Principal Components Analysis (PCA)

Brandon Goldney[1]

[1]*Department of Applied Mathematics, University of Washington, Email: Goldney@uw.edu*

Abstract

The purpose of this paper is to demonstrate the effectiveness of principal components analysis. PCA is applied on a series of four different experiments, each experiment being comprised of three videos. In all four experiments a paint can is recorded as it is displaced and simultaneously recorded by three different cameras. This is analogous to the classic mass-spring system in physics. The objective is to track the paint can and apply PCA to reduce noise which is introduced throughout the videos.

1 Section I: Introduction and Overview

Each of the four experiments in this paper pose a unique challenge. A brief overview of each experiment is provided here:

1. **Ideal Case** - A small displacement of the mass in the z direction and the ensuing oscillations. In this case, the entire motion is in the z directions with simple harmonic motion being observed.
2. **Noisy Case** - Repeat the ideal case experiment, but this time, introduce camera shake into the video recording. This should make it more difficult to extract the simple harmonic motion. But if the shake isn't too bad, the dynamics will still be extracted with the PCA algorithms
3. **Horizontal Displacement** - In this case, the mass is released off-center so as to produce motion in the $x\,y$ plane as well as the z direction. Thus there is both a pendulum motion and a simple harmonic oscillations.
4. **Horizontal Displacement and Rotation** - In this case, the mass is released off-center and rotates so as to produce motion in the $x\,y$ plane, rotation as well as the z direction. Thus there is both a pendulum motion and a simple harmonic oscillations.

In each case, there are three cameras in different locations recording the movement of the paint can. Each camera provides a standard colorized image. We'll transform the RBG pixels of each frame to grey scale and utilize pixel intensity to track the coordinates of the paint can. Leveraging that information, we will reduce the dimensionality of the problem and project the coordinates of the paint can onto the new orthonormal bases.

2 Section II: Theoretical Background

PCA can be applied using Singular Value Decomposition (SVD) or eigendecomposition. The eigen-decomposition method will be briefly explained as it provides a clear and intuitive approach to solving PCA.

Eigendecomposition The principle components can be thought of in terms of the covariance matrix. The diagonal entries of the covariance matrix are the variances for the i^{th} feature; therefore, terms with a largest magnitude explained the most variance. Similarly, terms on the diagonal with smaller magnitude explain less variance. Extending that understanding to terms in the off-diagonal, terms with a large magnitude reflect two features which have a high degree of redundancy. It logically follows that the dimensionality of the system can be reduced to those features which can explain the most variance (i.e. largest terms on the diagonal).

The covariance matrix is

$$C_x = \frac{1}{n-1} X X^T \tag{1}$$

"where the matrix X contains the data from the experiments, and $X \in C^{mxn}$ where m is the number of the probes or measuring positions, and n is the number of experimental data points."

Subsequently, the covariance matrix can be diagonalized by acknowledging the fact that XX^T is a square, symmetric m x m matrix.

$$XX^T = S\Lambda S^{-1} \tag{2}$$

Additionally, since it is a symmetric matrix the S can be written as a unitary matrix with $S^{-1} = S^T$, and Λ is a diagonal matrix whose entries correspond to the m distinct eigenvalues of XX^T. Therefore, we can write the following:

$$Y = S^T X \tag{3}$$

Now that we're in a new basis, we can calculate the covariance.

$$\begin{aligned} C_Y &= \frac{1}{n-1} YY^T \\ &= \frac{1}{n-1} \\ &= \frac{1}{n-1} S^T (XX^T) S \\ &= \frac{1}{n-1} \Lambda \end{aligned}$$

In the new form, the principal components are the eigenvectors of XX^T.

SVD

Based off that (hopefully) intuitive understanding of principal components, we'll look into SVD in greater detail.
By leveraging the appropriate pair of bases U and V, SVD can diagonalize any matrix.

$$Y = U * X \tag{4}$$

where U is the unitary transformation associated with the SVD: $X = U\Sigma V*$. In this new form, we can calculate the variance:

$$\begin{aligned} C_Y &= \frac{1}{n-1} YY^T \\ &= \frac{1}{n-1} (U * X)(U * X)^T \\ &= \frac{1}{n-1} U^* (XX^T) U \\ &= \frac{1}{n-1} U * U\Sigma^2 UU^* \\ &= \frac{1}{n-1} \Sigma^2 \end{aligned}$$

We can see that $\Sigma^2 = \Lambda$ from the eigendecomposition method.

3 Section III: Algorithm Implementation and Development

The PCA implementation in these experiments can be viewed as having two steps. The first step is to process the data into a digestable format so the coordinates of the paint can are able to be easily tracked. The second step is the actual PCA component, reducing the dimensionality of the 3 videos for each experiment.

Step 1: Processing data

We'll mainly use the first test case as the example because the methodology is nearly identical in the other cases. First, we need to determine the coordinates of the paint can in the first picture. We do this in Matlab, leveraging a built-in GUI allowing the user to click on an area and the GUI returns the x and y coordinates. Subsequently, we convert each image from RGB values to greyscale. The advantage of this is that it allows us to focus on one number rather than three (i.e. RGB values). Also, the paint can has a flashlight fixed to the top of it; therefore, we're able to simply extract the maximum value from the area around the paint can in order to ascertain the coordinates.

This raises a new issue: the flashlight is not always the brightest pixel in the image, bringing us back to the first step where we extracted the coordinates of the flashlight. The advantage of manually extracting the first image is that we're able to set a window around the flashlight and set all other values to zero (i.e. black). In other words, we determine the coordinates of the flashlight in the first image, set a window of 20 pixels around the coordinates, then set all other pixels to zero. We then search the next image within that window, and then update the coordinates of the window, and repeat the process. This step is important to reduce computing time, as well as accurately tracking the paint can. An example of this step is shown below.

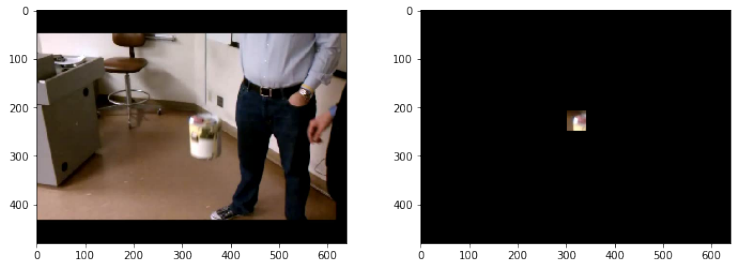


Figure 1. Left: Normal Image Right: Image with window applied

Step 2: PCA

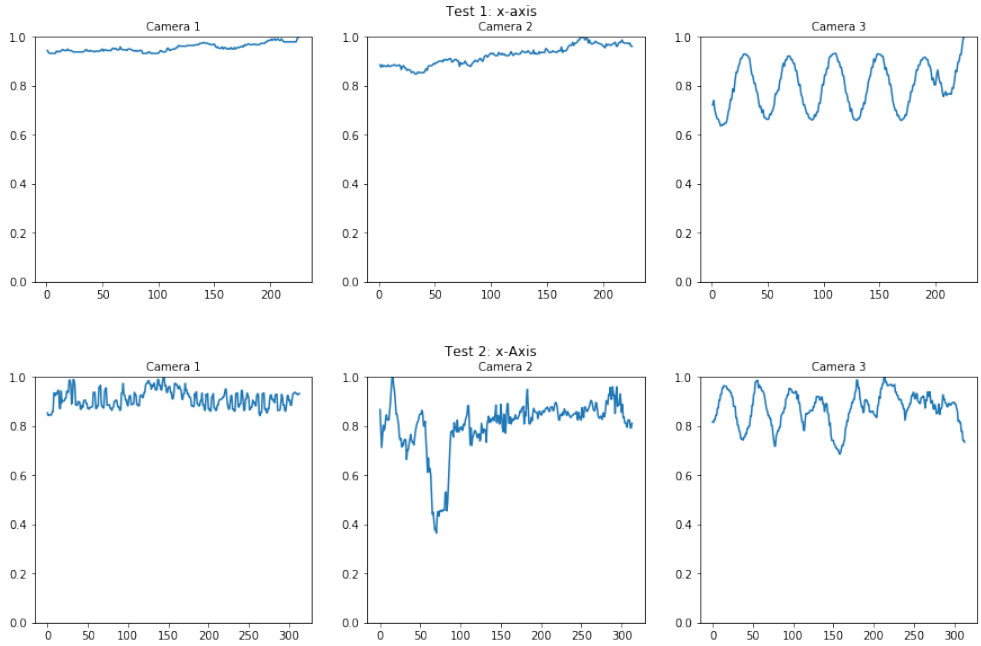
The process for calculating the principle components is fairly straightforward. We'll walk through the implementation of the algorithm, but defer topics pertaining to the "why" to Section 1. After normalizing the data, the covariance matrix is calculated. Subsequently, the SVD of the covariance matrix is calculated. At this point, we have all the information we need. The final step is to manipulate the data so it's in a digestible format. To better understand PCA, we look at the explained variance, which is similar to an R^2 score in linear regression. Explained variance shows the amount of variance explained by each component compared to the total variance.

4 Section IV: Computational Results

The computational results are most easily viewed through the lens of explained variance. However, prior to viewing the output, it's helpful to view the movement of the paint can along the x and y axis.

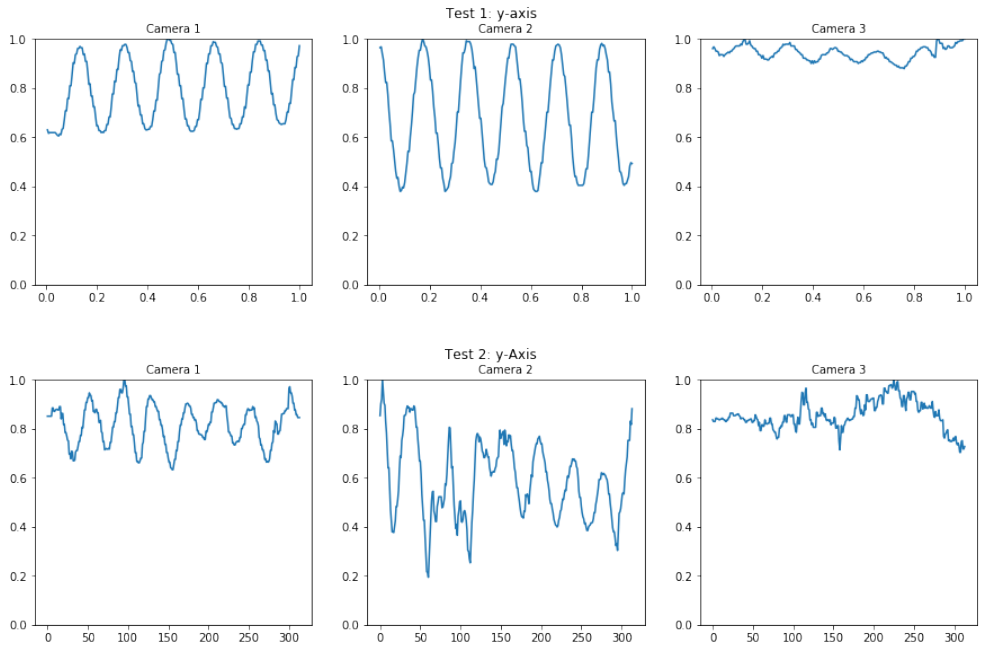
The Ideal Case and Noisy Case (i.e. tests 1 and 2) are shown first then the Horizontal Displacement and Horizontal Displacement with Rotation (i.e. test 3 and 4).

Ideal Case and Noisy Case: x-axis



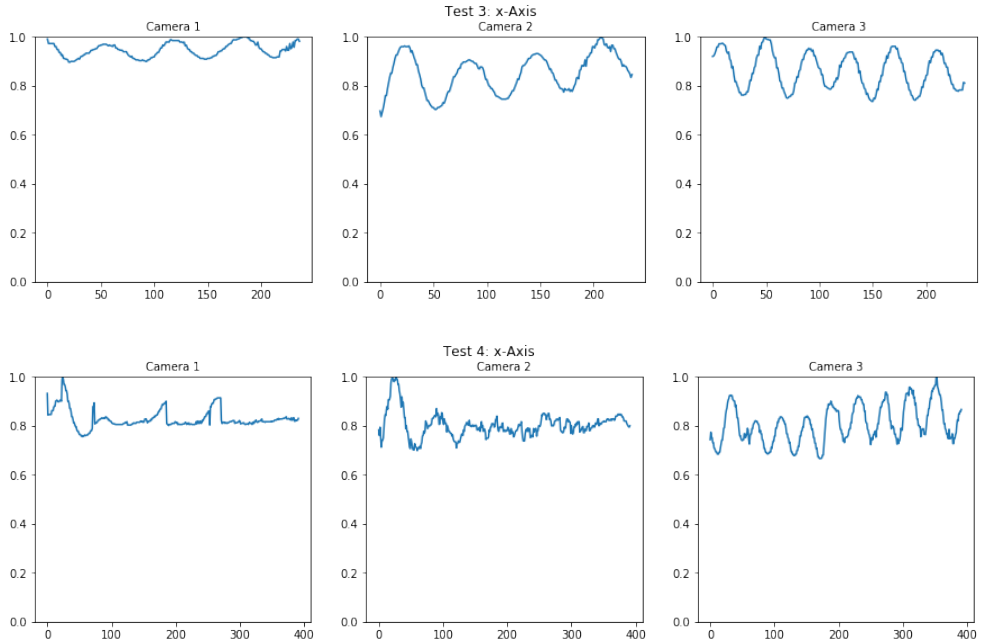
As expected in the Ideal Case, the x-axis has minimal volatility as the paint can was only moved in the vertical direction. However, notice the third camera demonstrates harmonic motion. This is because the camera was turned 90 degrees - this is evidenced by the relatively flat line for the y-axis (shown below). Additionally, we can see in the Noisy Case that the x-axis demonstrates some more volatility, due to noise being introduced.

