Homework 3: PCA

Minghu Zhou

February 17, 2020

Abstract

In this report we will use Principal Component Analysis(PCA) in a spring and mass system. The experiments are an attempt to illustrate various aspects of the PCA and its practical usefulness and the effects of noise on the PCA algorithms.

1. Introduction and Overview

There are four situations with different behavior in the spring-mass system, which we take videos of. For each situation, we take 3 videos at different locations to track the movement of the bucket tied to the spring. We can track the movement by locating the position of the light on the bucket in chosen frames, denoted by (x, y). Thus, each camera will have two measurements of x and y through frames. Combing all data, we have a matrix X having 6 measurements collected by camera A, B and C.

$$X = \begin{bmatrix} \vec{x_a} \\ \vec{y_a} \\ \vec{x_b} \\ \vec{y_b} \\ \vec{x_c} \\ \vec{y_c} \end{bmatrix}$$

With this matrix, we can apply PCA on it. We will reduce the noise and redundancy with PCA and find important principal components. In this way, we can reduce the dimension and still captures the bucket's general movement.

2. Theoretical Background

Principal Component Analysis(PCA) allows us to remove redundancies of the data in a matrix. It can lower dimensions of dynamics we are interested and give us the coordinate system which best captures the general behavior.

An assumption of PCA is that our data should be mean centered. So a data matrix X needs to subtract each measurement by the mean of it. To remove redundancy of a matrix X, the covariance matrix of it is attended:

$$C_X = \frac{1}{n-1} X X^T$$

The diagonal of the covariance matrix represents the variance of each measurement. The larger the variance is, the more important is that measurement. The elements off the diagonal corresponds to covariance of different measurements. If two measurements are very redundant, their covariance will be large; if they are very uncorrelated, their covariance will be small. In order to find a coordinate system with no redundancy, we have to make covariance between any two different measurements equal to 0. This means we need to diagonalize the covariance matrix.

We can use Singular value decomposition (SVD) to diagonalize the covariance matrix. A matrix X can be reformatted into the following:

$$X = U\Sigma V^T$$

where U and V are unitary and Σ is diagonal. With unitary transformation matrix U as a basis, the transformed variable Y can be defined as:

$$Y = U^{-1}X = U^T X$$

The covariance matrix of Y is:

$$\begin{aligned} C_Y &= \frac{1}{n-1} YY^T \\ &= \frac{1}{n-1} (U^T X)(U^T X)^T \\ &= \frac{1}{n-1} U^T X X^T U \\ &= \frac{1}{n-1} U^T U \Sigma V^T V \Sigma^T U^T U \\ &= \frac{1}{n-1} \Sigma^2 \end{aligned}$$

Since Σ is a diagonal matrix, C_Y is also diagonal. In this way, our covariance matrix is diagonalized and redundancy is removed.

3. Algorithms Implementation and Development

The algorithm is almost same for four experiments. First, we loaded the video data of three cameras. In one video, we first play it and make a filter matrix with 0 and 1 of the size of one frame to filter out areas that we are not interested. The left rectangular area includes the trajectory of the bucket. Then, we find the number of frames of the video and iterate over these frames. We convert each frame to grayscale. Since the bucket is white, we want to find the pixels with very high values. Thus, we create a threshold matrix to find all pixels that having value over a threshold number (subject to change in each video). These pixels have a 1 value in the threshold matrix. Using *find* and *ind2sub* functions we can get the x and y coordinate of these bright pixel. Average these coordinate, we can get a coordinate that can represent the bucket in one frame and we add averaged x and y to the data matrix. After iterations we will get a matrix with size 2 * number of frames. Apply the above process to all three videos we will have three matrices.

However, each video's number of frames is different, which means the three matrices have different sizes. Meanwhile, these videos do not synchronize. Thus, we need to trim these three matrices. First, we choose our start frame to the one where the bucket has the highest position in the first 20 frames. This corresponds to the smallest y in first two videos and smallest x in the third one. After trimming some beginning columns, the matrices are in sync. Then we find the shortest length of three matrices and cut columns over this length of the two longer matrices. Now these three matrices have the same size. We then combine them to a big matrix of size 6 * shortest length.

Next, we subtract each row by the mean of each row to make our data mean centered. Then we use *svd* function to our data matrix divided by square root of the length of the matrix minus 1 and get matrices u, s and v. We extract the diagonal of s as a vector and square each value in the vector to get variances of principal components. Then, we can plot the energy of each component, which is denoted by its variance divided by the sum of variances. In this way, we can know which principal components are important in the dynamics. Next, we multiply u transpose and our data matrix, so our data are projected into the new coordinate system and we can plot the data projected on important principal component bases.

