浙江大学

NA Project 报告

稀疏矩阵特征值

课程名称：	数值分析方法
姓　　名：	zbh
学　　院：	信息与电子工程学院
专　　业：	电子科学与技术
学　　号：	3220105xxx
指导老师：	余官定

2023 年 11 月 3 日

浙江大学 课程报告

专业：电子科学与技术
姓名：zbh
学号：3220105xxx
日期：2023 年 11 月 3 日
地点：西 2-213

课程名称：数值分析方法　指导老师：余官定　成绩：
作业名称：稀疏矩阵特征值　作业类型：大作业报告　同组学生姓名：无

目录

一、 背景介绍

在实际的工程和科学应用中，我们经常会遇到大规模的数据集，这些数据通常以矩阵的形式呈现。许多情况下，这些矩阵主要由零元素构成，只有少部分非零元素，我们称这种矩阵为“稀疏矩阵”。在诸如社交网络分析、图像处理、自然语言处理、有限元分析、物理学模拟等多个领域，我们都会遇到稀疏矩阵。由于这些领域的数据通常具有高维度，但又由于各种原因，矩阵中的数据元素只有极小一部分是非零的，因此，如何处理和利用这些稀疏矩阵变得非常重要。

作为一名正在学习数值分析方法课程的学生，我对这个问题有着浓厚的兴趣。我知道，计算和理解稀疏矩阵的特征值是一个具有挑战性的问题。然而，存在一些专门的数值计算方法，如 Power Method、QR 算法和 Arnoldi 迭代算法，可以帮助我们解决这个问题。

在本次期末大作业中，我打算深入研究这些算法，并将它们应用在普通矩阵和高维稀疏矩阵上，比较他们在不同类型矩阵上的性能。我相信，通过这项研究，我将能够更深入地理解稀疏矩阵的重要性，以及特征值计算方法的应用。

二、 设计思路

1. 设计目的

本研究的主要目的是探索和理解数值计算方法在处理稀疏矩阵特征值问题上的应用和性能表现。具体来说，设计目标可分为以下几点：

（1）深入理解 Power Method、QR 算法和 Arnoldi 迭代算法的工作原理和应用场景，以及如何将它们应用于稀疏矩阵的特征值计算。

（2）比较这些算法在普通矩阵和高维稀疏矩阵上的性能表现，包括计算速度、精度和稳定性等方面。

（3）深入理解稀疏矩阵在不同领域的应用，以及特征值计算在这些应用中的重要性。

（4）建立一个能够处理大规模稀疏矩阵特征值问题的计算模型，以展示这些数值计算方法在实际工程和科学应用中的效用。

通过完成这些目标，我们希望能够为稀疏矩阵特征值的计算提供一种有效的数值计算工具，同时也为理解和应用稀疏矩阵提供一种新的视角和理论支持。

2. 设计流程

- **Import necessary libraries**：首先，我们需要导入开发过程中需要用到的库。
- **Define functions**：接着，我们定义了一些数学计算函数，包括'sparse_sym_matrix'、'power_method'、'qr_iteration'、'arnoldi_iteration1' 和'arnoldi_iteration2'。这些函数分别用于生成稀疏矩阵、实现幂法、QR 迭代法和 Arnoldi 迭代法。
- **Define MainWindow class**：然后，我们定义了主窗口类'MainWindow'，该类主要用于构建用户界面和处理用户操作。
- **Define methods in MainWindow class**：在'MainWindow' 类中，我们定义了一些方法，包括'run_code' 和'open_readme'。'run_code' 方法用于获取用户输入，调用相应的函数执行计算，然后显示结果。'open_readme' 方法用于打开 GitHub 上的说明文档。
- **Create QApplication object and MainWindow object**：最后，我们创建了'QApplication' 对象和'MainWindow' 对象，并显示 GUI 窗口。