Ideal Case and Noisy Case: y-axis



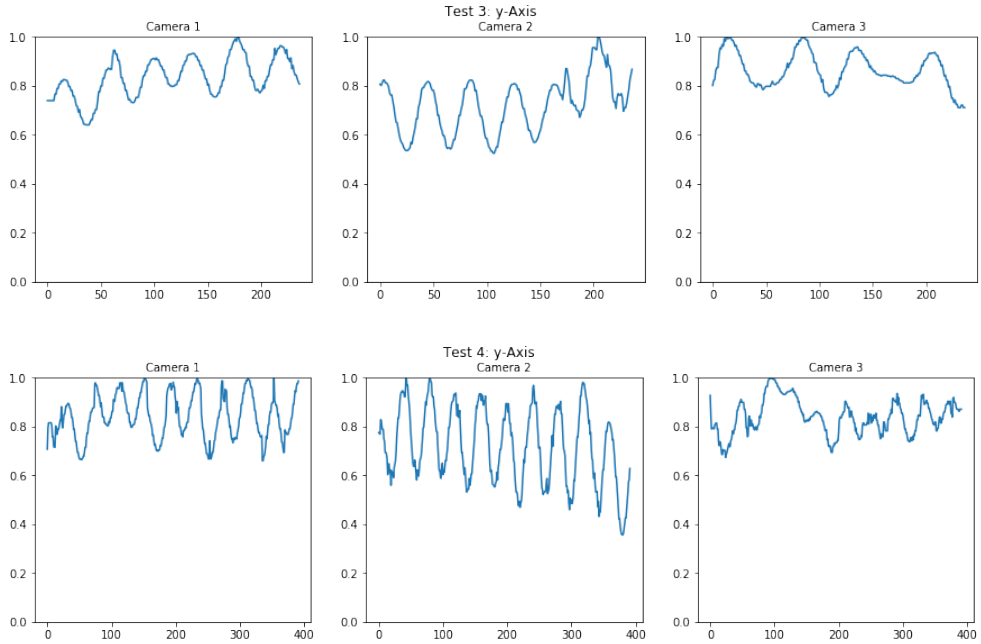
Again, we see the y-axis for the Noisy Case shows more volatility.

Horizontal Displacement and Horizontal Displacement with Rotation: x-axis

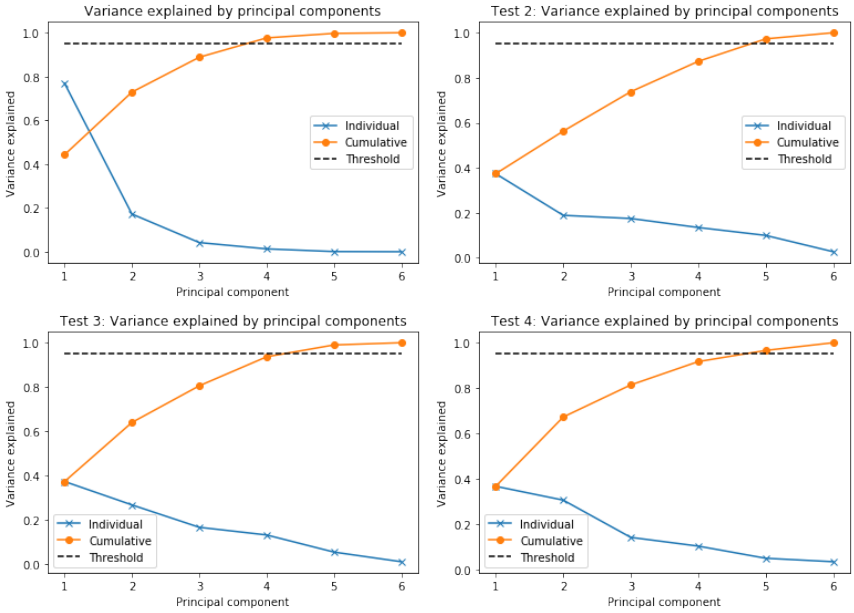


As the name suggests, in the Horizontal Displacement case we can see simple harmonic motion in the x-axis. In the case for Horizontal Displacement with Rotation, we can see much more noise, as the camera is not always readily observable.

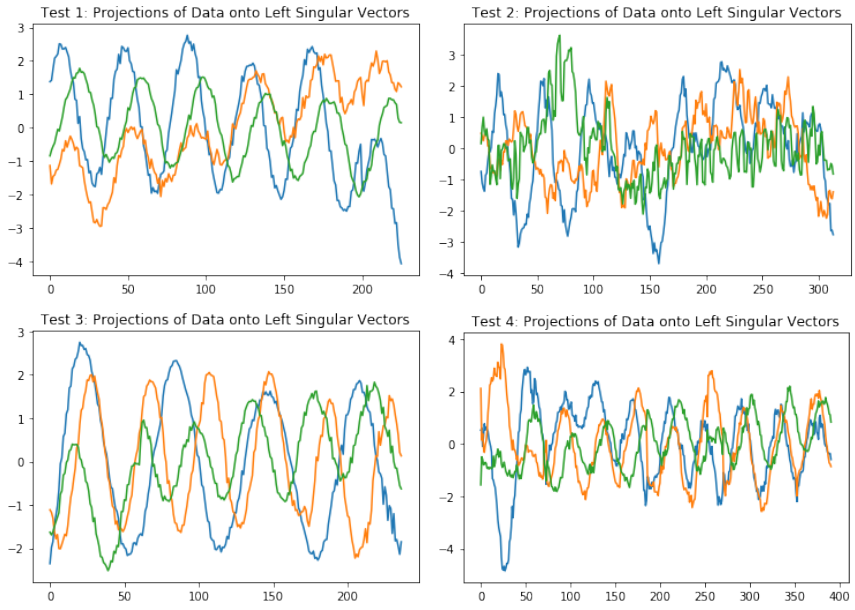
Horizontal Displacement and Horizontal Displacement with Rotation: y-axis



Explained Variance



We can clearly see, as noise is introduced into the system, it takes more principal components to explain 95% of the variation (95% was arbitrarily chose as the threshold for success).
Additionally, we can project the data onto each of the principle components. This reflects how the paint can traverses across each new basis. We can see as more noise is introduced, the variation of the third principle component becomes greater, as it explains more of the variance. Only three principle components are shown to increase clarity of the graphs; however, more components can be added.



5 Section V: Summary and Conclusions

This paper demonstrates the ability of PCA to successfully capture the dynamics of a system, despite moderate amounts of noise being introduced. In a variety of scenarios we were able to track a paint can that is moving semi-randomly, and apply PCA via singular value decomposition to reduce the dimensionality of the system. The output of the results, was aligned with our expectations and consistent with expectations from a mathematical perspective.

6 Appendix A: Python and Matlab functions used and brief implementation explanation

Matlab This was the code used to click on the flashlight and retrieve the x and y axis.

- figure(); imshow(vidFrames1(:,:,:,1); [x1, y1] = ginput(1)

Python

- rgb2gray: This example converts an image with RGB channels into an image with a single grayscale channel.
- np.unravel_index: Converts a flat index or array of flat indices into a tuple of coordinate arrays.
- np.cov: Estimate a covariance matrix, given data and weights.
- np.linalg.svd: Singular Value Decomposition.
- np.matmul: Matrix product of two arrays.

7 Appendix B: Python code

(see next page)

HW_3

March 16, 2020

1 Homework 3: PCA

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt # needed to play video
```

In order to transfer the data files from Matlab to Python, I had to write each camera's data to csv files. It's not possible to write a 3D (or greater) matrix to a csv file. In Matlab, to write the data to csv files, I have to reshape it. I reshaped each file into a (n,1) matrix. After loading the data into Python, I'll reshape the data into the proper 4D shape.

1.0.1 Test 1: Ideal Case

- Consider a small displacement of the mass in the z direction and the ensuing oscillations. In this case, the entire motion is in the z directions with simple harmonic motion being observed
- The dimensions of each file are as follows:

- vidFrames1_1: 480,640,3,226
- vidFrames2_1: 480,640,3,284
- vidFrames3_1: 480,640,3,232

Read files for the first video: - camN_1.mat where N=1,2,3

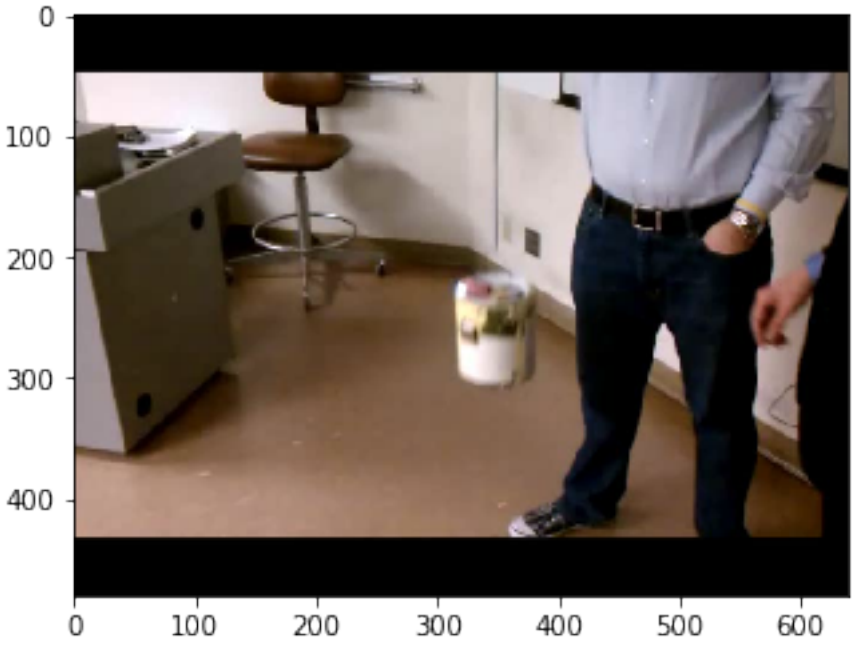
[1136]:
```
files = []
for i in range(1,4):
        files.append(pd.read_csv('cam'+ str(i) + '_1.csv',header=None))
```

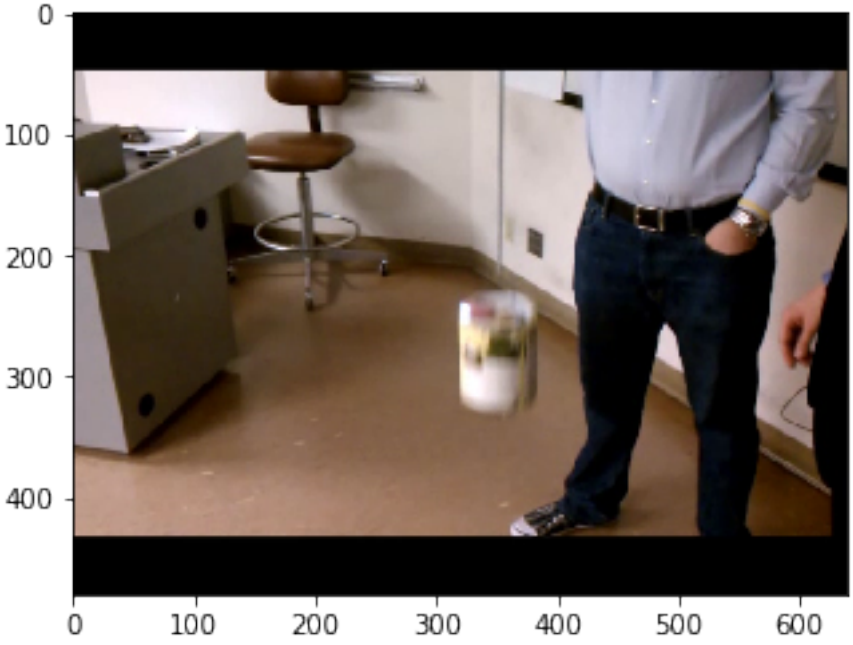
[1137]:
```
cam1_1 = files[0].to_numpy()
cam2_1 = files[1].to_numpy()
cam3_1 = files[2].to_numpy()

