**北京邮电大学软件学院**

**2019-2020学年第一学期实验报告**

**课程名称： 数值计算与分析**

**项目名称： 实验二：矩阵特征值算法实现**

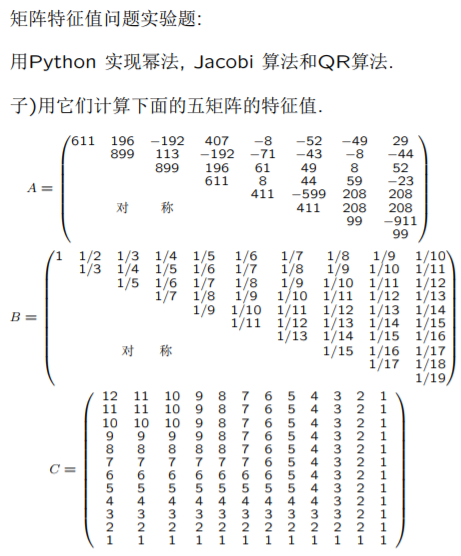
**项目完成人：**

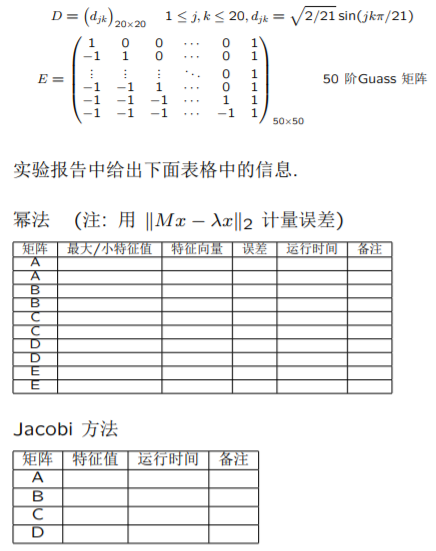
**姓名：\_刘子豪\_学号：\_2017211971\_**

**指导教师：**\_\_\_\_\_\_\_\_\_\_\_漆涛\_\_\_\_\_\_\_\_\_\_\_

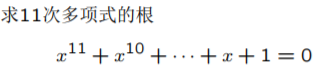
**日 期： 2019年 12 月 20 日**

1. **实验问题描述**
2. **矩阵特征值问题**

****

****

1. **多项式方程求解**

****

1. **实验说明**
2. **幂法、反幂法**

实现了基本的幂法和反幂法程序求解矩阵最大，最小特征值，并得到相应的特征向量。在程序中展示出了总迭代次数，最后得到的误差以及运行时间。

在实验算法的过程中，本考虑通过原点平移的方法进行算法的加速，但是由于没有一个合适的算法去找到ｐ的值，因此没有采用这个方法。

幂法的实现基于教材P247的定理8，算法中控制的最大迭代次数为1000，误差不大于1e-4.在每次迭代之后，误差的计算公式为.

反幂法的实现基于教材P251的定理10。每次迭代中都需要用到矩阵的逆,由于计算矩阵的逆耗时较长，因此转化为求线性方程组的解。在本次实验中使用**直接三角分解法**实现线性方程组的求解（该方法基于教材P152中的内容进行实现）。

1. **Jacobi算法求特征值**

Jacobi算法的实现基于讲义中的内容和网络上的资料，实现了矩阵特征值的求解，并展示出总迭代次数和运行时间。

1. **单步QR算法**

在实验的初期首先对单步的QR算法进行实现。在算法的实现中，涉及到了以下的几个内容，均通过阅读教材进行理解实现：

* 豪斯霍尔德变换，基于教材P255中的定理14进行实现。
* 将一般矩阵进行正交相似变换，得到上海森伯格矩阵，该问题基于教材P261中所讲的内容进行实现。
* 单步QR方法计算上海森伯格矩阵的特征值，该算法基于教材P270中的算法1进行实现。

1. **双步QR算法**

由于之前实现的单步QR算法无法求复数的特征值，因此在对矩阵E进行求解时结果不理想，而且在求多项式方程时，不仅求得复根，求实根的效果也很不理想。因此通过阅读教材，决定对双步QR算法进行实现，将该算法应用于矩阵E的求解和第二题中的多项式方程求解。

对双步QR算法进行实现时，首先阅读教材P273中双步QR方法的流程，理解了一次迭代的过程，之后又通过查找资料，文献等方法，理解了双步QR算法求一个矩阵所有特征值的整体流程，最终使用python语言进行了实现。

对多项式求根进行实现时，实现了一个函数，输入任意多项式方程的系数，在函数中能够将其转换成对应的矩阵进行特征值求解，从而实现了求任意阶数，任意系数的多项式方程的所有根。

1. **实验结果**

**所有的结果和运行时间都保留了四位小数，误差保留六位小数**

1. **幂法、反幂法**

**备注中给出一些异常的结果出现原因。**

**由于特征向量不方便写在表格中，因此在表格之后给出。**

**算法中控制的最大迭代次数为1000，误差不大于1e-4.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 矩阵 | 最大/小特征值 | 误差 | 运行时间 | 备注 |
| A | 1016.4980 | 1148.816076 | 0.1496 s | 迭代次数超上限 |
| A | 0.0000 | 0.000000 | 0.0000 s | 速度过快 |
| B | 1.7520 | 0.000063 | 0.0010 s |  |
| B | 0.0000 | 0.000001 | 0.0000 s | 速度过快 |
| C | 63.4092 | 0.000092 | 0.0030 s |  |
| C | 0.2540 | 0.000001 | 0.0788 s | 迭代次数为231，但是得到了结果 |
| D | 0.2428 | 4.160649 | 0.8677 s | 迭代次数超上限 |
| D | 4.1181 | 17.133797 | 0.8836 s | 迭代次数超上限 |
| E | 10.3130 | 30.240098 | 5.0306 s | 存在复数特征值 |
| E | 1.4976 | 0.105141 | 4.8032 s | 存在复数特征值 |

