Dimensionality Reduction of Oscillator Dynamics

Alex Mallen

Abstract

In this paper I will be demonstrating a technique to reduce the degrees of freedom of a set of measurements while quantifying how much information was lost in the compression. I apply this technique, called principle component analysis, to noisy and redundant recordings of an oscillator to find the underlying dynamics by projecting recordings onto a statistically independent set of orthogonal basis vectors.

1 Introduction

Principle component analysis (PCA) is a powerful tool to analyze and eliminate covariances between datasets. This can be seen as a form of compression that can extract the underlying dynamics behind measurements of an oscillator.

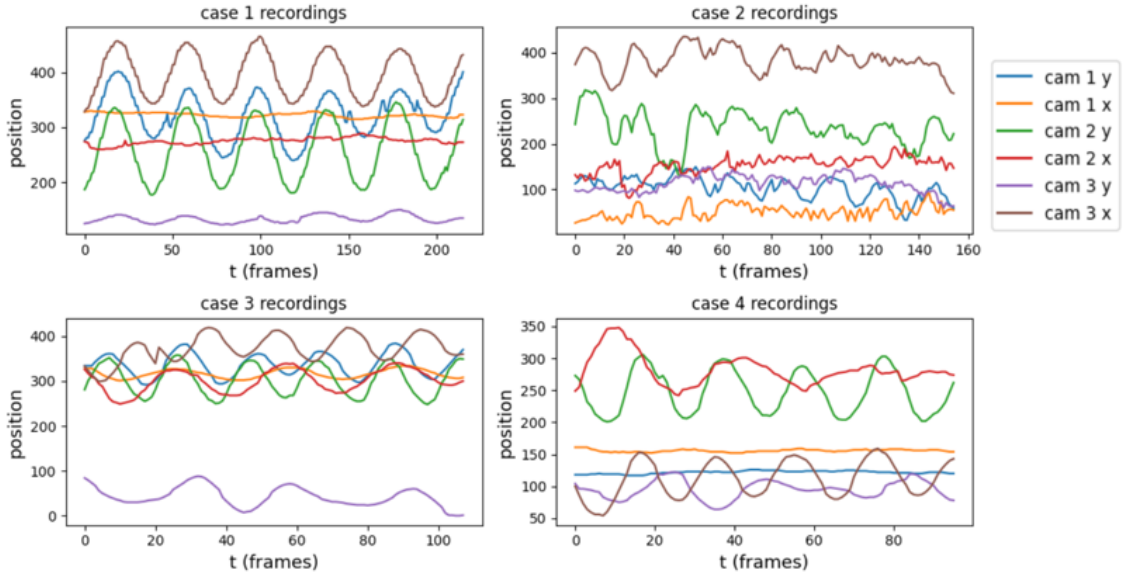

FIGURE 1: The 6 time series for each case describe the $x-$ and $y-$coordinates of the mass within each video over time. Notice that each case has some sinusoidal behavior and is otherwise roughly constant. The recording of case 4 has been trimmed because the pendulum dynamics decayed fairly quickly as seen in `cam 2 x` (the red line).

The measurements of the oscillator used in this demonstration are video recordings from 3 cameras with different perspectives of the oscillator. Several techniques described in section 3 were used to extract the dynamics shown in figure 1. These data were then analyzed with PCA. There were four cases:

1. Vertical simple harmonic oscillation of a mass on a spring
2. Same as 1, but with camera shake
3. Same as 1, but with an initial horizontal displacement to introduce pendulum-like dynamics
4. Same as 3, but with a rotation to produce torsion-pendulum-like dynamics

2 Theory

PCA takes a set of data X with features as rows and analyzes the covariance between these features. The covariance between features **a** and **b** can be understood as how much a deviation from the mean of **a** is related to the deviation from the mean of **b**. For **a** and **b** of length T, the sample covariance is defined as

$$\mathsf{cov}(\mathbf{a}, \mathbf{b}) = \frac{1}{T-1} \sum_{i=1}^{T} (\mathbf{a}_i - \bar{a})(\mathbf{b}_i - \bar{b}), \tag{1}$$

where $\bar{a}$ and $\bar{b}$ are the means of their respective features.

