# Lab 7 Report

Amlan Nayak MS18197

19th March 2021

## 1 Draw Gerschgorin Circle and get the bounds for

$$A = \begin{bmatrix} 4 & 1 & 1 \\ 0 & 2 & 1 \\ -2 & 0 & 9 \end{bmatrix} B = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 2 & 1 \\ 2 & 2 & 3 \end{bmatrix}$$

### 1.1 Algorithm & Discussion

For easy of plotting, I decided to use matplotlib module and save the figures. From class notes, we know roughs bounds of the eigenvalues in form of:

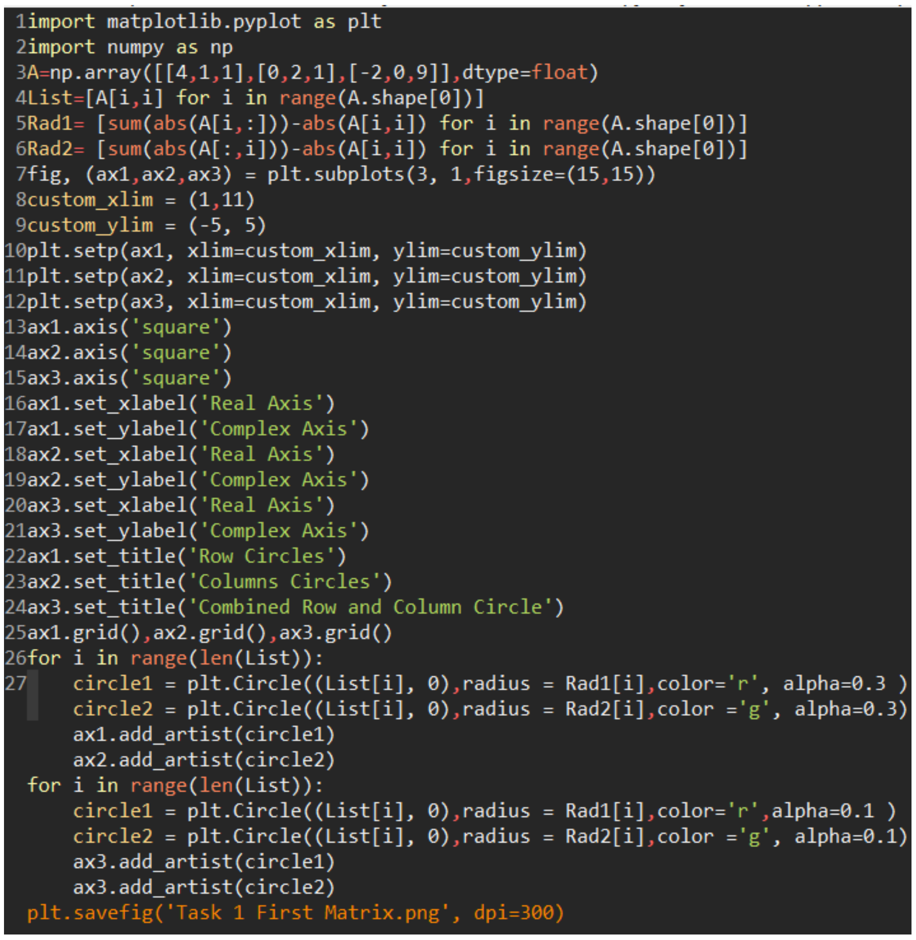
Max absolute row sum $|\lambda_i| \leq max_i[\sum_{j=1}^{n} A_{ij}]$

Max absolute column sum $|\lambda_i| \leq max_j[\sum_{i=1}^{n} A_{ij}]$

The bounds of the eigen values can be represented by Gerschgorin Circles as:

$$|\lambda_i - A_{kk}| \leq \sum_{j=1, j\neq k}^{n} A_{kj}$$

### 1.2 Code and Output



```python
import matplotlib.pyplot as plt
import numpy as np
A=np.array([[4,1,1],[0,2,1],[-2,0,9]],dtype=float)
List=[A[i,i] for i in range(A.shape[0])]
Rad1= [sum(abs(A[i,:]))-abs(A[i,i]) for i in range(A.shape[0])]
Rad2= [sum(abs(A[:,i]))-abs(A[i,i]) for i in range(A.shape[0])]
fig, (ax1,ax2,ax3) = plt.subplots(3, 1,figsize=(15,15))
custom_xlim = (1,11)
custom_ylim = (-5, 5)
plt.setp(ax1, xlim=custom_xlim, ylim=custom_ylim)
plt.setp(ax2, xlim=custom_xlim, ylim=custom_ylim)
plt.setp(ax3, xlim=custom_xlim, ylim=custom_ylim)
ax1.axis('square')
ax2.axis('square')
ax3.axis('square')
ax1.set_xlabel('Real Axis')
ax1.set_ylabel('Complex Axis')
ax2.set_xlabel('Real Axis')
ax2.set_ylabel('Complex Axis')
ax3.set_xlabel('Real Axis')
ax3.set_ylabel('Complex Axis')
ax1.set_title('Row Circles')
ax2.set_title('Columns Circles')
ax3.set_title('Combined Row and Column Circle')
ax1.grid(),ax2.grid(),ax3.grid()
for i in range(len(List)):
    circle1 = plt.Circle((List[i], 0),radius = Rad1[i],color='r', alpha=0.3 )
    circle2 = plt.Circle((List[i], 0),radius = Rad2[i],color ='g', alpha=0.3)
    ax1.add_artist(circle1)
    ax2.add_artist(circle2)
for i in range(len(List)):
    circle1 = plt.Circle((List[i], 0),radius = Rad1[i],color='r',alpha=0.1 )
    circle2 = plt.Circle((List[i], 0),radius = Rad2[i],color ='g', alpha=0.1)
    ax3.add_artist(circle1)
    ax3.add_artist(circle2)
plt.savefig('Task 1 First Matrix.png', dpi=300)
```

Figure 1: Code for drawing and saving figures of Gerschgorin circles

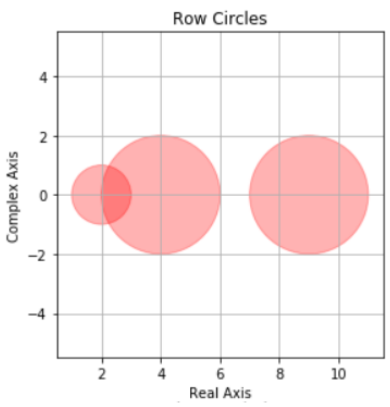


Figure 2: Gerschgorin circles for the rows of matrix A

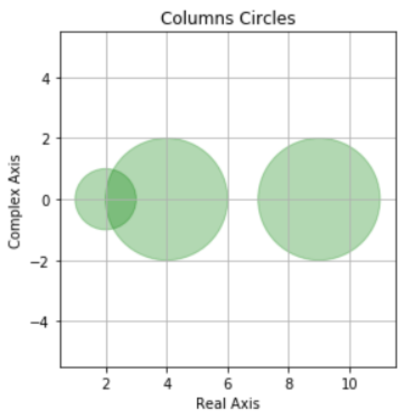


Figure 3: Gerschgorin circles for the columns of matrix A

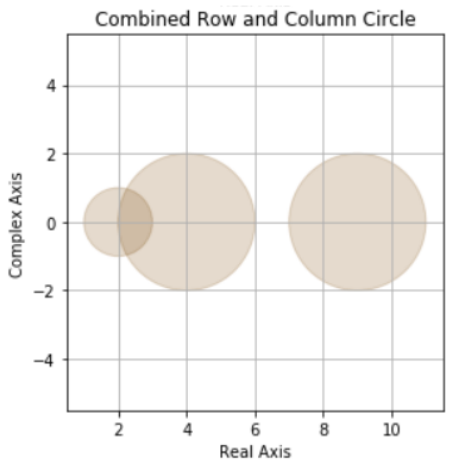


Figure 4: Combined row and column Gerschgorin circles for the matrix A

Since the two sets of circles overlap completely. The bounds for eigenvalues of matrix A on the real axis are: (1,6) and (5,11)