**特征向量**：

|  |  |
| --- | --- |
| 矩阵 | 最大/小特征值对应的特征向量 |
| A | [1.0000, 0.7157, 0.7157, 1.0000, -0.3536, -0.0085  , 0.2758, 0.4484] |
| A | [1.0000, 2.0000, -2.0000, -1.0000, 14.0000, 14.0000  , 7.0000, 7.0000] |
| B | [1.0000, 0.6090, 0.4531, 0.3653, 0.3078, 0.2667  , 0.2358, 0.2116, 0.1920, 0.1759] |
| B | [1.0000, -99.0039, 2376.1663, -24026.2697, 126140.4405,  -378427.6258, 672769.6971, -700238.1278, 393888.0591,  -92394.3387] |
| C | [1.0000, 0.9842, 0.9529, 0.9066, 0.8460, 0.7720  , 0.6859, 0.5889, 0.4827, 0.3689, 0.2492, 0.1256] |
| C | [1.0000, -2.9372, 4.6898, -6.1479, 7.2196, -7.8378  , 7.9635, -7.5888, 6.7373, -5.4624, 3.8442  , -1.9845] |
| D | [1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000  , 1.0000, 1.0000, 1.0000, 1.0000, 1.0000  , 1.0000, 1.0000, 1.0000, 1.0000, 1.0000  , 1.0000, 1.0000, 1.0000, 1.0000] |
| D | [1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000  , 1.0000, 1.0000, 1.0000, 1.0000, 1.0000  , 1.0000, 1.0000, 1.0000, 1.0000, 1.0000  , 1.0000, 1.0000, 1.0000, 1.0000] |
| E | [-0.0925, -0.0970, -0.1015, -0.1059, -0.1102, -0.1144  , -0.1185, -0.1224, -0.1262, -0.1296, -0.1328  , -0.1356, -0.1379, -0.1399, -0.1413, -0.1421  , -0.1423, -0.1417, -0.1403, -0.1380, -0.1347  , -0.1304, -0.1249, -0.1181, -0.1100, -0.1003  , -0.0891, -0.0762, -0.0614, -0.0447, -0.0259  , -0.0049, 0.0185, 0.0444, 0.0729, 0.1041  , 0.1383, 0.1755, 0.2159, 0.2596, 0.3068  , 0.3576, 0.4121, 0.4704, 0.5328, 0.5992  , 0.6698, 0.7446, 0.8238, 1.0000] |
| E | [1.0000, -0.6460, 0.4625, -0.3739, 0.2920, -0.1319  , -0.1507, 0.5216, -0.8485, 0.9036, -0.4042  , -0.8739, 2.9154, -5.3005, 7.1432, -7.1478  , 3.7338, 4.8039, -20.0549, 43.1092, -74.0887  , 111.8577, -154.1679, 198.1187, -240.5566, 278.3077  , -308.4553, 328.7142, -337.7609, 335.4423, -322.7586  , 301.5401, -273.9955, 242.3863, -208.9074, 175.6140  , -144.1900, 115.7455, -90.8778, 69.8073, -52.4974  , 38.6959, -27.9505, 19.7484, -13.6615, 9.3199  , -6.2750, 4.0753, -2.5934, 0.6141] |

**结果分析**：首先可以看到，对于一部分矩阵，幂法和反幂法的收敛速度很慢，最终导致误差极大。误差大与收敛速度慢的原因具体有下面几个：

* 矩阵本身的性质：根据教材，幂法的收敛速度取决于，反幂法的收敛速度取决于,因此对于不同矩阵，算法的收敛速度都会有区别。
* 算法中初始向量的选取。对于每个矩阵，如果找到合适的初始向量，会对收敛速度有很大的提升。
* p的选取。如果p的值选取合理，则会对算法速度有很大的提升。
* 存在复数特征值。对于矩阵E，有很多特征值是复数，而幂法和反幂法只能得到实数特征值。因此对于这种矩阵无法得到一个比较理想的结果。

1. **Jacobi算法**

**算法最大迭代次数为200，最小误差为1e-4。**

|  |  |  |  |
| --- | --- | --- | --- |
| 矩阵 | 特征值 | 运行时间 | 备注 |
| A | [1020.0490, 1020.0000, 1019.9020, 1000.0000,  1000.0000, 0.0980, 0.0000, -1020.0490] | 0.0559 s | 正确 52次迭代 |
| B | [1.7519, 0.3429, 0.0357, 0.0025, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000] | 0.1456 s | 正确 64次迭代 |
| C | [63.4091, 7.1201, 2.6180, 1.3790, 0.8707, 0.6153, 0.4705, 0.3820, 0.3256, 0.2892, 0.2665, 0.2540] | 0.6064 s | 正确 169次迭代 |
| D | [1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,  -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000] | 3.0060 s | 正确 189次迭代 |

对于题中的四个矩阵，Jacobi算法均能在最大迭代次数的限制下运行成功，得到误差小于1e-4的结果。根据矩阵的不同，算法的收敛速度也有所不同（根据迭代次数即可判断）。

