Object Tracking and Analysis Utilizing Singular Value Decomposition and Principle Component Analysis

Wyatt Scherer – *University of Washington*

AMATH 582 HW 3, Due 2/21/2020

GitHub Repository: https://github.com/wcscherer/DATA_ANALYSIS-AMATH582

ABSTRACT

This paper describes the development a simple algorithm for filtering video data for a moving object and determining that object's trajectory utilizing Principle Component Analysis (PCA). The analysis combines the video data of a paint can oscillating on a spring taken by three cameras. The can's location in the three videos is decomposed into the principle component axes of the can's motion utilizing the singular value decomposition algorithm. Four cases of can motion are analyzed: 1) vertical harmonic motion, 2) vertical harmonic motion with noise, 3) horizontal displacement, and 4) horizontal and rotational displacement.

1. Introduction

Tracking objects from video data and capturing their trajectory often proves difficult. Without knowing the all of the initial conditions and forces acting upon an object, an analytical solution to the objects motion is often impossible to accurately formulate. Even with large amount of data recording the object's location in space and time, the data may be of poor quality, the data may capture only a portion of the object's movement, different data sources may track movement along different axes, and other issues with data often arise. Principle Component Analysis (PCA) utilizing Singular Value Decomposition (SVD) can combine data of the same phenomenon from different sources and reduce that data to lower dimensional representations that reproduce the phenomenon with the least amount of components necessary. These lower dimensional representations, referred to as the Principle Components (PCs) or Proper Modes, are the minimum components necessary to reproduce the decomposed data.

To illustrate the use and properties of PCA utilizing SVD, the motion of a paint can tracked by three cameras will be analyzed. The mathematical representation of the can's motion is assumed to be unknown, so the PCs of the can's motion in each case will be used to describe the can's motion. The can is connected to a spring and held by an experimenter who displaces the can from equilibrium. The three cameras all capture the motion of the can from different angles and orientations. Since three cameras film the can's motion, some of the data will be redundant and only obscure the true motion of the can. Assuming no prior knowledge of the can's motion, no single camera data can be considered to be the best representation of the can's motion. Due to the large amount of redundant data and overall uncertainty of the true motion of the can, PCA is necessary to reduce the data to the PCs necessary to recreate the can's motion with the least amount of data. These PCs will be the best mathematical representation of the can's true motion. The can will be filmed in four separate cases: 1) vertical harmonic motion, 2) vertical harmonic motion with noise, 3) horizontal displacement, and 4) horizontal and rotational displacement.

To perform the PCA, first the videos of the can's motion must be filtered to record the can's location in each frame and the videos must be synchronized. Once the can's location in each frame is recorded, that recorded position data can be decomposed utilizing SVD to determine the PCs of the can's motion.

2. Technical Basis for PCA Utilizing SVD

To perform the PCA, first the video data must be filtered for the can's location in each frame. This can location data, recorded as (x_{pixel}, y_{pixel}) for each frame, is the data fed into the SVD algorithm for decomposition into the principle components.

2.1. Image Filtering and Object Tracking

To identify the can and record it's location in every frame, a naïve object tracking algorithm was developed for this analysis. All three cameras record the video data in frames of 480x640 RGB pixel resolution. The can tracked in each video has yellow paint on it and also has a flashlight attached to the top of the can. The bright light and yellow paint will be represented by a high pixel value when the image is

converted from RGB to grey-scale (pixel intensity from 0 to 255) – see the custom *greyscale* function described in the code in Appendix B.

Once the videos are converted to grey-scale, the bright pixels corresponding to the light and paint can be easily filtered for by searching for the index of pixels above a threshold (usually intensity > 245 or > 250 depending on video quality). These indices represent the x and y pixels with intensity above the threshold and these pixels should correspond to the can. There are frequently multiple pixels in each frame with intensities above the threshold. The can location in each frame is recorded as the average of the x index and y index of all pixels above the threshold. By averaging the location of the max pixels, it helps reduce the effect of recording pixels above the threshold that do not correspond to the light or paint on the can, as there should be an insignificant amount of pixels above the threshold in any frame above the threshold that are not on the can.

If there are no pixels above the threshold in a frame, the can's location is set to the center of the video frame. This modification makes the tracking algorithm naïve as it assumes the can has equal probability of being anywhere in the video frame, making the average of the can's location the center of the frame. While, naïve it is assumed that the can's true location is not captured for a small number of frames, making the effect of this averaging statistically insignificant with the large amount of data present. See the custom function *build_X* in Appendix B. This function outputs vectors **X** and **Y** corresponding to the can's x and y pixel location in each frame for each input camera feed.

To make filtering for the can's location easier, pixels that never capture the can's motion were set to zero using the custom functions *filter_imX and filter_imY* defined in Appendix B. These functions set all pixels along columns (X) and rows (Y) to zero along a range of pixels defined by the user. This helps reduce instances of recording pixels above the threshold that do not correspond to the can.

More robust tracking algorithms exist, such as *OpenCV* and MATLAB's *vision.PointTracker*. Utilizing either of these applications would have provided better can tracking data, but a goal of this paper is to illustrate the power of the SVD algorithm, even when operating on noisy data. Therefore, a naïve tracking algorithm is sufficient for exploring the properties of the SVD algorithm.

2.2. Singular Value Decomposition

Reducing large amounts of data into a smaller amount of data that faithfully represents the original data set simplifies analyzing complex processes. This is what Singular Value Decomposition (SVD) can do for large, seemingly uncorrelated data sets [1].

