Principal Component Analysis: Equation of Motion Extraction

Eli Miller

March 15, 2019

Abstract

In this exploration, we explore Principal Component Analysis (PCA) through the Singular Value Decomposition (SVD) as a method for representing data in a low-rank feature space. This is viewed through the lens of kinematics for a spring-mass system. We discuss how different factors can be translated into outcomes of the PCA analysis. We also discuss some limitations and challenges that these methods face with current implementation

1 Introduction and Overview

This project focuses on two main tasks: data extraction and PCA analysis. We have multiple camera angles of a paint can on a spring (ideal spring mass system) for different physical scenarios. Specifically, 1, 2, and 3 dimensional motion. We will also observe how excess noise in our signal manifests itself in the SVD. Data extraction works to process movie data into a time-series of Cartesian components using image processing techniques. Like most data analysis, this comprises the majority of the code and computational time necessary for this analysis.

Secondly is the SVD analysis, implementation, and interpretation. We will discuss how to interpret and use the result of the SVD and implement a low-rank reconstruction of our extracted data. This entire process will give a holistic look into the system in question through the lens of a coordinate transform into the "principal component space" which can be interpreted to have some physical manifestation due to the nature of our problem.

2 Theoretical Background

The Singular Value Decomposition aims to diagonal a data matrix A into orthogonal principal component directions. This takes the form of

$$A_{m,n} = U\Sigma V^* \tag{1}$$

where $U_{n,m}$ is the matrix of principal directions, $\Sigma_{m,m}$ is a diagonal matrix of principal components, and $V^*_{n,m}$ is a matrix of projections of principal components.

One thing important to note is that the SVD expects that our data has a mean zero. As a result, we need to subtract off the average of each row of A before preforming SVD analysis.

To preform a rank-r reconstruction of the data, we simply multiply out our decomposition with the first r components. Specifically:

$$A_{reconstructed} = U_{n,r}\Sigma_{r,r}V_{r,n} \tag{2}$$

3 Algorithmic Implementation and Development

3.1 Coordinate Extraction

we first will transform our stack of color images (type:uint8) into a stack of grayscale images (type:float64) for manipulation using a simple grayscale conversion. If in other applications, the color signature of our object was unique, we could use this information to guide our analysis. This is not implemented of necessary at this time. Like other applications of image processing using SVD, we will subtract the average image of our data set from our current image.

$$X = X_i - X_{avg}, i \in [0, t] \tag{3}$$

if we assume that our signature of interest has some "positive" signature, when subtracting the average, we expect that our signal will retain this "positivity". Thus we send all negative components of our relative image to zero

$$X_{i,j} < 0 = 0 \tag{4}$$

we then apply a Gaussian filter to the image in attempt to filter out small-scale noise associated with the background changing slightly in each frame. This should not suppress our intended image of the paint can, but rather filter out high frequency image features associated with small camera motions.

To find our can's location, we rely on the `scipy.ndimage` package for python (more details in appendix). In order to calculate the paint can location, we calculate each filtered image's center of mass

$$x_{cm} = \frac{\sum_{i=1}^{N} m_i x_i}{M}, \quad y_{cm} = \frac{\sum_{i=1}^{N} m_i y_i}{M} \tag{5}$$

where m_i is each relative pixels intensity after filtering.

3.2 Filtering and Data Assimilation

Notice that the position we are collecting is highly sinusoidal in nature with some noise from our position extraction algorithm. We can further clean up our paint can position by appropriately filtering our position data. We know that we are looking for some kind of periodic motion with a characteristic frequency ω_0. As a result, I choose to filter out all high frequencies using a first-order

