第三章作业

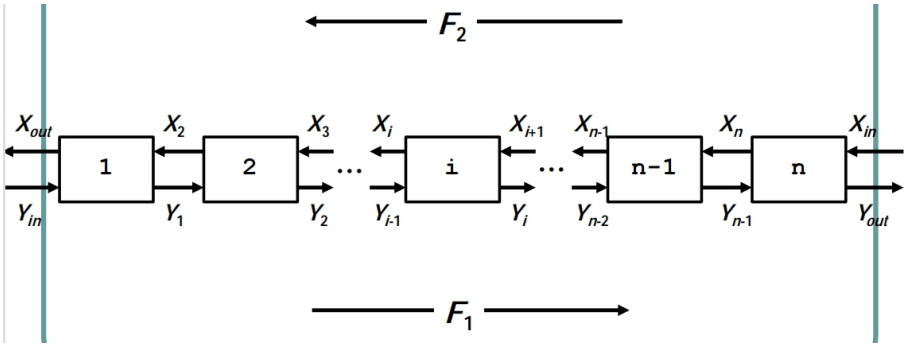
目录

一、问题叙述

下图为一个多级萃取过程，待萃取物的流率为 F_1，待萃取物的组分为 Y_{in}，萃取剂的流率为 F_2，萃取剂的组分为 X_{in}。第 i 级的质量守恒方程为:$F_1Y_{i-1} + F_2X_i = F_1Y_i + F2Xi$，且每一级均为平衡级，即 $K = X_i / Y_i$，K=4 为平衡常数。

（1）设 F_1=500kg/h、 F_2=300kg/h、Y_{in}=0.5、 X_{in}=0，试对不同级数（n=3、5、10、20、25、50、100）的萃取过程，分别计算 X_{out} 和 Y_{out} ，请用不同方法求解并进行对比（对第一级和最后一级需要修改方程）。

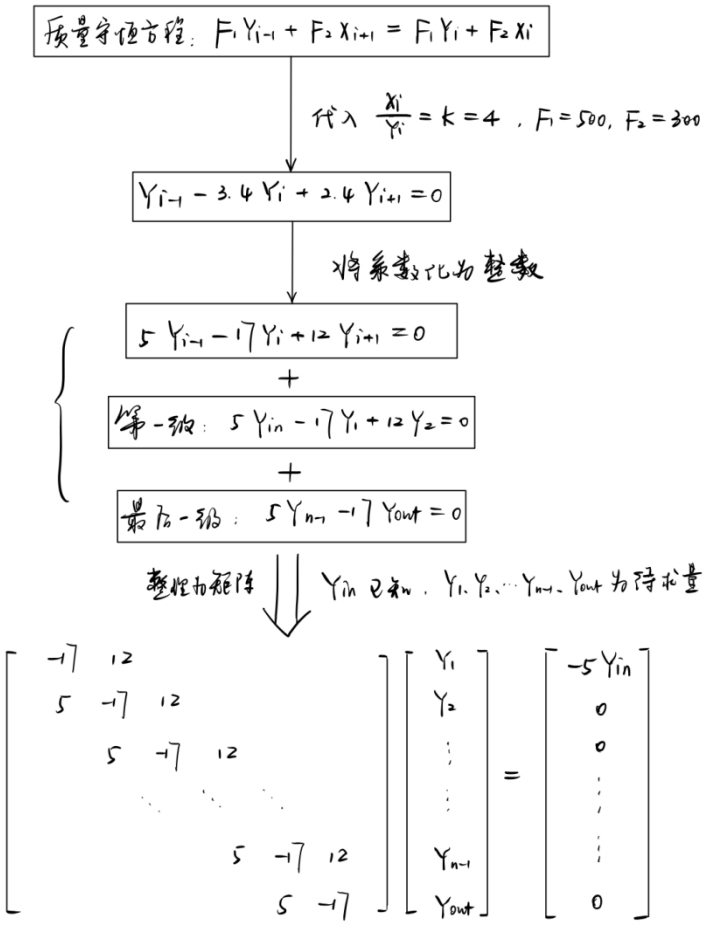
（2）设 F_1=500kg/h、 F_2=300kg/h、 X_{in}=0、级数 n=20，若 Y_{in} 分别为 0.3、0.5、0.7、0.9，计算 X_{out} 和 Y_{out} ，请用不同方法求解并进行对比？



二、问题分析

将已知条件带入后化简，并将系数化为整数，可以得到一个 n 阶的线性方程组，其中系数矩阵 A 的元素是已知的。求解得出的是各级萃取过程中 Yi 的值，题目所要求的 Yout 即为 Yn，而 Xout = 4*Y1。然后通过不断改变变量 n 和 Yin 的值，对不同解法的精度和收敛速度进行比较。

