MATH 494 Final Project

Nada El Arabi

April 24, 2023

Abstract

In this paper, we discuss the implementation of Sliding-Window Dynamic Mode Decomposition on two videos with the purpose of analyzing its performance in identifying fast and slow movement. The first video is one where dynamics are present on a single time scale while the second features dynamics present on two different time scales. Finally, we compare the performance of the method to the performance of the standard DMD algorithm.

1 Introduction and Overview

Dynamic mode decomposition is a data-driven method born out of necessity to find more efficient technique to analyze non-linear fluid dynamics as the size of our data sets continue to grow tremendously [1].
The heart of the technique lies in finding a linear operator $\mathbf{A}$ (that we may refer to as the "DMD matrix") which may take us from one time point in a dynamical system to the next. DMD allows to bypass much more computationally intensive traditional analyzes.

Though DMD is certainly very impressive it encounters some difficulty when applied to dynamical systems which feature evolutions on multiple timescales. Different time scale dynamical systems are quite common in nature, one can think of the variation in size of the Arctic ice sheet on the scale of a year, changing with the seasons, but also on the scale of decades where Global Warming's toll can be clearly observed[4]. Another example is oceanic tides where drift happens both on a daily scale and on the scale of months or years[1]. There are many more such systems and unfortunately, DMD struggles inherently in yielding accurate results. More specifically, the method struggles to identify dynamics occurring on longer time periods. Furthermore, a large quantity of data may be necessary to capture the longer term dynamics rendering the method computationally infeasible[1].

As a potential solution to these limitations, we suggest implementing DMD on a Sliding-Window over the snapshots of data in time. It is this augmentation to the standard DMD approach that will be the main subject of study in this report. We will attempt to demonstrate the effectiveness of Sliding-Window DMD by applying it to two videos, one where motion (representing the dynamics of interest) occurs on a single time scale and a second which features movement on two different time scales. We begin by defining a window size and extract a corresponding number of windows k from our data. We then apply the standard DMD method to compute k DMD matrices: $\mathbf{A^{(k)}}$. We proceed to compute the eigenvalues and eigenvectors corresponding to each matrix and apply a clustering algorithm to determine exact separation points. Finally, we apply the standard DMD algorithm and compare its effectiveness in separating the different time scale dynamics.

2 Theoretical Background

Suppose we have data matrix $\mathbf{X} = \begin{bmatrix} X_1 & X_2 & \dots & X_n \end{bmatrix}$, where $\mathbf{X} \in \mathbb{C}^{\mathbb{M}\times\mathbb{N}}$ whose columns are snapshots of our data in evenly spaced points in time. A window size T is then defined (the choice of this particular T will be discussed in the coming sections. Given a value for T, the data matrix $\mathbf{X}$ can be divided into a number of windows, indexed by $k = 1, ..., N - T$:

$$\mathbf{X}^{(k)} = \begin{bmatrix} \mathbf{X}_k & \mathbf{X}_{k+1} & \dots & \mathbf{X}_{k+T} \end{bmatrix} \tag{1}$$

where $\mathbf{X}^k \in \mathbb{C}^{\mathbb{M}\times\mathbb{T}}$ and each value of k represents a window of snapshots in time[1]. For every k, we can then apply the standard DMD method so that we may find each corresponding DMD matrix $\mathbf{A}^{(k)}$.

Since the method of interest is built entirely on Dynamic Mode Decomposition, we now recollect its theoretical foundations.
We introduce the data matrices $\mathbf{X} = \begin{bmatrix} X_1 & X_2 & \dots & X_n \end{bmatrix}$ and $\mathbf{Y} = \begin{bmatrix} Y_1 & Y_2 & \dots & Y_n \end{bmatrix}$ where $\mathbf{X}, \mathbf{Y} \in \mathbb{C}^{\mathbb{M}\times\mathbb{N}}$ and each column of both matrices is once again a snapshot in time. We further assume that Y_n is a snapshot in time in the future of X_n, $\forall n \in N$, for some unit of time h. Our goal is to find the best linear operator $\mathbf{A} \in \mathbb{C}^{\mathbb{N}\times\mathbb{N}}$ which forecasts the system from one point in time to the next. We thus seek to minimize the square of the Frobenius norm of the difference between the exact next point in time and its estimation by the linear operator $\mathbf{A}$, at every time point.
There is an exact solution to that minimization problem and it is given by:

$$\mathbf{A} = \mathbf{Y}\mathbf{X}^\dagger \tag{2}$$

where $\mathbf{X}^\dagger$ is the Moore-Penrose inverse of the data matrix $\mathbf{X}$[1].

