AMATH 482 Homework 3

Vineet Palepu

February 24, 2021

1. Abstract

In this paper, we are tasked with tracking and analyzing the motion of a can suspended on a spring. We are given four different cases, each with slightly different motions. In each case, we are given three videos, each from different angles, and are tasked with finding the position of the can in each camera, and then use Principal Component Analysis to remove redundant information about the motion of the can as well as make conclusions about how the can moves.

2. Introduction and Overview

We are given four cases, each containing three videos from different angles of a can oscillating on a spring. Each of the cases has slightly different motion: case 1 consists of purely vertical oscillation, case 2 also has only vertical oscillation but also includes random camera shake, case 3 has both vertical oscillation as well as horizontal movement, and finally case 4 has vertical movement and rotational movement. We are tasked with gathering the position of the can in each video, and from the 3 sets of position data for each case, performing Principal Component Analysis in order to determine how the can moves through space. We then plot the energies to determine which principal components are significant, and thus make conclusions about the can's movement.

3. Theoretical Background

3.1. Singular Value Decomposition

The Singular Value Decomposition (SVD) is a matrix factorization of the form shown in Equation (1), where U and V are unitary matrices and Σ is a diagonal matrix. Because unitary matrices do not stretch or compress a vector and diagonal matrices only stretch vectors, the SVD can be visualized as a rotation, followed by a dilation, followed by another rotation.

$$A = U\Sigma V^* \tag{1}$$
$$A \in \mathbb{R}^{m\times n}, U \in \mathbb{R}^{m\times m}, \Sigma \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}$$

In the SVD, U is the matrix of left singular vectors of A, V is the matrix of right singular vectors of A, and Σ is the matrix of singular values of A, with the singular values located on the diagonal. Note also that while V is the matrix of right singular vectors, the formula uses V^*, the conjugate transpose, which is the transpose of a matrix followed by taking the complex conjugate of each element. Additionally, the singular values are ordered from largest to smallest, thus ensuring that every matrix has a unique SVD. Important to note is that all matrices, no matter the size or shape, have a singular value decomposition.

Further, the SVD is useful for forming low-rank approximations of a matrix. If the matrix A is $m \times n$ and $rank(A) = r$, then it can be expressed as shown in Equation (2), where $u_i v_i^*$ is the outer product of $m \times 1$ and $1 \times n$ matrices. Further, if we instead sum up to N instead of r, where $N \leq r$, we can form the best rank N approximation of A.

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^* \tag{2}$$

In order to compute the SVD of a matrix, Equation (3) shows us that V and Σ^2 are the eigenvalues and eigenvectors of A^*A. This can also be done for AA^* as shown in Equation (4), showing that U and Σ^2 are the eigenvalues and eigenvectors of AA^*.

$$\begin{aligned} A^*A &= (U\Sigma V^*)^*(U\Sigma V^*) \\ &= V\Sigma U^* U\Sigma V^* \\ &= V\Sigma^2 V^* \\ A^*AV &= V\Sigma^2 \end{aligned} \tag{3}$$

$$AA^* = U\Sigma^2U^* \quad (4)$$
$$AA^*U = U\Sigma^2$$

3.2. Principal Component Analysis

Our main use of the SVD will be in Principal Component Analysis (PCA). PCA is a method that allows us to find a new basis for our data such that each new axis is uncorrelated.
If we are given a vector of data, a, we can calculate the variance σ^2 as shown in Equation (5). Further, we can calculate the covariance, which measures how variables vary with respect to each other, as in Equation (6). If the covariance is 0, the two variables are uncorrelated. An important detail is that we must subtract the mean from our data when doing PCA. This allows our equations for the variance and covariance to be correct.

