AMATH 482
Dynamic Mode Decomposition, Background Subtraction

Oliver Speltz

March 15, 2019

I Introduction

In this report I will make use of technique called Dynamic Mode Decomposition to analyze video data. Dynamic Mode Decomposition is a method to approximate a dynamical (varying in time) system where the underlying governing equations are unknown as a linear system of ordinary differential equations. Here I make use of the Dynamic Mode Decomposition to attempt to separate the background and foreground of various videos.

II Theoretical Background

Dynamic Mode Decomposition (DMD) is a technique that takes a dynamical system with many features or dimensions, and approximates it as a linear system. It falls under the assumption that the system being measured can be described by some system of differential equations, given generally by:

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = f(\vec{x}, t, \mu) \tag{1}$$

Where $\vec{x} \in \mathbb{C}^n$ are the features or dimensions of the system, typically with $n >> 1$, t is time, and μ is some collection of parameters. The catch is, we don't know f, the governing equations of the system. DMD gives us a way to describe and even predict this system, despite not knowing the governing equations. A strategy to do this would be very relevant in many areas with a lot of data but aren't well understood, such as neuroscience, turbulence, atmospheric science, etc.

Given m snapshots in time of an n dimensional system, the goal of DMD is to find the best $\mathbf{A}$ such that

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \mathbf{A}\vec{x} \tag{2}$$

Then the solution in time $\vec{x}(t)$ can be written as expansion in an eigenvector basis of $\mathbf{A}$,

$$\vec{x}(t) = \sum_{j=1}^{n} b_j \vec{\phi}_j e^{\lambda_j t} \tag{3}$$

Where $\vec{\phi}_j$ and λ_j are the eigenvectors and corresponding eigenvalues of $\mathbf{A}$ and b_j is the projection of the initial condition in the eigenvector basis. Discretely, we can represent our time shots as a $n \times (m-1)$ matrices $\mathbf{X}$ and $\mathbf{X}'$ such that that,

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 & \vec{x}_2 & \dots & \vec{x}_{m-1} \end{bmatrix} \quad \mathbf{X}' = \begin{bmatrix} \vec{x}_2 & \vec{x}_3 & \dots & \vec{x}_m \end{bmatrix} \quad \text{and} \quad \mathbf{X}' = \mathbf{A}\mathbf{X} \tag{4}$$

Figure 1: Stills from the three test videos.

This can solved by $A = \mathbf{X}'\mathbf{X}^\dagger$ where $\mathbf{X}^\dagger$ is the Moore-Penrose pseudo-inverse of $\mathbf{X}$. However, the problem then becomes huge expense of this operation. If n is large, $\mathbf{A}$ is $n \times n$ and hard to compute. Luckily, we can use the good ol' Singular Value Decomposition (SVD) to find a low rank approximation of $\mathbf{A}$. If we take the SVD of $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$, and $\mathbf{U}_r, \mathbf{\Sigma}_r,$ and $\mathbf{V}_r^*$ represent a rank r truncation of each, then

$$\begin{aligned}
\mathbf{X} &= \mathbf{X}'\mathbf{X}^\dagger \\
&= \mathbf{X}'\mathbf{V}_r\mathbf{\Sigma}_r^{-1}rU_R^* \qquad \mathbf{X}^\dagger = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*)^{-1} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^* \\
\tilde{\mathbf{A}} &= \mathbf{U}_r^*\mathbf{A}\mathbf{U}_r \\
&= \mathbf{U}_r^*\mathbf{X}'\mathbf{V}_r\mathbf{\Sigma}_r^{-1}
\end{aligned}$$

If we take an eigen-decomposition of this reduced $\tilde{\mathbf{A}}$, we can get an approximation of the eigenvectors and eigenvalues of $\mathbf{A}$ like,

$$\mathbf{\Phi} = \mathbf{X}'\mathbf{V}_r\mathbf{\Sigma}_r\mathbf{W} \tag{5}$$

Where $\mathbf{W}$ is a matrix of the eigenvectors of $\tilde{\mathbf{A}}$. Now $\mathbf{\Phi}$ is $n \times r$ matrix representing the *DMD modes* of $\mathbf{X}$. Now the dynamics of the system can be approximated as

$$\vec{x}(t) = \sum_{j=1}^{r} b_j \vec{\phi}_j e^{\omega_j t}$$

Where $\omega_j = \ln(\lambda_j)/\Delta t$, with λ_j being the eigenvalues of $\tilde{\mathbf{A}}$ and Δt being the time between snapshots. This is necessary for the discretization of the system.