使用 Graphviz 绘制的流程图如下所示：

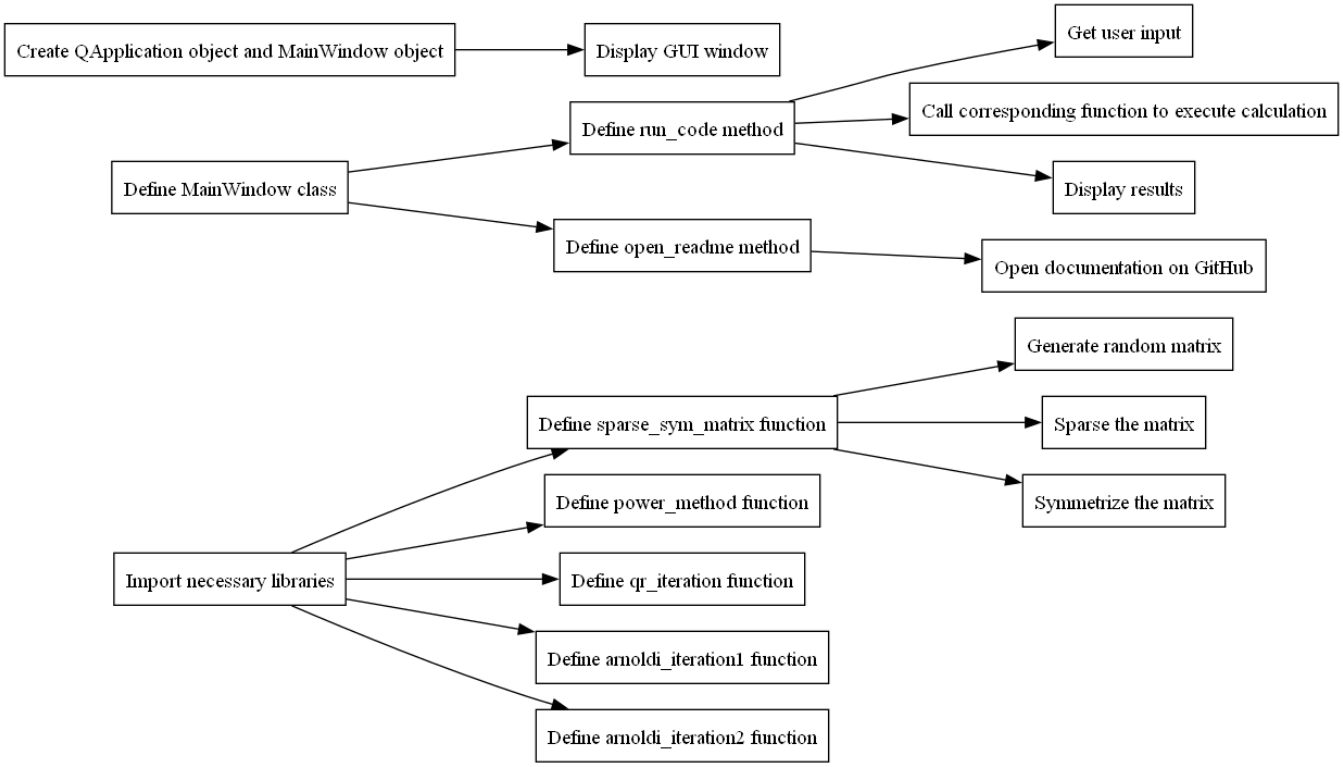

图 1: 设计流程图

3. 设计环境

运行环境为 Python 3.11.5，以下为运行 python 文件所需调用的库：

- **numpy**： Python 科学计算的核心库，提供了高性能的多维数组对象及相关工具。
- **scipy**：基于 numpy 的一种高级计算库，提供了许多有用的数学算法和函数。
- **PyQt5**： Python 编写的 GUI 框架，用于创建桌面应用程序。
- **sys**： Python 内置库，提供了访问与 Python 解释器紧密相关的变量和函数的途径。
- **webbrowser**： Python 内置库，提供了一个简单的接口，用于在 Web 浏览器中显示文档或打开 URL。

三、 算法描述

1. 稀疏矩阵的密度

矩阵 $A \in \mathbb{R}^{m\times n}$，其密度定义为非零元素数量与总元素数量的比值：

$$density = \frac{num(a_{ij} \neq 0)}{mn},$$

其中，a_{ij} 为矩阵 A 第 i 行第 j 列的元素。

2. Power Method

Power Method 是一种求解矩阵最大（绝对值）特征值和对应特征向量的算法。它的基本思想是，给定一个可对角化的矩阵 A 和一个非零向量 x ，重复计算 $y = Ax$ 和 $x = y/||y||$ ，直到 x 收敛到一个单位向量。这个单位向量就是 A 的一个特征向量，而 A 对这个特征向量的作用就是乘以一个标量，这个标量就是 A 的最大特征值。幂迭代法的优点是简单易实现，缺点是收敛速度较慢，且只能求解最大特征值和特征向量。

Algorithm 1 The Power Method

Require: dimension n; matrix A; vector x; tolerance TOL; maximum number of iterations N.

Ensure: approximate eigenvalue μ; approximate eigenvector x (with $\|x\|_\infty = 1$) or a message that the maximum number of iterations was exceeded.

Set $k = 1$.
Find the smallest integer $p \leq n$ such that $|x_p| = \|x\|_\infty$
Set $x = x/x_p$
while $k \leq N$ **do**
    Set $y = Ax$
    Set $\mu = y_p$
    Find the smallest integer $p \leq n$ such that $|y_p| = \|y\|_\infty$
    if $y_p = 0$ **then**
        OUTPUT ('Eigenvector'); STOP
    end if
    Set $ERR = \|x - (y/y_p)\|_\infty$
    Set $x = y/y_p$
    if $ERR < TOL$ **then**
        OUTPUT (μ, x); STOP
    end if
    Set $k = k + 1$
end while
OUTPUT ('Maximum number of iterations exceeded'); STOP

3. Givens 变换

又叫吉文斯旋转（Givens Rotation），一般用来数值求解矩阵的 QR 分解。数值求解矩阵的 QR 分解的基本思想是将原目标矩阵 A 左乘一系列正规正交变换矩阵 $H_1, H_2, ..., H_n$ 后得到一个上三角矩阵，该上三角矩阵即为 R 矩阵，$H_n...H_2H_1A = HA = Q^{-1}A = R$ ，其中 H 矩阵的逆矩阵即为 Q 矩阵。

吉文斯旋转，顾名思义，就是将原本的列向量保持长度不变旋转一个角度，使之平行于坐标轴即其中一个元素为零。一般使用吉文斯方法是逐列进行行变换的，从第一列开始，从下往上将对角线以下的元素都变为零，每次变换仅将一个元素变为零，例如找到一个变换矩阵

$$G = \begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix}$$

使得

$$\begin{pmatrix} g_1 & g_2 & \cdots & 0 \\ g_3 & g_4 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ 0 & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & b_{m,2} & \cdots & b_{m,n} \end{pmatrix}$$