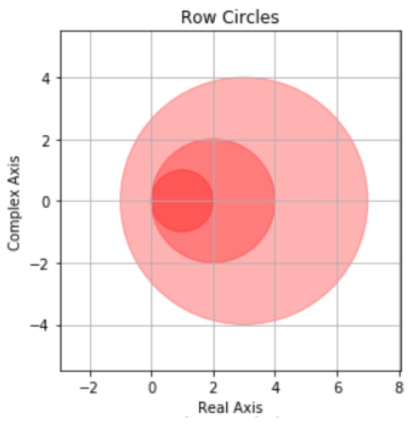


Figure 5: Gerschgorin circles for the rows of matrix B

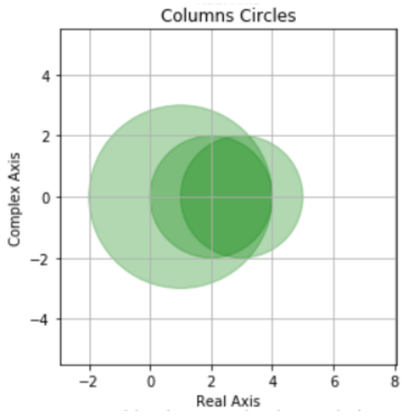


Figure 6: Gerschgorin circles for the columns of matrix B

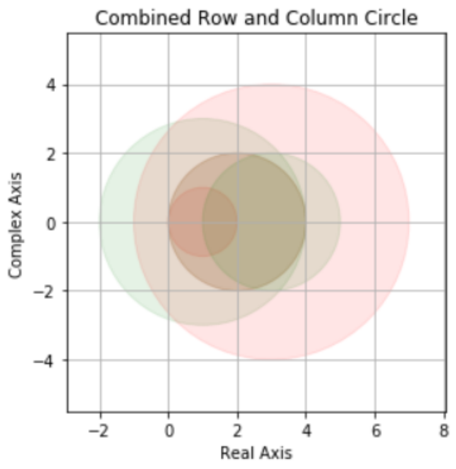


Figure 7: Combined row and column Gerschgorin circles for the matrix B

From the overlap of the two sets of circles, we can get the bounds. The bounds for eigenvalues of matrix B on the real axis are: (-1,5)

## 2 Find the largest eigenvalue and the corresponding eigenvector of the matrix

$$A = \begin{bmatrix} -2 & 0 & -1 \\ 1 & -1 & 1 \\ 2 & 2 & 0 \end{bmatrix}$$

### 2.1 Algorithm & Discussion

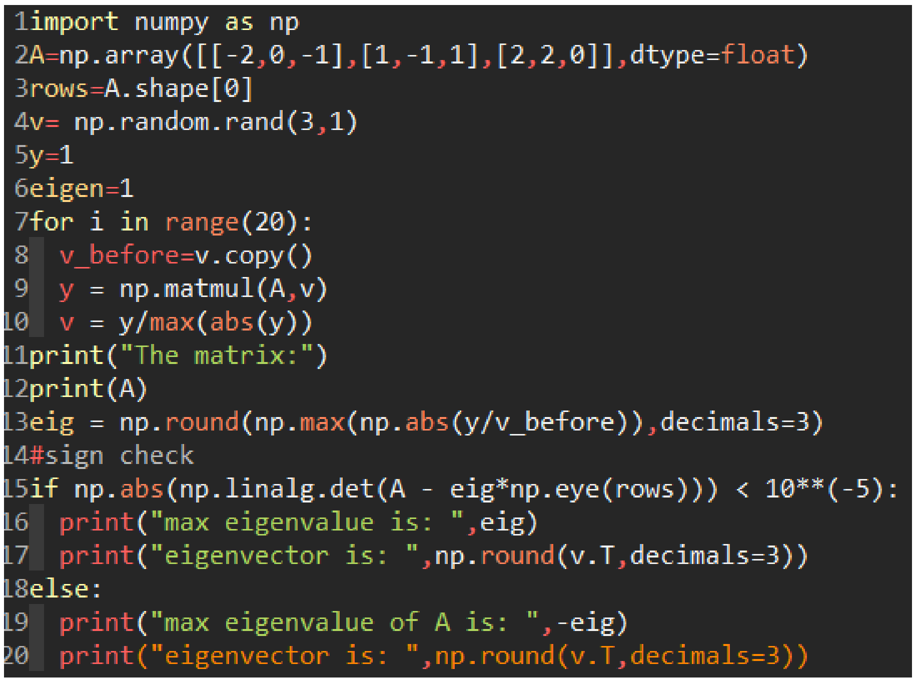
The method we use for finding the largest eigenvalue is called Power's method. In power's method, we use the following iteration relations:

$$y_{k+1} = Av_k$$
$$v_{k+1} = \frac{y_{k+1}}{max[y_{k+1}]}$$
$$k = 1, 2, 3...$$
$$\lambda_{max} = \lim_{k\to\infty} \frac{(y_{k+1})_r}{(v_k)_r}$$

where $v_0$ is our initial guess from which we determine $v_1$ and the rest $v_k$. We continue this process till successive $\lambda$ values converge. I chose 20 iteration as it ensured convergence. We choose the absolute maximum value in $\frac{(y_{k+1})_r}{(v_k)_r}$. This is magnitude of our largest eigenvalue.

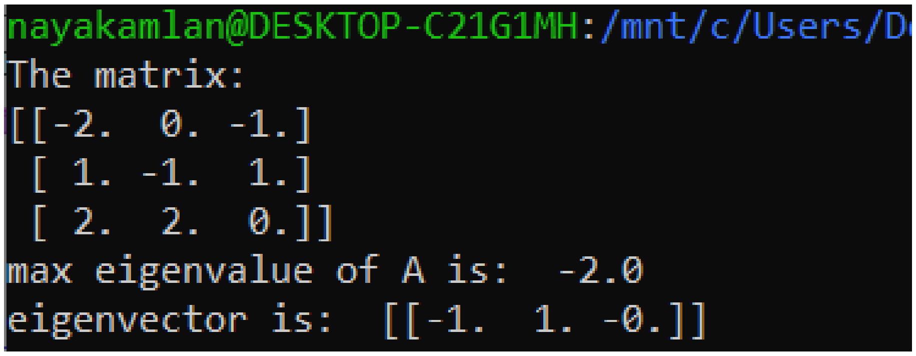
To find the sign of the eigenvalue, we check the determinant of $A - \lambda I$. If it is non zero, we take the negative of the largest absolute eigenvalue. If it is zero, we just take the eigenvalue as it is.

