MATH 494-A: Assignment 1
Ski Slopes and Formula Cars

Myriam Boucher-Pinard (40109509)

February 12, 2023

Abstract

Purpose: To conduct a Dynamic Mode Decomposition (DMD) analysis of two videos to isolate the salient information (the moving car or the skier) and the static information (the background). **Method:** Initiation of a low-rank approximation Singular Value Decomposition (SVD), resulting in the DMD, providing a time-bound, low-error forecast of the initial frame to the next. **Results:** An attempt at the reconstruction of the frames containing only the object(s) in motion or containing only the static background. **Conclusion:** SVD provides an ability to dissociate space and time information, resulting in accurate DMD due to the forecasting of one frame to the next.

1 Introduction and Overview

Videos as dynamic systems represent changes in space through time; indeed, videos mostly imply motion. However, videos may be further observed as having foregrounds and backgrounds, in such that the foreground is typically expressed as the objects in motion, the change of space in time, whereas the background is conversely expressed as the static information of the video; the portion that could mostly be expressed by very few frames due to the lack of change in the information displayed. This is precisely the expectation of this assignment; expressing four-dimensional matrices from two videos into two-dimensional matrices, with the dimensions being space and time, and separating their foregrounds and backgrounds. This shall be achieved by conducting the compression algorithm, the Singular Value Decomposition (SVD), and identifying the more salient information through the low-rank approximation of this algorithm, such that the approximation of the change in space over time deviates as little as possible from its actual change. This estimate is achieved by minimizing the distance between an initial frame of the video to the next, which results in Dynamic Mode Decomposition (DMD). Observing the eigenvalues over continuous time allows us to determine what is changing over time and what is not; for instance, in the first video (high salience), the change that space is experiencing over time is that of the cars racing around the corner to the finish line, and the flag waving at the end of the race. In the second video (low salience), the skier drops, disrupting some snow, and as such, the foreground should be much less occupied by the latter than by the former. Hence, the observations below shall provide insight into the salience of the information necessary in the compression of a video, while maintaining its integrity.

2 Theoretical Background

The mathematical basis required to achieve video compression and the isolation of the foreground compared to the background begins in linear algebra; indeed, the matrix is reduced from four dimensions into two, while remaining composed of only real values. This is done in order to generate the Singular Value Decomposition (SVD), which isolates space from time, through matrix factorization, as observed in equation (1) [1]. Here, U and V are unitary matrices with complex values, U composed of the space values (width by length) and V of time [1]. Together, they form the A matrix, in which each column represents a frame, and moving across the matrix may be compared to watching the video, or advancing in time.

$$A = U \sum V^* \tag{1}$$

However, due to the large quantity of data present within the videos, the low-rank approximation must be calculated instead, in order to begin pruning unnecessary data, or rather values within A that have little incidence on the video itself. This process is implemented in order to solve the minimization problem, as provided in equation (2). Rather, the Frobenius norm of equation (2) returns the next largest singular value of the difference between the SVD and the compact SVD (obtained through the elimination of the larger quantity of zero singular values). Thus, equation (2) demonstrates the minimum error between the matrix factorization of the original data and that of the compacted data, such that the approximation is as accurate as the equation may allow, hence the solution to the minimization problem [1].

$$argmin_{AA^*=I}||Y - AX||_F^2 \tag{2}$$

This low-rank approximation SVD may then be applied within Dynamic Mode Decomposition (DMD), a linear model of progression through time. This process implies taking iterations of the lower rank data matrix, such that matrix Y represented in equation (3) is matrix X in the future; in this relationship, A is a linear transformation in order to move from one time step to another. Thus, in order to solve for this linear transformation, one must calculate the pseudoinverse of X, as X is not a square matrix (due to the removal of the final column in the computation of iterations over time, such that equation (4) may be equivalently represented in equation (5)) [1], [2].

$$Y = AX \tag{3}$$

$$A = YX^{\dagger} \tag{4}$$

$$Z_{n+1} = AZ_n \tag{5}$$

Matrix A later allows for the definition of the eigenvalues and eigenvectors, the latter also known as the DMD modes [1]. The iterates described as Z in equation (5) may then be computed through equation (6), the sum over the number of optimal eigenvalues (s-rank) of the product of eigenvalues (to the power of the nth time frame), coefficients (calculated through the inner product of the first frame and the DMD modes), and the DMD modes. Due to the instability of a dynamical system, DMD serves to stabilize the data so that it may be mapped out onto the unit circle; however, some eigenvalues may appear inside the unit circle or outside it, which leads to the implementation of instability suppression within DMD [1]. This instability suppression is precisely the solution of equation (2).

