数值分析第二次大作业

参考书：《数值分析（第4版）》 颜庆津 北京航空航天大学出版社
题目：P239 第四题

一、问题重述

求解线性方程组$Ay=b$，其中

$$\mathbf{A}=\begin{bmatrix} a_1 & 10 & & & & & \\ 1 & a_2 & 10 & & & & \\ 10 & 1 & a_3 & 10 & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots & 10 \\ & & & & 10 & 1 & a_{1000} \end{bmatrix},\quad \mathbf{b}=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_{1000} \end{bmatrix} \tag{5}$$

而$a_{5(k-1)+i}(i=1,2,3,4,5)$是非线性方程组

$$\begin{cases} e^{-x_1}+e^{-2x_2}+x_3-2x_4+t_kx_5-5.3 &= 0 \\ e^{-2x_1}+e^{-x_2}-2x_3+t_kx_4-x_5+25.6 &= 0 \\ t_kx_1+3x_2+e^{-x_3}-3x_5+37.8 &= 0 \\ 2x_1+t_kx_2+x_3-e^{-x_4}+2e^{-2x_5}-31.3 &= 0 \\ x_1-2x_2-3t_kx_3+e^{-2x_4}+3e^{-x_5}+42.1 &= 0 \end{cases} \tag{6}$$

在区域$D=\{x_i\geq 2, i=1,2,3,4,5\}\subset\mathbb{R}^5$内的解，其中
$t_k=1+0.001(k-1), k=1,2,\cdots,200$

b_k是方程$e^{-t_kb_k}=t_k\ln b_k$的解，$k=1,2,\cdots,1000$，其中
$t_k=1+0.001(k-1), k=1,2,\cdots,1000$

二、向量b求解

2.1 参数初始化与非线性函数构建

首先导入相关库，初始化已知量t_k，并构建向量b对应的函数f_b。

```
import math
import numpy as np
from scipy.sparse import csr_matrix
```

```
t_list =  [1 + 0.001 * (k - 1) for k in range(1, 1001)]
```

```
def f_b(b, t):
    return math.exp(-t * b) - t * math.log(b)
```

2.2 基于割线法求解非线性方程

先要求解非线性方程$f(b)=0$，典型Newton的一个明显缺点是对每一轮迭代都需要计算$f'(b_k)$，因此此处使用割线法求解。求解公式为

$$x_{k+1}=x_k-\frac{f(x_k)(x_k-x_{k-1})}{f(x_k)-f(x_{k-1})}\quad(k=0,1,\cdots)\tag{7}$$

编写割线法求解代码如下，终止条件为$\frac{|x_k-x_{k-1}|}{|x_k|}\leq 10^{-12}$

```
def solve_b(k):
    iter_num = 0
    t = t_list[k - 1]
    x_ = 0
    x_k = 0.9
    x_k1 = 2
    while abs(x_k - x_) / abs(x_k) > 1e-12:
        iter_num += 1
        x_ = x_k
        x_k = x_k1
        x_k1 = x_k - (f_b(x_k, t) * (x_k - x_)) / (f_b(x_k, t) - f_b(x_, t))

#     print('Return after %d iterations' % iter_num)
    return x_k1
```

2.3 向量b求解结果

对k的每一个取值，求解b_k，并以e型输出向量b的前10个结果，b向量全部元素见附件。

```
b_list = [solve_b(k) for k in range(1, 1001)]

print('First 10 elements:')
for k in range(10):
    print('%6e' % b_list[k])
```

```
First 10 elements:
1.309800e+00
1.309197e+00
1.308596e+00
1.307997e+00
1.307399e+00
1.306803e+00
1.306208e+00
1.305614e+00
1.305023e+00
1.304432e+00
```

三、向量a求解

3.1 非线性方程组读入

首先输入非线性方程组(A.1)。

```
def F_A(x, k):
    # x为列向量，因此需二次索引
    x1 = x[0][0]
    x2 = x[1][0]
    x3 = x[2][0]
    x4 = x[3][0]
    x5 = x[4][0]
    t = t_list[k - 1]
    f1 = math.exp(-x1) + math.exp(-2 * x2) + x3 - 2 * x4 + t * x5 - 5.3
    f2 = math.exp(-2 * x1) + math.exp(-x2) - 2 * x3 + t * x4 - x5 + 25.6
    f3 = t * x1 + 3 * x2 + math.exp(-x3) - 3 * x5 + 37.8
    f4 = 2 * x1 + t * x2 + x3 - math.exp(-x4) + 2 * math.exp(-2 * x5) - 31.3
    f5 = x1 - 2 * x2 - 3 * t * x3 + math.exp(-2 * x4) + 3 * math.exp(-x5) +
42.1

    return np.array([[f1], [f2], [f3], [f4], [f5]])
```

