Dynamic Mode Decomposition for Background Removal in Video Streams

Eli Miller

March 15, 2019

Abstract

In this analysis, we explore the Dynamic Mode Decomposition (DMD) theory, algorithmic implementation, and modification for applications in video background removal. We explore the use of SVD rank truncation in the DMD algorithm to decrease computation time and represent our data in a low-dimensional subspace. We discuss different examples of background removal and the specific details of implementation.

1 Introduction and Overview

DMD aims to take data from a high-dimensional dynamic system and use data-driven techniques to create a linear transform that approximates our spatial-temporal system. Furthermore, we can make use of SVD low-dimension sub spaces in order to reduce the dimensional of our model and remove redundancy in the DMD modes. While the exact DMD can be computed in one step, it is unfeasible to implement in large systems, as it requires calculating a pseudo-inverse of all of our data at each time point. As a result, we implement an algorithm that used singular value decomposition to reconstruct our linear transform. In addition, this algorithm allows us to segment our DMD into two components: low-rank DMD and sparse DMD. This is useful for applications in video processing. By taking modes of our system with eigenvalues near 0, we are able to pull out the ”static” components of our system's discription by the DMD. In videos, these modes correspond to the background and are able to be separated from the rest of the video. We can then extract the components corresponding to the remaining modes as shown by Grosek and Kutz to properly reconstruct our system in pixel-space such that our two components sum to create the original image.

2 Theoretical Background

Let's assume we have some dynamic system that takes the form

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

We can approximate this system with a linear operator

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x} \tag{2}$$

and the solutions of a linear system of equations are well known and expressed as

$$\mathbf{x} = \sum_{j=1}^{n} b_j \phi_j e^{\lambda_j t} \tag{3}$$

where ϕ represents an eigenvalue of the system A, and λ is the corresponding eigenvalue.

Our goal then becomes finding some data-driven method for converting our non-linear system $\mathbf{f}$ into some linear system of differential equations A. Thus, we implement Dynamic Mode Decomposition. Say we have a collection of measurements of our system

$$X = \begin{bmatrix} x_1, x_2, \dots, x_{m-1} \end{bmatrix} \tag{4}$$

where the x_i are state vectors of our system. We also construct a set of measurements δt later in time

$$X' = \begin{bmatrix} x_2, x_2, \dots, x_m \end{bmatrix} \tag{5}$$

Note: In the case of our videos, each of these x_i measurements correspond to a frame flattened into a column vector. Because this is currently only set up for a 2-dimensional input, we will implement this algorithm on grayscale videos.

The exact DMD can theoretical be computed as

$$A = X'X^+ \tag{6}$$

where $+$ denotes a psuedo-inverse. However, this is impractical to implement as computing a pseudo-inverse is costly and unstable. As a result, we will use a DMD algorithm.

3 Algorithmic Development

3.1 DMD Algorithm

Preform SVD on our data matrix X with rank-r truncation

$$X = U\Sigma V^* \tag{7}$$

Construct an approximate linear transform $\tilde{A}$ using the decomposed matrix [1]

$$\tilde{A} = U^* X' V \Sigma^{-1} \tag{8}$$

[1]Note that in implementation, we would not actually compute the inverse, but rather compute the least-squared error solution for the matrix equation.

We can then preform eigenvalue decomposition on our matrix $\tilde{A}$

$$\tilde{A}\Omega = \mu\Omega \tag{9}$$

where Ω holds our eigenvectors, and μ is a diagonal matrix of eigenvalues.

We can now get the DMD modes Φ of our system

$$\Phi = X'V\Sigma^{-1}W \tag{10}$$

and initial amplitude of our DMD modes b computed by

$$\mathbf{x} = \Phi b \tag{11}$$

and compute our solution for any time t

$$\mathbf{x} = \Phi\texttt{exp}(\Omega t)b \tag{12}$$

3.2 Adaptation to Video Separation

We can leverage the dynamic significance of the eigenvalues μ for background removal. For eigenvalues near 0, the solution to our DMD approximation of our system will simply be of the form $\texttt{exp}(0t)$ which does not change in time. Thus we can extract these modes who's eigenvalues $\mu_j \approx 0$ to get a reconstruction of the background of our video. Let the low-rank DMD approximation of our video be $X^{\texttt{Low Rank}}$. It follows that we can segment our system measurements

$$\mathbf{X} = \mathbf{X}_{\text{DMD}}^{\text{Low}-\text{Rank}} + \mathbf{X}_{\text{DMD}}^{\text{Sparse}} \tag{13}$$

where $\mathbf{X}_{\text{DMD}}^{\text{Sparse}}$ corresponds to the portion of our video that is moving, or the foreground.

