Video Separation with DMD Modes

AMATH582: Homework 5
Maple Tan

March 15, 2019

1 Abstract

Using Dynamic Mode Decomposition (DMD), we will take a video clip containing a foreground and background object and separate the two objects into two different video streams, a foreground one and a background one.

2 Introduction and Overview

We begin with a set of different videos, each a couple of seconds long. We will look at two different cases of videos: one that has a foreground image moving against a static background and the other that has a foreground image moving against a moving background. Loading the data, we then reshape each frame of the video into a column vector and store them into a large matrix. Applying DMD to this large matrix, we can then use the DMD modes to separate the foreground and background objects.

3 Theoretical Background

The Dynamic Mode Decomposition can extract the dominant modes of our data. Suppose we have a vector of data that was collected from a dynamical system as follows:

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t, \mu)$$

Where $\vec{x}$ is a vector representing a state of our dynamical system at a time t, with a vector of parameters μ. The goal of DMD is to find the governing equations f that would generate our data $\frac{d\vec{x}}{dt}$. There is no easy way to explicitly solve for the governing equation, but we can construct a linear system that could approximate the system as follows:

$$\frac{d\vec{x}}{dt} = A\vec{x}$$

This system has a well known solution because this is an eigenvalue problem:

$$\vec{x}(t) = \sum_{k=1}^{n} \vec{\Phi}_k \exp(\omega_k t) b_k = \vec{\Phi} \exp(\Omega t)\vec{b}$$

Where $\vec{\Phi}_k$ and ω_k are the eigenvectors and eigenvalues of the matrix A respectively. The main question now is how to define A. On a continuous time range, we can sample at every $\triangle t$ to create a discrete time system. We will describe the state $\vec{x}$ at time $t = 0$ as $\vec{x}_0$ and $\vec{x}_1$ at time $t = 1$ and so on. So in other words, we have a dynamical system as follows:

$$\vec{x}_{k+1} = A\vec{x}_k$$

With an initial condition $\vec{x}_0$. So in other words, this matrix A will take the our state vector x one time t to the next time $t + \triangle t$. Using this equation, we can define our A matrix as follows:

$$A = \vec{x}_{k+1}\vec{x}_k^{(-1)}$$

However, this applies only to single vectors. For application to a data matrix, suppose we have a $m \times n$ matrix where m represents the number of measurements we make at time t while n represents the number of times we make a measurement in an experiment. We can define our initial state matrix called $\mathbb{X}_k$ where the columns are going from $\vec{x_0}$ to $\vec{x_{n-1}}$. Then for our next state matrix x_{k+1}'s will be a matrix called $\mathbb{X}_{k+1}$ whose columns are going from $\vec{x_1}$ to $\vec{x_n}$. Thus A in this case would be matrix that takes $\mathbb{X}_k$ to $\mathbb{X}_{k+1}$ and we would solve for A in the following manner:

$$A = \mathbb{X}_{k+1}\mathbb{X}_k^{-1}$$

One thing to note is that our $\mathbb{X}$ matrices may already start out large, which causes A to be a even larger matrix. Since A is such a huge matrix, we do not want to work directly with A. As such, it is more efficient to compute the $r \times r$ POD projection of that full A matrix, which we will call $\tilde{A}$. We start by applying the SVD to $\mathbb{X}_k$, we want to reconstruct the r-ranked version of our original matrix:

$$\mathbb{X}_k = U_r\Sigma_r V_r^* \Rightarrow U_r = \mathbb{X}_k V_r \Sigma_r^{-1}$$

Then inputting it into our original definition of A:

$$\begin{aligned} A &= \mathbb{X}_{k+1}\mathbb{X}_k^{-1} \\ &= \mathbb{X}_{k+1}(U_r\Sigma_r V_r^*)^{-1} \\ &= \mathbb{X}_{k+1}V_r\Sigma_r^{-1}U_r^* \end{aligned}$$

Then we have the $r \times r$ POD projection of A, defined as $\tilde{A}$ as follows:

$$\begin{aligned} \tilde{A} &= U_r A U_r^* \\ &= U_r^*(\mathbb{X}_{k+1}V_r\Sigma_r^{-1}U_r)U_r^* \\ &= U_r^*\mathbb{X}_{k+1}V_r\Sigma_r^{-1} \end{aligned}$$