### 2.2 Code and Output



```python
import numpy as np
A=np.array([[-2,0,-1],[1,-1,1],[2,2,0]],dtype=float)
rows=A.shape[0]
v= np.random.rand(3,1)
y=1
eigen=1
for i in range(20):
    v_before=v.copy()
    y = np.matmul(A,v)
    v = y/max(abs(y))
print("The matrix:")
print(A)
eig = np.round(np.max(np.abs(y/v_before)),decimals=3)
#sign check
if np.abs(np.linalg.det(A - eig*np.eye(rows))) < 10**(-5):
    print("max eigenvalue is: ",eig)
    print("eigenvector is: ",np.round(v.T,decimals=3))
else:
    print("max eigenvalue of A is: ",-eig)
    print("eigenvector is: ",np.round(v.T,decimals=3))
```

Figure 8: Code of Power's method for largest eigenvalue



```
nayakamlan@DESKTOP-C21G1MH:/mnt/c/Users/D
The matrix:
[[-2.  0. -1.]
 [ 1. -1.  1.]
 [ 2.  2.  0.]]
max eigenvalue of A is:  -2.0
eigenvector is:  [[-1.  1. -0.]]
```

Figure 9: Output of Power's method

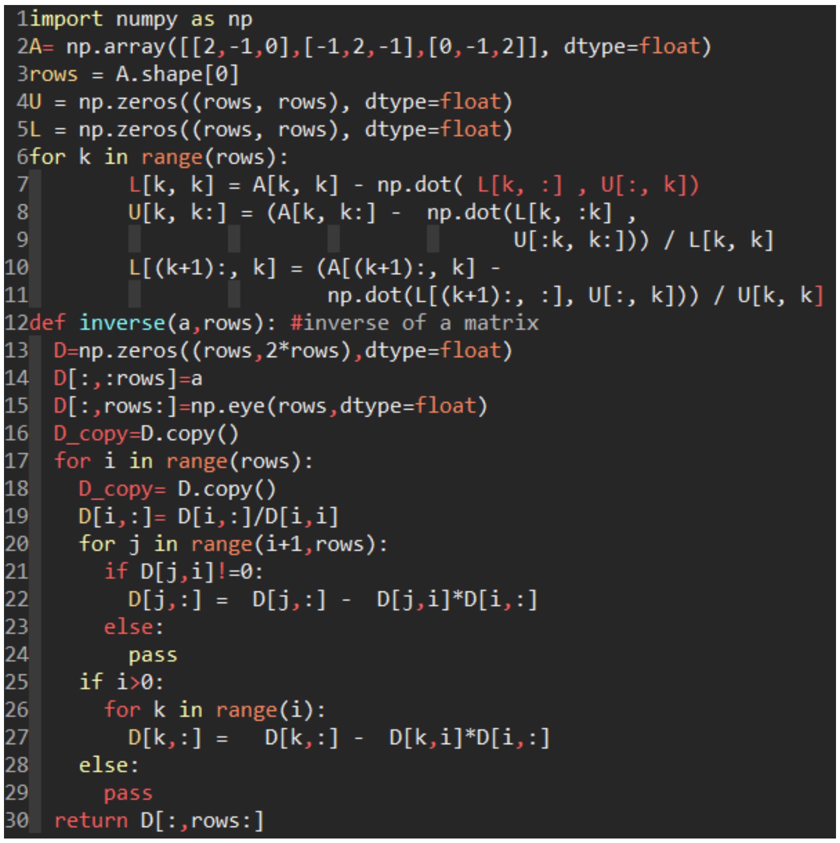
## 3 Find the smallest eigenvalue of the matrix