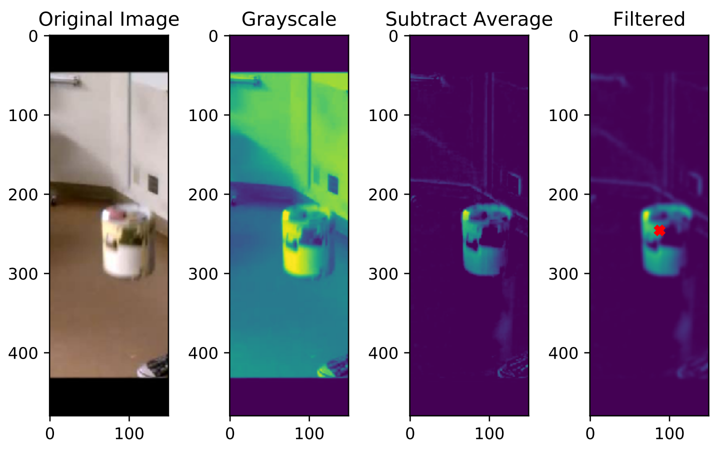

Figure 1: Visualization of coordinate extraction algorithm

low-pass filter with corner frequency $\omega = 0.15 * \omega_{max}$ where ω_{max} is the maximum frequency observed in each coordinates frequency spectrum. It may be important to look into whether this high-frequency component could be meaningful to the system. For this specific problem, however, we will let our physical intuition to guide this choice to filter our random noise.

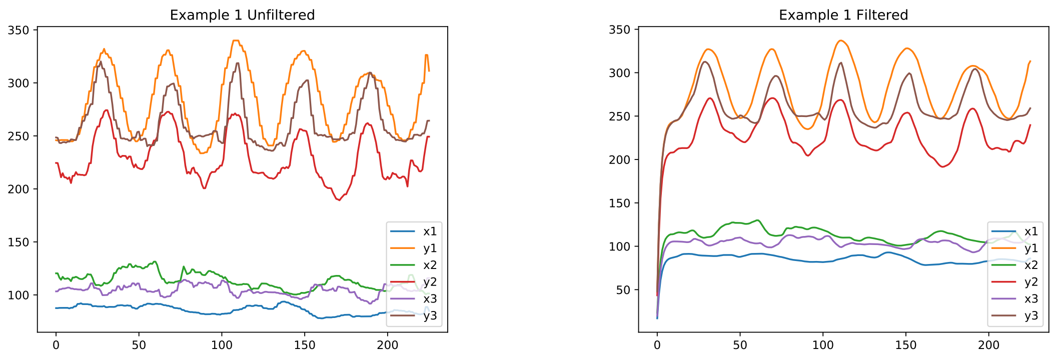

Figure 2: Visualization of filtering coordinate data (Pixel Position vs Time)

In addition to filtering, we attempt to align our data in time so that oscillatory components are relatively in phase. When watching the movies, we see that not all of the cameras begin filming at the same moment in time. For the PCA analysis, if our position measurements are not perfectly in phase, we expect that we may observe that we have more important principal components than actually present. Different signals will be changing at different times, when in reality they occurred at the same time. We also truncate our measurements so that we have the same number of data points $(t = 226)$ for all time series. This greatly simplifies our computational implementation.

3.3 Principal Component Analysis

We now have created a filtered, truncated data matrix where each column of A represents a collection of position measurements:

$$A(t) = \left[x_1, y_1, \ldots, x2, y_3\right]^T (t) \tag{6}$$

Using any modern computational language (Python, Matlab etc) obtaining the SVD of our matrix A is fairly trivial. We do need to structure our data such that each row has mean zero. Thus we subtract the average of each row from our matrix A before processing.

$$A_j = A_j - \text{mean}(A_j) \tag{7}$$

4 Computational Results

Our analysis was aimed at understanding the underlying dynamics of a spring-mass system as seen from different camera angles. We looked at four cases to guide our analysis:

1. Ideal 1-D motion: We expect that we have one dominant principal component who's direction governs the dynamics
2. Noisy 1-D motion: We expect more than one dominant principal component as a result of the noise
3. Ideal 2-D motion: We expect two dominant principal components corresponding to the two degrees of freedom
4. 2-D motion with rotation: We expect three dominant principal components if our algorithm has a way to track the rotation, hoIver without careful handling of the data this feature may not be "seen " by the SVD

4.1 Discussion of each Example

A summary of Principal Components can be found in Figure 3

While this data is useful, it is also useful to look rank reconstructions of our data to visualize how Ill our original signal is reconstructed. [1]

As expected, our Case 1 has one dominant mode corresponding to almost 60% of our signal's energy. Additionally, our other signals do not have as clean of a representation in the SVD projections. This is expected as they have more features associated with their measured data than Case 1.