Now that we have the our $\tilde{A}$ defined, we next will have to compute the eigendecomposition of $\tilde{A}$ for our solution:

$$\tilde{A}W = W\Lambda$$

Where W is the matrix of eigenvectors and Λ is a diagonal matrix with the corresponding eigenvalues. Next we then can use that to reconstruct A with our eigendecomposition, and obtain our DMD modes, Φ:

$$\Phi = \mathbb{X}_{k+1} V \Sigma^{-1} W$$

Where Φ (our DMD modes) are the eigenvectors of A. Then we must get ourselves back to our high dimmensional space. Reminder that:

$$\vec{x}(t) = \Phi e^{\omega t} b$$

So to solve for b, we have:

$$\begin{aligned} x(0) &= \Phi e^{\omega(t)} \vec{b} \\ x(0) &= \Phi \vec{b} \\ \Rightarrow b &= \Phi \ x(0) \end{aligned}$$

Then using this b, we can reconstruct our x vector and then write:

$$\vec{x}(t) = \sum_{k=1}^{r} \Phi_k \exp(\omega_k t_b) k = \Phi \exp(\omega t) \vec{b}$$

Where

$$\Phi = \mathbb{X}' V \Sigma^{-1} W$$

In this case, when it comes to image separation, we are assuming that there exists ω_p where $p \in \{1, 2, \ldots, l\}$ satisifies $||\omega_p|| \approx 0$ for all p and that there are ω_j for $j \neq p$ where $||\omega_p||$ are bounded away from 0. This means that our X_{DMD} is as follows:

$$X_{DMD} = b_p \phi_p e^{\omega_p t} + \sum_{j \neq p} b_j \phi_j e^{\omega_j t}$$

We then see that the first term corresponds with the background video and the second term corresponds with the foreground video. We first note that our original video X would theoretically be:

$$X = X_{BG} + X_{FG}$$

Where X_{BG} and X_{FG} correspond with the background and foreground respectively. Using a low rank reconstruction of our X_{DMD}, it follows that:

$$X_{BG} = X_{\text{Low Rank DMD}} = b_p \phi_p e^{\omega_p t}$$

Which corresponds with our background. Then to find our our foreground, we simply subtract the absolute value (since image values should only be positive) of X_{BG} from our original frames.

$$X_{FG} = X_{\text{Sparse DMD}} = X - |X_{BG}|$$

However note that since $|\cdot|$ yields the modulus of each element, this results in X_{FG} having some negative pixel intensities, which does not make sense. As such, we need to create a residuals matrix that can be put back into X_{BG} so that the magnitude of the imaginary values are accounted for.

4 Algorithm Implementation and Development

We start by importing our libraries and video data. These include: `numpy as np`, `imageio`, `matplotlib.pyplot as plt`, and `numpy.linalg as la` and `color` from `skimage`. We also have two different videos: one is a video of emperor penguins walking across the screen and the other is spider man jumping off a building with the cityscape in the background from *Into The Spiderverse*. The main difference between the two is that the background remains static in the penguin video while the cityscape also moves with spiderman as he jumps off the building.

Figure 1: Before and after removing the black bars of the video.

To prepare our data, we use the `get_reader` method from the `imageio` library to acquire all the frames from our video clip. To begin, we first extract the dimensions of our video using `np.shape`. Since we extracted our video off YouTube, we also arbitrarily crop the image to get rid of the black bars that frame the video. We also update our dimensions accordingly, as seen in the above figure.

Using this frame, we will then use `np.reshape` to reshape each of the pictures into a column vector and store the column into a large matrix. We apply the SVD to our large matrix and plot the singular values to figure out the appropriate rank needed for reconstruction. After reconstruction, we create our Φ matrix from the eigendecomposition of $\tilde{A}$ and then use that and our first frame to find our b vector. We next use a for loop to obtain the time dynamics at each value of t, which is the length of our constructed data matrix times $1/dt$, to get the time length of our video. In this case dt is the recorded frames per second (60 in this case). Once we have our time dynamics, we can multiply it by our Φ matrix to obtain our low rank DMD X.

From there, we can subtract this matrix from our original matrix to get our sparse DMD X. Then extracting the columns from both our $X_{\text{Low Rank DMD}}$ and $X_{\text{Sparse DMD}}$, reshaping back into a frame, we then plot the frames. We also invert the colors of our sparse matrix to get a better image of our foreground. Now we see the separate background and foreground video streams of our original video, with varying levels of success.