One of the most effective ways to compute $\mathbf{A}$ is through the SVD of $\mathbf{X}$, $\mathbf{X} = U\Sigma V^*$, where $U \in \mathbb{C}^{\mathbb{N}\times\mathbb{Q}}$, $V^* \in \mathbb{C}^{\mathbb{Q}\times\mathbb{M}}$ are unitary matrices and $\Sigma \in \mathbb{C}^{\mathbb{Q}\times\mathbb{Q}}$ is a diagonal matrix of positive, real singular values. We know that such a decomposition exists for every matrix, though it needs not be unique[1].
By equality 2 and the SVD decomposition of $\mathbf{X}$, we have:

$$\mathbf{A} = \mathbf{Y}\mathbf{X}^\dagger = \mathbf{Y}(U\Sigma V^*)^\dagger = \mathbf{Y}V\Sigma^{-1}U^* \tag{3}$$

In many fields where DMD presents itself as a choice data-driven method to help analyze large data sets, such as the study of fluid dynamics[5] or in digital image and video processing[3], it is not computationally feasible to calculate the exact DMD matrix $\mathbf{A}$. Instead, we aim to take advantage of a possible low-rank structure to our high-dimensional data by computing a reduced version of $\mathbf{A}$: $\tilde{A}$. This dimensionality reduction is achieved by finding an optimal truncation threshold for the singular values of the SVD decomposition of the data matrix. A number of approaches are possible, in this report we will refer to Gavish and Donoho's 2014 paper on optimal singular value truncation[2] and discuss the algorithm in a further section.
We suppose an optimal choice r for the dimension of the singular values matrix and write this reduced SVD as:

$$\mathbf{X} = U_r\Sigma_r V_r^* \tag{4}$$

We can find an r dimensional "reduced" matrix of $\mathbf{A}$: $\tilde{A}$ by projecting $\mathbf{A}$ onto the column space of U_r^*[5]:

$$\tilde{A} = U_r^*\mathbf{A}U_r \tag{5}$$

where $\tilde{A}$ is an $r \times r$ complex-valued matrix. We can thus write the eigendecomposition of $\tilde{A}$ as $W\Omega W^{-1}$ and together with equality 5 yields:

$$\mathbf{A}U_r = U_r\tilde{A} = U_rW\Omega W^{-1} \implies \mathbf{A}U_rW = U_rW\Omega \tag{6}$$

And so by the second equality in equality 6, the eigenvalues and eigenvectors of $\tilde{A}$ are the same as those of DMD matrix $\mathbf{A}$ up until its r-th rank[3]. We briefly mention the omission of the Residual error $\mathbf{R}$, as the system $Y_n = \mathbf{A}X_n$ is more exactly written as $Y_n = \mathbf{S}X_n + r$, where $r \in \mathbf{R}$, $\forall n \in N$. Discussions of this error can be found in [5] and [3]. We make the assumption that this error is insignificant and consider the matrix $\mathbf{A}$ to be an a reasonably accurate estimate of the DMD matrix.
The last point in our discussion of DMD leads us to remark that the real portion of the eigenvalues $\omega_i \in \Omega$ (for $i = 1, ..., r$) of $\tilde{A}$ informs us on the slow or fast nature of our dynamics in time. Higher values indicate short timescale (fast) dynamics while lower ones correspond to longer (slow) dynamics.

Now that we have recalled the theoretical foundations of the standard DMD method, we return to the discussion of a Sliding-Window augmentation. In contrast to DMD, the Sliding-Window method is not seeking a single DMD matrix $\mathbf{A}$ to approximate the next time snapshot of data ($X_{n+1} = \mathbf{A}X_n$), but instead

for a matrix $\mathbf{A}^k$ to approximate the next time point of the system in each window. More concisely, we seek $X_n^{k+1} = \mathbf{A}^k X_n^k$. This involves computing k SVDs, k matrices $\tilde{A}^{(k)}$ and extracting $r \times k$ where r is an agreed upon optimal SVD truncation.
This allows for a much more precise and identification of slower timescale dynamics since it enables us to extract and examine the progress of the eigenvalues as we calculate the DMD matrices multiples times through the overlapping windows created from the data set. To that purpose, we aim to find a threshold $\lambda > 0$ for which the eigenvalues (for all k windows) can be separated into "fast" and "slow" dynamics.

