MATH 494 Assignment 1

Alexander Vakkas (40098027)

February 12, 2023

Abstract

The goal of this assignment is to use Dynamic Mode Decomposition (DMD) to analyze two videos and isolate static and critical information. The method we will use involves the creation of a low-rank Singular Value Decomposition (SVD), which results in a DMD, which provides us with all of the necessary information to achieve our intended goal, which is to reconstruct the frames of this video and capture the motion of images against a static background. My intended audience is my graders and my future self.

1 Introduction and Overview

Dynamic mode decomposition (DMD) is a data-driven dimensionality reduction algorithm similar to matrix factorization and principle component analysis (PCA). DMD computes a set of dynamic modes from a multivariate time series data set, each method having a fixed oscillation frequency and decay/growth rate. DMD enables the interpretation of temporal data behaviors with physically meaningful modes. DMD is capable of performing multivariate time series forecasting, which is an important feature. Among the existing work, DMD models have a wide range of applications in spatiotemporal fluid flow analysis. In this assignment, we take two videos and use DMD to separate the foreground and background, but before we discuss the procedure we used to achieve these results, some theoretical knowledge on the subject is required.

2 Theoretical Background

2.1 Singular Value Decomposition

Now before we dive further into the theoretical process of DMDs, I would like to briefly review what the DMDs are based on. The singular value decomposition (SVD), is one of the most important matrix factorizations and is the foundation for almost all data-driven methods. The SVD provides a numerically stable matrix decomposition that can be used for various purposes. The SVD can be used to obtain low-rank approximations through a process called truncation and it is also used to perform pseudo-inverses of non-square matrices to find the solution of a system of equations [1].

When it comes to SVDs, we are interested in analyzing large data sets, i.e. A. The measurements in columns could be from simulations or experiments. Columns, for example, may represent images that have been reshaped into column vectors with the same number of elements as pixels in the image [1]. The column vectors may also represent the state of a time-evolving physical system, such as fluid velocity at a set of discrete points, a set of neural measurements, or the state of a one-square-kilometer-resolution weather simulation [1]. The SVD is a one-of-a-kind matrix decomposition for any complex-valued matrix A:

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

where U and V are unitary matrices with orthonormal columns and S is a matrix with real, non-negative diagonal entries and zeros off the diagonal. We demonstrated the definition of the SVD to introduce one of the SVD's most useful and defining properties, which is to provide an optimal low-rank approximation to a

matrix A, also known as B.

One of the most important and contentious decisions when using the SVD is deciding how many singular values to keep, i.e. r. Since a rank-r approximation is obtained by keeping the first r singular values and vectors and discarding the rest. There are numerous factors to consider, including the system's desired rank, noise magnitude, and the distribution of singular values. Typically, the SVD is truncated at a rank r that captures a predetermined amount of energy from the original data, such as 90 or 99 percent. Other techniques involve locating "elbows" or "knees" in the singular value distribution that may represent the transition from singular values representing important patterns to those representing noise. Truncation can be considered a hard threshold on singular values, where values greater than a threshold are rejected [1]. There is no better approximation for A than the truncated SVD approximation B for a given rank r.

Many data sets contain high-dimensional measurements, resulting in a large data matrix A. However, low-dimensional patterns are frequently dominant in the data. The truncated SVD transforms a high-dimensional measurement space into a low-dimensional pattern space via a coordinate transformation. This has the advantage of shrinking the size and dimensions of large data sets, resulting in a more manageable foundation for visualization and analysis [1].

2.1.1 Dynamic Mode Decomposition

DMD is primarily used to forecast data in time in a computationally cheap and efficient manner, simply by performing matrix multiplication from one time to the next. Even if the underlying dynamics are nonlinear, moving forward in time is accomplished via a linear transformation. Now to better explain this we will be expanding on the SVD information shown above. This low-rank SVD approximation can then be used within the DMD. This procedure entails repeating iterations of the lower rank data matrix until a matrix Y becomes a matrix X in the future. A linear transformation is used to move from one time step to the next. As X is not a square matrix, calculating the pseudo inverse of X is required to solve this linear transformation [2].

$$Zn = \sum m = 1^{M} \left(\mu m^{n} \langle Z0, \psi m \rangle \psi m\right) \tag{2}$$

The equation above shows, that Zn is the sum of the optimal eigenvalues (r-rank), coefficients, and eigenvectors. Due to the instabilities of a dynamical system, DMD serves to stabilize the data so that it can be mapped out onto the unit circle [2]. However, some eigenvalues may appear inside or outside the unit circle, necessitating the use of instability suppression within DMD. The process concludes with the separation of the foreground and background. The initial data and reconstructed data can be compared after the iterates are calculated. While the original data is strictly limited to the real plane, the reconstructed data must include complex values.