2.2.1. Mathematical Representation of SVD

Abstractly, SVD decomposes an input matrix transformation $\overleftrightarrow{\boldsymbol{A}}$ into three matrices **U, Σ,** and **V** that satisfy the following relationship [1]:

$$\boldsymbol{A} = \boldsymbol{U\Sigma V}^{*} \qquad (1)$$

Which can also be written as:

$$\boldsymbol{AV} = \boldsymbol{U\Sigma} \qquad (2)$$

As illustrated in Equations 1 and 2, the matrices **A, U**, **Σ,** and **V** have the following qualities:

$$\boldsymbol{A} \in \mathbb{R}^{mxn}, \quad (3)$$
$$\boldsymbol{U} \in \mathbb{C}^{mxm}\ is\ unitary \quad (4)$$
$$\boldsymbol{\Sigma} \in \mathbb{R}^{mxn}\ is\ diagonal, \quad (5)$$
$$\boldsymbol{V} \in \mathbb{C}^{nxn}\ is\ unitary \quad (6)$$

What Equation 1 effectively says is that, for a matrix **A**, there exists unitary transformation matrices **U**, and **V** that when combined with a diagonal scaling matrix **Σ,** recreate the matrix **A.** Writing Equation 2 as a transformation of **A** into a new matrix represented by **U** and **Σ.** This can be represented by Equation 7 [1]:

$$\boldsymbol{Av_j} = \sigma_j \boldsymbol{u}_j \; 1 \quad \leq j \leq n \quad (7)$$

Which when expanded into matrices looks like:

$$\left[\boldsymbol{A}\right][\boldsymbol{v_1} \quad \cdots \quad \boldsymbol{v_n}] = \left[\boldsymbol{u_1} \quad \cdots \quad \boldsymbol{u_n}\right]\begin{bmatrix}\sigma_1 & & \\ & \ddots & \\ & & \sigma_n\end{bmatrix} \quad (8)$$

From Equations 7 and 8, it can be reasoned that SVD decomposes a matrix **A** into principle singular values, represented by **Σ**, the unitary bases **U** of the components **Σ**, and the unitary transformation matrix **V** that transforms **A** along the basis **U** of the principle components **Σ.** The

theorem that formally states the existence of an SVD composition for **A** declares:

Theorem 1*: Every matrix* $\boldsymbol{A} \in \mathbb{C}^{mxn}$ *has a singular value decomposition. Furthermore, the singular values* σ_j *are uniquely determined, and if* $\boldsymbol{A}$ *is square and the* σ_j *distinct, the singular vectors* $\{\boldsymbol{u_j}\}$ *and* $\{\boldsymbol{v_j}\}$ *are uniquely determined up to complex coefficients.* [1]

Theorem 1 guarantees that a singular value decomposition exists for all $\boldsymbol{A} \in \mathbb{C}^{mxn}$, which means that a matrix of data can be decomposed into it's singular values and vectors. The principle components σ_j and their associated basis vectors $\boldsymbol{u_j}$ are what will be used to uncover the fundamental motion of the can in the video frames. Simply put, the SVD allows for diagonalization of any matrix **A** if the proper bases for the domain and range are used. This diagonalization transforms the underlying data in **A** along its preferred bases, which illuminates the processes generating the data in **A** [1].

2.2.2. Implementation of SVD

x The matrices **U**, **Σ,** and **V** are calculated from **A** as follows [1]:

$$\boldsymbol{A^T A} = (\boldsymbol{U\Sigma V^*})^{\boldsymbol{T}}(\boldsymbol{U\Sigma V^*}) = \boldsymbol{V\Sigma^2 V^*} \quad (9)$$

and

$$\boldsymbol{AA^T} = (\boldsymbol{U\Sigma V^*})(\boldsymbol{U\Sigma V^*})^{\boldsymbol{T}} = \boldsymbol{U\Sigma^2 U^*} \quad (10)$$

Multiplying Equation 9 by **V** and Equation 10 by **U** gives the following eigenvalue problems [1]:

$$\boldsymbol{A^T A} = \boldsymbol{V\Sigma^2} \quad (11)$$
$$\boldsymbol{AA^T} = \boldsymbol{U\Sigma^2} \quad (12)$$

Finding the normalized eigenvectors of Equations 11 and 12 produce the basis vectors **V** and **U**. The square root of the eigenvalues of Equations 11 and 12 produce the singular values σ_j of **A** [1].

2.2.3. Interpretation of SVD

Using the SVD to determine the principle components of a matrix requires defining **A** as a sum of rank one matrices:

$$A_N = \sum_{j=1}^{N} \sigma_j \boldsymbol{u}_j \boldsymbol{v}_j^* \ \ where\ 0 \le N \le rank(A) \quad (13)$$

From Eqn. 13, if N = min(m,n), set $\sigma_{N+1} = 0$ [1]. The 2-norm of $\mathbf{A}\text{-}\mathbf{A_N}$ can be defined as [1]:

$$\|\boldsymbol{A} - \boldsymbol{A_N}\|_2 = \sigma_{N+1} \quad (14)$$

And the Frobenius norm of $\mathbf{A}\text{-}\mathbf{A_N}$ can be defined as [1]:

$$\|\boldsymbol{A} - \boldsymbol{A_N}\|_N = \sqrt{\sigma_{N+1}^2 + \sigma_{N+2}^2 \dots \sigma_r^2} \quad (15)$$

where r is the rank of **A.**

Taking the ratio of the 2-norm of **A** and the Frobenius norm of **A** gives the 'energy' of that principle component. The 'energy' of the first principle component roughly translates to the proportion of the matrix **A** that can be represented by only using the projection of the first principle component σ_1[1]. If there are n total principle components and m < n, than the Frobenius norm of components σ_1 through σ_m divided by the Froebenius norm over all components represents the fraction of the total energy represented by components σ_1 through σ_m. Summing over all the components yields energy of 1, meaning the matrix **A** is completely represented. This interpretation demonstrates that the SVD offers a type of fitting algorithm that generates the minimum number of principle components necessary to recreate the decomposed matrix [1].

2.3. Implementation of SVD on Camera Data

The data from a single camera video is filtered into vectors of the indices x and y of the can's pixel location in each frame. What is desired from the vectors **x** and **y** is the *variance* of the location data. The reason for this is a high variance in location across a dimensions means there is a lot of movement (or noise) occurring along that dimension. For the can location, a high variance or covariance alone a dimension means the can is likely moving along that dimension. The variance of a distribution X with mean μ with n measurements is:

$$var(X) = \frac{1}{n-1}\sum_{j=1}^{n}\left(x_j - \mu\right)^2 \quad (16)$$

To compute the variance and also the covariance of the camera data, the data from each camera for each case is combined into a matrix **X**, for case N is defined as follows:

$$\boldsymbol{X}_N = \begin{bmatrix} x_{N1} \\ y_{N1} \\ x_{N2} \\ y_{N2} \\ x_{N3} \\ y_{N3} \end{bmatrix} \quad (17)$$