By this definition of covariance, one can see that the covariance between every possible pair of features in X can be computed as

$$C = \frac{1}{T-1} X_c X_c^T, \tag{2}$$

where X_c is the mean-subtracted version of X, such that each row sums to 0 [2]. C is called the covariance matrix. Along the diagonal is the covariance of every feature with itself, or the variance of every feature. We would like to produce a set of basis features that are statistically independent; in other words, we want the covariance matrix to be diagonal. Since it is of the form $X_c X_c^T$, we know that it is symmetric positive semi-definite. In other words, C has a diagonalization with non-negative real eigenvalues and orthogonal eigenvectors. Let

$$C = \frac{1}{T-1} X_c X_c^T = V \Lambda V^T \tag{3}$$

The columns of V, also known as the principle components, form a new basis for the mean-subtracted features. Then we can define Y_c as a projection of our X_c onto the new feature space as follows.

$$Y_c = V^T X_c \tag{4}$$

Then our new covariance matrix C_Y is diagonal because

$$\begin{aligned} C_Y &= \frac{1}{T-1} Y_c Y_c^T \\ &= \frac{1}{T-1} V^T X_c X_c^T V \\ &= \frac{1}{T-1} V^T (V \Lambda V^T) V \qquad \text{by eq 4} \\ &= \frac{1}{T-1} \Lambda \end{aligned}$$

Our covariance matrix is precisely our normalized eigenvalue matrix. Because the eigenvalues are proportional to the variances, eigenvectors with large eigenvalues contain most of the information in X, and the rest can be excluded. Furthermore, the n eigenvectors with the largest associated eigenvalues are the optimal linear basis of size n for representing the data. We should expect that in cases with few linear degrees of freedom, the first one or two eigenvectors explain the majority of the data.

The process we went through to produce the principle components is precisely calculating the singular value decomposition (SVD) of X_c, defined as

$$X_c = U \Sigma V^T, \tag{5}$$

where the V of the eigendecomposition is equal to the U of the SVD.

3 Implementation

I implemented all of my analysis in Python using a Jupyter Notebook.

Preprocessing

My goal in the preprocessing step was to produce a modified video from which I could take the argmax of each frame to locate the mass.

For each case, my first step in extracting the dynamics from video was to align the 3 videos by watching the frames (in MATLAB) and trimming them so that they start and end at the same time. I also cropped the frames of each video to the minimum size necessary to see the mass at all times.

Then, I created a grayscale copy of each video, and in the same pass I computed the average frame (background) of each video. This step is described further in Appendix A.

Cases 1, 3, and 4

My next step for these three cases was to subtract the background from each frame and apply a centered Gaussian filter to each frame to accentuate the middle of the frame. I also wanted to make sure that no individual pixel could mistakenly be extracted, so I convolved

each frame with a small Gaussian filter (21×21 pixels) to smooth out any such fluctuations. This step is further described in Appendix A. These cases did not have camera shake, so I performed a frame-to-frame subtraction of the preprocessed videos to extract the parts of the image in which there was motion. Specifically, I took the difference between each preprocessed frame and the frame 4 before it because I found that some adjacent frames were too similar to find motion. For cases 3 and 4, I also excluded every other frame to make the analysis faster.

Lastly, I called `extract_location` to find the argmax of each frame, with a distance penalty, as described in Appendix A. The `dist_weight` I used for these cases was 0.5. The distance penalty complemented the frame-to-frame subtraction nicely for cases 1, 3, and 4 because the distance penalty works best when the mass is stationary, exactly where frame-to-frame subtraction fails.

Case 2

I did not use frame-to-frame subtraction for case 2 because of camera shake. Instead, I made use of the color information in the video. Since the flashlight attached to the mass was bright *and* white I processed the video by first calculating the brightness of each pixel, then deducting a measure of how non-white the pixel is. Specifically, I subtracted a multiple of $2|F - R| + 3|F - G| + |F - B|$, where F is the brightness of the pixel (grayscale), and R, G and B are the red, green, and blue components of the pixel respectively. The increased weight attributed to R and G is because I found many of the mistakenly classified pixels were yellow, while the flashlight had a slightly blue tint.

I then called `extract_location` with a distance penalty weight of 0.02 for this case. The reason this weight is much smaller is mostly because the preprocessed frames had smaller values.

I tuned my extraction methods and parameters for all the cases by viewing the resulting frames to check where an argmax might confuse something else for the mass. I also looked at the resulting plots of the position versus time to make sure it was reasonable.

Extracting Principle Components

Once I obtained the matrix X, I computed the covariance matrix as in equation 2. I chose to compute the covariance matrix rather than simply calling an SVD function on the mean-subtracted data so that I could inspect the covariances and understand the analysis better. I then called `np.linalg.eig`, which is described in Appendix A, on the covariance matrix to obtain Λ and V. I then projected the mean-subtracted dynamics X_c onto the principle components in V, and plotted these projected dynamics. Based on a.) the relative magnitudes of the associated eigenvalues for each principle component and b.) the noisiness of the projection, I excluded the least meaningful principle components to obtain around 90% explained variance.