$$Z_n = \sum_{m=1}^{M} (\mu_m^n \langle Z_0, \psi_m \rangle \psi_m) \tag{6}$$

The final portion of the process relies on the isolation of the foreground and the background; once the iterates are calculated, the initial data and reconstructed data may be compared. While the original data belongs solely to the real plane, the reconstructed data shall contain complex values; thus, within the separation of the foreground and background continuous eigenvalues (continuous in that the omega represents the logarithm of the eigenvalues taken over the change in time, from equation(7)), the eigenvalues shall be divided according to the threshold set to isolate values near 0 and those near 1. Values near 0 represent the slow-moving components of a dynamical system, that of the background, input into the low-rank matrix; the values near 1 are those that contain more salient information in them, the values that contain the highest percentage of change through time [1], [2], [3]. Thus, the foreground, or the sparse matrix as seen in equation (7), represents that which moves through time, expressed through the absolute data, ensuring that there are no pixels with negative intensities [2], [3].

$$X = X_{DMD}^{Low-Rank} + X_{DMD}^{Sparse} \tag{7}$$

The outlined process above is particularly important as it considers data of high dimensionality with the ability to reduce it to data having low rank; the computation becomes linear and space may therefore be described with very little information.

3 Algorithm Implementation and Development

3.1 Assumptions and Greyscale

The initial portion of the video compression required setting up the four-dimensional components of the imported video, being the frame count, the width of each frame, the height of each frame, and RGB. As the data needed to be reduced to a two-dimensional matrix, the use of OpenCV's COLOR.BGR2GRAY function followed, in order to then only have one-third of the information for each pixel, and as such, ensuring that Python could run the future lines of code.
The data matrix now being three-dimensional, the last two components were reshaped (using the Numpy reshape function) so that the product of the width and length was instead computed, providing the area of each frame. At this point, the matrix was two-dimensional, a time-by-space matrix. However, as abovementioned, in order to later run SVD, the data must be a space-by-time matrix, and as such, using the Numpy transpose function, the data was rearranged in order to appear as needed.

3.2 SVD and Low-Rank Approximation

The following portion of the process required the running of the SVD algorithm, using the linalg.SVD existing program; however, due to the weight of the data, the full matrix could not be used, and as such, the economical one was obtained.
However, as mentioned in the mathematical background portion above, this SVD is not suitable, as the data is unstable and too large to be fully used for video compression. Hence, the low-rank decomposition was then run, first by computing the energy percentage to be maintained within the new s-rank SVD matrix factorization. This was obtained through a while-loop, where energy was maintained at a 0.99999 threshold. The low-rank SVD was then run, using a function defined as lowrankSVD, by claiming the s-rank to be an integer, the U-matrix an array, and the A matrix being the location of the imported data into the function.

3.3 DMD

Once the low-rank SVD was complete, the X and Y matrices were created, such that the last column of X was removed using the Numpy delete function, the same occurring to the first column of the Y matrix. The computation of the pseudo-inverse of X followed, through the linalg.PINV function, in order to then compute the A matrix (the DMD matrix). This matrix was calculated using the Numpy dot product function.
The eigenvalues and eigenvectors were obtained through the linalg.EIG function, and after computing the inverse of the eigenvectors matrix, the coefficient matrix was obtained, from the dot product of the eigenvector inverse matrix and the first frame (identified by taking the first column of the s-rank SVD matrix).
In order to later reconstruct, eigendecomposition must first occur; indeed, a double for-loop achieved the summation from equation (6) occurring over the s-rank and the time frames. The absolute value of the logarithm of the eigenvalues over the change in time was then computed (omega), in order for a for-if-else-loop to determine which omegas belong to the foreground and which to the background. The indexing of the eigenvalues was essential here, as it allowed us to determine the position of the eigenvalue in the eigendecomposition matrix that belonged to the background and change it to a value of zero in the eigenvalues of the foreground. Using the matplotlib.PYPLOT package and plot and margins functions, the continuous eigenvalues (omegas) were plotted (see figures 1, 2, 3, 4). Figures 1 and 3 represent both the logarithms and the absolute logarithms of the eigenvalues, while figures 2 and 4 are zoomed in, and as the values of the eigenvalues are all clustered near 0, the use of 0.0001 as a threshold for splitting the eigenvalues into the foreground and background was justified.