V

Where N is the case number and the following number is the camera number (3 in total). Here each of the position vectors have the mean of each position vector subtracted out. $\mathbf{X_N}$ has dimensions mxn where m is the number of cameras, and n is the number of frames represented by each vector. This allows for the generation of the covariance matrix C_x [1]:

$$C_X = \frac{1}{n-1}\boldsymbol{X}\boldsymbol{X}^{\boldsymbol{T}} \quad (18)$$

By performing SVD on the covariance matrix, the principle components showing will be the components that show the highest variance. These components are the principle components along which the can is moving. The SVD is performed by first defining $\mathbf{Y} = \mathbf{U}^*\mathbf{X}$ [1].

$$C_Y = \frac{1}{n-1}\boldsymbol{Y}\boldsymbol{Y}^{\boldsymbol{T}}$$
$$= \frac{1}{n-1}(\boldsymbol{U}^*\boldsymbol{X})(\boldsymbol{U}^*\boldsymbol{X})^T$$
$$C_Y = \frac{1}{n-1}\Sigma^2 \quad (19)$$

Therefore, to get the principle components of **X**, which determine the dimensions along which the can moves, is done by performing SVD on **X**.

3. Implementation of SVD Algorithm on Camera Data

Custom python functions used in this analysis are defined in Appendix B. Standard python package functions are briefly described in Appendix A.

4. Implementation of SVD Algorithm on Camera Data

The three cameras are oriented as follows for all four cases: Camera 1, the y pixel location is vertical displacement and the x pixel location is horizontal displacement; Camera 2, the y pixel location is vertical displacement and the x pixel location is horizontal displacement; Camera 3, the y pixel location is horizontal displacement and the x pixel location is vertical displacement.

4.1. Case 1: Simple Harmonic Motion

For case one, the can is subjected to simple harmonic motion along some unknown direction. There are 6 Principle Components since there are 6 input data vectors: 3 x location vectors and 3 y location vectors. The camera frames were synchronized to a common length of 226 frames and they were synchronized to the first peak of the camera 1 data. For case 1, **X** is 6x226 as defined by Equation 17. Figure 1 shows the raw and synchronized vertical displacement data from all three cameras.

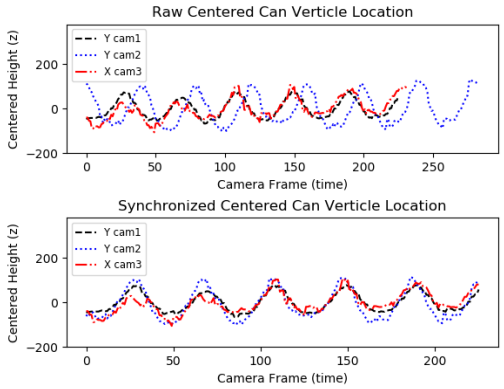

Figure 1: Raw and Synchronized Can Vertical Position for Case 1

With the camera data filtered for the can location and synchronized, SVD algorithm was performed on the transposed data in **X**. Figure 2 shows all 6 principle components (components numbered 0 -5). It also shows the first three principle bases from **V** and the first three transformation vectors **U**.

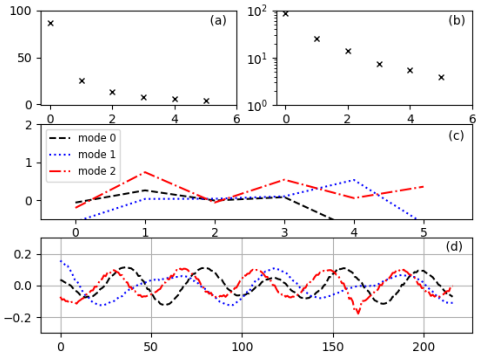

Figure 2: Case 1 Singular values on a standard plot (a) and on a log plot (b) showing the energy in each component. The first three linear component modes are illustrated in (c) and their time evolution behavior is illustrated in (d)

Note that for Case 1, almost all of the energy – approximately 94% is captured by the first mode. This suggests that the can appears to move principally along one single dimension in Case 1 – as expected.

4.2. Case 2: Noisy Harmonic Motion

For Case 2, the can is subjected to simple harmonic motion along some unknown direction with the addition of noisy by shaking the cameras. There are 6 Principle Components since there are 6 input data vectors: 3 x location vectors and 3 y location vectors. The camera frames were synchronized to a common length of 314 frames and they were synchronized to the first peak of the Camera 2 data. For case 1, **X** is 6x314 as defined by Equation 17. Figure 3 shows the raw and synchronized vertical displacement data from all three cameras.

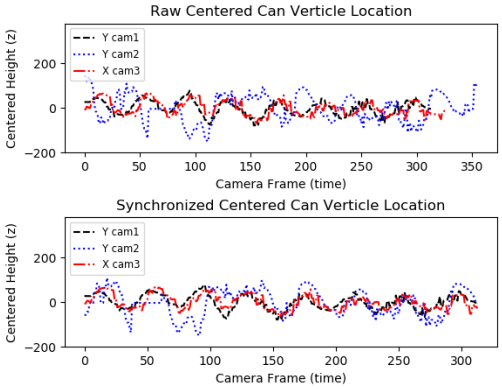

Figure 3: Raw and Synchronized Can Vertical Position for Case 2

With the camera data filtered for the can location and synchronized, SVD algorithm was performed on the transposed data in **X**. Notice that the data is significantly more noisy than Figure 1, this is to be expected from the shaking of the cameras. The synchronizing of peaks was therefore more difficult. Figure 4 shows all 6 principle components (components numbered 0 -5). It also shows the first three principle bases from **V** and the first three transformation vectors **U**.

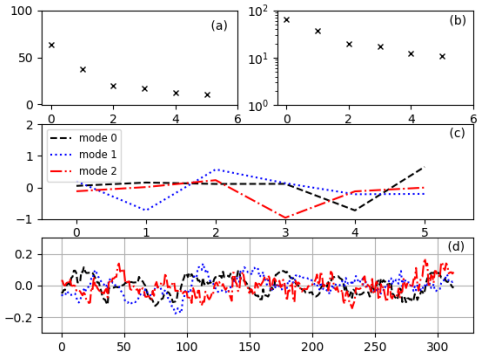

Figure 4: Case 2 Singular values on a standard plot (a) and on a log plot (b) showing the energy in each component. The first three linear component modes are illustrated in (c) and their time evolution behavior is illustrated in (d)

Note that for Case 2, almost all of the energy – approximately 92% is captured by the first two modes with the first mode capturing approximately 80% of the energy. This suggests that the can appears to move principally along two dimensions in Case 2 –this suggests horizontal and vertical movement.