本例中统一采用 Ax=B 的形式，A 为系数矩阵，B 为列矩阵，解得的矩阵 x 即 Yi 的值。同时先给出各个方法的实现代码，最后再统一进行比较。



三、构建矩阵

将矩阵维数 n 和 Yin 作为参数传入该函数，返回值为系数矩阵 A 和列向量 B。

```python
import numpy as np

def create_matrix(n, Yin):
    A = np.zeros(n)                 #initialize
    A[0] = -17
    A[1] = 12

    #add by line
    for i in range(1, n):
        temp = np.zeros(n)
        if i == n-1:
            temp[n-2] = 5
            temp[n-1] = -17
        else:
            temp[i-1] = 5
            temp[i] = -17
            temp[i+1] = 12
        A = np.vstack((A, temp))    #vertical splicing
    B = np.zeros(n)
    B[0] = -5 * Yin

    return A, B
```

四、判断是否为病态矩阵

将矩阵的最大系数缩放为 1，求出 A 和 A 逆的范数，然后计算普条件数，与 1 进行比较，发现无论怎样更改 n 都不会出现病态矩阵的情况。

```
from create_matrix import *

n = 10
Yin = 0.5
A,B = create_matrix(n, Yin)
A = A/(-17)                                          #set the maximum element as '1'
norm_A = max(np.linalg.eigvals(A @ A.T)) ** 0.5      #norm of A
A_inv = np.linalg.inv(A)
norm_A_inv = max(np.linalg.eigvals(A @ A.T)) ** 0.5  #norm of A^-1
con_num = norm_A * norm_A_inv                        #spectrum condition number
print(con_num)
```

```
n = 3, Yin = 0.5, condition number = 2.968197
n = 5, Yin = 0.5, condition number = 3.514536
n = 7, Yin = 0.5, condition number = 3.721847
n = 10, Yin = 0.5, condition number = 3.851391
```

五、高斯消元

本例中采用了原始的高斯顺序消去法，未使用主元的思想（在后续 LU 分解中使用）。通过计算因数对矩阵 A 进行前向消去，然后回代，把结果保留在矩阵 B 中：

```
#elimination
for i in range(1,n):
    for j in range(i,n):
        factor = A[j][i-1] / A[i-1][i-1]        #elimination factor
        for k in range(i-1,n):
            A[j][k] = A[j][k] - A[i-1][k]*factor
        B[j] = B[j] - B[i-1]*factor

#back substitution
B[n-1] = B[n-1] / A[n-1][n-1]
for i in range(n-2, -1, -1):
    for j in range(n-1, i,-1):
        B[i] = B[i] - A[i][j]*B[j]
    B[i] = B[i] / A[i][i]
```

六、Doolittle 分解

Doolittle 分解是三角分解的一种，将 A 分解为单位下三角矩阵 L 和上三角矩阵 U，然后进行前向消去和后向消去。方法与高斯消元差别不大，但是无需提前知道右端列向量，也不必计算中间结果。

```
#Doolittle decomposition
def Doolittle(A):
    n = A.shape[0]
    L = np.zeros((n,n))
    U = np.zeros((n,n))

    U[0,:] = A[0,:]
    for i in range(n):
        L[i,i] = 1
        L[i,0] = A[i,0] / U[0,0]
    for i in range(1, n):
        for j in range(i, n):
            U[i,j] = A[i,j] - np.dot(L[i,:i], U[:i,j])
            if(j+1 < n):
                L[j+1,i] = (A[j+1,i] - np.dot(L[j+1,:i], U[:i,i])) / U[i,i]
    return L,U

L,U = Doolittle(A)

#forward elimination
y = np.zeros(n)
for i in range(n):
    y[i] = (B[i] - np.dot(L[i,:], y.T))
#backward elimination
x = np.zeros(n)
for i in range(n-1, -1, -1):
    x[i] = (y[i] - np.dot(U[i,:], x.T)) / U[i,i]
```

七、Doolittle 分解列主元

顺序消去条件苛刻且数值不稳定，全主元则工作量偏大，算法复杂，因此列主元消去比较好。

```
#column pivot
for i in range(1, n):
    max_index = i               #store the index of the largest value in column
    max_value = float('-inf')   #initialize the max value

    for k in range(i, n):
        temp_value = matrix[k,i] - np.dot(L[k,:i], U[:i,i])
        if(max_value < temp_value):
            max_index = k
            max_value = temp_value
    matrix[[i,max_index],:] = matrix[[max_index,i],:]   #exchange
    L[[i,max_index],:i] = L[[max_index,i],:i]           #exchange

    for j in range(i, n):
        U[i,j] = matrix[i,j] - np.dot(L[i,:i], U[:i,j])
        if(j+1 < n):
            L[j+1,i] = (matrix[j+1,i] - np.dot(L[j+1,:i], U[:i,i])) / U[i,i]
```