$$\sigma_a^2 = \frac{1}{n-1}aa^T \quad (5)$$

$$\sigma_{ab}^2 = \frac{1}{n-a}ab^T \quad (6)$$

Now, if we are given a matrix constructed with row vectors as shown in Equation (7), we can compute all combination of covariances using the matrix $C_X = \frac{1}{n-1}XX^T$. We then diagonalize the matrix as shown in Equation (8) so that the off-diagonal values, the covariances, are all zero. The eigenvectors found in V are called the principal components, and the eigenvalues are the variances.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \quad (7)$$

$$C_X = V\Lambda V^{-1} \quad (8)$$

If we let $A = \frac{1}{\sqrt{n-1}}X$, then we see that $AA^T = U\Sigma^2U^T$, where U is our left singular vectors and Σ is our singular values. In order to transform our data to our new principal component basis, we can use Equation (9).

$$Y = U^TX \quad (9)$$

Equation (10) shows that the covariance matrix of our transformed data is equal to Σ^2, meaning that each variable in Y is uncorrelated since the off-diagonals are 0.

$$C_Y = \frac{1}{n-1}YY^T = \frac{1}{n-1}U^TXX^TU = U^TAA^TU = U^TU\Sigma^2U^TU = \Sigma^2 \quad (10)$$

Finally, from our singular values, we can calculate the energy of each principal component, which is related to how much each principal component contributes to the overall data. A high energy means that a particular principal component accounts for a large amount of the variance in the data. The energy for the n^{th} singular value can be calculated as shown in Equation (11).

$$E_{\sigma_k} = \frac{\sigma_k^2}{\sum_i^n \sigma_i^2} \quad (11)$$

4. Algorithm Implementation and Development

Our first step was to load the data and then scale it to values between 0 and 1. Once that was done, we needed to extract the position of the can from each camera feed and store that location. This code can be found in the function $get\_pos()$ in the Helper Functions section in Appendix B. In the videos, the camera has a flashlight leading to a bright spot on top of the camera. However, due to

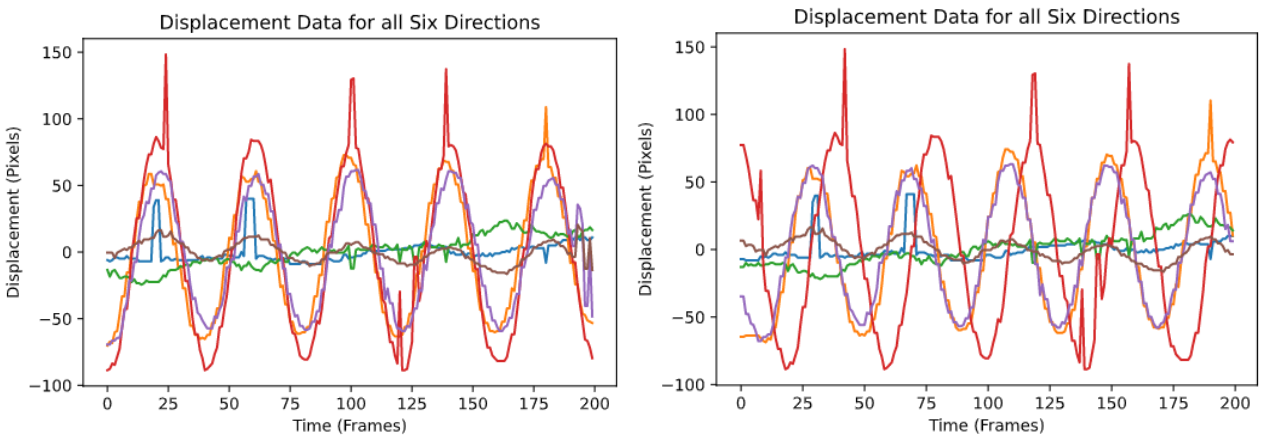

Figure 1. Here we can see the aligned (left) and unaligned versions of the displacement data. Alignment was done by finding the minimum value of the largely oscillating dimension of each camera, and then the data was clipped from that point till the end.

lighting and reflections, there are often multiple locations that have very high brightness. Thus, we used the fact that motion is relatively smooth, and thus the position in each frame should be somewhat close to the position in the previous frame. By reducing the area we search, we end up with a more accurate location estimate. So, we manually find the can and give an approximate position for the first frame. For subsequent frames, we create a mask that is the same size as our photo. The code for generating the mask can be found in the function $get\_mask()$ in the Helper Functions section in Appendix B. In the mask, all pixels are set to 0. We then use the previous location of our can and form a box around that location. Our box was 200 pixels by 200 pixels. In that 200×200 pixel box, we set the value for the mask equal to 1. We can then multiply our mask by our greyscale image to get a new image that is 0 everywhere except in the region near the previous location. We then find the pixel with maximum value and use that as the location of our can. This gave much better results than just searching the entire image, as reflections and bright spots far away from the can are completely ignored. There still are some instances where the bottom of the can is detected instead of the flashlight, but the position data is relatively smooth.