cam1_1 = cam1_1.reshape((480,640,3,226), order='F') # Convert to ndarray,␣
↪order='F' is to reshape using
# Fortran style which is first index first, whic is what Matlab uses
cam2_1 = cam2_1.reshape((480,640,3,284), order='F')
cam3_1 = cam3_1.reshape((480,640,3,232), order='F')
```

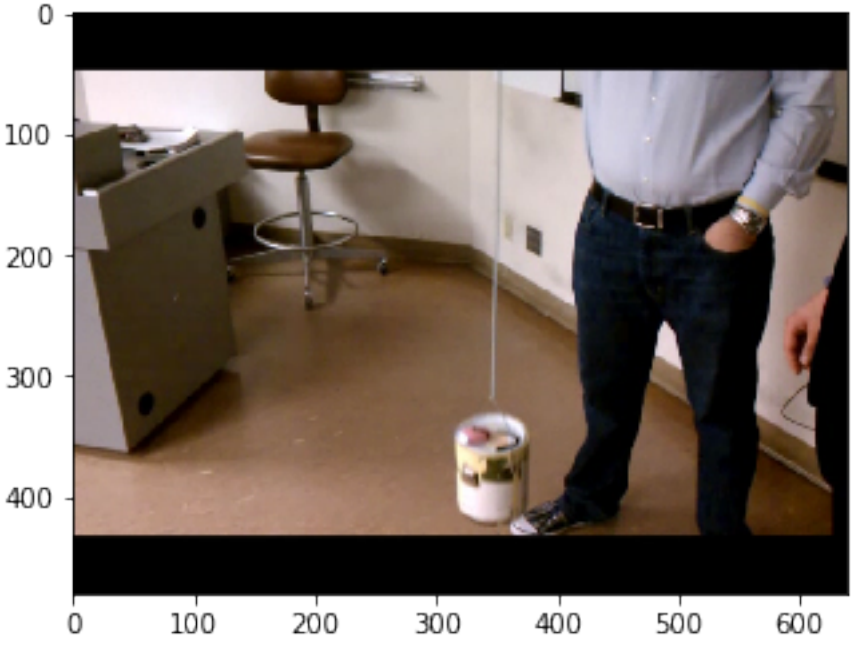
[1138]:
```
[a11,b11,c11,d11] = cam1_1.shape
[a21,b21,c21,d21] = cam2_1.shape
[a31,b31,c31,d31] = cam3_1.shape
```

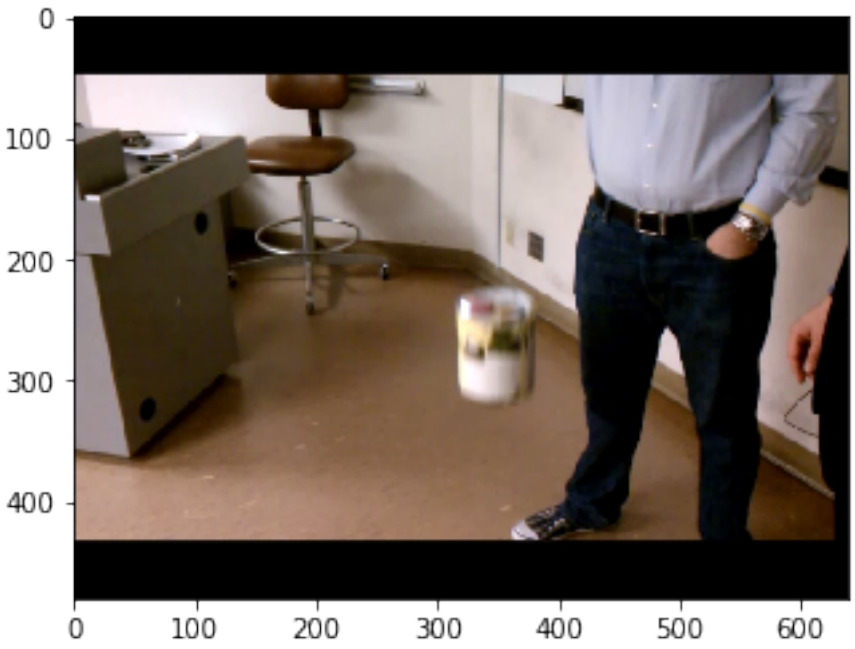
Show sequence of images to understand the trajectory of the paint can

[1139]:
```
for i in range(0,50,15):
    plt.imshow(cam1_1[:,:,:,i])
    plt.show()
```



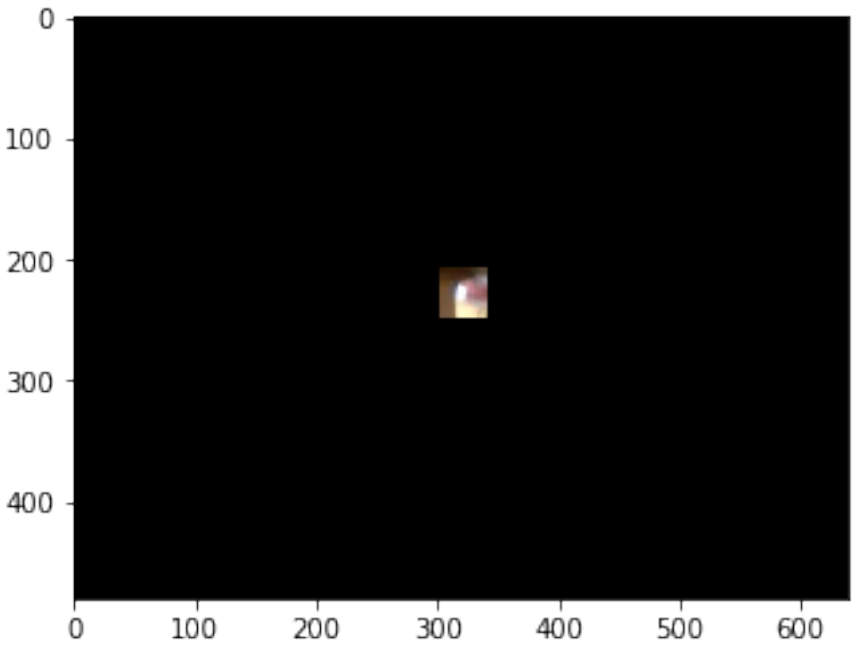

0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600



[1140]:
```
window = 20
x_axis = [322]
y_axis = [228]

test = cam1_1[:,:,:,0]

test[0:y_axis[-1]-window,:] = 0 # for some reason this changes the y axis
test[window+y_axis[-1]:,:] = 0
test[:,0:x_axis[-1]-window] = 0 # and this changes the x axis
test[:,window+x_axis[-1]:] = 0

plt.imshow(test)
```

[1140]: <matplotlib.image.AxesImage at 0x130752250>



From the pictures above we can see the paint can stays between the values of 300 and 400 on the x-axis. In order to save memory and to ensure only the flash light is captured, we'll assign all other pixels a value of 0. Additionally, we'll use a window to further reduce the search area (i.e the area from which the max value will be pulled). We'll determine the coordinates of the flashlight on the first image and then only search an area that's slightly above/below that window to pull the max value from.

Test 1: First Camera

To acquire the mouse coordinates I used ginput in Matlab. Matlab's seems to have better built-in functionality for this task. I used the following code in Matlab:

imshow(vidFrames1_1(:,:,:,1)))
[x1, y1] = ginput(1);

The mouse coordinates of the first image are [322,228]

[1141]:
```
from skimage.color import rgb2gray

x_axis1_1 = [322]
y_axis1_1 = [228]
window = 20

for i in range(1,d11):
    img = rgb2gray(cam1_1[:,:,:,i])
    img[0:y_axis1_1[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis1_1[-1]:,:] = 0
    img[:,0:x_axis1_1[-1]-window] = 0 # and this changes the x axis
```

```
    img[:,window+x_axis1_1[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis1_1.append(y_axis)
    x_axis1_1.append(x_axis)
    #print(y_axis2,x_axis2)
```

```
/opt/anaconda3/lib/python3.7/site-packages/skimage/util/dtype.py:135:
UserWarning: Possible precision loss when converting from int64 to float64
  .format(dtypeobj_in, dtypeobj_out))
```

Test 1: Second Camera

[1142]:
```
x_axis2_1 = [278]
y_axis2_1 = [272]
window = 20

for i in range(1,d21):
    img = rgb2gray(cam2_1[:,:,:,i])
    img[0:y_axis2_1[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis2_1[-1]:,:] = 0
    img[:,0:x_axis2_1[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis2_1[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis2_1.append(y_axis)
    x_axis2_1.append(x_axis)
    #print(y_axis2,x_axis2)
```

Test 1: Third Camera

[1143]:
```
x_axis3_1 = [320]
y_axis3_1 = [270]
window = 20

for i in range(1,d31):
    img = rgb2gray(cam3_1[:,:,:,i])
    img[0:y_axis3_1[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis3_1[-1]:,:] = 0
    img[:,0:x_axis3_1[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis3_1[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis3_1.append(y_axis)
    x_axis3_1.append(x_axis)
    #print(y_axis2,x_axis2)
```

To conserve memory we'll delete the three numpy ndarrays

[1144]:
```
del cam1_1, cam2_1, cam3_1
```

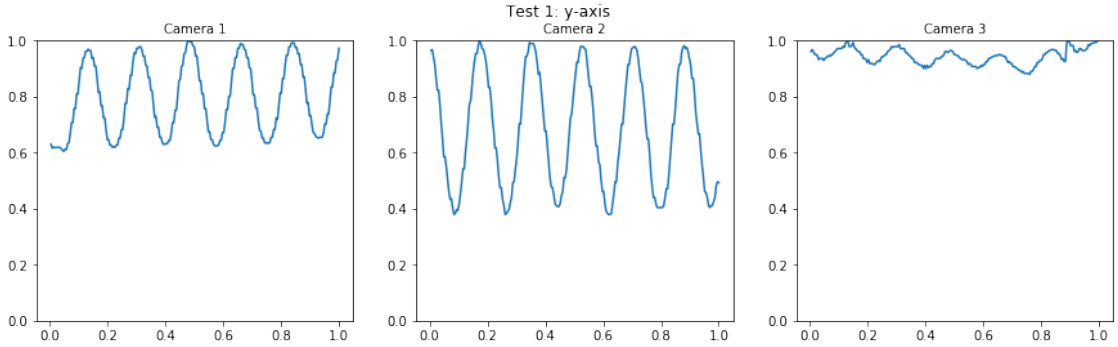
Ensure all the data has the same length

[1145]:
```
test1_ymin = min(len(y_axis1_1),len(y_axis2_1),len(y_axis3_1))
#test1_xmin = min(len(x_axis1_1),len(x_axis2_1),len(x_axis3_1))


y_axis1_1 = y_axis1_1[0:test1_ymin]
y_axis2_1 = y_axis2_1[0:test1_ymin]
y_axis3_1 = y_axis3_1[0:test1_ymin]

x_axis1_1 = x_axis1_1[0:test1_ymin] # since x-axis and y-axis have the same␣
 ↪length it doesn't matter which is used
x_axis2_1 = x_axis2_1[0:test1_ymin]
x_axis3_1 = x_axis3_1[0:test1_ymin]
```

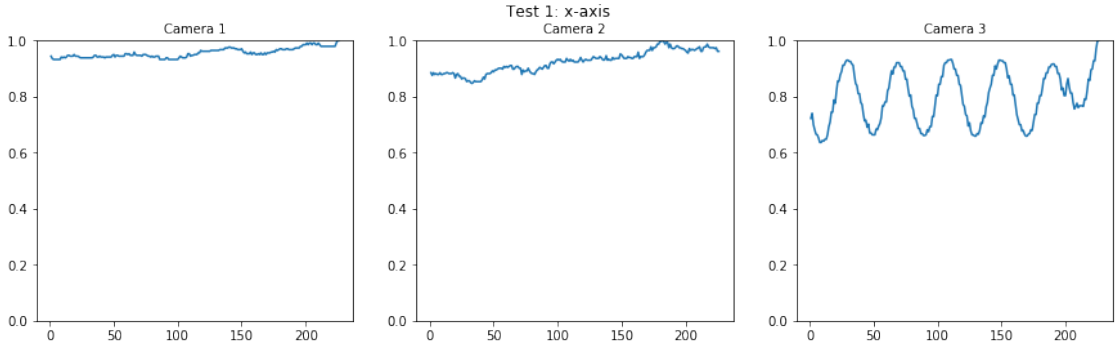
Normalize the data for the x and y axis

[1146]:
```
# Normalize y coordinates for the flashlight
yaxis1_1_norm = y_axis1_1 / max(y_axis1_1)
yaxis2_1_norm = y_axis2_1 / max(y_axis2_1)
yaxis3_1_norm = y_axis3_1 / max(y_axis3_1)

# Normalize x coordinates for the flashlight
xaxis1_1_norm = x_axis1_1 / max(x_axis1_1)
xaxis2_1_norm = x_axis2_1 / max(x_axis2_1)
xaxis3_1_norm = x_axis3_1 / max(x_axis3_1)
```

[1147]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(np.linspace(1,test1_ymin,test1_ymin) /test1_ymin,yaxis1_1_norm)
ax2.plot(np.linspace(1,test1_ymin,test1_ymin) /test1_ymin,yaxis2_1_norm)
ax3.plot(np.linspace(1,test1_ymin,test1_ymin) /test1_ymin,yaxis3_1_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 1: y-axis')

plt.show()
```