4. Computational Results

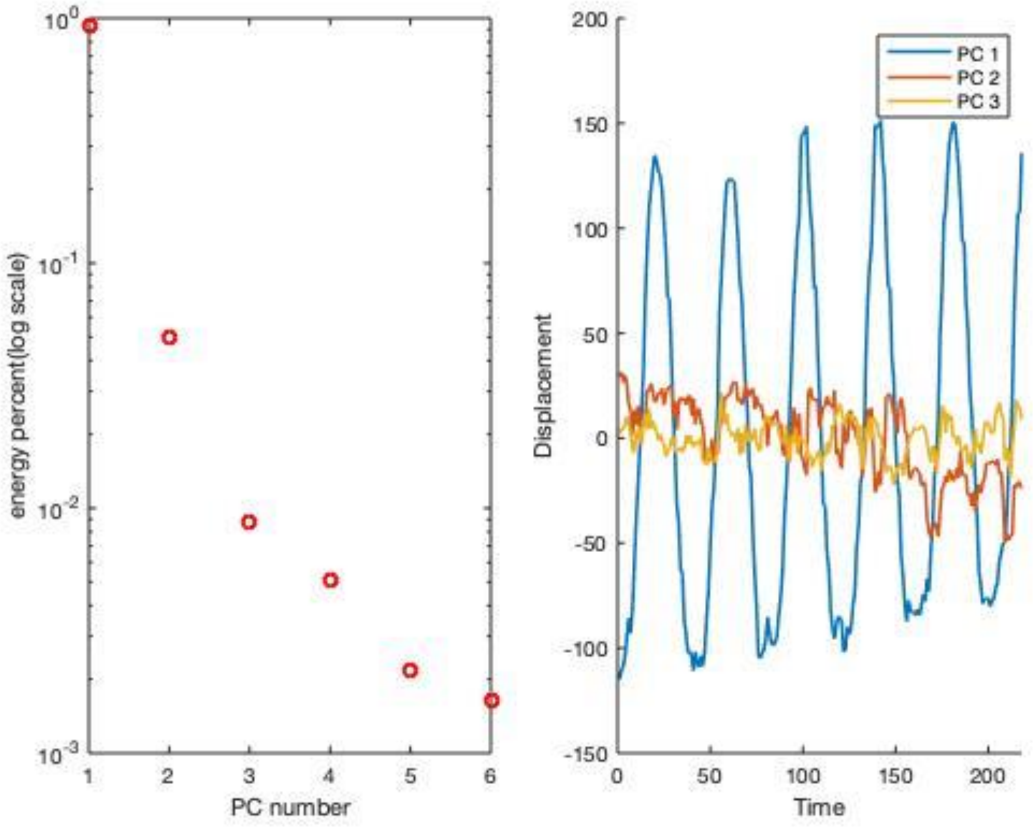

Figure 1: Case 1 principal component energy(left) and data projected on important principal component(right)

