Chap5 矩阵特征值计算 实验报告

李晨昊 2017011466

运行代码方法

环境要求：rust，gnuplot

运行代码

```
cargo run
```

q5.1

任务

实现幂法，求两个给定矩阵的模最大特征值λ_1和对应的特征向量x_1，要求$|(\lambda_1)_{k+1} - (\lambda_1)_k| < 10^{-5}$

解题思路

直接按课本描述的算法编写代码即可，每次更新λ时检查是否符合收敛标准，符合则退出。

初值方面，我选择了使用随机数生成器生成了一组值在$[0, 1]$间的浮点数作为特征向量的初值。

实验结果

输出如下

```
# 矩阵A
eigen = 12.254319296083574 vec = [0.6740200248549705, -1.0, 0.8895592930136067]
# 矩阵B
eigen = 98.52169779603172 vec = [-0.6039723423073206, 1.0, -0.2511351306025556, 0.1489534456311232]
```

心得体会

幂法实现简单，很适合平时编程时用于求特征值。

q5.3

任务

实现基本的QR算法，用它尝试求解一个矩阵的所有特征值，观察矩阵序列收敛的情况。

解题思路

基本的QR算法的核心即是QR分解算法，按照课本上的解法实现之即可。课本上给出的算法并没有直接给出Q矩阵，而是给出了一系列用于计算Q矩阵的向量，每个向量都对应一个Householder变换H，但是此时并不计算Hx，而是要计算RQ(反正我没想到怎么利用计算Hx的方法来计算RQ)，所以我暴力地把Q矩阵给算出来了。代码写的非常naive，时间复杂度达到$O(n^4)$，但是解决这种小规模的问题足够了。

实验结果

矩阵序列不会收敛，这里只给出前几个矩阵

```
iter 1:
[0.5000000000000002, 0.5, 0.5000000000000002, 0.5000000000000002]
[0.49999999999999994, 0.5000000000000004, -0.49999999999999994, -0.5]
[0.5000000000000001, -0.5, 0.4999999999999998, -0.4999999999999998]
[0.5000000000000001, -0.5, -0.4999999999999998, 0.4999999999999998]
```

```
iter 2:
[0.5000000000000007, 0.49999999999999967, -0.5000000000000001, -0.4999999999999995]
[0.49999999999999994, 0.5, 0.5000000000000003, 0.4999999999999999]
[-0.5000000000000004, 0.5000000000000001, 0.4999999999999993, -0.5000000000000001]
[-0.5, 0.4999999999999998, -0.4999999999999998, 0.4999999999999999]

iter 3:
[0.5000000000000004, 0.49999999999999983, 0.5000000000000009, 0.49999999999999956]
[0.4999999999999991, 0.500000000000001, -0.5000000000000008, -0.49999999999999983]
[0.5000000000000004, -0.5000000000000008, 0.49999999999999883, -0.5000000000000001]
[0.49999999999999967, -0.5000000000000001, -0.4999999999999997, 0.5000000000000002]

iter 4:
[0.500000000000001, 0.49999999999999883, -0.5000000000000007, -0.4999999999999989]
[0.4999999999999992, 0.5000000000000014, 0.5000000000000007, 0.4999999999999996]
[-0.5000000000000011, 0.5000000000000007, 0.4999999999999977, -0.5000000000000003]
[-0.49999999999999967, 0.49999999999999933, -0.5000000000000001, 0.5000000000000003]

iter 5:
[0.5000000000000011, 0.4999999999999991, 0.5000000000000022, 0.4999999999999996]
[0.49999999999999817, 0.5000000000000024, -0.5000000000000007, -0.49999999999999883]
[0.500000000000001, -0.5000000000000006, 0.4999999999999966, -0.500000000000001]
[0.49999999999999906, -0.4999999999999989, -0.5000000000000006, 0.5000000000000002]

iter 6:
[0.5000000000000024, 0.4999999999999978, -0.5000000000000013, -0.4999999999999977]
[0.49999999999999867, 0.5000000000000018, 0.5000000000000018, 0.4999999999999986]
[-0.5000000000000019, 0.5000000000000013, 0.49999999999999506, -0.5000000000000013]
[-0.49999999999999906, 0.49999999999999867, -0.500000000000001, 0.5000000000000006]

iter 7:
[0.5000000000000019, 0.49999999999999845, 0.5000000000000023, 0.4999999999999986]
[0.49999999999999734, 0.5000000000000027, -0.5000000000000012, -0.49999999999999784]
[0.5000000000000016, -0.5000000000000016, 0.49999999999999445, -0.5000000000000018]
[0.49999999999999806, -0.49999999999999833, -0.500000000000001, 0.5000000000000008]

iter 8:
[0.500000000000003, 0.49999999999999706, -0.5000000000000017, -0.4999999999999967]
[0.49999999999999806, 0.5000000000000022, 0.5000000000000027, 0.49999999999999795]
[-0.5000000000000022, 0.500000000000002, 0.4999999999999932, -0.5000000000000017]
[-0.49999999999999817, 0.4999999999999982, -0.5000000000000016, 0.5000000000000013]

iter 9:
[0.5000000000000022, 0.49999999999999756, 0.5000000000000032, 0.499999999999998]
[0.4999999999999964, 0.500000000000004, -0.5000000000000021, -0.4999999999999974]
[0.5000000000000018, -0.5000000000000022, 0.4999999999999924, -0.5000000000000023]
[0.4999999999999971, -0.49999999999999784, -0.5000000000000016, 0.5000000000000014]

iter 10:
[0.5000000000000032, 0.49999999999999556, -0.5000000000000023, -0.49999999999999567]
[0.4999999999999971, 0.5000000000000037, 0.5000000000000029, 0.49999999999999684]
[-0.5000000000000028, 0.5000000000000024, 0.49999999999999084, -0.5000000000000028]
[-0.49999999999999745, 0.4999999999999973, -0.5000000000000021, 0.5000000000000018]
```