3.2 基于离散牛顿法求解非线性方程组

为避免求导运算，使用离散牛顿法。值得注意的是，由于本题有定义域的限制，解应处于定义域$D=\{x_i\geq 2, i=1,2,3,4,5\}\subset\mathbb{R}^5$内，因此需要在迭代求解时对x的迭代轨迹施加约束，保证x不超出定义域。具体地，需对参考书上P92的离散牛顿法做如下两点改动：

改动一：设计获取$\boldsymbol{h}^{(k)}$的子函数，确保$\boldsymbol{x}^{(k)}+\boldsymbol{h}^{(k)}$在定义域D内

使用牛顿-斯蒂芬森方法确定$\boldsymbol{h}$，若$\boldsymbol{x}^{(k)}+\boldsymbol{h}^{(k)}$不在定义域内，由于本例中$\boldsymbol{x}$与$\boldsymbol{h}$均为正数，因此增大$\boldsymbol{h}$，直到$\boldsymbol{x}^{(k)}+\boldsymbol{h}^{(k)}$处于定义域内。代码实现如下：

```
def get_h(x, F, lower_bound):
    c = 2
    h = c * np.linalg.norm(F) # 此处是牛顿-斯蒂芬森法，c1=c2=...=c5
    x_ori = x
    x = x_ori + h * np.array([np.ones(5)]).T
    while sum(x >= lower_bound * np.array([np.ones(5)]).T) < 5:
        # 保证x + h在定义域内，否则继续增大h
        h = h * c
        x = x_ori + h * np.array([np.ones(5)]).T

    return h * np.ones(5)
```

```
def J(x, h, k):
    J = np.zeros((5,5))
    e = np.eye(5)
    for i in range(5):
        J[:,[i]] = (F_A(x + h[i] * e[:,[i]], k) - F_A(x, k)) / h[i]

    return J
```

改动二：对离散牛顿法设计变步长策略

为确保$\boldsymbol{x}$处于定义域内，对于离散牛顿法迭代公式的每一步
$\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}-\boldsymbol{J}\left(\boldsymbol{x}^{(k)}, \boldsymbol{h}^{(k)}\right)^{-1} \boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) \quad(k=0,1, \cdots)$，都检验迭代后的x是否在定义域内，若$\boldsymbol{x}$超出定义域，则对步长$\boldsymbol{J}\left(\boldsymbol{x}^{(k)}, \boldsymbol{h}^{(k)}\right)^{-1} \boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) \quad(k=0,1, \cdots)$乘以一个小于1的因子$\alpha$，再次检验迭代后的点是否在定义域内，如有必要，继续调整α，直到迭代后的点在定义域内。

综合上述两项改动，针对本例的离散牛顿法流程如下：
对于$k=0,1, \cdots$，执行

1. 选取$\boldsymbol{h}^{(k)}=\left(h_1^{(k)}, h_2^{(k)}, \cdots, h_n^{(k)}\right)^{\mathrm{T}}, h_j^{(k)} \neq 0(j=1,2, \cdots, n)$
2. 计算$\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right)$ 和 $\boldsymbol{J}\left(\boldsymbol{x}^{(k)}, \boldsymbol{h}^{(k)}\right)$
3. 计算$\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}-\alpha \boldsymbol{J}\left(\boldsymbol{x}^{(k)}, \boldsymbol{h}^{(k)}\right)^{-1} \boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) \quad(k=0,1, \cdots)$
4. 若满足终止条件$\left\|\mathbf{F}\left(x^{(k+1)}\right)\right\|_{\infty} \leq 10^{-12}$，停止迭代，否则转1继续迭代。