4 Results

The PCA-extracted dynamics for each case can be seen in figure 2. In case 1, the largest eigenvalue explained 93% of the variance in the data. The first principle component projection shows a clean sinusoidal pattern in time, representing the simple harmonic motion of the mass-spring system.

In case 2, the largest 3 principle components explained 86% of the variance. The fact that more principle components were needed to capture a smaller portion of the variance, despite it representing the same system as in case 1, indicates that PCA is not very noise

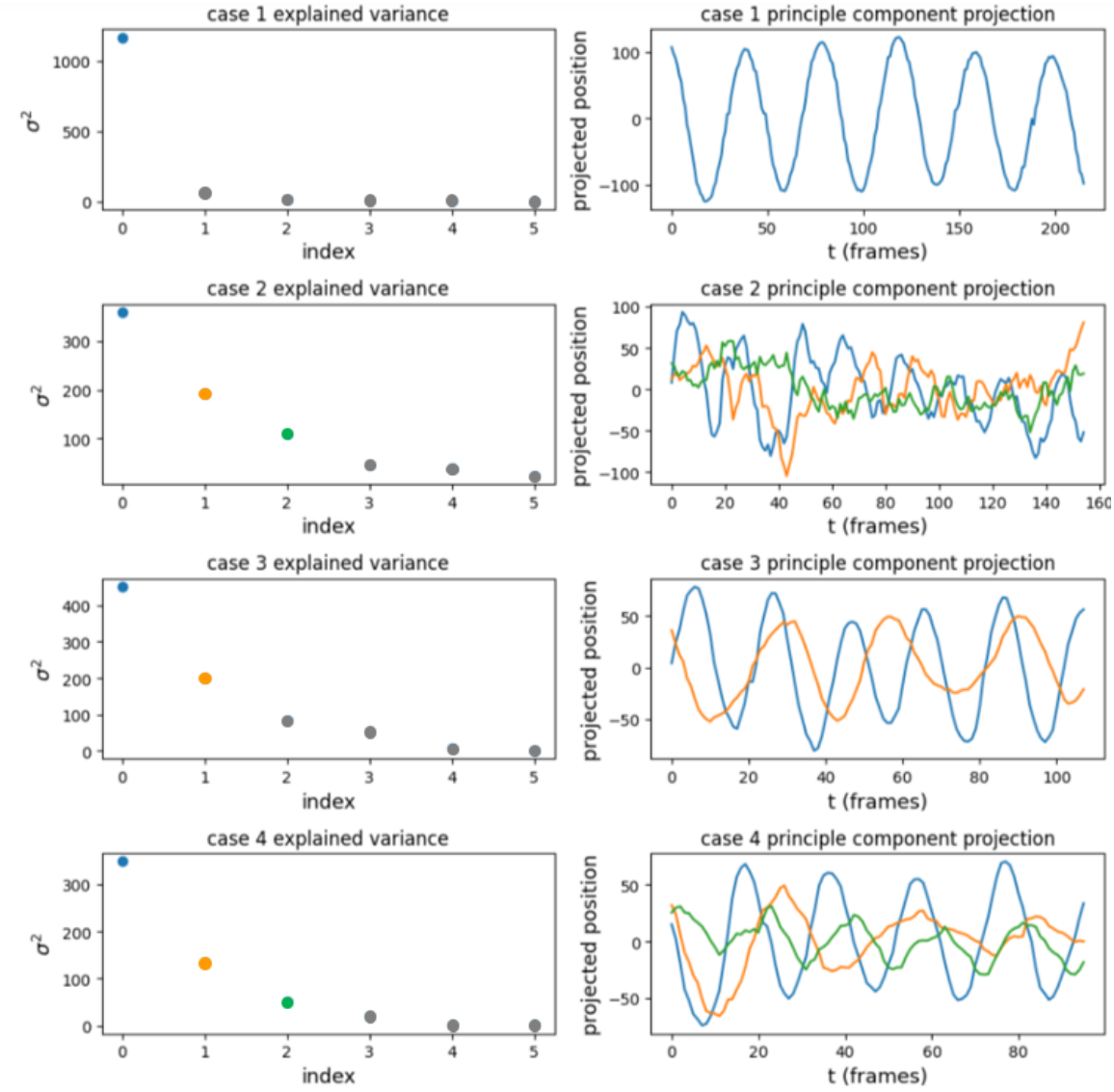

FIGURE 2: PCA results for each of the 4 cases. While case 3 variances look similar to case 4, the 3rd principle component of case 3 is noise, unlike for case 4. Note: σ^2 refers to the variance, not the squared singular value.

robust.