八、Thomas 算法

追赶法适用于带状矩阵，若用高斯消去或 LU 分解效率较低，显然本例中的矩阵满足对角占优条件。

```
#get the elements on the diagonal
a = np.diagonal(A, -1)
b = np.diagonal(A)
c = np.diagonal(A, 1)

#use formula of Thomas algorithm to get 'beta' in matrix 'U'
beta = np.zeros(n-1)
beta[0] = c[0]/b[0]
for i in range(1, n-1):
    beta[i] = c[i]/(b[i]-a[i]*beta[i-1])

#forward elimination
y = np.zeros(n)
y[0] = B[0]/b[0]
for i in range(1, n):
    y[i] = (B[i] - a[i-1]*y[i-1]) / (b[i] - a[i-1]*beta[i-1])

#backward elimination
x = np.zeros(n)
x[n-1] = y[n-1]
for i in range(n-2, -1, -1):
    x[i] = y[i] - beta[i]*x[i+1]
```

九、雅各比迭代法

迭代法需要构造向量序列 x^k，使得 $x^{k+1} = Jx^k + f$，类似于不动点的过程。一般可将 A 分裂为 $A = M - N$，则 $Mx = Nx + b$，$x = M^{-1}Nx + M^{-1}b$。

对雅各比迭代来说，选取 $M = D$，$N = D - A = L + U$，则 $J = D^{-1}(L+U)$，$f = D^{-1}b$。每迭代一次计算一次矩阵乘向量，即 Jx^k，计算过程中 A 不变，需要两组单元保存 x^{k+1} 和 x^k。

x^0 和 x^1 分别初始化为全 0 和全 1 的矩阵，满足收敛条件。

迭代终止条件通过设定近似百分比相对误差来完成。

```python
def jacobi(n, A, B, x0, x, tolerence, limit):
    """Jacobi
        Arguments:
            n: dimension
            x: current solution
            x0: previous solution
            tolerence: accuracy
            limit: maximum times of looping
    """
    times = 0

    while times < limit:
        for i in range(n):
            temp = 0
            for j in range(n):
                if i != j:
                    temp += x0[j] * A[i][j]
            x[i] = ((B[i] - temp) / A[i][i])
        error = max(abs(x - x0))

        if error < tolerence:
            return (x, times)
        else:
            x0 = x.copy()

        times += 1
```

十、高斯-赛德尔迭代法

选取 $M = D - L$，$N = M - A = U$，则 $G = (D - L)^{-1} U$，$f = (D - L)^{-1} b$。每迭代一次计算一次矩阵乘向量，只需要一组单元保存 x^{k+1} 和 x^k。

```python
def gauss_seidel(n, A, B, x, tolerence, limit):
    """Gauss-Seidel
        Arguments:
            n: dimension
            x: current solution
            tolerence: accuracy
            limit: maximum times of looping
    """
    times = 0

    while times < limit:
        x0 = x.copy()
        for i in range(n):
            x[i] = (B[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i+1:], x[i+1:])) / A[i][i]
```

```
        error = max(abs(x - x0))
        if error < tolerence:
            return (x, times)

        times += 1

    return np.ones(n), times
```

十一、逐次超松弛迭代法

在高斯-赛德尔方法上进行修改，加入松弛因子 w。0<w<1 时，当前迭代结果为上一次结果的加权平均；w>1 时，为超松弛方法。

```
def sor(n, A, B, x, tolerence, limit, w):
    """SOR
        Arguments:
            n: dimension
            x: current solution
            tolerence: accuracy
            limit: maximum times of looping
            w: relaxation factor
    """
    times = 0

    while times < limit:
        x0 = x.copy()
        for i in range(n):
            x[i] = (1-w) * x0[i] + w * (B[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i+1:], x[i+1:])) / A[i][i]

        error = max(abs(x - x0))
        if error < tolerence:
            return (x, times)

        times += 1
```

并且通过循环找到最适合的 w 值：

```
for i in range(1, 20):
    A,B = create_matrix(n,Yin)
    x = np.zeros(n)
    x, times = sor(n, A, B, x, tolerence, limit, i/10)
    Xout = 4 * x[0]
    Yout = x[n-1]
    print('Xout = %.10f, Yout = %.10f, w = %g, times = %d' % (Xout, Yout, i/10, times))
```