即

$$\begin{pmatrix} g_1 & g_2 \\ g_3 & g_4 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} \sqrt{a_1^2 + a_2^2} \\ 0 \end{pmatrix} = \begin{pmatrix} b_1 \\ 0 \end{pmatrix}$$

可以发现满足该需求的变换矩阵为吉文斯变换矩阵

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

其中

$$c = cos\theta = \frac{a_1}{\sqrt{a_1^2 + a_2^2}}, s = sin\theta = \frac{a_2}{\sqrt{a_1^2 + a_2^2}}$$

这里一般使用的吉文斯变换矩阵的 c 和 s 的位置是固定的。

Algorithm 2 Givens Reduction

Require: matrix A of size $n \times n$

Ensure: orthogonal matrix Q and upper triangular matrix R such that $A = QR$

Set $R = A$
Set $G\_list = []$
for $j = 0$ to $n - 1$ **do**
    for $i = j + 1$ to $n - 1$ **do**
        if $R[i][j] = 0$ **then**
            continue
        end if
        Set $a = R[j][j]$
        Set $b = R[i][j]$
        Set $base = \sqrt{a^2 + b^2}$
        Set $c = a/base$
        Set $s = b/base$
        Set $G = I_n$
        Set $G[j][j] = c$
        Set $G[i][j] = -s$
        Set $G[j][i] = s$
        Set $G[i][i] = c$
        Set $R = GR$
        Append G to G_list
    end for
end for

Algorithm 3 Givens Reduction(cont'd)

Set $R = R[0:n]$
Set $Q\_prime = I_n$
for each G in G_list **do**
    Set $Q\_prime = GQ\_prime$
end for
Set $Q = Q\_prime[0:n]^T$
Return Q, R

4. QR 算法

QR 算法的核心思路是利用矩阵的 QR 分解, 迭代构造一系列矩阵并最终收敛为上三角矩阵, 从而计算特征值。具体而言, 矩阵 $A \in \mathbb{R}^{n\times n}$ 可以通过 QR 分解得到 $A = QR$, 其中 Q 为正交矩阵，R 为上三角阵。通过构建矩阵序列 $A_{k+1} = R_kQ_k = Q_k^\top A_kQ_k$, 可以得到矩阵序列 $\{A_k\}$，其中每一个矩阵 A_k 都与矩阵 A 相似，即具有相同的特征值。假设特征值各不相同且降序排列, 可以证明矩阵 A_k 中下三角的元素满足 $\left|a_{ij}^{(k)}\right| = \mathcal{O}\left(\left|\lambda_i/\lambda_j\right|^k\right), i > j$ 。因此, 当 k 趋于无穷时, A_k 收敛为上三角矩阵, 此时的对角线元素即为特征值。

有三种方法可以用于实现 QR 分解：Gram-Schmidt 正交化过程，Householder 变换和 Givens 变换。这里我们采用了 Givens 变换法来进行 QR 分解，下方给出了 QR 迭代的伪代码。

Algorithm 4 QR Iteration

Require: matrix A of size $n \times n$; number of eigenvalues k
Ensure: a list of k largest eigenvalues of A in absolute value

Set $M = A$
for $i = 1$ to 1000 **do**
    Set $Q, R =$ Givens_reduce(M)
    Set $M = RQ$
end for
Set $M_1 = QR$
Set $eigenvalues = []$
for $i = 0$ to $n - 1$ **do**
    Append $M_1[i][i]$ to $eigenvalues$
end for
Sort $eigenvalues$ by absolute value in descending order
Return $eigenvalues[0:k]$

5. Arnoldi 迭代

Arnoldi 迭代算法的核心思路是将矩阵投影到低维 Krylov 子空间, 从而将矩阵约化为上 Hessenberg 矩阵, 从而计算其特征值。通过低维 Krylov 子空间, Arnoldi 迭代并不直接利用矩阵本身, 而是利用矩阵向量乘积, 从而减小处理高维矩阵时的复杂度。具体而言, 对于给定的矩阵 $A \in \mathbb{R}^{n\times n}$ 和向量 $b \in \mathbb{R}^n$, Krylov 序列为向量集合 $\{b, Ab, A^2b, A^3b, \cdots\}$ 。M 维 Krylov 子空间定义为长度为 m 的 Krylov 序列张成的空间（可以把序列中的向量理解为基向量):

$$\mathcal{K}_m(A, b) \triangleq \text{span}\left\{a, Ab, A^2b, \cdots, A^{m-1}b\right\}$$

Arnoldi 迭代期望将矩阵分解为 $A = QHQ^\top$ ，其中 H 为上 Hessenberg 矩阵，Q 为正交矩阵。将矩阵 Q 表示为列向量形式 $Q = [q_1 \,|q_2|\cdots|\, q_n]$, 并矩阵分解形式转换为 $AQ = QH$, 其第 m 列的结果可以表示为:

$$Aq_m = h_{1m}q_1 + h_{2m}q_2 + \cdots + h_{m+1,m}q_{m+1},$$

其中，$h_{i,m} = 0, i > m+1$ 。利用上式可以推导向量 q_{m+1} 的递归形式:

$$q_{m+1} = \frac{Aq_m - \sum_{i=1}^{m} h_{i,m}q_i}{h_{m+1,m}}$$

不难发现, 实际上这与 Gram-Schmidt 求正交基的方法非常类似, q_m 就是 Krylov 子空间的正交基。通过求解的正交矩阵 Q, 我们可以计算 Hessenberg 矩阵 $H = Q^\top AQ$ 。由于 H 与 A 相似, 可以利用已有算法计算 H 的特征值从而得到 A 的特征值。