Now with this algorithm in hand, we can make predictions of future times for a complex system where we do not know the dynamics. However, since this is a linear approximation, it will not be accurate for long into the future, especially for dynamics that are inherently nonlinear. Fortunately, the DMD modes are not expensive to compute, so for systems that are have data constantly coming in, the DMD modes can be continuously updated to more accurately track the system.

III Algorithm Implementation and Development

In this assignment, I will use the DMD to take a video with a stationary background and a moving subject in the foreground and subtract the background. For videos, I used my phone to record one video with a perfectly stationary background and one video with some motion in the background. After reading the mp4 files into python using `imageio.get_reader` and resizing them from 1920 by 1080 to 384 by 216, I arranged the flattened frames as columns into a tall skinny data matrix. Some stills from the videos can be seen in Figure 1, in one some weird kid rolls on a long board in a bike shop, in another, a talented young man rides a unicycle with some people walking in the background, in the last one, cars drive on a freeway. Figure 2 shows the normalized singular values and first 4 principle components of one of the videos. There is one dominant singular value which is associated with a principle component that is essentially the background. The other

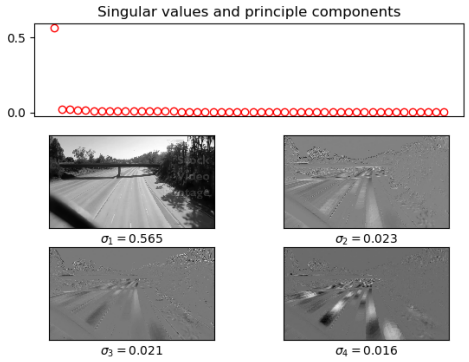

Figure 2: There is an obvious dominant singular value, which is associated with the background. The moving parts of the video are in the subsequent principle components.

videos had similar results from the SVD. Because the SVD alone seems to separate the moving from the still components quite well on its own, I will be comparing using the DMD to subtract the background to just using the SVD. To subtract the background using the SVD, I created a rank 1 approximation of the matrix $\mathbf{X}$ using only the first principal component, that is $\mathbf{X}_{bg} = \sigma_1 \vec{u}\vec{v}^*$, then subtracted this background component from all the frames.

To subtract the background using the DMD, I first followed the algorithm outlined in the previous section. I used a rank 10 truncation of the SVD of $\mathbf{X}$. Then the background components can be considered to be the DMD modes associated with ω_j values that are close to 0. This is because ω_j is the dynamics in time of that mode, if ω_j is close to 0, that mode will not be changing much in time, i.e. it will mostly likely be background. I decided to use a threshold of 0.01 for a mode to be considered background.

However, since I am dealing with pixel data, I must be careful with the negative or complex values that these algorithms may produce. In the SVD case, subtracting the background will result in negative values, luckily `pyplot.imshow` scales every values to a 0 to 255 scale to display an image. In the DMD case, the background mode to be subtracted from all frames will be complex, so I subtracted the absolute value of the background modes. If one were interested in saving these separated images as image files, the values would have to be rescaled to 0 to 255 scale. If one wanted to be able to add together a foreground matrix and a background matrix to get the original data, you would have to keep track of the complex residuals as well.

IV Computational Results

I found, that for my videos the SVD background subtraction works better or the same as the DMD background subtraction. See Figures 3, 4, 5 for a side by comparison of a few frames from each test video.

Figure 3 has possibly the worst separation through both techniques. I think this is because the strange man I recorded was moving rather slowly in the video, causing a blur of him to show up as "background". He also hit his head on one of the bikes hanging up, causing it to move and show up in the foreground. The DMD and SVD techniques seem to produce very similar results for this case.