3.4 Reconstruction

Reconstruction of the frames was possible by first running the same double for-loop as the eigendecomposition, this time with the eigenvalues of the foreground (and the coefficient matrix being adjusted for the eigenvalue equal to zero in this array). As with the methodology outlined in [2], [3], the initial background eigendecomposition matrix was formulated through the Numpy subtraction function of the eigendecomposition matrix and the foreground eigendecomposition matrix. Then, the absolute value of this background

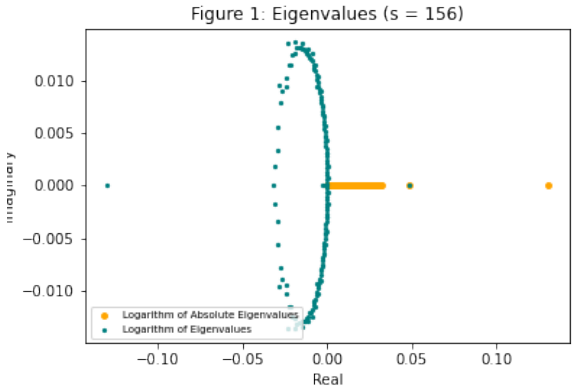

Figure 1: The plotted eigenvalues over an s-rank of 156 over the real and imaginary numbers.

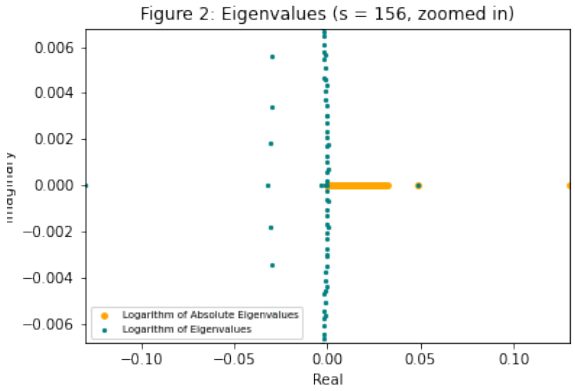

Figure 2: The zoomed-in plotted eigenvalues over an s-rank of 156 over the real and imaginary numbers.

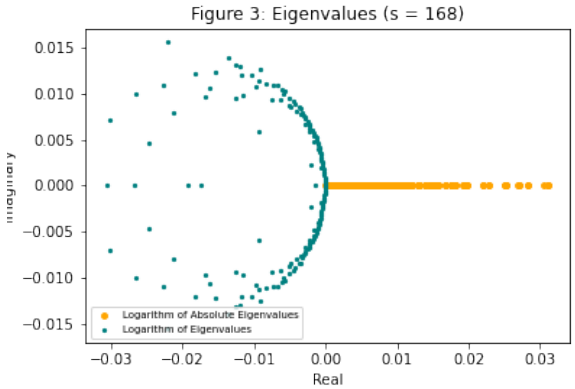

Figure 3: The plotted eigenvalues over an s-rank of 168 over the real and imaginary numbers.

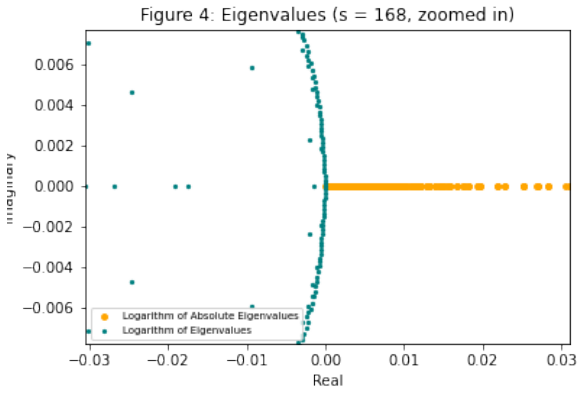

Figure 4: The zoomed-in plotted eigenvalues over an s-rank of 168 over the real and imaginary numbers.

matrix was taken, and a double for-loop with double indices was run in order to ensure that the data was positive and real-valued. This then allowed for the sparse DMD matrix (foreground) to be constructed, such that the data from the original video and the absolute-valued background matrix were subtracted (at this point, only real-valued data is present). The R matrix was finally created through another double for-loop, parsing between positive and negative data, and inputting negative data back into the R matrix in order to exclude it from the foreground. The final low-rank and sparse DMD matrices were obtained by adding the R matrix back into the low-rank matrix and subtracting the R matrix from the sparse matrix, using their respective operational Numpy functions.
Frames from the video were then finally reconstructed using the Image function, and two images were saved from the fully reconstructed greyscale matrix, from the sparse matrix, and from the low-rank matrix.