值得注意的是, 对于高维矩阵我们往往只关注其一部分的特征值, 比如只考虑前 $k \ll n$ 列。此时, 可以将矩阵分解简化为矩阵分解为 $AQ_k = Q_{k+1}\tilde{H}_k$, 其中 $Q_k = [q_1 \,|q_2|\cdots|\, q_k]$, $\tilde{H}_k = \begin{bmatrix} h_{11} & h_{12} & \cdots & & h_{1,k} \\ h_{21} & h_{22} & \cdots & & h_{2,k} \\ 0 & h_{32} & \ddots & h_{k-1,k-1} & \vdots \\ & 0 & \ddots & h_{k,k-1} & h_{k,k} \\ & & & 0 & h_{k+1,k} \end{bmatrix}$

Algorithm 5 Algorithm 1 Arnoldi Iteration

choose b arbitrarily, then $q_1 = b/\|b\|_2$
for $m = 1, 2, \ldots, k$ **do**
  $v = Aq_m$
  for $j = 1, 2, \ldots, m$ **do**
    $h_{jm} = v_j - q_j^T v$
    $v = v - h_{jm}q_j$
  end for
  $h_{m+1,m} = \|v_{m+1}\|_2$
  $q_{m+1} = v_{m+1}/h_{m+1,m}$
end for

四、 应用展示与计算分析

1. UI 界面

首先展示一下我们的 UI 界面，如下图所示：

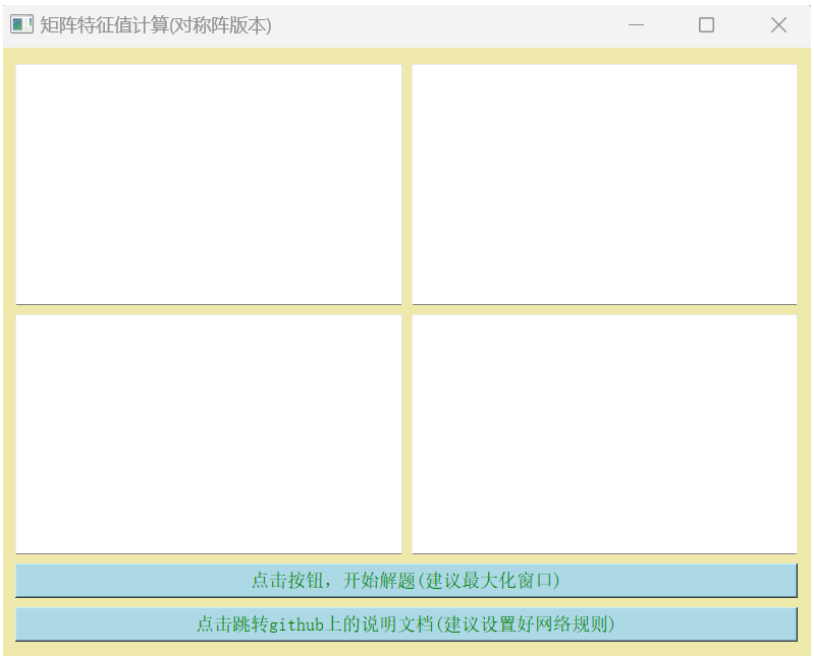

图 2: UI 界面

分为了四个界面，分别对应 4 道题的解答。同时在界面下方有两个按钮：

图 3: 解题按钮

解题按钮用于开启题目的计算，点击后稍等几秒，会在各个界面输出相应的计算结果，由于初始化界面较小，建议将界面最大化来清晰地比对结果，如下图所示：

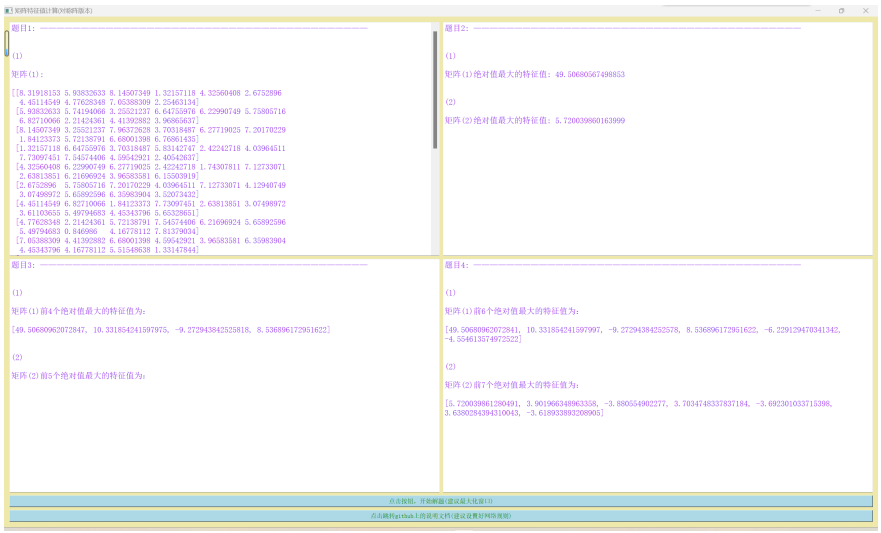

图 4: 计算结果

以及说明按钮：

图 5: 说明按钮

点击说明按钮，会在默认浏览器中打开本项目的 GitHub 网站上的说明文档 (github 有时候需要科学上网，所以请设置好自己的代理规则)，如下图所示：

图 6: 说明文档

2. 题目 1

生成随机矩阵
（1）生成一个 10×10 维度的随机矩阵。
（2）生成一个 10000×10000 维度且密度为 0.001 的随机稀疏矩阵，并统计矩阵中非零元素数量。
（3）利用库函数计算（1）和（2）中矩阵的特征值。