Thus, within the separation of the foreground and background, the eigenvalues must 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, the background, as input into the low-rank matrix, the values with the highest percentage of change over time [2]. After following the theory shown in our assignment our goal is to have a matrix X shown, X=Xlowrank DMD + Xsparese DMD.

3 Algorithm Implementation and Development

3.1 Set up

For the corresponding code, please refer to Appendix B. The code is the same for both videos used in this assignment. We begin by downloading the necessary programs that we will use throughout the python code. Once they are downloaded we use the function skvideo.io.vread to extract our video data, the data is given

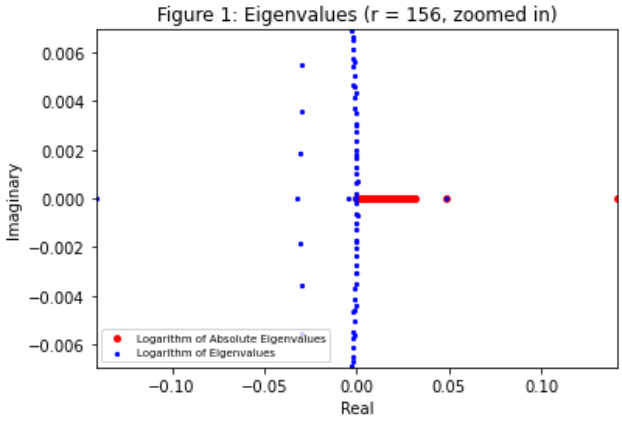

Figure 1: The plotted eigenvalues over an s-rank of 156 over the real and imaginary numbers.

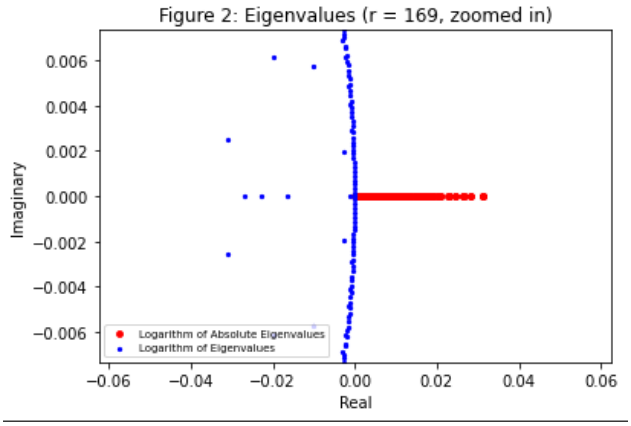

Figure 2: The plotted eigenvalues over an s-rank of 169 over the real and imaginary numbers.

as a 4D matrix. We then proceed to convert the video data into a grayscale and then reconstruct it into a 2D matrix. Now using this matrix we compute the SVD and found that python had a hard time running the function np.linalg.svd. We had no choice but to use the low-rank SVD in order to continue with the Assignment. We created a function that calculated a 0.99999 error which gave us our r value. Once that was identified we computed our low-rank SVD, and with that information, we then calculated our X and Y matrices. After computing the pseudo inverse of X, we got our DMD matrix A.

3.2 DMD Manipulation

Using our DMD matrix A, we use the function np.linalg.eig to find the eigenvalues and eigenvectors that will be used for the eigenvalue decomposition, the formula is shown on page 15 in [2]. To achieve this Zn we must first calculate our matrix of coefficients,i.e. “c”. We got this matrix c by taking the first frame of our low-rank SVD and multiplying it by the inverse eigenvector matrix. Now our Zn eigenvalue decomposition can be computed.

3.3 Video Reconstruction

After computing the eigenvalue decomposition, we plot to see the spread of eigenvalues to determine a cutoff to help calculate our fast and slow values. We calculate a Zn of fast eigenvalues to begin constructing our foreground matrix. We then get our Zn of slow eigenvalues by subtracting Zn by Zn fast. We then follow the procedure given to us in our assignment information, and we produce a reconstructed X matrix with