A method for computing $\mathbf{X}_{\text{DMD}}^{\text{Sparse}}$ is presented by Grosek and Kutz in *Dynamic Mode Decomposition for Real-Time Background/Foreground Separation in Video* such that we get a result of real, positive values indicative of pixel intensities. If we assume that

$$\mathbf{X}_{\text{DMD}}^{\text{Sparse}} = \mathbf{X} - \left|\mathbf{X}_{\text{DMD}}^{\text{Low-Rank}}\right| \tag{14}$$

we get negative pixel intensities. Therefore, we can take the values of residual negative values and collect them into a matrix R. We can now compute our foreground and background DMD approximations such that we take care of these negaitve values.

$$\mathbf{X}_{\text{DMD}}^{\text{Low}-\text{Rank}} \leftarrow \mathbf{R} + \left|\mathbf{X}_{\text{DMD}}^{\text{Low}-\text{Rank}}\right| \tag{15}$$

$$\mathbf{X}_{\text{DMD}}^{\text{Sparse}} \leftarrow \mathbf{X}_{\text{DMD}}^{\text{Sparse}} - \mathbf{R} \tag{16}$$

Which satisfies the condition set out in Equation 13

4 Computational Results

We will look at applications of using DMD for background separation. Namely, we will look at systems that exhibit linear, rotational, and multi-dimensional dynamics. This will provide insight to the applications and limitations of DMD. Let's highlight the details of implementation.

We trim each of our videos to a length of 250 frames to improve processing time. This algorithm could be extended for longer videos, or be implemented as near real-time process. In addition, frames are downsized from (1080, 1920) pixels by a factor of 5 to (216, 384). Again, this is in the effort of reducing computation time by a factor of 5^2. As mentioned above, we choose to convert our movies to gray-scale for easy handling of the data and quick visualization development. Because we are already condensing multiple dimensions' information (x and y), it follows naturally that this process could be extended to applications where changing color is playing an important role.

When separating into sparse and low-rank components, the values in the corresponding representations tend to be condensed to a narrow range of values. As a result, we choose to rescale our images by 255 divided by the maximum value for improved visualization.

4.1 Linear Motion

The first video in question is of a VW Beetle driving across the desert. The camera is stationary, and there are no other items moving within the frame. A summary of DMD results for one frame is pictured in Figure 1.

Notice that our algorithm did a good job of extracting the headlights, roof, wheels, and trim of the car. The car's body is not picked up in the foreground. This is probably due to the fact that the car's grayscale color representation is very close to the road's. This result could certainly used as input for a motion tracking or position extraction algorithm. Also, note that our algorithm preserves the representation of the original image when the components are added together.

4.2 Rotational Motion

The next video is a simple wind turbine. The motion in the video is simple, rotation of the turbine's blades. Again, the camera is stationary, and there are no other items moving within the frame. A summary of DMD results for one frame is pictured in Figure 2.

We can see the double shadow of our turbine in our background image. Additionally, we see that the body of our turbine appears as something that is ”moving” in our image. It is important to remember what measurements we are actually feeding into the DMD: pixel intensity values. The turbine in our video is causing fluctuating shadows that are picked up by the body of our turbine. As a result, the DMD algorithm shows us that the pixel values at these locations are changing, even if the physical object that we see is stationary.