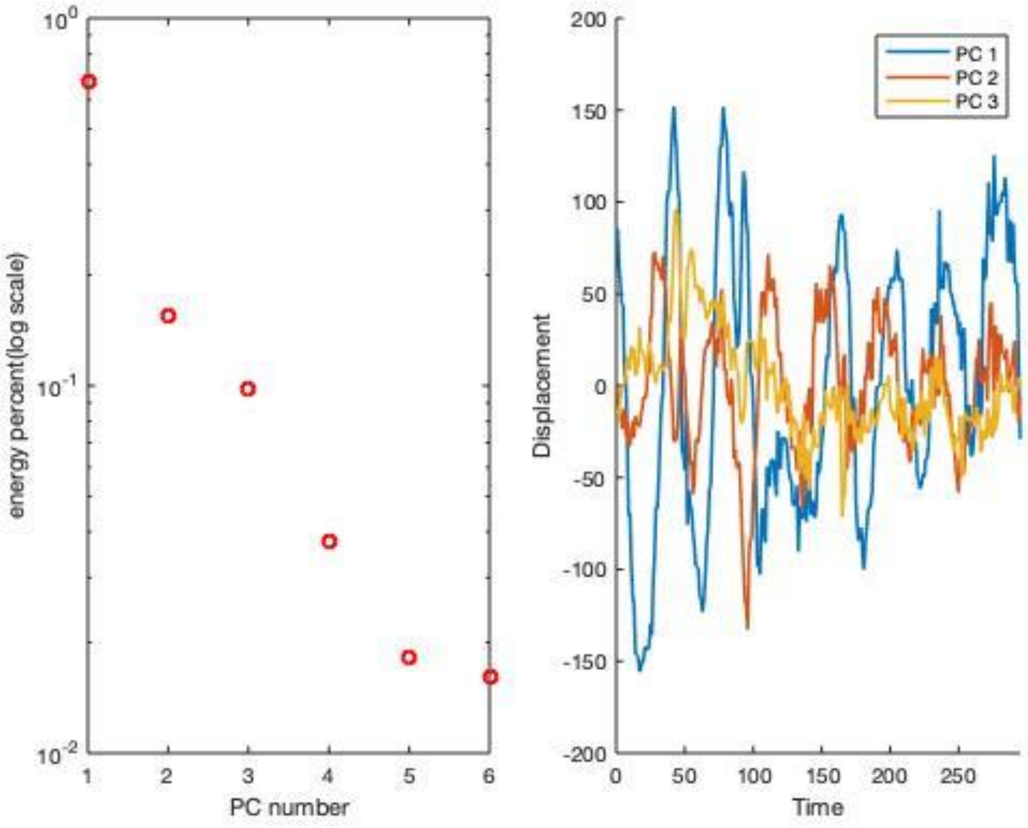

Figure 2: Case 2 principal component energy(left) and data projected on important principal component(right)

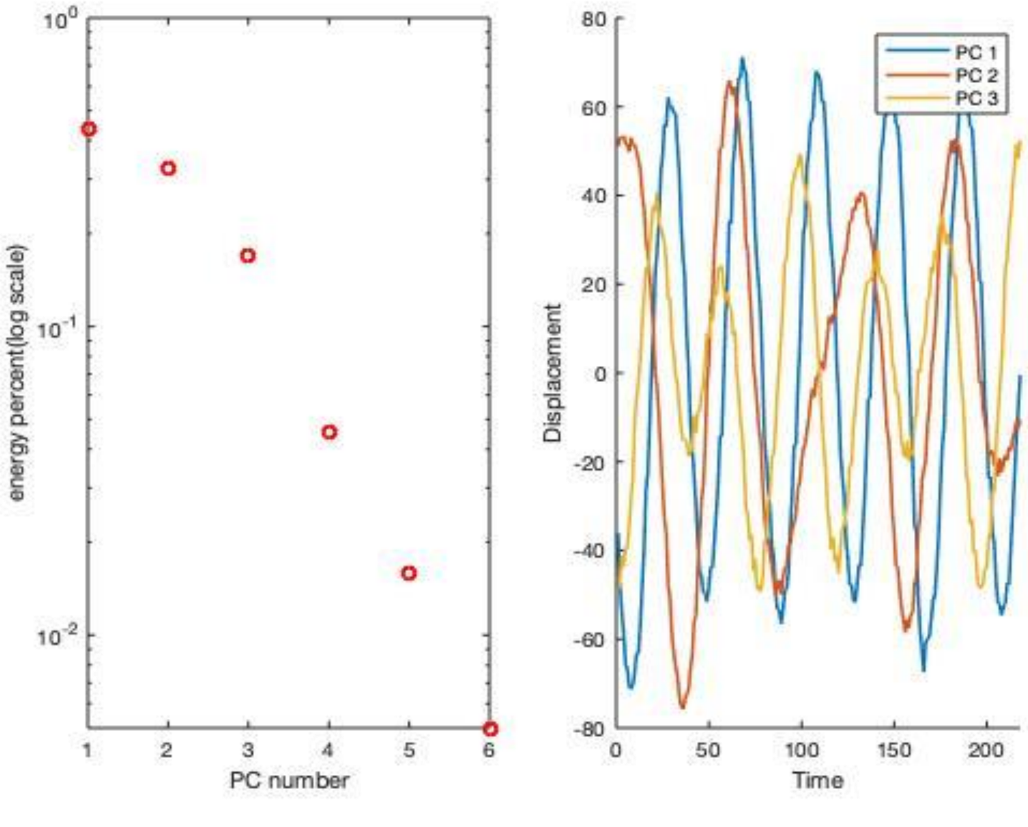

Figure 3: Case 3 principal component energy(left) and data projected on important principal component(right)

