AMATH 582 Homework 3

Daniel Burnham (https://github.com/burnhamdr)

February 21, 2020

Abstract

The work presented here is motivated by material covered in AMATH 582 Computational Methods For Data Analysis regarding the applications of Singular Value Decomposition (SVD). SVD is a linear algebra matrix factorization method that represents how the matrix of interest stretches/compresses and rotates a given set of vectors. The SVD is a powerful tool because the constitutive components of its factorization allow for the subsequent projection of a matrix onto low-dimensional representations. In the problem addressed here this is used to determine axes of motion of a spring-mass pendulum system from three different camera perspectives. Each camera records the action of the pendulum in a coordinate system relative to the camera which does not allow for complete characterization of the motion of the pendulum system from any one perspective. Gathering together positional information from each camera recording and applying SVD allows for the data matrix to subsequently be projected onto a low-dimensional space representing the axes where the most motion is detected in each camera. Principal Component Analysis (PCA) is the method used for this purpose and is an analysis concerned with the diagonalization and thus the dimensionality reduction of the covariance matrix of a system. The diagonalization in this instance will be performed using SVD allowing for the removal of redundancies and extraction of an ordered list of the axes of largest measurement variances (principal components). The experiments described in this work will describe PCA in more detail through practical application.

1 Introduction and Overview

The problem addressed here involves the analysis of data from three different cameras recording video perspectives of a spring-mass pendulum system. From this camera data the behavior of the system is described and the governing equations of motion determined. This analysis required work in two main categories: 1) Determining the position of the pendulum in each video frame for each camera 2) Applying PCA to position data from all camera perspectives to capture system dynamics. Three distinct cases of pendulum tracking and subsequent dynamics extraction will occur:

1. (Test 1) Ideal Case: Small displacement of the pendulum mass in the z direction. In this case, the entire motion is in the z direction.
2. (Test 2) Noisy Case: Repeat of the ideal case experiment, but in this case camera shake introduces noise to the video recording.
3. (Test 3) Horizontal Displacement Case: In this experiment, the pendulum mass is released off-center so as to produce motion in the xy plane as well as the z direction.
4. (Test 4) Horizontal Displacement and Rotation Case: In this experiment, the mass is released off-center and rotates so as to produce motion in the xy plane, rotation as well as the z direction.

2 Theoretical Background

2.1 Singular Value Decomposition (SVD)

SVD is a linear algebra method for factorization of a matrix into a number of constitutive components. It is rooted in the observation that during matrix vector multiplication of a vector **x** by a matrix **A** the

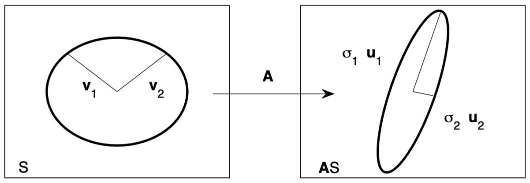

Figure 1: Image of a unit sphere S transformed into a hyperellipse $\mathbf{A}S$ in $\mathbb{R}^n$. The values of $\sigma 1$ and $\sigma 2$ are the singular values of the matrix $\mathbf{A}$ and represent the lengths of the semiaxes of the ellipse. (Kutz 2013)

resulting vector $\mathbf{y}$ has a new magnitude and direction. This transformation of the vector $\mathbf{x}$ by a matrix $\mathbf{A}$ implies that perhaps the action of matrix $\mathbf{A}$ can be replicated through component matrices that perform the same magnitude and direction manipulations. Furthermore, it would be beneficial if these components possessed properties that made them easy to work with, such as orthogonality and diagonality. The SVD factorization of the matrix $\mathbf{A}$ achieves these goals by expanding this observation of vector transformation under multiplication by a matrix to $\mathbb{R}^m$. SVD does this by building from the observation that the image of a unit sphere under any m×n matrix is a hyperellipse. This can be visualized more clearly in Figure 1. Following from this, one can represent this transformation as $\mathbf{A}\mathbf{v}_j = \sigma_j \mathbf{u}_j$ for $1 \leq j \leq n$. Where $\mathbf{v}_j$ are the vectors of the unit sphere transformed by $\mathbf{A}$, and $\sigma_j \mathbf{u}_j$ are the resulting transformations representing the the semiaxes of the hyperellipse. Rearranging this equation allows for the SVD factorization of $\mathbf{A}$ to be written as follows (Kutz 2013):

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{1}$$

In this form, $\mathbf{U}$ is unitary and contains the vectors $\mathbf{u}_j$ indicating the directions of the transformed hyperellipse semiaxes, $\mathbf{\Sigma}$ is diagonal and contains the scaling values corresponding to these semiaxes, and $\mathbf{V}^*$ is the Hermitian transpose of $\mathbf{V}$ which contains the orthonormal basis of the vectors that are transformed under $\mathbf{A}$. It is worth noting that it can be proved that every matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$ has a singular value decomposition and the singular values are uniquely determined (Kutz 2013). Additionally, if the matrix $\mathbf{A}$ is square, the singular vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ are also uniquely determined up to complex signs (Kutz 2013). This is significant because it allows for every $\mathbf{A} \in \mathbb{C}^{m \times n}$ to be factorized by SVD and subsequently represented with lower rank matrix approximations. This will be useful in the context of the pendulum problem explored here as it represents a way of reducing the dimensionality of the pendulum positional data.

