f(xb)
    fx = f(x)
    xroot.append(x)
    iter.append(i)
    if (fa*fx > 0.):
        xa = x
    else:
        xb = x
    if (abs(fx) < eps):
        print("found the root of x2-5=0 at x= %0.6f" %x)
        break
    if (i == N-1):
        print ("\n root not found after %d iterations" %N)
        break
#----plotting ------------------
pl.plot(iter,xroot,'r-o',label='bisection')
pl.xlabel('iteration',fontsize=15)
pl.ylabel('$x_0$',fontsize=15)
pl.xlim(0,35)
pl.ylim(1.5,4.5)
pl.legend(loc='lower right',fontsize=15)
pl.show()
```

Code-6-2

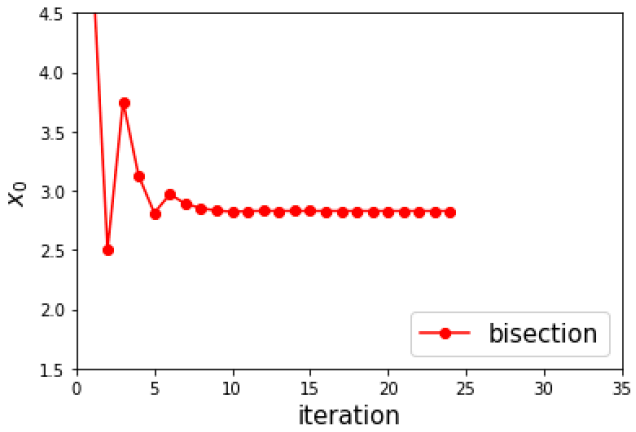

图 6.4 二分法得到的方程$x^2-8=0$的正根x_0随搜索步数的变化。

以上搜索法和二分法普适性较强，通常能够找到给定区间的所有根，而对方程的具体形式不敏感。但是以上两种方法计算效率较低，从而限制了其应用范围。牛顿-拉普逊法是一种高效率的非线性方程求根算法。相对于搜索法，牛顿-拉普逊法不仅利用了函数值的信息，而且利用了函数导数的信息，因此每一步的搜索目标更为明确。牛顿-拉普逊法主要思路是用直线方程代替原函数方程，通过迭代的方式得到方程的准确根。如图 6.5 所示，x^i为方程根的初始猜测值，则用x^i点的原函数切线代替原函数$f(x)$，其与 x 轴的交点为原方程根的近似值，即：

$$x^{i+1}=x^i-\frac{f(x^i)}{f'(x^i)} \qquad (6.1)$$

上式给出了牛顿-拉普逊方法求根的迭代格式，具体的实现步骤如下：

步骤 1：设定初始猜测根x^i；

步骤 2：计算$f(x^i)$和$f'(x^i)$的数值，判断$f(x^i)<\varepsilon$是否成立，若成立，则x^i为满足精度要求的根，否则执行下一步；

步骤 3：根据式(6.1)计算x^{i+1}，并令$x^i=x^{i+1}$；

重复步骤 2 和步骤 3 可的方程的根。

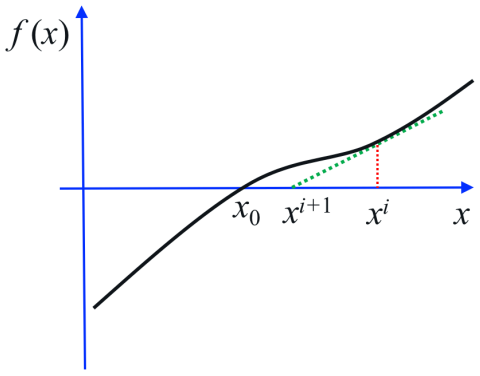

图 6.5　牛顿-拉普逊法求根示意图。

Code-6-3 给出了牛顿-拉普逊法求解非线性方程 $f(x) = x^2 - 8 = 0$ 的正根的过程。其中初始猜测根设为 1.0，精度要求为 10^{-6}。执行 code-6-3 得到的结果如图 6.6 所示。经过 4 步迭代搜索，得到满足精度要求的根为 2.828427。可见，相对于搜索法和二分法，由于利用了导数值的信息，牛顿-拉普逊方法的求根效率得到了大幅提升，这也使得牛顿-拉普逊法成为应用最广泛的非线性方程求根算法。