[1]While all possible reconstructions are omitted for brevity, they can be easily created in the attached code with the function `rank_plot(A, rank, coordinate)`

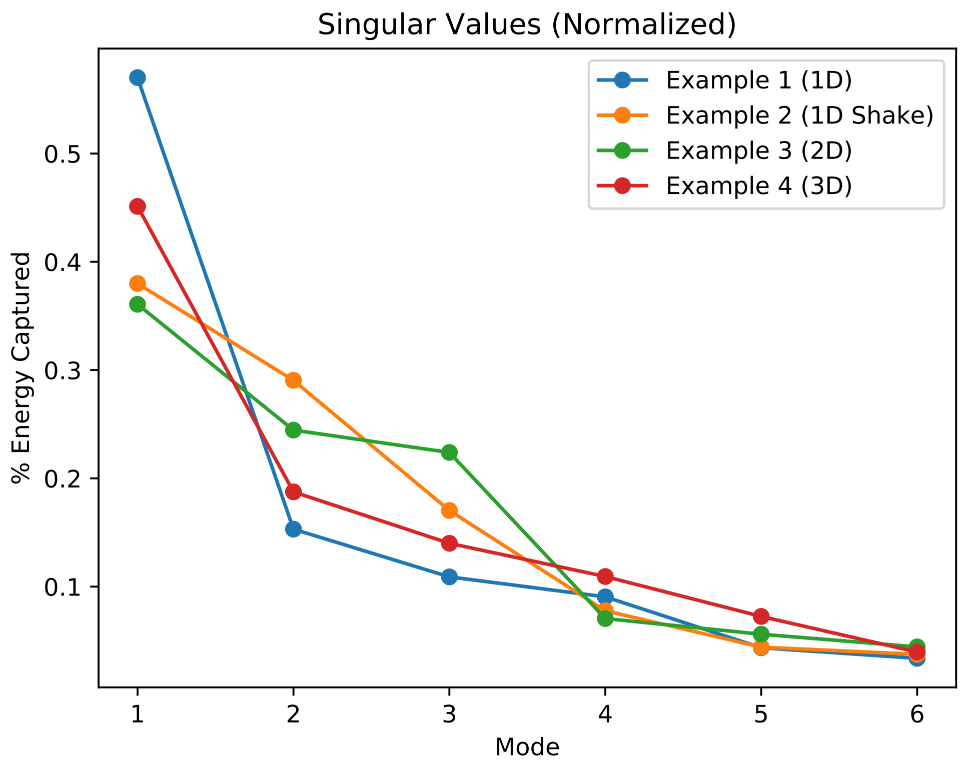

Figure 3: Summary of Principal Component's Energy for each example

Case 2, while recording the same information about one-dimensional motion has a much stronger 2nd and 3rd Principal component representation. This makes sense in the context of our measurements as tracking the paint can resulted in very noisy position data. We can think of the noise as manifesting itself as a "feature" of our recorded motion. Because we have noise in both x and y directions its makes sense that we have at least 3 components contributing to Case 2's accurate reconstruction.

Case 3 also has strong 1st, 2nd, and 3rd Principal Components. This time the extra components are arising from multiple degree-of-freedom motion. We see that our 1st rank reconstruction is highly inaccurate, which makes sense. There is more than one principal direction that describes the system's motion.

In Case 4, our physical intuition tells us that there should be one more large Principal Component than Case 3; the can is rotating in addition to undergoing planar motion. However, we see in Figure 3 that there is little difference between Cases 3 and 4's PCA signature. A reason that this may be occurring is that our measurements from the videos only consist of two components: x and y coordinates. It makes sense then, that we are not able to pick up rotation from this data. We are not measuring a feature corresponding to rotation, or deriving a synthetic feature that could help in picking out rotation. Rather, our analysis, and subsequently what we are able to learn from our data is bounded by *our* inputs. This highlights a common theme about the limitations of the SVD (and most data analysis techniques). ***While the SVD can work to decouple components of the types of data that we input, it cannot pull new information from thin air***