（1）

我选择直接使用 $'np.random.rand(10,10) * 10'$ 来生成元素大小在 0 到 10 之间的随机矩阵：

```
(1)

矩阵(1):

[[1.6671399  7.64520285 3.78155787 2.369026   9.59338551 0.03788529
  2.13531832 3.75638952 3.58540372 4.82500584]
 [8.70407807 9.18591152 9.34501425 2.92037875 5.1352615  2.102179
  5.88191786 9.92347835 6.78357504 5.91100569]
 [8.53108654 9.20272939 3.96618761 5.77079583 5.74167131 8.19904708
  8.41454372 0.31903549 2.57273462 1.89738801]
 [8.97696404 8.61995133 0.84264631 2.25438426 7.74532753 1.59585145
  4.04255945 8.78956418 4.78017321 2.65123908]
 [5.78462358 1.09684056 5.81431192 2.81023112 2.12661928 0.05467224
  2.78716823 9.99257211 9.31358151 2.38534872]
 [6.53962391 3.46801936 1.90391447 1.65251702 1.668157   0.9757846
  5.02526373 8.69325883 0.85033353 8.08632181]
 [3.47717466 3.79050801 1.48914098 6.11161372 0.61107049 6.2993239
  6.58615247 1.30633116 3.72485751 4.08418    ]
 [6.22195081 6.2054095  6.19642804 3.46383738 0.3403184  1.21344122
  7.4407202  4.94250026 9.02246576 8.99983121]
 [3.41409115 7.37097177 7.7950117  1.55988147 4.07928449 1.98571178
  1.2754164  4.04806649 3.72466762 1.67320986]
 [9.84331405 0.10833151 2.09031285 3.0450604  9.87822051 6.02597025
  7.7303269  7.46884749 1.3400377  4.01456167]]
```

图 7: 生成随机矩阵

当然这样生成出来的矩阵是一个不对称的矩阵,计算特征值的时候有很大概率是复数,所以我又使用 $'matrix = (matrix + matrix.T)/2'$ 生成一个对称矩阵：

```
(1)

矩阵(1):

[[0.67966229 5.58153248 2.88254033 2.4750723  6.41038925 2.90808371
  5.85940345 5.00113795 2.82411576 1.25963513]
 [5.58153248 9.10599673 4.44339343 4.88119003 6.17948754 5.12806149
  5.37065613 1.89041547 5.68646533 4.48479772]
 [2.88254033 4.44339343 0.33689124 7.34856362 2.78638581 7.65080922
  3.5393432  4.20454327 4.58901085 7.93922291]
 [2.4750723  4.88119003 7.34856362 5.7700472  3.73204847 3.67385784
  8.47680517 3.34205549 5.63808111 8.27352968]
 [6.41038925 6.17948754 2.78638581 3.73204847 7.36536921 8.76778541
  6.04937898 5.64437118 3.15173107 5.4401577 ]
 [2.90808371 5.12806149 7.65080922 3.67385784 8.76778541 1.89342768
  5.28023144 5.04227125 3.10694106 4.4946338 ]
 [5.85940345 5.37065613 3.5393432  8.47680517 6.04937898 5.28023144
  8.99261625 4.40595535 4.19971418 6.45193579]
 [5.00113795 1.89041547 4.20454327 3.34205549 5.64437118 5.04227125
  4.40595535 4.24922357 3.34115248 5.14445813]
 [2.82411576 5.68646533 4.58901085 5.63808111 3.15173107 3.10694106
  4.19971418 3.34115248 4.72758969 5.16144277]
 [1.25963513 4.48479772 7.93922291 8.27352968 5.4401577  4.4946338
  6.45193579 5.14445813 5.16144277 4.29364769]]
```

图 8: 生成随机对称矩阵

这样子生成出来的矩阵就是一个对称矩阵了，可以直接计算特征值，而且因为是实对称矩阵，特征值不会出现复数。

（2）

我使用 $'matrix = random(10000, 10000, density = 0.001) * 10'$ 来生成题目要求的随机稀疏矩阵，计算元非零素数量可以得到：

```
(2)

非零元素数量: 100000
```

图 9: 随机稀疏矩阵非零元素数量

然后我首先使用 $'np.triu_indices'$ 生成上三角矩阵的所有索引，然后从中随机选择一部分。使用 $'np.random.choice'$ 的 $'replace = False'$ 选项来确保所有选择的索引对都是唯一的。由于是在上三角矩阵中选择元素，然后再将其反射到下三角矩阵，因此，元素总数应该是 $n * (n + 1)/2$。

生成随机对称稀疏矩阵

```python
def sparse_sym_matrix(n, density):
    num_elements = int(n * (n + 1) * density / 2)
    indices = np.triu_indices(n)
    available_positions = len(indices[0])
    positions = np.random.choice(available_positions, size=num_elements, replace=False)
    row = indices[0][positions]
    col = indices[1][positions]
    data = np.random.rand(num_elements)
    upper_tri = sparse.coo_matrix((data, (row, col)), shape=(n, n))
    lower_tri = sparse.coo_matrix((data, (col, row)), shape=(n, n))
    return upper_tri + lower_tri
```