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

### 3.1 Algorithm & Discussion

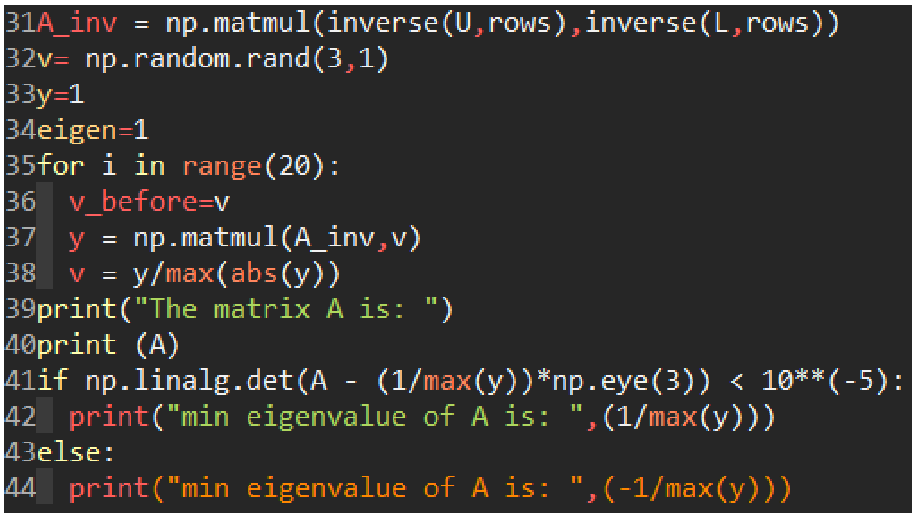
We use power's method as we had done in problem 2. However, we find the largest eigenvalue of $A^{-1}$ here. The reciprocal of the largest eigenvalue of $A^{-1}$ is the smallest eigenvalue of $A$.

### 3.2 Code and Output



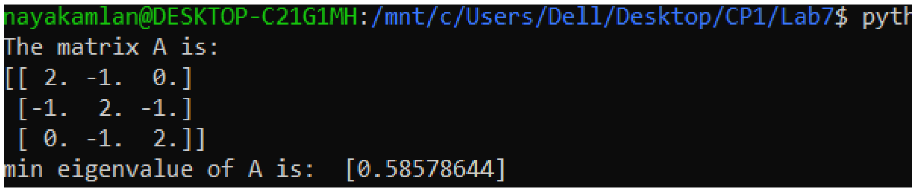
```
import numpy as np
A= np.array([[2,-1,0],[-1,2,-1],[0,-1,2]], dtype=float)
rows = A.shape[0]
U = np.zeros((rows, rows), dtype=float)
L = np.zeros((rows, rows), dtype=float)
for k in range(rows):
        L[k, k] = A[k, k] - np.dot( L[k, :] , U[:, k])
        U[k, k:] = (A[k, k:] -  np.dot(L[k, :k] ,
                                U[:k, k:])) / L[k, k]
        L[(k+1):, k] = (A[(k+1):, k] -
                        np.dot(L[(k+1):, :], U[:, k])) / U[k, k]
def inverse(a,rows): #inverse of a matrix
  D=np.zeros((rows,2*rows),dtype=float)
  D[:,:rows]=a
  D[:,rows:]=np.eye(rows,dtype=float)
  D_copy=D.copy()
  for i in range(rows):
    D_copy= D.copy()
    D[i,:]= D[i,:]/D[i,i]
    for j in range(i+1,rows):
      if D[j,i]!=0:
        D[j,:] =  D[j,:] -  D[j,i]*D[i,:]
      else:
        pass
    if i>0:
      for k in range(i):
        D[k,:] =   D[k,:] -  D[k,i]*D[i,:]
    else:
      pass
  return D[:,rows:]
```

Figure 10: Code for smallest eigenvalue of $A$ part 1



```
A_inv = np.matmul(inverse(U,rows),inverse(L,rows))
v= np.random.rand(3,1)
y=1
eigen=1
for i in range(20):
  v_before=v
  y = np.matmul(A_inv,v)
  v = y/max(abs(y))
print("The matrix A is: ")
print (A)
if np.linalg.det(A - (1/max(y))*np.eye(3)) < 10**(-5):
  print("min eigenvalue of A is: ",(1/max(y)))
else:
  print("min eigenvalue of A is: ",(-1/max(y)))
```

Figure 11: Code for smallest eigenvalue of $A$ part 2



```
nayakamlan@DESKTOP-C21G1MH:/mnt/c/Users/Dell/Desktop/CP1/Lab7$ pyth
The matrix A is:
[[ 2. -1.  0.]
 [-1.  2. -1.]
 [ 0. -1.  2.]]
min eigenvalue of A is:  [0.58578644]
```

Figure 12: Output

## 4 Use Given's method to get tri-diagonal system for

$$A = \begin{bmatrix} 15 & 4 & 3 & 2 & -1 \\ 4 & 25 & 6 & -7 & 8 \\ 3 & 6 & 27 & 8 & 9 \\ 2 & -7 & 8 & 319 & 10 \\ -1 & 8 & 9 & 10 & 100 \end{bmatrix}$$

### 4.1 Algorithm & Discussion

A symmetric matrix can be reduced into a tri-diagonal system using Given's method. We use loops to determine orthogonal rotation matrices $B$. The $\theta$ for the orthogonal matrices are determined from the row we are iterating over and the entry along the diagonal which is just above the diagonal of the square matrix.

