CP1_sda18001

November 3, 2024

Computer Project 1

ECE5101

Spencer Albano

`CP1_sda18001.ipynb` *can also be viewed at*

https://nbviewer.org/github/salbano108/ECE5101_CP1/blob/main/CP1_sda18001.ipynb/

or by installing Jupyter Lab

1 Problem 1

For each matrix below find:

(i) The dominant eigenvalue
(ii) A corresponding eigenvector
(iii) The power used to obtain them.

$$A_1 = \begin{bmatrix} 0.2 & 1.2 & 1.1 & 0.9 & 0.1 & 0 & 0.2 \\ 0.7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.82 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.97 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.87 & 0.2 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0.25 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 & 0 \\ 0.25 & 0.25 & 0 & 0.25 & 0.25 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 3 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 3 & 1 \end{bmatrix}$$

1.1 Solution:

[130]:
```
import numpy as np

def power_method(A, max_iterations=15, precision=1e-3):
    n, m = A.shape #Find row,col of A
    x = np.ones(n) # Set the initial vector to a vector of ones
    eigenvalue_temp = 0

    for k in range(1, max_iterations + 1): #test from 1 to max_int to find␣
 ↪Eigenvalue that fits the selected value of precision

        consant0 = x[0] # Store first component of previous vector
        x = np.dot(A, x) # Multiply by matrix A to get the next vector
        consant1 = x[0] # Store first component of current vector


        x = x / x[0] # Divide each vector by their first component to scale

        eigenvalue = consant1/consant0 # Approximate the eigenvalue from (k+1)/
 ↪(k) vector results

        # Check if eigenvalue is still changing for selected level of precision
        if abs(eigenvalue - eigenvalue_temp) < precision:
            eigenvalue = np.round(eigenvalue, 3)
            eigenvector = np.round(x, 3)
            return eigenvalue, eigenvector, k

        eigenvalue_temp = eigenvalue # Update the temporary eigenvalue for the␣
 ↪next iteration
```

1.1.1 Example Matrix

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

[131]:
```
A = np.array([[1, 2, 0], [2, 1, 2], [1, 2, 3]])
eigenvalue, eigenvector, k = power_method(A)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", eigenvalue)
print("Eigenvector:", eigenvector)
print("K:", k)
```

```
Dominant Eigenvalue: 4.82
Eigenvector: [1.000 1.910 2.648]
K: 9
```

$$\lambda_1 v_1 \approx 4.82 \begin{bmatrix} 1.000 \\ 1.910 \\ 2.648 \end{bmatrix} \; for \; k = 9$$

1.1.2 Matrix 1

$$A_1 = \begin{bmatrix} 0.2 & 1.2 & 1.1 & 0.9 & 0.1 & 0 & 0.2 \\ 0.7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.82 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.97 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.87 & 0.2 \end{bmatrix}$$

[132]:
```
A1 = np.array([[0.2, 1.2, 1.1, 0.9, 0.1, 0, 0.2], [0.7, 0, 0, 0, 0, 0, 0], [0,␣
↪0.82, 0, 0, 0, 0, 0], [0, 0, 0.97, 0, 0, 0, 0],[0, 0, 0, 0.97, 0, 0, 0], [0,␣
↪0, 0, 0, 0.9, 0, 0], [0, 0, 0, 0, 0, 0.87, 0.2]])
eigenvalue, eigenvector, k = power_method(A1)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", eigenvalue)
print("Eigenvector:", eigenvector)
print("K:", k)
```

```
Dominant Eigenvalue: 1.372
Eigenvector: [1.000 0.510 0.305 0.213 0.156 0.098 0.074]
K: 9
```

$$\lambda_1 v_1 \approx 1.372 \begin{bmatrix} 1.000 \\ 0.510 \\ 0.305 \\ 0.213 \\ 0.156 \\ 0.098 \\ 0.074 \end{bmatrix} \; for \; k = 9$$

1.1.3 Matrix 2

$$A_2 = \begin{bmatrix} 0.25 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 & 0 \\ 0.25 & 0.25 & 0 & 0.25 & 0.25 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