2.2 Principal Component Analysis (PCA)

PCA is a variant of SVD that is motivated by finding low-dimensional reduction of information from seemingly complex data. In the context of the experiments investigated in this report, this complex data is the pendulum positions recorded for each video frame captured by the perspective of each camera. The intuition for PCA builds from the identification of two characteristics of complex data sets: noise and redundancy. Noise in the data has the potential to obscure meaningful information represented by the data. Redundancy in the data refers to data observations that overlap in that they contribute to describing variance along the same dimension of the overall data. Addressing this redundancy is critical for data analysis and relies on the covariance matrix of the data. Covariance measures the statistical dependence or independence between two variables. The covariance matrix of a set of data $\mathbf{X}$ with itself will elucidate which observations within the dataset are redundant, or in other words, how much each observational measurement correlates with measurements of other observations of the dataset. This covariance matrix is thus defined as (Kutz 2013):

$$\mathbf{C}_{\mathbf{X}} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^T \tag{2}$$

The covariance matrix $\mathbf{C}_{\mathbf{X}}$ is square and symmetric with diagonal entries that represents the variance of of each observation. The off-diagonal terms are the covariances between observations. In this way, the covariance matrix represents how each observation correlates with every other observation in the data. Redundancies within the data are thus identifiable by off-diagonal terms, as large off-diagonal values indicate a high degree of correlation (redundancy) while the opposite is true for small off-diagonal values. The diagonal terms are of particular interest because large values here indicate a high degree of variance within a particular observation thus indicating a variable of potential interest in driving differentiation in output variables. In the case of the pendulum system investigated in this report, this output variable is position, and the observations are the x and y position data from each camera. A large diagonal term for this data matrix would thus indicate a dimension of a specific camera perspective that captures a significant portion of the pendulum system dynamics. The goal of PCA is to then take this covariance matrix and extract the most interesting descriptions of the underlying data. This is accomplished by diagonalizing the covariance matrix because through diagonalization the correct coordinate system relative to the data is determined and can be used to reduce the dimensionality of the data. SVD presents as a promising candidate for diagonalizing the covariance matrix. As discussed in Section 2.1 the SVD factorizes a matrix $\mathbf{A}$ into two sets of bases and a diagonal matrix. One way of conceptualizing this factorization is that the orthonormal bases $\mathbf{U}$ and $\mathbf{V}$ are the bases that diagonalize the matrix $\mathbf{A}$. The mathematical foundation of PCA begins with this conceptualization in order to diagonalize the covariance matrix. In order to find the appropriate bases for this diagonalization, the first step is to define a transformed variable $\mathbf{Y}$ as $\mathbf{Y} = \mathbf{U}^*\mathbf{X}$ where $\mathbf{U}$ is the unitary transformation associated with the SVD of $\mathbf{X}$ (Equation 1). From this, the variance in $\mathbf{Y}$ can be calculated as follows (Kutz 2013):

$$\begin{aligned}\mathbf{C_Y} &= \frac{1}{n-1}\mathbf{Y}\mathbf{Y}^T \\ &= \frac{1}{n-1}(\mathbf{U}^*\mathbf{X})(\mathbf{U}^*\mathbf{X})^T \\ &= \frac{1}{n-1}\mathbf{U}^*(\mathbf{X}\mathbf{X}^T)\mathbf{U} \\ &= \frac{1}{n-1}\mathbf{U}^*\mathbf{U}\boldsymbol{\Sigma}^2\mathbf{U}\mathbf{U}^* \\ &= \frac{1}{n-1}\boldsymbol{\Sigma}^2\end{aligned} \tag{3}$$

Through this diagonalization the ideal basis is found in which the covariance matrix $\mathbf{C}_X$ can be represented such that redundancies are removed, and the largest variances of particular observations are ordered. More specifically, the basis formed by the left singular vectors of $\mathbf{X}$ (the columns of $\mathbf{U}$) is the ideal basis in which to express $\mathbf{X}$ such that the covariance matrix ($\mathbf{C}_X$) is diagonal. The largest diagonal entries of $boldsymbol\Sigma$ thus correspond to the directions of largest variance of $\mathbf{X}$.

3 Algorithm Implementation and Development

The main steps of the algorithm are as follows:

1. Input the video file, prepare for analysis
2. For each camera locate paint can pendulum in each video frame
3. Collect positional data from each camera and perform PCA
4. Plot dynamics along the principal component directions by video frame for each test case to visualize the dynamics of the spring-mass pendulum system.

3.1 Pendulum Position Detection

The nature of the recorded videos presents a challenge in reliably detecting the paint can (the pendulum mass) in each video frame. In order to address this challenge some image processing techniques are employed based on observational evidence. Firstly, the pendulum oscillates largely in the foreground of the video allowing for foreground filtering (Otsu's method) to limit the interference of the background. Secondly, a high pass filter helps to emphasize objects in the foreground that have edges, such as the paint can pendulum. These two image processing methods allow for the white pixel features of the paint can to be made the most prominent feature of each video frame. After this preprocessing, a search window is subsequently iterated across each video frame searching for the windowed portion of the video frame with the highest average pixel value. The robustness of the preprocessing methods allow for the location of this window within the frame to reliably represent the position of the paint can.

Algorithm 1: Track Pendulum Position

```
Determine shape of data matrix (i.e. width and height of individual frames, and number of video frames)
for each video frame do
  convert video frame to grayscale
  apply otsu foreground filter to band of pixels around the edge of the frame
  apply high-pass filter to foreground filtered grayscale video frame
  for each x pixel coordinate do
    for each y pixel coordinate do
      if pixel value > threshold then
        set pixel value to maximum value (255)
      else
        set pixel value to zero
      end if
    end for
  end for
  for each x pixel coordinate within search window do
    for each y pixel coordinate within search window do
      calculate average pixel value
      if average pixel value > max average pixel value then
        set position of maximum average pixel value to the current location in pixel space of the upper
        left corner pixel of the search window
      else
        keep previous position of maximum average pixel value
      end if
    end for
  end for
  add position of maximum pixel value to list of positions tracking the pendulum position across all
  video frames
end for
```

4 Computational Results

4.1 Test 1: Simple Harmonic Oscillations

See Figure 2 for the results of applying PCA to the positional data returned by Algorithm 1. Oscillations are evident along a single dimension. These observed system dynamics are supported by the percent explained variance of the first principal component individually representing roughly a third of the overall variance of the system. The rest of the variance observed is attributable to noise based on the nature of the dynamics illustrated by the projections of the data along the second and third principal components. These findings are consistent with the conditions posed for test 1 where the entire motion of the spring-mass pendulum system was restricted to the z directions with the goal of observing simple harmonic motion.