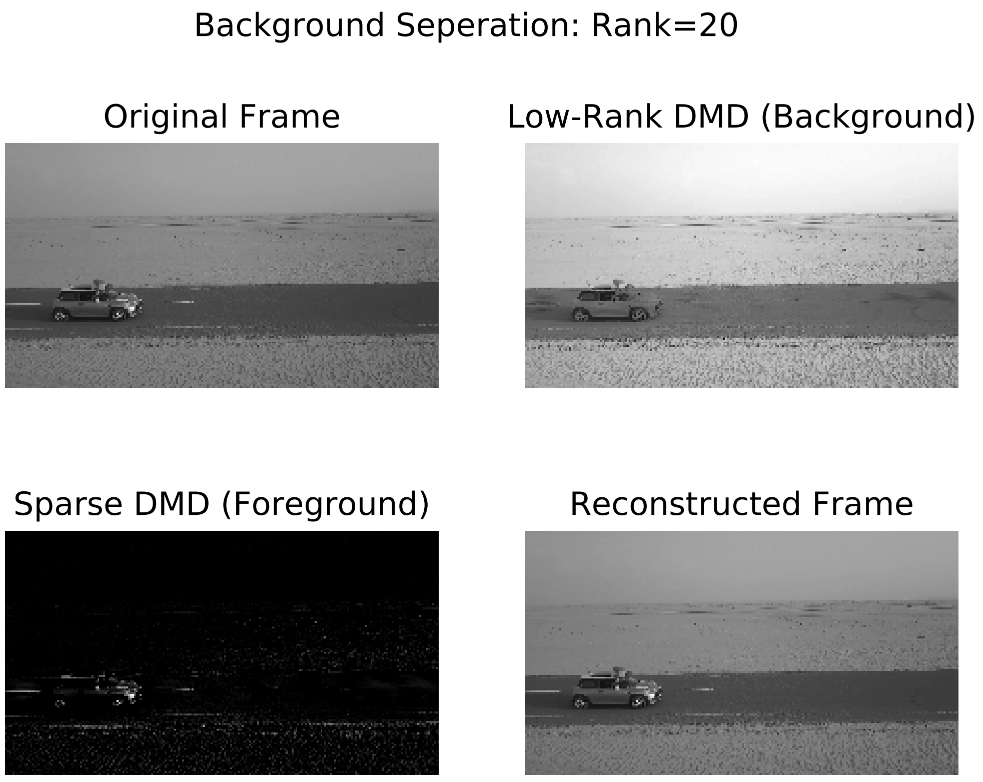

Figure 1: DMD foreground-background separation for linear motion

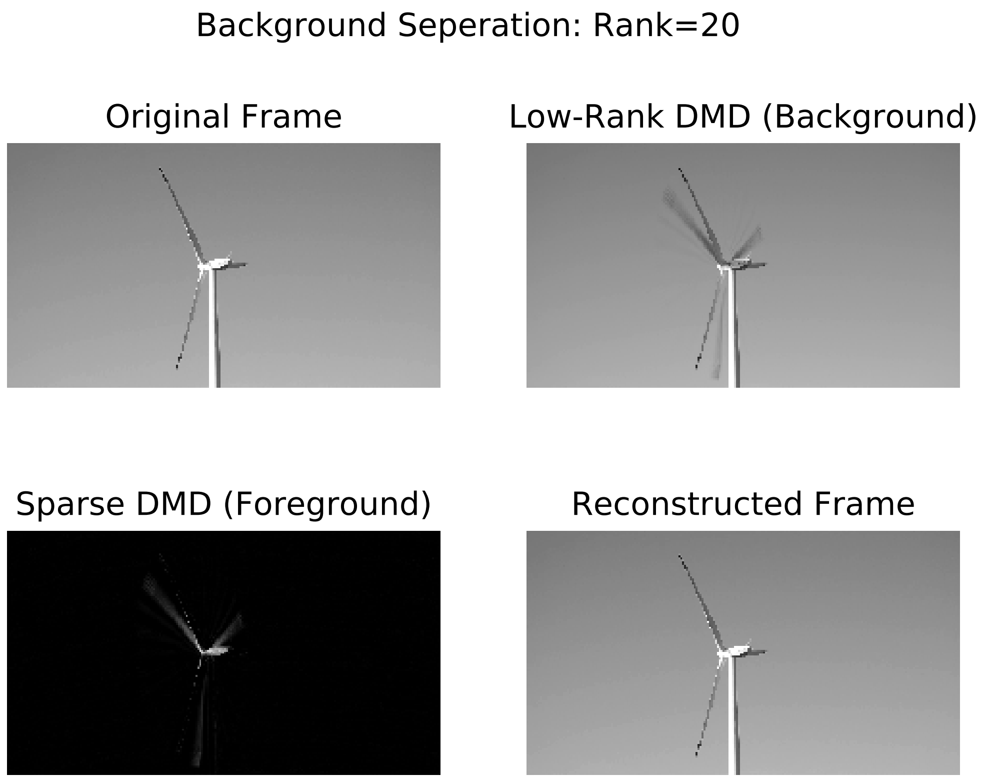

Figure 2: DMD foreground-background separation for rotational motion

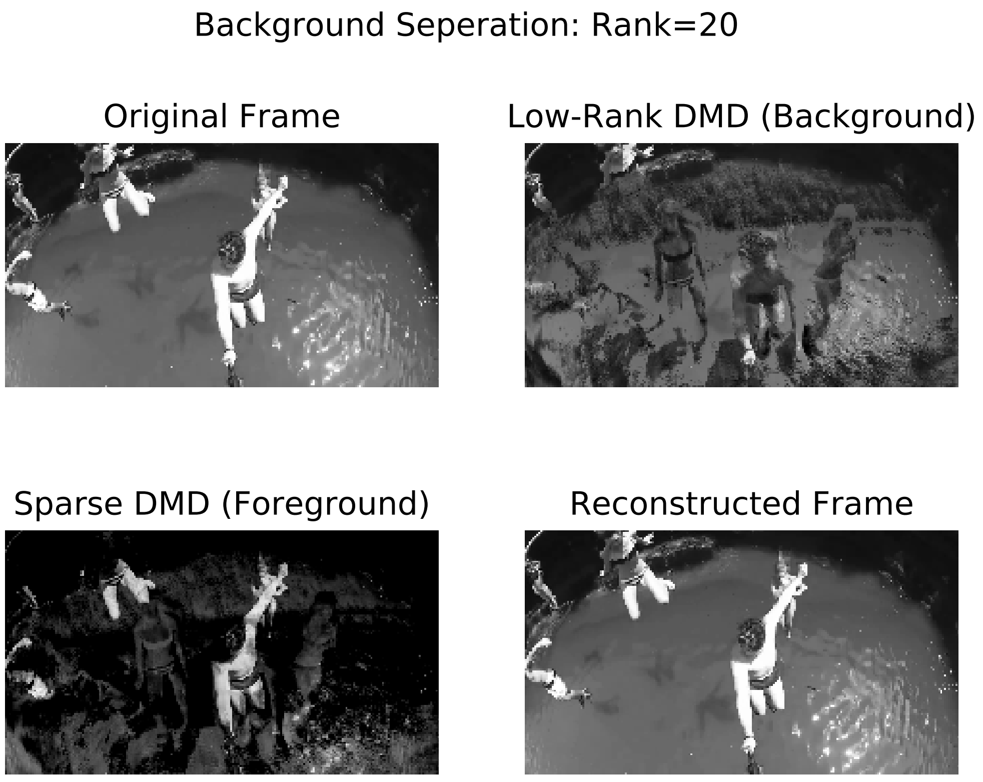

Figure 3: DMD foreground-background separation for multi-dimensional motion

4.3 Multi-Dimensional Motion

The final video is a shot from a selfie-stick-style cliff jumping video. The camera is pointed at the subjects, and while they are mostly in the same place in the frame throughout the movie, the video is messy. The background is changing drastically, the subjects are moving within the frame, and the lighting is changing. A summary of DMD results for one frame is pictured in Figure 3.

Despite the potential difficulties, the DMD does pretty well for this test case. We can see the person holding the camera is extracted quite well, and parts of the other people in the frame are pulled out in the foreground frame. The isolated background is quite messy with ghosts of subjects and background artifacts, however is pretty good considering that this process is entirely data-driven.

5 Summary and Conclusions

This report looked at applying Dynamic Mode Decomposition with the application of separation of foreground and background in video streams. We discussed math background, specific algorithm development, and how we are able to leverage the physical intuition that comes with eigenvalues ≈ 0 in order to extract DMD modes associated with static artifacts in our signals.