```python
# -*- coding: utf-8 -*-
# 牛顿法
import pylab as pl
def f(x):# 定义函数
    return x**2 - 8.0
def fprim(x):
    return 2*x
eps = 0.000001 #精度截断
xold = 1.0 #初始试探值
x = xold - f(xold)/fprim(xold)
xroot = []
iter= []
N = 1000

for i in range(0, N):
    xold = x
    x = xold - f(xold)/fprim(xold)
    xroot.append(x)
    iter.append(i)

    if (abs(f(x)) < eps):
        print("found the root of x2-5=0 at x= %0.6f" %x)
        break
    if (i == N-1):
        print ("\n root not found after %d iterations" %N)
        break
#----plotting ------------------
pl.plot(iter,xroot,'r-o',label='Newton-Raphson')
pl.xlabel('iteration',fontsize=15)
pl.ylabel('$x_0$',fontsize=15)
pl.xlim(0,35)
pl.ylim(1,3.5)
pl.legend(loc='lower right',fontsize=15)
pl.show()
pl.savefig('Newton-Raphson.png', dpi=150) #将图存储为png格式文件

```

Code-6-3

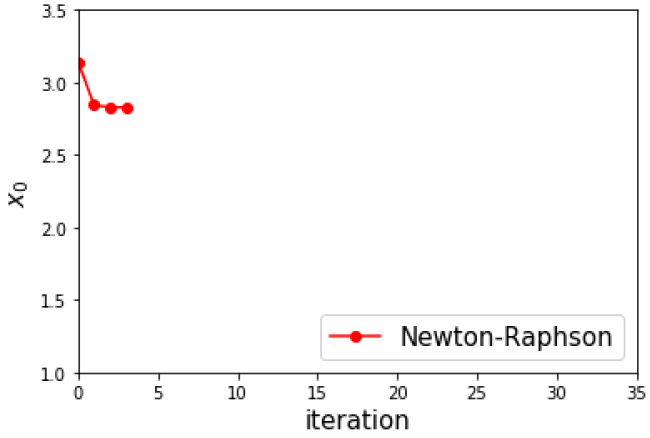

图 6.6 牛顿-拉普逊法得到的方程$x^2 - 8 = 0$的正根x_0随迭代步数的变化。

以上牛顿-拉普逊方法依赖于对函数$f(x)$的导数值的计算。在一些情况下，函数可能是以数值的形式给出，这时无法解析求解导数。解决的办法是用数值差分代替导数，即

$$f'(x') = \frac{f(x^i)-f(x^{i-1})}{x^i-x^{i-1}} \quad (6.2)$$

则式（6.1）的迭代格式可以写为

$$x^{i+1} = x^i - f(x^i)\frac{x^i-x^{i-1}}{f(x^i)-f(x^{i-1})x^i-x^{i-1}} \quad (6.3)$$

式（6.3）给出的求根迭代算法称为弦割法。相对于牛顿-拉普逊方法，直线方程不再对应x^i点的切线方程，而是由x^i和x^{i-1}两个点的函数值构建。因此，下一步迭代计算的方程根x^{i+1}不仅依赖于x^i点的函数值，而且依赖于x^{i-1}点的函数值，具体实现步骤如下：

步骤 1：设定初始猜测根x^{i-1}，x^i；

步骤 2：计算$f(x^{i-1})$和$f(x^i)$的数值，判断$f(x^i) < \varepsilon$是否成立，若成立，则x^i为满足精度要求的根，否则执行下一步；

步骤 3：根据式(6.3)计算x^{i+1}，并令$x^{i-1} = x^i$，$x^i = x^{i+1}$；

重复步骤 2 和步骤 3 可得方程的根。

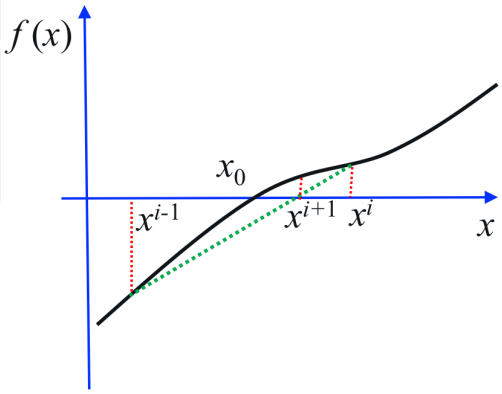

图 6.7　弦割法求根示意图。

Code-6-4 给出了弦割法求解非线性方程 $\boldsymbol{f(x) = x^2 - 8 = 0}$ 正根的过程。其中初始猜测根设为$x^{i-1} = 0.5$和$x^i = 1.0$，精度要求为 10^{-6}。执行 code-6-4 得到的结果如图 6.8 所示。经过 6 步迭代搜索，得到满足精度要求的根为 2.828427，其计算效率和牛顿-拉普逊法相近。