4.2 Test 2: Simple Harmonic Oscillations with Noise

See Figure 3 for the results of applying PCA to the positional data returned by Algorithm 1. Oscillations are evident along a single dimension. These observed system dynamics are supported by the percent explained variance of the first principal component individually representing over 35 percent of the overall variance of the system. The rest of the variance observed is attributable to noise (similarly to test 1) based on the nature of the dynamics illustrated by the projections of the data along the second and third principal components.

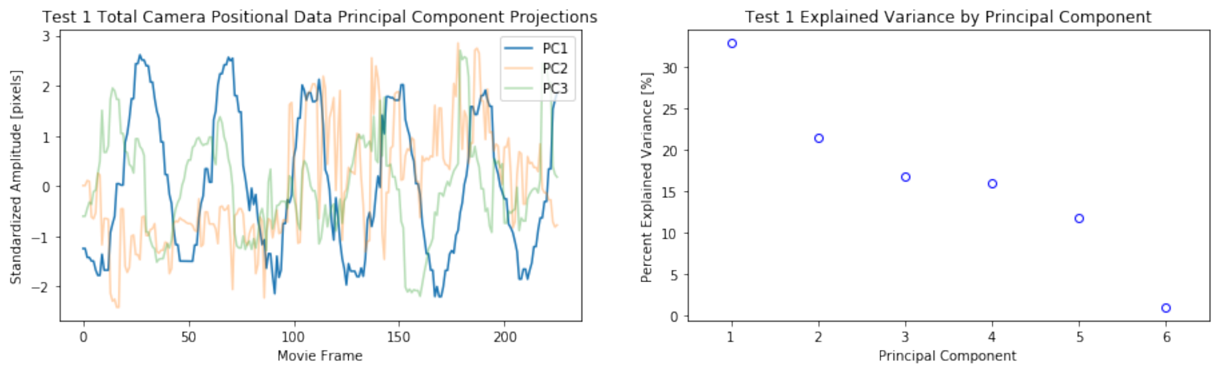

Figure 2: (Left) Plot of the principal component projections of the camera positional data for the top three principal components across the video frames recorded. (Right) Plot of the explained variance along each principal component

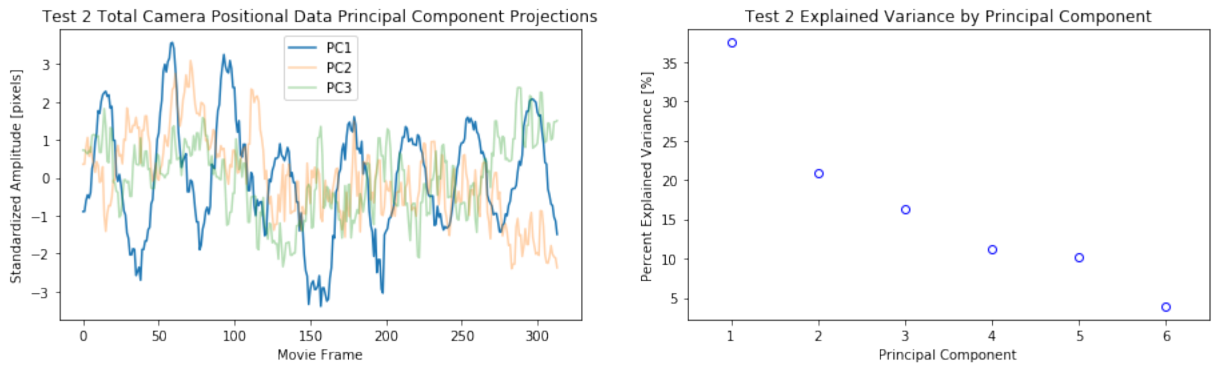

Figure 3: (Left) Plot of the principal component projections of the camera positional data for the top three principal components across the video frames recorded. (Right) Plot of the explained variance along each principal component

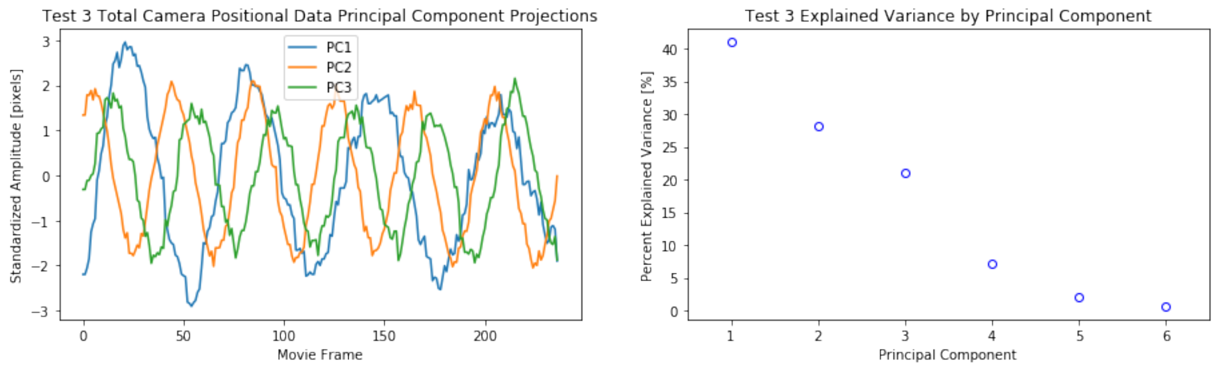

Figure 4: (Left) Plot of the principal component projections of the camera positional data for the top three principal components across the video frames recorded. (Right) Plot of the explained variance along each principal component