[1167]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(np.linspace(1,test1_ymin,test1_ymin) ,xaxis1_1_norm)
ax2.plot(np.linspace(1,test1_ymin,test1_ymin) ,xaxis2_1_norm)
ax3.plot(np.linspace(1,test1_ymin,test1_ymin) ,xaxis3_1_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 1: x-axis')

plt.show()
```



Apply PCA

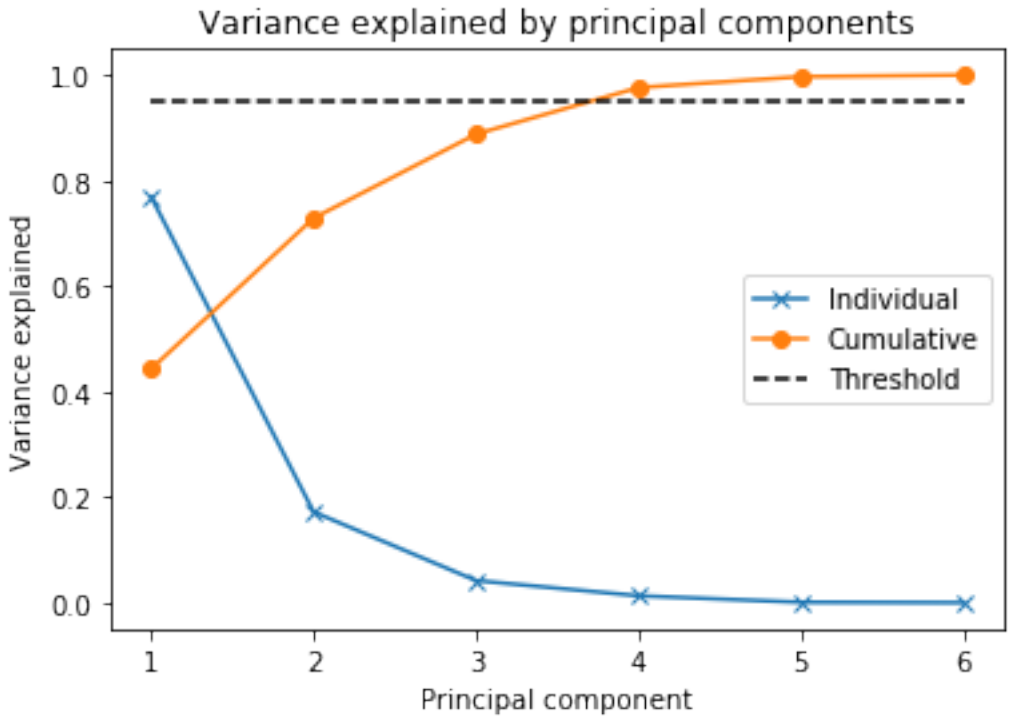
[1149]:
```
from numpy import linalg as LA

# Create 2 dataframes: one for x coordinates and one for y coordinates
test1_x = pd.DataFrame({'Camera 1':x_axis1_1, 'Camera 2': x_axis2_1,'Camera 3':␣
 ↪x_axis3_1}).T
test1_y = pd.DataFrame({'Camera 1':y_axis1_1, 'Camera 2': y_axis2_1,'Camera 3':␣
 ↪y_axis3_1}).T

# Subtract mean from each row
test1_x.iloc[0,:] = test1_x.iloc[0,:] - test1_x.iloc[0,:].mean()
test1_x.iloc[1,:] = test1_x.iloc[1,:] - test1_x.iloc[1,:].mean()
test1_x.iloc[2,:] = test1_x.iloc[2,:] - test1_x.iloc[2,:].mean()

test1_y.iloc[0,:] = test1_y.iloc[0,:] - test1_y.iloc[0,:].mean()
test1_y.iloc[1,:] = test1_y.iloc[1,:] - test1_y.iloc[1,:].mean()
test1_y.iloc[2,:] = test1_y.iloc[2,:] - test1_y.iloc[2,:].mean()


# Divide by standard deviation
test1_x.iloc[0,:] = test1_x.iloc[0,:] / np.std(test1_x.iloc[0,:])
test1_x.iloc[1,:] = test1_x.iloc[1,:] / np.std(test1_x.iloc[1,:])
test1_x.iloc[2,:] = test1_x.iloc[2,:] / np.std(test1_x.iloc[2,:])

test1_y.iloc[0,:] = test1_y.iloc[0,:] / np.std(test1_y.iloc[0,:])
test1_y.iloc[1,:] = test1_y.iloc[1,:] / np.std(test1_y.iloc[1,:])
test1_y.iloc[2,:] = test1_y.iloc[2,:] / np.std(test1_y.iloc[2,:])

# Append three rows of x values to three rows of y values
test1 = test1_x.append(test1_y)

# calculate covariance matrix
#covar = (1 / (len(test1.columns)) - 1) * np.cov(test1, rowvar=True)
covar = np.cov(test1, rowvar=True)

# Ensure covariance matrix is a square
print(covar.shape, 'notice covar is a square matrix')
```

```
(6, 6) notice covar is a square matrix
```

[1150]:
```
# Factor covariance matrix using SVD
[U,S,V] = np.linalg.svd(test1 / np.sqrt(226-1), full_matrices=False)
```

[1151]:
```
from scipy.linalg import svd

rho1 = (S*S) / (S*S).sum()

threshold = 0.95
```

```
plt.figure()
plt.plot(range(1,len(rho1)+1),rho,'x-')
plt.plot(range(1,len(rho1)+1),np.cumsum(rho1),'o-')
plt.plot([1,len(rho1)],[threshold, threshold],'k--')
plt.title('Variance explained by principal components');
plt.xlabel('Principal component');
plt.ylabel('Variance explained');
plt.legend(['Individual','Cumulative','Threshold'])
#plt.grid()
plt.show()
```



[1152]:
```
rho1
```

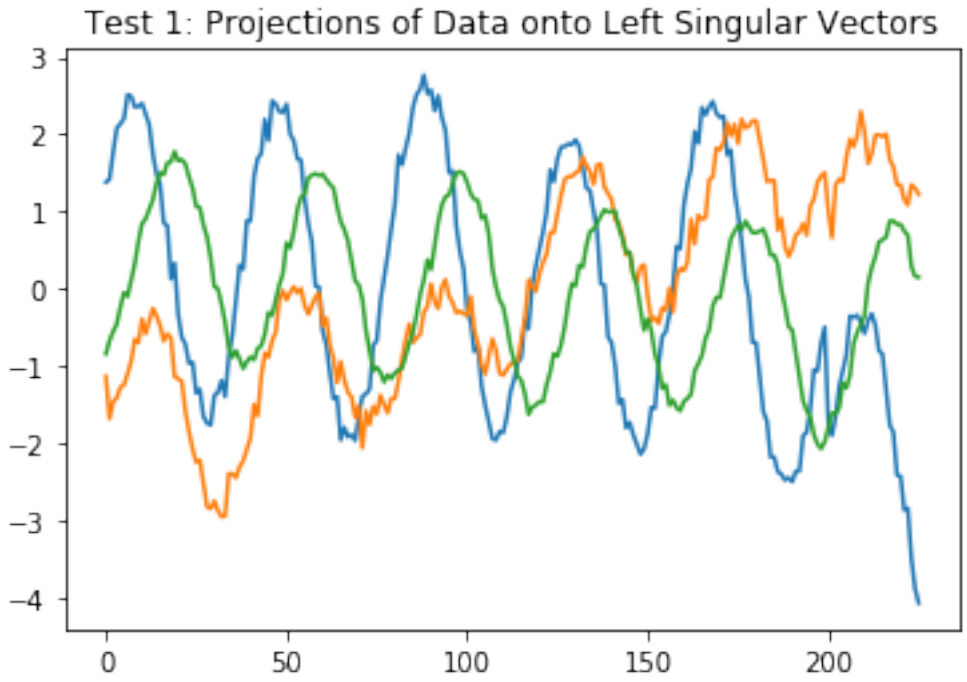
[1152]:
```
array([0.44355789, 0.28545   , 0.15911472, 0.08819275, 0.02061121,
       0.00307343])
```

Plot Projections of Data onto Left Singular Vectors

[1157]:
```
modes = np.(U.T,test1)
modes.iloc[0,:]

plt.plot(range(0,226), modes.iloc[0,:])
plt.plot(range(0,226), modes.iloc[1,:])
```

```
plt.plot(range(0,226), modes.iloc[2,:])

plt.title('Test 1: Projections of Data onto Left Singular Vectors')
```

[1157]: Text(0.5, 1.0, 'Test 1: Projections of Data onto Left Singular Vectors')



Confirm the values from the manual calculation are close to those from sklearn

[1158]:
```
from sklearn import decomposition
from sklearn import datasets

from sklearn.decomposition import PCA
pca = PCA(n_components=6, svd_solver='full')
principalComponents = pca.fit_transform(test1.T)
#pca.explained_variance_ratio_

# Check if components match using explained_variance_ from sklear
print(pca.explained_variance_[0] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪first component matches')
print(pca.explained_variance_[1] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪second component matches')
```

```
0.44355789030847725 sklearn: the first component matches
0.28544999604535765 sklearn: the second component matches
```

[1155]:
```
# Check if components match using eigenvalue decomposition
eig_val, eig_vec = np.linalg.eig(covar)
print(eig_val[0] / sum(eig_val), 'Evalue decomp: the first component matches')
print(eig_val[1] / sum(eig_val), 'Evalue decomp: the second component matches')
```

```
0.4435578903084776 Evalue decomp: the first component matches
0.2854499960453574 Evalue decomp: the second component matches
```

[1156]:
```
print(rho1[0], 'SVD: the first component matches')
print(rho1[1], 'SVD: the second component matches')
```

```
0.4435578903084773 SVD: the first component matches
0.2854499960453577 SVD: the second component matches
```

1.0.2 Test 2: Noisy Case

- Introduce camera shake into the video recording

[1049]:
```
files = []
for i in range(1,4):
        files.append(pd.read_csv('cam'+ str(i) + '_2.csv',header=None))
```

- The dimensions of each file are as follows:

- vidFrames1_2: 480,640,3,314

- vidFrames2_2: 480,640,3,356
- vidFrames3_2: 480,640,3,327

[1050]:
```
cam1_2 = files[0].to_numpy()
cam2_2 = files[1].to_numpy()
cam3_2 = files[2].to_numpy()

cam1_2 = cam1_2.reshape((480,640,3,314), order='F') # Convert to ndarray,␣
 ↪order='F' is to reshape using
# Fortran style which is first index first, which is what Matlab uses
cam2_2 = cam2_2.reshape((480,640,3,356), order='F')
cam3_2 = cam3_2.reshape((480,640,3,327), order='F')
```

[1051]:
```
[a12,b12,c12,d12] = cam1_2.shape
[a22,b22,c22,d22] = cam2_2.shape
[a32,b32,c32,d32] = cam3_2.shape
```

Test 2: First Camera

[1052]:
```
x_axis1_2 = [326]
y_axis1_2 = [304]
window = 30
```

```
for i in range(1,d12):
    img = rgb2gray(cam1_2[:,:,:,i])
    img[0:y_axis1_2[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis1_2[-1]:,:] = 0
    img[:,0:x_axis1_2[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis1_2[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis1_2.append(y_axis)
    x_axis1_2.append(x_axis)
    #print(y_axis2,x_axis2)
```

```
/opt/anaconda3/lib/python3.7/site-packages/skimage/util/dtype.py:135:
UserWarning: Possible precision loss when converting from int64 to float64
  .format(dtypeobj_in, dtypeobj_out))
```

Test 2: Second Camera

[1053]:
```
x_axis2_2 = [326]
y_axis2_2 = [304]
window = 30

for i in range(1,d22):
    img = rgb2gray(cam2_2[:,:,:,i])
    img[0:y_axis2_2[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis2_2[-1]:,:] = 0
    img[:,0:x_axis2_2[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis2_2[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis2_2.append(y_axis)
    x_axis2_2.append(x_axis)
    #print(y_axis2,x_axis2)
```

Test 2: Third Camera

[1054]:
```
x_axis3_2 = [348]
y_axis3_2 = [246]
window = 20

for i in range(1,d32):
    img = rgb2gray(cam3_2[:,:,:,i])
    img[0:y_axis3_2[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis3_2[-1]:,:] = 0
    img[:,0:x_axis3_2[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis3_2[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)
```

