Dynamic Mode Decomposition for Background Separation

Alex Mallen

Abstract

In this paper I demonstrate a technique to discover dynamics given measurements of a system for which the dynamical equations are unknown. This technique, called dynamic mode decomposition (DMD), finds a best-fit linear operator to advance our measurements one time step into the future. I apply it to two short video clips in order to separate the background from the video.

1 Introduction

Dynamical systems theory is well understood when the governing equations are known, and when they are linear. However, in many applications such as neuroscience and social networks, it is not reasonable to discover the dynamics analytically. This has given rise to data driven methods to discover the dynamics. One such method is dynamic mode decomposition (DMD), which will be explored in this paper. It learns a system of linear differential equations from data, while simultaneously extracting low-rank structure in the dynamics. This structure comes in the form of a sum of DMD modes that are oscillating and/or exponentially decaying/growing with time.

Two videos will be analyzed in this paper: a video of someone skiing off a small cliff, and a video of a few cars turning a corner in the Monte Carlo rally. Each video is around 5 seconds long. In the ski drop video, the camera is perfectly stationary, but there is significant camera shake in the racing video. The goal is to produce a video that represents the background—everything that doesn't move in the video—and the foreground—the cars and skiers.

2 Theory

When given measurements of a dynamical system $\mathbf{x}(t);\ t = 0, \Delta t, 2\Delta t, ..., T$ with unknown dynamics, DMD approximates the dynamics as linear. Specifically, it finds the single best linear operator A such that $\mathbf{x}(t + \Delta t) = A\mathbf{x}(t)$ for all t. This can be written as a matrix problem. Let X_1 be a matrix whose columns are $\mathbf{x}(t)$, in chronological order excluding the

last measurement at t=T. Let X_2 be a matrix whose columns are also $\mathbf{x}(t)$ in chronological order, but excluding the first measurement. Ideally, $X_2 = AX_1$. Thus, our best A is given by

$$A = X_2 X_1^{\dagger} \tag{1}$$

The product $A = X_2 X_1^{\dagger}$ is usually an enormous matrix, but with low-rank underlying structure. Therefore, the data is projected into a low-dimensional space where it evolves according to $\tilde{A}$. We do this by letting

$$A = U_r \tilde{A} U_r^*, \tag{2}$$

where U_r comes from computing the singular value decomposition (SVD) of $X_1 = U\Sigma V^T$ and then truncating it to rank r: $X_{1,r} = U_r \Sigma_r V_r^*$. Because A represents our dynamics, you can think of this similarity transform as projecting $\mathbf{x}(t)$ onto a better basis U_r, applying $\tilde{A}$, and then lifting it back into its original space. After substituting in the SVD, we get

$$\tilde{A} = U_r^* X_2 V_r \Sigma_r^{-1} \tag{3}$$

Now we have a linear dynamical system and can diagonalize it to easily find powers of $\tilde{A} = J\Lambda J^{-1}$, and then project the low rank dynamics back up into the original space.

$$\mathbf{x}_{DMD}(t) = \Phi e^{\ln(\Lambda)t} b \tag{4}$$

where

$$\Phi = X_2 V \Sigma^{-1} J \tag{5}$$

and

$$b = \Phi^{\dagger} \mathbf{x}(0). \text{ [2]} \tag{6}$$

The columns of Φ are the DMD modes. If an eigenvalue λ has complex modulus of approximately 1, then its logarithm is $\omega \approx 0$, which corresponds to a mode that does not significantly change with time. This column of Φ can be considered the background mode.

3 Implementation

I wrote all code in a Python jupyter notebook, and the source code can be found in [1]. My first step was to load each video with `imageio.get_reader` which is described in Appendix A 5.0.1. I looped over this reader object and appended each frame to a list, converting to grayscale by taking a weighted sum of the RGB values [3]. I then converted this list of frames into a floating point numpy array, and calculated the timestep dt = 1/framerate. I reshaped this 3D array of frames to obtain the snapshot matrix X whose columns represent flattened frames. I removed the last and first elements of this array to obtain X_1 and X_2 respectively. I then computed the SVD of X_1 using `np.linalg.svd` and followed equation 3 to compute $\tilde{A}$ with a rank of 5 in order to compromise speed and accuracy. I took the eigendecomposition