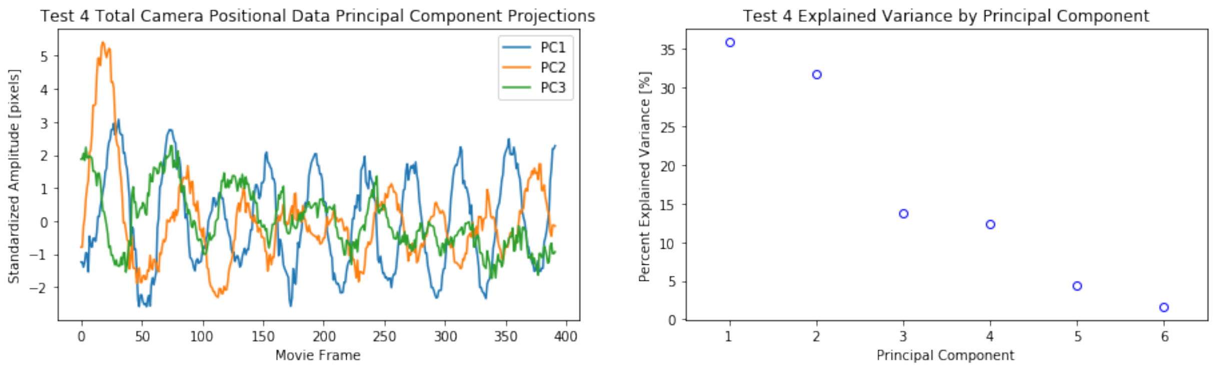

Figure 5: (Left) Plot of the principal component projections of the camera positional data for the top three principal components across the video frames recorded. (Right) Plot of the explained variance along each principal component

These findings are consistent with the conditions posed for test 2 where the entire motion of the spring-mass pendulum system was restricted to the z direction, but noise in the form of camera shake was introduced to the video recordings. Simple harmonic motion remains evident in projecting the data along the first principal component.

4.3 Test 3: Horizontal Pendulum Motion and Simple Harmonic Oscillations

See Figure 4 for the results of applying PCA to the positional data returned by Algorithm 1. Oscillations are evident along a three dimensions. These observed system dynamics are supported by the percent explained variance of the first, second, and third principal components individually representing approximately 40, 30, and 20 percent of the overall variance of the system respectively. The rest of the variance observed is attributable to noise. These findings are consistent with the conditions posed for test 3 where the motion of the spring-mass pendulum system should occur in the x, y, and z directions.

4.4 Test 4: Horizontal Pendulum Motion and Simple Harmonic Oscillations.. with a Twist

See Figure 5 for the results of applying PCA to the positional data returned by Algorithm 1. Oscillations are evident along a three dimensions. These observed system dynamics are supported by the percent explained variance of the first, second, and third principal components individually representing approximately 35, 30, and 15 percent of the overall variance of the system respectively. The rest of the variance observed is attributable to noise. These findings are consistent with the conditions posed for test 4 where the motion of the spring-mass pendulum system should occur in the x, y, and z directions. The rotation of the pendulum is not captured by the positional data and simply made tracking the pendulum between video frames more challenging.

5 Summary and Conclusions

Overall a video processing algorithm was successfully developed for the extraction of positional data from three unique camera perspectives characterizing a spring-mass pendulum system. This algorithm was tunable allowing for reliable pendulum tracking in all test cases. PCA was successfully implemented to elucidate the dynamics of this physical system along the most relevant axes for each experimental test case.

References

Kutz, Jose Nathan (2013). *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press.

Appendix A Python Functions

- `sklearn.decomposition.PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)`
  Principal component analysis (PCA). Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD. It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract. It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

- `numpy.fft.fft2(a, s=None, axes=(-2, -1), norm=None)`
  Compute the 2-dimensional discrete Fourier Transform. This function computes the n-dimensional discrete Fourier Transform over any axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

- `sklearn.preprocessing.StandardScaler(copy=True, with_mean=True, with_std=True)`
  Standardize features by removing the mean and scaling to unit variance

- `numpy.amax(a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>)`
  Return the maximum of an array or maximum along an axis.

- `skimage.filters.threshold_otsu(image, nbins=256)`
  Return threshold value based on Otsu's method.

Appendix B Python Code

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

# In[ ]:


#Import statements
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time
import matplotlib.animation
from PIL import Image
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import pandas as pd
import math
from skimage import filters
```

```
# In[ ]:


#Functions
'''Adapted from stack overflow response from Jeru Luke,
https://stackoverflow.com/questions/51285593/converting-an-image-to-grayscale-using-numpy

Takes in color image and converts to gray scale'''
def grayConversion(image):
    grayValue = 0.07 * image[:,:,2] + 0.72 * image[:,:,1] + 0.21 * image[:,:,0]
    gray_img = grayValue.astype(np.uint8)
    return gray_img
'''
    The bucketTracker function takes in a frame from a video of a paint bucket attached to a spring
    oscillating in time and tracks the position of the paint bucket across each frame of the video.
    This is accomplished by applying a foreground filter skimage.filters.threshold_otsu to the edges
    of the image (to eliminate instances of erroneously tracking elements of the video background),
    as well as a high pass edge detection filter. Both of the filters are controllable with funciton
    parameters. The amount of the image subjected to the foreground filter is controllable in both the
    x and y direction, and the extent to which low frequencies are attenuated by the high pass filter is
    also controllable. After filtering the function then proceeds to increase the contrast of the image
    blacking out (setting pixel values to zero) all pixels below a certain threshold while amplifying pi
    above the threshold to the maximum grayscale pixel value of 255. This is done because the color of t
    can is white, and amplifying the contrast of the pixel values of the can is desirable for good track
    performance. This filtered, enhanced contrast version of the original video frame is then searched u
    a window technique to find the square window of pixel values with the highest average pixel value. T
    correspond to the location of the paint bucket for the given frame.