5 Summary and Conclusion

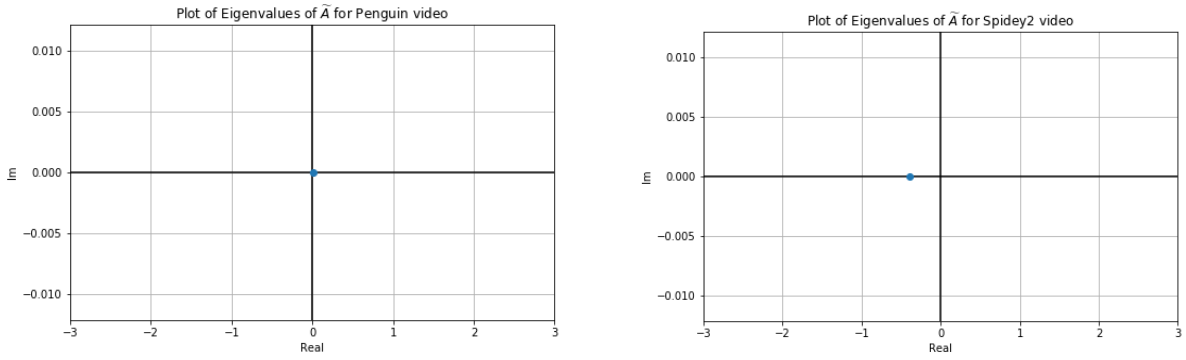

Figure 2: Eigenvalues graphed on real and imaginary axis. In the case the penguin video, we see that our eigenvalue is much close to zero while for the spider man video, we see that it is close but still has a substantial real part.

Figure 3: Two different frames in the two videos. Notice how the frame of reference remains the same in the penguin video but it changes a lot in the spiderman video.

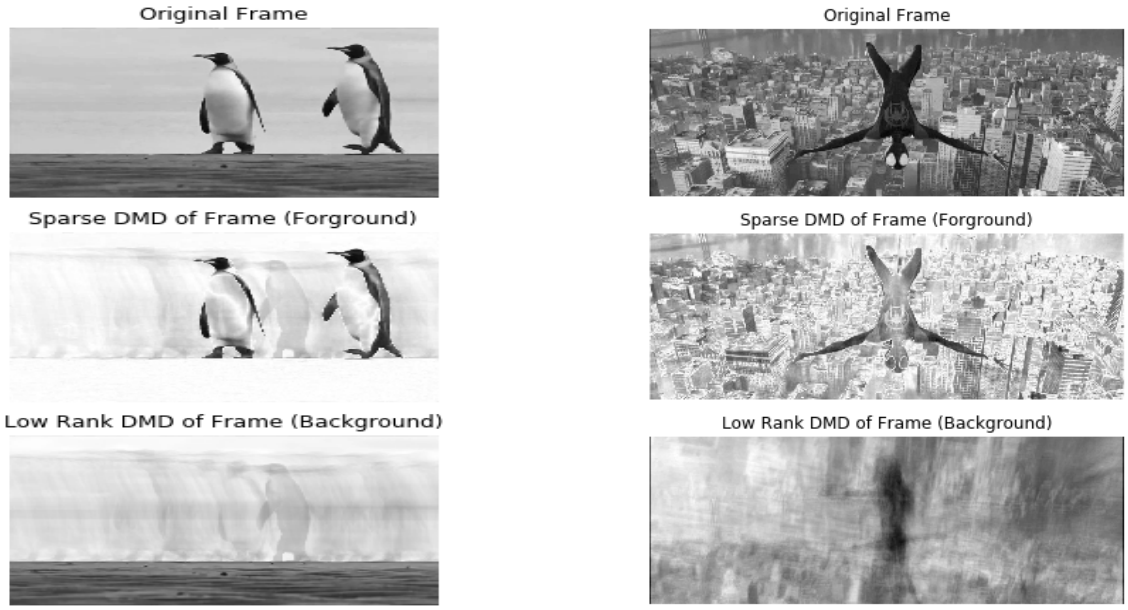

Figure 4: Here we can compare how well DMD worked on the two different videos. We can clearly see that using DMD works much better with the penguin video compared to the spiderman video. On top of the background of the penguin video being much simpler, this is due to how DMD depends on the differences between frames so it has a harder time extracting the foreground and background when the background is also changing.