We did this for each of the three cameras, giving us a total of 3 x and y position vectors. We then attempt to align the three videos temporally by looking at the graphs of the position data, finding the frame with the minimum value, and then using that frame as the starting point, shown in Figure 1. We then clip the ends of these vectors so we are left with 6 equal length position vectors. We then form these into a matrix and calculate the SVD. We also plotted the graphs of each of the position vectors on the same scale to check and make sure they are approximately aligned. We then calculate the energy values of each of the principal components and plot them. Finally, we project our data onto the principal component basis. This procedure was repeated for all 4 sets of videos, with the exception that we didn't attempt to align the video for cases 3 and 4 as it was difficult to do from the positional graphs.

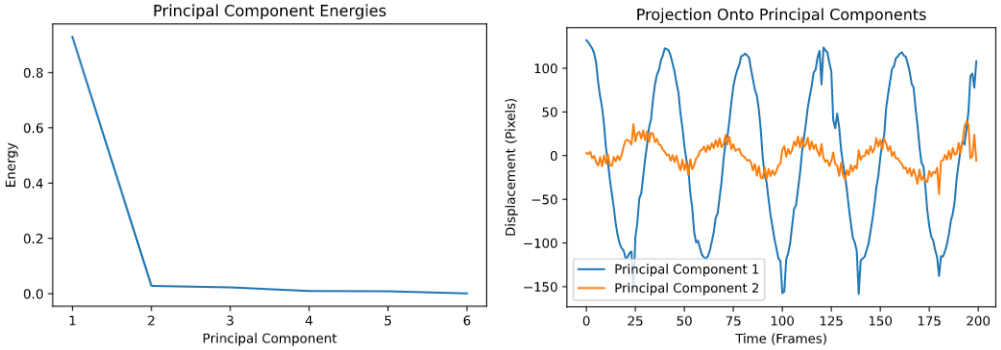

Figure 2. Here we have the results from Case 1. The energies of the principal components (left) quickly drop off since the first principal component is the only significant one. On the projection (right) we see that the first principal component represents the vertical oscillation of the can, and the second principal component is insignificant.

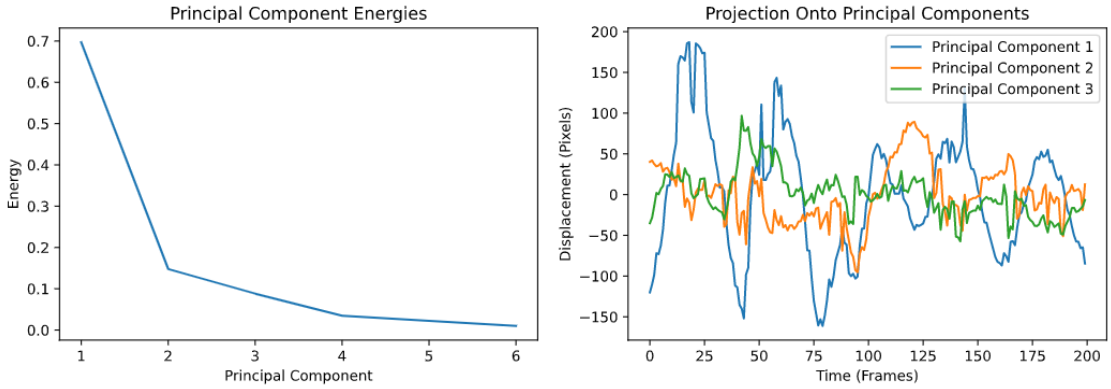

Figure 3. Here we have the results from Case 2. The principal component energies (left) again quickly drop off, however the second component is slightly larger when compared to Case 1. The projection of the data (right) shows a less clear oscillation in the first principal component. The second could be oscillations due to the camera shaking, but could also just be noise. The third principal component also appears to be noise.