    INPUTS:
    data: numpy.ndarray of signal to analyze.
    pix_thresh: scalar value indicating the relative (i.e. pix_thresh = x/255) pixel value below which
                pixels will be blacked out (i.e. set to 0) and above which will be amplified (i.e. set t
    prefilter: Boolean value indicating whether to use the contrast enhancing feature which amplifies
                pixel values above pix_thresh, and zeros out all values below the threshold.
    window_size: scalar value indicating the size of the search window to use when looking for the
                maximum average pixel value corresponding to the position of the bucket
    search_restrict: scalar value indicating the size of the square window around which the algorithm wi
                    begin searching relative to the last known position.
    error_thresh: scalar value indicating the tolerance for the tracking position to jump between frames
    keep_fraction: scalar value indicating the relative number of frequencies to keep during high pass f
    mask_length: scalar value indicating the thickness of the edge in the y-direction of the
                    foreground filter.
    mask_width: scalar value indicating the thicknes of the edge in the x-direction of the
                foreground filter.
    highPassFilter: boolean value indicating whether or not to apply the high pass filter

    OUTPUTS:
    positions: numpy.ndarray of the determined position (x,y pixel values) of the paint bucket in each f
    new_movie: nump.ndarray of the video frames after filtering and contrast enhancement to demonstrate
                what the function does to the video file. This aids in tuning the parameters of the fil
                and thresholds so as to best track the bucket.
'''
```

```
def bucketTracker(data, pix_thresh, prefilter=True, window_size=10, search_restrict=40,
                 error_thresh=0.70, keep_fraction=0.1, mask_length = 0.1, mask_width = 0.1,
                highPassFilter=True):
    #find bucket pixel position using the flashlight due to identifiably high whiteness.
    [w, h, rgb, tp] = data.shape
    new_movie = np.empty([w, h, tp])
    positions = np.empty([tp, 2])#array to hold positions of paint can in each movie frame
    start_j = 0#y coordinate for starting search for paint can
    start_k = 0#x coordinate for starting search for paint can
    stop_j = h#y coordinate to stop search for paint can
    stop_k = w#x coordinate to stop search for paint can

    #iterate over each frame of the movie
    for i in range(0, tp): #iterate over each frame
        print("Progress {:2.1%}".format(i / tp), end="\r")#print progress of paint can position acquisi
        gf = grayConversion(data[:,:,:,i])#convert RGB frame to gray scale pixel values
        val = filters.threshold_otsu(gf)
        foreground_mask = gf < val
        ledge = mask_length#specifies how much of the edge of the foreground filter to include in the x
        wedge = mask_width#specifies how much of the edge of the foreground filter to include in the y
        rfg, cfg = foreground_mask.shape#Set r and c to be the number of rows and columns of the array.
        foreground_mask[int(rfg*(wedge)):int(rfg*(1-wedge)),int(cfg*ledge):int(cfg*(1-ledge))] = 1

        if highPassFilter:
            imfft = np.fft.fft2(gf)#discrete fourier transform of movie frame
            #high pass filter for edge detection
            r, c = imfft.shape#Set r and c to be the number of rows and columns of the array.
            mask = np.zeros((r, c))#initialize mask of zeros to act as foundation for high pass filter
            # Set to 1 all rows with indices between r*keep_fraction and
            # r*(1-keep_fraction):
            mask[int(r*keep_fraction):int(r*(1-keep_fraction))] = 1

            # Similarly with the columns:
            mask[:, int(c*keep_fraction):int(c*(1-keep_fraction))] = 1
            im_lp = imfft*mask#Apply high pass filter mask
            imifft = np.fft.ifft2(im_lp)#inverse fourier transform
        else:
            imifft = 1.0

        gf = gf*((abs(imifft)/np.amax(abs(imifft))))*foreground_mask.astype(int)
        #iterate over each column of the frame and highlight the whitest pixels while blackening
        #every pixel below the set threshold. This serves to emphasize what pixels might be from
        #the flashlight on top of the paint can in the movie
        if prefilter:
            for l in range(0, h):
                vec = gf[:, l]#grab column vector of pixel data
                for m in range(0, w):#iterate over the column of pixel data
                    if vec[m] > pix_thresh*255:#if a pixel value is above the threshold for white
                        vec[m] = 255#maximize white value
                    else:
                        vec[m] = 0#otherwise zero out (blacken) the pixel
                gf[:, l] = vec#overwrite pixel data of the frame with new filtered pixel values

        #window_size = 10#set a window size to use for iterating over the frame to look for the flashli
        window_area = window_size**2#calculate the area of the window for calculating average window pi
        av_max = 0#initialize a maximum average window pixel value for initial maximum average value co
        pos = np.array([0, 0])#initialize an array to contain the determined position of the flashlight