For example, in the very first iteration, we get $\theta = tan^{-1}(\frac{A_{13}}{A_{12}})$. We then determine B as:

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 & 0 \\ 0 & \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

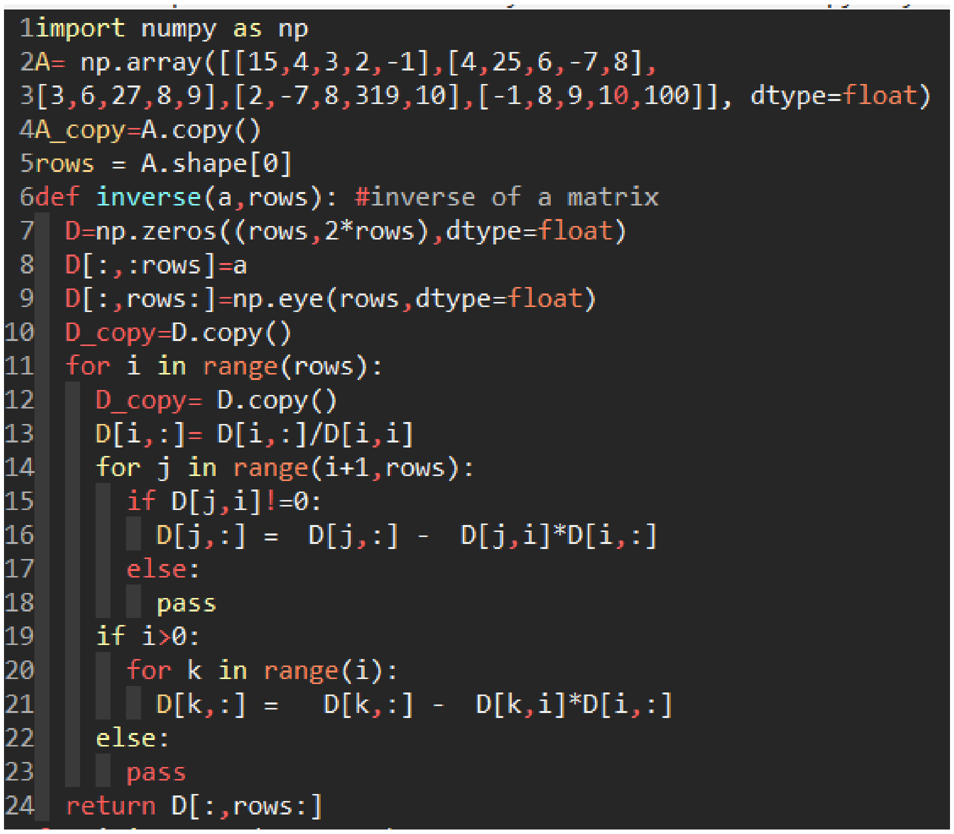
We update the value of $A$ as $B^{-1}AB$. We then determine a new $\theta = tan^{-1}(\frac{A_{14}}{A_{12}})$ where values $A_{14}\&A_{12}$ are from the updated A. We then determine a new $B$,

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

We do this for each entry along the row of the element of the diagonal just above the diagonal of the square matrix that we chose earlier. We repeat this for all entries along that diagonal.

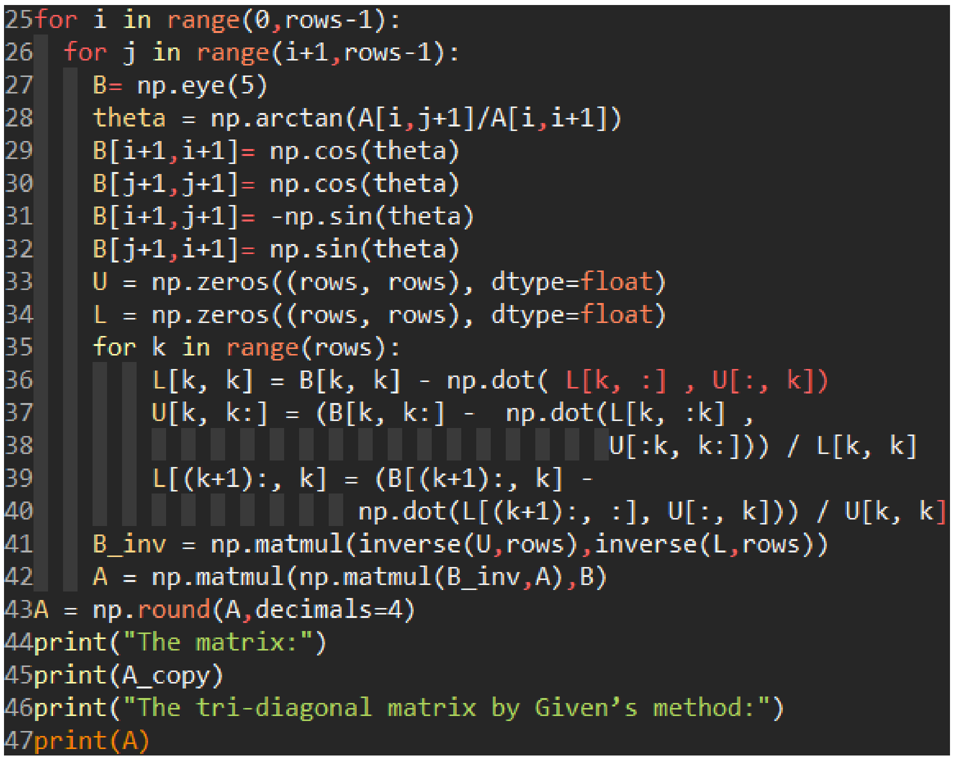
We are finally left with a tri-diagonal matrix at the end.