We examine video separation examples from three different test cases for linear, rotational, and multi-dimensional motion. We see that our algorithm does well on picking up motion for all three cases. Additionally, lighting changes are picked up as a result of the changing pixels values due to shadows.

Some areas for future exploration include extension into RGB color space, automatic ”zero eigenvalue” threshold setting, and general optimization for use with bigger systems. Additionally, we could look at pre-processing our images in a way that allows for the best possible DMD representation.

6 References

Kutz, J. N. & Grosek, J. (2014). Dynamic Mode Decomposition for Real-Time Background/Foreground Separation in Video.

Kutz, J. N. *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013.

Taylor, R. “Dynamic Mode Decomposition in Python.” Pyrunner, 25 July 2016, www.pyrunner.com/weblog/2016
python/.

7 Appendix A: Functions Used

For this analysis, I relied heavily on the numpy and matplotlib libraries for Python. For loading the audio files, I used skvideo.io. Below are some specific functions used:

`skvideo.io.load.load`: Load mp4 into Python with desired length, colorscheme, and other useful options

`np.linalg.svd`: Computes the Singular Value Decomposition

`np.linalg.eig`: Computes eigenvalues and eigenvectors

`np.conj`:Complex conjugate of numpy arrays. Remeber that while Matlab ’ operator takes the conjugate transpose, Python does not.