Figure 4 has better separation. This video was filmed outside, so there is some motion in the sky and the plants that corrupt the background and foreground, as well as people walking in the background. In this test, the SVD and DMD techniques are very similar.

Figure 5 has the best separation of all. The main details that show up erroneously in the foreground are the edges of the trees in the background that are swaying in the breeze. In this case, the SVD technique performs quite better than the DMD technique. Upon investigation, it seems that two DMD modes were

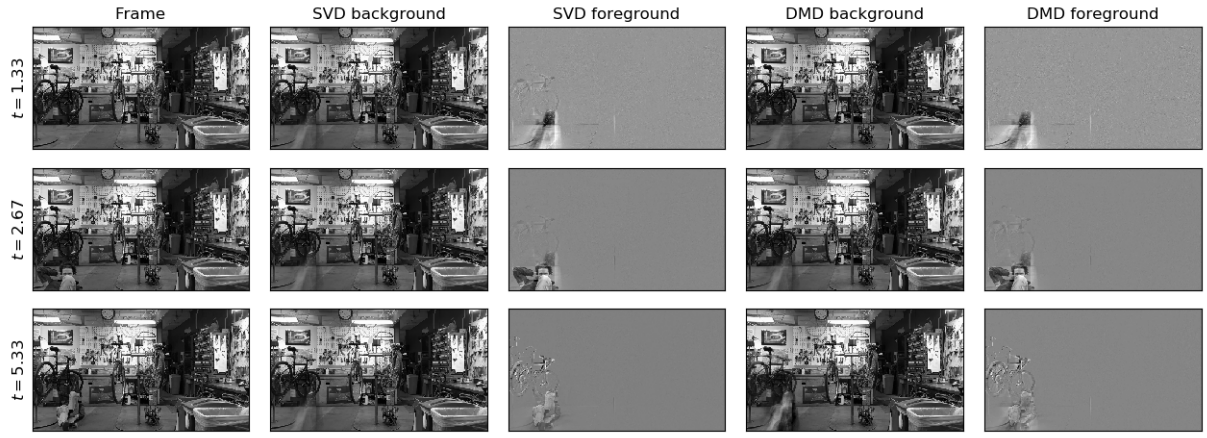

Figure 3: The weirdo in a bike shop.

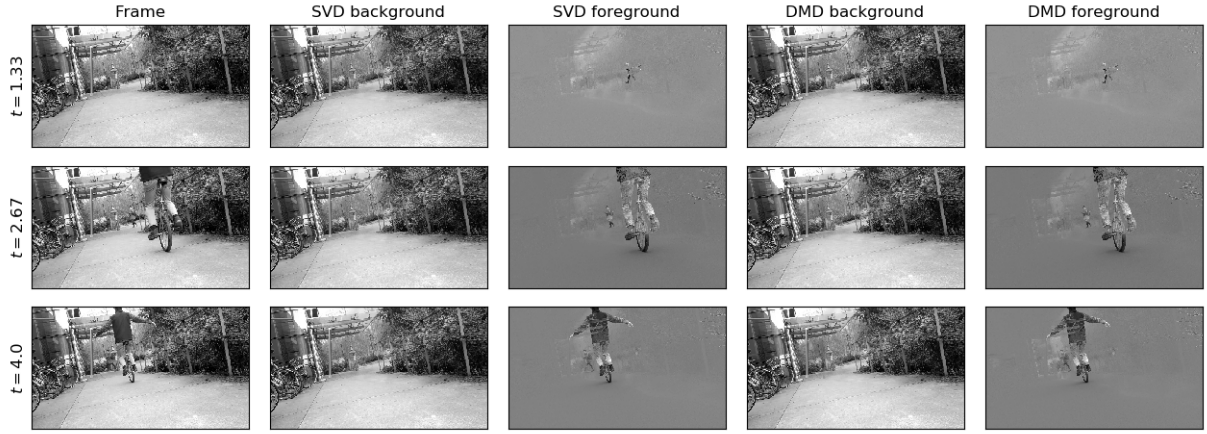

Figure 4: The handsome unicyclist.

included in the background creation. If I made it so that only mode was used, it again produced similar results to the SVD method.

