Principal Component Analysis and a Spring-Mass system

Yumin Yin

February 24, 2021

Abstract

Considering the scenario that a mass hanging from a spring and the mass bobs up an down, instead of simply using Newton's second law and Hooke's law, we want to understand the motion of the mass-spring system experimentally. We set up three cameras around the system and collect videos for four different cases. We will perform the principal component analysis (PCA) on the data about the displacement of the mass to understand its motion and to illustrate various aspects of the PCA, its practical usefulness, and the effects of noise on the PCA algorithms.

1 Introduction and Overview

We set up three cameras and record the movements of the mass hanging from a spring. We have four sets of tests corresponding to four different scenarios. The strategy we use to track the mass in each video is by looking for the bright spot on the top of it. Since the spot is white, it is fairly easy to identify after we convert each frame to grayscale. We need to synchronize and trim the videos to adjust the data we collected about the locations of the mass so that we are able to construct one data matrix to do the principal component analysis. After that, We compute the SVD of our data matrix divided by $\sqrt{n-1}$. This helps us to calculate the energies for our low-rank approximations and plot them to see which principal components are significant. Lastly, we plot the displacement of the mass across z-axis, xy plane, and significant principal component bases to compare the projection(s) with the experimental data.

2 Theoretical Background

2.1 The Singular Value Decomposition (SVD)

What is typically done is to construct a matrix $\mathbf{U}$ from $\hat{\mathbf{U}}$ by adding an addition $m-n$ columns that are orthonormal to the already existing set $\hat{\mathbf{U}}$. In this way $\mathbf{U}$ becomes a square $m \times m$ matrix, and in order to make the decomposition work, additional $m-n$ rows of zeros is also added to the matrix $\hat{\mathbf{\Sigma}}$, resulting in the matrix $\mathbf{\Sigma}$. This leads to the Equation (1), the **singular value decomposition** of the matrix $\mathbf{A}$:

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

where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are unitary matrices, and $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ is diagonal. The values σ_n on the diagonal of Σ are called the **singular values** of the matrix $\mathbf{A}$. The vectors u_n which make up the columns of $\mathbf{U}$ are called the **left singular vectors** of $\mathbf{A}$. The vectors v_n which make up the columns of $\mathbf{V}$ are called the **right singular vectors** of $\mathbf{A}$.

Geometrically, multiplying by $\mathbf{V}^*$ on the left rotates the hypersphere to align the $\mathbf{v_n}$ vectors with the axes. Then we multiply on the left by a diagonal matrix which stretches in each direction. After that, we multiply on the left by $\mathbf{U}$ to rotate the hyerellipse to the proper orientation.

2.2 Computing the SVD

The SVD can be done by deriving the Equation (2):

$$\begin{aligned}\mathbf{A}^*\mathbf{A} &= (\mathbf{U\Sigma V}^*)^*(\mathbf{U\Sigma V}^*)\\ &= \mathbf{U\Sigma V}^*\mathbf{U\Sigma V}^*\\ &= \mathbf{U\Sigma^2 V}^*.\end{aligned} \tag{2}$$

Multiplying on the right by $\mathbf{V}$ gives the Equation (3):

$$\mathbf{A}^*\mathbf{AV} = \mathbf{V\Sigma^2}, \tag{3}$$

so that the columns of $\mathbf{V}$ are eigenvectors of $\mathbf{A}^*\mathbf{A}$ with eigenvalues given by the square of the singular values. Similarly, the Equation (4)

$$\mathbf{AA}^* = \mathbf{U\Sigma^2 U}^*. \tag{4}$$

and so multiplying on the right by $\mathbf{U}$ gives the eigenvalue problem, the Equation (5)

$$\mathbf{AA}^*\mathbf{U} = \mathbf{U\Sigma^2}, \tag{5}$$

which can be used to solve for $\mathbf{U}$.

2.3 Comparing SVD and Eigenvalue Decomposition

There are two major advantages of the SVD. One is the SVD can be performed on matrices of any size, while only square matrices can be diagonalized. This is extremely important since real-world data does not always come in the form of a square matrix. The other one is the SVD finds two different bases, while eigenvalue decomposition only finds one. Furthermore, the SVD gives orthogonal bases while eigenvectors are not always orthogonal.