The largest 2 principle components of case 3 account for 82% of the variance. The mass in this system moves on a plane, so we would expect to find almost all of the variance accounted for by the first two principle components. While this is not exactly the case, the projection of the dynamics onto the first two principle components matches my observations: the mass oscillated up and down with a higher frequency than it swung horizontally. The remaining principle components did not appear to contribute meaningful information about the dynamics of the system.

The largest 3 principle components in case 4 account for 96% of the variance. The first two principle component projections correspond to the first two in case 3, but there is an additional principle component projection with sinusoidal dynamics which represents a third dimension of motion introduced by rotating the mass. Also note that the second principle component projection experiences decay, which matches my observation from the video that the pendulum dynamics decayed.

5 Conclusion

Principle component analysis proved to be a useful computational tool for finding and removing covariances in redundant data in order to uncover underlying dynamics of systems. However, it is not particularly noise robust. This can be seen in case 2, where PCA required more than 3 times as many principle components to account for 90% of the dynamics as it did for case 1, despite the dynamics being the same. Because PCA performs a least-squares fit to every principle component, it is especially susceptible to outliers. I noticed this problem when my location extraction technique was misclassifying other moving objects in the frame as the flashlight, which produced spikes in both the input dynamics and the principle components. This suggests the need for robust PCA.

Nonetheless, PCA's ability to find an optimal statistically independent basis for data has far-reaching applications. It can be used to determine how much information is contained in a signal and compress it accordingly. Because every matrix has a singular value decomposition without exception, it means that there is some orthogonal basis onto which every dataset can be projected to remove all covariances.

Appendix A

`scipy.ndimage.convolve`

Takes an image and a kernel as input and returns the image convolved with the given kernel.

The result will have the same size as the original image, and the way it handles edge pixels is determined by the mode parameter. Taking the convolution involves setting every pixel to some linear combination of its neighbors according to the weights given by the kernel. See [3].

`get_bw_avg`

This is the first step in extracting the location of the mass from the frames of the videos in cases 1, 2, and 4. It takes in a list of color videos and loops over all the frames to produce grayscale versions of each frame. The red, green, and blue pixels are given different weights because of the workings of our eyes. It computes the average grayscale frame for each video in the same pass, then returns a list of grayscale videos along with a list of average frames for each video.

`preprocess`

This is the second step in extracting the location of the mass from the the frames of the videos in cases 1, 3, and 4. It takes grayscale videos, the average frame for each video, a list Gaussian filters the same shape as the frames of the corresponding video, and a small Gaussian kernel. It returns a list of new videos that have been convolved with the kernel and multiplied by the corresponding Gaussian filter. There is also an optional step parameter to exclude frames and speed up the process.

`extract_location`

Takes a list of preprocessed videos and returns the pixel indices of the mass in each frame in the form of a matrix, where each row represents either the x- or y-axis of each video, and each column represents time.

It first finds the argmax coordinates within the first frame of each video, and then loops through the remaining frames. Rather than directly taking the argmax of these frames, it first subtracts off `dist_weight` times the squared distance from each pixel to the location of the mass in the previous frame in order to enforce continuity in the system because we know it's a physical system.

`np.linalg.eig`

Computes the diagonalization of the input as in equation 4, and returns a list of eigenvalues, and a matrix whose columns are the eigenvectors. See [4].

Appendix B

Python code

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

# In[2]:


import scipy.io
import numpy as np
import time

import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 300
get_ipython().run_line_magic('matplotlib', 'notebook')

from IPython.display import clear_output
import os

from scipy.ndimage import convolve


# In[3]:


def get_bw_avg(cams):
    camsbw = [] # black and white videos
    camavgs = [] # average frame in each video
    for i, cam in enumerate(cams):
        cambw = cam[:, :, 0, :] * 0.2989 + cam[:, :, 1, :] * 0.5870 +
            cam[:, :, 2, :] * 0.1140 # convert to grayscale
        camsbw.append(cambw)
        print("bw:", cambw.shape)
        camavg = np.mean(cambw, axis=2)
        camavgs.append(camavg)
    return camsbw, camavgs


# In[4]:


def preprocess(camsbw, camavgs, filts, kernel, step=1):
    filtereds = []
    for i, cambw in enumerate(camsbw):
```