5. Computational Results

For the video set from case 1, we found that the energy for our first principal component was $.929$, and our second principal component had an energy of $.028$. The plot of principal component energies can be seen in Figure 2 This means that our data was essentially of rank 1, since our second principal component was small enough to be considered noise. Further, projecting our data onto our first principal component shows motion consistent with a mass oscillating only vertically. We also showed the projection onto the second principal component, and saw oscillations between -25 to $+25$ pixels. These projections can be seen in Figure 2 This could be from any slight side to side motion of the can or cameras.

For the video set from case 2, we found the energy for our first principal component was $.697$, the second was $.148$, and the third was $.088$. The graph of principal component energies is shown in Figure 3, along with the data projected onto the principal components. Again, we assumed that only the first principal component was significant. In this case, compared to case 1, the first principal value is much smaller, and the second is much larger. Unlike case 1, where the video was relatively stable, in this case, there were shakes in the camera, which are likely what the second principal component was detecting. Even though our third principal value still had a relatively high value, we considered it noise, especially after plotting. The graph of our data projected onto each of the first three principal components helps clear this up. The first clearly shows large oscillations that are likely to be the up and down movement of the can. The second shows some higher frequency lower amplitude oscillation that could be a result of the camera shaking, however it isn't entirely clear, and could just be noise. Finally, the plot of the third principal component has no apparent pattern and thus is likely just additional noise.

For case 3, we found the energy of the first principal component to be $.487$, the second was $.218$, the third was .156, and the fourth was .097. The graph of principal component energies is seen in Figure 4, along with the projection of the data onto the principal components. Here, we assume our first two principal components to be significant. The first shows large oscillations we assume to represent the vertical movement. The second shows smaller amplitude oscillations at a slightly lower frequency, which we presume to be the horizontal motion by comparing to the video. Even though the third and fourth principal components have higher values, we couldn't see any pattern and assume them to be noise. As our position data from this case had many more miscalculations of position, that is to be expected.

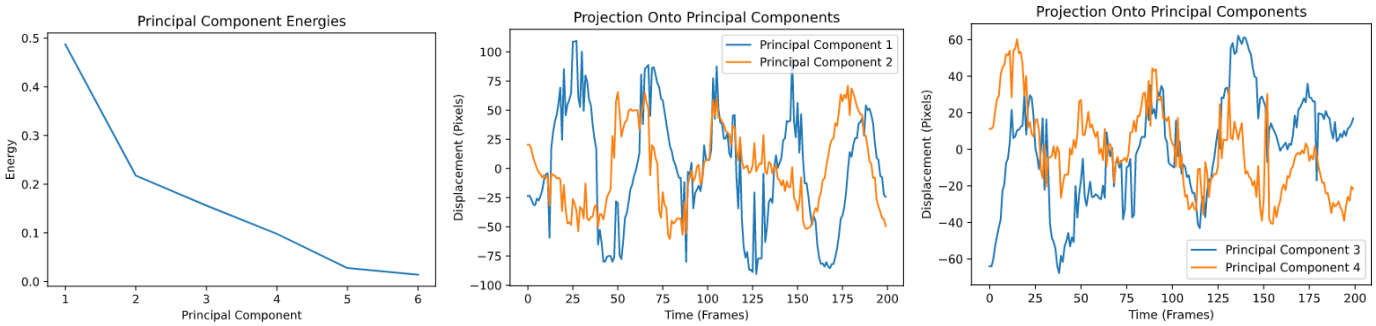

Figure 4. Here we have the results for Case 3. The principal component energy (left) is lower for the first few, likely due to the fact that our can now was moving in multiple dimensions. The projection onto the first two principal components (middle) shows clear oscillation for both components, which we predict to be due to the vertical and horizontal oscillation. The projection onto the third and fourth principal components (right) does show oscillation, but due to the low energies we concluded these were due to noise.