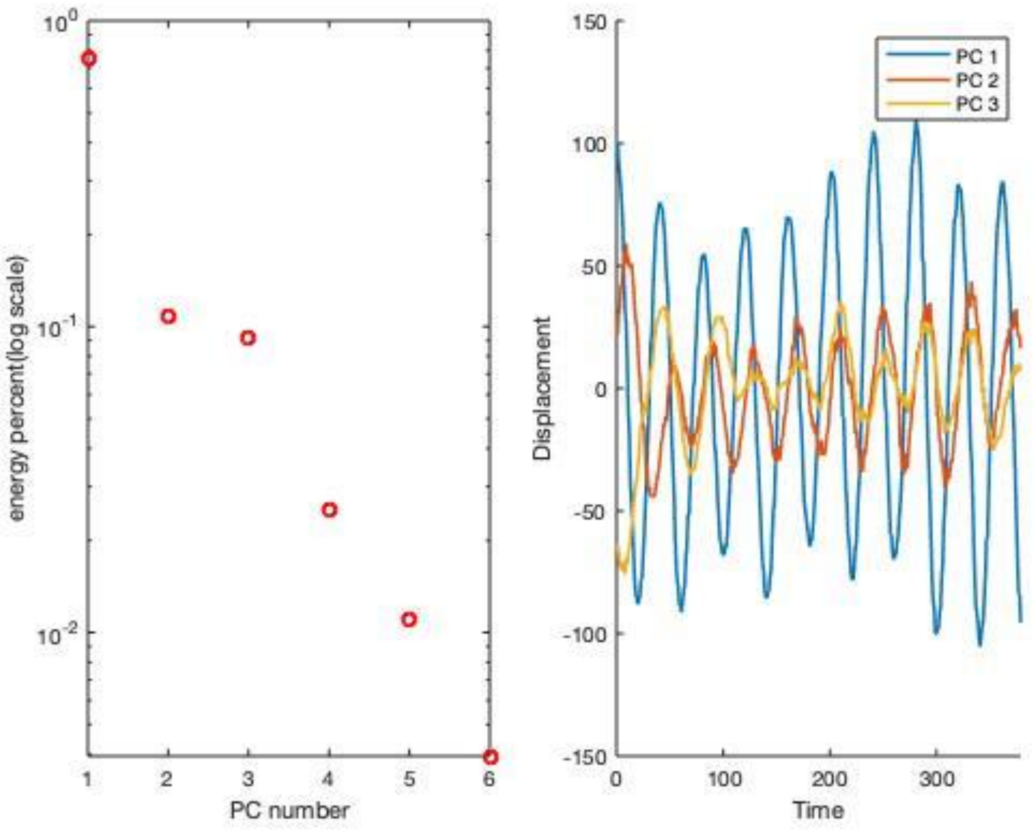

Figure 4: Case 4 principal component energy(left) and data projected on important principal component(right)

Case 1: ideal case, only oscillation in z direction

In figure 1, we see almost all the variation is in the first principal component. The second and the third principal component are worthless. Through the right graph, it is clear that the bucket only oscillates in one direction.

Case 2: add noise caused by camera shake to Case 1

The noise in this case lower the energy contained in the first principal component and increase energy in second and third component. However, the first principal component still takes much of the variance. From the right graph we can still observe primary oscillation in first principal component, although it is much noisier than that in Case 1.

Case 3: add horizontal displacement to Case 1

We can see variances in first, second and third principal components are all significant. In the right graph, we can tell the bucket oscillates in another direction. This is because the mass is released off-center, there is motion in the x − y plane.

Case 4: add horizontal displacement and rotation to Case 1

We can see first, second and third principal components have high energy. We also see another two dimensions of motion, which corresponds to motion in the x − y plane and rotation.

5. Summary and Conclusions

In this report we use Principal Component Analysis(PCA) to remove redundancy as well as noise and see which principal components are important in a spring and mass system. We successfully capture the major motion of the bucket by plotting our data projected onto important principal component bases

Appendix A.
MATLAB functions and brief implementation explanation

D = diag(v) returns a square diagonal matrix with the elements of vector v on the main diagonal.

k = find(X) returns a vector containing the linear indices of each nonzero element in array X.

[row,col] = ind2sub(sz,ind) returns the arrays row and col containing the equivalent row and column subscripts corresponding to the linear indices ind for a matrix of size sz. Here sz is a vector with two elements, where sz(1) specifies the number of rows and sz(2) specifies the number of columns.

I = rgb2gray(RGB) converts the truecolor image RGB to the grayscale image I.

[U,S,V] = svd(A,'econ') produces an economy-size decomposition of m-by-n matrix A: The economy-size decomposition removes extra rows or columns of zeros from the diagonal matrix of singular values, S, along with the columns in either U or V that multiply those zeros in the expression A = U*S*V'.