[133]:
```
A2 = np.array([[0.25, 0, 0.5, 0, 0.5], [0, 0, 1, 0, 0], [0.25, 0.25, 0, 0.25, 0.
↪25], [0, 0, 0.5, 0, 0.5],[0, 0, 0, 0, 1]])
eigenvalue, eigenvector, k = power_method(A2)
```

```
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", eigenvalue)
print("Eigenvector:", eigenvector)
print("K:", k)
```

```
Dominant Eigenvalue: 1.009
Eigenvector: [1.000 0.802 0.807 0.752 0.702]
K: 7
```

$$\lambda_1 v_1 \approx 1.009 \begin{bmatrix} 1.000 \\ 0.802 \\ 0.807 \\ 0.752 \\ 0.702 \end{bmatrix} for\ k = 7$$

1.1.4 Matrix 3

$$A_3 = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 3 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 3 & 1 \end{bmatrix}$$

[134]:
```
A3 = np.array([[1, 2, 1, 3], [2, 1, 2, 3], [0, 1, 1, 1], [1, 1, 3, 1]])
eigenvalue, eigenvector, k = power_method(A3)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", eigenvalue)
print("Eigenvector:", eigenvector)
print("K:", k)
```

```
Dominant Eigenvalue: 5.592
Eigenvector: [1.000 1.058 0.382 0.698]
K: 8
```

$$\lambda_1 v_1 \approx 5.592 \begin{bmatrix} 1.000 \\ 1.058 \\ 0.382 \\ 0.698 \end{bmatrix} for\ k = 8$$

2 Problem 2

[135]:
```
def dominant_eigen(A):
    eigenvalues, eigenvectors = np.linalg.eig(A) #Built in python/numpy␣
 ↪eigenvalue and vector command

    # Find the index of the largest eigenvalue
    largest_eigen_index = np.argmax(np.abs(eigenvalues))
```

```
    # Return the dominant eigenvalue and its corresponding eigenvector
    dominant_eigenvalue = eigenvalues[largest_eigen_index]
    dominant_eigenvector = eigenvectors[:, largest_eigen_index]

    if dominant_eigenvector[0] != 0: #avoid divide by 0
        dominant_eigenvector /= dominant_eigenvector[0] #scale first element to␣
↪1.000

    return dominant_eigenvalue.real, dominant_eigenvector.real #added .real due␣
↪to strange complex number glitch (should be floating point only)
```

2.0.1 Matrix 1

$$A_1 = \begin{bmatrix} 0.2 & 1.2 & 1.1 & 0.9 & 0.1 & 0 & 0.2 \\ 0.7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.82 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.97 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.87 & 0.2 \end{bmatrix}$$

[136]:
```
A1 = np.array([[0.2, 1.2, 1.1, 0.9, 0.1, 0, 0.2], [0.7, 0, 0, 0, 0, 0, 0], [0,␣
↪0.82, 0, 0, 0, 0, 0], [0, 0, 0.97, 0, 0, 0, 0],[0, 0, 0, 0.97, 0, 0, 0], [0,␣
↪0, 0, 0, 0.9, 0, 0], [0, 0, 0, 0, 0, 0.87, 0.2]])
eigenvalue, eigenvector = dominant_eigen(A1)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", np.round(eigenvalue, 3))
print("Eigenvector:", np.round(eigenvector,3))
```

```
Dominant Eigenvalue: 1.372
Eigenvector: [1.000 0.510 0.305 0.216 0.152 0.100 0.074]
```

To compare the actual vs calculated:

$$1.372 \begin{bmatrix} 1.000 \\ 0.510 \\ 0.305 \\ 0.216 \\ 0.152 \\ 0.100 \\ 0.074 \end{bmatrix} = \lambda_1 v_1 \approx 1.372 \begin{bmatrix} 1.000 \\ 0.510 \\ 0.305 \\ 0.213 \\ 0.156 \\ 0.098 \\ 0.074 \end{bmatrix}$$

2.0.2 Matrix 2

$$A_2 = \begin{bmatrix} 0.25 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 & 0 \\ 0.25 & 0.25 & 0 & 0.25 & 0.25 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