Finally, for case 4 we found the energies of the first four principal components to be $.52, .268, .128,$ and $.044$. The graph of the energies is shown in Figure 5, along with the projection of the data onto the principal components. The results for this case were comparable to case 3, albeit slightly better since our position detection had fewer mistakes than in case 3. Our first component has large oscillations which correspond with the vertical movement of the can. Our second component appears very similar to the first, with a slightly smaller amplitude. We interpreted this as the side to side motion of the flashlight spot as a result of the can's rotation. However, it should be noted that we again can't be sure. Just as in case 3, we believe the third and higher components to be noise as we couldn't deduce any pattern that could occur from some physical phenomenon.

Throughout the cases, it makes sense that then number of significant principal components corresponds with the number of dimensions the can is moving through. A matrix with only one significant principal component has a rank of 1, which would correspond to one dimensional data. Since the can only moves in one dimension in cases 1 and 2, this makes sense. It also follows that in case 2 we see a larger second principal component as the camera shakes are uncorrelated with the motion of the can. In cases 3 and 4, we see two significant principal components, which would correspond to a matrix of rank 2. Again, this makes sense since we have the vertical motion of the can as the first component, and the uncorrelated horizontal motion or rotation of the can for the second significant component.

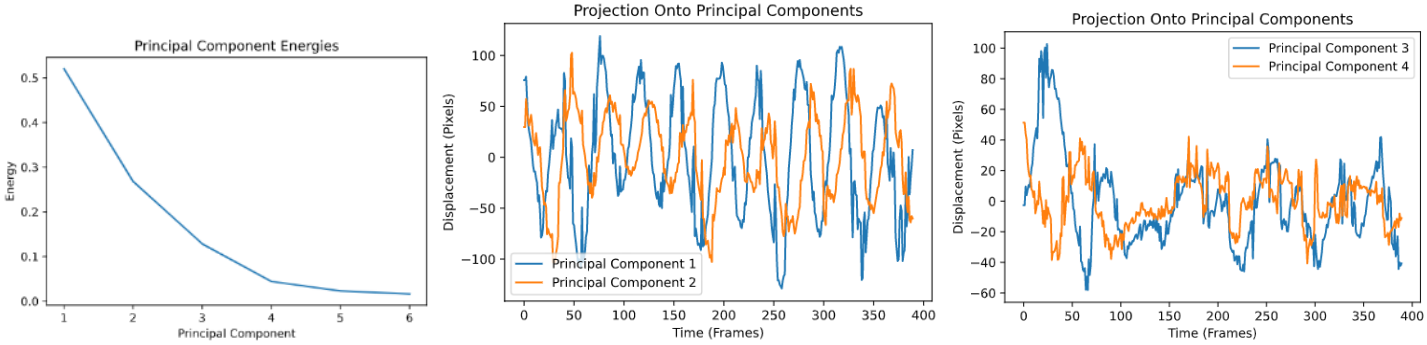

Figure 5. Here we can see the results from Case 4. The principal component energies (left) are similar to those from Case 3. Again, from the first two principal component projections (middle) we see clear oscillation, which we predict to be from the vertical oscillation and the rotation of the can. The projections onto the third and fourth principal components (right) are believed to be noise.

6. Summary and Conclusions

In the paper, we are given four different cases. For each case, we have three different camera views, and are tasked with performing PCA on positional data of a can on a spring to determine how it moves. By plotting the energies of each of the principal components, we were able to successfully determine the number of dimensions through which the can moves. Then, by projecting our data onto the principal component basis vectors, we were able to correlate the projected data with the motion of the can and then give some general descriptions of how it moves through those dimensions. By using PCA, we can eliminate redundancy in a dataset and reduce the dimensionality to the minimum required.

7. Appendix A: Functions Used

Python

NumPy

`np.abs` : Returns the absolute value of a matrix.

`np.amax` : Returns the largest value of a matrix. Used to normalize certain matrices.