求解非线性方程组的代码如下：

```
def solve_a(k):
    iter_num = 0
    lower_bound = 2
    epsilon = 1e-12
    x_array = np.array([[10],[10],[10],[10],[10]])
    while sum(F_A(x_array, k) > epsilon * np.array([np.ones(5)]).T) :
        # 只要F0中有一个元素超过epsilon,则继续迭代
        iter_num += 1
        h = get_h(x_array, F_A(x_array, k), lower_bound)
        s = np.dot(np.linalg.inv(J(x_array, h, k)), F_A(x_array, k))
        a = 1

        x_array_ori = x_array
        x_array = x_array_ori - a * s

        # 如果x1-x5中任意一个超出了定义域，则缩小迭代步长，重新由上一个点迭代一次
        while sum(x_array >= lower_bound * np.array([np.ones(5)]).T) < 5:
            a = a * 0.5
            x_array = x_array_ori - a * s

#     print('Return after %d iterations' % iter_num)
    x_array = np.transpose(x_array)

    # 返回一个长度为5的list
    return x_array.tolist()[0]
```

3.3 向量a求解结果

使用上述方法求解矩阵A中的对角线元素$a_1 \cdots a_{1000}$，输出向量$\boldsymbol{a}$的前10个元素（全部元素见附件），并检验每个a_k是否都在定义域内。可以看出，每个a都满足定义域约束。

```
a_list = []
for k in range(1, 201):
    a_list.extend(solve_a(k))

print('First 10 elements:')
for k in range(10):
    print('%6e' % a_list[k])

# 检验每个a是否都>=2
a_array = np.array(a_list)
print(sum(a_array > 2 * np.ones(1000)) == 1000)
```

```
First 10 elements:
5.160498e+00
1.567652e+01
5.302487e+00
1.500328e+01
2.999834e+01
5.165717e+00
1.560918e+01
5.343773e+00
1.500691e+01
2.993440e+01
True
```

四、方程$Ay = b$求解与分析

4.1 矩阵A存储

由于A是稀疏矩阵，构建一个二维数组存储A的所有元素将造成不必要的内存开支，因此此处不存储A的零元素。具体地，将A以压缩稀疏行矩阵(Compressed Sparse Row Matrix, CSR Matrix)的形式存储。在Python中，这一操作可以借助`scipy.sparse.csr_matrix`类实现。下面的代码展示了将A矩阵读入并转换为稀疏矩阵的过程。

```
def a(i, j):
    if i == j:
        return a_list[i]
    elif i == j + 1:
        return 1
    elif i == j - 1 or i == j + 2:
        return 10
    else:
        return 0

A = np.zeros((1000,1000))
for i in range(1000):
    for j in range(1000):
        A[i,j] = a(i, j)

A = csr_matrix(A)
```

4.2 基于Jacobi迭代法求解线性方程组

4.2.1 Jacobi迭代收敛条件判断

虽然Jacobi迭代的过程中无需显式地计算出矩阵D、L、U，但为判断Jacobi迭代矩阵G_J的谱半径是否能保证迭代收敛，而计算G_J矩阵需要矩阵D、L、U，因此，此处仍然给出A = D + L + U的分解代码：

```
def get_G(A):

    def d(i, j, A):
        if i == j:
            return A[i,j]
        else:
            return 0

    def l(i, j, A):
        if i > j:
            return A[i,j]
        else:
            return 0

    def u(i, j, A):
        if i < j:
            return A[i,j]
        else:
            return 0

    D = np.zeros((1000,1000))
    L = np.zeros((1000,1000))
    U = np.zeros((1000,1000))
    for i in range(1000):
        for j in range(1000):
            D[i,j] = d(i, j, A)
            L[i,j] = l(i, j, A)
            U[i,j] = u(i, j, A)

    G = np.dot(- np.linalg.inv(D),(L + U))

    return G
```

编写计算一个矩阵谱半径的子函数如下：

```
def spectral_radius(M):
    lam, alpha = np.linalg.eig(M) #a为特征值集合，b为特征值向量
    return max(abs(lam)) #返回谱半径
```

计算Jacobi迭代法中的G_J，并计算其谱半径

```
G = get_G(A)
print(spectral_radius(G))
```

```
2.31523937130752
```

4.2.2 基于高斯消去法的矩阵初等变换

由上述计算结果可见，G_J的谱半径大于1，因此直接使用Jacobi迭代法无法正确解出y的值，应对系数矩阵A做预处理，保证使用Jacobi迭代法构造出的G_J谱半径小于1。预处理的方式是构造Ay=b的同解方程组，即对(A|b)做初等行变换，一种可行的方法是将A变为上三角/下三角矩阵，此时易证Jacobi迭代法构造出的G_J特征值全为0，即可保证迭代收敛。初等行变换的一种可行方法是使用高斯消去法，具体代码见下：

```
def pre_condition(A, b):
    A = A.toarray()
    n = len(b)
    for k in range(n-1):
        for i in range(k+1,n):
            m = A[i,k] / A[k,k]
            A[i,k+1:] = A[i,k+1:] - m * A[k,k+1:]
            b[i] = b[i] - m * b[k]

    for j in range(n):
        for i in range (j+1, n):
            A[i, j] = 0

#     A = csr_matrix(A)
    return A, b
```