        #iterate over the specified search window to find the flashlight indicated by maximum average w
        #the first search will iterate across the entire frame. subsequent frame searches will restric
        #window to a smaller area surrounding the last known flashlight position
        for j in np.arange(start_j, stop_j, window_size):
            for k in np.arange(start_k, stop_k, window_size):
                if ((j + window_size) > h) and ((k + window_size) > w):#if the window is out of frame i
                    window = gf[j:h, k:w]#truncate window so the edge of the window is the edge of the
                elif (j + window_size) > h:#if the window is out of frame in the y direction
                    window = gf[j:h, k:k + window_size]#truncate window so the edge of the window is th
                elif (k + window_size) > w:#if the window is out of the frame in the x direction
                    window = gf[j:j + window_size, k:w]#truncate window so the edge of the window is th
                else:#otherwise the window size is unrestricted
                    window = gf[j:j + window_size, k:k + window_size]

                av = np.sum(np.sum(window)) / window_area#calculate the average pixel value of the wind
                av_max = max(av, av_max)#compare the average window value to the previous maximum value
                if av_max == av:#if the max is the newest calculated value update the position of the p
                    pos = np.array([k, j])

        #update the start of the search window to be just before the last known position
        start_j = pos[1] - search_restrict
        start_k = pos[0] - search_restrict
        #update the end of the search window to be just after the last known position
        stop_j = pos[1] + search_restrict
        stop_k = pos[0] + search_restrict

        #add bounds so that the subframe searched for the paint can cannot ever be out of bounds of the
        if start_j < 0:
            start_j = 0
        if start_k < 0:
            start_k = 0
        if stop_j > h:
            stop_j = h
        if stop_k > w:
            stop_k = w

        #safety check to exclude points that are obviously erronious.
        if (i > 1):#if analyzing any frame but the first frame
            #check if the y position of the flashlight is +/-70 percent the value of the previous posit
            false_y = (pos[1] < ((1 - error_thresh)*positions[i - 1, 1])) or (pos[1] > ((1 + error_thre
            #check if the x position of the flashlight is +/-70 percent the value of the previous posit
            false_x = (pos[0] < ((1 - error_thresh)*positions[i - 1, 0])) or (pos[0] > ((1 + error_thre
            if (false_y or false_x):# if the new calcluated position is outside this window +/- 70% for
                pos = np.array([positions[i - 1, 0], positions[i - 1, 1]])#rewrite the new position to

        positions[i,:] = pos#update array of paint can positions with the new position for the current
        new_movie[:,:,i] = gf
    return positions, new_movie
```

```
# In[ ]:


#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam1_1.mat')#load MATLAB .mat file into python as a dictionary
c1 = c_['vidFrames1_1']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam2_1.mat')#load MATLAB .mat file into python as a dictionary
c2 = c_['vidFrames2_1']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam3_1.mat')#load MATLAB .mat file into python as a dictionary
c3 = c_['vidFrames3_1']#raw camera data matrix

[w, h, rgb, tp] = c1.shape


# In[ ]:


#track the paint bucket for each camera
pc1, nm1 = bucketTracker(c1, 0.6, True, 10, 100, 0.70, 0.005, 0.3, 0.1, True)
pc2, nm2 = bucketTracker(c2, 0.7, True, 10, 100, 0.70, 0.0025, 0.3, 0.15, True)
pc3, nm3 = bucketTracker(c3, 0.6, True, 10, 100, 0.70, 0.005, 0.25, 0.1, True)


# In[ ]:


#show movie of first test with actual video
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(c3[:,:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc3[:, 0]
y = pc3[:, 1]

def animate(i):
    image.set_data(c3[:,:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:
```

```
#show movie of first test, with filtered movie frames
get_ipython().run_line_magic('matplotlib', 'notebook#necessary to show animations inline in jupyter note
f = plt.figure()#define figure
ax = f.gca()#define axes
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(nm3[:,:,0], animated=True)#define first animation frame
graph, = plt.plot([], [], 'bo')#define first animation graph of tracked position

x = pc3[:, 0]#x positional data
y = pc3[:, 1]#y positional data

def animate(i):
    image.set_data(nm3[:,:,i])#update animation video frame
    graph.set_data(x[i], y[i])#update position of tracked can
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)#animation
plt.show()


# In[ ]:


#plot positional data for each frame of the video for each camera in the x direction
get_ipython().run_line_magic('matplotlib', 'inline')
f, [ax1, ax2, ax3] = plt.subplots(3, 1, figsize=(10, 15))
ax1.plot(np.arange(0,226), pc1[:, 0])
ax1.set_title('Camera 1 X axis Oscillations')
ax1.set_ylabel('Amplitude [pixels]')
ax1.set_xlabel('Movie Frame')
ax2.plot(np.arange(0,284), pc2[:, 0])
ax2.set_title('Camera 2 X axis Oscillations')
ax2.set_ylabel('Amplitude [pixels]')
ax2.set_xlabel('Movie Frame')
ax3.plot(np.arange(0,232), pc3[:, 0])
ax3.set_title('Camera 3 X axis Oscillations')
ax3.set_ylabel('Amplitude [pixels]')
ax3.set_xlabel('Movie Frame')


# In[ ]:


#plot positional data for each frame of the video for each camera in the y direction
get_ipython().run_line_magic('matplotlib', 'inline')
f, [ax1, ax2, ax3] = plt.subplots(3, 1, figsize=(10, 15))
ax1.plot(np.arange(0,226), pc1[:, 1])
ax1.set_title('Camera 1 Y axis Oscillations')
ax1.set_ylabel('Amplitude [pixels]')
```

```
ax1.set_xlabel('Movie Frame')
ax2.plot(np.arange(0,284), pc2[:, 1])
ax2.set_title('Camera 2 Y axis Oscillations')
ax2.set_ylabel('Amplitude [pixels]')
ax2.set_xlabel('Movie Frame')
ax3.plot(np.arange(0,232), pc3[:, 1])
ax3.set_title('Camera 3 Y axis Oscillations')
ax3.set_ylabel('Amplitude [pixels]')
ax3.set_xlabel('Movie Frame')