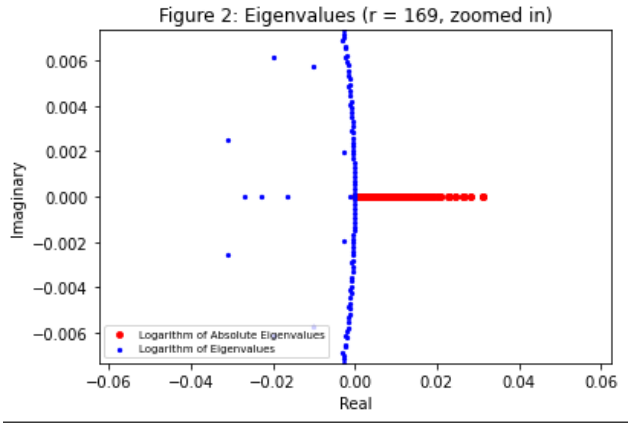

Figure 3: The plotted eigenvalues over an s-rank of 169 over the real and imaginary numbers.

two matrices, a low-rank DMD and a sparse DMD. With this achieved we print and examine our work, as shown in Appendix B.

4 Video Results

Sadly the code could not achieve a complete separation of the foreground and background. Figure 3 depicts the two separations of the Monte Carlo video, which show that the foreground played the entire video with some blur. At the same time, the background displayed a black screen throughout the video. For figure 4, the same results are reproduced for the Ski video.

5 Summary and Conclusions

Here we used Dynamic Mode decomposition to ascertain and deconstruct two videos. Our goal was to reconstruct these videos in two representing background and foreground. Our efforts were in the end not successful. Although the great majority of the code is correct. The final results seemed to be less than fruitful. Although the results were inconclusive, many hurdles had to be overcome throughout the coding process on python, and this newly acquired knowledge of the program will surely propel us forward in our future projects.

Appendix A MATLAB Functions

Add your important MATLAB functions here with a brief implementation explanation. This is how to make an **unordered** list:

- `y = linspace(x1,x2,n)` returns a row vector of `n` evenly spaced points between `x1` and `x2`.
- `[X,Y] = meshgrid(x,y)` returns 2-D grid coordinates based on the coordinates contained in the vectors `x` and `y`. X is a matrix where each row is a copy of `x`, and `Y` is a matrix where each column is a copy of `y`. The grid represented by the coordinates `X` and `Y` has `length(y)` rows and `length(x)` columns.