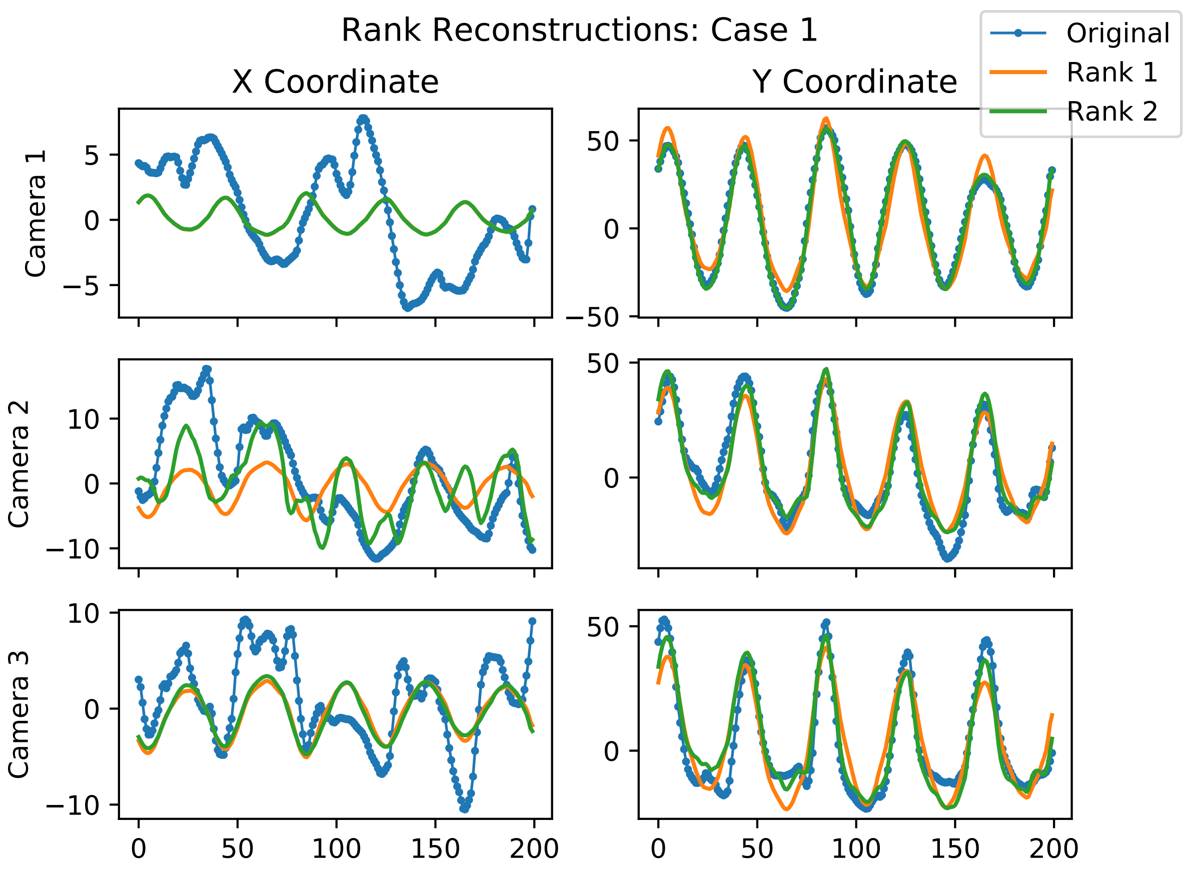

Figure 4: A look into Rank Reconstruction for the Ideal Case.

4.2 A Deeper Look at Rank Reconstruction

Now that we have discussed each specific case, we will dive deeper into looking at the Rank Reconstruction of Case 1. We will reconstruct with $r = 1, 2$ to observe the behavior of our data as represented in a Low-Rank Principal Component Space. Results are visually summarized in Figure 4. (Note that all of our plots have been shifted to have a mean = 0 for implementation with the SVD algorithm)

We see that for all of our y-components, the rank 1 approximation almost exactly matches our original measured data. While the rank 2 reconstruction does a bit better job at representing our data, no new ”features” arise. We still get an osculating shape with similar amplitude and period as our original signal for both reconstructions.