[137]:
```
A2 = np.array([[0.25, 0, 0.5, 0, 0.5], [0, 0, 1, 0, 0], [0.25, 0.25, 0, 0.25, 0.
↪25], [0, 0, 0.5, 0, 0.5],[0, 0, 0, 0, 1]])
eigenvalue, eigenvector = dominant_eigen(A2)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", np.round(eigenvalue, 3))
print("Eigenvector:", np.round(eigenvector,3))
```

```
Dominant Eigenvalue: 1.0
Eigenvector: [1.000 0.812 0.812 0.750 0.688]
```

To compare the actual vs calculated:

$$1.000 \begin{bmatrix} 1.000 \\ 0.812 \\ 0.812 \\ 0.750 \\ 0.688 \end{bmatrix} = \lambda_1 v_1 \approx 1.009 \begin{bmatrix} 1.000 \\ 0.802 \\ 0.807 \\ 0.752 \\ 0.702 \end{bmatrix}$$

2.0.3 Matrix 3

$$A_3 = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 3 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 3 & 1 \end{bmatrix}$$

[138]:
```
A3 = np.array([[1, 2, 1, 3], [2, 1, 2, 3], [0, 1, 1, 1], [1, 1, 3, 1]])
eigenvalue, eigenvector = dominant_eigen(A3)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", np.round(eigenvalue, 3))
print("Eigenvector:", np.round(eigenvector,3))
```

```
Dominant Eigenvalue: 5.592
Eigenvector: [1.000 1.058 0.382 0.698]
```

To compare the actual vs calculated:

$$5.592 \begin{bmatrix} 1.000 \\ 1.058 \\ 0.382 \\ 0.698 \end{bmatrix} = \lambda_1 v_1 \approx 5.592 \begin{bmatrix} 1.000 \\ 1.058 \\ 0.382 \\ 0.698 \end{bmatrix}$$

3 Problem 3

a)

[139]:
```
A4 = np.array([[-1, 1, 0], [0, 0, 1], [0, 0, 1]])
eigenvalue, eigenvector, k = power_method(A4)
np.set_printoptions(precision=3, floatmode='fixed')
print("Dominant Eigenvalue:", eigenvalue)
print("Eigenvector:", eigenvector)
print("K:", k)
```

```
Dominant Eigenvalue: 0.0
Eigenvector: [nan inf inf]
K: 1
```

```
C:\Users\salba\AppData\Local\Temp\ipykernel_9556\3918416573.py:15:
RuntimeWarning: divide by zero encountered in divide
  x = x / x[0] # Divide each vector by their first component to scale
C:\Users\salba\AppData\Local\Temp\ipykernel_9556\3918416573.py:15:
RuntimeWarning: invalid value encountered in divide
  x = x / x[0] # Divide each vector by their first component to scale
```

Trying to run this program results in a *Divide by Zero* error. This is due to the first elements in the vector *x[0]* equating to a zero after being multiplied by the provide A matrix.

b)

Using the definition of eigenvectors:

$$Av = \lambda v$$

We can algebraic prove:

$$(A + cI)v = Av + cIv$$

$$(A + cI)v = \lambda v + cIv$$

$$(A + cI)v = \lambda v + cv.$$

$$(A + cI)v = (\lambda + c)v$$

c)

[140]:
```
def power_method_modified(A,c):
    rows, cols = A.shape #get row,col of A matrix
    I = np.eye(rows) #produce matching square Identity matrix
    A_augmented = A + (c*I) #augment matrix per rule proven in 3b
```

```
    eigenvalue, eigenvector, k = power_method(A_augmented) #call power method␣
 ↪function from problem 1 with augment matrix
    return (eigenvalue-c,eigenvector), (-eigenvalue+c,-eigenvector+c) #return␣
 ↪eigenvalue/vector pair
```