```
        filtered = np.zeros((cambw.shape[0], cambw.shape[1], cambw.shape[2]
            // step))
        for t in range(0, filtered.shape[-1]):
            filtered[:, :, t] = convolve(cambw[:, :, t * step] - camavgs[i],
                kernel) * filts[i]
        filtereds.append(filtered)
    return filtereds


# In[5]:


def find_motion(filtereds, offset=2):
    camsdiff = []
    for i, filtered in enumerate(filtereds):
        camdiff = abs(filtered[:, :, offset:] - filtered[:, :, :-offset])
        camsdiff.append(camdiff)
    return filtereds


# In[6]:


def extract_location(camsdiff, dist_weight=0.5):
    tmax = camsdiff[0].shape[-1]
    grids = []
    for cam in camsdiff:
        grids.append(np.meshgrid(np.arange(cam.shape[1]),
            np.arange(cam.shape[0])))
    X = []
    i0, j0 = np.unravel_index(np.argmax(camsdiff[0][:,:,0]),
        camsdiff[0].shape[:2])
    i1, j1 = np.unravel_index(np.argmax(camsdiff[1][:,:,0]),
        camsdiff[1].shape[:2])
    i2, j2 = np.unravel_index(np.argmax(camsdiff[2][:,:,0]),
        camsdiff[2].shape[:2])
    X.append(np.array([i0, j0, i1, j1, i2, j2]))
    for t in range(1, tmax):
        i0, j0 = np.unravel_index(np.argmax(camsdiff[0][:,:,t] -
            dist_weight * ((grids[0][1] - X[-1][0])**2 + (grids[0][0] -
            X[-1][1])**2)), camsdiff[0].shape[:2])
        i1, j1 = np.unravel_index(np.argmax(camsdiff[1][:,:,t] -
            dist_weight * ((grids[1][1] - X[-1][2])**2 + (grids[1][0] -
            X[-1][3])**2)), camsdiff[1].shape[:2])
        i2, j2 = np.unravel_index(np.argmax(camsdiff[2][:,:,t] -
            dist_weight * ((grids[2][1] - X[-1][4])**2 + (grids[2][0] -
            X[-1][5])**2)), camsdiff[2].shape[:2])
```

```
            X.append(np.array([i0, j0, i1, j1, i2, j2]))

    return np.transpose(np.array(X, dtype=np.float32))


# In[7]:


def get_filter(xw, yw, scalex=400, scaley=400):
    filtX2, filtY2 = np.meshgrid(np.arange(xw), np.arange(yw),
        indexing="xy")
    filt = np.exp( -(filtX2 - xw/2)**2 / scalex**2 - (filtY2 - yw/2)**2 /
        scaley**2)
    get_ipython().run_line_magic('matplotlib', 'inline')
    plt.figure()
    plt.imshow(filt)
    plt.show()
    return filt


# In[8]:


width = 10
kernelX, kernelY = np.meshgrid(np.arange(-width, width), np.arange(-width,
    width))
scale = 6
kernel = np.exp( -(kernelX)**2 / scale**2 - (kernelY)**2 / scale**2)


# # Setup 1

# In[9]:


cam11 = scipy.io.loadmat("./cam1_1.mat") # loads matlab files into python
    as a dict of np.ndarrays
cam11 = cam11["vidFrames1_1"].astype(np.float32)
cam21 = scipy.io.loadmat("./cam2_1.mat")["vidFrames2_1"].astype(np.float32)
cam31 = scipy.io.loadmat("./cam3_1.mat")["vidFrames3_1"].astype(np.float32)


# In[10]:


cams = [cam11[:, :480, :, 10:226], cam21[:, :480, :, 21:237], cam31[150:,
    :, :, 10:226]]
```

```
# In[11]:


camsbw, camavgs = get_bw_avg(cams)


# In[12]:


filt11 = get_filter(cams[0].shape[1], cams[0].shape[0], scalex=200,
    scaley=400)
filt21 = get_filter(cams[1].shape[1], cams[1].shape[0], scalex=200,
    scaley=400)
filt31 = get_filter(cams[2].shape[1], cams[2].shape[0], scalex=400,
    scaley=200)


# In[13]:


filtered = preprocess(camsbw, camavgs, [filt11, filt21, filt31], kernel,
    step=1)


# In[14]:


camsdiff = find_motion(filtered, offset=4)


# In[15]:


get_ipython().run_line_magic('matplotlib', 'inline')
fig = plt.figure()
ax = fig.gca()
plt.imshow(camavgs[0], cmap="gray")
plt.show()
for i in range(0, 15, 1):
    plt.imshow(camsdiff[0][:,:,i])
    plt.colorbar()
    plt.show()
    plt.imshow(camsbw[0][:,:,i], cmap="gray")
    plt.show()
```