4 Computational Results

Unfortunately, although that which should have been observed was that of a foreground with objects in motion isolated over a black screen and a background of mostly blurry objects and space, the resulting images are that of a nearly identical foreground to the reconstructed video and a black screen for the background.
The issue is believed to lie in the cutting off of the eigenvalues, as a single eigenvalue may be observed for the low-rank matrix, and the remaining eigenvalues may be found in the sparse. It may be that these values should be nearly fully reversed, in that a smaller number of eigenvalues shall contain the information of the objects in motion, whereas a larger number of eigenvalues shall contain the background information (while maintaining the integrity of the video as determined through the s-rank matrix in the SVD component). This is assumed to be where the issue lies, as the reversal of this would lead to a black screen for the foreground and a near-complete reconstruction in the background. According to Kutz [3], the omegas must be cut off with a lambda less than epsilon much smaller than 1; further exploration of this issue shall be conducted in the following report.
Had the program concluded in the proper way, the reconstruction found in images 1 and 4 would have led to images 2 and 5 (the sparse matrix) only containing the moving cars and the waving flag in Monte Carlo and images 3 and 6 would have been the unmoving background and some blurriness for the moving objects. For the skiing video, the reconstructed video of images 7 and 10 would have led to the skier and perhaps some snow appearing in images 8 and 11, and images 9 and 12 would have been a mostly undisturbed landscape of snow and trees and some blurriness for the skier and falling snow caused by their drop.

Image 1: Monte Carlo Reconstruction, frame 3.

Image 2: Monte Carlo Sparse Reconstruction, frame 3.

Image 3: Monte Carlo Low-Rank Reconstruction, frame 3.

Image 4: Monte Carlo Reconstruction, frame 375.

Image 5: Monte Carlo Sparse Reconstruction, frame 375.

Image 6: Monte Carlo Low-Rank Reconstruction, frame 375.

Image 7: Ski Drop Reconstruction, frame 3.

Image 8: Ski Drop Sparse Reconstruction, frame 3.

Image 9: Ski Drop Low-Rank Reconstruction, frame 3.

Image 10: Ski Drop Reconstruction, frame 451.

Image 11: Ski Drop Sparse Reconstruction, frame 451.

Image 12: Ski Drop Low-Rank Reconstruction, frame 451.

5 Summary and Conclusions

Although the process was not conclusive, the idea behind DMD is that of linearly moving through time in a frame-by-frame low-rank approximation of data (in this case, two videos), in order to compress the video and exclude unnecessary data within it. This was attempted through the minimization problem, using the SVD as a way of factoring space and time to isolate the eigenvalues and eigenvectors, which through DMD, would then give rise to the foreground and background through eigendecomposition and reconstruction.
Although inconclusive, several troubleshooting opportunities did present themselves and remain interesting for future research; indeed, the first attempts included maintaining the RGB values, which led to the program refusing to run, as this led to each pixel having thrice the amount of information and having to stack this information individually for each pixel, so that the information was not spread out throughout the data matrix. Next, hexadecimal was attempted, in an effort to maintain a colour gradient for the video; unfortunately, this led to its own set of issues, as the hexadecimal system has 16-bit unsigned integers, compared to the 8-bit unsigned integers of greyscale. Thus, this program was also too heavy to run with the information at hand, and as such, was infeasible. Perhaps a future direction shall be to attempt running the entire process (up to reconstruction) by isolating each of the RGB values, such that eigendecomposition could occur for red, green, and blue separately, and then reconstructed as one at the very end. Furthermore, as abovementioned, another issue encountered throughout this process was that of the lack of foreground and background separation; indeed, further exploration shall have to be conducted on the threshold used for splitting the eigenvalues between the sparse and low-rank matrices.
In the end, one must give credit where credit is due, and thus a very special thank you to Samir, Nada, Serly, and Alex for your help and support for the coding throughout this process.

Appendix A Python Premade Functions

- `cv.VideoCapture` Imports a video.
- `get(cv.CAP.PROP.FRAME.COUNT)` Defines the variables of the imported video (equivalent for width and height).
- `np.empty` Creates an empty array as a buffer.
- `cv.COLOR.BGR2GRAY` Transforms RGB data into greyscale.
- `np.transpose` Generates the transpose of the input matrix.
- `np.linalg.svd` Generates the SVD of a matrix.
- `np.save` Saves a matrix as a file.
- `np.load` Loads a saved matrix back into the program.
- `np.linalg.matrix.rank` Computes the rank of a matrix.
- `np.dot` Computes the dot product of two arguments.
- `np.delete` Used to delete specified columns of the matrix.
- `np.linalg.pinv` Computes the pseudo-inverse of the specified matrix.
- `np.linalg.eig` Produces the eigenvalues and eigenvectors of a matrix.
- `np.linalg.inv` Computes the inverse of the specified matrix.
- `np.reshape` Reshapes the specified matrix to the specified shape.
- `np.abs` Computes the absolute value of the argument.
- `np.log` Computes the logarithm of the argument.