1. **单步QR算法**

**求每个特征值时最大迭代次数为100，最小误差为1e-6**

**下标展示单步QR算法对所有的五个矩阵的求解情况。**

|  |  |  |  |
| --- | --- | --- | --- |
| 矩阵 | 特征值 | 运行时间 | 备注 |
| A | [1000.0000, 1019.9020, 1020.0000, 1020.0490, 1000.0000, 0.0000, -1020.0490, 0.0980] | 0.0090 s | 正确 |
| B | [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0001, 0.0025, 0.0357, 1.7519, 0.3429] | 0.0200 s | 正确 |
| C | [0.2665, 0.2540, 0.2892, 0.3256, 0.3820, 0.4705, 0.6153, 0.8707, 1.3790, 2.6180, 63.4091, 7.1201] | 0.0439 s | 正确 |
| D | [1.0000, -1.0000, 1.0000, -1.0000, 1.0000,  -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000,  -1.0000, 1.0000, -1.0000, -1.0000, 1.0000] | 0.3221 s | 正确 |
| E(单步QR算法) | [1.4981, 1.4978, 1.4985, 1.4973, 1.4981, 1.4977  , 1.4974, 1.4983, 1.4966, 1.4989, 1.4957  , 1.4995, 1.4951, 1.4998, 1.4951, 1.4995  , 1.4959, 1.4982, 1.4976, 1.4959, 1.4991  , 1.4936, 1.4983, 1.4934, 1.4939, 1.4965  , 1.4910, 1.4976, 1.4960, 1.4901, 1.4898  , 1.4928, 1.4916, 1.4857, 1.4823, 1.4866  , 1.4797, 1.4750, 1.4632, 1.4654, 1.4513  , 1.4236, 1.4408, 1.3031, 0.9626, 1.3696  , 0.1432, 0.1432, -12.1739, -5.7762] | 15.3140 s | 存在复根，因此效果不理想 |
| E(双步QR算法) | [-1.498+0.048j, -1.498-0.048j, -1.498+0.016j, -1.498-0.016j, -1.498+0.081j, -1.498-0.081j  , -1.498+0.114j, -1.498-0.114j,  -1.498+0.148j, -1.498-0.148j, -1.498+0.183j  , -1.498-0.183j, -1.497+0.220j,  -1.497-0.220j, -1.497+0.260j, -1.497-0.260j  , -1.497+0.302j, -1.497-0.302j,  -1.497+0.347j, -1.497-0.347j, -1.496+0.397j  , -1.496-0.397j, -1.496+0.451j,  -1.496-0.451j, -1.495+0.513j, -1.495-0.513j  , -1.494+0.583j, -1.494-0.583j,  -1.493+0.664j, -1.493-0.664j, -1.491+0.760j  , -1.491-0.760j, -1.489+0.877j,  -1.489-0.877j, -1.484+1.022j, -1.484-1.022j  , -1.477+1.210j, -1.477-1.210j,  -1.464+1.466j, -1.464-1.466j, -1.437+1.835j  , -1.437-1.835j, -1.372+2.421j,  -1.372-2.421j, -1.166+3.492j, -1.166-3.492j  , -0.143+5.937j, -0.143-5.937j,  -8.975+9.485j, -8.975-9.485j] | 151.6825 s | 运行时间很长，但是得到了最后的结果 |

对于前四个矩阵，单步QR算法均求出了正确的结果，而且运行时间远小于Jacobi算法。第五个矩阵中由于有复数特征值的存在，单步QR算法没有得到理想的结果。

1. **双步QR算法**

**求每个特征值时最大迭代次数为100，最小误差为1e-6**

|  |  |  |  |
| --- | --- | --- | --- |
| 矩阵 | 特征值 | 运行时间 | 备注 |
| A | [1000.0000, -1019.9020, -1020.0000, 1020.0490, 1000.0000, 0.0000, -0.0980,  -1020.0490] | 0.0872 s | 正确 |
| B | [0.0000, 0.0000, 0.0000, 0.0000, -0.0000,  -0.0001, 0.0025, 0.0357, 1.7519, 0.3429] | 0.0347 s | 正确 |
| C | [0.2665, -0.2540, -0.2892, -0.3256, -0.3820, -0.4705, -0.6153, 0.8707, 1.3790, 2.6180, 63.4091, 7.1201] | 0.3569 s | 正确 |
| D | [1.0000, -1.0000, 1.0000, -1.0000, 1.0000,  -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000,  -1.0000, 1.0000, -1.0000, 1.0000, -1.0000] | 0.4184 s | 正确 |
| E | [-1.498+0.048j, -1.498-0.048j, -1.498+0.016j, -1.498-0.016j, -1.498+0.081j, -1.498-0.081j  , -1.498+0.114j, -1.498-0.114j,  -1.498+0.148j, -1.498-0.148j, -1.498+0.183j  , -1.498-0.183j, -1.497+0.220j,  -1.497-0.220j, -1.497+0.260j, -1.497-0.260j  , -1.497+0.302j, -1.497-0.302j,  -1.497+0.347j, -1.497-0.347j, -1.496+0.397j  , -1.496-0.397j, -1.496+0.451j,  -1.496-0.451j, -1.495+0.513j, -1.495-0.513j  , -1.494+0.583j, -1.494-0.583j,  -1.493+0.664j, -1.493-0.664j, -1.491+0.760j  , -1.491-0.760j, -1.489+0.877j,  -1.489-0.877j, -1.484+1.022j, -1.484-1.022j  , -1.477+1.210j, -1.477-1.210j,  -1.464+1.466j, -1.464-1.466j, -1.437+1.835j  , -1.437-1.835j, -1.372+2.421j,  -1.372-2.421j, -1.166+3.492j, -1.166-3.492j  , -0.143+5.937j, -0.143-5.937j,  -8.975+9.485j, -8.975-9.485j]] | 151.6825 s | 运行时间很长，但是得到了正确的结果 |

对前四个矩阵应用双步QR算法能够取得正确的结果，但是运行时间比单步QR算法要长。运行双步QR算法对矩阵E进行求解的时候花了相当长的时间，但是最后得到了符合我们要求的特征值。

1. **多项式方程的求解**

首先将多项式转化为相应的矩阵，然后使用双步QR算法，对该矩阵的特征值进行求解，结果如下：

[0.500+0.866j, 0.500-0.866j, 0.866+0.500j, 0.866-0.500j, -1.0000, 0.000+1.000j, 0.000-1.000j, -0.500+0.866j, -0.500-0.866j, 0.866+0.500j, 0.866-0.500j]

运行时间为0.4388 s.