```
# In[23]:


X = extract_location(camsdiff, dist_weight=0.5)


# In[24]:


plt.figure()
plt.plot(np.transpose(X)[:, :])
plt.show()


# In[25]:


X_c = X - np.mean(X, axis=1, keepdims=True) # mean-subtracted
cov = X_c @ np.transpose(X_c) / (X_c.shape[-1] - 1)
L, V = np.linalg.eig(cov)
Y = np.transpose(V) @ X_c
covY = Y @ np.transpose(Y) / (Y.shape[-1] - 1)


# In[30]:


# covariance matrix visualization
plt.figure()
plt.imshow(np.log(abs(covY) + 1))
plt.colorbar()
plt.show()
covY


# In[27]:


# look at the relationship between the first two principle components
plt.scatter(Y[0], Y[1])
Y[0] @ Y[1] / (len(Y[0]) - 1)


# In[28]:
```

```
plt.figure()
plt.scatter(np.arange(len(L)), L)
# plt.semilogy()
plt.show()


# In[29]:


# calculate percent variance
plt.figure()
plt.plot(np.transpose(Y))
plt.show()


# # Setup 2 Camera Shake
# don't do any diffing frame to frame
# crop more
#

# In[31]:


cam12 = scipy.io.loadmat("./cam1_2.mat")["vidFrames1_2"].astype(np.float32)
cam22 = scipy.io.loadmat("./cam2_2.mat")["vidFrames2_2"].astype(np.float32)
cam32 = scipy.io.loadmat("./cam3_2.mat")["vidFrames3_2"].astype(np.float32)


# In[32]:


print(cam12.shape, cam22.shape, cam32.shape)


# In[33]:


cams2 = [cam12[200:, 300:480, :, 4:314], cam22[:, 150:480, :, 26:336],
    cam32[150:, :, :, 7:317]]
camsbw, camavgs = get_bw_avg(cams2)


# In[34]:


filt12 = get_filter(cams2[0].shape[1], cams2[0].shape[0], scalex=200,
    scaley=400)
```

```
filt22 = get_filter(cams2[1].shape[1], cams2[1].shape[0], scalex=200,
    scaley=400)
filt32 = get_filter(cams2[2].shape[1], cams2[2].shape[0], scalex=400,
    scaley=200)


# In[35]:


filts = [filt12, filt22, filt32]
step=2
filtered = []
for i, cam in enumerate(cams2):
    fltd = np.zeros((cam.shape[0], cam.shape[1], cam.shape[-1] // step))
    for t in range(0, fltd.shape[-1]):
        frame = camsbw[i][:, :, t*step]
        deviation = 2 * abs(cam[:, :, 0, t*step] - frame) + 3 * abs(cam[:,
            :, 0, t*step] - frame) + abs(cam[:, :, 0, t*step] - frame)
        fltd[:, :, t] = (frame - 0.5*deviation) * filts[i]
    filtered.append(fltd)


# In[36]:


get_ipython().run_line_magic('matplotlib', 'inline')
fig = plt.figure()
ax = fig.gca()
plt.imshow(camavgs[1], cmap="gray")
plt.show()
for i in range(100, 155, 1):
    plt.imshow(filtered[1][:,:,i])
    plt.colorbar()
    plt.show()
    plt.imshow(camsbw[1][:,:,i], cmap="gray")
    plt.show()


# In[37]:


X2 = extract_location(filtered, dist_weight=0.02)


# In[38]:
```

```
get_ipython().run_line_magic('matplotlib', 'notebook')
plt.figure()
plt.plot(np.transpose(X2)[:, :])
plt.show()


# In[39]:


X2_c = X2 - np.mean(X2, axis=1, keepdims=True) # mean-subtracted
cov = X2_c @ np.transpose(X2_c) / (X2_c.shape[-1] - 1)
L2, V = np.linalg.eig(cov)
Y2 = np.transpose(V) @ X2_c
covY = Y2 @ np.transpose(Y2) / (Y2.shape[-1] - 1)


# In[40]:


L2


# In[41]:


plt.figure()
plt.scatter(np.arange(len(L2)), L2)
# plt.semilogy()
plt.xlabel("index")
plt.ylabel("$\sigma$")
plt.show()


# In[43]:


plt.figure()
plt.plot(np.transpose(Y2)[:, :3])
plt.show()