A Code

```
import imageio
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
from skimage import color

# %% Load data
# Necessary File Names and Variables for Different Trials
# stuff = ['penguin.mp4', 'varp.png', ...
 'evalsp.png', 'penguin1.png', 'framediffp.png', 60]
stuff = ['spidey2.wmv', 'vars.png',...
 'evalss.png', 'spidey1.png', 'framediffs.png', 120]
file = stuff[0]
vid = imageio.get_reader(file, 'ffmpeg')
dt = 1/60       # 1/Framerate
k = 0
for n, frame in enumerate(vid):
    k = k+1
hei = np.shape(frame)[0]
wid = np.shape(frame)[1]
croph = stuff[5]
cropw = 0
hnew=(hei-2*croph)
wnew = (wid-2*cropw)
hei = hnew
wid = wnew
X = np.zeros([hei*wid, k])

# Reshape and Store into Matrix
for n, frame in enumerate(vid):
    f = color.rgb2gray(frame)
    fc = f[croph:-croph,:]
    X[:,n] = np.reshape(fc, [hei*wid, 1])[:,0]

plt.imshow(fc, cmap='gray')
plt.show()

#%% Preview Frames as necessary
```

```
#plt.figure(figsize=(5,4))
#plt.imshow(color.rgb2gray(frame), cmap='gray')
#plt.axis('off')
#plt.savefig('uncropped.png')
#
#
##%%
#plt.figure(figsize=(5,4))
#plt.imshow(np.reshape(X[:,6], [hei,wid]),cmap='gray')
#plt.axis('off')
#plt.savefig('cropped.png')


#%%
endp = np.shape(X)[1]
X1 = X[:, np.arange(0, endp - 1)]
X2 = X[:, np.arange(1, endp)]

#%%
u,s,vh = np.linalg.svd(X1, full_matrices=False)
v = np.transpose(vh)


#%% Plot Singular Values
sl = np.shape(s)[0]
plt.scatter(np.arange(0,sl), s**2 / sum(s**2), color = 'red')
plt.xlabel('Principal Component')
plt.ylabel('Variance')
plt.title('Amount of Variance in each Principal Component')
plt.ylim([0, 1])
plt.savefig(stuff[1])
plt.show()


# %%
r=1
u_r = u[:, 0:r]
s_r = np.diag(s[0:r])
v_r = v[:, 0:r]
Atilde = np.transpose(u_r)@X2@v_r@la.inv(s_r)
```

```
# %%
w, d = la.eig(Atilde)
Phi = X2@v_r@la.inv(s_r)@d

# %% Plot Eigenvalues
omega = np.log(w)/dt
x = np.real(omega)
y = np.imag(omega)
plt.figure(figsize=(8,5))
plt.axhline(color='black')
plt.axvline(color='black')
plt.title('Plot of Eigenvalues of $\widetilde{A}$ for ' + ...
stuff[0][0].upper() + stuff[0][1:-4] + ' video')
plt.xlabel('Real')
plt.ylabel('Im')
plt.scatter(x,y, zorder=3)
plt.xlim([-3, 3])
plt.grid()
plt.savefig(stuff[2])

#%% Solve for b
x0 = X1[:,0:1]
b = la.lstsq(Phi, x0, rcond=None)[0]

# %% Create DMD Modes
mm1 = np.shape(X1)[1]
time_dyn = np.zeros([r, mm1], dtype=complex)
t = np.arange(0, mm1)*dt
for tt in np.arange(len(t)):
    time_dyn[:, tt] = (b[:,0]*np.exp(omega*t[tt]))


Xdmd = Phi@time_dyn

#%%
res = Xdmd - abs(Xdmd)
X_sparse = X1 - abs(Xdmd)
```

```
#%% Plot Forground and Background Frames against Original
n = 50
frp_act = np.reshape(X[:, n], [hei,wid])
frp_sparse = np.reshape(Xdmd[:, n], [hei,wid])
frp_dmd = np.reshape(X_sparse[:, n], [hei,wid])
frp_res = np.reshape(res[:, n], [hei,wid])

fig,ax = plt.subplots(3,1, figsize=(6,8))
ax[0].imshow(frp_act, cmap='gray')
ax[0].axis('off')
ax[0].set_title('Original Frame')
ax[2].imshow(abs(frp_sparse - frp_res), cmap='gray')
ax[2].axis('off')
ax[2].set_title('Low Rank DMD of Frame (Background)')
ax[1].imshow(255-abs(frp_dmd), cmap='gray')
ax[1].axis('off')
ax[1].set_title('Sparse DMD of Frame (Forground)')
plt.savefig(stuff[3])
plt.show()

#%% Plot Different Frames
n = 40
frp_beg = np.reshape(X[:,4], [hei,wid])
frp_end = np.reshape(X[:,n], [hei, wid])

fig,axs = plt.subplots(2,1)
axs[0].imshow(frp_beg, cmap='gray')
axs[0].axis('off')
axs[0].set_title('Frame 4')
axs[1].imshow(frp_end, cmap='gray')
axs[1].axis('off')
axs[1].set_title('Frame ' + str(n))
plt.savefig(stuff[4])
```