`np.arange` : Generates a vector with consecutive integer values.

`np.argmax` : Gets the index of the maximal element

`np.linalg.svd` : Calculates the SVD of a matrix. Used to get the singular values and U matrix for PCA.

`np.mean` : Calculates the mean of a values of a matrix along a specified axis. Used to generate the grayscale image from RGB images.

`np.vstack` : Verticall concatenates matrices. Used to construct our data matrix from the individual position data vectors.

MatPlotLib

`pyplot.plot` : Creates a graph of the data. We used this to visualize various data.

SciPy

`io.loadmat` : Used to load a MATLAB matrix into Python

8. Appendix B: Code

```
## Imports
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt

## Helper Functions
# Converts rgb image to grayscale
def rgb2gray(img):
    return np.mean(img, axis=-1)

# Returns the x y coordinates of the max value of a matrix
def argmax_xy(img):
    return np.flip(np.unravel_index(np.argmax(img), img.shape))

# Constructs a image mask of a certain size centered around a specific location
def get_mask(img, center, size=100):
    mask = np.zeros(img.shape)
    box_x_low = center[1] - size
    box_x_high = center[1] + size
    box_y_low = center[0] - size
    box_y_high = center[0] + size

    box_x_low = 0 if box_x_low < 0 else box_x_low
    box_x_high = mask.shape[1] if box_x_high > mask.shape[1] else box_x_high
    box_y_low = 0 if box_y_low < 0 else box_y_low
    box_y_high = mask.shape[0] if box_y_high > mask.shape[0] else box_y_high

    mask[box_x_low:box_x_high, box_y_low:box_y_high] = 1
    return mask

# Constructs the position vector from camera data
def get_pos(cam, initial_pos, debug_frame=-1):
    pos = np.zeros((2, cam.shape[-1]), dtype=int)
    prev_pos = initial_pos
    for i in range(cam.shape[-1]):
        img = cam[:,:,:,i]
        gray = rgb2gray(img)

        mask = get_mask(gray, prev_pos)

        pos[:, i] = argmax_xy(gray * mask)

        if debug_frame != -1 and i == debug_frame:
            plt.plot(pos[0, i], pos[1, i], 'bx') # Current Pos
            plt.plot(pos[0, i - 1], pos[1, i - 1], 'ro') # Previous Pos, also cen
ter of filter
            plt.imshow(gray * mask, cmap='gray')
            plt.show()
```

```
        prev_pos = pos[:, i]

    return pos

## Case 1

# %%
cam1_1 = loadmat('cam1_1.mat')['vidFrames1_1'] / 255
cam2_1 = loadmat('cam2_1.mat')['vidFrames2_1'] / 255
cam3_1 = loadmat('cam3_1.mat')['vidFrames3_1'] / 255


# %%
pos1 = get_pos(cam1_1, (325, 225))
pos2 = get_pos(cam2_1, (275, 250))
pos3 = get_pos(cam3_1, (325, 325))


# %%
plt.plot(np.arange(pos1.shape[1]), pos1[1,:])
plt.show()
plt.plot(np.arange(pos2.shape[1]), pos2[1,:])
plt.show()
plt.plot(np.arange(pos3.shape[1]), pos3[0,:])
plt.show()


# %%
# Find bottom of oscillation to align videos
v1_start = np.argmin(pos1[1, 0:25])
v2_start = np.argmin(pos2[1, 0:40])
v3_start = np.argmin(pos3[0, 0:25]) # Cam 3 has x-
y axes flipped, vertical motion on x axis

# Change the start of each to align
pos_mat = np.vstack((pos1[:, v1_start:v1_start + 200], pos2[:, v2_start:v2_start
+ 200], pos3[:, v3_start:v3_start + 200]))
pos_mat = (pos_mat - np.mean(pos_mat, axis=1)[:, None])