## 4.2 Code and Output



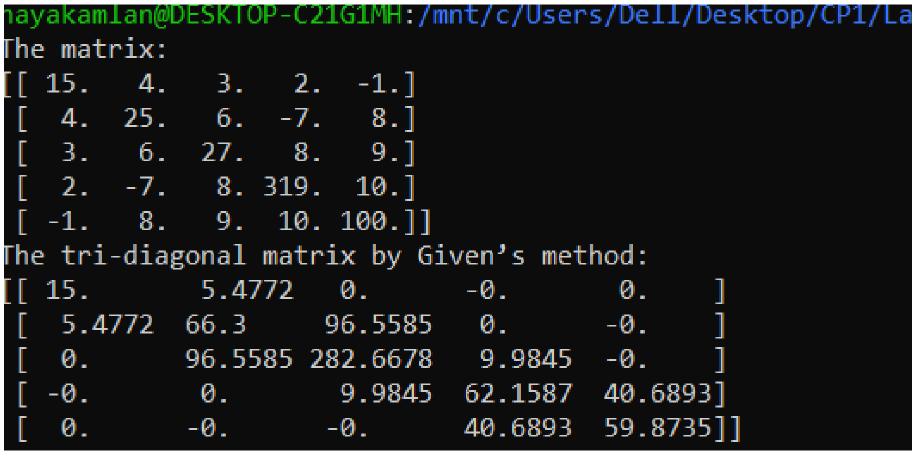
```
import numpy as np
A= np.array([[15,4,3,2,-1],[4,25,6,-7,8],
[3,6,27,8,9],[2,-7,8,319,10],[-1,8,9,10,100]], dtype=float)
A_copy=A.copy()
rows = A.shape[0]
def inverse(a,rows): #inverse of a matrix
    D=np.zeros((rows,2*rows),dtype=float)
    D[:,:rows]=a
    D[:,rows:]=np.eye(rows,dtype=float)
    D_copy=D.copy()
    for i in range(rows):
        D_copy= D.copy()
        D[i,:]= D[i,:]/D[i,i]
        for j in range(i+1,rows):
            if D[j,i]!=0:
                D[j,:] =  D[j,:] -  D[j,i]*D[i,:]
            else:
                pass
        if i>0:
            for k in range(i):
                D[k,:] =   D[k,:] -  D[k,i]*D[i,:]
        else:
            pass
    return D[:,rows:]
```

Figure 13: Code for Given's method part 1



```
for i in range(0,rows-1):
    for j in range(i+1,rows-1):
        B= np.eye(5)
        theta = np.arctan(A[i,j+1]/A[i,i+1])
        B[i+1,i+1]= np.cos(theta)
        B[j+1,j+1]= np.cos(theta)
        B[i+1,j+1]= -np.sin(theta)
        B[j+1,i+1]= np.sin(theta)
        U = np.zeros((rows, rows), dtype=float)
        L = np.zeros((rows, rows), dtype=float)
        for k in range(rows):
            L[k, k] = B[k, k] - np.dot( L[k, :] , U[:, k])
            U[k, k:] = (B[k, k:] -  np.dot(L[k, :k] ,
                                        U[:k, k:])) / L[k, k]
            L[(k+1):, k] = (B[(k+1):, k] -
                            np.dot(L[(k+1):, :], U[:, k])) / U[k, k]
        B_inv = np.matmul(inverse(U,rows),inverse(L,rows))
        A = np.matmul(np.matmul(B_inv,A),B)
A = np.round(A,decimals=4)
print("The matrix:")
print(A_copy)
print("The tri-diagonal matrix by Given's method:")
print(A)
```

Figure 14: Code for Given's method part 2



```
nayakamlan@DESKTOP-C21G1MH:/mnt/c/Users/Dell/Desktop/CP1/La
The matrix:
[[ 15.   4.   3.   2.  -1.]
 [  4.  25.   6.  -7.   8.]
 [  3.   6.  27.   8.   9.]
 [  2.  -7.   8. 319.  10.]
 [ -1.   8.   9.  10. 100.]]
The tri-diagonal matrix by Given's method:
[[ 15.       5.4772   0.      -0.       0.    ]
 [  5.4772  66.3     96.5585   0.      -0.    ]
 [  0.      96.5585 282.6678   9.9845  -0.    ]
 [ -0.       0.       9.9845  62.1587  40.6893]
 [  0.      -0.      -0.      40.6893  59.8735]]
```