这种算法的好处在于避免了对很大的矩阵进行直接的操作运算，生成速度很快，缺点在于由于生成元素具有随机性，所以生成的矩阵的密度可能会与预期有一点点不符。值得一提的是，我使用 $'matrix2 = csc\_matrix(matrix\_2)'$ 将稀疏矩阵以稀疏矩阵格式存储，极大地节省了内存空间。

（3）

接下来是使用库函数来计算我们生成矩阵的特征值，这里我使用了 $'np.linalg.eigvals'$ 和 $'scipy.sparse.linalg.eigs'$ 来计算特征值，结果如下：

```
(3)

矩阵(1)特征值:

[47.57027465+0.j          -5.77427048+0.j          -3.98146678+3.8053475j
 -3.98146678-3.8053475j   -1.93229809+5.88109418j -1.93229809-5.88109418j
  0.60922349+2.19593739j   0.60922349-2.19593739j   4.12849389+1.34134407j
  4.12849389-1.34134407j]

矩阵(2)特征值:

[50.06775765 +0.j           6.03092229-17.46822375j
  6.03092229+17.46822375j 16.66281057 -7.8536371j
 16.66281057 +7.8536371j    6.75407385-17.10748408j
  6.75407385+17.10748408j -0.63337197-18.34036872j]
```

(a) 非对称阵特征值

```
(2)

非零元素数量: 100000

(3)

矩阵(1)特征值:

[49.71305017 -9.89308741   8.56943007   5.88741249   4.13755035   1.16554782
 -5.04865489 -1.15876119 -3.39302293 -2.56499293]

矩阵(2)特征值:

[ 5.7784884 +0.j   4.13116138+0.j -4.13534675+0.j   4.02253503+0.j
 -4.0646134 +0.j   4.00078419+0.j -4.01366758+0.j]
```

(b) 对称阵特征值

图 10: 库函数特征值计算结果

可以看到，对于非对称矩阵，库函数计算出来的特征值有可能是复数，而对于对称矩阵，库函数计算出来的特征值都是实数，与预期相同。

注意：后面的题目对应的非对称阵和对称阵均对应该题产生的非对称阵和对称阵。

3. 题目 2

（1）利用 Power Method 计算题目 1（1）中矩阵的绝对值最大的特征值。
（2）利用 Power Method 计算题目 1（2）中稀疏矩阵的绝对值最大的特征值。

根据算法部分写出的伪代码，我们可以得到以下代码：

Power Method 代码

```python
# 实现power method
def power_method(matrix, tol, max_iter):
    k = 1
    n = matrix.shape[0]
    x = np.random.rand(n)
    x = x / np.linalg.norm(x, np.inf)  # 归一化，使得||x||∞ = 1
    while k <= max_iter:
        p = np.argmin(np.abs(x))
        x = x / x[p]
        y = matrix @ x
        p = np.argmin(np.abs(y))
        if y[p] == 0:
            return "y[p] == 0", x
        mu = y[p]
        err = np.linalg.norm(x - y / y[p], np.inf)
        x = y / y[p]
        if err < tol:
            return mu, x
        k += 1
    return "The maximum number of iterations exceeded"
```

求解即可得到：

```
题目2：————————————————————————————
(1)
矩阵(1)绝对值最大的特征值：47.57026906928842
(2)
矩阵(2)绝对值最大的特征值：50.06775802785183
```

(a) 非对称阵特征值

```
题目2：————————————————————————————
(1)
矩阵(1)绝对值最大的特征值：49.71303826461942
(2)
矩阵(2)绝对值最大的特征值：5.778488399195702
```

(b) 对称阵特征值

图 11: Power Method 特征值计算结果

对比库函数计算结果，我们可以发现幂法的计算结果到达预期。需要注意的（2）有时候可能遇到 $y[p] == 0$ 的情况，会输出 $'y[p] == 0'$，这种情况就需要重新计算了。

该代码中最消耗运行时间的操作在于 while 循环，在其中我们进行了 3 个关键操作：

a. 矩阵和向量的乘法：y = matrix @ x，其时间复杂度为 $O(n^2)$

b. 向量的归一化：x = y / y[p] 和 x = x / x[p]，两者都需要 $O(n)$ 的运算

c. 计算无穷范数（调用了 np.linalg.norm 函数），此操作也需要 $O(n)$ 的运算

因为所有这些操作，都在每次迭代中执行，而且我们有最多 max_iter 次迭代，所以总的时间复杂度就是 $O(max\_iter * n^2)$。

在这段代码中，除了输入的矩阵之外，我们只存储了几个大小为 n 的向量 (x, y) 以及一些单元素变量（如 mu，err 和 p）。虽然在实际的运行过程中，尽管我们没有直接保存更大的结构，但 Python 解释器或 Numpy 库可能会在后台使用额外的内存来完成某些操作。然而，从算法本身的角度来看，我们只需考虑我们明确要求存储的数据结构，那么空间复杂度就是 $O(n)$。

4. 题目 3

（1）利用 QR 算法计算题目 1（1）中矩阵的前 4 个绝对值最大的特征值。

（2）利用 QR 算法计算题目 1（2）中稀疏矩阵的前 5 个绝对值最大的特征值。

（1）

我们使用基于 Givens 变换的 QR 分解法，再利用 QR 迭代求解特征值，其中 QR 迭代代码如下，我们没有再通过设置 tol 和 max_iter 的方式来判断结束条件，因为那样可能导致稀疏矩阵输出不了结果，我们索性设置了 1000 次迭代：