- `lib.scatter` Generates a scatter plot.
- `np.zeros` Generates an array of zeros.
- `copy.deepcopy` Creates a copy of an existing array.
- `np.subtract` Computes the difference of the arguments.
- `np.add` Computes the sum of the arguments.
- `im.fromarray` Generates an image from a matrix.

Appendix B Python Code

The code for both videos is the exact same apart from some of the values found within it. The Monte Carlo code is the one included below.

```
! pip install numpy
import numpy as np
np.__version__ #checks proper installation of package
import cv2 as cv
data = cv.VideoCapture('monte_carlo_low.mp4')
#using Monte Carlo low as the higher resolution version lengthens processing times
#defining the variables of the empty matrix:
framecount = int(data.get(cv.CAP_PROP_FRAME_COUNT))
framewidth = int(data.get(cv.CAP_PROP_FRAME_WIDTH))
frameheight = int(data.get(cv.CAP_PROP_FRAME_HEIGHT))
#creating a buffer in order to be able to insert the information from the video into it:
buffer = np.empty((framecount, frameheight, framewidth, 3), np.dtype('uint8')) #8-bit unsigned integer,
↪  desired output (due to the RGB values)
#the buffer stores information temporarily

##TURNING RGB TO GREYSCALE:
#creating a buffer in order to be able to insert the information from the video into it:
buffer_grey = np.empty((framecount, frameheight, framewidth), np.dtype('uint8'))
fc = 0
ret = True
while (fc < framecount and ret):
    ret, buffer[fc] = data.read()
    buffer_grey[fc] = cv.cvtColor(buffer[fc], cv.COLOR_BGR2GRAY) #converts each frame to greyscale
    fc += 1
data.release()
#print(buffer.shape) #(379 [frames], 540 [width], 960 [height], 3 [RGB])
#print(buffer)
print(buffer_grey.shape)
print(buffer_grey)
#organizing the variables for reshaping:
frames = buffer.shape[0]
height = buffer.shape[1]
width = buffer.shape[2]
RGB = buffer.shape[3]
#X = np.reshape(buffer, (frames, height*width, RGB)) #3-D matrix - this was useful for the RGB and hexadecimal
↪  trials
X_grey = np.reshape(buffer_grey, (frames, height*width)) #2-D matrix
print(X_grey.shape)
print(X_grey)
#matrix has now been reshaped, the last column has been removed (RGB)
#rows were stacked, such that we have a TIME X SPACE matrix (frames x height*width)
#take the transpose (not the Hermitian, as there are no complex values)
XT_grey = np.transpose(X_grey)
print(XT_grey.shape)
print(XT_grey)
#applying the SVD for the first round - it will run heavy
#to run SVD - XT_grey will be split into U, S, and VT matrices
#U,S,VT = np.linalg.svd(XT_grey, full_matrices=True) #"True" condition returns the full SVD (however, too heavy
↪  to run)
```

```
U,S,VT = np.linalg.svd(XT_grey, full_matrices=False) #"False" condition returns a reduced SVD, allowing the
↪  program to run
#the "False" condition truncates the bottom rows that are only 0
np.save('U.npy', U)
np.save('S.npy', S)
np.save('VT.npy', VT)
#checking to see whether the original matrix may be reconstructed:
XT_grey_remake = (U @ np.diag(S) @ VT)
print(XT_grey_remake.shape)
print(XT_grey_remake)
U = np.load("U.npy")
S = np.load("S.npy")
VT = np.load("VT.npy")
print(U.shape)
print(S.shape)
print(VT.shape)
print(U)
print(S)
print(VT)
##LOW-RANK DECOMPOSITION:
#this allows for the identification of the more salient pieces of information
#Eckart-Young-Mirsky theorem:
#computing the rank of the matrix
rank_XT = np.linalg.matrix_rank(XT_grey)
print("Rank of matrix: ", rank_XT)
#print("S = ", S)
#calculating the p-rank: (to avoid redundancy with S)
p = 0
energy = 0
while energy <= 0.99999:
    energy = energy + (S[p]**2 / np.sum(np.diag(S)**2))
    p += 1
print("Optimal p-rank: ", p)
#take the transpose (Hermitian) of U in order to run the low-rank SVD:
def lowrankSVD(p: 'int', U: 'np.ndarry', A):