```
    y_axis3_2.append(y_axis)
    x_axis3_2.append(x_axis)
    #print(y_axis2,x_axis2)
```

To conserve memory we'll delete the three numpy ndarrays

[777]:
```
del cam1_2, cam2_2, cam3_2
```

Ensure all the data has the same length

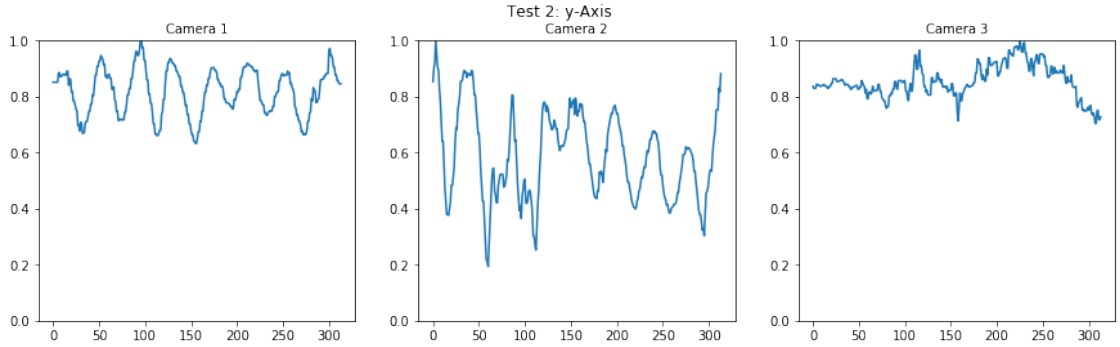
[1055]:
```
test2_min = min(len(y_axis1_2),len(y_axis2_2),len(y_axis3_2))
#test1_xmin = min(len(x_axis1_1),len(x_axis2_1),len(x_axis3_1))

y_axis1_2 = y_axis1_2[0:test2_min]
y_axis2_2 = y_axis2_2[0:test2_min]
y_axis3_2 = y_axis3_2[0:test2_min]

x_axis1_2 = x_axis1_2[0:test2_min] # since x-axis and y-axis have the same␣
 ↪length it doesn't matter which is used
x_axis2_2 = x_axis2_2[0:test2_min]
x_axis3_2 = x_axis3_2[0:test2_min]
```

Normalize the data

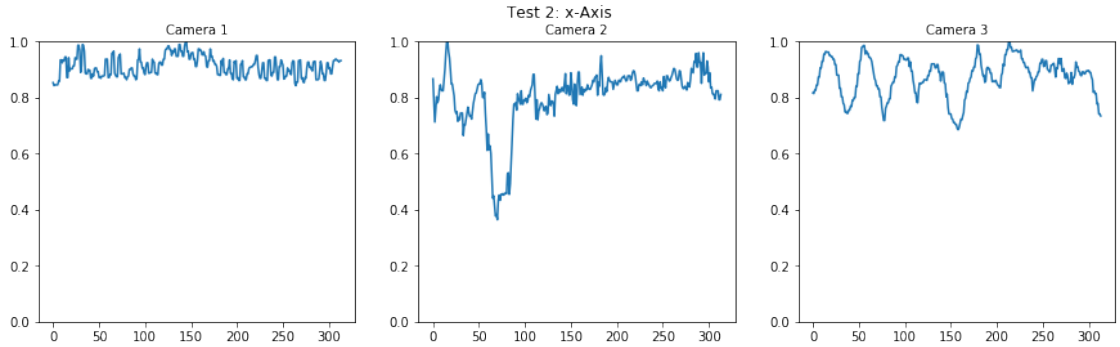
[1010]:
```
yaxis1_2_norm = y_axis1_2 / max(y_axis1_2)
yaxis2_2_norm = y_axis2_2 / max(y_axis2_2)
yaxis3_2_norm = y_axis3_2 / max(y_axis3_2)

xaxis1_2_norm = x_axis1_2 / max(x_axis1_2)
xaxis2_2_norm = x_axis2_2 / max(x_axis2_2)
xaxis3_2_norm = x_axis3_2 / max(x_axis3_2)
```

[1166]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(range(0,test2_min),yaxis1_2_norm)
ax2.plot(range(0,test2_min),yaxis2_2_norm)
ax3.plot(range(0,test2_min),yaxis3_2_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 2: y-Axis')
```

```
plt.show()
```



[1012]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(range(0,test2_min),xaxis1_2_norm)
ax2.plot(range(0,test2_min),xaxis2_2_norm)
ax3.plot(range(0,test2_min),xaxis3_2_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim((0,1))
ax2.set_ylim((0,1))
ax3.set_ylim((0,1))

fig.suptitle('Test 2: x-Axis')

plt.show()
```



Apply PCA

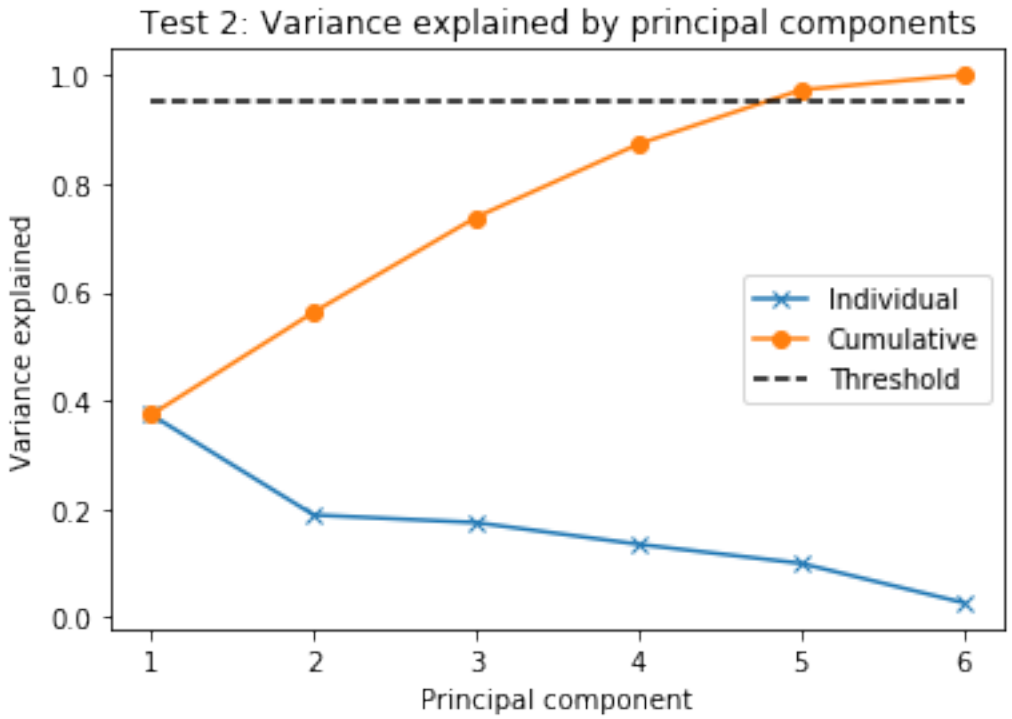
[1056]:
```
from numpy import linalg as LA

# Create 2 dataframes: one for x coordinates and one for y coordinates
test2_x = pd.DataFrame({'Camera 1':x_axis1_2, 'Camera 2': x_axis2_2,'Camera 3':␣
 ↪x_axis3_2}).T
test2_y = pd.DataFrame({'Camera 1':y_axis1_2, 'Camera 2': y_axis2_2,'Camera 3':␣
 ↪y_axis3_2}).T

# Subtract mean from each row
test2_x.iloc[0,:] = test2_x.iloc[0,:] - test2_x.iloc[0,:].mean()
test2_x.iloc[1,:] = test2_x.iloc[1,:] - test2_x.iloc[1,:].mean()
test2_x.iloc[2,:] = test2_x.iloc[2,:] - test2_x.iloc[2,:].mean()

test2_y.iloc[0,:] = test2_y.iloc[0,:] - test2_y.iloc[0,:].mean()
test2_y.iloc[1,:] = test2_y.iloc[1,:] - test2_y.iloc[1,:].mean()
test2_y.iloc[2,:] = test2_y.iloc[2,:] - test2_y.iloc[2,:].mean()

# Divide by standard deviation
test2_x.iloc[0,:] = test2_x.iloc[0,:] / np.std(test2_x.iloc[0,:])
test2_x.iloc[1,:] = test2_x.iloc[1,:] / np.std(test2_x.iloc[1,:])
test2_x.iloc[2,:] = test2_x.iloc[2,:] / np.std(test2_x.iloc[2,:])

test2_y.iloc[0,:] = test2_y.iloc[0,:] / np.std(test2_y.iloc[0,:])
test2_y.iloc[1,:] = test2_y.iloc[1,:] / np.std(test2_y.iloc[1,:])
test2_y.iloc[2,:] = test2_y.iloc[2,:] / np.std(test2_y.iloc[2,:])

# Append three rows of x values to three rows of y values
test2 = test2_x.append(test2_y)

# calculate covariance matrix
#covar = (1 / (len(test1.columns)) - 1) * np.cov(test1, rowvar=True)
covar = np.cov(test2, rowvar=True)

# Ensure covariance matrix is a square
print(covar.shape, 'notice covar is a square matrix')
```

```
(6, 6) notice covar is a square matrix
```

[1064]:
```
# Factor covariance matrix using SVD
[U,S,V] = np.linalg.svd(test2 / np.sqrt(len(test2.columns)-1),␣
 ↪full_matrices=False)
```

[1065]:
```
from scipy.linalg import svd

rho2 = (S*S) / (S*S).sum()

threshold = 0.95
```

```
plt.figure()
plt.plot(range(1,len(rho2)+1),rho2,'x-')
plt.plot(range(1,len(rho2)+1),np.cumsum(rho2),'o-')
plt.plot([1,len(rho2)],[threshold, threshold],'k--')
plt.title('Test 2: Variance explained by principal components');
plt.xlabel('Principal component');
plt.ylabel('Variance explained');
plt.legend(['Individual','Cumulative','Threshold'])
#plt.grid()
plt.show()
```



[1066]:
```
rho2
```

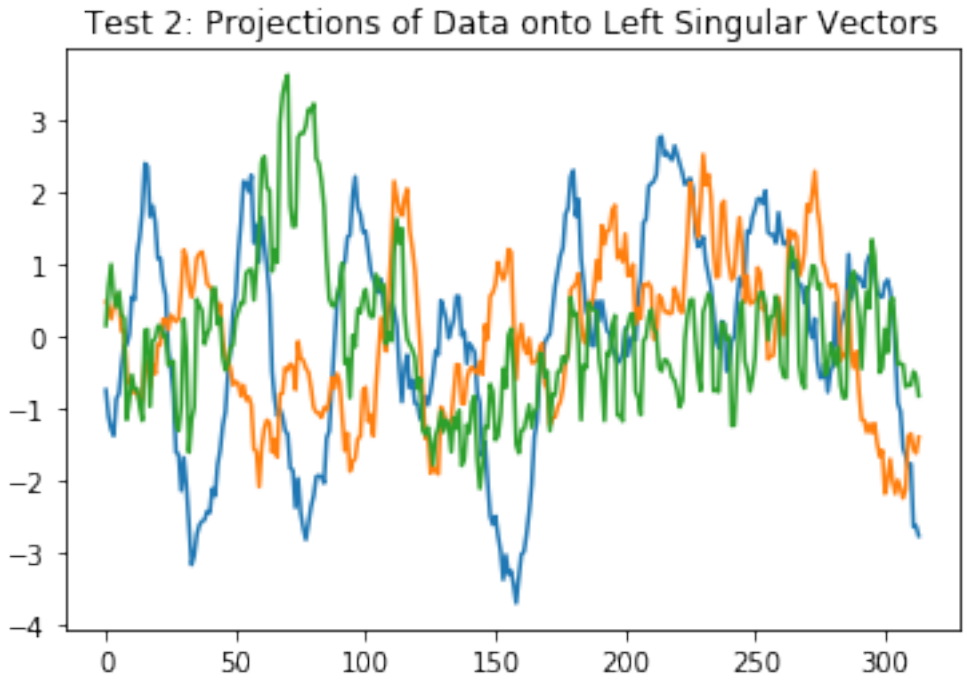
[1066]:
```
array([0.37384235, 0.18908515, 0.17495074, 0.13497455, 0.09980951,
       0.0273377 ])
```

Plot Projections of Data onto Left Singular Vectors

[1067]:
```
modes = np.matmul(U.T,test2)
modes.iloc[0,:]

plt.plot(range(0,len(test2.columns)), modes.iloc[0,:])
plt.plot(range(0,len(test2.columns)), modes.iloc[1,:])
```

```
plt.plot(range(0,len(test2.columns)), modes.iloc[2,:])

plt.title('Test 2: Projections of Data onto Left Singular Vectors')
```

[1067]: Text(0.5, 1.0, 'Test 2: Projections of Data onto Left Singular Vectors')



Confirm the values from the manual calculation are close to those from sklearn

[1070]:
```
from sklearn import decomposition
from sklearn import datasets

from sklearn.decomposition import PCA
pca = PCA(n_components=6, svd_solver='full')
principalComponents = pca.fit_transform(test2.T)
#pca.explained_variance_ratio_

# Check if components match using explained_variance_ from sklear
print(pca.explained_variance_[0] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪first component matches')
print(pca.explained_variance_[1] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪second component matches')
```

```
0.37384235460191223 sklearn: the first component matches
0.18908514649822067 sklearn: the second component matches
```

[1095]:
```
# Check if components match using eigenvalue decomposition
eig_val, eig_vec = np.linalg.eig(covar)
print(eig_val[0] / sum(eig_val), 'Evalue decomp: the first component matches')
print(eig_val[1] / sum(eig_val), 'Evalue decomp: the second no match (check␣
 ↪instability of Eig Decomp)')
```

```
0.3726356225830119 Evalue decomp: the first component matches
0.010050951888024347 Evalue decomp: the second no match (check instability of
Eig Decomp)
```

[1072]:
```
print(rho2[0], 'SVD: the first component matches')
print(rho2[1], 'SVD: the second component matches')
```