可以看到，该多项式方程存在1个实根和10个复根，其中这十个复根中存在５对共轭复数，使用双步QR算法均可将它们求出。

1. **程序源代码**

**在贴出源代码之前，首先对所有代码文件进行展示，并说明实现了什么功能：**

* **MyMatrix.py:** 生成题中所要求的5个矩阵, 其中矩阵A手工输入, 其他矩阵通过算法生成.
* **MyFunction.py:** 实现了实验中所要用到的基本矩阵运算,主要包括矩阵乘法、转置, 向量取模，矩阵的LU分解以及方程求解。
* **power.py:** 实现了幂法和反幂法的算法.
* **jacobi.py:** 实现了Jacobi算法.
* **QR.py:** 实现了单步QR算法, 其中还实现了Household变换, 上海森伯格矩阵的生成.
* **doubleQR.py:** 实现了双步QR方法, 其中调用了QR.py中的函数.
* **polySolve.py:** 实现了任意阶数,任意系数的多项式方程求解. 核心部分是将多项式转化为相应的矩阵.

**下面贴出所有文件的程序代码：**

1. **MyMatrix.py**

**import** numpy **as** np  
**import** math  
  
  
*# 题目中的5个矩阵定义***def** getA():  
 A = np.array([[611, 196, -192, 407, -8, -52, -49, 29],  
 [196, 899, 113, -192, -71, -43, -8, -44],  
 [-192, 113, 899, 196, 61, 49, 8, 52],  
 [407, -192, 196, 611, 8, 44, 59, -23],  
 [-8, -71, 61, 8, 411, -599, 208, 208],  
 [-52, -43, 49, 44, -599, 411, 208, 208],  
 [-49, -8, 8, 59, 208, 208, 99, -911],  
 [29, -44, 52, -23, 208, 208, -911, 99]],  
 dtype=float)  
 **return** A  
  
  
**def** getB():  
 B = np.ones((10, 10))  
 **for** i **in** range(10):  
 **for** j **in** range(10):  
 B[i][j] = 1 / (i + j + 1)  
 **return** B  
  
  
**def** getC():  
 C = np.ones((12, 12))  
 **for** i **in** range(12):  
 **for** j **in** range(12):  
 k = max(i, j)  
 C[i][j] = float(12 - k)  
 **return** C  
  
  
**def** getD():  
 D = np.ones((20, 20))  
 **for** i **in** range(1, 21):  
 **for** j **in** range(1, 21):  
 D[i - 1][j - 1] = math.sqrt(2 / 21) \* math.sin(i \* j \* math.pi / 21)  
 **return** D  
  
  
**def** getE():  
 E = np.ones((50, 50))  
 **for** i **in** range(50):  
 **for** j **in** range(50):  
 **if** i == j **or** j == 49:  
 E[i][j] = 1  
 **elif** i < j:  
 E[i][j] = 0  
 **else**:  
 E[i][j] = -1  
 **return** E

1. **MyFunction.py**

**import** numpy **as** np  
**import** math  
  
  
*# 对实验中需要用到的矩阵基本运算进行实现***def** dot(A, B):  
 *"""  
 矩阵乘法的实现  
  
 参数：参与乘法的矩阵A和矩阵B  
 返回值：乘法运算得到的矩阵C  
 异常：如果A的列数不等于B的行数，则抛出异常ArithmeticError  
 """* Arow = np.shape(A)[0]  
 Acol = np.shape(A)[1]  
 Brow = np.shape(B)[0]  
 Bcol = np.shape(B)[1]  
  
 **if** Acol != Brow: *# 如果A的列数不等于B的行数，则抛出异常* ex = ArithmeticError(**"不符合矩阵乘法规则"**)  
 **raise** ex  
 **else**:  
 C = np.zeros((Arow, Bcol))  
 **for** i **in** range(Arow):  
 **for** j **in** range(Bcol):  
 **for** k **in** range(Acol):  
 C[i][j] += A[i][k] \* B[k][j]  
 **return** C  
  
  
**def** norm(Arr):  
 *"""  
 对向量进行取模操作  
  
 参数：向量Arr  
 返回值：向量Arr的模长，范数为2  
 异常：如果Arr不是向量，则抛出异常ArithmeticError  
 """* len = 0  
 **if** np.shape(Arr)[0] == 1: *# 行向量* len = np.shape(Arr)[1]  
 **elif** np.shape(Arr)[1] == 1: *# 列向量* len = np.shape(Arr)[0]  
 **else**:  
 ex = ArithmeticError(**"参数不是一个向量"**)  
 **raise** ex  
  
 sum = 0  
 **for** i **in** range(len):  
 sum += Arr[i][0] \* Arr[i][0]  
 sum = math.sqrt(sum)  
 **return** sum  
  
  
**def** transpose(A):  
 *"""  
 对矩阵/向量进行转置  
  
 参数：转置前矩阵A  
 返回值：转置后矩阵A1  
 """* row = np.shape(A)[0]  
 col = np.shape(A)[1]  
 A1 = np.ones((col, row))  
 **for** i **in** range(row):  
 **for** j **in** range(col):  
 A1[j][i] = A[i][j]  
 **return** A1  
  
  
**def** LU\_decomposition(A):  
 *"""  
 使用直接三角分解法进行矩阵的LU分解  
  
 参数：分解的矩阵A  
 返回值：分解得到的矩阵L和矩阵U，其中L为单位下三角矩阵，U为上三角矩阵  
 """* n = np.shape(A)[0]  
 L = np.zeros((n, n))  
 U = np.zeros((n, n))  
 **for** i **in** range(n):  
 L[i][i] = 1  
  
