CENG 371

Scientific Computing

Homework 3

BARTU KILIÇKAYA / 2380640

Question 1)

a)

starting vector = [1,1,1,1,1] (can be changed)

1000 iterations: (can be changed)

Maximum eigenvalue: 3.7320508075688776 (corresponds to 2 + sqrt(3))

Eigenvector: [0.5 , -0.8660254 ,1, -0.8660254 , 0.5]

```
C:\Users\lebro\Desktop\THE3\venv\Scripts\python.exe C:/Users/lebro/Desktop/THE3/main.py
Eigenvalue:  3.7320508075688776
Eigenvector:  [ 0.5       -0.8660254  1.        -0.8660254  0.5      ]

Process finished with exit code 0
```

b)

starting vector = [1,1,1,1,1] (can be changed)

1000 iterations: (can be changed)

alfa = 2.2 (can be changed but I selected this in order to find eigenvalue = 2)

closest eigenvalue = 2

corresponding eigenvector: Eigenvector: [1, 0, -1, 0, 1]

```
C:\Users\lebro\Desktop\THE3\venv\Scripts\python.exe C:/Users/lebro/Desktop/THE3/main.py
Eigenvalue:  2.0
Eigenvector:  [ 1. -0. -1.  0.  1.]

Process finished with exit code 0
```

Since the closest eigenvalue to 2.2 is 2, we get 2 as a resulting eigenvalue and this corresponding eigenvector.

c)

For the largest -in magnitude- eigenvalue, we can use power method that we implemented in 1-a.

Using power method:

Maximum eigenvalue: 3.7320508075688776 (corresponds to 2 + sqrt(3))

Eigenvector: [0.5 , -0.8660254 ,1, -0.8660254 , 0.5]

For the smallest -in magnitude- eigenvalue, we can use inverse power method. We have already implemented shifted inverse power method in 1-b. Giving alfa = 0, we have inverse power method:

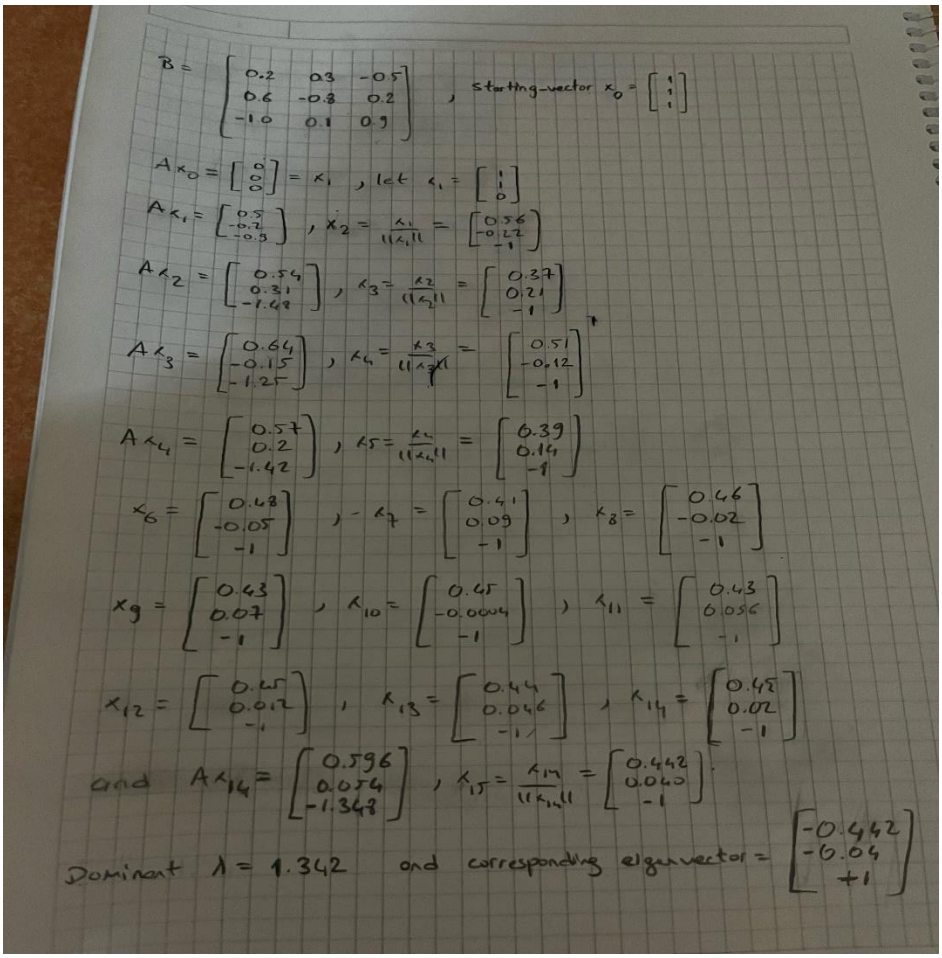
Smallest eigenvalue: 0.26794919243112275 (corresponds to 2- sqrt(3))

