利用迭代法求解定非线形方程及方程组

欧阳鑫健， 4121156012， 电信学部

要求：

1） 误差不超过10^{-8}

2） 必要时可应用迭代加速技术

Q1

7.2 利用简单迭代法、牛顿法、弦割法求解方程

$$f(x) = x^6 - 5x^5 + 3x^4 + x^3 - 7x^2 + 7x - 20 = 0$$

在区间 $[-1, 5]$ 内的全部实根 (先用二分法作根的隔离).

In [1]:

```
import numpy as np
from math import *

Delta = 1e-8
```

用二分法实现根的隔离

先确定方程 $f(x) = 0$ 的所有实根所在的区间 $[a, b]$，再按照选定的步长 $h = (b-a)/n$（n为正整数），取点 $x_k = a + k_h (k = 0, 1, \dots, n)$，逐次计算函数值 $f(x_k)$，依据函数值的异号及实根的个数确定根的隔离区间。

In [2]:

```
def f(x):
    return pow(x,6)-5*pow(x,5)+3*pow(x,4)+pow(x,3)-7*pow(x,2)+7*x-20

a, b = -1, 5
n = 100
h = (b-a)/n
h = round(h,3)
R, X = [], []
for x in np.arange(a,b+h,h):
    x=round(x,3)
    X.append(x)
    #零点存在定理
    if f(x)*f(x+h)<=0:
        R.append([x,x+h])

print('根的区间',R)
```

```
根的区间 [[4.28, 4.34]]
```

简单迭代法+松弛加速技术

$$x_{k+1} = \phi(x_k)$$

In [3]:

```
# 定义合适的迭代格式
def Phi(x):
        return -(pow(x,6)-5*pow(x,5)+3*pow(x,4)+pow(x,3)-7*pow(x,2)-20)/7
```

Note

上述迭代格式不收敛,使用松弛加速技术使其收敛.

将原来的方程$x = \phi(x)$作同解变形，在方程两端减去$\omega x (\omega \neq 1)$（ω称为**松弛因子**），得$x - \omega x = \phi(x) - \omega x$. 由此可得

$$x = \frac{\phi(x) - \omega x}{1 - \omega} \triangleq \psi(x),$$

则有$x = \psi(x)$. 由此可得迭代格式

$$x_{k+1} = \psi(x_k) = \frac{\phi(x_k) - \omega x_k}{1 - \omega}.$$

通常取$\omega = \phi'(\bar{x})$, 其中$\bar{x}$ 是x^*的一个好的近似值（例如$\bar{x}$用二分法求得.），得迭代格式

$$x_{k+1} = \frac{\phi(x_k) - \phi'(\bar{x})\, x_k}{1 - \phi'(\bar{x})}.$$

虽然该迭代格式的$\psi'(x^*) \neq 0$，但$|\psi'(x^*)| < |\phi'(x^*)|$. 这就大大的提高了收敛速度.

In [4]:

```
K = 20
xk = 4.28 # 初值选为区间左端点
delta_k = np.inf
# 选定松弛因子
x = (4.28+4.34)/2
w = -(6*pow(x,5)-25*pow(x,4)+12*pow(x,3)+3*pow(x,2)-14*x)/7
# print(x,w)

for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        # 简单迭代法
        xk1 = (Phi(xk)-w*xk)/(1-w)
        delta_k = abs(xk1-xk)
        xk = xk1
        print("step",k,": x", xk, "error",delta_k)
```

```
step 0 : x 4.333375465873948 error 0.05337546587394737
step 1 : x 4.333777323662569 error 0.00040185778862156244
step 2 : x 4.333754176483076 error 2.314717949314371e-05
step 3 : x 4.33375552066089 error 1.3441778143885585e-06
step 4 : x 4.333755442639675 error 7.802121526623296e-08
step 5 : x 4.333755447168447 error 4.528772024059435e-09
iterations end
```

In [5]:

```
print("solution x:", xk,'\n f(x):',f(xk))
```

```
solution x: 4.333755447168447
 f(x): 3.31195469271961e-07
```