4.3. Case 3: Horizontal Displacement

For Case 3, the can is subjected to simple harmonic motion along 2 unknown directions. There are 6 Principle Components since there are 6 input data vectors: 3 x location vectors and 3 y location vectors. The camera frames were synchronized to a common length of 217 frames and they were synchronized to the first peak of the Camera 3 data. For case 1, **X** is 6x217 as defined by Equation 17. Figure 5 shows the raw and synchronized horizontal displacement data from all three cameras.

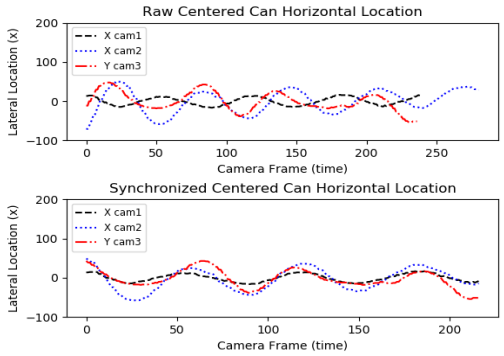

Figure 5: Raw and Synchronized Can Horizontal Position for Case 3

With the camera data filtered for the can location and synchronized, SVD algorithm was performed on the transposed data in **X**. Because the can is initially displaced horizontally, synchronizing the data is difficult. Figure 6 shows all 6 principle components (components numbered 0 -5). It also shows the first three principle bases from **V** and the first three transformation vectors **U**.

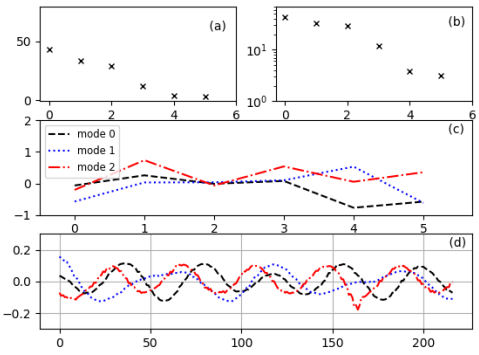

Figure 6: Case 3 Singular values on a standard plot (a) and on a log plot (b) showing the energy in each component. The first three linear component modes are illustrated in (c) and their time evolution behavior is illustrated in (d)

Note that for Case 3, it takes the first three components to capture 98% of the energy. The first two modes capture 86% of the energy. This suggests that the can appears to move principally along two dimensions with some additional displacement along a third dimension in Case 3. Also, Figure 6 (d) shows that the modes are not synchronized despite the camera frames being synchronized along their respective horizontal data – shown in Figure 5. This lack of synchronization could contribute to more components needed to represent the data.

4.4. Case 4: Horizontal and Rotational Displacement

For Case 4, the can is subjected to harmonic and rotational motion along 3 unknown directions. There are 6 Principle Components since there are 6 input data vectors: 3 x location vectors and 3 y location vectors. The camera frames were synchronized to a common length of 358 frames and they were synchronized to the first peak of the Camera 4 data. For case 1, **X** is 6x358 as defined by Equation 17. Figure 7 shows the raw and synchronized horizontal displacement data from all three cameras.

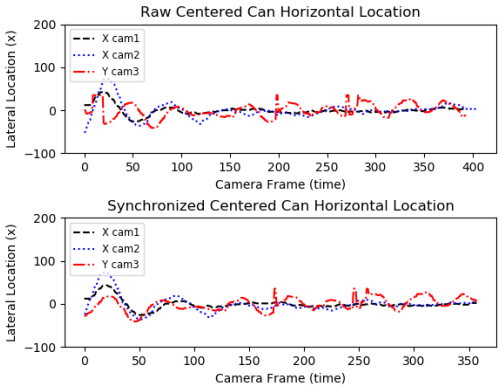

Figure 7: Raw and Synchronized Can Horizontal Position for Case 4

With the camera data filtered for the can location and synchronized, SVD algorithm was performed on the transposed data in **X**. Because the can is initially displaced horizontally, synchronizing the data is difficult. Figure 6 shows all 6 principle components (components numbered 0 -5). It also shows the first three principle bases from **V** and the first three transformation vectors **U**.

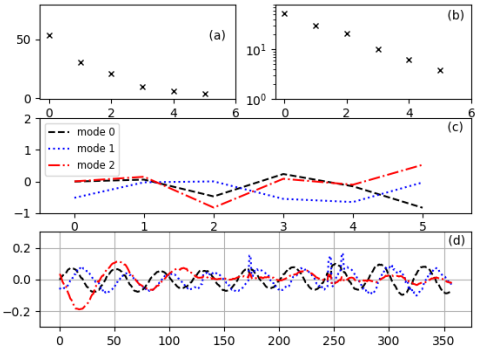

Figure 8: Case 4 Singular values on a standard plot (a) and on a log plot (b) showing the energy in each component. The first three linear component modes are illustrated in (c) and their time evolution behavior is illustrated in (d)

Note that for Case 4, it takes the first two components to capture 93% of the energy. The first three modes capture 98% of the energy. This suggests that the can appears to move principally along two dimensions with some additional displacement along a third dimension in Case 4. Also, Figure 8 (d) shows that the

modes are not synchronized despite the camera frames being synchronized along their respective horizontal data – shown in Figure 7. This lack of synchronization could contribute to more components needed to represent the data; however, since rotational displacement is observed the first two components could be capturing the x and y displacement of the can while the third mode captures the z displacement.

5. Conclusions of SVD Principle Component Analysis

Figure 9 plots the cumulative energy for each principle component for all 4 Cases. Note all 6 modes represent 100% of the energy for all 4 Cases, as expected.