B = repmat(A,r1,...,rN) specifies a list of scalars, r1,..,rN, that describes how copies of A are arranged in each dimension.

Appendix B.
MATLAB codes

```
clear all; close all; clc;
%% Case1
load('cam1_1.mat')
load('cam2_1.mat')
load('cam3_1.mat')
data1_1= dataExtraction(vidFrames1_1, 170:430, 300:400, 250);
data2_1= dataExtraction(vidFrames2_1, 100:400, 225:355, 250);
data3_1= dataExtraction(vidFrames3_1, 200:350, 235:480, 247);
syncData1 = syncData(data1_1, data2_1, data3_1);
PCAplot(syncData1);

%% Case2
load('cam1_2.mat')
load('cam2_2.mat')
load('cam3_2.mat')
data1_2= dataExtraction(vidFrames1_2, 170:430, 300:450, 250);
data2_2= dataExtraction(vidFrames2_2, 50:475, 165:450, 249);
data3_2= dataExtraction(vidFrames3_2, 180:350, 235:500, 246);
syncData2 = syncData(data1_2, data2_2, data3_2);
PCAplot(syncData2);

%% Case3
load('cam1_3.mat')
load('cam2_3.mat')
load('cam3_3.mat')
data1_3= dataExtraction(vidFrames1_3, 225:450, 275:450, 250);
data2_3= dataExtraction(vidFrames2_3, 100:425, 165:415, 249);
data3_3= dataExtraction(vidFrames3_3, 160:365, 235:495, 246);
syncData3 = syncData(data1_3, data2_3, data3_3);
PCAplot(syncData3);

%% Case4
load('cam1_4.mat')
load('cam2_4.mat')
load('cam3_4.mat')
data1_4= dataExtraction(vidFrames1_4, 225:450, 275:470, 245);
data2_4= dataExtraction(vidFrames2_4, 50:400, 165:425, 249);
data3_4= dataExtraction(vidFrames3_4, 100:300, 270:505, 230);
syncData4 = syncData(data1_4, data2_4, data3_4);
PCAplot(syncData4);
%% functions
function data = dataExtraction(vidFrames, filter_y, filter_x, thresh)
    %Play video
    %implay(vidFrames);

    filter = zeros(480,640);
    filter(filter_y, filter_x) = 1;
    data = [];
    numFrames = size(vidFrames,4);
    %figure()
    for j=1:numFrames
        X=vidFrames(:,:,:,j);
        Xgrey = rgb2gray(X);
        Xgrey = double(Xgrey);
        Xf = Xgrey.*filter;
        threshold = Xf > thresh;
        indeces = find(threshold);
        [y, x] = ind2sub(size(threshold),indeces);
        data = [data, [mean(x); mean(y)]];
%       subplot(1,2,1)
%       imshow(uint8((threshold * 255)));
%       drawnow
%       subplot(1,2,2)
%       imshow(uint8(Xf)); drawnow
    end
end

function syncData = syncData(data1, data2, data3)
    [minimum, startIndex]= min(data1(2,1:20));
    data1 = data1(:, startIndex:end);
    [minimum, startIndex]= min(data2(2,1:20));
    data2 = data2(:, startIndex:end);
    [minimum, startIndex]= min(data3(1,1:20));
    data3 = data3(:, startIndex:end);
    len = min([length(data1) length(data2) length(data3)]);
    data1 = data1(:, 1:len);
    data2 = data2(:, 1:len);
    data3 = data3(:, 1:len);
    syncData = [data1;data2;data3];
end

function PCAplot(syncData)
    [m,n]=size(syncData);
    mn=mean(syncData,2);
    syncData =syncData-repmat(mn,1,n);
    [u,s,v]=svd(syncData/sqrt(n-1), 'econ');
    lambda=diag(s).^2;
    project= u'*syncData ;
    sig=diag(s);

    figure(1)
    clf;

    % Plot energy
    subplot(1,2,1)
    semilogy(lambda/sum(lambda), 'ro', 'linewidth', 2)
    xlabel('PC number')
    ylabel('energy percent(log scale)')

    % Plot Displacement in principal component direction
    subplot(1,2,2)
    hold on
    plot(1:length(project), project(1:3,:),'linewidth', 1.5)
    xlabel('Time')
    ylabel('Displacement')
    xlim([0, length(project)])
    legend('PC 1','PC 2', 'PC 3')
end
```