Eigenvector: Eigenvector: [0.5, 0.8660254, 1, 0.8660254, 0.5]

```
C:\Users\lebro\Desktop\THE3\venv\Scripts\python.exe C:/Users/lebro/Desktop/THE3/main.py
Eigenvalue:  0.26794919243112275
Eigenvector:  [0.5       0.8660254 1.        0.8660254 0.5      ]

Process finished with exit code 0
```

d)




```
Eigenvalue for 1-d:  1.3426860441876562
Eigenvector for 1-d:  [-0.44583638 -0.03150337  1.        ]

Process finished with exit code 0
```

Results are pretty close but not exactly the same, since I have only done 15 iterations, that is probably why eigenvector is slightly different. Python result is the correct one. Additionally in paper, I got [0,0,0] vector after first iteration so I

had to pick arbitrary vector after this, however in computer this is not the case, because we get small rounding errors that makes the algorithm still working.

Question 2)

a)

Since matrix A is symmetric, eigenvectors are orthogonal in other means their dot product is equal to zero, so this method can be used for this type of matrices. But if the matrix was not symmetric, we wouldn't be able to find other eigenvalues besides the largest one.

b)

starting vector = [1,1,1,1,1] (can be changed)

1000 iterations: (can be changed)

K = 2 (can be changed)

Largest 2 eigenvalues: [3.7320508075688776, 3.0]

Corresponding eigenvectors: [[0.5 , -0.8660254, 1. , -0.8660254, 0.5], [-1., 1., -0., -1., 1.]]

```
Eigenvalues in power_k with k = 2:  [3.7320508075688776, 3.0]
Eigenvectors in power_k with k = 2:  [array([ 0.5      , -0.8660254,  1.       , -0.8660254,  0.5      ]), array([-1.,  1., -0., -1.,  1.])]
```

c)

starting matrix (nxk);n=5,k=2 = [1,1], [1,1], [1,1], [1,1], [1,1] (can be changed)

1000 iterations: (can be changed)

K = 2 (can be changed)

Largest 2 eigenvalues: [3.7320508075688776, 3.0]

Corresponding eigenvectors: [[0.5 , -0.8660254, 1. , -0.8660254, 0.5], [-1., 1., -0., -1., 1.]]

```
Eigenvalues in subspace with k = 2:  [3.7320508075688776, 3.0]
Eigenvectors in subspace with k = 2:  [array([ 0.5      , -0.8660254,  1.       , -0.8660254,  0.5      ]), array([-1.,  1., -0., -1.,  1.])]

Process finished with exit code 0
```

d)

For large values in k , i.e k = 20; (results are in seconds)

```
Time elapsed in power_k with can_229:  2.646218776702881
Time elapsed in subspace iteration with can_229:  9.183182716369629

Process finished with exit code 0
```

Power method runs faster than subspace iteration. I think it is expected since in subspace iteration we have additionally QR factorization which is O(n^3) instead of power_k's substraction. That's probably why power method runs faster for large k values.