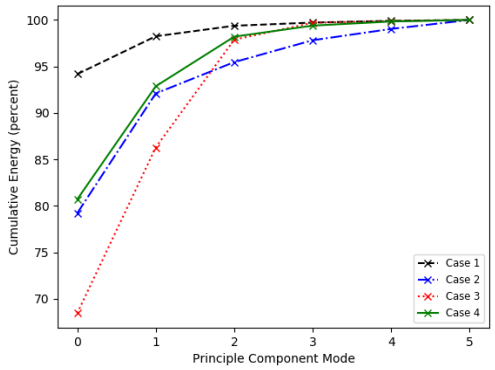

Figure 9: Cumulative energy of each Principle Component for all Four Cases

5.1. Case 1 Results

For Case 1, the almost all of the energy – approximately 94% is captured by the first mode. This suggests that the can appears to move principally along one single dimension in Case 1. This corresponds with visually inspecting the video data – the can moves only vertically along the z dimension.

5.2. Case 2 Results

For Case 2, approximately 92% of the energy is captured by the first two modes, with the first mode capturing approximately 80% of the energy. This suggests that the can appears to move principally along two dimensions in Case 2. This suggests the noise adds a perceived displacement along the horizontal dimension. A better object tracking package may be better at tracking the can in noisy data.

5.3. Case 3 Results

For Case 3, it takes the first three components to capture 98% of the energy. The first two modes capture 86% of the energy. This suggests that the can appears to move principally along two dimensions, likely horizontally and vertically with some additional displacement along a third lateral dimension in Case 3. This result is validated by viewing the camera data.

5.4. Case 4 Results

Note that for Case 4, it takes the first two components to capture 93% of the energy. The first three modes capture 98% of the energy. This suggests that the can appears to move principally along three dimensions, likely horizontally and laterally with some additional displacement along a third vertical dimension in Case 4. This result is validated by viewing the camera data.

6. References

1. "Ch. 15." *Data-Driven Modeling Et Scientific Computation: Methods for Complex Systems Et Big Data*, by J. Nathan. Kutz, Oxford Univ. Press, 2013.

APPENDIX A – Python/Numpy Functions Used

- *np.shape* returns the dimensions of a numpy array
- *loadmat* reads in MATLAB .m files
- *np.linalg.svd* implements the SVD algorithm on an input matrix and returns the three matrices defined in Section 2
- *np.vstack* stacks lists as rows into a numpy array
- *np.power* raises the elements of a numpy array to a power