```python
# -*- coding: utf-8 -*-
#弦割法
import pylab as pl
def f(x):# 定义函数
    return x**2 - 8.0
eps = 0.000001 #精度截断
x1 = 0.5 #初始试探值
x2 = 1.0 #初始试探值
x3 = x2 - f(x2)*(x2-x1)/(f(x2)-f(x1))
xroot = []
iter = []
N = 1000
for i in range(0, N):
    if (abs(f(x3)) < eps):
        print("found the root of x2-8=0 at x= %0.6f" %x3)
        break
    if (i == N-1):
        print ("\n root not found after %d iterations" %N)
        break
    x1 = x2
    x2 = x3
    x3 = x2 - f(x2)*(x2-x1)/(f(x2)-f(x1))
    xroot.append(x3)
    iter.append(i)
#----plotting ------------------
pl.plot(iter,xroot,'r-o',label='Chord Secant')
pl.xlabel('iteration',fontsize=15)
pl.ylabel('$x_0$',fontsize=15)
pl.xlim(0,35)
pl.ylim(1,3)
pl.legend(loc='lower right',fontsize=15)
pl.show()
```

Code-6-4

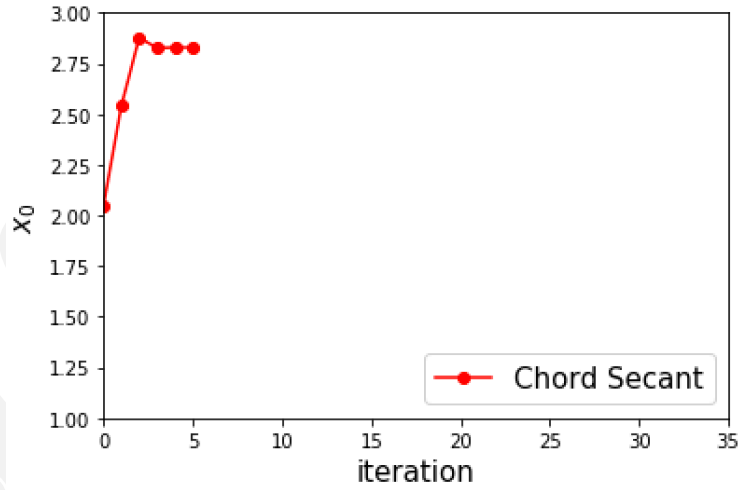

图 6.8 弦割法得到的方程$x^2-8=0$的正根x_0随迭代步数的变化。

以上非线性方程求根算法各有优缺点。牛顿-拉普逊法和弦割法计算效率高，但如果函数在方程根附近存在拐点时，则会遇到不收敛的问题。而搜索法和二分法对函数的具体形式不太依赖，但缺点是越逼近真实根时，其计算效率越低。因此，一个有效的做法是先用搜索法或二分法找到方程根的大致位置，然后再基于牛顿-拉普逊法或弦割法进一步迭代得到更精确的方程根。

6.2 无限深势阱本征值问题求解：打靶法

描述量子系统微观运动的薛定谔方程只在一些特殊情况下才有精确解析解。而在绝大部分的情况下，需要利用数值方法求解。打靶法是一类重要的求解本征值问题的方法。这里我们以无限深势阱本征能量和本征函数的求解为例（图 6.9 左），介绍打靶法的实现过程。

描述微观粒子运动的定态薛定谔方程为：

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2}+V(x)=E\psi \qquad (6.4)$$

其中

$$V(x)=\begin{cases}\infty; & x\le 0 \text{ or } x\ge 1.0\\ 0; & 0<x\le 1.0\end{cases}$$

为无限深势阱中质量为 m 的粒子的势能。令

$$k^2=\frac{2m}{\hbar^2}[E-V(x)]$$

则方程(6.4)可写为如下形式：

$$\frac{d^2\psi}{dx^2}+k^2\psi=0 \qquad (6.5)$$

式（6.5）为线性齐次方程。根据波函数的物理意义，其方程的解满足边界条件:$\psi(0)=0,\ \psi(L)=0$。对这类方程，只有在特定的一系列 k 值下，方程的解才能满足边界条件。这些能够找到满足方程边界条件解的k_n值即为方程的本征值，所对应的解ψ_n为本征函数。以上无限深势阱问题与两端固定的自由弦振动问题为同一类问题（图 6.9 右），求解方法相同。

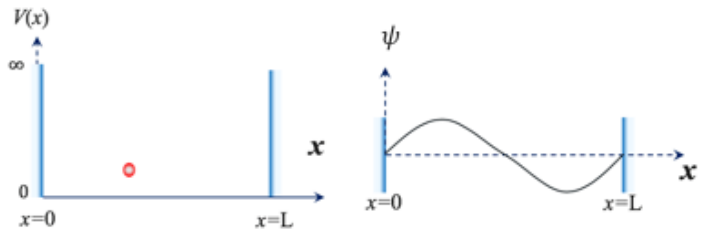

图 6.9 无限深势阱问题（左）和两端固定弦振动问题（右）示意图。

打靶法的基本思路是把寻找本征值k_n的问题看作是非线性方程求根的问题，而非线性方程由薛定谔方程和边界条件给定。具体实现时，先猜测一初始k_n值，将方程(6.5)作为满足$\psi(0)=0$的初值问题来求解二阶微分方程（6.5），得到右边界波函数的值$\psi(L)$。若$\psi(L)$满足右边界条件，即$\psi(L)=0$，则初始k_n为方程的本征值。采用打靶法求本征值问题的实现步骤如下：

步骤 1：设定初始试探 k 值、搜索步长Δk, 以及精度要求ε；

步骤 2：设定条件$\psi(0)=0$和$\psi'(0)=\alpha$，求解二阶微分方程式（6.5），得到$\psi(L)$的值。其中α为任意小量。由于方程为齐次线性微分方程，因此选取不同的α值，不影响本征值和归一化本征函数的求解。

步骤 3：判断$|\psi(L)|<\varepsilon$是否成立；若成立，则 k 为满足边界条件的本征值；否则根据搜索法规则继续搜索新的 k 值。

重复步骤 2 和步骤 3 可得本征值和本征函数。从不同的初始 k 值出发，可以得到不同范围内的本征值和本征函数。

Code-6-5 给出的是具体的实现过程。其中，势阱宽度 L=1, 精度要求为 10^{-6}，初始 k 值为 20, k 值搜索初始步长为$\Delta k=0.1$，$\alpha=0.01$, 微分方程求解步长为 0.01.