3 Algorithm Implementation and Development

We begin this section with a few specifications on the implementation. In this project, both DMD and Sliding-Window DMD are implemented, but since Sliding-Window DMD is built entirely on DMD, we will discuss both through our exposition of S.W DMD. Second, there are several values for the window size for which the methods were implemented, and there are two videos. As a consequence, we only present the code for one video and one value of T, the window size, as the implementations are almost identical.

We now begin our discussion of the algorithm implementation.
The first step in our implementation is to capture the video frames which will make up our data sets. To achieve this we use the open-source Python library OpenCV as well as NumPy, another of Python's library for array processing. We pull the frames from the video and stack them into a numpy array. We then reshape the array into a matrix where each column is a snapshot in time (please refer to lines 1-28 in Appendix A)
The second step is to create our sliding windows. In order to do that, we must first select a window size T. We can expect the window size to have a considerable impact on the performance of the method since a window size that is too large would fail to give us any significant advantage over standard DMD and one too small might be unable to capture the longer timescale dynamics. To test the effects of a different window size, we pick three different ones for each video with the expected optimal one to correspond to capturing about 1 second in each window (dividing the number of frames by the length in seconds of the video yields the window size). Once T is determined, we can compute the number of windows by $k = N - T$, where N is the number of frames. We can then iterate k times through our data matrix appending the current window to a matrix "matrixWindows" (see lines 42-50).

Our next goal is to then compute the SVD decomposition of each window. To do so, we first define arrays to store each components of the SVD. It is easily noted that by and large, Sliding-Window DMD amounts to adding an extra dimension to most calculations required for standard DMD. We then compute the SVD's through NumPy's np.linalg.svd function and store the results in the corresponding arrays (see lines 52-66).
With the SVDs of every window in hand, it is the moment to determine optimal rank truncation. To that purpose, we refer to [2] which allows us to do so with a simple formula which takes advantage of the rank-dimensionality relationship of our data (variable "beta"). A polynomial function of third degree takes beta as input and gives us back the variable omega, the scaling factor of the median of all singular values for each window. The scaled medians are then used as a cutoff point for the most important singular values (which are expected to be greater than their corresponding scaled median). We establish a counter of significant singular values and store the count for every window in an array (see lines 78-94). Though we now have an optimal truncation for every SVD, we are still faced with deciding what the rank of the DMD matrices $\mathbf{A}^{(k)}$ should be. Making the DMD matrices the same size significantly simplifies the processing and analysis of the data. In this instance, the largest optimal rank is chosen as a truncation point. We hold the hypothesis that more singular values will yield better results, but this is certainly worth exploring in the future.

Once a rank r is established, we reduce our data accordingly and compute the $r \times r$ DMD matrices (see lines 108-156). We then extract the eigenvalues and eigenvectors of every DMD matrix into corresponding arrays with NumPy's np.linalg.eig function. We then compute continuous eigenvalues from the discrete ones given by the DMD matrices. We then square the eigenvalues and take their absolute value. The purpose in doing so is to amplify the differences between the fast and slow dynamics (see lines 162-220).

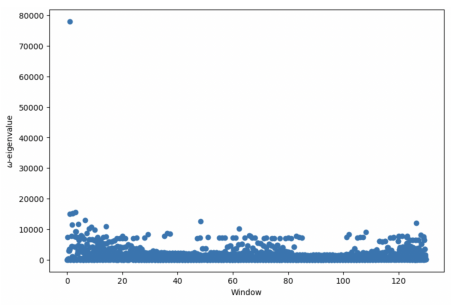
Our final goals then are to visualize the eigenvalues and attempt to identify more exact delineations with a clustering algorithm. First we concatenate the the eigenvalues of all windows into a single vector and then plot that vector through Python's data visualization library, matplotlib. The plots provide adequate visual appraisal of the separation of dynamics, but for more precise assessment we use k-means clustering. For the video with a single motion timescale, we set the number of clusters to two hoping to separate the foreground of the video from the static background. For the video with two different motion timescales, we set the number of clusters to 3 in an attempt to separate the background from the slow movement as well as the fast movement. The number of clusters determined for each video are the same for both Sliding-Window DMD and standard DMD. The k-means clustering algorithm is implemented through sklearn's KMeans library (see lines 232-268).