 **if** i == 0: *# 求U的第一行和L的第一列* U[0][0] = A[0][0]  
 **for** j **in** range(1, n):  
 U[0][j] = A[0][j]  
 L[j][0] = A[j][0] / U[0][0]  
 **else**:  
 **for** j **in** range(i, n): *# 求矩阵U的元素* temp = 0  
 **for** k **in** range(0, i):  
 temp = temp + L[i][k] \* U[k][j]  
 U[i][j] = A[i][j] - temp  
 **for** j **in** range(i + 1, n): *# 求矩阵L的元素* temp = 0  
 **for** k **in** range(0, i):  
 temp = temp + L[j][k] \* U[k][i]  
 L[j][i] = (A[j][i] - temp) / U[i][i]  
 **return** L, U  
  
  
**def** solveLU(L, U, x):  
 *"""  
 使用分解得到的LU矩阵，对线性方程组 LUy = x 进行求解  
  
 参数：单位下三角矩阵L、上三角矩阵U、向量x  
 返回值：方程的解 y向量  
 """* v = solve\_L(L, x)  
 y = solve\_U(U, v)  
 **return** y  
  
  
**def** solve\_L(L, x):  
 *"""  
 子函数，求解 Lv = x  
 """* n = np.shape(x)[0]  
 v = np.ones((n, 1))  
 **for** i **in** range(n):  
 v[i][0] = x[i][0]  
 **for** j **in** range(i):  
 v[i][0] -= v[j][0] \* L[i][j]  
 **return** v  
  
  
**def** solve\_U(U, v):  
 *"""  
 子函数，求解 Uy = v  
 """* n = np.shape(v)[0]  
 y = np.ones((n, 1))  
 **for** i **in** range(n - 1, -1, -1):  
 y[i][0] = v[i][0]  
 **for** j **in** range(i + 1, n):  
 y[i][0] -= U[i][j] \* y[j][0]  
 y[i][0] /= U[i][i]  
 **return** y  
  
  
**def** printVector(v):  
 *"""  
 将向量v进行格式化输出，每一个元素保留4位小数  
 注意：如果向量中的元素是复数，则在原来的基础上输出虚部的值  
 """* print(**"["**, end=**""**)  
 **for** i **in** range(np.shape(v)[0]):  
 **if** i != 0:  
 print(**", "**, end=**""**)  
 **if** v[i].imag == 0:  
 print(**"%.4f"** % v[i], end=**""**)  
 **else**:  
 **if** v[i].imag < 0:  
 print(**"{0.real:.3f}{0.imag:.3f}j"**.format(v[i]), end=**""**)  
 **else**:  
 print(**"{0.real:.3f}+{0.imag:.3f}j"**.format(v[i]), end=**""**)  
  
  
 **if** i != 0 **and** i % 5 == 0:  
 print(**""**)  
 print(**"]"**)  
  
  
**def** sgn(a):  
 *"""  
 求解函数sgn(n)的值  
 """* **if** a > 0:  
 **return** 1  
 **elif** a == 0:  
 **return** 0  
 **else**:  
 **return** -1

1. **power.py**

**from** numpy **import** \*  
**import** numpy **as** np  
**import** time  
  
*# 引用自己定义的库***import** MyMatrix  
**import** MyFunction **as** mf  
  
  
**def** power(matrix, precision, maxIterNum):  
 *"""  
 使用幂法求解矩阵的最大特征值  
  
 参数：  
 matrix：要求解的矩阵  
 precision：精度，误差小于该精度时结束迭代  
 maxIterNum：最大迭代次数，如果迭代次数大于该值则结束迭代  
 """* iter = 0  
 x = np.ones((np.shape(matrix)[0], 1)) *# 初始化x向量的值全为1* m = 0  
 temp = mf.dot(matrix, x) - m \* x  
 delta = mf.norm(temp)  
 t0 = time.time()  
 **while** delta >= precision **and** iter < maxIterNum:  
 iter += 1  
 y = mf.dot(matrix, x)  
 m = max(abs(y))  
 x = y / m  
 temp = mf.dot(matrix, x) - m \* x  
 delta = mf.norm(temp)  
 t1 = time.time()  
 print(**"\*\*\*\*\*\*\*\*幂法求最大特征值\*\*\*\*\*\*\*\*"**)  
 print(**"主要特征值: %.4f"** % m)  
 print(**"特征向量: "**)  
 mf.printVector(x)  
 print(**"迭代次数: {}"**.format(iter))  
 print(**"误差: %.6f"** % mf.norm(temp))  
 print(**"时间: %.4f s"** % (t1 - t0))  
  
  
*# 反幂法求最小特征值***def** inversePower(matrix, minDelta, maxIterNum):  
 *"""  
 使用反幂法求解矩阵的最小特征值  
  
 参数：  
 matrix：要求解的矩阵  
 precision：精度，小于该精度时结束迭代  
 maxIterNum：最大迭代次数，如果迭代次数大于该值则结束迭代  
 """* iter = 0  
 x = np.ones((np.shape(matrix)[0], 1)) *# 初始化x向量的值全为1* m = 0  
 L, U = mf.LU\_decomposition(matrix) *# 对所求矩阵进行LU分解* temp = mf.dot(matrix, x) - m \* x  
 delta = mf.norm(temp)  
 t0 = time.time()  
 **while** delta >= minDelta **and** iter < maxIterNum:  
 iter += 1  
 y = mf.solveLU(L, U, x) *# 使用LU矩阵进行线性方程组求解* m = max(abs(y))  
 x = y / m  
 temp = mf.dot(matrix, x) - (1 / m) \* x  
 delta = mf.norm(temp)  
 t1 = time.time()  
 m = 1 / m  
 print(**"\*\*\*\*\*\*\*\*反幂法求最小特征值\*\*\*\*\*\*\*\*"**)  
 print(**"最小特征值: %.4f"** % m)  
 print(**"特征向量: "**)  
 mf.printVector(x / x[0])  
 print(**"迭代次数: {}"**.format(iter))  
 temp = mf.dot(matrix, x) - m \* x  
 print(**"误差: %.6f"** % (mf.norm(temp)))  
 print(**"时间: %.4f s"** % (t1 - t0))  
  