```python
# -*- coding: utf-8 -*-
#无限深势阱本征值问题
from numpy import zeros
import pylab as pl
# some parameters
L = 1.0  # 区间长度
N = 100
dt = L/N  #x步长
eps = 0.000001 #精度阶段
k =20.0  #k的初始试探值
dk = 0.1  #k的初始步长
freturn = zeros(2)
psiphi = zeros(2)
psiphitemp = zeros(2)
x = zeros(N)
psi = zeros(N)
def f1(psiphi,k):  #定义子函数
    freturn[0] = -k*k*psiphi[1]   #phi的斜率 -k^2 phi
    freturn[1] = psiphi[0] #psi的斜率  psi
    return freturn
def rk4(dt, k, N): #四阶龙格-库塔法求出给定k时, psi(1)的值
    psiphi[0] = 0.01  #phi的初值, 即左端点的phi值
    psiphi[1] = 0.0 #psi的初值, 即左端点的psi值
    t = 0
    for i in range(N):
        fR = f1(psiphi, k)
        k1 = fR[0]
        l1 = fR[1]

        psiphitemp[0] = psiphi[0] + k1*dt/2
        psiphitemp[1] = psiphi[1] + l1*dt/2
        fR = f1(psiphitemp, k)
        k2 = fR[0]
        l2 = fR[1]

        psiphitemp[0] = psiphi[0] + k2*dt/2
        psiphitemp[1] = psiphi[1] + l2*dt/2
        fR = f1(psiphitemp, k)
        k3 = fR[0]
        l3 = fR[1]

        psiphitemp[0] = psiphi[0] + k3*dt
        psiphitemp[1] = psiphi[1] + l3*dt
        fR = f1(psiphitemp, k)
        k4 = fR[0]
        l4 = fR[1]