4 Computational Results

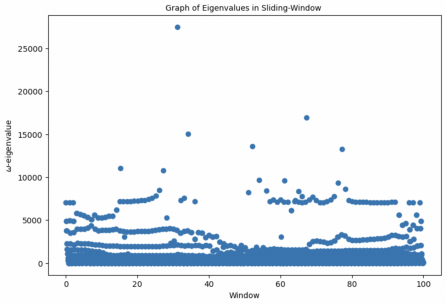
We now take a look at the results of our algorithm implementation.
We first examine the figures and note that the contents of the video (the movement in different frames) do not seem to be identifiable from looking at the eigenvalue distribution across windows. The only potential exception is found in figure 1b, where the at the halfway point the eigenvalues seem to collapse which roughly does correspond to the moment where the cat pauses its stride briefly in the video. We can also observe that for both videos, as the value of T increases the degree of separation seems to increase as well.

We can now take a look at the results of the clustering algorithms found in the tables below. A consistent trend seems to be that as the window size increases the all three of the clusters' means decrease. From table 1 and table 3 we can notice that the performance of the standard DMD algorithm is quite close to the one of Sliding-Window for the higher window size.

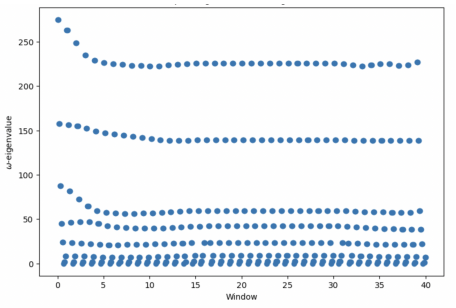
Overall, the results are unfortunately quite difficult to interpret and seem to be inconclusive.



(a) $T = 30$

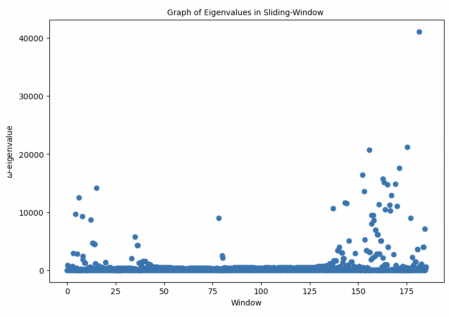


(b) $T = 60$

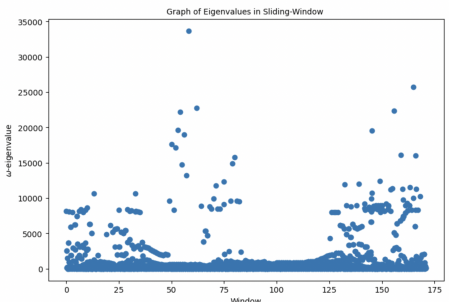


(c) $T = 120$

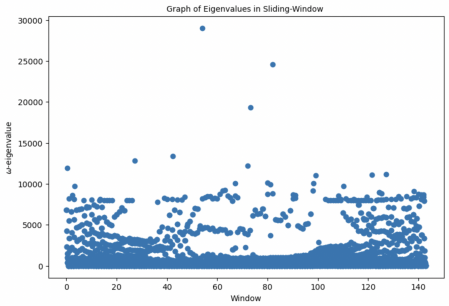
Figure 1: Distribution of the square of eigenvalues for different window sizes, single timescale video



(a) $T = 14$



(b) $T = 28$



(c) $T = 56$

Figure 2: Distribution of the square of eigenvalues for different window sizes, multiple timescales video

Table 1: Results of k-means clustering for multiple timescale dynamics video, Sliding-Window

Window size T	**1st cluster mean**	**2nd cluster mean**	**3rd cluster mean**
14	398.99	11 530.38	41 124.39
28	514.62	7625.40	19 823.12
56	397.95	3185.86	7436.86

Table 2: Results of k-means clustering for single timescale dynamics video, Sliding-Window

Window size T	**1st cluster mean**	**2nd cluster mean**
30	754.79	6349.97
60	542.97	5400.80
120	24.54	184.91

Table 3: Results of k-means clustering, standard DMD

Video	**1st cluster mean**	**2nd cluster mean**	**3rd cluster mean**
single timescale	235.82	1692.49	x
multiple timescales	399.20	2691.11	7519.17

5 Summary and Conclusions

In conclusion, in this paper we attempted to implement Sliding-Window DMD as a possible solution to the standard DMD method shortcomings when it comes to dynamical systems which unfold on multiple timescales. Our data sets consisted of two videos, one where movement unfolds on a single timescale and another which features both rapid and slower movement. Identification of slow and rapid movement was

relatively successful, but assessing the two methods' performance against the other remains difficult.