牛顿法

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

In [6]:

```
def f1(x):
    return 6*pow(x,5)-25*pow(x,4)+12*pow(x,3)+3*pow(x,2)-14*x+7

x0 = 4.28
K = 10
xk = x0
delta_k = np.inf
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        # 牛顿法
        delta_x = f(xk)/f1(xk)
        xk1 = xk - delta_x
        delta_k = abs(delta_x)
        print("step",k,": error",delta_k)
        xk = xk1
```

```
step 0 : error 0.05742535402223874
step 1 : error 0.0036541711824864 22
step 2 : error 1.5735628964185795e-05
step 3 : error 2.907926876517772e-10
iterations end
```

In [7]:

```
print("solution x:", xk,'\n f(x):',f(xk))
```

```
solution x: 4.333755446919996
 f(x): 1.2931877790833823e-12
```

弦割法

用差商代替求导

$$f'(x_k) \approx \frac{f(x_k) - f(x_0)}{x_k - x_0}$$

In [8]:

```
x0 = 4.28
K = 10
xk = x0+h/10
delta_k = np.inf
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        # 弦割法迭代格式
        f1 = (f(xk)-f(x0))/(xk-x0)
        delta_x = f(xk)/f1
        xk1 = xk - delta_x
        delta_k = abs(delta_x)
        print("step",k,": error",delta_k)
        xk = xk1
```

```
step 0 : error 0.05100096908890893
step 1 : error 0.003457845367855087
step 2 : error 0.0002262490228629362
step 3 : error 1.4839035973882955e-05
step 4 : error 9.730983039276228e-07
step 5 : error 6.381344680216001e-08
step 6 : error 4.184730275455165e-09
iterations end
```

In [9]:

```
print("solution x:", xk,'\n f(x):',f(xk))
```

```
solution x: 4.333755447177531
 f(x): 3.4330486897715673e-07
```

Q2

7.3 用牛顿法、弦割法、布洛依登法求以下方程组的解：

(1) $\begin{cases} x_1^2 + x_2^2 + x_3^2 - 1.0 = 0, \\ 2x_1^2 + x_2^2 - 4x_3 = 0, \\ 3x_1^2 - 4x_2^2 + x_3^2 = 0, \end{cases}$ 给定初始向量 $x^{(0)} = (1.0, 1.0, 1.0)^{\mathrm{T}}$;

(2) $\begin{cases} \cos(x_1^2 + 0.4x_2) + x_1^2 + x_2^2 - 1.6 = 0, \\ 1.5x_1^2 - \dfrac{1}{0.36}x_2^2 - 1.0 = 0, \end{cases}$ 给定初始向量 $x^{(0)} = (1.04, 0.47)^{\mathrm{T}}$.

Q2.1

$$(1)\begin{cases} x_1^2+x_2^2+x_3^2-1.0=0,\\ 2x_1^2+x_2^2-4x_3=0,\\ 3x_1^2-4x_2^2+x_3^2=0, \end{cases} \qquad \text{给定初始向量 } x^{(0)}=(1.0,1.0,1.0)^{\mathrm{T}};$$

Newton Method

$$J_f(x^{(k)})\Delta x^{(k)} = -f(x^{(k)})$$
$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$$

In [10]:

```
def f(x_k):
    f1,f2,f3 = x_k[0]**2 + x_k[1]**2 + x_k[2]**2 -1.0,\
                  2*x_k[0]**2 + x_k[1]**2 - 4*x_k[2],\
                  3*x_k[0]**2 - 4*x_k[1]**2 + x_k[2]**2
    return np.array([f1,f2,f3])

x0 = np.array([1.0,1.0,1.0]).T
# print(np.shape(x0))
K = 10
x_k = x0
delta_k = np.inf
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        f_k = f(x_k)
        # Jacobi矩阵 计算
        J_k = np.array([[2*x_k[0],2 *x_k[1], 2*x_k[2]],\
              [4*x_k[0], 2*x_k[1], -4],\
              [6*x_k[0], -8*x_k[1], 2*x_k[2]]])
        #print(f_k, '\n', J_k)
        delta_x =  -np.linalg.inv(J_k) @ f_k
        delta_k = np.linalg.norm(delta_x,2) / np.linalg.norm(x_k,2)
        print("step",k,": error",delta_k)
        x_k = x_k + delta_x
```

```
step 0 : error 0.37124185219410927
step 1 : error 0.14919233549665556
step 2 : error 0.015591670926282905
step 3 : error 0.0001380424179155084
step 4 : error 1.1479643249364914e-08
step 5 : error 1.3296062323438945e-16
iterations end
```