# In[ ]:
```

```
# # Setup 3 Oscillation and Pendulum

# In[44]:


cam13 = scipy.io.loadmat("./cam1_3.mat")["vidFrames1_3"].astype(np.float32)
cam23 = scipy.io.loadmat("./cam2_3.mat")["vidFrames2_3"].astype(np.float32)
cam33 = scipy.io.loadmat("./cam3_3.mat")["vidFrames3_3"].astype(np.float32)


# In[45]:


cam33.shape


# In[46]:


cams3 = [cam13[:, :450, :, 2:219], cam23[:, :450, :, 32:249], cam33[200:,
    :, :, 20:]]
camsbw, camavgs = get_bw_avg(cams3)


# In[47]:


filt13 = get_filter(cams3[0].shape[1], cams3[0].shape[0], scalex=400,
    scaley=400)
filt23 = get_filter(cams3[1].shape[1], cams3[1].shape[0], scalex=400,
    scaley=400)
filt33 = get_filter(cams3[2].shape[1], cams3[2].shape[0], scalex=400,
    scaley=400)


# In[ ]:




# In[48]:


filtered = preprocess(camsbw, camavgs, [filt13, filt23, filt33], kernel,
    step=2)
```

```
# In[49]:


camsdiff = find_motion(filtered, offset=4)


# In[50]:


get_ipython().run_line_magic('matplotlib', 'inline')
fig = plt.figure()
ax = fig.gca()
plt.imshow(camavgs[0], cmap="gray")
plt.show()
for i in range(0, 10, 1):
#    plt.imshow(np.log(abs(camsbw[0][:,:,i + 1] - camsbw[0][:,:,i]) + 1))
    plt.imshow(camsdiff[0][:,:,i])
    plt.colorbar()
    plt.show()
    plt.imshow(camsbw[0][:,:,i], cmap="gray")
    plt.show()


# In[51]:


X3 = extract_location(camsdiff, dist_weight=0.5)


# In[52]:


get_ipython().run_line_magic('matplotlib', 'notebook')
plt.figure()
plt.plot(np.transpose(X3)[:, :])
plt.show()


# In[53]:


X3_c = X3 - np.mean(X3, axis=1, keepdims=True) # mean-subtracted
cov = X3_c @ np.transpose(X3_c) / (X3_c.shape[-1] - 1)
L3, V = np.linalg.eig(cov)
Y3 = np.transpose(V) @ X3_c
covY = Y3 @ np.transpose(Y3) / (Y3.shape[-1] - 1)
```

```
# In[54]:


plt.figure()
plt.scatter(np.arange(len(L3)), L3)
# plt.semilogy()
plt.show()


# In[56]:


plt.figure()
plt.plot(np.transpose(Y3)[:, [0, 1, 3]])
plt.show()


# In[ ]:




# # Setup 4 Oscillation and Rotation

# In[57]:


cam14 = scipy.io.loadmat("./cam1_4.mat")["vidFrames1_4"].astype(np.float32)
cam24 = scipy.io.loadmat("./cam2_4.mat")["vidFrames2_4"].astype(np.float32)
cam34 = scipy.io.loadmat("./cam3_4.mat")["vidFrames3_4"].astype(np.float32)


# In[58]:


cams4 = [cam14[:, :460, :, :192], cam24[:, :460, :, 4:196], cam34[100:,
    300:, :, :192]]
camsbw, camavgs = get_bw_avg(cams4)


# In[59]:
```

```
filt14 = get_filter(cams4[0].shape[1], cams4[0].shape[0], scalex=400,
    scaley=400)
filt24 = get_filter(cams4[1].shape[1], cams4[1].shape[0], scalex=400,
    scaley=400)
filt34 = get_filter(cams4[2].shape[1], cams4[2].shape[0], scalex=400,
    scaley=400)


# In[60]:


filtered = preprocess(camsbw, camavgs, [filt14, filt24, filt34], kernel,
    step=2)


# In[61]:


camsdiff = find_motion(filtered, offset=4)


# In[62]:


get_ipython().run_line_magic('matplotlib', 'inline')
fig = plt.figure()
ax = fig.gca()
plt.imshow(camavgs[2], cmap="gray")
plt.show()
for i in range(0, 30, 1):
    plt.imshow(camsdiff[2][:,:,i])
    plt.colorbar()
    plt.show()
    plt.imshow(camsbw[2][:,:,i], cmap="gray")
    plt.show()


# In[63]:


X4 = extract_location(camsdiff, dist_weight=0.5)


# In[64]:


get_ipython().run_line_magic('matplotlib', 'notebook')
```

```
plt.figure()
plt.plot(np.transpose(X4)[:, :])
plt.show()