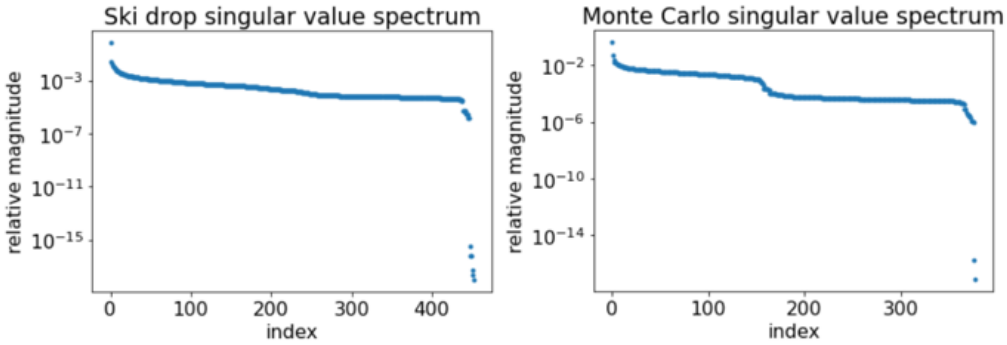

FIGURE 1: The singular value spectrum of each video.

of $\tilde{A}$ using `np.linalg.eig` and let $\Omega = \ln(\Lambda)/\mathrm{d}t$. I also computed the DMD modes Φ using equation 5 and the initial values in Φ-space b using equation 6.

There were several methods I used to determine which DMD mode represented the background X_{back}. Firstly, I looked for the mode whose corresponding $\omega \approx 0 + 0i$. I also looked at the values of b, since the background mode tended to have the largest coefficient. And lastly, I plotted the reshaped modes in Φ to see which one looked like the background. One I obtained the index i_b of the background mode I let $X_{\text{back}} = b[i]\Phi[i]$.

Foreground Extraction

The foreground X_{fore} is simply the original matrix X with the background subtracted from each frame. In the ski drop video, the foreground is a black skier on a white background. This means that X_{fore} is negative where the skier is present. I therefore took the negative of this matrix so that the skier is positive, and set the remaining negative pixels to 0. I finally reshaped each frame and plotted them. In the Monte Carlo rally video, I reshaped and plotted the absolute value of X_{fore} because the road was gray while the cars had both dark and light colors.

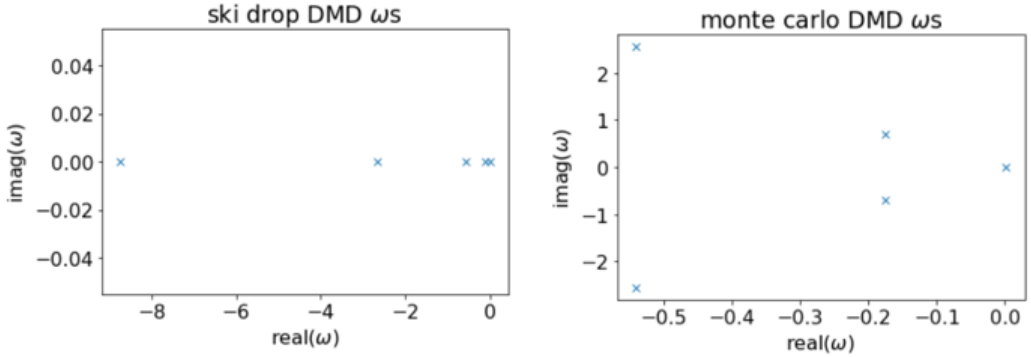

FIGURE 2: The temporal frequencies of the DMD modes.

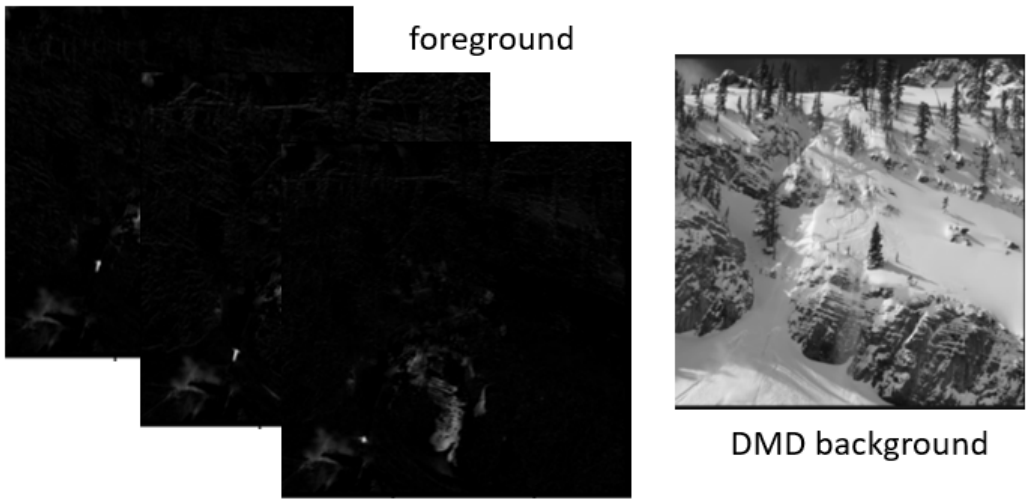

FIGURE 3: Foreground and background of the ski drop video.