# Visually check approximate alignment
for i in range(pos_mat.shape[0]):
    plt.plot(np.arange(pos_mat[i, :].shape[0]), pos_mat[i, :])
plt.title("Displacement Data for all Six Directions")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.show()


# %%
U, S, V = np.linalg.svd(pos_mat / np.sqrt(pos_mat.shape[1] - 1))
```

```
S_energy = S**2 / np.sum(S**2)
print(S_energy)
plt.plot(np.arange(1, 7), S_energy)
plt.title("Principal Component Energies")
plt.xlabel("Principal Component")
plt.ylabel("Energy")
plt.show()


# %%
# Project Data onto our PC Basis
Y = U.T @ pos_mat
plt.plot(Y[0, :])
plt.plot(Y[1, :])
plt.title("Projection Onto Principal Components")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.legend(("Principal Component 1", "Principal Component 2"))
plt.show()

## Case 2

# %%
cam1_1 = None
cam2_1 = None
cam3_1 = None

cam1_2 = loadmat('cam1_2.mat')['vidFrames1_2']
cam2_2 = loadmat('cam2_2.mat')['vidFrames2_2']
cam3_2 = loadmat('cam3_2.mat')['vidFrames3_2']


# %%
pos1 = get_pos(cam1_2, (325, 300))
pos2 = get_pos(cam2_2, (300, 350))
pos3 = get_pos(cam3_2, (350, 250))


# %%
plt.plot(pos1[0, :])
plt.show()
plt.plot(pos1[1, :])
plt.show()
plt.plot(pos2[0, :])
plt.show()
plt.plot(pos2[1, :])
plt.show()
plt.plot(pos3[0, :])
plt.show()
plt.plot(pos3[1, :])
plt.show()
```

```
# %%
v1_start = np.argmin(pos1[1, 0:50])
v2_start = np.argmin(pos2[1, 0:30])
v3_start = np.argmin(pos3[0, 0:50]) # Cam 3 has x-
y axes flipped, vertical motion on x axis
# Use these to align the videos
pos_mat = np.vstack((pos1[:, v1_start:v1_start + 200], pos2[:, v2_start:v2_start
+ 200], pos3[:, v3_start:v3_start + 200]))
pos_mat = (pos_mat - np.mean(pos_mat, axis=1)[:, None])

# Visually check approximate alignment
for i in range(pos_mat.shape[0]):
    plt.plot(np.arange(pos_mat[i, :].shape[0]), pos_mat[i, :])

plt.title("Displacement Data for all Six Directions")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.show()


# %%
U, S, V = np.linalg.svd(pos_mat / np.sqrt(pos_mat.shape[1] - 1))
S_energy = S**2 / np.sum(S**2)
print(S_energy)
plt.plot(np.arange(1, 7), S_energy)
plt.title("Principal Component Energies")
plt.xlabel("Principal Component")
plt.ylabel("Energy")
plt.show()


# %%
# Project Data onto our PC Basis
Y = U.T @ pos_mat
plt.plot(Y[0, :])
plt.plot(Y[1, :])
plt.plot(Y[2, :])

plt.title("Projection Onto Principal Components")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.legend(("Principal Component 1", "Principal Component 2", "Principal Componen
t 3"))
plt.show()

## Case 3

# %%
cam1_2 = None
```

```
cam2_2 = None
cam3_2 = None

cam1_3 = loadmat('cam1_3.mat')['vidFrames1_3']
cam2_3 = loadmat('cam2_3.mat')['vidFrames2_3']
cam3_3 = loadmat('cam3_3.mat')['vidFrames3_3']


# %%
pos1 = get_pos(cam1_3, (325, 300))
pos2 = get_pos(cam2_3, (300, 350))
pos3 = get_pos(cam3_3, (350, 250))


# %%
plt.plot(pos1[0, :])
plt.show()
plt.plot(pos1[1, :])
plt.show()
plt.plot(pos2[0, :])
plt.show()
plt.plot(pos2[1, :])
plt.show()
plt.plot(pos3[0, :])
plt.show()
plt.plot(pos3[1, :])
plt.show()


# %%
v1_start = 0#np.argmin(pos1[1, 0:50])
v2_start = 0#np.argmin(pos2[1, 0:30])
v3_start = 0#np.argmin(pos3[0, 0:50]) # Cam 3 has x-