```
0.37384235460191234 SVD: the first component matches
0.18908514649822059 SVD: the second component matches
```

1.0.3 Test 3: horizontal displacement

- The dimensions of each file are as follows:

- vidFrames1_2: 480,640,3,239
- vidFrames2_2: 480,640,3,281
- vidFrames3_2: 480,640,3,237

[792]:
```
files = []
for i in range(1,4):
        files.append(pd.read_csv('cam'+ str(i) + '_3.csv',header=None))
```

[793]:
```
cam1_3 = files[0].to_numpy()
cam2_3 = files[1].to_numpy()
cam3_3 = files[2].to_numpy()

cam1_3 = cam1_3.reshape((480,640,3,239), order='F') # Convert to ndarray,␣
 ↪order='F' is to reshape using
# Fortran style which is first index first, whic is what Matlab uses
cam2_3 = cam2_3.reshape((480,640,3,281), order='F')
cam3_3 = cam3_3.reshape((480,640,3,237), order='F')
```

[794]:
```
[a13,b13,c13,d13] = cam1_3.shape
[a23,b23,c23,d23] = cam2_3.shape
[a33,b33,c33,d33] = cam3_3.shape
```

Test 3: First Camera

[795]:
```
x_axis1_3 = [326]
y_axis1_3 = [284]
window = 20
```

```
for i in range(1,d13):
    img = rgb2gray(cam1_3[:,:,:,i])
    img[0:y_axis1_3[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis1_3[-1]:,:] = 0
    img[:,0:x_axis1_3[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis1_3[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis1_3.append(y_axis)
    x_axis1_3.append(x_axis)
    #print(y_axis2,x_axis2)
```

```
/opt/anaconda3/lib/python3.7/site-packages/skimage/util/dtype.py:135:
UserWarning: Possible precision loss when converting from int64 to float64
  .format(dtypeobj_in, dtypeobj_out))
```

Test 3: Second Camera

[796]:
```
x_axis2_3 = [244]
y_axis2_3 = [292]
window = 20

for i in range(1,d23):
    img = rgb2gray(cam2_3[:,:,:,i])
    img[0:y_axis2_3[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis2_3[-1]:,:] = 0
    img[:,0:x_axis2_3[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis2_3[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis2_3.append(y_axis)
    x_axis2_3.append(x_axis)
    #print(y_axis2,x_axis2)
```

Test 3: Third Camera

[797]:
```
x_axis3_3 = [356]
y_axis3_3 = [230]
window = 20

for i in range(1,d33):
    img = rgb2gray(cam3_3[:,:,:,i])
    img[0:y_axis3_3[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis3_3[-1]:,:] = 0
    img[:,0:x_axis3_3[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis3_3[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)
```

```
    y_axis3_3.append(y_axis)
    x_axis3_3.append(x_axis)
    #print(y_axis2,x_axis2)
```

[798]:
```
del cam1_3, cam2_3, cam3_3
```

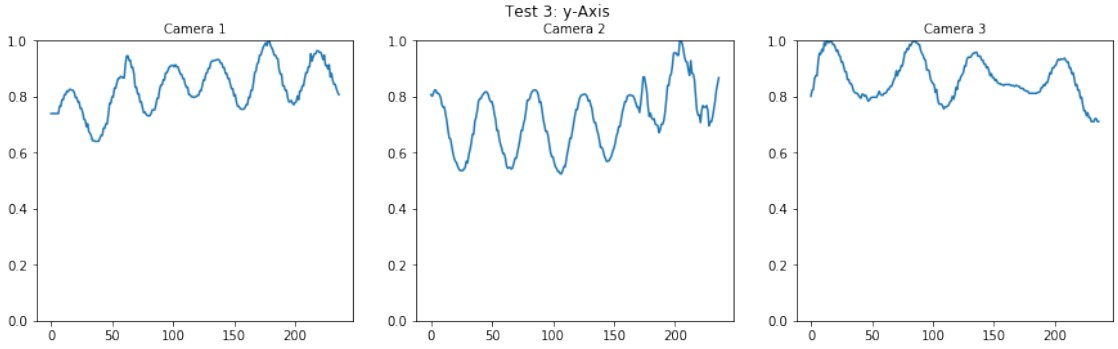
Ensure all the data has the same length

[799]:
```
test3_min = min(len(y_axis1_3),len(y_axis2_3),len(y_axis3_3))
#test1_xmin = min(len(x_axis1_1),len(x_axis2_1),len(x_axis3_1))

y_axis1_3 = y_axis1_3[0:test3_min]
y_axis2_3 = y_axis2_3[0:test3_min]
y_axis3_3 = y_axis3_3[0:test3_min]

x_axis1_3 = x_axis1_3[0:test3_min] # since x-axis and y-axis have the same␣
↪length it doesn't matter which is used
x_axis2_3 = x_axis2_3[0:test3_min]
x_axis3_3 = x_axis3_3[0:test3_min]
```

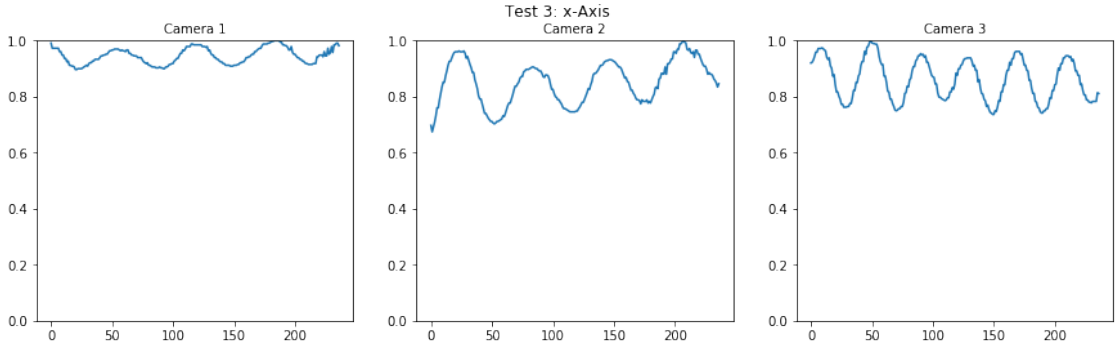
Normalize the data

[1077]:
```
yaxis1_3_norm = y_axis1_3 / max(y_axis1_3)
yaxis2_3_norm = y_axis2_3 / max(y_axis2_3)
yaxis3_3_norm = y_axis3_3 / max(y_axis3_3)

xaxis1_3_norm = x_axis1_3 / max(x_axis1_3)
xaxis2_3_norm = x_axis2_3 / max(x_axis2_3)
xaxis3_3_norm = x_axis3_3 / max(x_axis3_3)
```

[1078]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(range(0,test3_min),yaxis1_3_norm)
ax2.plot(range(0,test3_min),yaxis2_3_norm)
ax3.plot(range(0,test3_min),yaxis3_3_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 3: y-Axis')

plt.show()
```



[1079]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(range(0,test3_min),xaxis1_3_norm)
ax2.plot(range(0,test3_min),xaxis2_3_norm)
ax3.plot(range(0,test3_min),xaxis3_3_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 3: x-Axis')

plt.show()
```



Apply PCA

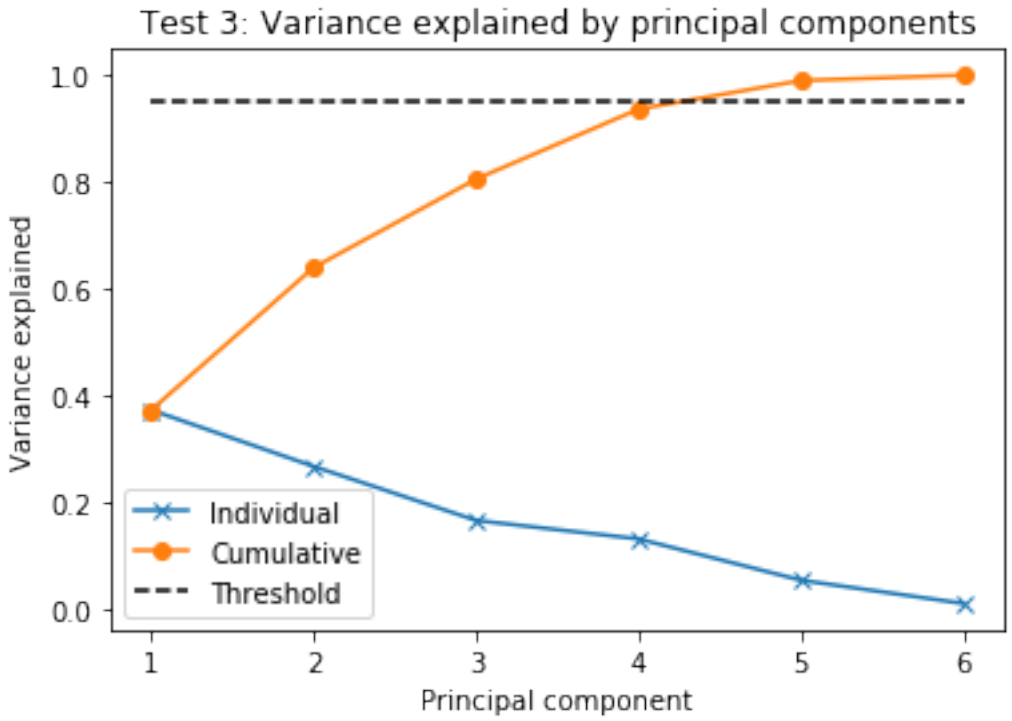
[1080]:
```
from numpy import linalg as LA

# Create 2 dataframes: one for x coordinates and one for y coordinates
test3_x = pd.DataFrame({'Camera 1':x_axis1_3, 'Camera 2': x_axis2_3,'Camera 3':␣
 ↪x_axis3_3}).T
test3_y = pd.DataFrame({'Camera 1':y_axis1_3, 'Camera 2': y_axis2_3,'Camera 3':␣
 ↪y_axis3_3}).T

# Subtract mean from each row
test3_x.iloc[0,:] = test3_x.iloc[0,:] - test3_x.iloc[0,:].mean()
test3_x.iloc[1,:] = test3_x.iloc[1,:] - test3_x.iloc[1,:].mean()
test3_x.iloc[2,:] = test3_x.iloc[2,:] - test3_x.iloc[2,:].mean()

test3_y.iloc[0,:] = test3_y.iloc[0,:] - test3_y.iloc[0,:].mean()
test3_y.iloc[1,:] = test3_y.iloc[1,:] - test3_y.iloc[1,:].mean()
test3_y.iloc[2,:] = test3_y.iloc[2,:] - test3_y.iloc[2,:].mean()

# Divide by standard deviation
test3_x.iloc[0,:] = test3_x.iloc[0,:] / np.std(test3_x.iloc[0,:])
test3_x.iloc[1,:] = test3_x.iloc[1,:] / np.std(test3_x.iloc[1,:])
test3_x.iloc[2,:] = test3_x.iloc[2,:] / np.std(test3_x.iloc[2,:])

test3_y.iloc[0,:] = test3_y.iloc[0,:] / np.std(test3_y.iloc[0,:])
test3_y.iloc[1,:] = test3_y.iloc[1,:] / np.std(test3_y.iloc[1,:])
test3_y.iloc[2,:] = test3_y.iloc[2,:] / np.std(test3_y.iloc[2,:])