References

[1] Jason Bramburger. *Data-Driven Methods for Dynamic Systems: A new perspective on old problems, lectures 11-13*. 2023.

[2] Matan Gavish. *The Optimal Hard Threshold for Singular Values is 4/sqrt(3)*. 2014.

[3] Nathan Kutz. *Dynamic Mode Decomposition for Real-Time Background/Foreground Separation in Video*. 2014.

[4] NASA. *Arctic Sea Ice Minimum Extent*. 2023.

[5] Jonathan Tu. *Dynamic Mode Decomposition: Theory and Applications*. 2013.

Appendix A Python Code

```
import cv2 as cv
import numpy as np
from scipy import linalg
from matplotlib import pyplot as plt


# In[2]:


video = cv.VideoCapture("/Users/nadaelarabi/Data-driven-dynamical-systems/SlidingWindowDMD/V2R2.mp4")

frames = []
ret = True
while ret:
    ret,img = video.read()
    if ret:
        greyscale = np.mean(img,-1)
        frames.append(greyscale)
videoData = np.stack(frames,axis=0)
videoDataTranspose = videoData.T #Tjis is possibly unecessary?

widthVideo = videoDataTranspose.shape[0]
heightVideo = videoDataTranspose.shape[1]
numberPixelsPerFrame = widthVideo * heightVideo
numberOfFrames = videoDataTranspose.shape[2]

#Here we can reshape the data to have each frame represented as columns stacked
dataMat = (np.reshape(videoData,(numberOfFrames,numberPixelsPerFrame))).transpose()


# In[3]:


dataMat.shape


# In[4]:


#We create the Windows

#T refers to the size of each window
T = 60
numberOfWindows = numberOfFrames - T

#This matrix will contain the Windows
```

```
matrixWindows = []
for i in range(numberOfWindows):
    matrixWindows.append(dataMat[:,i:i+T])
matrixWindows = np.array(matrixWindows)

#Now we can make matrices to contain the values of the SVD decomposition
u_matrix = []
vt_matrix = []
sigma_matrix = []

#Here we calculate the SVD of each of the Windows
for i in range(numberOfWindows):
    u,s,vt = np.linalg.svd(matrixWindows[i],full_matrices=False)
    u_matrix.append(u)
    vt_matrix.append(vt)
    sigma_matrix.append(s)

u_matrix = np.array(u_matrix)
vt_matrix = np.array(vt_matrix)
sigma_matrix = np.array(sigma_matrix)


# In[5]:


#Now that we have the SVD for each Window, we can find the number of significant modes for
    each

#Question: do the DMD matrices A^k have to all be the same size? Why does the book say they
    would have N eigenvalues?
#Answer: They should preferrably be the same size to make analysis easier


significant_singularVal_Matrix = []
tau_matrix = []
nbRowsWindow = numberPixelsPerFrame
nbColumnsWindow = matrixWindows.shape[2]
beta = 1-(nbColumnsWindow/nbRowsWindow)
omega = 0.56*(beta*beta*beta) - 0.95*(beta*beta) + 1.82*beta + 1.43

for i in range(numberOfWindows):
    median = np.median(sigma_matrix[i])
    tau = omega * median
    tau_matrix.append(tau)
    counter = 0
    j =0
    while tau<sigma_matrix[i][j]:
        j += 1
        counter += 1
    significant_singularVal_Matrix.append(counter)


# In[6]:


#We've only defined the tau matrix to do some manual verification

significant_singularVal_Matrix = np.array(significant_singularVal_Matrix)
tau_matrix = np.array(tau_matrix)

#Now we should determine how many significant values we should keep for all DMD matrices
#One approach would be to take the maximum:

nbSingularValues = np.max(significant_singularVal_Matrix)
nbSingularValues

#Now we can apply the DMD algorithm to every window