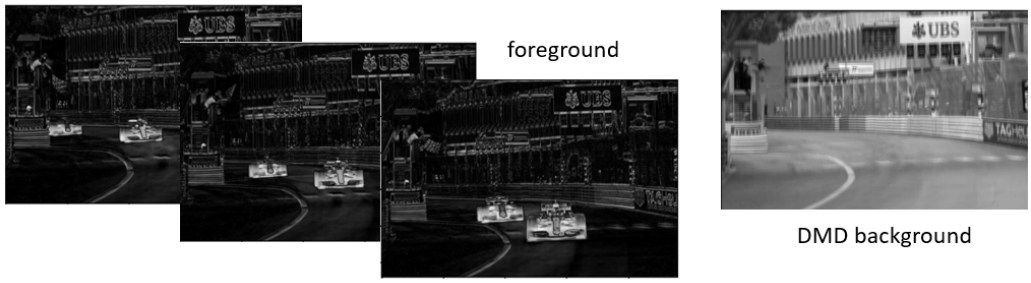

FIGURE 4: Foreground and background of the Monte Carlo rally video.

4 Results

The singular value spectrum for each video can be seen in figure 1. Figure 2 shows the temporal frequencies $\omega = \ln(\lambda)/\mathrm{d}t$. In both cases, the temporal frequency closest to zero turned out to be the one representing the background, as expected. The remaining frequencies seemed to vaguely represent various other moving parts of each scene, such as the snow falling over the rocky surface in the ski drop video.

Figure 3 shows the extracted foreground at frames 285, 295, and 305 of the ski drop video, as well as the DMD-extracted background. While small, you can clearly see the skier as they drop off the edge, and in the last frame bend their knees to land in the snow below. Other pixels are included in the foreground as well. Snow movement caused confusion because it was present for most of the video but didn't always look the same, so it appeared as one of the foreground objects along with the skier.

Figure 4 shows the extracted foreground at frames 0, 20, and 40 of the Monte Carlo video, as well as the DMD-extracted background. The background is still clearly visible in the foreground because the camera shake caused edges to appear in the foreground. The

majority of the background pixels are nearly black, except the edges because the camera shake caused a contrast between neighboring objects that occupy the same pixel at different times.

5 Conclusion

DMD was able to successfully extract the background from these videos, but not the foreground. This is not a surprise because, to an individual pixel, the skier is a momentary change in brightness that occurs once and never again. This cannot be characterized as a complex exponential as DMD assumes. A stationary background, however, can, so DMD extracts it successfully. Still, this is not a very impressive ability, because similar results can be obtained by averaging the frames or taking a cropped combination of frames.

DMD is a useful tool to approximate weakly nonlinear dynamical systems as linear, but clearly, the change in pixel values of a video of a skier is nonlinear. However, there is still theoretical reason to believe that we can model this system as linear. In equation 2, we linearly projected the dynamics onto a low-rank subspace before applying the time-stepping linear operator A. This produced another linear operator $\tilde{A}$. However, this change of coordinates does not need to be linear, and can instead be represented by an arbitrary function g, in which case $A = g \circ K \circ g^{-1}$, where K is a high-dimensional linear operator. A is generally a nonlinear operator governing the dynamics, but it is similar to a linear operator which means g only needs to be applied once at the beginning and g^{-1} at the end, and powers of A are still easy to compute. Koopman theory tells us that any dynamical system is similar to a linear dynamical system in infinite dimensions, but we can sometimes find low-dimensional subspaces in which a linear operator is an excellent approximation of the dynamics.

Appendix A

Important functions and their implementations

5.0.1 `imageio.get_reader`

Takes the directory to a video file and the option "ffmpeg" and returns a reader object that can iterate over the frames of the video. This reader object will not necessarily tell you how many frames the video has, so a dynamically sized array should be used to load all of the frames into memory. Installing imageio-ffmpeg through pip automatically installs ffmpeg software, which is what can turn an MP4 file into an array of frames. The resulting reader object also has a `get_meta_data` function that returns information like the resolution and framerate of the video.