2.4 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a method that uses the SVD to produce low-rank approximations of a data set. It is commonly used for dimensionality reduction. For matrix $\mathbf{X}$, we can compute all the variances and covariances between the rows of $\mathbf{X}$ with one matrix multiplication, the Equation (6):

$$\mathbf{C_x} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^\mathbf{T}. \tag{6}$$

$\mathbf{C_x}$ is a square symmetric matrix. Unsurprisingly, it is called the **covariance matrix**. The goal of principal component analysis is to find a new set of coordinates (a change of basis) so that the variables are now uncorrelated. That will mean that each variable contains completely new information, i.e., no redundancies. It would also be nice to know which variables have the largest variance because these contain the most important information about our data. Therefore, we want to diagonalize this matrix, the Equation (7), so that all off-diagonal elements (covariances) are zeros:

$$\mathbf{C_x} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-\mathbf{1}}. \tag{7}$$

The basis of eigenvectors contained in $\mathbf{V}$ are called the **principal components**. They are uncorrelated since they are orthogonal. The data in the new coordinate is, the Equation (8):

$$\mathbf{Y} = \mathbf{U}^\mathbf{T}\mathbf{X}. \tag{8}$$

The covariance of $\mathbf{Y}$ is, the Equation (9):

$$\mathbf{C_y} = \frac{1}{n-1}\mathbf{Y}\mathbf{Y}^\mathbf{T} = \frac{1}{n-1}\mathbf{U}^\mathbf{T}\mathbf{X}\mathbf{X}^\mathbf{T}\mathbf{U} = \mathbf{U}^\mathbf{T}\mathbf{A}\mathbf{A}^\mathbf{T}\mathbf{U} = \mathbf{U}^\mathbf{T}\mathbf{U}\mathbf{\Sigma}^\mathbf{2}\mathbf{U}^\mathbf{T}\mathbf{U} = \mathbf{\Sigma}^\mathbf{2}. \tag{9}$$

2.5 Determine How Good Is Our Approximation

We will measure this be determining how much **energy** from the full system is contained in each node. Assuming the full matrix is rank r, we measure the energy contained in the rank-N approximation by the Equation (10):

$$\text{energy}_N = \frac{\sigma_1^2 + \sigma_2^2 + ...\sigma_N^2}{\sigma_1^2 + \sigma_2^2 + ...\sigma_r^2} = \frac{||X_N||_F^2}{||X||_F^2} \tag{10}$$

2.6 Function Expansions

Given a function, $f(x,t)$, we would like to expand it into the sum of some basis functions with time-dependent coefficients, the Equation (11):

$$f(x,t) = \sum_{n=1}^{\infty} a_n(t)\phi_n(x). \tag{11}$$

We can then approximate the function by truncating to a finite number of terms in the series, the Equation (12):

$$f(x,t) \approx \sum_{n=1}^{\infty} a_n(t)\phi_n(x). \tag{12}$$

3 Algorithm Implementation and Development

3.1 Tracking the Mass

For each frame in the video, we convert it to the grayscale image and the values in the image to double precision. We want to narrow each frame down to approximately where the mass could be, so we zero out the values of pixels outside where we are interested in. Taking the advantage of that there is a bright (white) spot on the top of the mass, we can look for white pixels in each frame to keep track of the mass. Generally, we consider the value of a pixel that is greater than 250 to be white. We have to lower the standard to be white due to the flaws in the data. After collecting the x and y coordinates of all white pixels, we average these coordinates so that there is one pair of x-y coordinates representing the mass in each frame. This is analogous to that we define the center of mass in physics.

3.2 Cleaning the Data

To synchronize three videos, we start each video with the frame corresponding with the lowest y coordinates. After that, we adjust the length of each video, in terms of the number of frames, so that we are able to put the x and y coordinates of the mass in each frame from each video into one big data matrix. The last step is to subtract the mean of each row from the data so that each row has zero mean.

3.3 Performing PCA and Plotting

We compute the SVD of our data matrix divided by $\sqrt{n-1}$. We then calculate the energies for our low-rank approximations and plot them to see which principal components are significant. Lastly, we plot the displacement of the mass across z-axis, xy plane, and significant principal component bases to compare the projection(s) with the experimental data.

3.4 Repeating

We repeat the algorithm described above for all four cases.

4 Computational Results

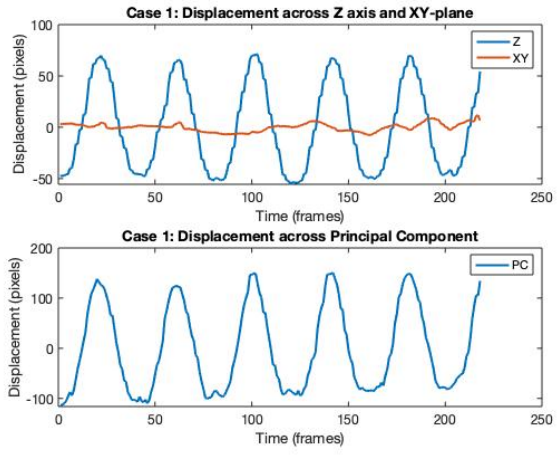

Figure 1: Displacement of the mass (Test 1)

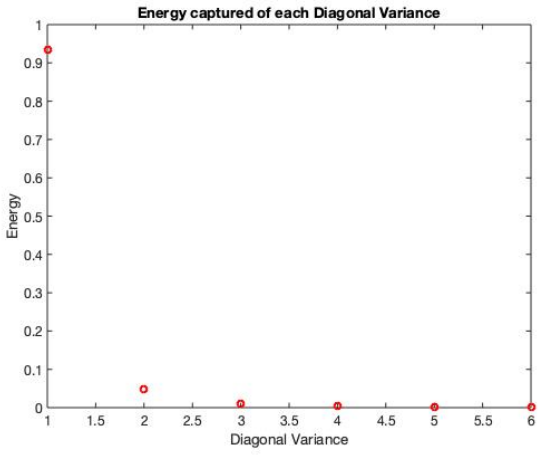

Figure 2: Energy captured by each PC (Test 1)

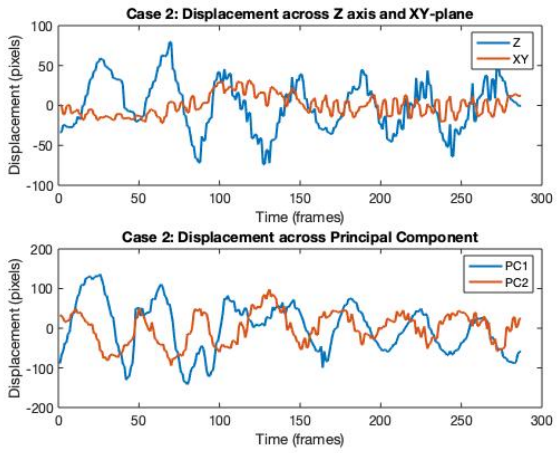

Figure 3: Displacement of the mass (Test 2)

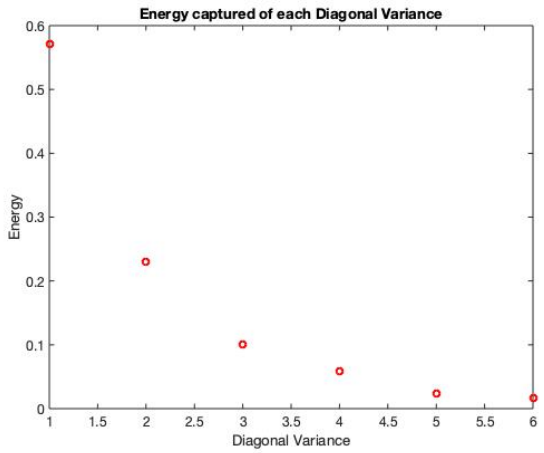

Figure 4: Energy captured by each PC (Test 2)

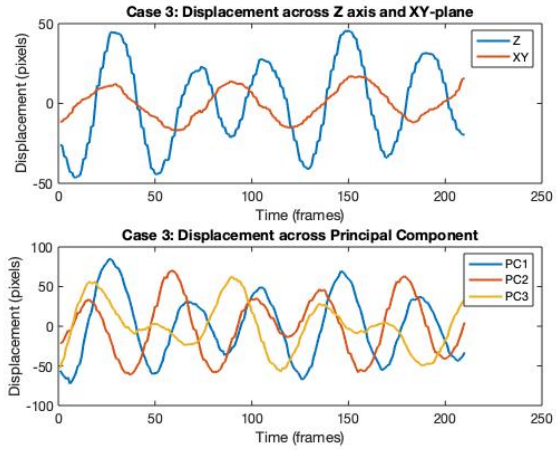

Figure 5: Displacement of the mass (Test 3)

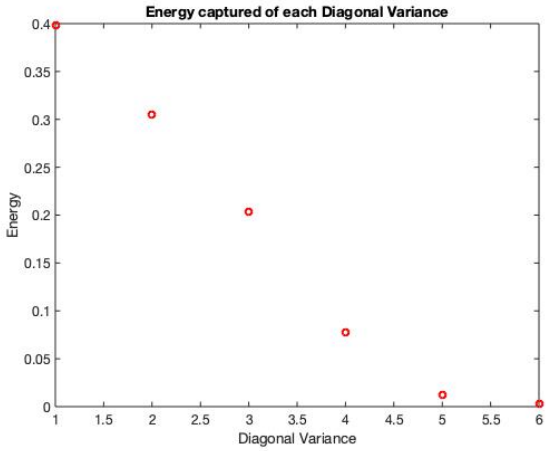

Figure 6: Energy captured by each PC (Test 3)

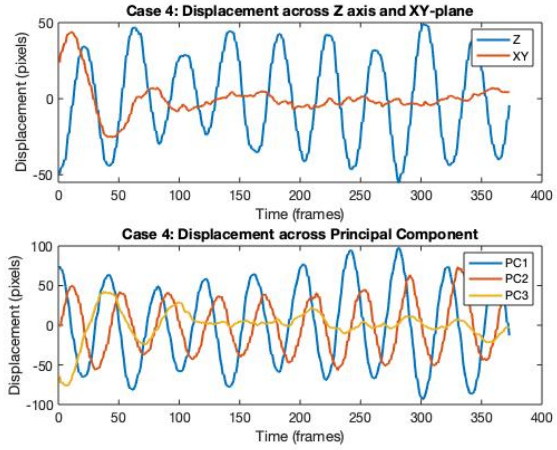

Figure 7: Displacement of the mass (Test 4)

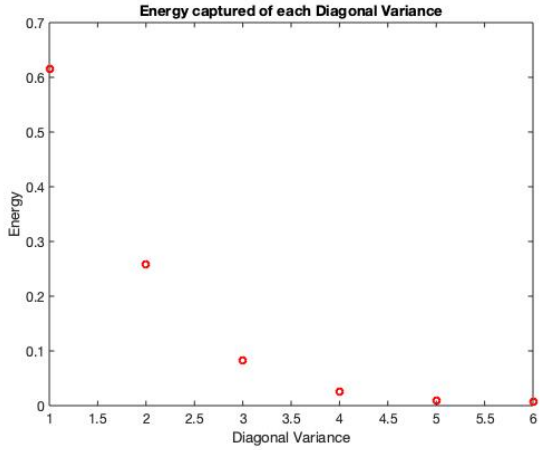

Figure 8: Energy captured by each PC (Test 4)

4.1 Ideal Case

The first graph in the Figure (1) tells us the entire motion is simple harmonic motion moving in the z direction. We can read off from the Figure (2) that the first principal component captures the majority of the energy ($\sim$91%). This verifies the motion of the mass is in one direction and indicates the one significant principal component is good enough to project the motion accurately. Compared two graphs in the Figure (1), we find out the projected motion onto the first principal component basis perfectly fits with the observed data from our experiment. The PCA also concludes the entire motion is simple harmonic motion moving in the z direction.

4.2 Noisy Case

The first graph in the Figure (3) tells us the mass is mainly moving in the z direction. Unfortunately, camera shakes causing a slight horizontal movement of the mass. We can read off from the Figure (4) that the first two principal components capture the majority of the energy ($\sim$58% and $\sim$23%). The first principal component captures significantly much energy than all the others, representing the dominant up-down motion of the mass. We choose to project the motion onto the two principal component bases, as this is what the Figure (4) suggests. Compared two graphs in the Figure (3), we find out PCA still decently projects the oscillatory movement of the mass in the z direction. Even though this test is repeating the ideal case experiment, it seem there are some movements in other directions. This is because the noise throw things off a little bit.

4.3 Horizontal Displacement

The first graph in the Figure (5) suggests the motion is pendulum motion moving in the $x-y$ plane as well as the z direction. We can read off from the Figure (6) that the first three principal components capture the most of the energy ($\sim$40% , $\sim$30%, and $\sim$20%). This confirms that different from the Test 1, the mass in this test is moving in three directions, in the $x-y$ plane and the z direction. Two graphs in the Figure (5) show the companion between the original motion and the projection onto the first three principal component bases.

4.4 Horizontal Displacement and Rotation

We can read off from the Figure (8) that the first three principal components capture the most of the energy ($\sim$61% , $\sim$26%, and $\sim$10%). This indicates that different from our benchmark, Test 1, the motion of the mass in this test is three dimensional. In addition, it is also different with the Test 3, since the Figure (7) and (8) tell a different story than the Figure (5) and (6) do. By actually watching the video, we know that besides moving in a pendulum-fashion, the mass is rotating. This test tells us that the PCA is able to capture the both three-dimensional movement and rotation of the mass. Two graphs in the Figure (7) show the companion between the original motion and the projection onto the first three principal component bases.

5 Summary and Conclusions

We have succeed in applying principal component analysis to project the motion of the mass hanging from a spring. Looking for the white pixels, which are the bright spot on the top of the mass, in each grayscale frame of videos is a pleasant way to keep track of the mass. The principal component analysis is able to capture dimensonality of the motion of the mass. Also, it can tell if the mass is spinning. The projection of the entire motion of the mass onto the significant principal component bases is fairly accurate. Certainly, the accuracy depends on the noise in the data.

Appendix A MATLAB Functions

- `sz = size(A)` returns a row vector whose elements are the lengths of the corresponding dimensions of `A`. For example, if `A` is a 3-by-4 matrix, then `size(A)` returns the vector [3 4].
- `M = mean(A)` returns the mean of the elements of `A` along the first array dimension whose size does not equal 1.
- `B = repmat(A,r1,...,rN)` specifies a list of scalars, `r1,..,rN`, that describes how copies of `A` are arranged in each dimension.
- `[U,S,V] = svd(A)` performs a singular value decomposition of matrix `A`, such that `A = U*S*V'`.
- `x = diag(A)` returns a column vector of the main diagonal elements of `A`.
- `S = sum(A)` returns the sum of the elements of `A` along the first array dimension whose size does not equal 1.
- `L = length(X)` returns the length of the largest array dimension in `X`.
- `find (X == Y)` returns the indices where `X = Y`.
- `[I1,I2,...,In] = ind2sub(sz,ind)` returns `n` arrays `I1,I2,...,In` containing the equivalent multidimensional subscripts corresponding to the linear indices `ind` for a multidimensional array of size `sz`. Here `sz` is a vector with `n` elements that specifies the size of each array dimension.
- `X = zeros(sz1,...,szN)` returns an `sz1-by-...-by-szN` array of zeros where `sz1,...,szN` indicate the size of each dimension.
- `I = rgb2gray(RGB)` converts the truecolor image RGB to the grayscale image I.
- `Y = double(X)` converts the values in `X` to double precision.
- `M = min(A)` returns the minimum elements of an array.
- `plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)` sets the line style, marker type, and color for each line.

Appendix B MATLAB Code

```
%% Ideal Case

clear all; close all; clc

load('cam1_1.mat'); load('cam2_1.mat'); load('cam3_1.mat');
mass1_1 = trackMass(vidFrames1_1, 170, 430, 300, 400, 250);
mass2_1 = trackMass(vidFrames2_1, 100, 450, 200, 350, 250);
mass3_1 = trackMass(vidFrames3_1, 200, 350, 235, 500, 245);

% Put videos in sync
mass1_1 = trimmer(mass1_1);
mass2_1 = trimmer(mass2_1);
mass3_1 = trimmer(mass3_1);

case1 = [mass1_1'; mass2_1(1:length(mass1_1),:)'; mass3_1(1:length(mass1_1),:)
    '];

% Subtract the mean of each row from the data
```

```
[m,n] = size(case1);
mean_of_row = mean(case1, 2);
case1 = case1 - repmat(mean_of_row, 1, n);

% Compute the SVD
[u,s,v] = svd(case1/sqrt(n-1));
DV = diag(s).^2; % diagonal variances
PC = u' * case1; % principal components

% plot
figure()
plot(1:6, DV/sum(DV), 'ro', 'Linewidth', 2);
title("Energy captured of each Diagonal Variance");
xlabel("Diagonal Variance");
ylabel("Energy");
set(gca, 'FontSize', 12)

figure()
subplot(2,1,1)
plot(1:length(mass1_1), case1(2,:),1:length(mass1_1), case1(1,:), 'Linewidth',
    2)
title("Case 1: Displacement across Z axis and XY-plane");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("Z", "XY")
set(gca, 'FontSize', 12)
subplot(2,1,2)
plot(1:length(mass1_1), PC(1,:),'Linewidth', 2)
title("Case 1: Displacement across Principal Component");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("PC")
set(gca, 'FontSize', 12)

%% Noisy Case

clear all; close all; clc

load('cam1_2.mat'); load('cam2_2.mat'); load('cam3_2.mat');
mass1_2 = trackMass(vidFrames1_2, 170, 430, 300, 450, 250);
mass2_2 = trackMass(vidFrames2_2, 50, 475, 150, 450, 245);
mass3_2 = trackMass(vidFrames3_2, 200, 400, 210, 500, 245);

% Put videos in sync
mass1_2 = trimmer(mass1_2);
mass2_2 = trimmer(mass2_2);
mass3_2 = trimmer(mass3_2);

case2 = [mass1_2'; mass2_2(1:length(mass1_2),:)'; mass3_2(1:length(mass1_2),:)
    '];

% Subtract the mean of each row from the data
[m,n] = size(case2);
mean_of_row = mean(case2, 2);
```

```
case2 = case2 - repmat(mean_of_row, 1, n);

% Compute the SVD
[u,s,v] = svd(case2/sqrt(n-1));
DV = diag(s).^2; % diagonal variances
PC = u' * case2; % principal components

% plot
figure()
plot(1:6, DV/sum(DV), 'ro', 'Linewidth', 2);
title("Energy captured of each Diagonal Variance");
xlabel("Diagonal Variance");
ylabel("Energy");
set(gca, 'FontSize', 12)

figure()
subplot(2,1,1)
plot(1:length(mass1_2), case2(2,:),1:length(mass1_2), case2(1,:), 'Linewidth',
    2)
title("Case 2: Displacement across Z axis and XY-plane");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("Z", "XY")
set(gca, 'FontSize', 12)
subplot(2,1,2)
plot(1:length(mass1_2), PC(1,:), 1:length(mass1_2), PC(2,:), 'Linewidth', 2)
title("Case 2: Displacement across Principal Component");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("PC1", "PC2")
set(gca, 'FontSize', 12)

%% Horizontal Displacement

clear all; close all; clc

load('cam1_3.mat'); load('cam2_3.mat'); load('cam3_3.mat');
mass1_3 = trackMass(vidFrames1_3, 225, 450, 275, 450, 250);
mass2_3 = trackMass(vidFrames2_3, 100, 450, 150, 425, 250);
mass3_3 = trackMass(vidFrames3_3, 150, 365, 210, 500, 245);

% Put videos in sync
mass1_3 = trimmer(mass1_3);
mass2_3 = trimmer(mass2_3);
mass3_3 = trimmer(mass3_3);

case3 = [mass1_3'; mass2_3(1:length(mass1_3),:)'; mass3_3(1:length(mass1_3),:)
    '];

% Subtract the mean of each row from the data
[m,n] = size(case3);
mean_of_row = mean(case3, 2);
case3 = case3 - repmat(mean_of_row, 1, n);

```

```
% Compute the SVD
[u,s,v] = svd(case3/sqrt(n-1));
DV = diag(s).^2; % diagonal variances
PC = u' * case3; % principal components

% plot
figure()
plot(1:6, DV/sum(DV), 'ro', 'Linewidth', 2);
title("Energy captured of each Diagonal Variance");
xlabel("Diagonal Variance");
ylabel("Energy");
set(gca, 'FontSize', 12)

figure()
subplot(2,1,1)
plot(1:length(mass1_3), case3(2,:),1:length(mass1_3), case3(1,:), 'Linewidth', 2)
title("Case 3: Displacement across Z axis and XY-plane");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("Z", "XY")
set(gca, 'FontSize', 12)
subplot(2,1,2)
plot(1:length(mass1_3), PC(1,:), 1:length(mass1_3), PC(2,:), 1:length(mass1_3), PC(3,:), 'Linewidth', 2)
title("Case 3: Displacement across Principal Component");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("PC1", "PC2", "PC3")
set(gca, 'FontSize', 12)

%% Horizontal Displacement and Rotation

clear all; close all; clc

load('cam1_4.mat'); load('cam2_4.mat'); load('cam3_4.mat');
mass1_4 = trackMass(vidFrames1_4, 225, 450, 275, 450, 245);
mass2_4 = trackMass(vidFrames2_4, 100, 450, 150, 425, 245);
mass3_4 = trackMass(vidFrames3_4, 100, 300, 260, 500, 230);

% Put videos in sync
mass1_4 = trimmer(mass1_4);
mass2_4 = trimmer(mass2_4);
mass3_4 = trimmer(mass3_4);

case4 = [mass1_4(1:length(mass3_4),:)'; mass2_4(1:length(mass3_4),:)'; mass3_4'];

% Subtract the mean of each row from the data
[m,n] = size(case4);
mean_of_row = mean(case4, 2);
case4 = case4 - repmat(mean_of_row, 1, n);

% Compute the SVD
```

```
[u,s,v] = svd(case4/sqrt(n-1));
DV = diag(s).^2; % diagonal variances
PC = u' * case4; % principal components

% plot
figure()
plot(1:6, DV/sum(DV), 'ro', 'Linewidth', 2);
title("Energy captured of each Diagonal Variance");
xlabel("Diagonal Variance");
ylabel("Energy");
set(gca, 'FontSize', 12)

figure()
subplot(2,1,1)
plot(1:length(mass3_4), case4(2,:),1:length(mass3_4), case4(1,:), 'Linewidth', 2)
title("Case 4: Displacement across Z axis and XY-plane");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("Z", "XY")
set(gca, 'FontSize', 12)
subplot(2,1,2)
plot(1:length(mass3_4), PC(1,:), 1:length(mass3_4), PC(2,:), 1:length(mass3_4), PC(3,:), 'Linewidth', 2)
title("Case 4: Displacement across Principal Component");
xlabel("Time (frames)");
ylabel("Displacement (pixels)");
legend("PC1", "PC2", "PC3")
set(gca, 'FontSize', 12)


%% functions

function mass = trackMass(video, ymin, ymax, xmin, xmax, white)
    numOfFrames = size(video, 4); % number of frames in the given video
    mass = []; % x and y coordinates of the mass in each frame
    filter = zeros(480,640);
    filter(ymin:1:ymax, xmin:1:xmax) = 1;
    for i = 1:numOfFrames
        I = rgb2gray(video(:,:,:,i));
        I = double(I).*filter; % zero out the values of pixels outside where we are interested in
        [Y, X] = ind2sub(size(I), find(I > white));
        mass = [mass; mean(X), mean(Y)]; % define the center of mass
    end
end

function sync_video = trimmer(raw_video)
    [M, I] = min(raw_video(1:30,2)); % start with the frame has the lowest y coordinate
    sync_video = raw_video(I:end,:);
end
```