Interestingly, the DMD method can be used to predict the "future" of the videos. One might expect in the future to see the weird kid on a skateboard homeless and lonely, the man on a unicycle winning some award, the cars on the freeway all driverless or flying. However when we use the DMD to predict this system into the future, only the background remains. Deep. Is the DMD trying to tell us about the fragility and impermanence of human existence? When you go even further in the future, not even the background remains, is this perhaps an allegory to the heat death of the universe when entropy will consume everything? Sadly not, it is just a mathematical algorithm and cannot make metaphors. This is just because the real parts of ω all are negative, so $e^{\omega_j t}$, the scaling of each DMD mode, will eventually go to zero.

V Summary and Conclusions

In conclusion, although the DMD did not turn out to be the best method to remove the background from my videos, it is a powerful technique in many other applications. I could imagine that it would work much at removing the background in a video where the background *is* moving, such as a slow stable pan of a scene

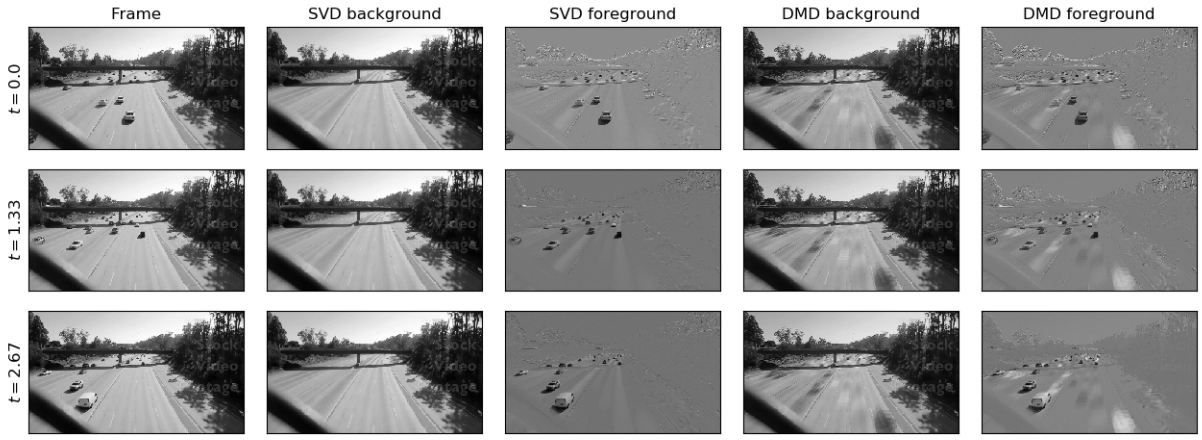

Figure 5: Cars on a freeway.

Figure 6: The sad future of the system.

with things moving in the foreground. In this case, the motion of the foreground would hopefully be of a higher frequency than the slow frequency of the pan, and the DMD would be able to differentiate between them. The SVD method in this situation would fail spectacularly. Applied to other situations, the DMD can be a powerful tool in prediction and analysis.

Appendix A: Python Functions Used

`imageio.get_reader(path,method)`
: Returns an object which can iterate over the frames of the video found at `path`. Returns the frames as `numpy` arrays. Uses the implementation of `method`, usually `method = 'ffmpeg'`.

`numpy.linalg.svd(a,full_matrices)`
: The SVD algorithm built into `numpy`'s linear algebra submodule. Returns the U, Σ and V^* of the SVD of `a` as `numpy` arrays, the Σ returned is not actually a diagonal matrix, but rather a 1D array of the diagonal elements. `full_matrices` is a boolean that if true the function returns the full SVD, if false, it returns the reduced SVD.

`numpy.linalg.eig(A)`
: Returns `lam` and `w`, the eigenvalues of `A`, unsorted, and the eigenvectors in 2d array such that `w[:,i]` is the ith eigenvector.