APPENDIX B – Python Code

```
## AMATH 582 HW 3 - Principle Component Analysis; Wyatt Scherer, Due 2/21
import numpy as np
from scipy.io import loadmat
from matplotlib import pyplot as plt

def greyscale(inpt_npar):
    #import numpy as np
    """Converts RGB data in a numpy array into grey-scale where
    the numpy array is converted to an array of the minimum size
    for all arrays considered

    inpt_npar is a numpy array of pictures
    min_size is an integer of the minimum length to cut the number of pictures
    """
    grey_np = []

    length = np.shape(inpt_npar)[3]


    for i in range(int(length)):

        grey_np.append(np.dot(inpt_npar[:,:,:,i],[0.2989, 0.5870, 0.1140]))

    return(grey_np)

def setzerox(input_npar, xmin, xmax):
    #import numpy as np
    """Sets all pixels along x,y from x=0 to xmin and from xmax to end of the
    x domain in the picture format.

    input_npar is a numpy array of a single image
    xmin is the min column to set to zero
    xmax is the max column to set to zero
    """

    end = np.shape(input_npar)[1]

    for i in range(int(xmin)):

        input_npar[:,i] = 0*input_npar[:,i]

    for k in range(int(xmax),int(end)):

        input_npar[:,k] = 0*input_npar[:,k]

    return(input_npar)



def setzeroy(input_npar, ymin, ymax):
    #import numpy as np
    """Sets all pixels along x,y from y=0 to ymin and from ymax to end of the
    y domain in the picture format.

    input_npar is a numpy array of a single image
    ymin is the min row to set to zero
    ymax is the max row to set to zero
    """

    end = np.shape(input_npar)[0]

    for i in range(int(ymin)):

        input_npar[i] = 0*input_npar[i]

    for k in range(int(ymax),int(end)-1):
```

```
        input_npar[k] = 0*input_npar[k]

    return(input_npar)



def filter_imX(input_npar,minval,maxval):
    """Applys the setzero filter to array of image arrays

    input_array is the input numpy array list of all the grey image files
    minval is the minimum pixel value to filter up to - either x or y
    maxval is the maximum pixel value to filter through the end - x or y
    xory is a string that says to filter along the x or y dimension

    """
    #array = np.copy(input_npar)
    try:
        itr = np.shape(input_npar)[0]

        for i in range(itr):

            input_npar[i] = setzerox(input_npar[i],minval,maxval)

        return(input_npar)

    except:

        return ('ERROR')


def filter_imY(input_npar,minval,maxval):
    """Applys the setzero filter to array of image arrays

    input_array is the input numpy array list of all the grey image files
    minval is the minimum pixel value to filter up to - either x or y
    maxval is the maximum pixel value to filter through the end - x or y
    xory is a string that says to filter along the x or y dimension

    """
    #array = np.copy(input_npar)
    try:
        itr = np.shape(input_npar)[0]

        for i in range(itr):

            input_npar[i] = setzeroy(input_npar[i],minval,maxval)

        return(input_npar)

    except:

        return ('ERROR')



def build_X(input_npar, filtmin):
    """
    This function finds the max pixel value of an input greyscale filtered image
    and returns the max value and its locations in x, y pixels
    """

    itr =np.shape(input_npar)[0]

    Xavg = []
    Yavg = []

    for i in range(int(itr)):
```

```
        Y, X = np.where(input_npar[i] > filtmin)

#       Xavg.append(round(np.average(X)))
#       Yavg.append(round(np.average(Y)))

        xavg = np.average(X)
        yavg = np.average(Y)

        if (np.isnan(xavg)) or (np.isnan(yavg)):
            Xavg.append(np.trunc((680-1)/2))
            Yavg.append(np.trunc((480-1)/2))
        else:

            Xavg.append(round(xavg))
            Yavg.append(round(yavg))



    return(np.transpose(Xavg), np.transpose(Yavg))

def energy_s(s_mat):
    """
    Returns the energy of each component of the s matrix from SVD decomposition

    s is a list of singular values

    """

    itr = int(np.shape(s_mat)[0])

    fnorm = np.linalg.norm(s_mat)

    norm_2 = s_mat[0]/fnorm
    norm_e = []

    for i in range(itr):

        norm_e.append(np.linalg.norm(s_mat[0:i+1])/fnorm)

    return(norm_2, norm_e)


#%%
# Test Case 1 - simple harmonic motion along z dimension
# import the first data sets from the three cameras
c11 = loadmat('cam1_1.mat')
c21 = loadmat('cam2_1.mat')
c31 = loadmat('cam3_1.mat')

# camera data stored as rgb matrices for each video frame
c11_data = c11['vidFrames1_1']
c21_data = c21['vidFrames2_1']
c31_data = c31['vidFrames3_1']

dat_length = [c11_data.shape[3],c21_data.shape[3],c31_data.shape[3]]
min_l = min(dat_length)

# convert all rgb images to grescale images for easier analysis
c11_dgy = greyscale(c11_data)
c21_dgy = greyscale(c21_data)
c31_dgy = greyscale(c31_data)

# filter the images to show only pixels of the moving can
c11_filt = filter_imX(c11_dgy,300,450)
c21_filt = filter_imX(c21_dgy,250,350)
c31_filt = filter_imY(c31_dgy,235,330)

# create the X matrix to SVD
```

```
X1, Y1 = build_X(c11_filt,254)
X2, Y2 = build_X(c21_filt,252)
X3, Y3 = build_X(c31_filt,246)

X1 = X1 - np.average(X1)
Y1 = Y1 - np.average(Y1)

X2 = X2 - np.average(X2)
Y2 = Y2 - np.average(Y2)

X3 = X3 - np.average(X3)
Y3 = Y3 - np.average(Y3)

X = np.vstack((X1,Y1,X2[12:238],Y2[12:238],X3[:226],Y3[:226]))

# perform singular value decomposition on the Covariance matric Cx
Cx = np.cov(X)
u, s1, v = np.linalg.svd(np.transpose(X)/np.sqrt(min_l-1), full_matrices=False)

sig12 = np.power(s1,2)

fig1 = plt.figure(1)
plt.subplot(3,2,1)
plt.plot(range(int(len(s1))),s1,'kx',ms=4)
plt.ylim((-0.8,100))
plt.xlim((-0.3,6))
plt.text(5,85,' (a)')

plt.subplot(3,2,2)
plt.semilogy(range(int(len(s1))),s1,'kx', ms = 4)
plt.xlim((-0.3,6))
plt.ylim((1,100))