[141]:
```
A4 = np.array([[-1, 1, 0], [0, 0, 1], [0, 0, 1]]) #Define A4
eigen1, eigen2 = power_method_modified(A4,2)

np.set_printoptions(precision=3, floatmode='fixed')
print("First Eigenvalue:", eigen1[0])
print("First Eigenvector:", eigen1[1])
print("Second Eigenvalue:", eigen2[0])
print("Second Eigenvector:", eigen2[1])
```

```
First Eigenvalue: 1.0
First Eigenvector: [1.000 2.000 2.000]
Second Eigenvalue: -1.0
Second Eigenvector: [1.000 0.000 0.000]
```

$$\lambda_1 v_1 = 1.000 \begin{bmatrix} 2.000 \\ 2.000 \\ 2.000 \end{bmatrix}$$

$$\lambda_2 v_2 = -1.000 \begin{bmatrix} 1.000 \\ 0.000 \\ 0.000 \end{bmatrix}$$

[142]:
```
eigenvalues, eigenvectors = np.linalg.eig(A4) #Built in python/numpy eigenvalue␣
 ↪and vector command

sorted_indices = np.argsort(np.abs(eigenvalues))[::-1] #sort all eigenvalues
largest_eigen_index1, largest_eigen_index2 = sorted_indices[:2] #select two␣
 ↪largest eigenvalues (this will be the pair)

# Extract value based on index
eigenval1 = eigenvalues[largest_eigen_index1].real
eigenvect1 = eigenvectors[:, largest_eigen_index1].real
eigenval2 = eigenvalues[largest_eigen_index2].real
eigenvect2 = eigenvectors[:, largest_eigen_index2].real


if eigenvect1[0] != 0: #avoid divide by 0
    eigenvect1 /= eigenvect1[0] #scale element 1 to 1.000
if eigenvect2[0] != 0:
    eigenvect2 /= eigenvect2[0]
```

```
np.set_printoptions(precision=3, floatmode='fixed')
print("First Eigenvalue:", np.round(eigenval1, 3))
print("First Eigenvector:", np.round(eigenvect1, 3))
print("Second Eigenvalue:", np.round(eigenval2, 3))
print("Second Eigenvector:", np.round(eigenvect2, 3))
```

```
First Eigenvalue: 1.0
First Eigenvector: [1.000 2.000 2.000]
Second Eigenvalue: -1.0
Second Eigenvector: [1.000 0.000 0.000]
```

$$\lambda_1 v_1 = 1.000 \begin{bmatrix} 2.000 \\ 2.000 \\ 2.000 \end{bmatrix}$$

$$\lambda_2 v_2 = -1.000 \begin{bmatrix} 1.000 \\ 0.000 \\ 0.000 \end{bmatrix}$$

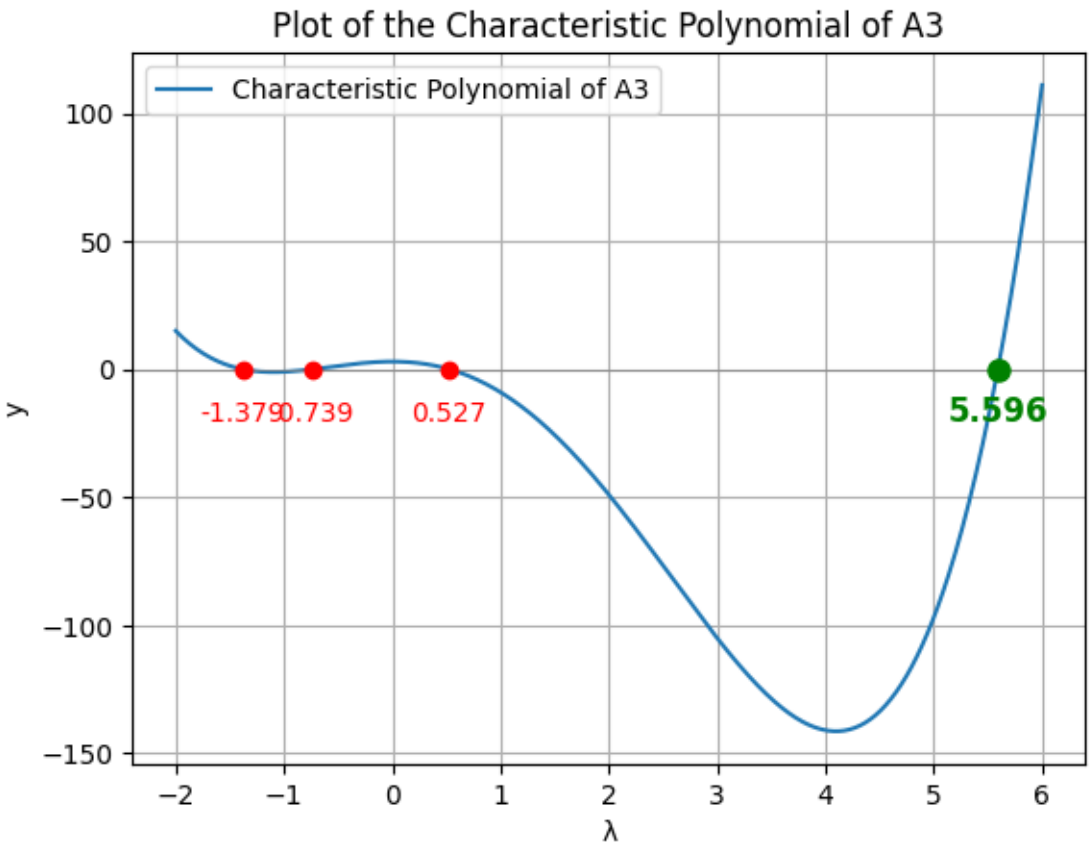
Results are identical using both methods