The SVD reconstruction provides a bit more insight when looking at the x-components. Because of the coordinate tracking algorithm, there is quite a bit of noise in the x-component reconstruction. When we reconstruct the x-components with rank 1, we see a very nice oscillatory behavior much like our components in y, but with a much smaller amplitude. When watching the raw videos, this is actually quite close to what is occurring in the physical system. While the majority of the motion is in the y-direction of the video frame, it is not perfectly aligned with one column of pixels. While it may only be translating in 1-dimension physically, our videos are not perfectly aligned with this principal direction. We pick up some of this motion in both directions through PCA.

5 Summary and Conclusions

This analysis explored the implementation of Singular Value Decomposition for Principal Component Analysis. WE worked to process image data into a set of coordinates that were fed into the SVD. In addition to looking at the PCA representation of our data, we leveraged the basis formed by the SVD to perform a low rank reconstruction that can approximate our data in just a few principal components. We also discussed the limitations of this kind of analysis and explored what effects tweaks to our input data can have.

Some steps moving forward with this kind of analysis could include leveraging the SVD for bigger and more complex systems. If this experiment were repeated, we could have used accelerator data or measurements of pitch, roll, and azimuth to gain insight into this system. It is important to realize that while the SVD is a useful for this assignment in the sense that it is easy to visualize what is going on in the system, the applications stray far and wide from just kinematic analysis. In moving forward to more advanced statistical analysis or Machine Learning techniques, the SVD and PCA are imperative tools for data preparation and feature engineering.

6 References

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

Mathworks. rgb2gray. Retrieved from https://www.mathworks.com/help/matlab/ref/rgb2gray.html

7 Appendix A: Functions Used

For this analysis, I relied heavily on the numpy and matplotlib libraries for Python. Additionally, I extensively explored using the scipy.ndimage ”n-dimensional image processing package” and a few functions from the scipy.signal library. Below are some specific functions used:

`scipy.io.loadmat`: Loads matlab files into a dictionary with relevant information. The key ’data’ extracts the necessary variables.

`scipy.ndimage.gaussian_filter`: Returns a Gaussian filtered image with desired filter width

`scipy.ndimage.measurements.center_of_mass`: Returns a ”center of mass” of an image(either boolean in nature or weighted). This was very useful in creating my position extraction algorithm as it alloId considering many points, and not just finding one maximum pixel that tends to jump around `scipy.signal.butter`: Creates signal parameters associated with a Butterworth filter of desired order. First order filter was implemented as a quick approach to a simple low pass filter.

`scipy.signal.lfilter`: Reads the parameters output by 'butter' and outputs a filtered signal. This is a very simple yet powerful command within the scipy.signal library.

`np.linalg.svd`: The heart of the math portion of this analysis. Computes the Singular Value Decomposition

`np.rot90`: Rotates an array by 90°about a particular axis. Used to rotate camera 3 and simplify cropping the video frames.

8 Appendix B: Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 14 14:29:54 2019

@author: elimiller
"""

import numpy as np
from numpy import pi
import matplotlib.pyplot as plt
from scipy.io import loadmat
import scipy.ndimage
import scipy.signal

def rgb2gray(image):
    R = image[:, :, 0]
    G = image[:, :, 1]
    B = image[:, :, 2]
    gray = 0.2989 * R + 0.5870 * G + 0.1140 * B
    return np.array(gray)

def averageim(stack):
    if len(np.shape(stack)) == 4:
        stack = rgb2gray(stack)
    average_sum = np.sum(stack, axis=2)
    return average_sum / np.ma.size(stack, axis=2)

def plot_fft(y):
    #frequencies normalized from [-1, 1]  for digital filter frequency
    y_freq = np.abs(np.fft.fftshift(np.fft.fft(y)))
    n = len(y_freq)
    k = 2/n * np.concatenate((np.arange(0, (n/2)), np.arange(-n/2, 0)))
    #fftshift frequency domain
    ks = np.fft.fftshift(k)
    plt.figure()
    plt.plot(ks, y_freq)

def low_pass(y, pct_max_freq):
    b, a = scipy.signal.butter(1, pct_max_freq, 'lowpass' )
    y_filt = scipy.signal.lfilter(b, a, y)
    return y_filt

def get_com(
        framestack,
        xbounds,
        startval=0,
        SHOW_AVERAGE=False,
        FILTER_COM=True,
        PLOT_TRACK=False):