```

```
#Here we reduce our u_matrix depending on the selected number of singular values
u_reduced = []
for i in range(numberOfWindows):
    u_reduced.append(u_matrix[i][:,0:nbSingularValues])
u_reduced = np.array(u_reduced)


# In[8]:


data_reduced = []
for i in range(numberOfWindows):
    data_reduced.append(np.matmul(u_reduced[i].T,matrixWindows[i]))
data_reduced = np.array(data_reduced)


# In[9]:


#Now we can create our X and Y matrices
n = data_reduced[0].shape[1]
X_matrix = []
Y_matrix = []
for i in range(numberOfWindows):
    X_matrix.append(data_reduced[i][:,0:n-1])
    Y_matrix.append(data_reduced[i][:,1:n])
X_matrix = np.array(X_matrix)
Y_matrix = np.array(Y_matrix)


# In[10]:


#Now we can calculate the transpose of each X:
Xt_matrix = []
for i in range(numberOfWindows):
    Xt_matrix.append(np.linalg.pinv(X_matrix[i]))
Xt_matrix = np.array(Xt_matrix)

#We proceed to finding the DMD matrices
DMD_matrices = []
for i in range(numberOfWindows):
    DMD_matrices.append(np.matmul(Y_matrix[i],Xt_matrix[i]))
DMD_matrices = np.array(DMD_matrices)


# In[12]:


#Now we can extract all eigenvalues and eigenvectors from the DMD matrices
eigenvalues_matrix = []
eigenvector_matrix = []
for i in range(numberOfWindows):
    eigenvalues, eigenvectors = np.linalg.eig(DMD_matrices[i])
    eigenvalues_matrix.append(eigenvalues)
    eigenvector_matrix.append(eigenvectors)
eigenvalues_matrix = np.array(eigenvalues_matrix)
eigenvector_matrix = np.array(eigenvector_matrix)


# In[13]:


reals = eigenvalues_matrix[80].real
imagin = eigenvalues_matrix[80].imag
plt.plot(reals,imagin,'g*') #where I divided by 6/379
plt.ylabel('Imaginary')
plt.xlabel('Real')
```

```
plt.title("Plot of Discrete Time step Eigenvalues $\mu$",fontsize=10)
plt.show()


# In[14]:


omega = [] # set of transformed eigenvalues
dt =6/numberOfFrames

for i in range(numberOfWindows):
    omega_log = np.log(eigenvalues_matrix[i])/dt
    omega.append(omega_log)
omega = np.array(omega)

# for eigenvalue in eigenvalues:
#   omegak = np.log(eigenvalue)/dt
#   omega.append(omegak)

# omega = np.array(omega)


# We can plot the continuous time eigenvalues
plt.figure(figsize=(5,5))
plt.xlabel("Real")
plt.ylabel("Imaginary")
plt.title("Plot of Continuous Time Eigenvalues $\omega$",fontsize=10)
plt.scatter((omega[5]).real,(omega[5]).imag)


# In[15]:


#Let's try and see if we get a proper separation

#First let's take the absolute value of all eigenvalues and square them

for i in range(numberOfWindows):
    omega[i] = np.absolute(omega[i])
    omega[i] = np.square(omega[i])

#Now we concatenate all eigenvalues in a single vector
array_nbColumns = numberOfWindows*omega[0].shape[0]
# concatenated_eigenvalues = np.zeros((1,array_nbColumns))
# position = 0
# for i in range(numberOfWindows):
#     concatenated_eigenvalues[:,position:i+nbSingularValues] = omega[i]
#     position = i*nbSingularValues

#Maybe just reshape the omega vector

concatenated_eigenvalues = np.reshape(omega,(array_nbColumns,1))


# In[17]:


array_nbColumns


# In[19]:


x_axis = np.linspace(0,139,array_nbColumns)
plt.figure(figsize=(9,6))
plt.xlabel("Window")
plt.ylabel("$\omega$-eigenvalue")
plt.title("Graph of Eigenvalues in Sliding-Window ",fontsize=10)
```

```
plt.scatter(x_axis,concatenated_eigenvalues.real)


# In[21]:


concatenated_eigenvalues = np.real(concatenated_eigenvalues)


# In[23]:


from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=0).fit(concatenated_eigenvalues)


# In[1]:


kmeans.cluster_centers_


# In[ ]:
```