  
*# 对五个矩阵分别运行算法，并展示结果***if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 A = MyMatrix.getA()  
 B = MyMatrix.getB()  
 C = MyMatrix.getC()  
 D = MyMatrix.getD()  
 E = MyMatrix.getE()  
  
 *# 对实验中的五个矩阵进行求解* matlist = [A, B, C, D, E]  
 matname = [**'A'**, **'B'**, **'C'**, **'D'**, **'E'**]  
 **for** i **in** range(5):  
 print(**"=======矩阵"**, matname[i], **"======="**)  
 a, b = np.linalg.eig(matlist[i])  
 power(matlist[i], 1e-4, 1000)  
 print(**"numpy函数求得的最大特征值: {: .4f}"**.format(max(abs(a))))  
 inversePower(matlist[i], 1e-6, 1000)  
 print(**"numpy函数求得的最小特征值: {: .4f}\n\n"**.format(min(abs(a))))

1. **jacobi.py**

**import** numpy **as** np  
**import** math  
**import** time  
  
*# 引用自己定义的库***import** MyFunction **as** mf  
**import** MyMatrix  
  
**def** maxElement(matrix):  
 *"""  
 子函数，计算矩阵非对角元素中的最大元素  
  
 参数：矩阵matrix  
 返回值：  
 maxv：最大的那个元素值  
 maxi，maxj：最大元素所在的位置坐标  
 """* maxv = -1  
 maxi = 0  
 maxj = 0  
 **for** i **in** range(np.shape(matrix)[0]):  
 **for** j **in** range(i + 1, np.shape(matrix)[1]):  
 **if** abs(matrix[i][j]) > maxv:  
 maxv = abs(matrix[i][j])  
 maxi = i  
 maxj = j  
 **return** maxv, maxi, maxj  
  
  
**def** rotateMatrix(A, theta, p, q):  
 *"""  
 子函数，对矩阵A的一个平面进行旋转  
  
 参数：  
 A：将要旋转的矩阵  
 theta：旋转角度  
 p,q：进行平面旋转变换的坐标系{p, q}  
  
 """  
 # 预先计算出theta的三角函数值，提高效率* cosT = math.cos(theta)  
 sinT = math.sin(theta)  
 size = np.shape(A)[0]  
 *# 旋转矩阵计算* rotateMat = np.eye(size)  
 rotateMat[p][p] = cosT  
 rotateMat[p][q] = -1 \* sinT  
 rotateMat[q][p] = sinT  
 rotateMat[q][q] = cosT  
 *# 对矩阵A进行旋转变换* B = mf.dot(mf.dot(np.transpose(rotateMat), A), rotateMat)  
 **return** B  
  
  
**def** jacobi(matrix, precision, maxIterNum):  
 *"""  
 使用Jacobi法求解矩阵的所有特征值  
  
 参数：  
 matrix：要求解的矩阵  
 precision：精度，误差小于该精度时结束迭代  
 maxIterNum：最大迭代次数，如果迭代次数大于该值则结束迭代  
 """* iter = 0  
 t0 = time.time()  
 len = np.shape(matrix)[0]  
 **while True**:  
 *# 找矩阵的非对角元素中的最大元素A[p][q]* (maxa, p, q) = maxElement(matrix)  
  
 *# 如果此时满足精度要求，则结束迭代* **if** maxa < precision **or** iter >= maxIterNum:  
 **break** *# 得到旋转角度theta* **if** matrix[p][p] == matrix[q][q]:  
 theta = math.pi / 4  
 **else**:  
 tg = -2 \* matrix[p][q] / (matrix[q][q] - matrix[p][p])  
 theta = math.atan(tg) / 2  
  
 *# 对矩阵matrix进行旋转* matrix = rotateMatrix(matrix, theta, p, q)  
 iter += 1  
 t1 = time.time()  
 res = [matrix[i][i] **for** i **in** range(len)]  
 res.sort(reverse=**True**)  
 print(**"特征值: "**.format(res))  
 mf.printVector(res)  
 print(**"迭代次数: {}"**.format(iter))  
 print(**"时间: %.4f s"** % (t1 - t0))  
  
  
*# 对四个矩阵分别运行算法，并展示结果***if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 A = MyMatrix.getA()  
 B = MyMatrix.getB()  
 C = MyMatrix.getC()  
 D = MyMatrix.getD()  
  
 matlist = [A, B, C, D]  
 matname = [**'A'**, **'B'**, **'C'**, **'D'**]  
 **for** i **in** range(len(matlist)):  
 print(**"=======矩阵"**, matname[i], **"======="**)  
 a, b = np.linalg.eig(matlist[i])  
 jacobi(matlist[i], 1e-4, 200)  
 print(**"使用numpy自带函数求得的特征值: "**)  
 mf.printVector(a)

1. **QR.py**

**from** numpy **import** \*  
**import** numpy **as** np  
**import** math  
**import** time  
**import** copy  
  
*# 引用自己定义的库***import** MyMatrix  
**import** MyFunction **as** mf  
  
  
**def** household(a):  
 *"""  
 求出向量a对应的Household矩阵  
  
 参数：要求解的向量a  
 返回值：Household矩阵H  
 """  
 # 规范化* maxv = max(abs(a))  
 **if** maxv != 0:  
 x1 = a / maxv  
  
  
 *# 求sigma的值  
 # 这里在P255的基础上进行修改，适用于x1[0]为零的情况* **if** x1[0] == 0:  
 sigma = mf.norm(x1)  
 **else**:  
 sigma = np.sign(x1[0]) \* mf.norm(x1)  
  
 u = copy.deepcopy(x1)  
 u[0] += sigma  
 beta = sigma \* (sigma + x1[0])  
  