    STORELEN = 226
    rightbnd = xbounds[0]
    leftbnd = xbounds[1]
    vid_frames = framestack[:,rightbnd:leftbnd,:,:]
    average_frame = averageim(vid_frames)

    if SHOW_AVERAGE:
        plt.figure()
        plt.imshow(average_frame)

    com_x = []
    com_y = []
    for j in range(np.ma.size(vid_frames, axis=3)):
        frame = vid_frames[:,:,:,j]
        frame_bw = rgb2gray(frame)
        frame_delta =  frame_bw - average_frame
        #once we do this, we have transformed out of [0, 255]
        #now our pixel values are relative!

        frame_delta[frame_delta < 0] = 0
        # send negative values of frame_delta to zero

        frame_filter = scipy.ndimage.gaussian_filter(frame_delta, 2)
        com = scipy.ndimage.measurements.center_of_mass(frame_filter)
        com_x.append(com[1])
        com_y.append(com[0])

        if PLOT_TRACK and j == 1:
            #this is solely to produce a nice figure for writeup
            bigfig, axs = plt.subplots(1, 4)
```

```
                axs[0].imshow(frame)
                axs[1].imshow(frame_bw)
                axs[2].imshow(frame_delta)
                axs[3].imshow(frame_filter)
                axs[3].plot(com[1], com[0], 'rx')

    #truncate to the same length
    com_x = com_x[startval:startval+STORELEN]
    com_y = com_y[startval:startval+STORELEN]

    if FILTER_COM:
        com_x_filter = low_pass(np.array(com_x), .15)
        com_y_filter = low_pass(np.array(com_y), .15)
        return com_x_filter, com_y_filter
    else:
        return np.array(com_x), np.array(com_y)

def check_com(framestack, com_x, com_y, num_frames):
    for j in range(num_frames):
        plt.imshow(rgb2gray(framestack[:,:,:,j]))
        plt.plot(com_x[j], com_y[j], 'rx')
        plt.pause(.1)

def rank_approx(A_rel, rank):
    #this expects rows with mean 0
    U, S_vec, V = np.linalg.svd(A_rel)
    S = np.diag(S_vec)
    A_approx = U[:,0:rank] @ (S[0:rank, 0:rank] @ V[0:rank, :])
    return A_approx

def rank_plot(A, rank, coordinate):
    A_relative = A - A.mean(axis=1, keepdims=True)
    A_approx = rank_approx(A_relative, rank)
    plt.figure()
    plt.plot(A_relative[coordinate,:], '.-')
    plt.plot(A_approx[coordinate,:], '-')
    plt.title('Rank %d Reconstruction' %rank )
    plt.legend(['Original Data', 'Low-Rank Reconstruction'])

plt.close('all')
cameras = [1, 2, 3]
examples = [1, 2, 3, 4]
bounds = [(250, 400),(200, 400),(100, 300)]
#indicies of windows of where to look for can
startvalstore = [[0, 10, 0],
                 [0, 10, 0],
                 [0, 0, 0],
                 [0, 8, 0]]
```

```python
#indicies of where to start videos to align them in time

for example in examples:
    bnd = 0
    STORELEN = 226
    startvals = startvalstore[bnd]
    A_temp = np.zeros((1, STORELEN))
    for camera in cameras:
        path = '/Users/elimiller/Desktop/AMATH482/HW3/camfiles/cam%d_%d.ma
                camera, example)
        vid_frames_dict = loadmat(
                path)
        leftbnd = bounds[bnd][0]
        rightbnd = bounds[bnd][1]
        startval = startvals[bnd]

        vid_frames = vid_frames_dict['vidFrames%d_%d'%(camera, example)]

        if camera == 3:
            #the 3rd camera can be rotated because its sideways
            #we could forgo this and the SVD wouldn't care
            #however, it makes implementation easier for trimming frames
            vid_frames = np.rot90(vid_frames,k=-1)