`np.T`: Transpose numpy array. Preserves complex values (and causes little bugs that take forever to fix)

8 Appendix B: Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
Created on Tue Mar 12 13:42:35 2019

@author: elimiller
"""


import numpy as np
import matplotlib.pyplot as plt
from skimage.transform import resize
import skvideo.io
plt.close('all')

def get_x(filename, num_frames, size=(216, 384)):
    num_frames = num_frames
    shape = size

    video = skvideo.io.vread(
            filename,
            num_frames=num_frames,
            as_grey=True, )[:, :, :, 0]
    A = []

    for frame in video:
        frame = resize(frame, shape)
        A.append(list(frame.ravel(order='C')))
    A = np.array(A).T
    average = A.mean(axis=0, keepdims=True)
    average = np.zeros_like(average)
    A = A - average
    return A, average

def DMD(A, rank, threshold):

    rank = rank
    threshold = threshold
    dt = 1

    X1 = A[:, :-1]
    X2 = A[:, 1:]

    U2, S_vec2, V2 = np.linalg.svd(X1, full_matrices=False)
    U = U2[:,0:rank]
    S = np.diag(S_vec2)[0:rank, 0:rank]
    V = V2.conj().T[:,0:rank]
    X = U @ S @ V.T
    if True:
        plt.figure()
        plt.plot(S_vec2 / np.sum(np.diag((S_vec2))))

    A_tilde = np.linalg.lstsq(U.conj().T @ X2 @ V, S, rcond=None)[0]
    mu, W = np.linalg.eig(A_tilde)
    Phi = np.linalg.lstsq((X2 @ V).T, S, rcond=None)[0] @ W

    omega = np.log(mu, dtype='complex128') / dt

    if True:
        fig, axs = plt.subplots(1, 2)
        fig.suptitle('Summary of Eigenvalues of $A_{tilde}$')
        axs[0].plot(omega.real, omega.imag,'o')
        axs[0].set_xlabel('Real Component')
        axs[0].set_ylabel('Imaginary Component')

        axs[1].plot(np.abs(omega),'o')
        axs[1].set_ylabel('Eigenvalue Magnitude')
        plt.tight_layout()

    omega_lr  = np.zeros_like(omega)
#     omega_sparse = np.zeros_like(omega)

    for j in range(len(omega)):
        #this could be faster with list comprehension
        if np.abs(omega[j]) <= threshold:
            omega_lr[j] = omega[j]

    t = np.arange(np.shape(A)[1])
    DMD_lr = np.zeros((rank, len(t)), dtype='complex128')
    b = np.linalg.lstsq(Phi, A[:,0], rcond=None)[0]

    for j in range(len(t)):
        DMD_lr[:, j] = b * np.exp(omega_lr * dt)

    X_lr = Phi @ DMD_lr
    X_sparse = A - np.abs(X_lr)
    R = np.clip(X_sparse, a_min=None, a_max=0)
    X_lr = R + np.abs(X_lr)
    X_sparse = X_sparse - R
    X_recon = X_lr + X_sparse

    return X_lr, X_sparse, X_recon


def image_out(X_lr, X_sparse, A, average, frame, shape):
    og_image = np.abs(np.reshape((A+average)[:,frame], shape))
    lr_image = np.abs(np.reshape((X_lr+average)[:, frame], shape))
    sparse_image = np.abs(np.reshape((X_sparse+average)[:, frame], shape))
    sparse_image *= 255.0/sparse_image.max()
    recon_image = np.abs(np.reshape((A+average)[:, frame], shape))
    return [og_image, lr_image, sparse_image, recon_image]


def image(im_out, rank, savename):
    fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
    plt.suptitle('Background Seperation: Rank=%d' %rank)
    axs[0, 0].imshow(im_out[0], cmap='gray')
    axs[0, 1].imshow(im_out[1], cmap='gray')
    axs[1, 0].imshow(im_out[2], cmap='gray')
    axs[1, 1].imshow(im_out[3], cmap='gray')

    titles = ['Original Frame',
              'Low-Rank DMD (Background)',
              'Sparse DMD (Foreground)',
              'Reconstructed Frame']

    axs[0, 0].set_title(titles[0])
    axs[0, 1].set_title(titles[1])
    axs[1, 0].set_title(titles[2])
    axs[1, 1].set_title(titles[3])

    for ax in axs.ravel(): ax.axis('off')
    plt.savefig(savename)

rank = 20
threshold = .05
filename = 'test3.mp4'
size = (216, 384)

A, average = get_x(filename, 100)
X_lr, X_sparse, X_recon = DMD(A, rank, threshold)

frame = 90
im_out= image_out(X_lr, X_sparse, A, average, frame, size)

image(im_out, rank, '%sfigure.pdf'%filename.replace('.mp4', ""))
```