 H = np.eye(np.shape(a)[0])  
 H = H - (1 / beta) \* mf.dot(u, mf.transpose(u))  
 **return** H  
  
  
**def** Hessenberg(matrix):  
 *"""  
 用正交相似变换约化一般矩阵为上海森伯格矩阵  
  
 参数：要求解的矩阵matrix  
 返回值：相似变换得到的上海森伯格矩阵A  
 """* A = copy.deepcopy(matrix)  
 len = np.shape(A)[0]  
 **for** i **in** range(len - 1):  
 *# 找列向量* c = A[(i + 1):, i]  
 c = c.reshape((c.shape[0], 1))  
 **if not**(np.all(c == 0)): *# c向量为零向量时，跳过  
 # 求HouseHold矩阵* R = household(c)  
 U = np.eye(len)  
 U[(i + 1):, (i + 1):] = R  
 A = mf.dot(mf.dot(U, A), U)  
 **return** A  
  
  
**def** single\_QR\_decomposition(matrix, precision, maxIterNum):  
 *"""  
 使用QR算法求解矩阵的所有特征值（使用单步QR方法）  
  
 参数：  
 matrix：要求解的矩阵  
 precision：精度，误差小于该精度时结束迭代  
 maxIterNum：最大迭代次数，如果迭代次数大于该值则结束迭代  
 """* resList = []  
 t0 = time.time()  
 M = copy.deepcopy(matrix)  
 H = Hessenberg(M) *# 用正交相似变换约化矩阵M为上海森伯格矩阵* n = np.shape(H)[0]  
 **while** n >= 2: *# 每次循环求矩阵的某一个特征值* iter = 0  
 **while** iter < maxIterNum:  
 *# 如果此时满足精度要求，则结束迭代* **if** abs(H[n - 1][n - 2]) < precision:  
 **break** *# 每一次迭代中，求新的H，并将原来的H覆盖* Plist = []  
 s = H[n - 1][n - 1]  
 H[0][0] = H[0][0] - s  
 **for** k **in** range(n - 1):  
 H[k + 1][k + 1] = H[k + 1][k + 1] - s  
 *# 确定所有的旋转矩阵P(i,i+1)* rk = math.sqrt(H[k][k] \* H[k][k] + H[k + 1][k] \* H[k + 1][k])  
 cosk = H[k][k] / rk  
 sink = H[k + 1][k] / rk  
 Plist.append((sink, cosk))  
 *# 左变换计算* **for** j **in** range(k, n):  
 tmp1 = H[k][j] \* cosk + H[k + 1][j] \* sink  
 tmp2 = H[k + 1][j] \* cosk - H[k][j] \* sink  
 H[k][j] = tmp1  
 H[k + 1][j] = tmp2  
  
 **for** k **in** range(n - 1):  
 sink = Plist[k][0]  
 cosk = Plist[k][1]  
 *# 右变换计算* **for** i **in** range(k + 2):  
 tmp1 = H[i][k] \* cosk + H[i][k + 1] \* sink  
 tmp2 = H[i][k + 1] \* cosk - H[i][k] \* sink  
 H[i][k] = tmp1  
 H[i][k + 1] = tmp2  
 H[k][k] = H[k][k] + s  
  
 H[n - 1][n - 1] = H[n - 1][n - 1] + s  
 iter += 1  
  
 *# 得到第n个特征值* **if** n == 2:  
 *# 如果此时矩阵为2\*2，则直接将对角线的值直接取出，作为特征值* resList.append(H[0][0])  
 resList.append((H[1][1]))  
 **else**:  
 resList.append(H[n - 1][n - 1])  
  
 *# 对H进行降阶* H = H[:n - 1, :n - 1]  
 n = np.shape(H)[0]  
 t1 = time.time()  
 print(**"特征值: "**)  
 mf.printVector(resList)  
 print(**"运行时间: %.4f s"** % (t1 - t0))  
  
  
*# 对五个矩阵分别运行算法，并展示结果***if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 A = MyMatrix.getA()  
 B = MyMatrix.getB()  
 C = MyMatrix.getC()  
 D = MyMatrix.getD()  
 E = MyMatrix.getE()  
  
 matlist = [A, B, C, D, E]  
 matname = [**'A'**, **'B'**, **'C'**, **'D'**, **'E'**]  
 **for** i **in** range(5):  
 print(**"=======矩阵"**, matname[i], **"======="**)  
 a, b = np.linalg.eig(matlist[i])  
 single\_QR\_decomposition(matlist[i], 1e-6, 50)  
 print(**"使用numpy自带函数求得的特征值: "**)  
 mf.printVector(a)

1. **doubleQR.py**

**from** numpy **import** \*  
**import** numpy **as** np  
**import** math  
**import** time  
**import** copy  
  
*# 引用自己定义的库***import** MyMatrix  
**import** MyFunction **as** mf  
**from** QR **import** household, Hessenberg  
  
  
**def** get\_root(b, c):  
 *"""  
 求一元二次方程x^2+bx+c=0的根(包括复根)  
  
 参数：方程的两个系数b，c  
 返回值：方程的两个根x1, x2（可能为复数）  
 """* delta = b \* b - 4 \* c  
 **if** delta >= 0:  
 x1 = -1 \* b / 2 + math.sqrt(delta) / 2  
 x2 = -1 \* b / 2 - math.sqrt(delta) / 2  
 **return** x1, x2  
 **else**:  
 x1 = complex(-1 \* b / 2, math.sqrt(-delta) / 2)  
 x2 = complex(-1 \* b / 2, -math.sqrt(-delta) / 2)  
 **return** x1, x2  
  
  
**def** double\_QR\_decomposition(matrix, precision, maxIterNum):  
 *"""  
 使用QR算法求解矩阵的所有复数特征值（使用双步QR方法）  
 参考教材P274中的算法和文献[41]的内容  
  