        com_x, com_y = get_com(
                vid_frames, (leftbnd, rightbnd), startval=startval,
                SHOW_AVERAGE=False, PLOT_TRACK=True, FILTER_COM=True)

        temp = np.vstack([com_x, com_y])
        A_temp = np.vstack([A_temp, temp])

        print('Example %d, Camera %d'%(
                example, camera))
        print('Range of X=%.3f and Range of Y=%.3f' %(
                np.ptp(com_x), np.ptp(com_y)))

        bnd += 1

        if False:
            #Plots x and y coordinate of current camera
            fig, axs = plt.subplots(2,1, sharey=True)
            axs[0].plot(com_x, '.-')
            axs[0].set_title('x coordinate')
            axs[1].plot(com_y, '.-')
            axs[1].set_title('y coordinate')

    #Store in different matricies so that we don't have to run this again
    if example == 1:
        A1 = A_temp[1::]
    if example == 2:
        A2 = A_temp[1::]
    if example == 3:
        A3 = A_temp[1::]
    if example == 4:
        A4 = A_temp[1::]


#Plot all components for each case.  Used to align video frames by hand
for A in [A1, A2, A3, A4]:
    plt.figure()
    for j in range(6): plt.plot(A[j, :])
    plt.legend(['x1', 'y1', 'x2', 'y2', 'x3', 'y3'], loc='lower right')

plt.figure()
#plot of Singluar values for each case
for A in [A1, A2, A3, A4]:
    A_rel = A - A.mean(axis=1, keepdims=True)
    U, S, V = np.linalg.svd(A_rel)
    plt.plot(range(1, 6+1), S / np.sum(S), 'o-')
    plt.yscale('linear')
    plt.ylabel('% Energy Captured')
    plt.xlabel('Mode')
    plt.legend(('Example 1 (1D)',
               'Example 2 (1D Shake)',
               'Example 3 (2D)',
               'Example 4 (3D)' ))

    plt.title('Singular Values (Normalized)')
    plt.savefig('SingularValues.pdf')


#Produce plots of rank reconstruction
#Investigate mean squared error for each case

#plt.close('all')
error_store = np.zeros((6, 6, 4))

rankval = np.arange(1, 6+1)
coordinates = np.arange(0, 5+1)

As = [A1, A2, A3, A4]
#As = [A4]
layer = 0
for A in As:
    for rank in rankval:
        for j in range(len(coordinates)):
            coordinate = coordinates[j]
            A_relative = A - A.mean(axis=1, keepdims=True)
            A_approx = rank_approx(A_relative, rank)
            error = np.sqrt(np.sum((A_approx[j,:] - A_relative[j,:])**2))
            error_store[coordinate, rank-1, layer] = error
    layer += 1

if False:
    plt.plot(A_relative[coordinate,:], '.-')
    plt.plot(A_approx[coordinate,:], '-')
    plt.title('Rank %d Reconstruction with error %d' %(rank, error))
    plt.legend(['Original Data', 'Low-Rank Reconstruction'])


#Lets make an informative rank reconstruction figure!
#plt.close('all')
trimstart = 26
#trim off the beginning to condense in the y direction

A_relative = (A1 - A1.mean(axis=1, keepdims=True))[:,trimstart::]
rankval = [1, 2]
ax_index = [(0, 0), (0, 1), (1, 0), (1, 1), (2,0), (2,1)]

fig, axs = plt.subplots(3, 2, sharex=True, sharey=False)
for j in range(6):
    axs[ax_index[j]].plot(A_relative[j,:], '.-', linewidth=1, markersize=4
    for rank in rankval:
        A_approx = rank_approx(A_relative, rank)
        axs[ax_index[j]].plot(A_approx[j,:])

fig.suptitle('Rank Reconstructions: Case 1')
fig.legend(['Original', 'Rank 1', 'Rank 2'])

axs[0,0].set_title('X Coordinate')
axs[0,1].set_title('Y Coordinate')
for j in range(3):
    axs[(j,0)].set_ylabel('Camera %d' %(j+1))

plt.savefig('RankReconstruction.pdf')
```