QR Iteration 代码

```python
# 实现QR迭代求解特征值
def qr_iteration(matrix, k):
    n = matrix.shape[0]
    for i in range(1000):
        q, r = givens_reduce(matrix)
        matrix = np.dot(r, q)
    matrix_1 = np.dot(q, r)
    eigenvalues = []
    for i in range(n):
        eigenvalues.append(matrix_1[i, i])
        eigenvalues = sorted(eigenvalues, key=lambda x: abs(x), reverse=True)[:k]
    return eigenvalues
```

计算题目 1（1）随机矩阵特征值，我们得到图 12 所示的结果：

我们可以看到，非对称阵关于复数特征值的计算存在误差，但是对于对称阵，QR 算法对于实数特征值计算出来的特征值与库函数计算结果相差很小。

（2）

我们可以看到题目 3（2）是没有输出的，因为我把输出与计算部分的函数注释掉了，因为使用 python，加上我的 QR 算法可能存在的问题，导致计算 10000×10000 的稀疏矩阵特征值很慢，于是我这里选择用小一点的稀疏矩阵来测试，生成 100×100 且密度为 0.01 的稀疏矩阵，然后取消我代码中注释掉的部分，运行程序，得到结果如图 13 所示：

可以观察到对于稀疏矩阵，最大绝对值的特征值计算还是准确的，但随着继续计算余下的特征值，误差越来越大，而对于小的密集矩阵，计算结果就没有问题。

```
题目3：————————————————————————————————————————————————————

(1)

矩阵(1)前4个绝对值最大的特征值为：

[47.57027464712886, 6.956866297693818, -5.774270478127092, -4.200200497755604]
```

(a) 非对称阵特征值

```
题目3：————————————————————————————————————————————————————

(1)

矩阵(1)前4个绝对值最大的特征值为：

[49.71305017199713, -9.893087412795618, 8.569430071601326, 5.887412486319471]
```

(b) 对称阵特征值

图 12: QR 特征值计算结果

```
(3)

矩阵(1)特征值：

[ 49.82701784 -13.1242112   -9.46391321  -7.66228555   6.27180039
   5.66602545   4.42918551  -5.08442033   1.7133749   -1.48522812]

矩阵(2)特征值：

[ 1.79456523+0.j   1.30591449+0.j   1.2431851 +0.j   1.21468667+0.j
 -1.30591449+0.j -1.2431851 +0.j -1.21468667+0.j]
```

(a) 库函数特征值计算结果

```
题目3：————————————————————————————————————————————————————

(1)

矩阵(1)前4个绝对值最大的特征值为：

[49.827017839713825, -13.124211196209172, -9.46391320992059, -7.662285550838423]

(2)

矩阵(2)前5个绝对值最大的特征值为：

[1.7945652283039153, -0.9867379644010551, 0.9576160025862197, -0.5783442478344106, 0.5279778184897216]
```

(b) QR 法特征值计算结果

图 13: 特征值计算结果

在 *'givens_reduce'* 函数中，存在两层 for 循环。在最坏的情况下，它们需要进行 $n*(n-1)/2$ 次迭代（其中 n 是输入矩阵的维数），每次迭代中都要计算矩阵乘法，这个操作的时间复杂度约为 $O(n^3)$。在 *'qr_iteration'* 函数中，可能会执行 1000 次 *'givens_reduce'* 调用。因此，该功能的总体时间复杂度大约为 $O(1000*n^4)$。因此，整体来看，这段代码的时间复杂度约为 $O(n^4)$。

'givens_reduce' 函数和 *'qr_iteration'* 函数都创建了一些额外的数据结构（比如矩阵 G, Q, R 等），但所有这些结构的大小都与输入矩阵的尺寸相当。*'qr_iteration'* 函数创建了一个名为 eigenvalues 的额外列表来存储计算出的特征值，但其大小只取决于输入矩阵的列数（即 n）。因此，这段代码的空间复杂度约为 $O(n^2)$，其中 n 是输入矩阵的维数。

5. 题目 4

（1）利用 Arnoldi 迭代算法计算题目 1（1）中矩阵的前 6 个绝对值最大的特征值。
（2）利用 Arnoldi 迭代算法计算题目 1（2）中稀疏矩阵的前 7 个绝对值最大的特征值。

我们先使用 Arnoldi 迭代得到矩阵 H，然后对矩阵 H 使用 QR 迭代法求解特征值。

Arnoldi Iteration 代码

```python
# 实现Arnoldi迭代求解特征值
def arnoldi_iteration1(matrix, k):
    n = matrix.shape[0]
    Q = np.zeros((n, n + 1))
    H = np.zeros((n + 1, n))
    b = np.random.rand(n)
    Q[:, 0] = b / np.linalg.norm(b)
    for j in range(n):
        v = np.dot(matrix, Q[:, j])
        for i in range(j + 1):
            H[i, j] = np.dot(Q[:, i], v)
            v = v - H[i, j] * Q[:, i]
        H[j + 1, j] = np.linalg.norm(v)
        if H[j + 1, j] == 0:
            break
        Q[:, j + 1] = v / H[j + 1, j]
    eigenvalues = qr_iteration(H[:n,], k)
    return eigenvalues
```