y axes flipped, vertical motion on x axis
# Use these to align the videos
pos_mat = np.vstack((pos1[:, v1_start:v1_start + 200], pos2[:, v2_start:v2_start
+ 200], pos3[:, v3_start:v3_start + 200]))
pos_mat = (pos_mat - np.mean(pos_mat, axis=1)[:, None])

# Visually check approximate alignment
for i in range(pos_mat.shape[0]):
    plt.plot(np.arange(pos_mat[i, :].shape[0]), pos_mat[i, :])
plt.title("Displacement Data for all Six Directions")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.show()


# %%
U, S, V = np.linalg.svd(pos_mat / np.sqrt(pos_mat.shape[1] - 1))
S_energy = S**2 / np.sum(S**2)
```

```
print(S_energy)
plt.plot(np.arange(1, 7), S_energy)
plt.title("Principal Component Energies")
plt.xlabel("Principal Component")
plt.ylabel("Energy")
plt.show()


# %%
# Project Data onto our PC Basis
Y = U.T @ pos_mat
plt.plot(Y[0, :])
plt.plot(Y[1, :])

plt.title("Projection Onto Principal Components")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.legend(("Principal Component 1", "Principal Component 2"))
plt.show()

plt.plot(Y[2, :])
plt.plot(Y[3, :])
plt.title("Projection Onto Principal Components")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.legend(("Principal Component 3", "Principal Component 4"))
plt.show()

## Case 4

# %%
cam1_3 = None
cam2_3 = None
cam3_3 = None

cam1_4 = loadmat('cam1_4.mat')['vidFrames1_4']
cam2_4 = loadmat('cam2_4.mat')['vidFrames2_4']
cam3_4 = loadmat('cam3_4.mat')['vidFrames3_4']


# %%
pos1 = get_pos(cam1_4, (400, 325))
pos2 = get_pos(cam2_4, (250, 250))
pos3 = get_pos(cam3_4, (350, 250))


# %%
plt.plot(pos1[0, :])
plt.show()
plt.plot(pos1[1, :])
plt.show()
```

```
plt.plot(pos2[0, :])
plt.show()
plt.plot(pos2[1, :])
plt.show()
plt.plot(pos3[0, :])
plt.show()
plt.plot(pos3[1, :])
plt.show()


# %%
v1_start = 0#np.argmin(pos1[1, 0:50])
v2_start = 0#np.argmin(pos2[1, 0:30])
v3_start = 0#np.argmin(pos3[0, 0:50]) # Cam 3 has x-
y axes flipped, vertical motion on x axis
# Use these to align the videos
pos_mat = np.vstack((pos1[:, v1_start:v1_start + 390], pos2[:, v2_start:v2_start
+ 390], pos3[:, v3_start:v3_start + 390]))
pos_mat = (pos_mat - np.mean(pos_mat, axis=1)[:, None])

# Visually check approximate alignment
for i in range(pos_mat.shape[0]):
    plt.plot(np.arange(pos_mat[i, :].shape[0]), pos_mat[i, :])
plt.title("Displacement Data for all Six Directions")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.show()


# %%
U, S, V = np.linalg.svd(pos_mat / np.sqrt(pos_mat.shape[1] - 1))
S_energy = S**2 / np.sum(S**2)
print(S_energy)
plt.plot(np.arange(1, 7), S_energy)
plt.title("Principal Component Energies")
plt.xlabel("Principal Component")
plt.ylabel("Energy")
plt.show()


# %%
# Project Data onto our PC Basis
Y = U.T @ pos_mat
plt.plot(Y[0, :])
plt.plot(Y[1, :])

plt.title("Projection Onto Principal Components")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.legend(("Principal Component 1", "Principal Component 2"))
plt.show()
```

```
plt.plot(Y[2, :])
plt.plot(Y[3, :])
plt.title("Projection Onto Principal Components")
plt.xlabel("Time (Frames)")
plt.ylabel("Displacement (Pixels)")
plt.legend(("Principal Component 3", "Principal Component 4"))
plt.show()
```