    return np.dot(np.transpose(U[:, :p]), A)
XT_greys = lowrankSVD(156, U, XT_grey) #computing the low-rank approximation of U and did the dot product with
↪  XT_grey
print(XT_greys.shape)
print(XT_greys)
##CREATING THE X AND Y:
X = np.delete(XT_greys, -1, 1) #this refers to the removal of the final column, as condition "-1" moves
↪  backwards, targeting the final column, and condition "1" targets columns over rows
Y = np.delete(XT_greys, 0, 1) #this refers to the removal of the first column, as condition "0" targets the
↪  first column, and condition "1" targets columns over rows
print(X.shape)
print(X)
print(Y.shape)
print(Y)
## Y = AX; TAKING THE PSEUDOINVERSE OF X:
PX = np.linalg.pinv(X) #pseudoinverse of X
print(PX)
##CALCULATING THE DMD MATRIX:
A = np.dot(Y, PX) #multiply the pseudoinverse of X with Y in order to retrieve the matrix A, the DMD matrix
print(A.shape)
print(A)
##APPLY DMD TO FIND THE EIGENVALUES AND EIGENVECTORS:
eigenvalues, eigenvectors = np.linalg.eig(A) #much like the SVD function, the output must be sorted
print(eigenvalues.shape)
print(eigenvalues)
print(eigenvectors.shape)
print(eigenvectors) #DMD modes (the eigenvectors are referred to as the DMD modes and are assumed to have unit
↪  Euclidean norm, p.15)
#inverse of the eigenvectors in order to take the low-rank decomposition of each frame:
eigenvectors_inverse = np.linalg.inv(eigenvectors)
print(eigenvectors_inverse.shape)
print(eigenvectors_inverse)
##LOW-RANK OF FIRST FRAME:
```

```
frame_1 = XT_greys[:,0] #this considers the first frame, which is the first column
print(frame_1.shape)
print(frame_1)
#matrix of coefficients:
#matrix of eigenvectors x matrix of coefficients = initial condition (frame 1)
#inverse of eigenvector matrix is thus taken to solve matrix of coefficients
C = np.dot(eigenvectors_inverse, frame_1)
print(C.shape)
print(C)
##EIGENDECOMPOSITION - PART I:
import math
#M = 156 - optimal p-rank
#N = 379 - total frames
C_new = C.reshape(156,1)
eigenva = eigenvalues.reshape(156,1)
Zn = np.empty([156,379], dtype = np.cdouble)
decomp = np.empty([156,1], dtype = np.cdouble)
#defining number of rows as p-rank and number of columns as frames
decomp.reshape(156,1)
print(decomp.shape)
print(decomp.dtype)
print(eigenva.shape)
print(C_new.shape)
print(eigenvectors.shape)
print(Zn.shape)
print(Zn.dtype)
##EIGENDECOMPOSITION - PART II:
#using the Zn formula on page 15 of the course notes
#m = 0
#n = 1
for n in range(379):
    for m in range(155): #as the first row is 0
        #print(eigenvectors[:,m]) #to check whether the slicing is 'good'
        decomp = decomp + (((eigenva[m+1])**(n+1))*(C_new[m+1])*eigenvectors[:,[m+1]]) #using [m+1] as it is
        ↪  the only way to get around ValueError: could not broadcast input array from shape (156,156) into
        ↪  shape (156,)
    Zn[:,[n]] = decomp
    decomp = np.empty([156,1]) #to initialize next column of Zn
np.save('Zn.npy', Zn)
print(Zn.shape)
print(Zn)
#importing necessary package for plotting:
import matplotlib.pyplot as lib
from matplotlib.pyplot import plot
from matplotlib.pyplot import margins
##DETERMINING THE CUTOFF: DMD SPECTRA
delta_t = 379/6 #frames/second
omega = np.abs(np.log(eigenva))/delta_t
omega_prime = np.log(eigenva)/delta_t
print(omega.shape)
print(omega)
print(omega_prime.shape)
print(omega_prime)
##PLOTTING THE EIGENVALUES (NOT CUTOFF):
#this is for observation purposes, to see the cutoff range
lib.scatter(omega.real, omega.imag, marker='o', c = 'orange', s = 15, label = 'Logarithm of Absolute
↪  Eigenvalues')
lib.scatter(omega_prime.real, omega_prime.imag, marker='o', c = 'teal', s = 5, label = 'Logarithm of
↪  Eigenvalues')
lib.margins(x=0.05,y=0.05)
lib.title("Figure 1: Eigenvalues (s = 156)")
lib.xlabel("Real")
lib.ylabel("Imaginary")