 参数：  
 matrix：要求解的矩阵  
 precision：精度，误差小于该精度时结束迭代  
 maxIterNum：最大迭代次数，如果迭代次数大于该值则结束迭代  
 """* resList = []  
 t0 = time.time()  
 A = copy.deepcopy(matrix)  
 H = Hessenberg(A) *# 用正交相似变换约化矩阵M为上海森伯格矩阵* **for** iter **in** range(maxIterNum):  
 n = np.shape(H)[0]  
 iter += 1  
 **if** n == 1: *# 如果矩阵为1\*1，则里面的元素就是特征值* resList.append(H[0][0])  
 **break  
 elif** n == 2: *# 如果矩阵为2\*2，则直接通过解一元二次方程求特征值* b = -1 \* (H[n - 1][n - 1] + H[n - 2][n - 2])  
 c = H[n - 1][n - 1] \* H[n - 2][n - 2] - H[n - 1][n - 2] \* H[n - 2][n - 1]  
  
 x1, x2 = get\_root(b, c)  
 resList.append(x1)  
 resList.append(x2)  
 **break  
 elif** abs(H[n-1][n-2]) < precision: *# 满足精度要求，提取右下角元素作为特征值* resList.append(H[n-1][n-1])  
 H = H[:n - 1, :n - 1] *# 将矩阵降一阶* **continue  
 else**:  
 *# 求出s, t (s和t的求法见教材)* s = H[n - 1][n - 1] + H[n - 2][n - 2]  
 t = H[n - 1][n - 1] \* H[n - 2][n - 2] - H[n - 1][n - 2] \* H[n - 2][n - 1]  
 **if** abs(H[n-2][n-3]) < precision: *# 提取右下角的2\*2矩阵，求该矩阵的特征值* x1, x2 = get\_root(s, t)  
 resList.append(x1)  
 resList.append(x2)  
 H = H[:n - 2, :n - 2] *# 将矩阵降两阶* **continue  
 else**: *# 使用双步QR算法进行迭代，每次从Hk迭代到Hk+2，参考教材P274中的算法  
 # 计算矩阵M的第一列(x,y,z,0,...,0)T  
 # 对于H1, H2, M矩阵为(H1-s1I)(H2-s2I)* x0 = H[0][0] \* H[0][0] + H[0][1] \* H[1][0] - s \* H[0][0] + t  
 y0 = H[1][0] \* (H[0][0] + H[1][1] - s)  
 z0 = H[1][0] \* H[2][1]  
 xyz = np.zeros((3, 1))  
 xyz[0][0] = x0  
 xyz[1][0] = y0  
 xyz[2][0] = z0  
 *# 求出xyz向量对应的初等反射矩阵R0, 使用Household变换* R0 = household(xyz)  
 *# 求出初等反射矩阵P0* P0 = np.eye(n)  
 P0[:3, :3] = R0  
  
 *# 对P0\*H\*P0进行正交相似变换* tmp = mf.dot(mf.dot(P0, H), P0)  
 H = Hessenberg(tmp)  
  
 **if** iter == maxIterNum:  
 print(**"未能求出特征值"**)  
 mf.printVector(resList)  
 **else**:  
 t1 = time.time()  
 print(**"特征值: "**)  
 mf.printVector(resList)  
 print(**"运行时间: %.4f s"** % (t1 - t0))  
  
  
**def** poly\_solve(p):  
 *"""  
 求多项式方程 x^n + p1\*x^(n-1) + p2\*x^(n-2)+...+pn=0 的所有根（包括复数根）  
  
 参数：方程系数组成的向量(p1,p2,...,pn)T  
 """* num = np.shape(p)[0]  
 pMatrix = np.zeros((num, num)) *# 将方程求解转化为矩阵特征值问题* **for** i **in** range(1, num):  
 pMatrix[i][i-1] = 1  
 **for** i **in** range(num):  
 pMatrix[0][i] = -p[i]  
 *# 调用双步QR算法，对矩阵特征值进行求解* double\_QR\_decomposition(pMatrix, 1e-6, 100)  
  
*# 对五个矩阵分别运行算法，并展示结果***if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 A = MyMatrix.getA()  
 B = MyMatrix.getB()  
 C = MyMatrix.getC()  
 D = MyMatrix.getD()  
 E = MyMatrix.getE()  
  
 matlist = [A, B, C, D, E]  
 matname = [**'A'**, **'B'**, **'C'**, **'D'**, **'E'**]  
 **for** i **in** range(5):  
 print(**"=======矩阵"**, matname[i], **"======="**)  
 a, b = np.linalg.eig(matlist[i])  
 double\_QR\_decomposition(matlist[i], 1e-6, 50)  
 print(**"使用numpy自带函数求得的特征值: "**)  
 mf.printVector(a)

1. **polySolve.py**

**from** numpy **import** \*  
**import** numpy **as** np  
**import** math  
**import** time  
**import** copy  
  
*# 引用自己定义的库***import** MyMatrix  
**import** doubleQR **as** dqr  
  
**def** poly\_solve(p):  
 *"""  
 求多项式方程 x^n + p1\*x^(n-1) + p2\*x^(n-2)+...+pn=0 的所有根（包括复数根）  
  
 参数：方程系数组成的向量(p1,p2,...,pn)T  
 """* num = np.shape(p)[0]  
 pMatrix = np.zeros((num, num))  
 **for** i **in** range(1, num):  
 pMatrix[i][i-1] = 1  
 **for** i **in** range(num):  
 pMatrix[0][i] = -p[i]  
  
 *# 调用QR算法，对矩阵特征值进行求解* dqr.double\_QR\_decomposition(pMatrix, 1e-6, 100)  
  
  
*# 对题中的方程进行求解*p = np.ones(11)  
poly\_solve(p)