        psiphi[0] = psiphi[0] + (k1 + 2*k2 + 2*k3 +k4)*dt/6
        psiphi[1] = psiphi[1] + (l1 + 2*l2 + 2*l3 +l4)*dt/6

        x[i] = t
        psi[i] = psiphi[1]
        t = t + dt
    return psiphi
psiphi = rk4(dt, k, N)
phiold = psiphi[1]
while abs(dk) > eps:  #搜索法求出满足psi(1)=0的k值
    k = k + dk
    psiphi = rk4(dt,k,N)
    phinew = psiphi[1]
    if phinew*phiold > 0:
        continue
    k = k - dk
    dk = dk/2
print('k=',k)
pl.plot(x, psi, 'r-',linewidth=1.0)
pl.xlabel('x',fontsize=15)
pl.ylabel('$\psi$',fontsize=15)
pl.show()
```

Code-6-5

选取不同的初始试探 k 值，可以得到不同的本征值与本征函数。例如，计算得到的最低的三个本征值为 k_1=3.141592, k_2=6.283185, k_3=9.424783，而相应的本征函数如图 6.10 所示，结果能够与解析解($k_n=n\pi$, $\psi_n=A\cdot\sin(n\pi x)$)很好符合。

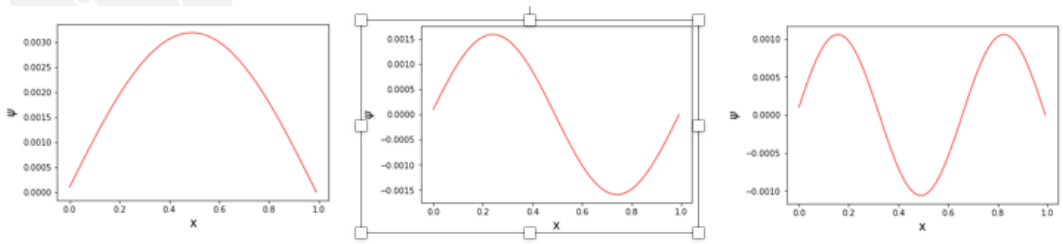

图 6.10. 对应最低三个本征值的本征函数。

6.3 最优化问题：最速下降法

最优化问题普遍存在于物理问题的求解过程中。例如，分子结构预测对应于寻找能量最低的分子构象问题（如 6.11）。寻找满足实验约束的理论模型参数是在参数空间中寻找对应最小残差的点。其他领域也会经常遇到最优化问题，例如旅行商问题、投资最大收益问题等。因此，寻找问题的最优解是基本的数值计算问题。最优化方法分为两类，包括局部最优算法和全局最有算法。局部最优只能找到局部最优解，典型的算法有最速下降法和共轭梯度法。而全局算法可以找到全局最优解，例如遗传算法、模拟退火算法等。这里主要介绍最速下降法和共轭梯度法两个局部最优算法，而模拟退火算法将在第七章介绍。

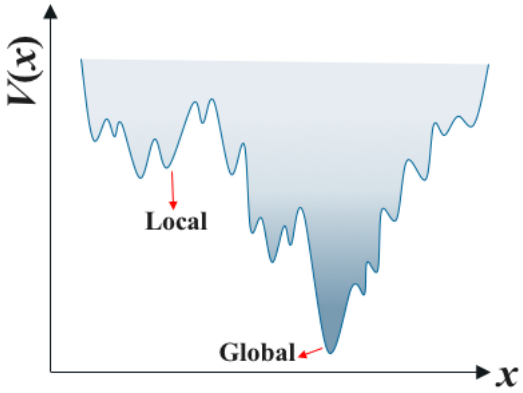

图 6.11. 粗糙能量面最优化问题示意图。

最速下降法的思路是最优解的搜索沿能量下降最快的方向，即负梯度的方向进行。例如，对于二维空间能量函数$U(x,y)=x^2+10y^2+4xy$，其能量最低点可由最速下降法得到。具体步骤如下：

步骤 1: 设定精度要求ϵ，初始搜索位置和搜索步长 h；
步骤 2: 计算负梯度方向，并沿负梯度方向移动距离 h，得到试探新位置(x',y')，并计算新位置的能量$U(x',y')$；
步骤 3: 如果$U(x',y')>U(x,y)$成立，则试探新位置舍弃，并令 $h=h/2$；反之将当前位置设置为试探新位置，即$(x,y)=(x',y')$。
步骤 4: 判断 h<ϵ是否成立，若成立则找到局部最优解，结束计算；否则返回步骤 2.

Code-6-6 给出了二维空间能量函数$U(x,y)=x^2+10y^2+4xy$最低能量点的搜索过程，其中初始搜索步长设为 1.0，初始位置设为(12,11)，精度要求ϵ设为 10^{-6}。得到的能量最低点为(2.333662e-07 9.163557e-07)，与精确解(0,0)接近。需要指出的是，以上计算步骤中搜索步长是直接给定的。在实际应用中，搜索步长也可以通过优化当前搜索步的能量函数来确定。