plt.text(5.1,50,' (b)')

plt.subplot(3,1,2)
plt.plot(range(np.shape(v)[0]),v[:,0],'k--',label = 'mode 0')
plt.plot(range(np.shape(v)[0]),v[:,1],'b:',label = 'mode 1')
plt.plot(range(np.shape(v)[0]),v[:,2],'r-.',label = 'mode 2')
plt.ylim((-0.5,2))
plt.xlim((-0.5,5.7))
plt.legend(loc='upper left', fontsize = 'small')
plt.text(5.3,1.6,' (c)')

plt.subplot(3,1,3)
plt.plot(range(np.shape(u)[0]),u[:,0],'k--')
plt.plot(range(np.shape(u)[0]),u[:,1],'b:')
plt.plot(range(np.shape(u)[0]),u[:,2],'r-.')
plt.ylim((-0.3,0.3))
plt.grid()
plt.text(210,0.22,' (d)')
plt.savefig('Case1_everything.png')
plt.close()


fig2 = plt.figure(2)
plt.subplot(2,1,1)
plt.plot(range(np.shape(Y1)[0]),Y1,'k--',label = 'Y cam1')
plt.plot(range(np.shape(Y2)[0]),Y2,'b:', label = 'Y cam2')
plt.plot(range(np.shape(X3)[0]),X3,'r-.', label = 'X cam3')
plt.title('Raw Centered Can Verticle Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-200,380))
plt.ylabel('Centered Height (z)')
plt.legend( loc='upper left', fontsize = 'small')

plt.subplot(2,1,2)
plt.plot(range(np.shape(Y1)[0]),Y1,'k--',label = 'Y cam1')
plt.plot(range(np.shape(Y2[12:238])[0]),Y2[12:238],'b:', label = 'Y cam2')
plt.plot(range(np.shape(X3[:226])[0]),X3[:226],'r-.', label = 'X cam3')
```

```
plt.title('Synchronized Centered Can Verticle Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-200,380))
plt.ylabel('Centered Height (z)')
plt.legend( loc='upper left', fontsize = 'small')
plt.subplots_adjust(hspace=0.5)
plt.savefig('Case1_ydat.png')
plt.close()


#%%
# Test Case 2 - harmonic motion along z dimension with noisy data
# import the first data sets from the three cameras

c12 = loadmat('cam1_2.mat')
c22 = loadmat('cam2_2.mat')
c32 = loadmat('cam3_2.mat')

# camera data stored as rgb matrices for each video frame
c12_data = c12['vidFrames1_2']
c22_data = c22['vidFrames2_2']
c32_data = c32['vidFrames3_2']

dat_length = [c12_data.shape[3],c22_data.shape[3],c32_data.shape[3]]
min_l = min(dat_length)

# convert all rgb images to grescale images for easier analysis
c12_dgy = greyscale(c12_data)
c22_dgy = greyscale(c22_data)
c32_dgy = greyscale(c32_data)

# filter the images to show only pixels of the moving can
c12_filt = filter_imX(c12_dgy,300,390)
c22_filt = filter_imX(c22_dgy,275,370)
c32_filt = filter_imY(c32_dgy,220,300)

# create the X matrix to SVD
X12, Y12 = build_X(c12_filt,251)
X22, Y22 = build_X(c22_filt,250)
X32, Y32 = build_X(c32_filt,246)

X12 = X12 - np.average(X12)
Y12 = Y12 - np.average(Y12)

X22 = X22 - np.average(X22)
Y22 = Y22 - np.average(Y22)

X32 = X32 - np.average(X32)
Y32 = Y32 - np.average(Y32)

X = np.vstack((X12,Y12,X22[20:334],Y22[20:334],X32[:min_l],Y32[:min_l]))

# perform singular value decomposition on the Covariance matric Cx
Cx = np.cov(X)
u, s2, v = np.linalg.svd(np.transpose(X)/np.sqrt(min_l-1))

sig2 = np.power(s2,2)

fig1 = plt.figure(1)
plt.subplot(3,2,1)
plt.plot(range(int(len(s2))),s2,'kx',ms=4)
plt.ylim((-0.8,100))
plt.xlim((-0.3,6))
plt.text(5,80,' (a)')

plt.subplot(3,2,2)
plt.semilogy(range(int(len(s2))),s2,'kx', ms = 4)
plt.xlim((-0.3,6))
plt.ylim((1,100))
```

```
plt.text(5.1,50,' (b)')

plt.subplot(3,1,2)
plt.plot(range(np.shape(v)[0]),v[:,0],'k--',label = 'mode 0')
plt.plot(range(np.shape(v)[0]),v[:,1],'b:',label = 'mode 1')
plt.plot(range(np.shape(v)[0]),v[:,2],'r-.',label = 'mode 2')
plt.ylim((-1,2))
plt.xlim((-0.5,5.7))
plt.legend(loc='upper left', fontsize = 'small')
plt.text(5.3,1.6,' (c)')

plt.subplot(3,1,3)
plt.plot(range(np.shape(u)[0]),u[:,0],'k--')
plt.plot(range(np.shape(u)[0]),u[:,1],'b:')
plt.plot(range(np.shape(u)[0]),u[:,2],'r-.')
plt.ylim((-0.3,0.3))
plt.grid()
plt.text(305,0.22,' (d)')
plt.savefig('Case2_everything.png')
plt.close()


fig2 = plt.figure(2)
plt.subplot(2,1,1)
plt.plot(range(np.shape(Y12)[0]),Y12,'k--',label = 'Y cam1')
plt.plot(range(np.shape(Y22)[0]),Y22,'b:', label = 'Y cam2')
plt.plot(range(np.shape(X32)[0]),X32,'r-.', label = 'X cam3')
plt.title('Raw Centered Can Verticle Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-200,380))
plt.ylabel('Centered Height (z)')
plt.legend( loc='upper left', fontsize = 'small')

plt.subplot(2,1,2)
plt.plot(range(np.shape(Y12)[0]),Y12,'k--',label = 'Y cam1')
plt.plot(range(np.shape(Y22[20:334])[0]),Y22[20:334],'b:', label = 'Y cam2')
plt.plot(range(np.shape(X32[:min_l])[0]),X32[:min_l],'r-.', label = 'X cam3')
plt.title('Synchronized Centered Can Verticle Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-200,380))
plt.ylabel('Centered Height (z)')
plt.legend( loc='upper left', fontsize = 'small')
plt.subplots_adjust(hspace=0.5)
plt.savefig('Case2_ydat.png')
plt.close()


#%%
# Test Case 3 - harmonic motion along z with horizontal displacement
# import the first data sets from the three cameras

c13 = loadmat('cam1_3.mat')
c23 = loadmat('cam2_3.mat')
c33 = loadmat('cam3_3.mat')

# camera data stored as rgb matrices for each video frame
c13_data = c13['vidFrames1_3']
c23_data = c23['vidFrames2_3']
c33_data = c33['vidFrames3_3']

dat_length = [c13_data.shape[3],c23_data.shape[3],c33_data.shape[3]]
min_l = min(dat_length)

# convert all rgb images to grescale images for easier analysis
c13_dgy = greyscale(c13_data)
c23_dgy = greyscale(c23_data)
c33_dgy = greyscale(c33_data)

# filter the images to show only pixels of the moving can
```

```
c13_filt = filter_imX(c13_dgy,270,380)
c23_filt = filter_imX(c23_dgy,220,410)
c33_filt = filter_imY(c33_dgy,180,300)

# create the X matrix to SVD
X13, Y13 = build_X(c13_filt,254)
X23, Y23 = build_X(c23_filt,250)
X33, Y33 = build_X(c33_filt,248)

X13 = X13 - np.average(X13)
Y13 = Y13 - np.average(Y13)

X23 = X23 - np.average(X23)
Y23 = Y23 - np.average(Y23)

X33 = X33 - np.average(X33)
Y33 = Y33 - np.average(Y33)
#
X = np.vstack((X13[:217],Y13[:217],X23[26:243],Y23[26:243],X33[20:],Y33[20:]))