# In[ ]:


#prepare data for principal component analysis through stadard scaling of each camera's positional data
camdf = pd.DataFrame(data=pc1)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_1 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc2)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_2 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc3)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_3 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])


# In[ ]:


get_ipython().run_line_magic('matplotlib', 'inline')
d = {'cam1x': pDf_1.loc[:, 'PC1'], 'cam1y': pDf_1.loc[:, 'PC2'], 'cam2x': pDf_2.loc[:, 'PC1'],
     'cam2y': pDf_2.loc[:, 'PC2'], 'cam3x': pDf_3.loc[:, 'PC1'], 'cam3y': pDf_3.loc[:, 'PC2']}

df = pd.DataFrame(data=d)
df = df.iloc[0:226]
dfs = StandardScaler().fit_transform(df)


pca = PCA(n_components=6)
principalComponents = pca.fit_transform(dfs)
principalDf = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2', 'PC3','PC4', 'PC5', 'PC

#Principle component oscillations
```

```
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(0,226), principalDf.loc[:,'PC1'], label='PC1')
ax1.plot(np.arange(0,226), principalDf.loc[:,'PC2'], label='PC2', alpha=0.35)
ax1.plot(np.arange(0,226), principalDf.loc[:,'PC3'], label='PC3', alpha=0.35)
ax1.set_title('Test 1 Total Camera Positional Data Principal Component Projections')
ax1.set_ylabel('Standardized Amplitude [pixels]')
ax1.set_xlabel('Movie Frame')
ax1.legend()


# In[ ]:


#Principle component oscillations
get_ipython().run_line_magic('matplotlib', 'inline')
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), 100*pca.explained_variance_/sum(pc
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
ax1.set_title('Test 1 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
ax1.set_xlim([0.5, 6.5])


# In[ ]:


#Test 2
#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam1_2.mat')#load MATLAB .mat file into python as a dictionary
c1 = c_['vidFrames1_2']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam2_2.mat')#load MATLAB .mat file into python as a dictionary
c2 = c_['vidFrames2_2']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam3_2.mat')#load MATLAB .mat file into python as a dictionary
c3 = c_['vidFrames3_2']#raw camera data matrix

[w, h, rgb, tp] = c1.shape


# In[ ]:


pc1, nm1 = bucketTracker(c1, 0.6, True, 10, 100, 0.70, 0.0025, 0.3, 0.15, True)
pc2, nm2 = bucketTracker(c2, 0.85, True, 10, 100, 0.70, 0.0015, 0.3, 0.15, True)
pc3, nm3 = bucketTracker(c3, 0.6, True, 10, 100, 0.70, 0.0025, 0.25, 0.2, True)


# In[ ]:
```

```
#show movie of second test, with filtered movie frames
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(nm3[:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc3[:, 0]
y = pc3[:, 1]

def animate(i):
    image.set_data(nm3[:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:


#show movie of second test with actual video
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(c3[:,:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc3[:, 0]
y = pc3[:, 1]

def animate(i):
    image.set_data(c3[:,:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:


#Run principal component analysis after stadard scaling on each camera's positional data to determine a
#maximal variance
```

```
get_ipython().run_line_magic('matplotlib', 'inline')
camdf = pd.DataFrame(data=pc1)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_1 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc2)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_2 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc3)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_3 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

#build dictionary of principal component data for each camera for analysis of all cameras together
d = {'cam1x': pDf_1.loc[:, 'PC1'], 'cam1y': pDf_1.loc[:, 'PC2'], 'cam2x': pDf_2.loc[:, 'PC1'],
     'cam2y': pDf_2.loc[:, 'PC2'], 'cam3x': pDf_3.loc[:, 'PC1'], 'cam3y': pDf_3.loc[:, 'PC2']}

df = pd.DataFrame(data=d)
df = df.iloc[0:314]#truncate data depending on the camera with the least number of recorded frames
dfs = StandardScaler().fit_transform(df)

#perform PCA on all the camera data together
pca = PCA(n_components=6)
principalComponents = pca.fit_transform(dfs)
principalDf = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2', 'PC3','PC4', 'PC5', 'PC

#Principle component oscillations
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(0,314), principalDf.loc[:,'PC1'], label='PC1')
ax1.plot(np.arange(0,314), principalDf.loc[:,'PC2'], label='PC2', alpha=0.35)
ax1.plot(np.arange(0,314), principalDf.loc[:,'PC3'], label='PC3', alpha=0.35)
ax1.set_title('Total Camera Positional Data Principal Component Projections')
ax1.set_ylabel('Standardized Amplitude [pixels]')
ax1.set_xlabel('Movie Frame')
ax1.legend()


# In[ ]:


#Principle component oscillations
get_ipython().run_line_magic('matplotlib', 'inline')
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), 100*pca.explained_variance_/sum(p
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
```

```
ax1.set_title('Test 2 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
ax1.set_xlim([0.5, 6.5])


# In[ ]:


#Test 3
#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam1_3.mat')#load MATLAB .mat file into python as a dictionary
c1 = c_['vidFrames1_3']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam2_3.mat')#load MATLAB .mat file into python as a dictionary
c2 = c_['vidFrames2_3']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam3_3.mat')#load MATLAB .mat file into python as a dictionary
c3 = c_['vidFrames3_3']#raw camera data matrix

[w, h, rgb, tp] = c1.shape


# In[ ]:


pc1, nm1 = bucketTracker(c1, 0.6, True, 10, 100, 0.70, 0.0025, 0.3, 0.15, True)
pc2, nm2 = bucketTracker(c2, 0.9, True, 10, 100, 0.70, 0.00075, 0.3, 0.15, True)
pc3, nm3 = bucketTracker(c3, 0.6, True, 10, 100, 0.70, 0.0025, 0.25, 0.2, True)