```python
# -*- coding: utf-8 -*-
#最陡下降优化
import numpy as np
import pylab as pl

F=np.zeros([100,100])
for i in np.arange(0,100): #函数
    for j in np.arange(0,100):
        x0 = i*0.2-10
        y0 = j*0.2-10
        F[i][j] = x0**2+10*y0**2+4*x0*y0
x=10.234  #初始位置
y=11.537  #初始位置
deltar=1.0  #初始步长
fold = x**2+10*y**2+4*x*y #函数值
xx=[]
yy=[]
xx.append(x)
yy.append(y)
for i in np.arange(0,1000):
    dfdx = -2*x-4*y  #梯度
    dfdy = -20*y-4*x #梯度
    norm = (dfdx**2+dfdy**2)**0.5 #归一化
    dx = dfdx/norm #梯度方向
    dy = dfdy/norm #梯度方向
    deltax = deltar * dx #搜索步长 x
    deltay = deltar * dy #搜索步长 y
    x = x + deltax  #更新x
    y = y + deltay  #更新y
    fnew = x**2+10*y**2+4*x*y #新函数值
    if (fnew>fold):
        x = x - deltax
        y = y - deltay
        deltar = deltar/2.0
    fold = fnew
    xx.append(x)
    yy.append(y)
    if(deltar<1.0e-6):
        print x,y
        break
extent = [-12, 12, -12, 12]
fig = pl.figure(figsize=(10,4))
# contour line
ax1 =fig.add_subplot(1,2,1)
levels = np.arange(0.0,2000.0,50.0)
cs = ax1.contour(F,levels,origin='lower', \
                 linewidths=2,extent=extent)
ax1.clabel(cs)
ax1.plot(xx, yy, 'r-',linewidth=1.0)
ax1.set_ylabel(r'Y', fontsize=20)
ax1.set_xlabel(r'X', fontsize=20)
ax1.set_xlim(-1,12)
ax1.set_ylim(-12,12)
pl.show()
```

Code-6-6

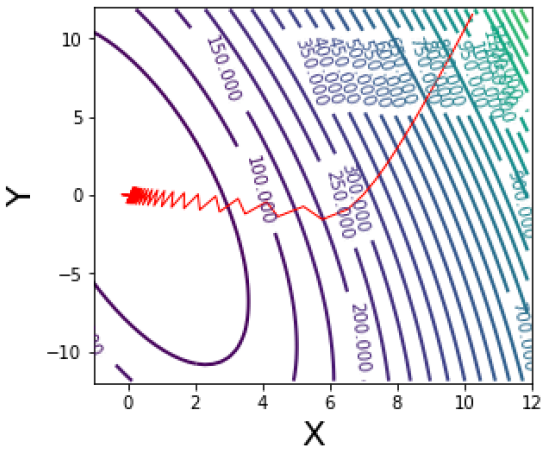

图 6.12. 最速下降法寻找最低能量点的轨迹。

以上最速下降优化法得到的搜索轨迹在接近最优点时，存在震荡现象，因此会影响最优解的搜索效率。共轭梯度法可以克服这一震荡现象。相对于最速下降法，共轭梯度法不仅利用了当前梯度信息，还整合了历史搜索信息，即：

$$\Delta(x_n, y_n) = G(x_n, y_n) + \lambda_n \Delta(x_{n-1}, y_{n-1}) \qquad (6.6)$$

其中$\Delta(x_n, y_n)$为当前搜索方向，$G(x_n, y_n)$为当前梯度，$\Delta(x_{n-1}, y_{n-1})$为上一步搜索方向。λ_n决定了对梯度和历史信息的整合方式。λ_n有多种选取方式，其中 Fletcher-Reeves 法是比较常见的一种方法，即：

$$\lambda_n = \frac{|G(x_n,y_n)|^2}{|G(x_{n-1},y_{n-1})|^2} \quad (6.7)$$

Code-6-7 给出了采用共轭梯度法来求解最优化求解的实现过程，得到的结果如图 6.13 所示。相对于最速下降法，最优解附近的震荡行为得到改善。共轭梯度法由于增加了历史信息的整合过程，因此计算复杂度增加。通常在实际使用中，先用最速下降法找到离最优解较接近的位置，然后改为共轭梯度法，可以实现最高效率的求解。