Appendix B

Python code

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

# In[174]:
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 16})

import imageio

# In[176]:
filename = 'ski_drop_low.mp4'
vid_reader = imageio.get_reader(filename, 'ffmpeg')
metadata = vid_reader.get_meta_data()
metadata
# In[178]:
vid = []
step = 1
for i, image in enumerate(vid_reader.iter_data()):
    if i % step == 0:
        frame = image[:, :, 0] * 0.2125 + image[:, :, 1] * 0.7154 +
            image[:, :, 2] * 0.0721
        vid.append(frame[20:, 215:670]) #
            https://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_rgb_to_gray.

vid = np.array(vid, dtype=np.float32)
# In[179]:
vid.size, vid.dtype, vid.shape, vid.itemsize # time by x by y
# In[180]:
dt = step / metadata["fps"]

# # SVD
# In[181]:
X = vid.reshape((vid.shape[0], vid.shape[1] * vid.shape[2])).T # space by
    time
X1 = X[:, :-1]
X2 = X[:, 1:]

# In[182]:
U, S, Vh = np.linalg.svd(X1, full_matrices=False)
# In[185]:
plt.figure()
plt.title("Ski drop singular value spectrum")
```

```
plt.xlabel("index")
plt.ylabel("relative magnitude")
plt.semilogy()
plt.plot(S / sum(S), ".")
plt.show()

# # DMD
# In[150]:
r = 5
# Want to solve $AX_1 = X_2$ best fit, but use SVD
# In 151]:
A_tilde = U[:, :r].T @ X2 @ Vh[:r].T @ np.diag(1 / S[:r])
# In[152]:
lambs, J = np.linalg.eig(A_tilde)
# In[153]:
Phi = X2 @ Vh[:r].T @ np.diag(1 / S[:r]) @ J
# In[154]:
omegas = np.log(lambs) / dt # to convert to continuous time
# In[155]:
x0 = X[:, 0]
b = np.linalg.pinv(Phi) @ x0
# In[156]:
get_ipython().run_line_magic('matplotlib', 'notebook')
plt.figure()
plt.plot(np.real(Phi)[:, :3])
plt.show()
# In[157]:
get_ipython().run_line_magic('matplotlib', 'inline')
plt.figure()
plt.title("Ski drop omegas")
plt.xlabel("index")
plt.ylabel("$\omega$")
plt.plot(np.absolute(omegas), ".")
plt.semilogy()
plt.show()
# In[158]:
plt.plot(np.absolute(b))
# In[159]:
omegas # if eigenvalues all have large negative real part I don't see the
    skier
# In[128]:
back_idx = 1
background = b[back_idx] * np.reshape(Phi[:, back_idx], (vid.shape[1],
    vid.shape[2]))
# In[162]:
plt.figure(dpi=150)
plt.imshow(np.absolute(background), cmap="gray")
```

```
plt.colorbar()
# In[161]:
for i in range(len(b)):
    frame = b[i] * np.reshape(Phi[:, i], (vid.shape[1], vid.shape[2]))
    plt.imshow(np.absolute(frame))
    plt.colorbar()
    plt.show()
# In[132]:
time_dynamics = np.zeros((r, vid.shape[0]), dtype=np.complex64)
for i in range(0, vid.shape[0]):
    t = i * dt
    time_dynamics[:, i] = b * np.exp(omegas * t)
reconX = Phi @ time_dynamics
# In[134]:
recon_vid = reconX.T.reshape((vid.shape[0], *vid.shape[1:]))
# In[135]:
plt.figure(dpi=150)
plt.imshow(np.absolute(recon_vid[1]))
plt.colorbar()

# ## Foreground Extraction (backround subtraction)
# In[163]:
foreground_vid = -(vid - np.absolute(background))
# In[164]:
# R is negative
R = np.where(foreground_vid < 0, foreground_vid, 0) # contains all the
    negative values of foreground vid, otherwise 0
# In[165]:
foreground_vid -= R
# In[167]:
for i in range(35, 40, 1):
    plt.figure(dpi=150)
    plt.imshow(foreground_vid[i], cmap="gray")
    plt.colorbar()
# In[160]:
plt.plot(np.real(omegas), np.imag(omegas), "x")
plt.xlabel("real($\omega$)")
plt.ylabel("imag($\omega$)")
plt.title("ski drop DMD $\omega$s")