# In[ ]:


#show movie of third test, with filtered movie frames
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(nm3[:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc3[:, 0]
y = pc3[:, 1]

def animate(i):
    image.set_data(nm3[:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,
ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:


#show movie of second test with actual video
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(c1[:,:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc1[:, 0]
y = pc1[:, 1]

def animate(i):
    image.set_data(c1[:,:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:


get_ipython().run_line_magic('matplotlib', 'inline')
camdf = pd.DataFrame(data=pc1)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_1 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc2)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_2 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc3)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
```

```
pDf_3 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

d = {'cam1x': pDf_1.loc[:, 'PC1'], 'cam1y': pDf_1.loc[:, 'PC2'], 'cam2x': pDf_2.loc[:, 'PC1'],
     'cam2y': pDf_2.loc[:, 'PC2'], 'cam3x': pDf_3.loc[:, 'PC1'], 'cam3y': pDf_3.loc[:, 'PC2']}

df = pd.DataFrame(data=d)
df = df.iloc[0:237]
dfs = StandardScaler().fit_transform(df)


pca = PCA(n_components=6)
principalComponents = pca.fit_transform(dfs)
principalDf = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2', 'PC3','PC4', 'PC5', 'PC

#Principle component oscillations
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(0,237), principalDf.loc[:,'PC1'], label='PC1')
ax1.plot(np.arange(0,237), principalDf.loc[:,'PC2'], label='PC2')
ax1.plot(np.arange(0,237), principalDf.loc[:,'PC3'], label='PC3')
ax1.set_title('Total Camera Positional Data Principal Component Projections')
ax1.set_ylabel('Standardized Amplitude [pixels]')
ax1.set_xlabel('Movie Frame')
ax1.legend()


# In[ ]:


#Principle component oscillations
get_ipython().run_line_magic('matplotlib', 'inline')
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), 100*pca.explained_variance_/sum(p
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
ax1.set_title('Test 3 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
ax1.set_xlim([0.5, 6.5])


# In[ ]:


#Test 4
#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam1_4.mat')#load MATLAB .mat file into python as a dictionary
c1 = c_['vidFrames1_4']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam2_4.mat')#load MATLAB .mat file into python as a dictionary
c2 = c_['vidFrames2_4']#raw camera data matrix

#Data matrix contains camera pixel data RGB for spring mass movies
c_ = loadmat('cam3_4.mat')#load MATLAB .mat file into python as a dictionary
c3 = c_['vidFrames3_4']#raw camera data matrix
```

```
[w, h, rgb, tp] = c1.shape


# In[ ]:


pc1, nm1 = bucketTracker(c1, 0.6, True, 10, 100, 0.70, 0.0025, 0.3, 0.15, True)
pc2, nm2 = bucketTracker(c2, 0.9, True, 10, 100, 0.70, 0.00075, 0.3, 0.15, True)
pc3, nm3 = bucketTracker(c3, 0.6, True, 10, 100, 0.70, 0.0025, 0.25, 0.2, True)


# In[ ]:


#show movie of fourth test, with filtered movie frames
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(nm3[:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc3[:, 0]
y = pc3[:, 1]

def animate(i):
    image.set_data(nm3[:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:


#show movie of second test with actual video
get_ipython().run_line_magic('matplotlib', 'notebook')
f = plt.figure()
ax = f.gca()
ax.set_xlim(( 0, h))
ax.set_ylim((w, 0))

image = plt.imshow(c3[:,:,:,0], animated=True)
graph, = plt.plot([], [], 'bo')

x = pc3[:, 0]
y = pc3[:, 1]
```

```
def animate(i):
    image.set_data(c3[:,:,:,i])
    graph.set_data(x[i], y[i])
    ax.set_title(i)
    return image, graph,

ani = matplotlib.animation.FuncAnimation(f, animate, frames=tp, interval=200)
plt.show()


# In[ ]:


get_ipython().run_line_magic('matplotlib', 'inline')
camdf = pd.DataFrame(data=pc1)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_1 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc2)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_2 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

camdf = pd.DataFrame(data=pc3)
camdfs = StandardScaler().fit_transform(camdf)

pca = PCA(n_components=2)
principalComponents = pca.fit_transform(camdfs)
pDf_3 = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2'])

d = {'cam1x': pDf_1.loc[:, 'PC1'], 'cam1y': pDf_1.loc[:, 'PC2'], 'cam2x': pDf_2.loc[:, 'PC1'],
     'cam2y': pDf_2.loc[:, 'PC2'], 'cam3x': pDf_3.loc[:, 'PC1'], 'cam3y': pDf_3.loc[:, 'PC2']}

df = pd.DataFrame(data=d)
df = df.iloc[0:392]
dfs = StandardScaler().fit_transform(df)


pca = PCA(n_components=6)
principalComponents = pca.fit_transform(dfs)
principalDf = pd.DataFrame(data = principalComponents,
                           columns = ['PC1', 'PC2', 'PC3','PC4', 'PC5', 'PC6'])

#Principle component oscillations
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(0,392), principalDf.loc[:,'PC1'], label='PC1')
ax1.plot(np.arange(0,392), principalDf.loc[:,'PC2'], label='PC2')
ax1.plot(np.arange(0,392), principalDf.loc[:,'PC3'], label='PC3')
ax1.set_title('Total Camera Positional Data Principal Component Projections')
```

```
ax1.set_ylabel('Standardized Amplitude [pixels]')
ax1.set_xlabel('Movie Frame')
ax1.legend()


# In[ ]:


#Principle component oscillations
get_ipython().run_line_magic('matplotlib', 'inline')
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), 100*pca.explained_variance_/sum(p
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
ax1.set_title('Test 4 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
ax1.set_xlim([0.5, 6.5])
```