```python
# -*- coding: utf-8 -*-
#共轭梯度优化
import numpy as np
import pylab as pl

F=np.zeros([100,100])
for i in np.arange(0,100): #函数
    for j in np.arange(0,100):
        x0 = i*0.2-10
        y0 = j*0.2-10
        F[i][j] = x0**2+10*y0**2+4*x0*y0
x=10.234  #初始位置
y=11.537  #初始位置
dx_old = 0
dy_old = 0
dfdx_old = 0
dfdy_old = 0
deltar=1.0  #初始步长
fold = x**2+10*y**2+4*x*y #函数值
xx=[]
yy=[]
xx.append(x)
yy.append(y)
for i in np.arange(0,2000):
    dfdx = -2*x-4*y  #梯度
    dfdy = -20*y-4*x #梯度
    if i > 0:
        beta = (dfdx**2+dfdy**2)/(dfdx_old**2+dfdy_old**2)
    else:
        beta = 0

    dfdx0 = dfdx/(dfdx**2+dfdy**2)**0.5 #当前梯度方向
    dfdy0 = dfdy/(dfdx**2+dfdy**2)**0.5

    dx = dfdx0 + beta*dx_old   #当前搜索方向
    dy = dfdy0 + beta*dy_old

    norm = (dx**2 + dy**2)**0.5 #归一化
    dx = dx/norm
    dy = dy/norm
    dx_old = dx
    dy_old = dy
    dfdx_old = dfdx
    dfdy_old = dfdy
    deltax = deltar * dx #搜索步长 x
    deltay = deltar * dy #搜索步长 y
    x = x + deltax  #更新x
    y = y + deltay  #更新y
    fnew = x**2+10*y**2+4*x*y #新函数值
    if (fnew>fold):
        x = x - deltax
        y = y - deltay
        deltar = deltar/2.0
    fold = fnew
    xx.append(x)
    yy.append(y)
    if(deltar<1.0e-6):
        print x,y,deltax,deltay,deltar
        break
extent = [-12, 12, -12, 12]
fig = pl.figure(figsize=(10,4))
# contour line
ax1 =fig.add_subplot(1,2,1)
levels = np.arange(0.0,2000.0,50.0)
cs = ax1.contour(F,levels,origin='lower',\
                 linewidths=2,extent=extent)
ax1.clabel(cs)
ax1.plot(xx, yy, 'r-',linewidth=1.0)
ax1.set_ylabel(r'Y', fontsize=20)
ax1.set_xlabel(r'X', fontsize=20)
ax1.set_xlim(-1,12)
ax1.set_ylim(-12,12)
pl.show()
```

Code-6-7

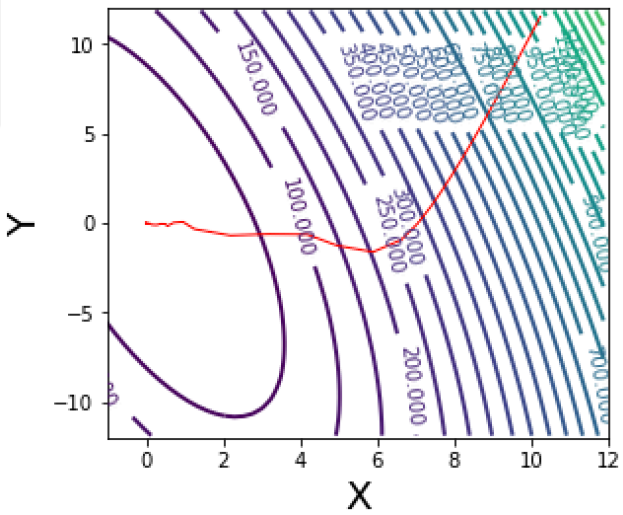

图 6.13 共轭梯度法寻找最低能量点的轨迹