使用高斯消去法对矩阵进行预处理，再次检验Jacobi迭代法中G_J的谱半径。

```
A, b_list = pre_condition(A, b_list)
G = get_G(A)
print(spectral_radius(G))
```

```
0.0
```

将矩阵A化成上三角矩阵后，G_J的谱半径为0，与理论分析结果吻合，下面可以开始使用Jacobi迭代法求解$Ay = b$。

4.2.3 Jacobi迭代法

Jacobi迭代法代码如下，终止条件设置为$\left\|\mathbf{y}^{k}-\mathbf{y}^{k-1}\right\|_{\infty} \leq 10^{-10}$。

```
def solve_y(A, b_list, y0):
    iter_num = 0
    # y为array型行向量
    n = y0.size
    y_next = y0
    y = y_next - np.ones(n) # 该值无意义，仅为使while循环开始

    while max(abs(y_next - y)) > 1e-10:
        iter_num += 1
        y = y_next
        y_hat = np.zeros(n)
        for i in range(n):
            y_hat[i] = (- sum([A[i,j] * y[j] for j in range(n) if j != i]) +
b_list[i]) / A[i,i]

        y_next = y_hat
```

```
16
17      print('Return after %d iteration(s)' % iter_num)
18      return y_next
```

4.3 向量y求解结果

使用上述Jacobi迭代法，以0为初值，求解y并输出向量y的前10个元素（全部元素见附件）。

```
y0 = np.zeros(1000)
y = solve_y(A, b_list, y0)

print('First 10 elements:')
for k in range(10):
    print('%6e' % y[k])
```

```
Return after 55 iteration(s)
First 10 elements:
1.005432e-01
7.909468e-02
-3.127517e-03
2.406533e-02
1.591185e-02
8.372796e-02
6.177224e-02
9.914706e-03
3.535806e-02
1.467703e-02
```

五、总结与思考

1. 就本文求解这个特定的线性方程组问题而言，比较容易直接想到的方法是三角分解法（参考书P24），因为A矩阵是典型的带状线性方程组。本文"初等变换-Jacobi迭代"的方法虽然不局限于解带状方程组，但是计算较为繁琐，计算代价也不低。
2. 在将A矩阵变换为上三角矩阵时，可以借助高斯消去法，但是不可直接按照参考书P15的方法实施，因为P15的消元过程实际并未将A矩阵主对角线以下元素化为0，这是因为高斯消去法回带的过程未调用这些元素。若要借助高斯消去法将某个矩阵化为上三角矩阵，一种易犯的错误是，在高斯消元过程中，将下列公式中的j从k开始取值。这样虽然理论上可以使A变成上三角矩阵，但实际上，由于数值误差的存在，此时每行主对角线元素左边相邻元素的值常常是一个接近0的很小的值（是两个float型相减产生的），这种并不精确的置零可能导致某些问题。因此，理想的做法，要么是使用如下公式变换后，再将主对角线以下元素手动置零（本文做法），要么是不存储A主对角线以下元素。

$$
\begin{aligned}
m_{ik} &= a_{ik}^{(k)} / a_{kk}^{(k)} \\
a_{ij}^{(k+1)} &= a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)} \quad (j = k+1, k+2, \cdots, n \\
b_i^{(k+1)} &= b_i^{(k)} - m_{ik} b_k^{(k)}
\end{aligned} \tag{8}
$$

3. 使用CSR Matrix存储稀疏矩阵，实际上是牺牲了索引矩阵元素的速度，换来了内存开销的降低。
4. 在编程中值得注意的一个问题是，Python统一使用引用传递，对于可变(mutable)对象，包括list,dict等，子函数对变量的操作是直接在变量的原地址操作，因此若子函数改变了变量的值，主程序中变量的值也会更改；对于不可变（immutable）对象，包括strings,tuples,numbers等，子函数对变量值的操作是对新拷贝的一个副本操作，因此即使子函数改变了变量的值，主程序中变量的值也不会更改。