# # MONTE CARLO ANALYSIS
# In[166]:
filename = "monte_carlo_low.mp4"
vid_reader = imageio.get_reader(filename, 'ffmpeg')
metadata = vid_reader.get_meta_data()
metadata
```

```
# In[167]:
vid = []
step = 1
for i, image in enumerate(vid_reader.iter_data()):
    if i % step == 0:
        vid.append(image[:, :, 0] * 0.2125 + image[:, :, 1] * 0.7154 +
            image[:, :, 2] * 0.0721) #
            https://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_rgb_to_gray.

vid = np.array(vid, dtype=np.float32) # DWOIJT
# In[168]:
vid.size, vid.dtype, vid.shape, vid.itemsize # time by x by y
# In[169]:
dt = step / metadata["fps"]
# # SVD
# In[170]:
X = vid.reshape((vid.shape[0], vid.shape[1] * vid.shape[2])).T # space by
    time
X1 = X[:, :-1]
X2 = X[:, 1:]
# In[171]:
U, S, Vh = np.linalg.svd(X1, full_matrices=False)
# In[175]:
plt.title("Monte Carlo singular value spectrum")
plt.xlabel("index")
plt.ylabel("relative magnitude")
plt.semilogy()
plt.plot(S / sum(S), ".")
plt.show()
# # DMD
# In[30]:
r = 5
# Want to solve $AX_1 = X_2$ best fit, but use SVD
# In[31]:
A_tilde = U[:, :r].T @ X2 @ Vh[:r].T @ np.diag(1 / S[:r])
# In[32]:
lambs, J = np.linalg.eig(A_tilde)
# In[33]:
Phi = X2 @ Vh[:r].T @ np.diag(1 / S[:r]) @ J
# In[34]:
omegas = np.log(lambs) / dt # to convert to continuous time
# In[35]:
x0 = X[:, 0]
b = np.linalg.pinv(Phi) @ x0
# In[36]:
get_ipython().run_line_magic('matplotlib', 'notebook')
plt.figure()
```

```
plt.plot(np.real(Phi)[:, :3])
plt.show()
# In[37]:
get_ipython().run_line_magic('matplotlib', 'inline')
plt.figure()
plt.title("Ski drop omegas")
plt.xlabel("index")
plt.ylabel("$\omega$")
plt.plot(np.real(omegas), ".")
plt.show()
# In[38]:
omegas # if eigenvalues all have large negative real part I don't see the
    skier
# In[56]:
back_idx = 2
background = b[back_idx] * np.reshape(Phi[:, back_idx], (vid.shape[1],
    vid.shape[2]))
# In[57]:
plt.figure(dpi=150)
plt.imshow(np.absolute(background), cmap="gray")
plt.colorbar()
# In[58]:
for i in range(len(b)):
    frame = b[i] * np.reshape(Phi[:, i], (vid.shape[1], vid.shape[2]))
    plt.imshow(np.absolute(frame))
    plt.colorbar()
    plt.show()
# In[60]:
plt.plot(b)
# In[61]:
time_dynamics = np.zeros((r, vid.shape[0]), dtype=np.complex64)
for i in range(0, vid.shape[0]):
    t = i * dt
    time_dynamics[:, i] = b * np.exp(omegas * t)
reconX = Phi @ time_dynamics
# In[63]:
recon_vid = reconX.T.reshape((vid.shape[0], *vid.shape[1:]))
# In[64]:
plt.figure(dpi=150)
plt.imshow(np.absolute(recon_vid[0]))
plt.colorbar()
# ## Foreground Extraction (backround subtraction)
# In[68]:
foreground_vid = np.absolute(vid - np.absolute(background))
# In[71]:
plt.close("all")
# In[74]:
```

```
for i in range(0, 60, 20):
    plt.figure(dpi=150)
    plt.imshow(foreground_vid[i], cmap="gray")
    plt.colorbar()
# In[93]:
plt.plot(np.real(omegas), np.imag(omegas), "x")
plt.xlabel("real($\omega$)")
plt.ylabel("imag($\omega$)")
plt.title("monte carlo DMD $\omega$s")
```

References

[1] Mallen, Alex Troy. GitHub source code.
https://github.com/AlexTMallen/AMATH-582/tree/master/HW5

[2] Kutz, Jose Nathan. Dynamic Mode Decomposition (Theory)
https://www.youtube.com/watch?v=bYfGVQ1Sg98

[3] Scikit-image. rgb2gray conversion. https://scikit-image.org/docs/dev/auto_examples/color