Appendix B: Main Python Code

```
import numpy as np
import matplotlib.pyplot as plt
import imageio as imio
from PIL import Image
la = np.linalg
plt.rcParams['image.cmap'] = 'gray'
# (height, width)
scale = 2
imsize = (scale*108,scale*192)
fps = 30

def load_vid(test,rotate=False):
    '''loads the video and saves it as a 2d np array, each frame
    flattened as a column'''
    path = 'vids/' + test
    vid = imio.get_reader(path + '.mp4','ffmpeg')
    X = []
    for frame in vid:
        im = Image.fromarray(frame)
        if rotate:
            im = im.rotate(90,expand=True)
        im = im.convert('L') # grayscale
        im = im.resize((imsize[1],imsize[0]))
        X.append(np.array(im).flatten())
    vid.close()
    # frames in the columns
    X = np.array(X).T
    np.save(path,X)
    return X

def imshow(im,ax):
    if len(im.shape) < 2:
        im = im.reshape(imsize)
    ax.imshow(im)
    ax.set_xticks([]),ax.set_yticks([])
```

```
test = 'test6'
X = load_vid(test)

u,s,v = la.svd(X,full_matrices=False)
# show principal components of the video
fig = plt.figure()
ax = fig.add_subplot(311)
ax.plot((s/s.sum())[:50],'ro',fillstyle='none')
ax.set_xticks([]); ax.set_yticks([0,0.5])
ax.set_title('Singular values and principle components')
axes = [fig.add_subplot(3,2,i) for i in range(3,7)]
for i in range(4):
    imshow(-1*u[:,i],axes[i]) # PCs are inverted
    sig = np.round(s[i]/s.sum(),3)
    axes[i].set_xlabel('$\sigma_{} = {}$'.format(i+1,sig))

# subtract first principle component from
# each frame
ubg = s[0] * u[:,0:1] @ v[0:1,:]
Xfg1 = X - ubg

# dmd
X1 = X[:,:-1]; X2 = X[:,1:]
U,S,V = la.svd(X1,full_matrices=False)
r = 100
Ur = U[:,:r]; Sr = S[:r]; Vr = V[:r,:]
A_til = Ur.T @ X2 @ Vr.T @ np.diag(1/Sr)
lam,W = la.eig(A_til)
Phi = X2 @ Vr.T @ np.diag(1/Sr) @ W
omega = np.log(lam)

# threshold for ln of DMD frequency to be
# considered part of the background
thresh = 0.01
# this command returns a tuple, the first value
# are the indices we need
bg_i = np.nonzero(np.abs(omega) < thresh)[0]

x0 = X[:,0]
Phibg = Phi[:,bg_i]
# initial DMD projection of background
bg0 = la.pinv(Phibg) @ x0

# it needs to hold complex values, so that must
# be declared off the bat
bg_modes = np.zeros((len(bg_i),X.shape[1]),dtype=complex)
# DMD projection of background for each frame
for i in range(X.shape[1]):
    bg_modes[:,i] = bg0 * np.exp(omega[bg_i]*i)
Xbg = Phibg @ bg_modes
Xfg2 = X - np.abs(Xbg)
```

```
fig,axes = plt.subplots(3,5)
titles = ['Frame','SVD background','SVD foreground',
          'DMD background','DMD foreground']
for i in range(5):
    axes[0,i].set_title(titles[i])
for i,frame in enumerate([40,80,160]):
    imshow(X[:,frame],axes[i,0])
    imshow(ubg[:,frame],axes[i,1])
    imshow(Xfg1[:,frame],axes[i,2])
    imshow(np.abs(Xbg)[:,frame],axes[i,3])
    imshow(Xfg2[:,frame],axes[i,4])
    t = np.round(frame/fps,2)
    axes[i,0].set_ylabel('$t = {}$'.format(t),
        fontsize=12)

# predict "future" of video
t = [100,100000,1000000000]
y0 = la.pinv(Phi) @ x0
modes = np.zeros((r,len(t)),dtype=complex)
for i in range(len(t)):
    modes[:,i] = y0 * np.exp(omega*t[i])
X_future = np.abs( Phi @ modes )
fig,axes = plt.subplots(1,len(t))
for i in range(len(t)):
    imshow(X_future[:,i],axes[i])
    time = np.round(t[i]/fps,1)
    axes[i].set_title('$t = {} s$'.format(time))
```