Figure 15: Output

## 5 Get the eigenvalues of the tri-diagonal setup achieved from 4 using Sturm Sequence

### 5.1 Algorithm & Discussion

In class lectures, we had determined sturm sequences for a tri-diagonal matrix of the form

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ c_1 & b_2 & c_2 & 0 & 0 \\ 0 & c_2 & b_3 & c_3 & 0 \\ 0 & 0 & c_4 & b_4 & c_5 \\ 0 & 0 & 0 & c_5 & b_5 \end{bmatrix}$$

The sturm sequences were of the form:

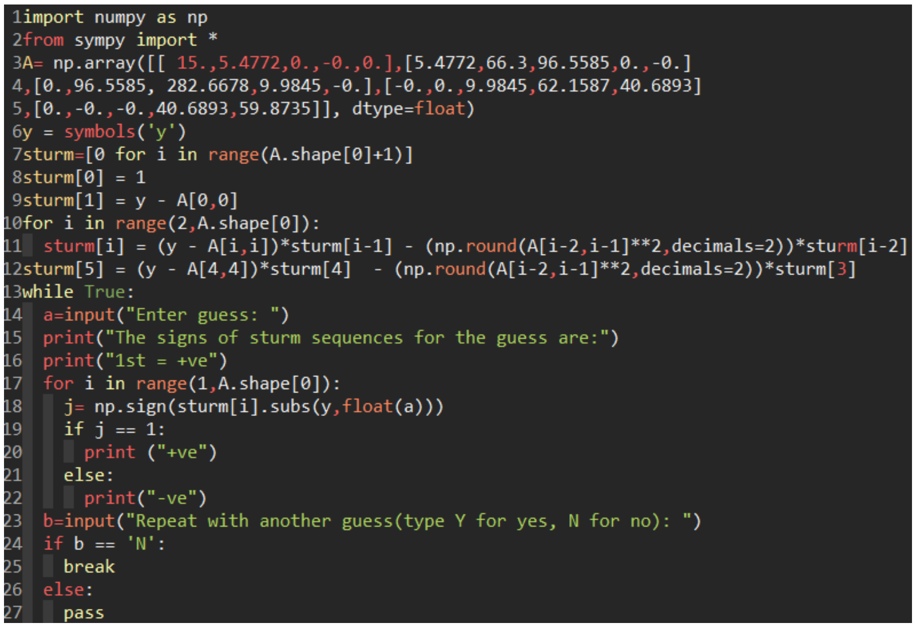
$$f_0(\lambda) = 1$$

$$f_1(\lambda) = 1 - b_1$$
$$f_k(\lambda) = (\lambda - b_k)f_{k-1} - c_{k-1}^2 f_{k-2}$$
$$k = 2, 3, 4....n$$

We then check where the signs of the sturm sequences flip and locate the eigenvalue. We find the difference in the changes of sign for different values of $\lambda$. We can find an interval which gives us the approximate value of the eigenvalue.

The sturm sequences were determined by sympy module in python and guess were checked.

## 5.2 Code & Results



```python
import numpy as np
from sympy import *
A= np.array([[ 15.,5.4772,0.,-0.,0.],[5.4772,66.3,96.5585,0.,-0.]
,[0.,96.5585, 282.6678,9.9845,-0.],[-0.,0.,9.9845,62.1587,40.6893]
,[0.,-0.,-0.,40.6893,59.8735]], dtype=float)
y = symbols('y')
sturm=[0 for i in range(A.shape[0]+1)]
sturm[0] = 1
sturm[1] = y - A[0,0]
for i in range(2,A.shape[0]):
  sturm[i] = (y - A[i,i])*sturm[i-1] - (np.round(A[i-2,i-1]**2,decimals=2))*sturm[i-2]
sturm[5] = (y - A[4,4])*sturm[4]  - (np.round(A[i-2,i-1]**2,decimals=2))*sturm[3]
while True:
  a=input("Enter guess: ")
  print("The signs of sturm sequences for the guess are:")
  print("1st = +ve")
  for i in range(1,A.shape[0]):
    j= np.sign(sturm[i].subs(y,float(a)))
    if j == 1:
      print ("+ve")
    else:
      print("-ve")
  b=input("Repeat with another guess(type Y for yes, N for no): ")
  if b == 'N':
    break
  else:
    pass
```

Figure 16: Code in emacs

From various guesses, the approximate values of eigenvalues are 319, 101, 31, 13 and 19.