lib.legend(loc='lower left',prop={'size':7})
lib.savefig("Graph1.png")
lib.show()
##ZOOMING IN:
```

```
lib.scatter(omega.real, omega.imag, marker='o', c = 'orange', s = 15, label = 'Logarithm of Absolute
↪  Eigenvalues')
lib.scatter(omega_prime.real, omega_prime.imag, marker='o', c = 'teal', s = 5, label = 'Logarithm of
↪  Eigenvalues')
lib.margins(x=0,y=-0.25) #zooming in
lib.title("Figure 2: Eigenvalues (s = 156, zoomed in)")
lib.xlabel("Real")
lib.ylabel("Imaginary")
lib.legend(loc='lower left',prop={'size':7})
lib.savefig("Graph2.png")
lib.show()
##CUTTING-OFF EIGENVALUES:
#reasonable cut off = 0.0001, considering the clustering of the eigenvalues at 0
#separating the eigendecompositional matrix into fast and slow modes
fast = np.zeros([156,1])
slow = np.zeros([156,1])
#empty array designated towards including 'fast' modes
fast_index = 0
slow_index = 0
i = 0
slow_position = 0
for element in omega:
    if element > 0.0001: #leads to a single mode showing up in the low rank
    #if element > 0.001: #leads to 7 modes showing up in the low rank - with no change on the result
        #constructed a larger cutoff point to demonstate what happens when more eigenvalues go into the
        ↪  background
        fast[fast_index] = element
        fast_index = fast_index + 1
        i += 1
    else:
        slow[slow_index] = element
        slow_index = slow_index + 1
        slow_position = i
        i += 1
print(fast_index, slow_index)
print(fast)
print(slow)
print(i)
print("The index of the slow mode is: ", slow_position)
print("The slow mode is the 23 element in omega")
#this means that the element with index 22 in the coefficient matrix is the coefficient for the slow
↪  eigendecomposition
##RECONSTRUCTING THE FOREGROUND AND BACKGROUND - PART I:
#FOREGROUND - X_FAST
import copy
fast_C = copy.deepcopy(C_new)
fast_C[22] = 0
print(fast_C)
print(fast_C[22])
print(C_new[22])
Zn_fast = np.empty([156,379], dtype = np.cdouble)
decomp_new = np.empty([156,1], dtype = np.cdouble)
#defining number of rows as p-rank and number of columns as frames
decomp_new.reshape(156,1)
for n in range(379):
    for m in range(155): #as the first row is 0
        decomp_new = decomp_new + (((eigenva[m+1])**(n+1))*(fast_C[m+1])*eigenvectors[:,[m+1]]) #using [m+1]
        ↪  as it is the only way to get around ValueError: could not broadcast input array from shape
        ↪  (156,156) into shape (156,)
    Zn_fast[:,[n]] = decomp_new
    decomp_new = np.empty([156,1]) #to initialize next column of Zn_fast
print(Zn.shape)
print(Zn_fast.shape)
print(Zn_fast)
##RECONSTRUCTING THE FOREGROUND AND BACKGROUND - PART II:
#BACKGROUND - X_SLOW
Zn_slow = np.subtract(Zn, Zn_fast)
print(Zn_slow.shape)
```

```
print(Zn_slow)
##CONSTRUCTING THE R MATRIX - PART I: MERCI NADA!
absZn_slow = np.abs(Zn_slow)
LabsZn_slow = np.zeros([518400, 379])
for i in range(379):
    for j in range(156):
        LabsZn_slow[j][i] = absZn_slow[j][i]
print(LabsZn_slow.shape)
print(LabsZn_slow)
##CONSTRUCTING THE R MATRIX - PART II:
sparse = np.subtract(XT_grey,LabsZn_slow)
#must subtract the slow from the transposed video data
print(sparse.shape)
print(sparse)
R = np.zeros([518400,379], dtype = 'float')
for j in range(379):
    for l in range(156):
        if sparse[l][j] < 0.00:
            R[l][j] = sparse[l][j] #double indices due to 2-D array
print(R.shape)
print(R)
##FINALIZING SLOW AND FAST MODES:
lowrank = np.add(R, LabsZn_slow)
final_sparse = np.subtract(sparse, R) #as described in the methodology
print(lowrank.shape)
print(lowrank)
print(final_sparse.shape)
print(final_sparse)
##RECONSTRUCTING VIDEO:
#now, lowrank and final_sparse should no longer contain negative values
print(XT_grey.shape)
print(XT_grey)
RECON_VIDEO = np.add(lowrank, final_sparse)
print(RECON_VIDEO.shape)