# Append three rows of x values to three rows of y values
test3 = test3_x.append(test3_y)

# calculate covariance matrix
covar = (1 / (len(test3.columns)) - 1) * np.cov(test3, rowvar=True)

# Ensure covariance matrix is a square
print(covar.shape, 'notice covar is a square matrix')
```

```
(6, 6) notice covar is a square matrix
```

[1087]:
```
# Factor covariance matrix using SVD
[U,S,V] = np.linalg.svd(test3 / np.sqrt(len(test3.
 ↪columns)-1),full_matrices=False)

lambda_ = np.diag(S) **2
```

[1088]:
```
from scipy.linalg import svd

rho3 = (S*S) / (S*S).sum()
```

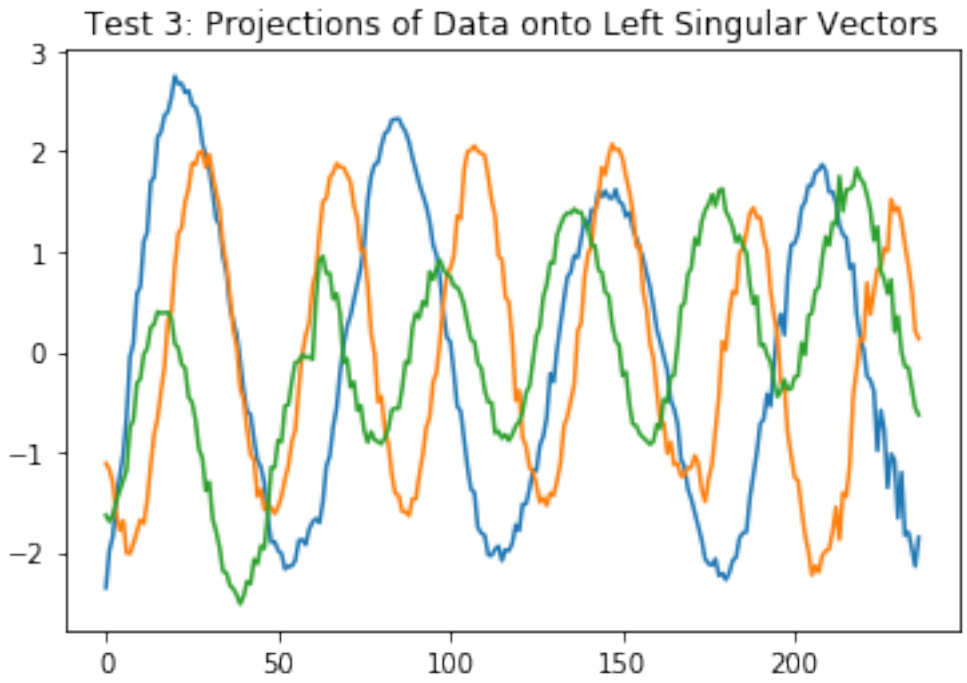
```
threshold = 0.95

plt.figure()
plt.plot(range(1,len(rho3)+1),rho3,'x-')
plt.plot(range(1,len(rho3)+1),np.cumsum(rho3),'o-')
plt.plot([1,len(rho3)],[threshold, threshold],'k--')
plt.title('Test 3: Variance explained by principal components');
plt.xlabel('Principal component');
plt.ylabel('Variance explained');
plt.legend(['Individual','Cumulative','Threshold'])
#plt.grid()
plt.show()
```



Plot Projections of Data onto Left Singular Vectors

[1089]:
```
modes = np.matmul(U.T,test3)
modes.iloc[0,:]

plt.plot(range(0,len(test3.columns)), modes.iloc[0,:])
plt.plot(range(0,len(test3.columns)), modes.iloc[1,:])
plt.plot(range(0,len(test3.columns)), modes.iloc[2,:])

plt.title('Test 3: Projections of Data onto Left Singular Vectors')
```

[1089]: Text(0.5, 1.0, 'Test 3: Projections of Data onto Left Singular Vectors')



Confirm the values from the manual calculation are close to those from sklearn

[1090]:
```
from sklearn import decomposition
from sklearn import datasets

from sklearn.decomposition import PCA
pca = PCA(n_components=5, svd_solver='full')
principalComponents = pca.fit_transform(test3.T)
#pca.explained_variance_ratio_

# Check if components match using explained_variance_ from sklear
print(pca.explained_variance_[0] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪first component matches')
print(pca.explained_variance_[1] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪second component matches')
```

```
0.3764189917589197 sklearn: the first component matches
0.26963623605531983 sklearn: the second component matches
```

[1096]:
```
# Check if components match using eigenvalue decomposition
eig_val, eig_vec = np.linalg.eig(covar)
print(eig_val[0] / sum(eig_val), 'Evalue decomp: the first component matches')
```

```
print(eig_val[1] / sum(eig_val), 'Evalue decomp: no match (check instability of␣
↪Eig Decomp)')
```

```
0.3726356225830119 Evalue decomp: the first component matches
0.010050951888024347 Evalue decomp: no match (check instability of Eig Decomp)
```

[1092]:
```
print(rho3[0], 'SVD: the first component matches')
print(rho3[1], 'SVD: the second component matches')
```

```
0.3726356225830115 SVD: the first component matches
0.26692613521946046 SVD: the second component matches
```

1.0.4 Test 4: horizontal displacement and rotation

- The dimensions of each file are as follows:

- vidFrames1_2: 480,640,3,392
- vidFrames2_2: 480,640,3,405
- vidFrames3_2: 480,640,3,394

[810]:
```
files = []
for i in range(1,4):
        files.append(pd.read_csv('cam'+ str(i) + '_4.csv',header=None))
```

[811]:
```
cam1_4 = files[0].to_numpy()
cam2_4 = files[1].to_numpy()
cam3_4 = files[2].to_numpy()

cam1_4 = cam1_4.reshape((480,640,3,392), order='F') # Convert to ndarray,␣
↪order='F' is to reshape using
# Fortran style which is first index first, whic is what Matlab uses
cam2_4 = cam2_4.reshape((480,640,3,405), order='F')
cam3_4 = cam3_4.reshape((480,640,3,394), order='F')
```

[812]:
```
[a14,b14,c14,d14] = cam1_4.shape
[a24,b24,c24,d24] = cam2_4.shape
[a34,b34,c34,d34] = cam3_4.shape
```

Test 4: First Camera

[813]:
```
x_axis1_4 = [412]
y_axis1_4 = [268]
window = 40

for i in range(1,d14):
    img = rgb2gray(cam1_4[:,:,:,i])
    img[0:y_axis1_4[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis1_4[-1]:,:] = 0
```

```
    img[:,0:x_axis1_4[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis1_4[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis1_4.append(y_axis)
    x_axis1_4.append(x_axis)
    #print(y_axis2,x_axis2)
```

```
/opt/anaconda3/lib/python3.7/site-packages/skimage/util/dtype.py:135:
UserWarning: Possible precision loss when converting from int64 to float64
  .format(dtypeobj_in, dtypeobj_out))
```

Test 4: Second Camera

[814]:
```
x_axis2_4 = [278]
y_axis2_4 = [248]
window = 20

for i in range(1,d24):
    img = rgb2gray(cam2_4[:,:,:,i])
    img[0:y_axis2_4[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis2_4[-1]:,:] = 0
    img[:,0:x_axis2_4[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis2_4[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis2_4.append(y_axis)
    x_axis2_4.append(x_axis)
    #print(y_axis2,x_axis2)
```

Test 4: Third Camera

[815]:
```
x_axis3_4 = [368]
y_axis3_4 = [230]
window = 20

for i in range(1,d34):
    img = rgb2gray(cam3_4[:,:,:,i])
    img[0:y_axis3_4[-1]-window,:] = 0 # for some reason this changes the y axis
    img[window+y_axis3_4[-1]:,:] = 0
    img[:,0:x_axis3_4[-1]-window] = 0 # and this changes the x axis
    img[:,window+x_axis3_4[-1]:] = 0    #img[:,227+window:] = 0
    [y_axis,x_axis] = np.unravel_index(np.argmax(img, axis=None), img.shape)

    y_axis3_4.append(y_axis)
    x_axis3_4.append(x_axis)
```

[816]:
```
del cam1_4, cam2_4, cam3_4
```

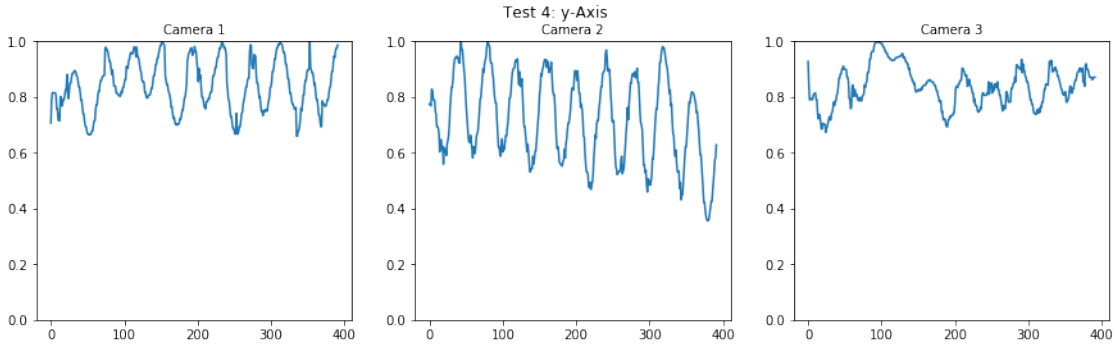
Ensure all the data has the same length

[817]:
```
test4_min = min(len(y_axis1_4),len(y_axis2_4),len(y_axis3_4))
#test1_xmin = min(len(x_axis1_1),len(x_axis2_1),len(x_axis3_1))


y_axis1_4 = y_axis1_4[0:test4_min]
y_axis2_4 = y_axis2_4[0:test4_min]
y_axis3_4 = y_axis3_4[0:test4_min]

x_axis1_4 = x_axis1_4[0:test4_min] # since x-axis and y-axis have the same␣
↪length it doesn't matter which is used
x_axis2_4 = x_axis2_4[0:test4_min]
x_axis3_4 = x_axis3_4[0:test4_min]
```

Normalize the data

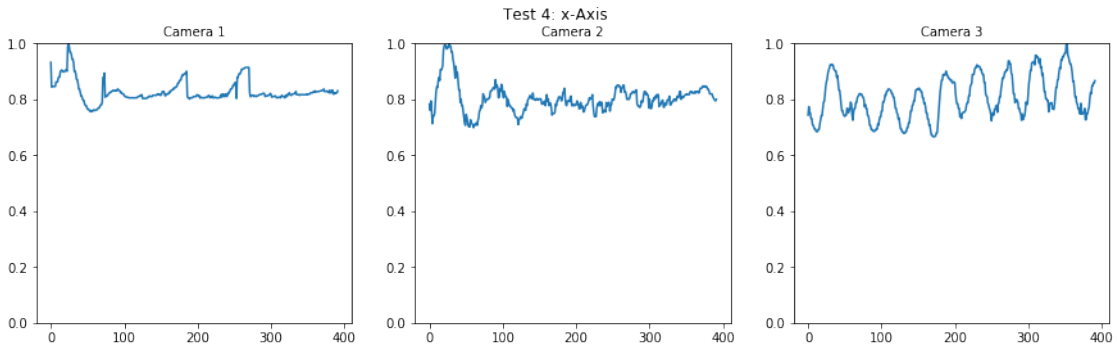
[1097]:
```
yaxis1_4_norm = y_axis1_4 / max(y_axis1_4)
yaxis2_4_norm = y_axis2_4 / max(y_axis2_4)
yaxis3_4_norm = y_axis3_4 / max(y_axis3_4)

xaxis1_4_norm = x_axis1_4 / max(x_axis1_4)
xaxis2_4_norm = x_axis2_4 / max(x_axis2_4)
xaxis3_4_norm = x_axis3_4 / max(x_axis3_4)
```

[1098]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(range(0,test4_min),yaxis1_4_norm)
ax2.plot(range(0,test4_min),yaxis2_4_norm)
ax3.plot(range(0,test4_min),yaxis3_4_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 4: y-Axis')

plt.show()
```



[1118]:
```
fig,(ax1, ax2, ax3) =  plt.subplots(1,3,figsize=(15,4))

ax1.plot(range(0,test4_min),xaxis1_4_norm)
ax2.plot(range(0,test4_min),xaxis2_4_norm)
ax3.plot(range(0,test4_min),xaxis3_4_norm)

ax1.set_title('Camera 1', fontsize=10)
ax2.set_title('Camera 2', fontsize=10)
ax3.set_title('Camera 3', fontsize=10)

ax1.set_ylim([0, 1])
ax2.set_ylim([0, 1])
ax3.set_ylim([0, 1])

fig.suptitle('Test 4: x-Axis')

fig.show()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:17:
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```



Apply PCA

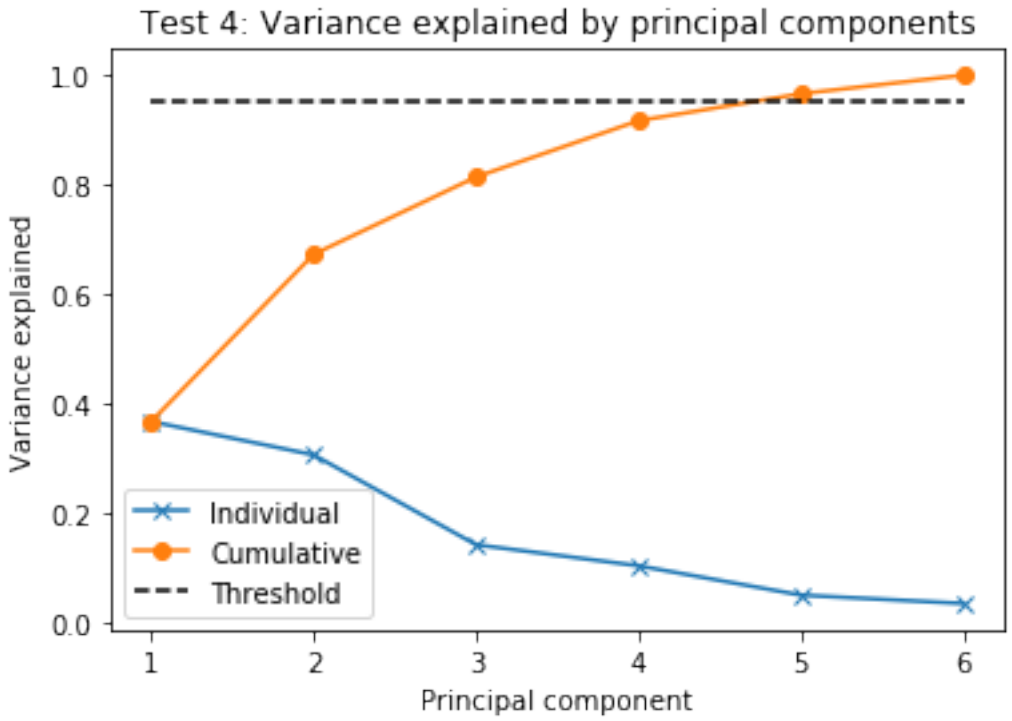
[1100]:
```
from numpy import linalg as LA