pip install Pygments

Appendix B Python Code

```
#downloading video data
import skvideo.io
```

```
import numpy as np
video_array = skvideo.io.vread("/content/drive/MyDrive/494 videos/monte_carlo_low (1).mp4")

#convert to grayscale to manipulate data easier
video_array_gray = skvideo.utils.rgb2gray(video_array)
print(video_array_gray)

video_array_gray.shape

#creating the data matrix
c = []
for frame in video_array_gray:
    b=frame.reshape((540*960))
    c.append(b)

print(np.shape(c))

data = np.transpose(c)

print(np.shape(data))

print(data)

#calculating SVD of data matrix
u, s, vh = np.linalg.svd(data, full_matrices=False)
u.shape, s.shape, vh.shape

#finding the energy by taking the error of 2 frobinius norms to find our low rank
#gives you optimal rank r
r=0
energy=0
while energy<= 0.99999:
  energy  = energy + (s[r]**2)/np.sum(np.square(s))
  r+=1
print(r)

#to find low rank A, we take the original matrix and multiply it by the U* of r matrix(dot product)
def lowRankSVD(R: int, U: np.ndarray, A):
  return np.dot(np.transpose(U[:,:R]),A)

A_low_rank = lowRankSVD(156, u, data)

#X is the new low rank matrix A minus its final collumn
X = np.delete(A_low_rank,-1,1)
print(X)

#Y is the new low rank matrix A minus its first collumn
Y = np.delete(A_low_rank,0,1)
print(Y)

#psuedo invers of X
psuX = np.linalg.pinv(X)
print(psuX)

#multiply the Y matrix by the psuedo inverse of X to get the DMD
DMD = np.matmul(Y,psuX)
print(DMD)

#Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(DMD)

print(eigenvalues)
print(eigenvectors)

#matrix of eignevectors * matrix of bp = first frame of low rank A (which is first collume of it) **just
↪  inverse**
#solving this to get matrix of bp
#then use to get general decomposition formula pg 22
```

```
eigvecinverse = np.linalg.inv(eigenvectors)
frame1= A_low_rank[:, 0]
print(frame1)
matrix_C = np.matmul(eigvecinverse,frame1)
print(matrix_C)
#values on the diagonal!

#Compute eigen decomposition matrix:

#Part 1 - The set up
import math

#R = 156 - optimal p-rank
#N = 379 - total frames

C = matrix_C.reshape(156,1)
E = eigenvalues.reshape(156,1)
Zn = np.empty([156,379], dtype = np.cdouble)
#Zn.reshape(59124,1)
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(E.shape)
print(C.shape)
print(eigenvectors.shape)
print(Zn.shape)
print(Zn.dtype)

# Part 2:
#using the Zn formula on page 15 of textbook [2]
#r = 0
#n = 1

for n in range(379):
    for r in range(155): #as the first row is 0
        #print(eigenvectors[:,r]) #to check whether the slicing is 'good'
        #using [r+1] as it is the only way to get around ValueError: could not broadcast input array from shape
        ↪  (156,156) into shape (156,)
        decomp = decomp + (((E[r+1])**(n+1))*(C[r+1])*eigenvectors[:,[r+1]])
    Zn[:,[n]] = decomp
    #to initialize next column of Zn
    decomp = np.empty([156,1])

np.save('Zn.npy', Zn)

print(Zn.shape)
print(Zn)


#importing necessary package for plotting:

import matplotlib.pyplot as plt
from matplotlib.pyplot import plot
from matplotlib.pyplot import margins
#Determining the cutoff of eigevalues

#frames/second
delta_t = 379/6

omega = np.abs(np.log(E))/delta_t
omega_prime = np.log(E)/delta_t

print(omega.shape)
print(omega)
```

```
print(omega_prime.shape)
print(omega_prime)

#Plotting the eigenvalues without the cutoff
#This is to observe what we will be cutting off

plt.scatter(omega.real, omega.imag, marker='o', c = 'red', s = 15, label = 'Logarithm of Absolute
↪  Eigenvalues')
plt.scatter(omega_prime.real, omega_prime.imag, marker='o', c = 'blue', s = 5, label = 'Logarithm of
↪  Eigenvalues')

#This line controls the zooming
plt.margins(x=0,y=-0.25)

plt.title("Figure 1: Eigenvalues (r = 156, zoomed in)")
plt.xlabel("Real")
plt.ylabel("Imaginary")

plt.legend(loc='lower left',prop={'size':7})

plt.show()

#Cutting off the eigenvalue:
#The cut off is at 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:
        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 9th element in omega")
#this means that the element with index 8 in the coefficient matrix is the coefficient for the slow
↪  eigendecomposition

#Reconstruction the foreground and background

#Part 1:The Foreground = X_fast

import copy
fast_C = copy.deepcopy(C)
fast_C[8] = 0

print(fast_C)
print(fast_C[8])
print(C[8])
```

```
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 r in range(155):
        #using [r+1] as it is the only way to get around ValueError: could not broadcast input array from shape
        ↪  (156,156) into shape (156,)
        decomp_new = decomp_new + (((E[r+1])**(n+1))*(fast_C[r+1])*eigenvectors[:,[r+1]])
    Zn_fast[:,[n]] = decomp_new
    #to initialize next column of Zn_fast
    decomp_new = np.empty([156,1])

print(Zn.shape)
print(Zn_fast.shape)
print(Zn_fast)

#Part 2: The Background = X_slow

Zn_slow = np.subtract(Zn, Zn_fast)

print(Zn_slow.shape)
print(Zn_slow)

##CONSTRUCTING THE R MATRIX

#Part 1:

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)

#Part 2:

#print(data)
sparse = np.subtract(data,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)

#as described in the methodology
final_sparse = np.subtract(sparse, R)

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(data.shape)
print(data)

RECON_VIDEO = np.add(lowrank, final_sparse)

print(RECON_VIDEO.shape)
print(RECON_VIDEO)

from PIL import Image as im
#Frame Images of reconstructed video

RECON_IM1 = RECON_VIDEO[:,[370]]
RECON_IM1.shape
RECON_IM1 = RECON_IM1.reshape(540,960)
#RECON_IM1 = RECON_IM1.astype('uint8')

IM1 = im.fromarray(RECON_IM1)
IM1 = IM1.convert("L") #solves OSError: cannot write mode F as PNG

IM1.save("IM1.png")

#preparing slow and fast dmd
Lowrank = np.transpose(lowrank)
Lowrank.shape
Final_sparse = np.transpose(final_sparse)

#Video save and download method
foreground = np.reshape(Final_sparse,(379, 540, 960, 1))
foreground = foreground.astype(np.uint8)

skvideo.io.vwrite("foreground.mp4", foreground)

background = np.reshape(Lowrank,(379, 540, 960, 1))
background = background.astype(np.uint8)

skvideo.io.vwrite("background.mp4", background)
#END OF CODE
```