简单的计算可以得出，矩阵A的QR分解为

$$\begin{bmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \\ 0.5 & -0.5 & 0.5 & -0.5 \\ 0.5 & -0.5 & -0.5 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \\ 0.5 & -0.5 & 0.5 & -0.5 \\ 0.5 & -0.5 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

这就是说事实上$A = RQ$，因此这个迭代方法会一直停留在A，不会收敛。

心得体会

基本的QR分解的确是一个很不可靠的方法...

q5.4

任务

实现带原点位移的QR分解算法，重复q5.3的任务

解题思路

按课本描述的算法编写代码即可，但是为了简单起见我没有实现Givens旋转，直接使用了q5.3实现的基于Householder变换的QR分解算法。

课本上循环的判断条件为`a[k][k - 1] != 0`，对于浮点数这一般是不太现实的，因此我把它改成了与设定的误差限eps做比较(选择$eps = 10^{-10}$)。

由于我的矩阵的实现方法的限制，我不能在不改变矩阵的元素布局的情况下直接改变矩阵的大小(我采用了连续式的存储，而非嵌套数组)，所以在带原点位移的QR分解算法中需要修改矩阵大小时我必须把已经计算出来的特征值保存下来。

实验结果

输出如下

```
iter 1
[0.5, 0.5, 0.5, 0.5]
[0.5, 0.5, -0.5, -0.5]
[0.5, -0.5, 0.5, -0.5]
[0.5, -0.5, -0.5, 0.5]

iter 2
[-0.4999999999999999, 0.6708203932499368, -0.4391550328268399, -0.32732683535398843]
[0.6708203932499369, 0.7000000000000002, 0.19639610121239298, 0.14638501094227988]
[-0.4391550328268399, 0.19639610121239298, 0.8714285714285714, -0.09583148474999117]
[-0.32732683535398854, 0.14638501094227988, -0.09583148474999105, 0.9285714285714286]

iter 3
[-0.9990859232175497, -0.03490023456327143, 0.02015375403100566, -0.014251942128061468]
[-0.03490023456327115, 0.9993907083440362, 0.0003518461787176509, -0.000248811778160515]
[0.020153754031005697, 0.0003518461787176955, 0.9997968202382774, 0.0001436807356690415]
[-0.01425194212806158, -0.0002488117781605237, 0.0001436807356691943, 0.9998983946352368]

iter 4
[-0.9999999999976378, 0.000001774061544793 8938, -0.000001024254910645991]
[0.000001774061545094 3925, 0.9999999999984265, 0.000000000000908432338742 3652]
[-0.000001024254910614 2702, 0.000000000000908473299420 9947, 0.9999999999994754]

iter 5
[-0.9999999999976378, 0.000001774061544793 8938]
[0.000001774061545094 3925, 0.9999999999984265]


iter 6
[-0.9999999999992119]

# eigens
[0.9999999999997378, 0.9999999999994754, 1.0000000000000002, -0.9999999999992119]
```

可见矩阵序列正确地收敛到了由特征值构成的对角阵，最终也正确的求出了特征值。

心得体会

相比于基本的QR算法，带原点位移的QR分解算法收敛速度更快，收敛的可能性也更大，综合来讲增加的这一点复杂性时值得的。