# Create 2 dataframes: one for x coordinates and one for y coordinates
test4_x = pd.DataFrame({'Camera 1':x_axis1_4, 'Camera 2': x_axis2_4,'Camera 3':␣
 ↪x_axis3_4}).T
test4_y = pd.DataFrame({'Camera 1':y_axis1_4, 'Camera 2': y_axis2_4,'Camera 3':␣
 ↪y_axis3_4}).T

# Subtract mean from each row
test4_x.iloc[0,:] = test4_x.iloc[0,:] - test4_x.iloc[0,:].mean()
test4_x.iloc[1,:] = test4_x.iloc[1,:] - test4_x.iloc[1,:].mean()
test4_x.iloc[2,:] = test4_x.iloc[2,:] - test4_x.iloc[2,:].mean()

test4_y.iloc[0,:] = test4_y.iloc[0,:] - test4_y.iloc[0,:].mean()
test4_y.iloc[1,:] = test4_y.iloc[1,:] - test4_y.iloc[1,:].mean()
test4_y.iloc[2,:] = test4_y.iloc[2,:] - test4_y.iloc[2,:].mean()

# Divide by standard deviation
test4_x.iloc[0,:] = test4_x.iloc[0,:] / np.std(test4_x.iloc[0,:])
test4_x.iloc[1,:] = test4_x.iloc[1,:] / np.std(test4_x.iloc[1,:])
test4_x.iloc[2,:] = test4_x.iloc[2,:] / np.std(test4_x.iloc[2,:])

test4_y.iloc[0,:] = test4_y.iloc[0,:] / np.std(test4_y.iloc[0,:])
test4_y.iloc[1,:] = test4_y.iloc[1,:] / np.std(test4_y.iloc[1,:])
test4_y.iloc[2,:] = test4_y.iloc[2,:] / np.std(test4_y.iloc[2,:])

# Append three rows of x values to three rows of y values
test4 = test4_x.append(test4_y)

# calculate covariance matrix
covar = (1 / (len(test4.columns)) - 1) * np.cov(test4, rowvar=True)

# Ensure covariance matrix is a square
print(covar.shape, 'notice covar is a square matrix')
```

```
(6, 6) notice covar is a square matrix
```

[1101]:
```
# Factor covariance matrix using SVD
[U,S,V] = np.linalg.svd(test4 / np.sqrt(len(test4.columns)-1))

lambda_ = np.diag(S) **2
```

[1102]:
```
from scipy.linalg import svd
```

```
rho4 = (S*S) / (S*S).sum()

threshold = 0.95

plt.figure()
plt.plot(range(1,len(rho4)+1),rho4,'x-')
plt.plot(range(1,len(rho4)+1),np.cumsum(rho4),'o-')
plt.plot([1,len(rho4)],[threshold, threshold],'k--')
plt.title('Test 4: Variance explained by principal components');
plt.xlabel('Principal component');
plt.ylabel('Variance explained');
plt.legend(['Individual','Cumulative','Threshold'])
#plt.grid()
plt.show()
```

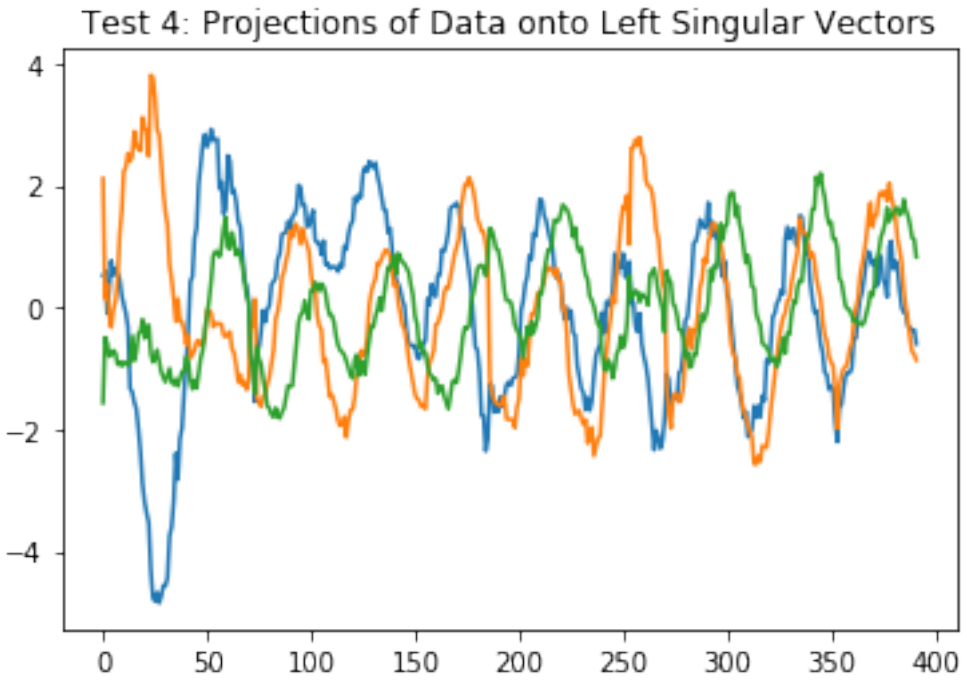


Plot Projections of Data onto Left Singular Vectors

[1104]:
```
modes = np.matmul(U.T,test4)
modes.iloc[0,:]

plt.plot(range(0,len(test4.columns)), modes.iloc[0,:])
plt.plot(range(0,len(test4.columns)), modes.iloc[1,:])
plt.plot(range(0,len(test4.columns)), modes.iloc[2,:])
```

```
plt.title('Test 4: Projections of Data onto Left Singular Vectors')
```

[1104]:
```
Text(0.5, 1.0, 'Test 4: Projections of Data onto Left Singular Vectors')
```



Confirm the values from the manual calculation are close to those from sklearn

[1105]:
```
from sklearn import decomposition
from sklearn import datasets

from sklearn.decomposition import PCA
pca = PCA(n_components=6, svd_solver='full')
principalComponents = pca.fit_transform(test4.T)
#pca.explained_variance_ratio_

# Check if components match using explained_variance_ from sklear
print(pca.explained_variance_[0] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪first component matches')
print(pca.explained_variance_[1] / sum(pca.explained_variance_), 'sklearn: the␣
 ↪second component matches')
```

```
0.36671560999751246 sklearn: the first component matches
0.30568404352508505 sklearn: the second component matches
```

[1106]:
```
# Check if components match using eigenvalue decomposition
eig_val, eig_vec = np.linalg.eig(covar)
```

```
print(eig_val[0] / sum(eig_val), 'Evalue decomp: the first component matches')
print(eig_val[1] / sum(eig_val), 'Evalue decomp: no match (check instability of␣
↪Eig Decomp)')
```

```
0.3667156099975123 Evalue decomp: the first component matches
0.3056840435250854 Evalue decomp: no match (check instability of Eig Decomp)
```

[1107]:
```
print(rho4[0], 'SVD: the first component matches')
print(rho4[1], 'SVD: the second component matches')
```

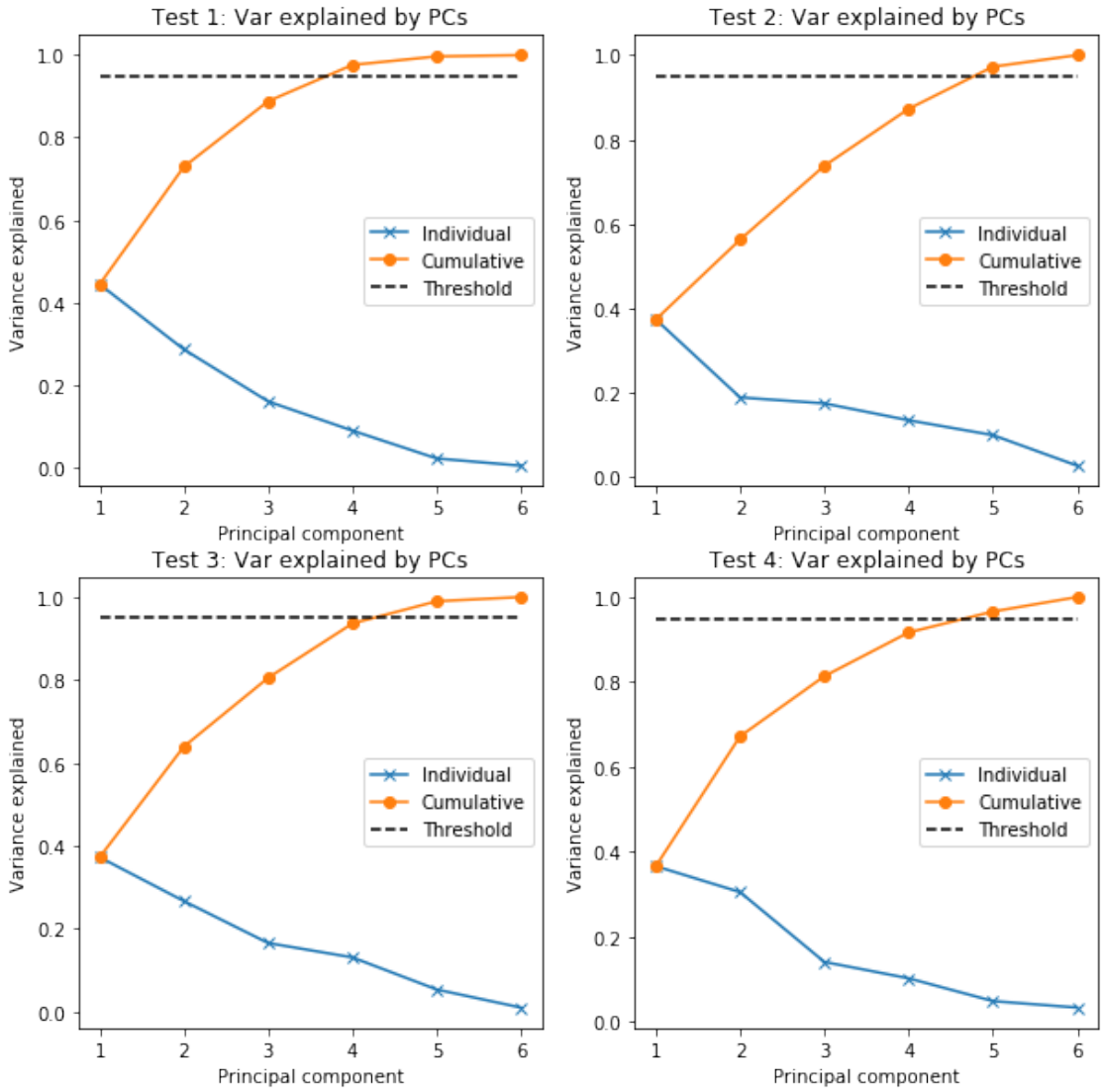
```
0.36671560999751246 SVD: the first component matches
0.30568404352508544 SVD: the second component matches
```

Recap the explained variance for each test

[1134]:
```
fig,[[ax1, ax2], [ax3, ax4]] =  plt.subplots(2,2,figsize=(10,10))

# Test 1
ax1.plot(range(1,len(rho1)+1),rho1,'x-')
ax1.plot(range(1,len(rho1)+1),np.cumsum(rho1),'o-')
ax1.plot([1,len(rho1)],[threshold, threshold],'k--')
ax1.set_title('Test 1: Var explained by PCs');
ax1.set_xlabel('Principal component');
ax1.set_ylabel('Variance explained');#
ax1.legend(['Individual','Cumulative','Threshold'])

# Test 2
ax2.plot(range(1,len(rho2)+1),rho2,'x-')
ax2.plot(range(1,len(rho2)+1),np.cumsum(rho2),'o-')
ax2.plot([1,len(rho2)],[threshold, threshold],'k--')
ax2.set_title('Test 2: Var explained by PCs');
ax2.set_xlabel('Principal component');
ax2.set_ylabel('Variance explained');
ax2.legend(['Individual','Cumulative','Threshold'])

# Test 3
ax3.plot(range(1,len(rho3)+1),rho3,'x-')
ax3.plot(range(1,len(rho3)+1),np.cumsum(rho3),'o-')
ax3.plot([1,len(rho3)],[threshold, threshold],'k--')
ax3.set_title('Test 3: Var explained by PCs');
ax3.set_xlabel('Principal component');
ax3.set_ylabel('Variance explained');
ax3.legend(['Individual','Cumulative','Threshold'])

# Test 4
ax4.plot(range(1,len(rho4)+1),rho4,'x-')
ax4.plot(range(1,len(rho4)+1),np.cumsum(rho4),'o-')
ax4.plot([1,len(rho4)],[threshold, threshold],'k--')
```

```
ax4.set_title('Test 4: Var explained by PCs');
ax4.set_xlabel('Principal component');
ax4.set_ylabel('Variance explained');
ax4.legend(['Individual','Cumulative','Threshold'])

fig.show()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:39:
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```



[ ]: