Amath 582 Homework 4: PCA

Due: February 20, 2016

Brian de Silva[1]

Abstract
In this homework writeup we use the Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) to remove redundant information and extract information about the movement of a mass-spring system from a series of video recordings. We test the performance of PCA on increasingly complex movements and on noisy measurements.

[1] *Department of Applied Mathematics, University of Washington, Seattle*

Contents

1. Introduction

The purpose of this assignment is to practice cleaning up messy data and to better understand Principle Component Analysis (PCA) and the Singular Vale Decomposition (SVD). Prior to the course Nathan set up a classic physics system: a mass suspended from a spring. He then gave the mass four different initial pushes and had three people record the resulting motion from various angles. Furthermore during certain runs he had them introduce artificial noise by shaking their cameras. In the first instance he pushes the mass straight down and instructs the camera-holders to keep their instruments still. In the second he again pushes the mass straight down, but this time he has the recorders shake their cameras while they are filming, creating noise. Next he shoves the weight in such a way that it exhibits vertical and horizontal movement. Finally he pushes it so that it moves vertically and horizontally and also rotates.

We have been given the twelve video files produced by observers and it is our objective to use them analyze the movement of the weight using PCA. We must first extract the positions of the weight throughout the videos. Then, using PCA, we aim to filter some noise from the observed mass locations, determine the best basis in which to represent the movement, and remove redundancy from the measurements.

This writeup is structured as follows: in Section 2 we give a brief review of the theory behind our techniques. Next we discuss our algorithm and its implementation in MATLAB in Section 3 before discussing our numerical results in Section 4. Finally we make some closing remarks in Section 5.

2. Theoretical Background

2.1 The Singular Value Decomposition (SVD)

Let us interpret the transformation of a vector $\mathbf{x}$ by a matrix $\mathbf{A}$ (by left-multiplication) geometrically. This transformation can always be decomposed into a series of rotations, scalings, then more rotations, all in various orthogonal directions. *The Singular Value Decomposition* of $\mathbf{A}$ uncovers these directions and the scaling factors associated with the rescalings. More concretely, the SVD of $\mathbf{A}$ is

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

If $\mathbf{A}$ is an $n \times m$ matrix then $\mathbf{U} \in \mathbb{C}^{m\times m}$ and $\mathbf{V} \in \mathbb{C}^{n\times n}$ are both unitary and $\mathbf{\Sigma} \in \mathbb{R}^{n\times m}$ is diagonal with nonnegative entries on the diagonal (and are assumed to be in nonincreasing order). From this factorization of $\mathbf{A}$ we see that its action on a vector is to first rotate it via left multiplication by the unitary matrix $\mathbf{V}^*$, then to rescale it through multiplication by $\mathbf{\Sigma}$, and finally to rotate it again by multiplication by the unitary matrix $\mathbf{U}$. There are a number of theorems regarding the SVD, most of which we will not list here. However, among the most important of them are the following:

Theorem 1 *Every matrix has a Singular Value Decomposition.*

Theorem 2 *For any N so that $0 \leq N \leq r$, we can define the partial sum*

$$\mathbf{A}_N = \sum_{j=1}^{N} \sigma_j \mathbf{u_j} \mathbf{v_j}^*. \tag{2}$$

And if $N = \min\{m,n\}$, define $\sigma_{N+1} = 0$. Then

$$\|\mathbf{A} - \mathbf{A}_N\|_2 = \sigma_{N+1}. \tag{3}$$

Likewise, if using the Frobenius norm, then

$$\|\mathbf{A} - \mathbf{A}_N\|_F = \sqrt{\sigma_{N+1}^2 + \sigma_{N+2}^2 + \cdots + \sigma_r^2}. \tag{4}$$

Here $\mathbf{u_i}$ and $\mathbf{v_j}$ are the i-th and j-th columns of $\mathbf{U}$ and $\mathbf{V}$, respectively, σ_j is the j-th diagonal entry of $\mathbf{\Sigma}$ (the j-th singular value of $\mathbf{A}$), and $r = \text{Rank}(\mathbf{A})$.

2.2 Principal Component Analysis (PCA)

Suppose we have a set of m sensors which each takes n measurements which we store in row vectors $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m}$, each of length n (so the measurements from the i-th sensor are stored in $\mathbf{x_i}$). Assume that each set of measurements has mean zero. We can stack all these measurements into an $m \times n$ matrix

$$\mathbf{X} = \begin{pmatrix} -\mathbf{x_1}- \\ -\mathbf{x_2}- \\ \vdots \\ -\mathbf{x_m}- \end{pmatrix}. \tag{5}$$

We can determine how much redundancy exists in two sets of measurements $\mathbf{x_i}$ and $\mathbf{x_j}$ by computing their covariance

$$\sigma_{ij}^2 = \frac{1}{n-1} \mathbf{x_i} \mathbf{x_j}^T. \tag{6}$$

A large covariance σ_{ij}^2 indicates that $\mathbf{x_i}$ and $\mathbf{x_j}$ are highly correlated, and therefore that there is a large amount of redundancy between them, whereas a low covariance is indicative of a small amount of redundancy. The variance of a set of measurements

$$\sigma_i^2 = \frac{1}{n-1} \mathbf{x_i} \mathbf{x_i}^T = \frac{1}{n-1} \|\mathbf{x_i}\|_2^2. \tag{7}$$

A larger variance (for a mean zero measurement vector) means that it is likely that interesting dynamics are being captured by the sensor. Indeed we operate under this assumption in this assignment. One can construct a matrix summarizing all the pairwise covariances and variances of the data vectors. This is referred to as the *covariance matrix* and is defined as follows

$$\mathbf{C_X} = \frac{1}{n-1} \mathbf{X} \mathbf{X}^T. \tag{8}$$

It is not too hard to show that $\mathbf{C_X}_{ij} = \sigma_{ij}^2$. If all the sensors captured independent interesting dynamics then $\mathbf{C_X}$ would have large diagonal entries and very small off-diagonal entries (because of the low redundancy between measurements). Often this will not be the case and multiple sensors will pick up on the same dynamics, leading to redundancy, or they will observe unimportant information. The goal of PCA is to find a basis in which $\mathbf{C_X}$ is diagonal. If this can be accomplished then one can easily extract the dominant dynamics and the best directions in which to represent the data.

The approach we use to diagonalize the covariance matrix is the SVD. Suppose $\mathbf{X}$ has the SVD $\mathbf{X} = \mathbf{U\Sigma V}^*$. If we then project the data onto the left singular values and call the result $\mathbf{Y}$ (so $\mathbf{Y} = \mathbf{U}^*\mathbf{X}$) then the covariance matrix of $\mathbf{Y}$ is given by

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

Thus the basis formed by the left singular vectors of $\mathbf{X}$ is the ideal one in which to express the data stored in $\mathbf{X}$ in the sense that this representation leads to a diagonal covariance matrix. To determine the directions along which the imporant dynamics occur, we need only find the largest diagonal entries of $\mathbf{\Sigma}$. Typically, if redundancy is present in the measurements, $\mathbf{X}$ will have only a few large singular values, and the rest will be miniscule in comparison. Since every matrix has a SVD, this computation may always be carried out.

The information presented in this section was taken from the course textbook and [1].

3. Algorithms Implementation and Development

We used two scripts to accomplish the major goals of this assignment. The first extracted the positions of the bucket throughout the videos and the second applied PCA to these positions. Both are outlined below.

3.1 Extracting Bucket Locations

The main technique we used to find the bucket positions was simply to follow the flashlight placed on top of the bucket. The pixels showing this flashlight were much whiter than the nearby pixels, allowing us to track its movement frame-by-frame. There were, however, some regions of the videos which had pixels that were sometimes whiter than that of the flashlight. To deal with this obstacle in each frame of the videos we simply searched for white pixels only within a small window of the previous flashlight location. This also has the advantage on cutting down on the computational cost of the algorithm, since we only need consider tiny subsets of

the sizeable frames. For the method to work we are required to provide the position of the flashlight in the initial frame. It should be noted that in the fourth test, where the bucket is also spinning, the flashlight is only visible about a third of the time making this technique infeasible. Fortunately, the bucket has a large white section wrapping around its exterior. In the fourth test we simply follow this stripe instead of the flashlight as it is always visible. The stripe occupies a larger area in the videos than the light does and the detected location jumps around somewhat within this region which adds some noise to the extracted positions. But this is an acceptable price to pay considering the alternative of only being able to automatically locate the bucket a third of the time.

In particular we carried out the following steps for each of the sets of three recordings.

1. Locate the starting point of the bucket

To accomplish this we use a simple GUI which allows one to feed in the coordinates of the flashlight by clicking on it in a displayed version of the first frame. For the fourth test we click the white stripe instead.

2. Track the movement of the bucket over the remaining frames

As mentioned before, to find the position of the flashlight in a given frame we search a small (rectangular) window around its previous location for the whitest point. Unfortunately we must customize the size of the window for each video since in each the bucket moves at different rates and the flashlight varies in intensity. One must make the window large enough to account for the movement of the bucket between frames, but not so large that another white point is mistaken for the flashlight. We also check that our window does not extend outside of the image, and if it does we limit its size accordingly. To ensure we did a satisfactory job of following the bucket we watch the video with the detected location superimposed over it.

3. Save the measurements

We store the detected positions in .mat files for the second script to access. This way we do not have to run the algorithm every time we want access to the data.

3.2 Principle Component Analysis

Once the approximate bucket locations are known relative to each camera it is fairly straightforward to apply Principle Component Analysis to rid all these measurements of redundancy. Our main tool for accomplishing this is the SVD, but first we must clean up the data a bit. Since the videos were filmed at various angles the scales of the measured positions vary somewhat between cameras. The cameras were also not all started and stopped at the same instants in time so some videos include more frames than others. Recall that in order to use the SVD we need a set of observation vectors which all have the same length.

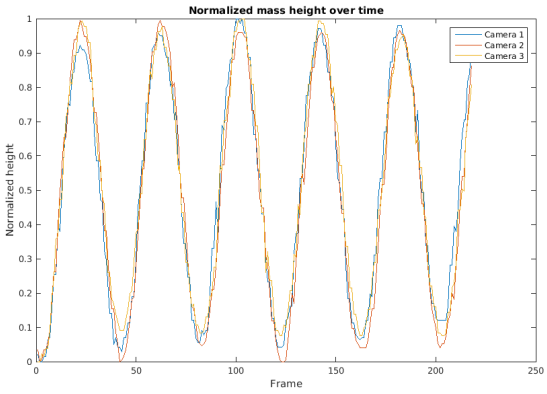

Figure 1. The normalized height of the weight over time as measured by the three cameras in test 1

1. Normalize the positions so that the minimum height is 0 and the maximum is 1

We carry out this step for each video stream, in both the horizontal and vertical directions separately.

2. Align the position vectors so that each starts at the same time

For each test we find the data stream which begins at the latest time (manually, by inspection of the measured positions) and remove some entries from the other position vectors so that they all start at approximately the same time.

3. Enforce that every position vector be the same length

We find which camera stops recording first and remove entries in the position vectors corresponding to measurements taken after it has stopped. Once the number of recorded positions is known we also normalize time so that the first set of measurements occur at $t = 0$ and the final set occurs at $t = 1$.

4. Stack the measurements in a data matrix X and apply Principal Component Analysis

In order to prepare X for PCA we first set the mean and variance of each of its rows to 0 and 1, respectively. We then take the SVD of X and analyze the matrices involved in the decomposition.

4. Computational Results

4.1 Test 1: Ideal case

For the first test there was very little noise in the data and the flashlight was in plain view for the entirety of all three videos so we were able to accurately extract the bucket's position with our automated system. Figure 1 shows the measured height of the bucket for each of the three cameras. This is after the height and time have both been normalized to lie in $[0, 1]$. Apart from some minor variations in amplitude over different periods, the three cameras have captured essentially the same signal, i.e. there is a lot of redundancy in the data.

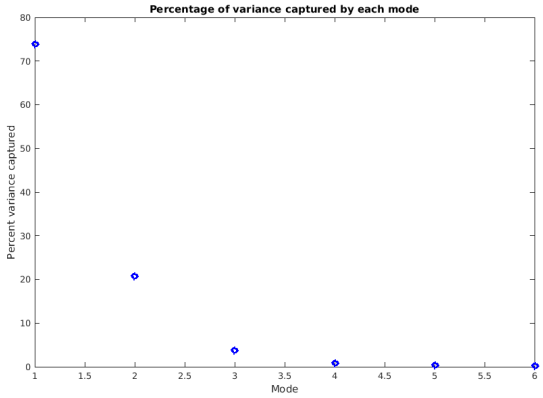

Figure 2. The percentage of the variance captured by the modes computed by the SVD in test 1

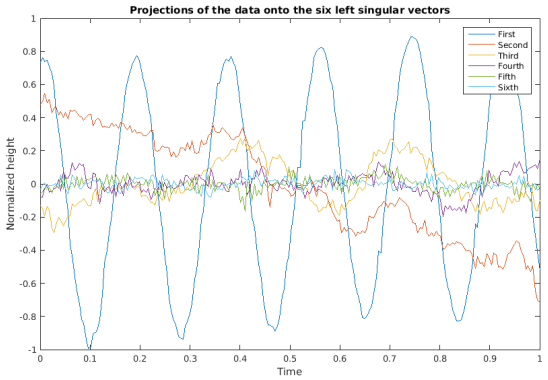

Figure 3. The projections of the data onto all six left singular vectors computed by the SVD in test 1

This is reflected in the percentage of the variance captured by each of the principal directions, shown in Figure 2. The first explains almost 75 percent of the variance and the second over 20 percent. Between these two modes we can reconstruct over 95 percent of the dynamics displayed in the videos.

Figure 6 shows a visualization of the six projections of the measurements onto the left singular vector of the data matrix. Notice that the projection onto the first singular vector (the one with the large corresponding singular value) essentially gives all the relevant behavior of the system-sinusoidal oscillation. The second had a somewhat sizeable singular value and appears to both capture some noise and also indicate that the amplitude of the oscillating mass decreased over time (note the downward trend). This is likely due to air resistance. The remaining projections all seem to correspond to noise. They did not capture much variance and can essentially be ignored.

Things worked out very nicely in this case, but that is primarily because of the cleanliness of the data. We will see the devastating effect noise can have on the PCA in the next test.

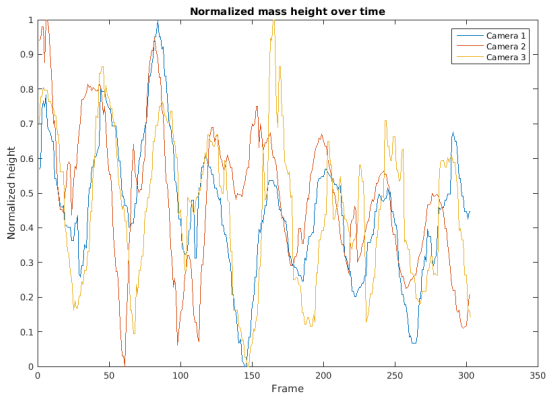

Figure 4. The normalized height of the weight over time as measured by the three cameras in test 2

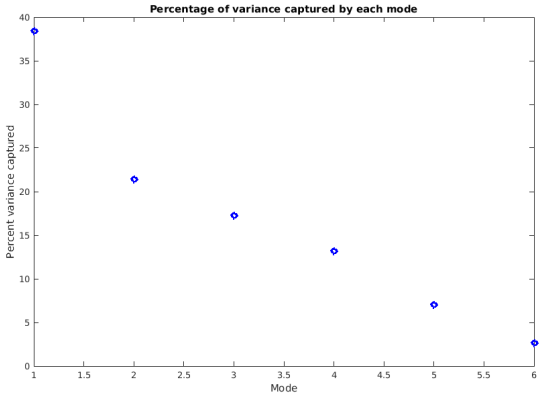

Figure 5. The percentage of the variance captured by the modes computed by the SVD in test 2

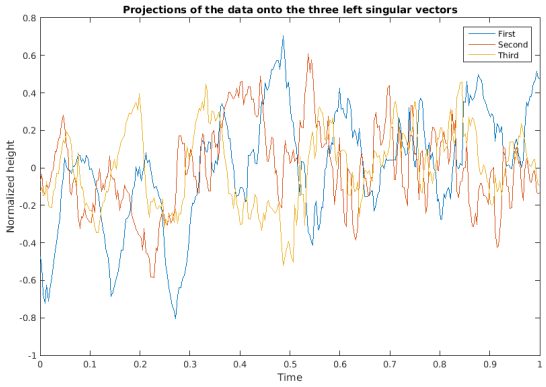

Figure 6. The projections of the data onto the first three left singular vectors computed by the SVD in test 2

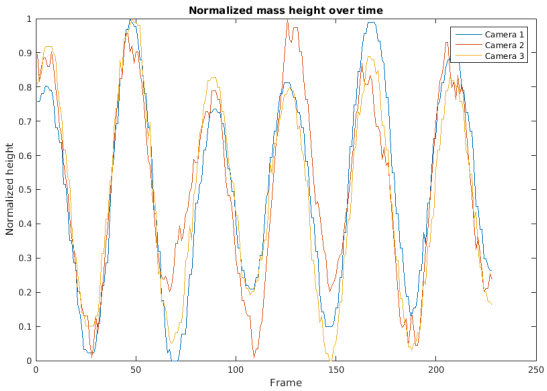

Figure 7. The normalized height of the weight over time as measured by the three cameras in test 3

4.2 Test 2: Noisy case

In the second test a large amount of artificial noise was introduced by the people holding the cameras. This is easy to spot in Figure 4 which shows the extracted height of the mass throughout the recordings. The data is clearly very noisy so we should not expect PCA to do as good a job of reducing the dimensionality of the dynamics. Indeed, we see in Figure 5 that the variance is much more evenly distributed among the modes discovered by the SVD.

The projections of the data onto the first three principal directions are given in Figure 6. They are much less comprehensible than in the last test. We see some periodic motion, but it is still very noisy. PCA is unable to adequately deal with all the noise introduced into the data.

4.3 Test 3: Horizontal displacement

In this test the mass was pushed horizontally as well as vertically so that it is both bobbing up and down and swaying from left to right. Thus we would expect PCA to discover two important modes (or perhaps three if the bucket was not pushed perfectly horizontally). As Figure 7 shows, we were able to extract fairly clean height measurements from the cameras. However, the horizontal positions extracted from the videos

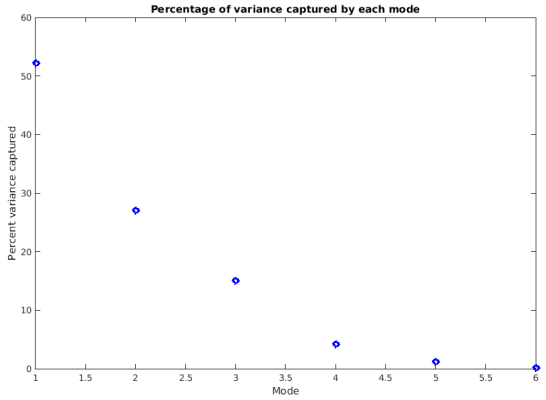

Figure 8. The percentage of the variance captured by the modes computed by the SVD in test 3

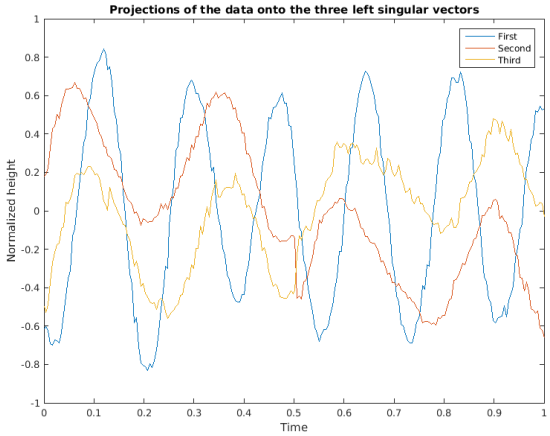

Figure 9. The projections of the data onto the first three left singular vectors computed by the SVD in test 3

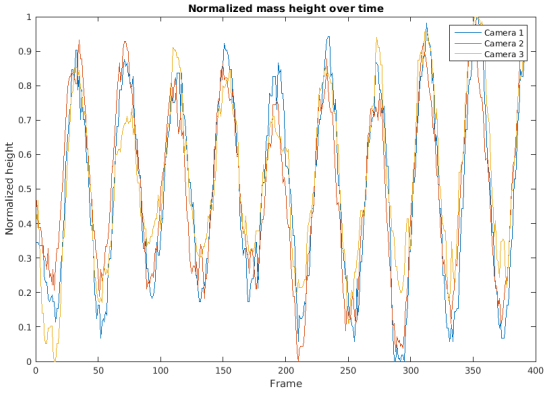

Figure 10. The normalized height of the weight over time as measured by the three cameras in test 4

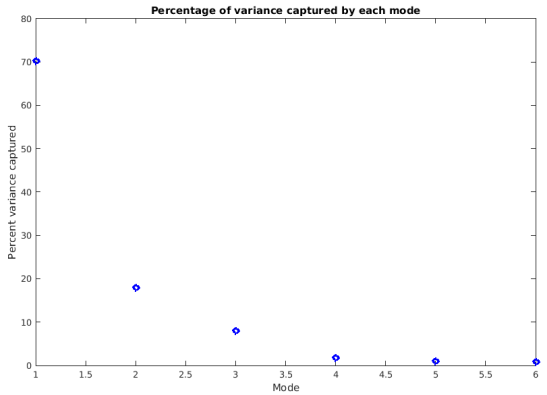

Figure 11. The percentage of the variance captured by the modes computed by the SVD in test 4

were not especially nice. This is evident in Figure 8, where we see that the amount of variance captured by the modes uncovered by PCA does not drop off as rapidly as in the first test. Nevertheless, the first three left singular vectors account for more than 90 percent of the variance. The projections onto these modes is shown in Figure 9. The first two probably show the vertical and horizontal oscillation exhibited by the swinging mass. The third looks to be highly correlated with the second.

4.4 Test 4: Horizontal displacement and rotation

For the fourth test the bucket was initially pushed in a manner similar to the first test, but it was also spun, making the flashlight difficult to track between frames. Ignoring the rotation, which we should not hope to be able to resolve without following the flashlight, we expect the dynamics to be similar to the first test. The measured height of the weight is given in Figure 10. Indeed it resembles Figure 1, except with more periods of oscillation and a bit more noise. From Figure 11 we see that most of the bucket movement can be described with just the first mode as it contains a whopping 70 percent of the variance. Figure 12, which shows the projections of the

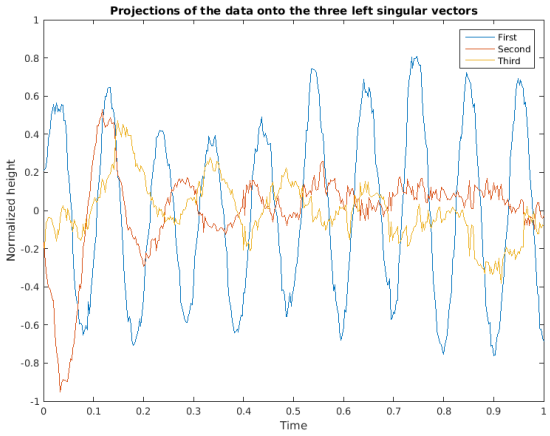

Figure 12. The projections of the data onto the first three left singular vectors computed by the SVD in test 4

data onto the first three left singular vectors, makes an even stronger case for the first mode containing the majority of the information. We observe the exact oscillatory behavior we saw in the first test.

5. Summary and Conclusions

We used the SVD and PCA to discover the lower dimensional behavior of a mass swinging from a string in three out of four tests. Given sets of recordings from three cameras we developed a methodology for extracting the vertical and horizontal position of the mass from each video frame. We then cleaned up these data and used PCA to remove redundancy and determine the optimal directions in which to represent the measurements. In the first and fourth tests we were able to uncover the one-dimensional oscillatory behavior of the mass. In the third we found the dynamics of the mass to be two-dimensional–it both oscillated up and down and swayed from side to side simultaneously. The collected positions from the videos in the second case were so rife with noise that we were unable to extract much meaningful information from them using the PCA.

References

[1] Jonathon Shlens. A tutorial on principal component analysis. 2005.

Appendix A: MATLAB Functions

Here we outline the nonstandard MATLAB functions used to complete this assignment.

`exist(filename,option)`: Returns true if the object at the location specified by `filename` of the type specified by `option` exists and returns false otherwise. For this assignment we use it to check whether or not certain variables exist so we can avoid redundant costly `load` commands.

`[R,C] = find(X)`: Locates the nonzero entries stored in the 2-dimensional array `A` and stores their row indices in `R` and their column indices in `C`.

`B = repmat(A,n,m)`: Creates a matrix `B` out of $n \cdot m$ copies of `A` (`n` copies in the vertical direction and `m` in the horizontal direction).

`[U,S,V] = svd(A)`: Computes the Singular Value Decomposition of the matrix `A`. `U`, `V`, and `S` are the matrices containing the left singular vectors, the right singular vectors, and the singular values of `A`, respectively. `AV = US` up to machine precision in MATLAB.

Appendix B: MATLAB Code

See the following pages published in MATLAB for the implementation of the algorithm presented in Section 3.

```
% Amath 582 Homework 4
% Brian de Silva
% 142824

% This script can be used to extract position data from the bucket
% recordings
```

Extract locations of buckets for analysis

```
testno = 1;     % Specify which test we are using

% Test 1: Ideal signal
% Test 2: Noisy signal
% Test 3: Horizontal displacement
% Test 4: Horizontal displacement and rotation

% Load the video data if it's not already loaded
if ~exist(['vidFrames1_' num2str(testno)],'var')
    load(['cam1_' num2str(testno) '.mat'])
end
if ~exist(['vidFrames2_' num2str(testno)],'var')
    load(['cam2_' num2str(testno) '.mat'])
end
if ~exist(['vidFrames3_' num2str(testno)],'var')
    load(['cam3_' num2str(testno) '.mat'])
end

% Give all video matrices the same name
switch testno
    case 1
        vidFrames1 = vidFrames1_1;
        vidFrames2 = vidFrames2_1;
        vidFrames3 = vidFrames3_1;
    case 2
        vidFrames1 = vidFrames1_2;
        vidFrames2 = vidFrames2_2;
        vidFrames3 = vidFrames3_2;
    case 3
        vidFrames1 = vidFrames1_3;
        vidFrames2 = vidFrames2_3;
        vidFrames3 = vidFrames3_3;
    otherwise
        vidFrames1 = vidFrames1_4;
        vidFrames2 = vidFrames2_4;
        vidFrames3 = vidFrames3_4;
end
```

Get location of bucket at each time step:

```
% Prime first position by actual inspection then only check for new
% position within a window of pixels near previous position
```

```
indices1 = zeros(2,size(vidFrames1,4));
figure()
imshow(vidFrames1(:,:,:,1)), title('Please click the flashlight at the
 top of the bucket')
[x1, y1] = ginput(1);
indices1(:,1) = [y1; x1];

indices2 = zeros(2,size(vidFrames2,4));
figure()
imshow(vidFrames2(:,:,:,1)), title('Please click the flashlight at the
 top of the bucket')
[x2, y2] = ginput(1);
indices2(:,1) = [y2; x2];

indices3 = zeros(2,size(vidFrames3,4));
figure()
imshow(vidFrames3(:,:,:,1)), title('Please click the flashlight at the
 top of the bucket')
[x3, y3] = ginput(1);
indices3(:,1) = [y3; x3];

close all

% Set parameters for first camera
rwindsize1 = 15;                          % Size of row window about
 previously known bucket location about which to look for new location
cwindsize1 = 15;                           % Size of column window
 about previously known bucket location about which to look for new
 location
local1 = uint8(zeros(rwindsize1+1,cwindsize1+1,3));

% Set parameters for second camera
rwindsize2 = 15;                          % Size of row window about
 previously known bucket location about which to look for new location
cwindsize2 = 15;                           % Size of column window
 about previously known bucket location about which to look for new
 location
rgb2 = [170,255,255];                % RGB values near the flashlight
 in test 2.2
local2 = uint8(zeros(rwindsize2+1,cwindsize2+1,3));

% Set parameters for third camera
rwindsize3 = 15;                             % Size of row window
 about previously known bucket location about which to look for new
 location
cwindsize3 = 15;                           % Size of column window
 about previously known bucket location about which to look for new
 location
local3 = uint8(zeros(rwindsize3+1,cwindsize3+1,3));

% Temporary window size variables in case window extends outside the
% image
temprsize = 0;
tempcsize = 0;
```

```
    % --------------First camera-----------------
for k=2:size(vidFrames1,4)
    row1 = indices1(1,k-1);            % Previous row location of
 bucket
    col1 = indices1(2,k-1);             % Previous column location of
 bucket

    % Make sure window doesn't extend outside image
    if (row1 - rwindsize1) < 1
        temprsize = row1-1 + (row1==1);
    elseif (row1 + rwindsize1) > size(vidFrames1,1)
        temprsize = size(vidFrames1,1) - row1 + (row1 ==
 size(vidFrames1,1));
    else
        temprsize = rwindsize1;
    end
    if (col1 - cwindsize1) < 1
        tempcsize = col1-1 + (col1==1);
    elseif (col1 + cwindsize1) > size(vidFrames1,1)
        tempcsize = size(vidFrames1,1) - col1 + (col1 ==
 size(vidFrames1,1));
    else
        tempcsize = cwindsize1;
    end

    % Look in a small window about previous bucket location
    local1 = vidFrames1((row1-temprsize):(row1+temprsize),(col1-
tempcsize):(col1+tempcsize),:,k);

    % Find pixel with maximum intensity
    local_filtered1 = (sum(local1,3)==max(max(sum(local1,3))));

    % Use top-left most pixel for location
    [r,c] = find(local_filtered1);
    indices1(1,k) = r(1) + (row1-temprsize)-1;
    indices1(2,k) = c(1) + (col1-tempcsize)-1;
end


% --------------Second camera-----------------
for k=2:size(vidFrames2,4)
    row2 = indices2(1,k-1);            % Previous row location of
 bucket
    col2 = indices2(2,k-1);             % Previous column location of
 bucket

    % Make sure window doesn't extend outside image
    if (row2 - rwindsize2) < 1
        temprsize = row2-1 + (row2==1);
    elseif (row2 + rwindsize2) > size(vidFrames2,1)
        temprsize = size(vidFrames2,1) - row2 + (row2 ==
 size(vidFrames2,1));
    else
```

```
        temprsize = rwindsize2;
    end
    if (col2 - cwindsize2) < 1
        tempcsize = col2-1 + (col2==1);
    elseif (col2 + cwindsize2) > size(vidFrames2,1)
        tempcsize = size(vidFrames2,1) - col2 + (col2 ==
 size(vidFrames2,1));
    else
        tempcsize = cwindsize2;
    end

    % Look in a small window about previous bucket location
    local2 = vidFrames2((row2-temprsize):(row2+temprsize),(col2-
tempcsize):(col2+tempcsize),:,k);

    % Find pixel with maximum intensity (whitest pixel)
    local_filtered2 = (sum(local2,3)==max(max(sum(local2,3))));

    % Use top-left most pixel for location
    [r,c] = find(local_filtered2);
    indices2(1,k) = r(1) + (row2-temprsize)-1;
    indices2(2,k) = c(1) + (col2-tempcsize)-1;
end


% --------------Third camera-----------------
for k=2:size(vidFrames3,4)
    row3 = indices3(1,k-1);            % Previous row location of
 bucket
    col3 = indices3(2,k-1);             % Previous column location of
 bucket

    % Make sure window doesn't extend outside image
    if (row3 - rwindsize3) < 1
        temprsize = row3-1 + (row3==1);
    elseif (row3 + rwindsize3) > size(vidFrames3,1)
        temprsize = size(vidFrames3,1) - row3 + (row3 ==
 size(vidFrames3,1));
    else
        temprsize = rwindsize3;
    end
    if (col3 - cwindsize3) < 1
        tempcsize = col3-1 + (col3==1);
    elseif (col3 + cwindsize3) > size(vidFrames3,1)
        tempcsize = size(vidFrames3,1) - col3 + (col3 ==
 size(vidFrames3,1));
    else
        tempcsize = cwindsize3;
    end

    % Look in a small window about previous bucket location
    local3 = vidFrames3((row3-temprsize):(row3+temprsize),(col3-
tempcsize):(col3+tempcsize),:,k);
```

```
    % Find pixel with maximum intensity (whitest pixel)
    local_filtered3 = (sum(local3,3)==max(max(sum(local3,3))));

    % Use top-left most pixel for location
    [r,c] = find(local_filtered3);
    indices3(1,k) = r(1) + (row3-temprsize)-1;
    indices3(2,k) = c(1) + (col3-tempcsize)-1;
end

% % Play the video
% for k=1:size(vidFrames1_1,4)
%     imshow(vidFrames1_1(:,:,:,k));
%     title(['t= ' num2str(k)])
%     drawnow
% end

% % Sanity check: plot locations the computer found against true video
% figure()
% for k=1:size(vidFrames2,4)
%     temp = vidFrames2(:,:,:,k);
%     temp((indices2(1,k)-3):(indices2(1,k)+3),(indices2(2,k)-3):
(indices2(2,k)+3),:)=0;
%     imshow(temp), axis on
%     title(['t= ' num2str(k)])
%     drawnow
% end


% Save the position measurements
% save(['test' num2str(testno) '.mat'], 'indices1', 'indices2',
 'indices3');
```

Published with MATLAB® R2015b

Amath 582 Homework 4

Brian de Silva 1422824

```
% This script is used to clean up and analyze the data extracted by
% extract_positions.m

close all
```

Load and clean up data

```
testno = 1;                     % Specify which test we are using
apply_filter = false;        % Specify whether or not to smooth data
 using fft

% Load the data produced by extract_positions
load(['test' num2str(testno) '.mat'])

% Max and min height should be the same for each camera so we
 normalize to
% [0, 1]
for i = 1:2
    indices1(i,:) = indices1(i,:) - min(indices1(i,:));
    indices1(i,:) = indices1(i,:) / max(indices1(i,:));

    indices2(i,:) = indices2(i,:) - min(indices2(i,:));
    indices2(i,:) = indices2(i,:) / max(indices2(i,:));

    indices3(i,:) = indices3(i,:) - min(indices3(i,:));
    indices3(i,:) = indices3(i,:) / max(indices3(i,:));
end

% Throw out some early data so that all three sources start at the
 same
% instant rather than at different times
switch testno
    case 1
        indices1 = indices1(:,9:end);
        indices2 = indices2(:,18:end);
        indices3 = indices3(:,8:end);
    case 2
        indices1 = indices1(:,12:end);
        indices2 = indices2(:,:);
        indices3 = indices3(:,12:end);
    case 3
        indices1 = indices1(:,12:end);
        indices2 = indices2(:,1:end);
        indices3 = indices3(:,4:end);
    otherwise
        indices1 = indices1(:,2:end);
        indices2 = indices2(:,10:end);
        indices3 = indices3(:,1:end);
```

```
end

% We want the same number of time steps for each camera so we cut off
 all
% the sources as soon as one of the cameras stops recording
minsteps = min([size(indices1,2), size(indices2,2),
 size(indices3,2)]);

% Put all the positions into a data matrix
X = zeros(6,minsteps);
X(1,:) = indices1(1,1:minsteps);
X(2,:) = indices1(2,1:minsteps);
X(3,:) = indices2(1,1:minsteps);
X(4,:) = indices2(2,1:minsteps);
X(5,:) = indices3(1,1:minsteps);
X(6,:) = indices3(2,1:minsteps);

% Plot the normalized heights
figure()
plot(X(1,:)), hold on
plot(X(3,:))
plot(X(6,:))
xlabel('Frame'), ylabel('Normalized height'), title('Normalized mass
 height over time')
legend('Camera 1', 'Camera 2', 'Camera 3')

% Bonus: what if we try smoothing the data with the fft? (not included
 in writeup)
if apply_filter
    f1 = [fftshift(fft(X(1,:))); fftshift(fft(X(2,:)))];
    f2 = [fftshift(fft(X(3,:))); fftshift(fft(X(4,:)))];
    f3 = [fftshift(fft(X(6,:))); fftshift(fft(X(5,:)))];
    figure()
    plot(abs(f1(1,:))), hold on
    plot(abs(f2(1,:)))
    plot(abs(f3(2,:)))
    title('Magnitudes of Fourier Coefficients')
    legend('Camera 1', 'Camera 2', 'Camera 3')

    % Create filter for smoothing
    [~, center] = max(abs(f1(1,:)+f2(1,:)+f3(2,:)));       % Center
 frequency after fftshift
    width = 0.005;
    k = 1:size(X,2);
    filter = exp(-width*(k-center).^2);
    filter = repmat(filter,2, 1);
    f1 = f1.*filter;
    f2 = f2.*filter;
    f3 = f3.*filter;

    X(1:2,:) = [ifft(ifftshift(f1(1,:))); ifft(ifftshift(f1(2,:)))];
    X(3:4,:) = [ifft(ifftshift(f2(1,:))); ifft(ifftshift(f2(2,:)))];
    X(5:6,:) = [ifft(ifftshift(f3(1,:))); ifft(ifftshift(f3(2,:)))];
```

```
    figure()
    plot(X(1,:)), hold on
    plot(X(3,:))
    plot(X(6,:))
    xlabel('Frame'), ylabel('Row index'), title('Row indices for
 cameras over time (filtered)')
    legend('Camera 1', 'Camera 2', 'Camera 3')
end

% Normalize time to [0,1]
t = linspace(0,1,minsteps);

% Subtract off the mean for each row of data
X = X - repmat(mean(X, 2), 1, minsteps);

% Compute SVD
[U, S, V] = svd(X / sqrt(minsteps-1));
lambda = diag(S).^2;

% Plot percentage of variance captured by different modes
figure()
plot((lambda / sum(lambda)) * 100,'bo','Linewidth',3)
title('Percentage of variance captured by each mode')
xlabel('Mode')
ylabel('Percent variance captured')

% Project onto the left singular vectors
Y = U' * X;

% Plot projections
figure()
plot(t,Y(1,:),t,Y(2,:),t,Y(3,:)) %,t,Y(4,:),t,Y(5,:),t,Y(6,:))
legend('First','Second','Third') %,'Fourth','Fifth','Sixth')
xlabel('Time')
ylabel('Normalized height')
title('Projections of the data onto the three left singular vectors')
```