对矩阵（1）使用 Arnoldi 算法时，由于迭代次数少，我们直接选 10 列来迭代，让计算结果更准确（这里展示的就是针对小的密集矩阵的代码）；而对于稀疏矩阵，我们选择一个较少的列来迭代（这里我们选择前 30 列，如果想让计算结果更准确可以选择更多的列数），增加迭代速度，同时也能得到较准确的结果：

```
题目4：————————————————————————————————————————————

(1)

矩阵(1)前6个绝对值最大的特征值为：

[47.570274647128954, 6.881071278634113, -5.774270478127079, -4.411565311500248, -3.5513682547298466,
-2.4885644440308328]

(2)

矩阵(2)前7个绝对值最大的特征值为：

[50.06795272074899, -8.962782333164652, 8.212380262964468, 7.773887004794749, -6.442010305481496,
-4.27344134761224, -0.9462374451086519]
```

图 14: 非对称阵特征值

```
题目4：————————————————————————————————————————————

(1)

矩阵(1)前6个绝对值最大的特征值为：

[49.71305017199707, -9.893087412795584, 8.569430071601358, 5.887412486319513, -5.04865489447157,
4.1375503511496605]

(2)

矩阵(2)前7个绝对值最大的特征值为：

[5.778488399241921, 4.080638541648307, -4.071120700091999, -4.05394892801095, 4.004468372895848,
3.8997952138439147, -3.8761917668098618]
```

图 15: 对称阵特征值

观察结果，我们可以发现对于密集矩阵，实数的特征值求解结果准确，但对于复数特征值求解存在误差；对于稀疏矩阵，我们可以看到，Arnoldi 迭代法求解特征值的结果与 QR 迭代法求解特征值的结果相差很小，但是 Arnoldi 迭代法的计算速度要快很多。同样，因为只选取前 m 列（$m << n$），计算得到的最大值准确，越往后计算误差越大，可以适当增加 k 值来提升计算精度，但计算时间也会增加。

对于取 n 列计算的 arnoldi_iteration1 函数，包含一个 for 循环，该循环在最坏情况下需要迭代 n 次（其中 n 是输入矩阵的维数）。在这个 for 循环中，我们执行一次矩阵向量乘法操作（时间复杂度为 $O(n^2)$），然后有一个内部 for 循环（最多运行 j 遍）在当中进行了一些点积操作（时间复杂度为 $O(n)$）。最后，调用 qr_iteration 函数可能要执行最多 1000 次操作，每次都会引发两次 $n \times n$ 矩阵之间的乘法。故此步操作的时间复杂度约为 $O(1000 * n^3)$。因此，整体来看，这段代码的时间复杂度约为 $O(n^3)$，其中大部分时间消耗在对 qr_iteration 函数的调用上。

arnoldi_iteration1 函数创建了几个额外的数据结构（如 Q, H 等），但所有这些数据结构的大小都与输入矩阵大小相当。它还创建了一个名为 eigenvalues 的额外列表来存储特征值，其大小仅取决于输入矩阵的列数（即 n）。因此，这段代码的空间复杂度约为 $O(n^2)$，其中 n 是输入矩阵的维数。

通过选取前 m 列来进行迭代的 arnoldi_iteration2 函数，分析方法类似，函数中包含了一个可能会迭代到 m 次（这里预设的常量 m 为 30）的循环。在这个循环中，我们执行了一个矩阵与向量的乘法运算（它的时间复杂度为 $O(n^2)$），然后再内部循环中，执行了一些点积运算（每个运算的时间复杂度都是 $O(n)$），算上 QR 迭代，总的时间复杂度约为 $O(n * m^2)$ 以及空间复杂度为 $O(m^2)$

五、 使用说明

你可以在这里找到更多信息：`https://github.com/Sekiro1233/NA-Project/tree/master`

六、 总结与展望

1. 优点

（1）我的程序可以计算矩阵的特征值，而且对于稀疏矩阵的计算也有较好的效果，对于实数特征值的计算不存在太大问题。

（2）加入了 UI 界面，可以直观地展示计算结果，同时可以在界面下方的说明按钮中查看本项目的说明文档，方便使用。

（3）对于题目中的矩阵，我都进行了一定的处理，使得计算结果更加准确，比如对于非对称矩阵，我将其转化为对称矩阵，对于稀疏矩阵，我将其转化为稀疏矩阵格式，这样可以节省内存空间，加快计算速度。

（4）使用 Python 语言，代码简洁，易于理解，同时使用了 numpy 库，加快了计算速度。

2. 不足

（1）对于复数特征值的计算，我的程序存在一定的误差。

（2）对于稀疏矩阵的计算，QR 算法计算结果存在一定误差，同时 QR 算法计算稀疏矩阵特征值的速度较慢。

（3）对于 Arnoldi 算法，我只是简单地选取了前 m 列来进行迭代，这样计算出来的特征值可能会有一定的误差，如果想要计算出更准确的特征值，需要增加迭代次数，但这样会增加计算时间。

（4）整体程序较为简陋，没有加入太多的交互功能，比如可以让用户自己输入矩阵，而是直接在代码中设置了矩阵，这样用户体验不是很好。

3. 展望

（1）对于复数特征值的计算，可以优化现有算法或使用其他方法来提高计算精度。

（2）对于稀疏矩阵的计算，可以使用其他方法来提高计算精度，同时可以考虑使用其他语言来编写程序，比如 C++，这样可以加快计算速度。

（3）可以考虑增加 UI 的交互性与实用性。