十二、判断收敛性

雅各比迭代需满足： ρ (J)<1，高斯迭代需满足： ρ (G)<1

```
n = 100
Yin = 0.5
A, B = create_matrix(n, Yin)

L = -np.tril(A, k=-1)
U = -np.triu(A, k=1)
D = np.triu(np.tril(A))

#Jacobi
lambda_J = max(abs(np.linalg.eigvals(np.linalg.inv(D) @ (L+U))))
print(lambda_J)

#Gauss-Seidel
lambda_G = max(abs(np.linalg.eigvals(np.linalg.inv(D-L) @ U)))
print(lambda_G)
```

n = 3,　lambda(J) = 0.644379,　lambda(G) = 0.415225
n = 5,　lambda(J) = 0.789200,　lambda(G) = 0.622837
n = 10,　lambda(J) = 0.874377,　lambda(G) = 0.764534
n = 20,　lambda(J) = 0.901112,　lambda(G) = 0.812003
n = 25,　lambda(J) = 0.904646,　lambda(G) = 0.818384
n = 50,　lambda(J) = 0.909562,　lambda(G) = 0.827303
n = 100,　lambda(J) = 0.910850,　lambda(G) = 0.829652

本例中无论 n 取何值，均能满足收敛的条件。

十三、Yin = 0.5，改变 n

分别令 n = 3、5、10、20、25、50、100。

Xout

	高斯消元	LU 分解	列主元	Thomas	Jacobi/次数	G-S/n	SOR/n
N = 3	0.797076	0.797076	0.797076	0.797076	0.796632/15	0.796892/8	0.797024/5
N = 5	0.827196	0.827196	0.827196	0.827196	0.826645/24	0.826645/12	0.827261/6
N = 10	0.833257	0.833257	0.833257	0.833257	0.832243/30	0.832243/15	0.833187/5
N = 20	0.833333	0.833333	0.833333	0.833333	0.832252/30	0.832252/15	0.833187/5
N = 25	0.833333	0.833333	0.833333	0.833333	0.832252/30	0.832252/15	0.833187/5
N = 50	0.833333	0.833333	0.833333	0.833333	0.832252/30	0.832252/15	0.833187/5
N = 100	0.833333	0.833333	0.833333	0.833333	0.832252/30	0.832252/15	0.833187/5

Yout

	高斯消元	LU 分解	列主元	Thomas	Jacobi	G-S	SOR/w
N = 3	0.021754	0.021754	0.021754	0.021754	0.021708	0.021746	0.021753/1.2
N = 5	0.003682	0.003682	0.003682	0.003682	0.003658	0.003673	0.003682/1.3
N = 10	0.000046	0.000046	0.000046	0.000046	0.000041	0.000045	0.000046/1.4
N = 20	7.26e-09	7.26e-09	7.26e-09	7.26e-09	9.46e-10	5.22e-09	7.25e-09/1.4
N = 25	9.11e-11	9.11e-11	9.11e-11	9.11e-11	1.26e-12	4.85e-11	9.10e-11/1.4
N = 50	2.85e-20	2.85e-20	2.85e-20	2.85e-20	0	8.54e-22	2.82e-20/1.4
N = 100	2.78e-39	2.78e-39	2.78e-39	2.78e-39	0	4.49e-45	2.66e-39/1.4

十四、n = 20，改变 Yin

Xout

	高斯消元	LU 分解	列主元	Thomas	Jacobi/次数	G-S/n	SOR/n
Yin = 0.3	0.5	0.5	0.5	0.5	0.498894/26	0.498894/13	0.499802/4
Yin = 0.5	0.833333	0.833333	0.833333	0.833333	0.832252/30	0.832252/15	0.833187/5
Yin = 0.7	1.166667	1.166667	1.166667	1.166667	1.165499/32	1.165499/16	1.166462/5
Yin = 0.9	1.5	1.5	1.5	1.5	1.498839/34	1.498839/17	1.499736/5

Yout

	高斯消元	LU 分解	列主元	Thomas	Jacobi	G-S	SOR/w
Yin = 0.3	4.35e-09	4.35e-09	4.35e-09	4.35e-09	2.07e-10	2.71e-09	4.35e-09/1.4
Yin = 0.5	7.26e-09	7.26e-09	7.26e-09	7.26e-09	9.46e-10	5.22e-09	7.25e-09/1.4
Yin = 0.7	1.02e-08	1.02e-08	1.02e-08	1.02e-08	1.89e-09	7.71e-09	1.02e-08/1.4
Yin = 0.9	1.31e-08	1.31e-08	1.31e-08	1.31e-08	3.23e-09	1.04e-08	1.31e-08/1.4

十五、结果分析

可以看到，不论 n 和 Yin 的取值如何变化，直接法得出的结果都几乎一致。迭代法中超松弛迭代的收敛效果最好，迭代次数最少，松弛因数 w 一般都选取为 1.4，高斯迭代的收敛性稍稍差一些，雅各比再次之。同时，在计算 Xout 时各

方法的结果都相差不大，可是 Yout 的结果受到截断误差和舍入误差的影响较大，雅各比迭代求得的 Yout 与直接法相差明显，高斯迭代稍稍好一些，而超松弛迭代在选取适合的 w 后能做到与直接法的结果几乎完全一致，且迭代次数仅需要 5 次左右，计算效率也很高。