In [11]:

```
print("solution x:", x_k,'\n f(x):',f_k)
```

```
solution x: [0.69828861 0.6285243  0.34256419]
 f(x): [2.22044605e-16 0.00000000e+00 4.16333634e-16]
```

弦割法

用差商代替偏导数，即用差商矩阵代替Jacobi矩阵。

$$\frac{\partial f_i(x^{(k)})}{\partial x_j} \approx \frac{f_i(x^{(k)} + e_j h) - f_i(x^{(k)})}{h}$$

In [12]:

```
def f(x_k):
    f1,f2,f3 = x_k[0]**2 + x_k[1]**2 + x_k[2]**2 -1.0,\
                2*x_k[0]**2 + x_k[1]**2 - 4*x_k[2],\
                3*x_k[0]**2 - 4*x_k[1]**2 + x_k[2]**2
    return np.array([f1,f2,f3], dtype='float')

x0 = np.ones((3,1))
# print(x0,f(x0))
K = 20
x_k = x0
delta_k = np.inf
h = 0.00001
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        f_k = f(x_k)
        # 差商计算，用差商代替求导
        J_k = np.zeros((3,3))
        for i in range(3):
            for j in range(3):
                diff = np.zeros((3,1))
                diff[j] = h
                J_k[i,j] = (f(x_k+diff)-f(x_k))[i] / h
        #print(f_k, '\n', J_k)
        delta_x =  -np.linalg.inv(J_k) @ f_k
        x_k = x_k + delta_x
        delta_k = np.linalg.norm(delta_x,2) / np.linalg.norm(x_k,2)
        print("step",k,": error",delta_k)
```

```
step 0 : error 0.5408450837651694
step 1 : error 0.174648314198659 42
step 2 : error 0.015834637482813083
step 3 : error 0.00013821235660823093
step 4 : error 1.250115701184215e-08
step 5 : error 9.04044620626625e-14
iterations end
```

In [13]:

```
print("solution x:", x_k.reshape((1,3)),'\n f(x):',f_k.reshape((1,3)))
```

```
solution x: [[0.69828861 0.6285243  0.34256419]]
 f(x): [[1.53654867e-13 2.74669176e-13 2.31523134e-13]]
```

Broyden's method

(1) 给定 $x^{(0)} \in D, \delta > 0, \varepsilon > 0$ 以及最大迭代次数 K.

(2) 计算 $A_0 = J_f(x^{(0)}), x^{(1)} = x^{(0)} - A_0^{-1} f(x^{(0)})$. 取 $k = 1$.

(3) 计算

$$
\begin{aligned}
& s^{(k)} = x^{(k)} - x^{(k-1)}, \qquad y^{(k)} = f(x^{(k)}) - f(x^{(k-1)}), \\
& A_k^{-1} = A_{k-1}^{-1} + \frac{\left(s^{(k)} - A_{k-1}^{-1} y^{(k)}\right) s^{(k)\mathrm{T}} A_{k-1}^{-1}}{s^{(k)\mathrm{T}} A_{k-1}^{-1} y^{(k)}}, \\
& x^{(k+1)} = x^{(k)} - A_k^{-1} f(x^{(k)}).
\end{aligned}
$$

(4) 若 $\|x^{(k+1)} - x^{(k)}\| < \delta$ 或 $\|f(x^{(k+1)})\| < \varepsilon$, 停止计算, 取 $x^* \approx x^{(k+1)}$. 否则

若 $k = K$ 停止运算, 输出 K 次迭代不满足精度要求的信息;
否则令 $k = k + 1$ 转 (3).