4 Problem 4

[143]:
```
import sympy as sp #used to display equation on output
import matplotlib.pyplot as plt #used for plotting

# Define the matrix A3, again
A3 = np.array([[1, 2, 1, 3], [2, 1, 2, 3], [0, 1, 1, 1], [1, 1, 3, 1]])

#characteristic polynomial
 = sp.symbols(' ')
A3_sym = sp.Matrix(A3)
char_poly = A3_sym.charpoly( )  # get the characteristic polynomial of A3

print("Characteristic Polynomial of A3:")
print(char_poly.as_expr())


eigenvalues = np.linalg.eigvals(A3) #find eigenvalue with built in function␣
 ↪again
coefficients = np.poly(eigenvalues)  #create numpy version of characteristic␣
 ↪polynomial (previous was for graphical display)

def char_poly_eval(x):
    return np.polyval(coefficients, x)

#plot
```

```
x_vals = np.linspace(-2, 6, 500)
y_vals = [char_poly_eval(x) for x in x_vals]

#plot formatting
plt.plot(x_vals, y_vals, label='Characteristic Polynomial of A3')
plt.axhline(0, color='gray', lw=0.5)
plt.xlabel(' ')
plt.ylabel('y')
plt.title('Plot of the Characteristic Polynomial of A3')
plt.legend()
plt.grid(True)

#find the roots (crossings where y=0)
dense_x_vals = np.linspace(-2, 6, 1000)
dense_y_vals = [char_poly_eval(x) for x in dense_x_vals]

sign_changes = np.where(np.diff(np.sign(dense_y_vals)))[0] #find sign changes␣
 ↪(ie change in direction of function)

#find midpoint between sign changes
crossings = []
for index in sign_changes:
    x_cross = (dense_x_vals[index] + dense_x_vals[index + 1]) / 2
    crossings.append(x_cross)

#label roots and mark max root in green
for roots in crossings:
    if roots == max(crossings):
        plt.plot(roots, 0, 'go', markersize=8)
        plt.text(roots, -20, f'{roots:.3f}', fontsize=12, ha='center',␣
 ↪color='green', fontweight='bold')
    else:
        plt.plot(roots, 0, 'ro')
        plt.text(roots, -20, f'{roots:.3f}', fontsize=10, ha='center',␣
 ↪color='red')


plt.show()
```

```
Characteristic Polynomial of A3:
 **4 - 4* **3 - 9* **2 + 3
```



$$Characteristic - Polynomial : \lambda^4 - 4\lambda^3 - 9\lambda^2 + 3$$

$$max(root) = 5.596$$