print(RECON_VIDEO)
from PIL import Image as im #importing module to view frames as images
##FRAME IMAGES - PART I:
#image 1 - one of the first frames
RECON_IM1 = RECON_VIDEO[:,[2]]
RECON_IM1.shape
RECON_IM1 = RECON_IM1.reshape(540,960)
IM1 = im.fromarray(RECON_IM1)
IM1 = IM1.convert("L") #solves OSError: cannot write mode F as PNG (must use this for greyscale images)
IM1.save("IM1.png")
##FRAME IMAGES - PART II:
#image 2 - one of the last frames
RECON_IM2 = RECON_VIDEO[:,[374]]
RECON_IM2.shape
RECON_IM2 = RECON_IM2.reshape(540,960)
IM2 = im.fromarray(RECON_IM2)
IM2 = IM2.convert("L") #solves OSError: cannot write mode F as PNG (must use this for greyscale images)
IM2.save("IM2.png")
##RECONSTRUCTING FRAME IMAGES - PART I: LOW RANK
#image 3 - one of the first frames
LOW_IM1 = lowrank[:,[2]]
LOW_IM1.shape
LOW_IM1 = LOW_IM1.reshape(540,960)
LIM1 = im.fromarray(LOW_IM1)
LIM1 = LIM1.convert("L") #solves OSError: cannot write mode F as PNG (must use this for greyscale images)
LIM1.save("LIM1.png")
##RECONSTRUCTING FRAME IMAGES - PART II: LOW RANK
#image 4 - one of the first frames
LOW_IM2 = lowrank[:,[374]]
LOW_IM2.shape
LOW_IM2 = LOW_IM2.reshape(540,960)
LIM2 = im.fromarray(LOW_IM2)
LIM2 = LIM2.convert("L") #solves OSError: cannot write mode F as PNG (must use this for greyscale images)
LIM2.save("LIM2.png")
```

```
##RECONSTRUCTING FRAME IMAGES - PART I: SPARSE
#image 5 - one of the first frames
S_IM1 = final_sparse[:,[2]]
S_IM1.shape
S_IM1 = S_IM1.reshape(540,960)
SIM1 = im.fromarray(S_IM1)
SIM1 = SIM1.convert("L") #solves OSError: cannot write mode F as PNG (must use this for greyscale images)
SIM1.save("SIM1.png")
##RECONSTRUCTING FRAME IMAGES - PART II: SPARSE
#image 6 - one of the first frames
S_IM2 = final_sparse[:,[374]]
S_IM2.shape
S_IM2 = S_IM2.reshape(540,960)
SIM2 = im.fromarray(S_IM2)
SIM2 = SIM2.convert("L") #solves OSError: cannot write mode F as PNG (must use this for greyscale images)
SIM2.save("SIM2.png")

"""

##ATTEMPT : STACKING RGB ROWS (FAILED)
#it has failed as it never stopped running - there are too many data points and 16 Go of RAM is not enough to
↪  run the reshaping of each pixel
#turning a 3-D matrix into a 2-D matrix (maintaining RGB):
print(X[0].shape)
print(X[0][0])
print(np.reshape(X[0][0],(3,1)))
#print(np.append(np.reshape(X[0][0],(3,1)),np.reshape(X[0][0],(3,1)),0)) #attempt on a single pixel
X_new = np.empty((frames,X[0].shape[0]*X[0].shape[1]),np.dtype('uint8'))
for frame in X:
    temp_frame = np.empty((1,X[0].shape[0]*X[0].shape[1]),np.dtype('uint8'))
    for pixel in frame:
        np.reshape(pixel,(3,1))
        np.append(temp_frame,pixel)
    np.append(X_new,temp_frame,axis=0)
    print(temp_frame)
print(X_new.shape)
print(X_new)

#installing library necessary for hexadecimal: (FAILED - SEE BELOW)
! pip install matplotlib.colors
from matplotlib.colors import rgb2hex

##TURNING RGB TO HEX (FAILED)
#turning a 3-D matrix into a 2-D matrix (hex):
print(X.shape)
print(X[0].shape)
X_new = np.empty((frames,X[0].shape[0]*X[0].shape[1]),np.dtype('uint32')) #as hex is the hexademical system,
↪  'uint16' must be used
for frame in X:
    temp_frame = np.empty((1,X[0].sh1ape[0]*X[0].shape[1]),np.dtype('uint32'))
    for pixel in frame:
        rgb2hex(pixel/255) #due to value error: RGBA values should be within 0-1 range
        np.append(temp_frame,pixel)
    np.append(X_new,temp_frame,axis=0)
    #print(temp_frame)
print(X_new.shape)
#print(X_new)

"""
```

References

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

[2] J. Grosek and J. Nathan Kutz. "Dynamic Mode Decomposition for Real-Time Background/Foreground Separation in Video". In: (2014).

[3] Jose Nathan Kutz. *Dynamic Mode Decomposition (Chapter 4): Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013.