In [14]:

```
def f(x_k):
    f1,f2,f3 = x_k[0]**2 + x_k[1]**2 + x_k[2]**2 -1.0,\
               2*x_k[0]**2 + x_k[1]**2 - 4*x_k[2],\
               3*x_k[0]**2 - 4*x_k[1]**2 + x_k[2]**2
    return np.array([f1,f2,f3], dtype='float')

x0 = np.ones((3,1))
# print(x0,f(x0))
K = 20
x_k = x0
delta_k = np.inf
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break

    if k == 1:
        # A0 = J(x0)
        A0 = np.array([[2*x_k[0],2 *x_k[1], 2*x_k[2]],\
                  [4*x_k[0], 2*x_k[1], -4],\
                  [6*x_k[0], -8*x_k[1], 2*x_k[2]]], dtype='float')
        delta_x =  -np.linalg.inv(A0) @ f(x_k)
        delta_k = np.linalg.norm(delta_x,2)
        print("step",k,": error",delta_k)
        x_k1 = x_k
        x_k = x_k + delta_x
        A_k = A0
        #print(x_k1,x_k,A_k)

    if k > 1:
        # A矩阵 计算
        s_k = x_k - x_k1
        y_k = f(x_k) - f(x_k1)
        A_inv = np.linalg.inv(A_k) + (s_k - np.linalg.inv(A_k)@y_k)@s_k.T@np.linalg.

        delta_x =  -A_inv @ f(x_k)
        delta_k = np.linalg.norm(delta_x,2)
        print("step",k,": error",delta_k)
        x_k1 = x_k
        x_k = x_k + delta_x
        A_k = np.linalg.inv(A_inv)
```

```
step 1 : error 0.6430097498961727
step 2 : error 0.14393421246462418
step 3 : error 0.04825258604381852
step 4 : error 0.004344430750854101
step 5 : error 0.0008556656446347084
step 6 : error 0.0002584992591148905
step 7 : error 4.3049546576100495e-05
step 8 : error 1.0511292757385567e-05
step 9 : error 3.019601766808327e-07
step 10 : error 1.5027279942336136e-10
iterations end
```

In [15]:

```
print("solution x:", x_k.reshape((1,3)),'\n f(x):',f_k.reshape((1,3)))
```

```
solution x: [[0.69828861 0.6285243  0.34256419]]
 f(x): [[1.53654867e-13 2.74669176e-13 2.31523134e-13]]
```

Q2.2

$$(2)\begin{cases}\cos(x_1^2+0.4x_2)+x_1^2+x_2^2-1.6=0,\\ 1.5x_1^2-\dfrac{1}{0.36}x_2^2-1.0=0,\end{cases}\quad \text{给定初始向量 } x^{(0)}=(1.04,0.47)^{\mathrm{T}}.$$

Newton Method

In [16]:

```
def f(x):
    f1,f2 = cos(x[0]**2 + 0.4*x[1]) + x[0]**2 + x[1]**2 - 1.6,\
                  1.5*x[0]**2 - 1/0.36*x[1]**2 - 1.0
    return np.array([f1,f2],dtype='float')

x0 = np.zeros((2,1))
x0[0], x0[1] = 1.04,0.47
K = 20
x_k = x0
delta_k = np.inf
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        f_k = f(x_k)
        # Jacobi矩阵 计算
        J_k = np.array([ [(-sin(x_k[0]**2 + 0.4*x_k[1])*2*x_k[0] + 2*x_k[0])[0], (-0
                        [3*x_k[0], -2/0.36*x_k[1]]], dtype='float')
        #print(f_k, '\n', J_k)
        delta_x =  -np.linalg.inv(J_k) @ f_k
        #print(delta_x)
        delta_k = np.linalg.norm(delta_x,2) / np.linalg.norm(x_k,2)
        print("step",k,": error",delta_k)
        x_k = x_k + delta_x
```

```
step 0 : error 0.0019331087357013642
step 1 : error 3.7570860322303766e-06
step 2 : error 1.9136417446461214e-11
iterations end
```

In [17]:

```
print("solution x:", x_k.reshape((2)),'\n f(x):',f_k.reshape((2)))
```

```
solution x: [1.03862924 0.47172595]
 f(x): [ 1.24120714e-11 -4.02901046e-11]
```