# In[65]:


X4_c = X4 - np.mean(X4, axis=1, keepdims=True) # mean-subtracted
cov = X4_c @ np.transpose(X4_c) / (X4_c.shape[-1] - 1)
L4, V = np.linalg.eig(cov)
Y4 = np.transpose(V) @ X4_c
covY = Y4 @ np.transpose(Y4) / (Y4.shape[-1] - 1)


# In[66]:


plt.figure()
plt.scatter(np.arange(len(L4)), L4)
# plt.semilogy()
plt.show()


# In[68]:


plt.figure()
plt.plot(np.transpose(Y4)[:, :3])
plt.show()


# In[ ]:




# # Figures

# In[318]:


get_ipython().run_line_magic('matplotlib', 'inline')
mpl.rcParams['figure.dpi'] = 100

width = 10
```

```
height = 6

fig, axs = plt.subplots(2, 2, figsize=(width, height))

plt.subplot(2, 2, 1)
plt.plot(np.transpose(X))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("position", fontsize=13)
plt.title("case 1 recordings")

plt.subplot(2, 2, 2)
plt.plot(np.transpose(X2))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("position", fontsize=13)
plt.title("case 2 recordings")

plt.subplot(2, 2, 3)
plt.plot(np.transpose(X3))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("position", fontsize=13)
plt.title("case 3 recordings")

plt.subplot(2, 2, 4)
plt.plot(np.transpose(X4))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("position", fontsize=13)
plt.title("case 4 recordings")

# plt.legend(["cam 1 y", "cam 1 x", "cam 2 y", "cam 2 x", "cam 3 y", "cam
    3 x"], bbox_to_anchor=(0, -0.5), loc="upper right")
plt.tight_layout()
plt.show()


# In[333]:


width = 10
height = 10

fig, axs = plt.subplots(4, 2, figsize=(width, height))

plt.subplot(4, 2, 1)
plt.scatter(np.arange(len(L)), np.sort(L)[::-1] / (len(L) - 1))
plt.xlabel("index", fontsize=13)
plt.ylabel("$\sigma^2$", fontsize=13)
plt.title("case 1 explained variance")
```

```
plt.subplot(4, 2, 2)
plt.plot(np.transpose(Y[0]))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("projected position", fontsize=13)
plt.title("case 1 principle component projection")



plt.subplot(4, 2, 3)
plt.scatter(np.arange(len(L2)), np.sort(L2)[::-1] / (len(L2) - 1))
plt.xlabel("index", fontsize=13)
plt.ylabel("$\sigma^2$", fontsize=13)
plt.title("case 2 explained variance")

plt.subplot(4, 2, 4)
plt.plot(np.transpose(Y2[[0, 1, 2]]))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("projected position", fontsize=13)
plt.title("case 2 principle component projection")



plt.subplot(4, 2, 5)
plt.scatter(np.arange(len(L3)), np.sort(L3)[::-1] / (len(L3) - 1))
plt.xlabel("index", fontsize=13)
plt.ylabel("$\sigma^2$", fontsize=13)
plt.title("case 3 explained variance")

plt.subplot(4, 2, 6)
plt.plot(np.transpose(Y3[[0, 1]]))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("projected position", fontsize=13)
plt.title("case 3 principle component projection")



plt.subplot(4, 2, 7)
plt.scatter(np.arange(len(L4)), np.sort(L4)[::-1] / (len(L4) - 1))
plt.xlabel("index", fontsize=13)
plt.ylabel("$\sigma^2$", fontsize=13)
plt.title("case 4 explained variance")

plt.subplot(4, 2, 8)
plt.plot(np.transpose(Y4[[0, 1, 2]]))
plt.xlabel("t (frames)", fontsize=13)
plt.ylabel("projected position", fontsize=13)
```

```
plt.title("case 4 principle component projection")


plt.tight_layout()
plt.show()


# In[69]:


sum(L[:1]) / sum(L)


# In[325]:


sum(L2[:3]) / sum(L2)


# In[328]:


sum(L3[:2]) / sum(L3)


# In[327]:


sum(L4[:3]) / sum(L4)
```

References

[1] Mallen, Alex Troy. GitHub source code.
https://github.com/AlexTMallen/AMATH-582/tree/master/HW3

[2] Kutz, Jose Nathan. Lecture: Principal Componenet Analysis (PCA)
https://www.youtube.com/watch?v=a9jdQGybYmE

[3] Scipy convolve.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html

[4] Numpy eig.
https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html