#
## perform singular value decomposition on the Covariance matric Cx
#Cx = np.cov(X)
u, s3, v = np.linalg.svd(np.transpose(X)/np.sqrt(min_l-1))
#

sig3 = np.power(s3,2)

fig1 = plt.figure(1)
plt.subplot(3,2,1)
plt.plot(range(int(len(s3))),s3,'kx',ms=4)
plt.ylim((-0.8,80))
plt.xlim((-0.3,6))
plt.text(5,60,' (a)')

plt.subplot(3,2,2)
plt.semilogy(range(int(len(s3))),s3,'kx', ms = 4)
plt.xlim((-0.3,6))
plt.ylim((1,70))
plt.text(5.1,30,' (b)')

plt.subplot(3,1,2)
plt.plot(range(np.shape(v)[0]),v[:,0],'k--',label = 'mode 0')
plt.plot(range(np.shape(v)[0]),v[:,1],'b:',label = 'mode 1')
plt.plot(range(np.shape(v)[0]),v[:,2],'r-.',label = 'mode 2')
plt.ylim((-1,2))
plt.xlim((-0.5,5.7))
plt.legend(loc='upper left', fontsize = 'small')
plt.text(5.3,1.6,' (c)')

plt.subplot(3,1,3)
plt.plot(range(np.shape(u)[0]),u[:,0],'k--')
plt.plot(range(np.shape(u)[0]),u[:,1],'b:')
plt.plot(range(np.shape(u)[0]),u[:,2],'r-.')
plt.ylim((-0.3,0.3))
plt.grid()
plt.text(212,0.22,' (d)')
plt.savefig('Case3_everything.png')
plt.close()


fig2 = plt.figure(2)
plt.subplot(2,1,1)
plt.plot(range(np.shape(X13)[0]),X13,'k--',label = 'X cam1')
plt.plot(range(np.shape(X23)[0]),X23,'b:', label = 'X cam2')
plt.plot(range(np.shape(Y33)[0]),Y33,'r-.', label = 'Y cam3')
plt.title('Raw Centered Can Horizontal Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-100,200))
plt.ylabel('Lateral Location (x)')
```

```
plt.legend( loc='upper left', fontsize = 'small')

plt.subplot(2,1,2)
plt.plot(range(np.shape(X13[:217])[0]),X13[:217],'k--',label = 'X cam1')
plt.plot(range(np.shape(X23[26:243])[0]),X23[26:243],'b:', label = 'X cam2')
plt.plot(range(np.shape(Y33[20:])[0]),Y33[20:],'r-.', label = 'Y cam3')
plt.title('Synchronized Centered Can Horizontal Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-100,200))
plt.ylabel('Lateral Location (x)')
plt.legend( loc='upper left', fontsize = 'small')
plt.subplots_adjust(hspace=0.5)
plt.savefig('Case3_ydat.png')
plt.close()


#%%
# Test Case 4 - harmonic motion along z with circular motion in xy
# import the first data sets from the three cameras

c14 = loadmat('cam1_4.mat')
c24 = loadmat('cam2_4.mat')
c34 = loadmat('cam3_4.mat')

# camera data stored as rgb matrices for each video frame
c14_data = c14['vidFrames1_4']
c24_data = c24['vidFrames2_4']
c34_data = c34['vidFrames3_4']

dat_length = [c14_data.shape[3],c24_data.shape[3],c34_data.shape[3]]
min_l = min(dat_length)
startn1 = 0
startn2 = 40
# convert all rgb images to grescale images for easier analysis
c14_dgy = greyscale(c14_data)
c24_dgy = greyscale(c24_data)
c34_dgy = greyscale(c34_data)

# filter the images to show only pixels of the moving can
c14_filt = filter_imX(c14_dgy,320,460)
c24_filt = filter_imX(c24_dgy,210,410)
c34_filt = filter_imY(c34_dgy,150,360)
c34_filt = filter_imX(c34_filt,200,600)

# create the X matrix to SVD
X14, Y14 = build_X(c14_filt,245)
X24, Y24 = build_X(c24_filt,245)
X34, Y34 = build_X(c34_filt,240)

X14 = X14 - np.average(X14)
Y14 = Y14 - np.average(Y14)

X24 = X24 - np.average(X24)
Y24 = Y24 - np.average(Y24)

X34 = X34 - np.average(X34)
Y34 = Y34 - np.average(Y34)

X = np.vstack((X14[:358],Y14[:358],X24[5:363],Y24[5:363],X34[25:383],Y34[25:383]))
#
## perform singular value decomposition on the Covariance matric Cx
#Cx = np.cov(X)
u, s4, v = np.linalg.svd(np.transpose(X)/np.sqrt(min_l-1))
#

sig4 = np.power(s4,2)

fig1 = plt.figure(1)
plt.subplot(3,2,1)
```

```
plt.plot(range(int(len(s4))),s4,'kx',ms=4)
plt.ylim((-0.8,80))
plt.xlim((-0.3,6))
plt.text(5,50,' (a)')

plt.subplot(3,2,2)
plt.semilogy(range(int(len(s4))),s4,'kx', ms = 4)
plt.xlim((-0.3,6))
plt.ylim((1,80))
plt.text(5.1,40,' (b)')

plt.subplot(3,1,2)
plt.plot(range(np.shape(v)[0]),v[:,0],'k--',label = 'mode 0')
plt.plot(range(np.shape(v)[0]),v[:,1],'b:',label = 'mode 1')
plt.plot(range(np.shape(v)[0]),v[:,2],'r-.',label = 'mode 2')
plt.ylim((-1,2))
plt.xlim((-0.5,5.7))
plt.legend(loc='upper left', fontsize = 'small')
plt.text(5.3,1.6,' (c)')

plt.subplot(3,1,3)
plt.plot(range(np.shape(u)[0]),u[:,0],'k--')
plt.plot(range(np.shape(u)[0]),u[:,1],'b:')
plt.plot(range(np.shape(u)[0]),u[:,2],'r-.')
plt.ylim((-0.3,0.3))
plt.grid()
plt.text(350,0.22,' (d)')
plt.savefig('Case4_everything.png')
plt.close()


fig2 = plt.figure(2)
plt.subplot(2,1,1)
plt.plot(range(np.shape(X14)[0]),X14,'k--',label = 'X cam1')
plt.plot(range(np.shape(X24)[0]),X24,'b:', label = 'X cam2')
plt.plot(range(np.shape(Y34)[0]),Y34,'r-.', label = 'Y cam3')
plt.title('Raw Centered Can Horizontal Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-100,200))
plt.ylabel('Lateral Location (x)')
plt.legend( loc='upper left', fontsize = 'small')

plt.subplot(2,1,2)
plt.plot(range(np.shape(X14[:358])[0]),X14[:358],'k--',label = 'X cam1')
plt.plot(range(np.shape(X24[5:363])[0]),X24[5:363],'b:', label = 'X cam2')
plt.plot(range(np.shape(Y34[25:383])[0]),Y34[25:383],'r-.', label = 'Y cam3')
plt.title('Synchronized Centered Can Horizontal Location')
plt.xlabel('Camera Frame (time)')
plt.ylim((-100,200))
plt.ylabel('Lateral Location (x)')
plt.legend( loc='upper left', fontsize = 'small')
plt.subplots_adjust(hspace=0.5)
plt.savefig('Case4_ydat.png')
plt.close()

#%%
# Final comparison of the four cases - comparing the energy

s1_n2, s1_ne = energy_s(s1)
s2_n2, s2_ne = energy_s(s2)
s3_n2, s3_ne = energy_s(s3)
s4_n2, s4_ne = energy_s(s4)

fig5 = plt.figure(5)
plt.plot(range(6),np.multiply(100,s1_ne),'kx--',label = 'Case 1')
plt.plot(range(6),np.multiply(100,s2_ne),'bx-.',label = 'Case 2')
plt.plot(range(6),np.multiply(100,s3_ne),'rx:', label = 'Case 3')
plt.plot(range(6),np.multiply(100,s4_ne),'gx-', label = 'Case 4')
plt.legend(loc = 'lower right', fontsize = 'small')
```

```
plt.xlabel('Principle Component Mode')
plt.ylabel('Cumulative Energy (percent)')
plt.savefig('comp_adat.png')
plt.close()
```