弦割法

In [18]:

```
def f(x):
    f1,f2 = cos(x[0]**2 + 0.4*x[1]) + x[0]**2 + x[1]**2 - 1.6,\
                  1.5*x[0]**2 - 1/0.36*x[1]**2 - 1.0
    return np.array([f1,f2],dtype='float')

x0 = np.zeros((2,1))
x0[0], x0[1] = 1.04,0.47
K = 20
x_k = x0
# print(x0,'\n',f(x0))
delta_k = np.inf
h = 0.00001
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break
    else:
        f_k = f(x_k)
        # 差商计算，用差商代替偏导
        J_k = np.zeros((2,2))
        for i in range(2):
            for j in range(2):
                diff = np.zeros((2,1))
                diff[j] = h
                J_k[i][j] = (f(x_k+diff)-f(x_k))[i] / h
        #print(f_k, '\n', J_k)
        delta_x =  -np.linalg.inv(J_k) @ f_k
        x_k = x_k + delta_x
        delta_k = np.linalg.norm(delta_x,2) / np.linalg.norm(x_k,2)
        print("step",k,": error",delta_k)
```

```
step 0 : error 0.001933990591758431
step 1 : error 3.7179114669046912e-06
step 2 : error 6.366846130191359e-11
iterations end
```

In [19]:

```
print("solution x:", x_k.reshape((2)),'\n f(x):',f_k.reshape((2)))
```

```
solution x: [1.03862924 0.47172595]
 f(x): [ 4.11903844e-11 -1.25297106e-10]
```

Broyden's Method

In [20]:

```
def f(x):
    f1,f2 = cos(x[0]**2 + 0.4*x[1]) + x[0]**2 + x[1]**2 - 1.6,\
                  1.5*x[0]**2 - 1/0.36*x[1]**2 - 1.0
    return np.array([f1,f2],dtype='float')

x0 = np.zeros((2,1))
x0[0], x0[1] = 1.04,0.47
# print(x0,f(x0),np.shape(f(x0)))
K = 20
x_k = x0
delta_k = np.inf
for k in range(K):
    if delta_k < Delta:
        print("iterations end")
        break

    if k == 1:
        # A0 = J(x0)
        A0 = np.array([ [(-sin(pow(x_k[0], 2) + 0.4*x_k[1])*2*x_k[0] + 2*x_k[0])[0],
                       [3*x_k[0], -2/0.36*x_k[1]]], dtype='float')
        #print(A0)
        delta_x =  -np.linalg.inv(A0) @ f(x_k)
        delta_k = np.linalg.norm(delta_x,2)
        print("step",k,": error",delta_k)
        x_k1 = x_k
        x_k = x_k + delta_x
        A_k = A0
        #print(x_k1,x_k,A_k)

    if k > 1:
        # A矩阵 计算
        s_k = x_k - x_k1
        y_k = f(x_k) - f(x_k1)
        A_inv = np.linalg.inv(A_k) + (s_k - np.linalg.inv(A_k)@y_k)@s_k.T@np.linalg.

        delta_x =  -A_inv @ f(x_k)
        delta_k = np.linalg.norm(delta_x,2)
        print("step",k,": error",delta_k)
        x_k1 = x_k
        x_k = x_k + delta_x
        A_k = np.linalg.inv(A_inv)
```

```
step 1 : error 0.0022062013672188943
step 2 : error 4.311478595807153e-06
step 3 : error 2.566361026530101e-08
step 4 : error 2.251475960223992e-12
iterations end
```

In [21]:

```
print("solution x:", x_k.reshape((2)),'\n f(x):',f_k.reshape((2)))
```

```
solution x: [1.03862924 0.47172595]
 f(x): [ 4.11903844e-11 -1.25297106e-10]
```

Note

对于小规模非线性方程组，三种迭代解法(牛顿法、弦割法、布洛伊登法)收敛速度相差不大。对于上述两个算例，均在10次迭代之内收敛。

三种解法解得的方程组的解也几乎一样，满足误差不超过 10^{-8} 的要求。

In [ ]: