Image processing - SVD

Hyesu Lee

February 21, 2020

Abstract

Singular value decomposition(SVD) is important tool in data analysis. It allows us to analyze unknown data in low-dimensional system. This assignment is to practice principle component analysis(PCA), one of the SVD applications, with three given data sets of paint can movement. The student is expected to perform data analysis without using the knowledge of spring equation. The algorithm is implemented in MATLAB.

1 Introduction and Overview

With overflowing data, it is easy to collect data set with overlapping components. However, it would be unnecessary to maintain the redundant data. One of the property of singular value decomposition(SVD) is that it could be used to reduce the dimension to present the data. For this assignment, there are four different cases of object movement and for each case there are three different perspectives of the object. By using SVD, student is expected to perform principle component analysis(PCA). As a conclusion, the behaviors of principle components and energy conserved with those basis will be presented.

2 Theoretical Background

2.1 Singular Value Decomposition (SVD)

Singular value decomposition is a factorization of a matrix into a number of constitutive components all of which has a specific meaning in application: rotations and stretch/compression[2]. Suppose there is data, V , with two vector representation: v_1 and v_2. After applying some matrix A to the the data, it is now represented by two vector: $\sigma_1 u_1$, $\sigma_2 u_2$. Vector u_j is an orthogonal directional vector for the new representation of data and σ_j is a scaling factor of each u_j vector. For the simplicity of the example, let every u_j to have 2-norm of 1.
Now, let us consider data, V, to be in higher dimensional system. Let U and Σ to be the matrix of u_j vectors and σ_j accordingly. Σ is an n by n diagonal matrix with σ_j. Convention assumes that the singular values, σ_j, are ordered with descending order: $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n \geq 0$. According to theorem 2, there will be r nonzero σ, when r is rank of A. If A is m by n matrix where $m > n$, then at most n of the σ_j will be nonzero. Then the system is represented as equation 2 as matrix representation and equation 3.

$$Av_j = \sigma_j u_j \quad 1 \leq j \leq n \tag{1}$$

$$\begin{bmatrix} \\ A \\ \\ \end{bmatrix} \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix} \tag{2}$$

$$AV = U\Sigma \qquad\qquad A = U\Sigma V^* \tag{3}$$

There are two versions of SVD depends on the size of U and Σ: reduced singular value decomposition and standard singular value decomposition. For reduced SVD, matrix U is m by n matrix and Σ is a n by n diagonal matrix when A is m by n matrix, equation 4.

$$\begin{aligned} U &\in \mathbb{C}^{m*n} \\ V &\in \mathbb{C}^{n*n} \\ \Sigma &\in \mathbb{R}^{n*n} \end{aligned} \tag{4}$$

To make reduced SVD to standard SVD, U matrix get additional silent columns so that it becomes m by m unitary matrix. Σ, also, gets additional silent rows so that it becomes m by n diagonal matrix. Therefore, equation 5 is the dimensions of matrices in standard SVD. As shown, SVD explains the multiplication of matrix A with rotation, stretch/compression, and rotation.

$$\begin{aligned} U &\in \mathbb{C}^{m*m} \\ V &\in \mathbb{C}^{n*n} \\ \Sigma &\in \mathbb{R}^{m*n} \end{aligned} \tag{5}$$

U and V are unitary matrices while Σ is diagonal matrix. With SVD, it is explained that if the normalized eigenvectors are found for equation 6 and equation 7, then the orthonormal basis vectors are produced for U and V. These equations also suggests the relation between eigenvalue and singular values: square root of eigenvalues are singular values of the given equation.

$$\begin{aligned} A^TA &= (U\Sigma V^*)^T(U\Sigma V^*) \\ A^TA &= V\Sigma^2V^* \\ A^TAV &= V\Sigma^2 \end{aligned} \tag{6}$$

$$\begin{aligned} AA^T &= (U\Sigma V^*)(U\Sigma V^*)^T \\ AA^T &= U\Sigma^2U^* \\ AA^TU &= U\Sigma^2 \end{aligned} \tag{7}$$

This proof suggest the theorem 5 and the theorem produces unitary matrices U and V. When U and V are unitary matrices, a few theorems are derived. First, according to theorem 4, energy of singular value could be measured by 2-norm and Frobenius norm. Frobenius norm presents the total matrix energy while the 2-norm represents the energy of largest singular value. The ratio of these two norms could be used to measure the portion of the energy in each basis vectors, u_j. Second, the determinant of the matrix A is multiplication of all the singular values in matrix σ, theorem 7.

2.2 Eigenvalue decomposition

Eigenvalue problem is derived when the multiplication of a matrix and vector results multiplication of constant and the vector. The vector is called eigenvector and the constant is called eigenvalue.

$$Ax = \lambda x \tag{8}$$

In the eigenvalue problem, x is often an eigenvector and λ is often an eigenvalue. In order to find the eigenvectors and eigenvalues, the solution of equation 9 needs to be considered

$$Ax - \lambda Ix = (A - \lambda)x = 0 \tag{9}$$

For the nontrivial solution, two conditions should satisfy: $(A - I\lambda) = 0$ and $x \neq 0$. This promises the matrix, $(A - I\lambda)$, to be a singular and a non-invertible matrix. Eigenvalue decomposition does not guarantee the solution to the problem. If it does, then λ are chosen such that two restrictions of nontrivial solution satisfies.
If equation 10 is all the eigenvalues and eigenvectors of the given matrix, A, then the collection of eigenvalues

and eigenvectors could be represented as equation 11 where S is collection of eigenvectors and Λ is a diagonalized matrix with corresponding eigenvalues. Then, matrix A could be represented as equation 15

$$\begin{aligned} Ax_1 &= \lambda_1 x_1 \\ Ax_2 &= \lambda_2 x_2 \\ &\dots \\ Ax_n &= \lambda_n x_n \end{aligned} \tag{10}$$

$$AS = S\Lambda \tag{11}$$

$$A = S\Lambda S^{-1} \tag{12}$$

It is important to remember that Λ is diagonal matrix and, thus, eigenvalue decomposition is one of the ways to diagonalize a matrix.

2.3 Diagonalization

Diagonalization is the process of transforming a matrix into diagonal form.[1] As shown in section 2.2, eigenvalue decomposition diagonalizes the given matrix. SVD could diagonalize a matrix, as well, if the proper bases for the domain and range are used.[2] Let U and V to be orthonormal bases in $\mathbb{C}^{m*m}$ and $\mathbb{C}^{n*n}$ respectively. Then any vector within the space could be expressed as U and V with some weighting, $\hat{b}$ and $\hat{x}$.

$$\begin{aligned} b &\in \mathbb{C}^{m*m} \\ x &\in \mathbb{C}^{n*n} \\ b &= U\hat{b} \\ x &= V\hat{x} \end{aligned} \tag{13}$$

As shown in equation 14, matrix A is reduced to diagonal matrix Σ, when range is expressed in terms of basis vectors of U and the domain is expressed in terms of the basis vectors of V.[2]

$$\begin{aligned} Ax &= b \\ U^*b &= U^*Ax \\ U^*b &= U^*U\Sigma V^*x \\ \hat{b} &= \Sigma\hat{x} \end{aligned} \tag{14}$$

One of the properties of the diagonalization is the simplicity of calculation with exponents. For example, if eigenvalue decomposition is multiplied m times where m is some positive integer, then the computation of A^m could be simplified as equation 15. Since Λ is diagonal matrix, the diagonal entries will be λ_j^m, equation 16.

$$A^m = S\Lambda^m S^{-1} \tag{15}$$

$$\Lambda^m = \begin{bmatrix} \lambda_1^m & 0 & \dots & 0 \\ 0 & \ddots & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n^m \end{bmatrix} \tag{16}$$

2.4 Difference between SVD and eigenvalue decomposition

Although both SVD and eigenvalue decomposition diagonalize a given matrix, they are different.

- SVD performs the diagonalization using two different basis, U and V, while the eigenvalue decomposition uses a single basis, x.
- The SVD uses an orthonormal basis while the basis of eigenvalue decomposition is linearly independent but not necessarily orthogonal.
- SVD is guaranteed to exist for any matrix A while eigenvalue decomposition might not exist.

2.5 Principal Component Analysis (PCA)

The basic idea of principal component analysis is to extract principal components from unknown redundant data, so that the given data could be presented in simpler form. This process is also known as dimensionality reduction. Covariant is a statistical tool to find the redundancy between two data sets. Equation **??** is covariance matrix when the matrix X contains m data sets as rows and n data points. $\frac{1}{n-1}$ is for normalization.

$$C_X = \frac{1}{n-1} X X^T \tag{17}$$

This covariance matrix captures every possible correlations of the entries. It is a square and symmetric matrix whose diagonal values are the variance of particular measurements. The other values are covariances between measurement types. If the off-diagonal terms are small, then the measured quantities behaves independently and have low redundancy. Also, the large diagonal terms, or the large variances, represents the dynamics of interest. Therefore,to remove redundancy and identify principal component, the ideal covariance matrix should contain variance of decreasing order and zero for off-diagonal entries. Notice that this covariance matrix suggests the idea of diagonalization. Therefore, SVD and eigenvalue decomposition could be used for PCA. The diagonalization promises the ideal basis in which and the the covariance matrix could be described without redundancy. Further, the diagonalization promises order from the largest variances to the smallest. For the purpose of report, I will only elaborate usage of SVD for PCA.
As mentioned in section 2.1, SVD can diagonalize any matrix with appropriate bases, U and V. Thus, define the principal component projection as equation 18, where U is unitary transformation associated with SVD.

$$Y = U^* X \tag{18}$$

Then, the covariance matrix would be as equation 19.

$$\begin{aligned} C_Y &= \frac{1}{n-1} Y Y^T \\ &= \frac{1}{n-1} (U^* X)(U^* X)^T \\ &= \frac{1}{n-1} U^* (X X^T) U \\ &= \frac{1}{n-1} U^* U \Sigma^2 U U^* \\ C_Y &= \frac{1}{n-1} \Sigma^2 \end{aligned} \tag{19}$$

3 Algorithm Implementation and Development

The purpose of the assignment is to extract movement of paint can from three different camera perspective and analyze principal components and singular values by suing SVD. There are four different test cases to analyze: ideal case, noise case, horizontal displacement, and horizontal displacement and rotation. Following procedures are applicable for all the test cases.

1. Import movie of paint can movement.
2. For each image frame of each movie, change the colors into gray.
3. Find placements within the image frame where it is brighter than certain value and find the average x and y coordinates.
4. Save the x-y coordinates for every frame of the movie.
5. Plot three different perspective data sets to align the peaks and to trim so that all of them has same data points, image 1.
6. For each x and y axis, subtract the mean value for covariance computation.

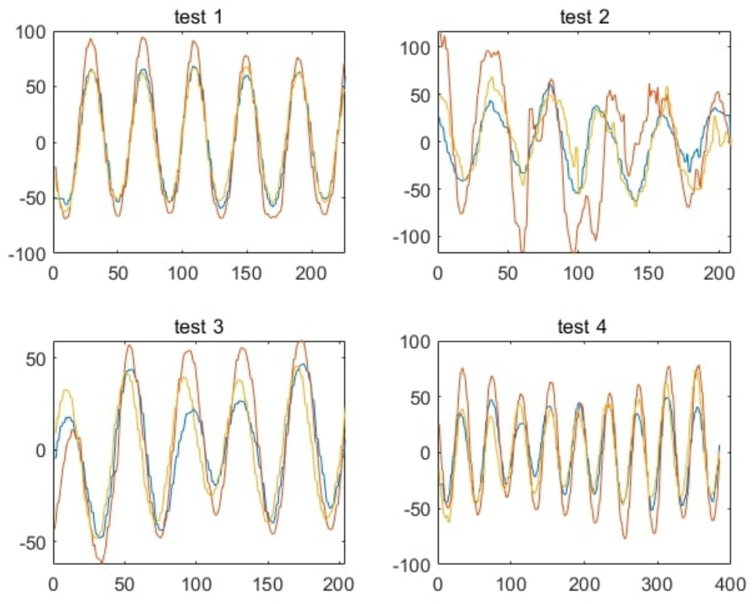

Figure 1: Aligned data set of four test cases

7. Build X matrix with 6 different perspective data sets as rows.
8. Build covariance matrix and use SVD to find three matrices: U, S, V.
9. Find the energy of singular values to determine the principal components
10. Plot the energy level of singular values and plot the graphs which shows the behavior of the paint can with principal components.

Lastly, there are two assumptions made for this project

- Assumption of linearity
- Assumption that larger variance represent more significant dynamics.

4 Computational Results

4.1 Test1

For test1, the movement of paint can only occurs on z-axis, so it is expected to have one singular value capturing most of the energy of the matrix such that one basis is enough to describe the behavior of the paint can. As shown in image 2, the first singular value contains 77.61% of total energy. The next two singular values contains significantly smaller energy compares to the first singular value. The first three singular values captures 95.32% of the total energy. Notice that in the second graph, basis 1 has constant oscillation which is the z-axis movement.

4.2 Test2

Test2, also, captures the same movement of test1 but with some noise. Due to the noise, the energy of matrix should be distributed to multiple singular values. In image 3, the second and third singular value contains more energy compares to test1. The first singular value contains 40.66% of total energy and the

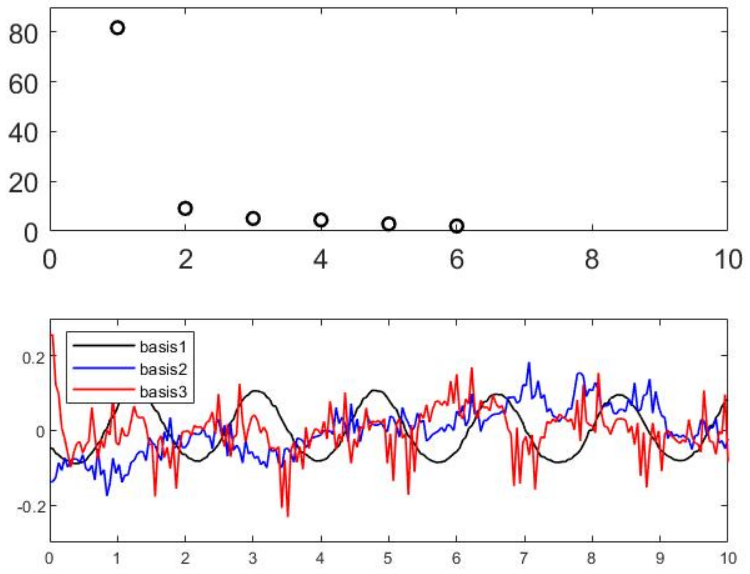

Figure 2: result for Test1(above: energy graph, below: the behavior of each basis)

first three singular values captures 77.84% of total energy. The second graph in image 3 shows that one basis is not enough to describe the movement of paint can and noise. The first basis, which corresponds to the first singular value, shows rough oscillation, but not as clear as test1 result.

4.3 Test3

Test3 videos contain the z-axis oscillation movement and some motion in x-y plane. Therefore, at least two singular values should contain majority of total energy. In image 4, the first two singular values have significantly greater energy compares to other singular values. The first two singular values capture 75.15% of the total energy. From the second graph of image 4, the basis 1 captures the z-axis movement while the basis 2 captures most of the x-y plane movement.

4.4 Test4

The test4 videos capture horizontal displacement and rotation movement along with z-axis oscillation. In image 5, the first singular value captures most of the energy, 52.49%, but second and the third contains some such that the first three singular value captures 82.91% of total energy. In the behavior graph of image 5, the first basis captures the z-axis oscillation and the other two basis shows the horizontal displacement and rotational movement. The the periodic movement shown by basis3 seems to capture horizontal rotational moment..

5 Summary and Conclusions

By using SVD, it was possible to extract the principle component of the redundant data set and perform dimensionality reduction. Further, with four different test cases, it was possible to observe different energy distributions within the singular values and how motion is described with principle components.

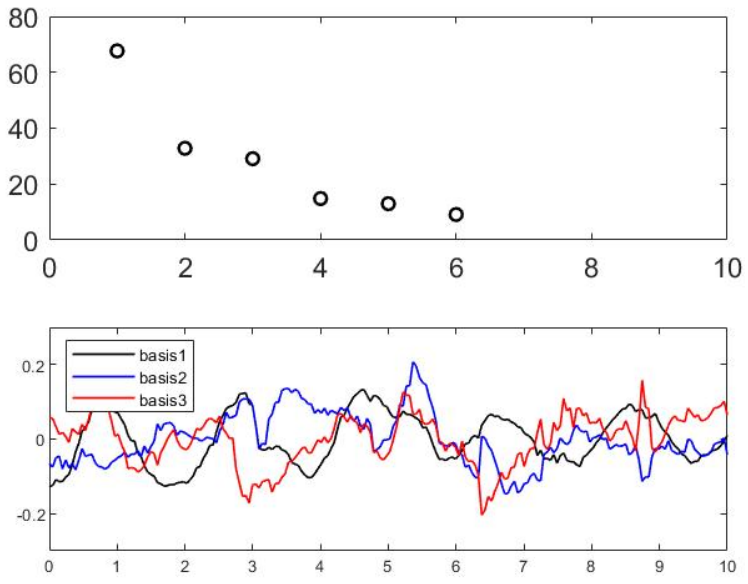

Figure 3: result for Test2(above: energy graph, below: the behavior of each basis)

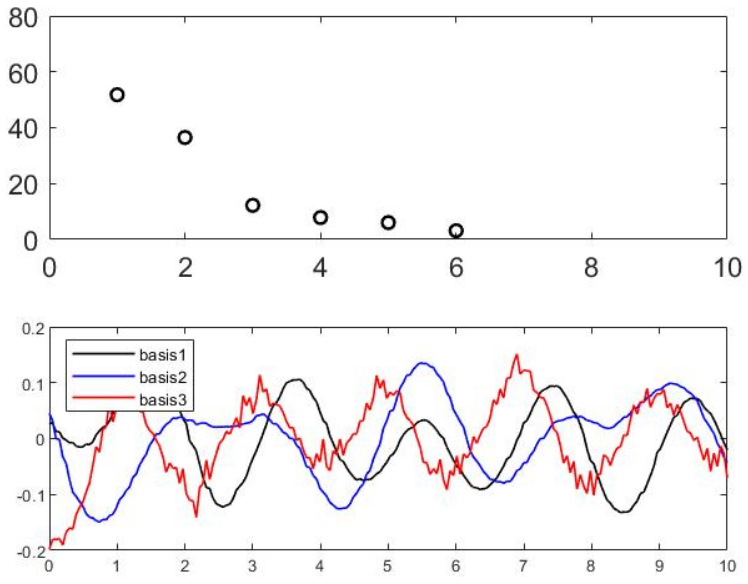

Figure 4: result for Test3(above: energy graph, below: the behavior of each basis)

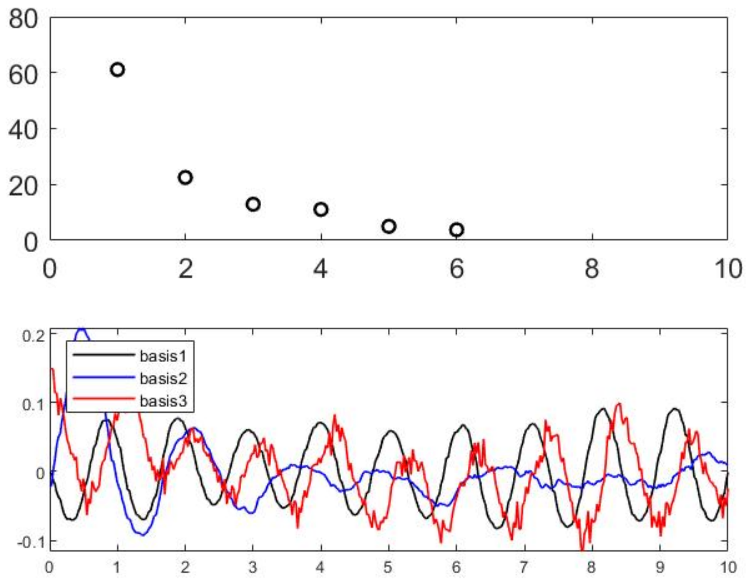

Figure 5: result for Test4(above: energy graph, below: the behavior of each basis)

References

[1] Paul Bohan-Broderick. *Diagonalization: Definition Example*. 2020. URL: `https://study.com/academy/lesson/diagonalization-definition-example.html/`.

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

Appendix A Theorems [2]

- **Theorem 1** *Every matrix $A \in \mathbf{C}^{m*n}$ has a singular value decomposition. Furthermore, the singular values σ_j are uniquely determined, and, if A is square and the σ_j distinct, the singular vectors u_j and v_j are uniquely determined up to complex signs (complex scalar factors of absolute value 1).*
- **Theorem 2** *If the rank of A is r, then there are r nonzero singular values. If $A{=}U\Sigma V^*$ and U and V are full rank, then rank of A is number of σ, number of singular values.*
- **Theorem 3** *The range$(A) =< u_1, u_2, u_3, ..., u_r >$ and null$(A) =< v_{r+1}, v_{r+2}, v_{r+3}, ..., v_n >$.*
- **Theorem 4** *The norm $||A||_2 = \sigma_1$ and $||A||_F = \sqrt{\sigma_1^2 + \sigma_2^2 + \sigma_3^2 + ... + \sigma_r^2}$.*
- **Theorem 5** *The nonzero singular values of A are the square roots of the nonzero eigenvalues of $A*A$ or $AA*$.*
- **Theorem 6** *If $A = A^*$, then the singular values of A are the absolute values of the eigenvalues of A.*
- **Theorem 7** *For $A \in \mathbf{C}^{m*m}$, the determinant is given by $|det(A)| = \prod_{j=1}^{m} \sigma_i$.*
- **Theorem 8** *A is the sum of r rank-one matrices.*
- **Theorem 9** *For any N so that $0 \geq N \geq r$, we can define the partial sum: $A_N = \sum_{j=1}^{N} \sigma_j u_j v_j^*$. If $N = min(m,n)$, define $\sigma_{n+1} = 0$, then. Likewise, if using the Frobenius-norm, then $||A - A_N||_N = \sqrt{\sigma_{N+1}^2 + \sigma_{N+2}^2 + ... + \sigma_r^2}$.*

Appendix B MATLAB Functions

Followings are the MATLAB Functions used for the project.

- `B = reshape(A, sz)` return reshaped matrix `A`, using the size vector `sz` for size of matrix `B`.
- `M = mean(A)` return the mean value of the elements in A when A is a vector
- `imshow(A)` display the image A.
- `D = size(X)` for M by N matrix X, returns the two-element row vector
- `[U,S,V] = svd(X, 'econ')` produces a diagonal matrix S and unitary matrices U and V so that $X = U * S * V'$. With 'econ' parameter, executes the reduced SVD.
- `lambda = diag(S)` returns the diagonal entries of matrix S as vector.
- `S = sum(X)` returns the sum of the elements of the vector X.
- `y = linspace(x1,x2, n)` returns a row vector of `n` evenly spaced points between `x1` and `x2`.
- `plot(X,Y)` plots vector Y versus vector X.
- `imggray = rgb2gray(img)` converts RGB image or color map to gray scale and returns the changed image.
- `idx=find(test)` returns the linear indices corresponding to the nonzero entries which satisfies the given test.
- `[y, x] = ind2sub(sz, ind)` returns three arrays containing equivalent multidimensional subscripts corresponding to the linear indices ind for a multidimensional array of size sz. Since the return value format is [row,col], assignment for x-axis and y-axis should be ordered as [y-axis, x-axis].
- `[X,Y] = meshgrid(x,y)` returns 2-D grid coordinates based on the coordinates contained in the vectors `x` and `y`. X is a matrix where each row is a copy of `x`, and `Y` is a matrix where each column is a copy of `y`. The grid represented by the coordinates `X` and `Y` has `length(y)` rows and `length(x)` columns.

Appendix C MATLAB Code

```
clc; close all; clear all;
%
load('cam1_1.mat'); load('cam2_1.mat'); load('cam3_1.mat');
load('cam1_2.mat'); load('cam2_2.mat'); load('cam3_2.mat');
load('cam1_3.mat'); load('cam2_3.mat'); load('cam3_3.mat');
load('cam1_4.mat'); load('cam2_4.mat'); load('cam3_4.mat');

size_camera_1 = size(vidFrames1_1);
placement = [0,0];
x_points = size_camera_1(2);
y_points = size_camera_1(1);
x=linspace(-x_points, x_points);
y=linspace(-y_points, y_points);
[X,Y]=meshgrid(x,y);

placement1_1 = track_can1(vidFrames1_1);
placement2_1 = track_can2(vidFrames2_1); % out of the focus
placement3_1 = track_can3(vidFrames3_1); % tracks the brightest which is the reflection of the board
```

```
% Cropping the data so that every data has same length
placement1_1 = placement1_1(2:227, :);
placement2_1 = placement2_1(12:237, :);
placement3_1 = placement3_1(2:227, :);

%% saving the x value and y value for each camera (test 1)
y_axis = placement1_1(:,1) - mean(placement1_1(:,1));
x_axis = placement1_1(:,2) - mean(placement1_1(:,2));
save test1_camera1_y.dat y_axis -ascii;
save test1_camera1_x.dat x_axis -ascii;
subplot(2,2,1)
plot(y_axis);
hold on
y_axis = placement2_1(:,1) - mean(placement2_1(:,1));
x_axis = placement2_1(:,2) - mean(placement2_1(:,2));
save test1_camera2_y.dat y_axis -ascii;
save test1_camera2_x.dat x_axis -ascii;
plot(y_axis);
hold on
y_axis = placement3_1(:,2) - mean(placement3_1(:,2));
x_axis = placement3_1(:,1) - mean(placement3_1(:,1));
save test1_camera3_y.dat y_axis -ascii;
save test1_camera3_x.dat x_axis -ascii;
plot(y_axis);
hold on
title('test 1')
%% test2

% make sure to change cropping value for better data
placement1_2 = track_can1(vidFrames1_2);
placement2_2 = track_can2(vidFrames2_2);
placement3_2 = track_can3(vidFrames3_2);

% Cropping the data so that every data has same length
placement1_2 = placement1_2(18:225, :);
placement2_2 = placement2_2(2:209, :);
placement3_2 = placement3_2(20:227, :);

%% saving the x value and y value for each camera (test 2)
y_axis = placement1_2(:,1) - mean(placement1_2(:,1));
x_axis = placement1_2(:,2) - mean(placement1_2(:,2));
subplot(2,2,2)
plot(y_axis)
hold on
save test2_camera1_y.dat y_axis -ascii;
save test2_camera1_x.dat x_axis -ascii;
y_axis = placement2_2(:,1) - mean(placement2_2(:,1));
x_axis = placement2_2(:,2) - mean(placement2_2(:,2));
plot(y_axis)
hold on
save test2_camera2_y.dat y_axis -ascii;
save test2_camera2_x.dat x_axis -ascii;
y_axis = placement3_2(:,2) - mean(placement3_2(:,2));
x_axis = placement3_2(:,1) - mean(placement3_2(:,1));
```

```
plot(y_axis)
save test2_camera3_y.dat y_axis -ascii;
save test2_camera3_x.dat x_axis -ascii;
title('test 2')

%% test3

% make sure to change cropping value for better data
placement1_3 = track_can1(vidFrames1_3);
placement2_3 = track_can2(vidFrames2_3);
placement3_3 = track_can3(vidFrames3_3);

% Cropping the data so that every data has same length
placement1_3 = placement1_3(7:210, :); %2
placement2_3 = placement2_3(34:237, :);%29
placement3_3 = placement3_3(2:205, :);

%% saving the x value and y value for each camera (test 3)
y_axis = placement1_3(:,1) - mean(placement1_3(:,1));
x_axis = placement1_3(:,2) - mean(placement1_3(:,2));
save test3_camera1_y.dat y_axis -ascii;
save test3_camera1_x.dat x_axis -ascii;
subplot(2,2,3)
plot(y_axis)
hold on
y_axis = placement2_3(:,1) - mean(placement2_3(:,1));
x_axis = placement2_3(:,2) - mean(placement2_3(:,2));
save test3_camera2_y.dat y_axis -ascii;
save test3_camera2_x.dat x_axis -ascii;
plot(y_axis)
hold on
y_axis = placement3_3(:,2) - mean(placement3_3(:,2));
x_axis = placement3_3(:,1) - mean(placement3_3(:,1));
save test3_camera3_y.dat y_axis -ascii;
save test3_camera3_x.dat x_axis -ascii;
plot(y_axis)
hold on
title('test 3')
%% test4

% make sure to change cropping value for better data
placement1_4 = track_can1(vidFrames1_4);
placement2_4 = track_can2(vidFrames2_4);
placement3_4 = track_can3(vidFrames3_4);

% Cropping the data so that every data has same length
placement1_4 = placement1_4(2:385, :); %2
placement2_4 = placement2_4(7:390, :);%29
placement3_4 = placement3_4(2:385, :);

%% saving the x value and y value for each camera (test 4)
y_axis = placement1_4(:,1) - mean(placement1_4(:,1));
x_axis = placement1_4(:,2) - mean(placement1_4(:,2));
subplot(2,2,4)
```

```
plot(y_axis)
hold on
save test4_camera1_y.dat y_axis -ascii;
save test4_camera1_x.dat x_axis -ascii;
y_axis = placement2_4(:,1) - mean(placement2_4(:,1));
x_axis = placement2_4(:,2) - mean(placement2_4(:,2));
plot(y_axis)
hold on
save test4_camera2_y.dat y_axis -ascii;
save test4_camera2_x.dat x_axis -ascii;
y_axis = placement3_4(:,2) - mean(placement3_4(:,2));
x_axis = placement3_4(:,1) - mean(placement3_4(:,1));
plot(y_axis)
hold on
save test4_camera3_y.dat y_axis -ascii;
save test4_camera3_x.dat x_axis -ascii;
title('test 4 ')
%%
load('cam2_1.mat')
placement2 = track_can1(vidFrames1_1);
for i = 1:314
   imshow(vidFrames1_1(:,:,:,i)), hold on
   %imshow(vidFrames2_1(:,:,:,i)), hold on
   %imshow(vidFrames3_1(:,:,:,i)), hold on
   plot(placement2(i+1,2),placement2(i+1,1),'ro','MarkerSize',10,'MarkerFaceColor','r')
   %plot(placement2(i+1,2),placement2(i+1,1),'ro','MarkerSize',10,'MarkerFaceColor','k')
   %plot(placement3(i+1,2),placement3(i+1,1),'ro','MarkerSize',10,'MarkerFaceColor','b')
   hold off
   pause(0.1)
end

%% function to track paint can with camera 1
function placement = track_can1(vidFrames)
    size_camera = size(vidFrames);
    placement = [0,0];
    for i = 1:size_camera(4)
        img = vidFrames(200:450, 50:500,:,i); % for test1,2,3, 4
        sumed_img = rgb2gray(img);
        idx = find(sumed_img(:) > 240); % return anything that is above the 250 value
        [row, col] = ind2sub(size(sumed_img),idx);
        row = mean(row(:));
        col = mean(col(:));
        placement = [placement; 200 + row, 50 + col];
    end
    save camera1_1.dat placement -ascii;
end
%% function to track paint can with camera 2
function placement = track_can2(vidFrames)
    size_camera = size(vidFrames);
    placement = [0,0];
    for i = 1:size_camera(4)
        img = vidFrames(140:440, 70:417,:,i); % for test 4
        %img = vidFrames(90:420, 120:510,:, i);% for test 2
        %img = vidFrames(140:415, 180:490, :, i); %for test 3
        %img = vidFrames(190:450, 60:420,:,i); % for test 1


        sumed_img = rgb2gray(img);
        idx = find(sumed_img(:) > 240); % return anything that is above the 250 value
        [row, col] = ind2sub(size(sumed_img),idx);
        row = mean(row(:));
        col = mean(col(:));
        % remember to change the constant value according to the test
        placement = [placement; 140 + row, 70 + col];
    end
    %placement = placement(2:end, :);
    save camera2_1.dat placement -ascii;
end

%% function to track paint can with camera 3
function placement = track_can3(vidFrames)
    size_camera = size(vidFrames);
    placement = [0,0];
    for i = 1:size_camera(4)
        img = vidFrames(130:330,220:570,:,i); % for test 4
        %img = vidFrames(160:460, 240:620, :, i); %for test 2 / for test 3
        %img = vidFrames(10:480,190:430,:,i); % for test 1

        sumed_img = rgb2gray(img);
        idx = find(sumed_img(:) > 230);
        % test 1, 2, 3, 240
        % return anything that is above the 250 value
        [row, col] = ind2sub(size(sumed_img),idx);
        row = mean(row(:));
        col = mean(col(:));
        placement = [placement; 130 + row, 220 + col];
    end
    %placement = placement(2:end, :);
    save camera3_1.dat placement -ascii;
end

%%PCA
clc; close all; clear all;
% Plot singular values
load test1_camera1_x.dat;load test1_camera1_y.dat;
load test1_camera2_x.dat;load test1_camera2_y.dat;
load test1_camera3_x.dat;load test1_camera3_y.dat;

X = [test1_camera1_x';
     test1_camera1_y';
     test1_camera2_x';
     test1_camera2_y';
     test1_camera3_x';
     test1_camera3_y'
     ];

 [m,n] = size(X);
 [U,S,V] = svd(X'/sqrt(n-1),'econ');
 %(X, 'econ');%

 sig = diag(S);
 energy1 = sig(1)/sum(sig);
 energy2 = sum(sig(1:2))/sum(sig);
 energy3 = sum(sig(1:4))/sum(sig);
 subplot(2,1,1)
 plot(sig, 'ko', 'Linewidth', [1.5])
 axis([0 10 0 90])
 set(gca, 'Fontsize', [13], 'Xtick', [0 2 4 6 8 10 ])

 x = linspace(0,10,226);
 subplot(2,1,2)
 plot(x, U(:,1), 'k',x, U(:,2), 'b',x, U(:,3), 'r', 'Linewidth', [1])
 set(gca, 'Fontsize', [5])
 legend('mode1', 'mode2', 'mode3', 'Location', 'NorthWest')


% Identify the numbers of singular values
% calculate energy and select the singular values
% plot linear POD modes
% Plot time evolution behavior
%% test2

load test2_camera1_x.dat;load test2_camera1_y.dat;
load test2_camera2_x.dat;load test2_camera2_y.dat;
load test2_camera3_x.dat;load test2_camera3_y.dat;

X = [test2_camera1_x';
     test2_camera1_y';
     test2_camera2_x';
     test2_camera2_y';
     test2_camera3_x';
     test2_camera3_y'
     ];

 [m,n] = size(X);
 [U,S,V] = svd(X'/sqrt(n-1),'econ');

 sig = diag(S);
 energy1 = sig(1)/sum(sig);
 energy2 = sum(sig(1:2))/sum(sig);
 energy3 = sum(sig(1:3))/sum(sig);
 energy4 = sum(sig(1:4))/sum(sig);
 subplot(2, 1, 1), plot(sig, 'ko', 'Linewidth', [1.5])
 axis([0 10 0 80])
 set(gca, 'Fontsize', [13], 'Xtick', [0 2 4 6 8 10 ])

 x = linspace(0,10,208);
 subplot(2,1,2)
 plot(x, U(:,1), 'k',x, U(:,2), 'b',x, U(:,3), 'r', 'Linewidth', [1])
 set(gca, 'Fontsize', [5])
 legend('mode1', 'mode2', 'mode3', 'Location', 'NorthWest')
```

```
%% test 3
clc; clear all; close all;
load test3_camera1_x.dat;load test3_camera1_y.dat;
load test3_camera2_x.dat;load test3_camera2_y.dat;
load test3_camera3_x.dat;load test3_camera3_y.dat;

X = [test3_camera1_x';
     test3_camera1_y';
     test3_camera2_x';
     test3_camera2_y';
     test3_camera3_x';
     test3_camera3_y'
     ];

 [m,n] = size(X);
 [U,S,V] = svd(X'/sqrt(n-1),'econ');

 sig = diag(S);
 energy1 = sig(1)/sum(sig);
 energy2 = sum(sig(1:2))/sum(sig);
 energy3 = sum(sig(1:3))/sum(sig);
 energy4 = sum(sig(1:4))/sum(sig);
 subplot(2, 1, 1), plot(sig, 'ko', 'Linewidth', [1.5])
 axis([0 10 0 80])
 set(gca, 'Fontsize', [13], 'Xtick', [0 2 4 6 8 10 ])


 x = linspace(0,10,204);
 subplot(2,1,2)
 plot(x, U(:,1), 'k',x, U(:,2), 'b',x, U(:,3), 'r', 'Linewidth', [1])
 set(gca, 'Fontsize', [5])
 legend('mode1', 'mode2', 'mode3', 'Location', 'NorthWest')
%% test 4
clc; clear all; close all;
load test4_camera1_x.dat;load test4_camera1_y.dat;
load test4_camera2_x.dat;load test4_camera2_y.dat;
load test4_camera3_x.dat;load test4_camera3_y.dat;

X = [test4_camera1_x';
     test4_camera1_y';
     test4_camera2_x';
     test4_camera2_y';
     test4_camera3_x';
     test4_camera3_y'
     ];

 [m,n] = size(X);
 [U,S,V] = svd(X'/sqrt(n-1),'econ');

 sig = diag(S);
 energy1 = sig(1)/sum(sig);
 energy2 = sum(sig(1:2))/sum(sig);
 energy3 = sum(sig(1:3))/sum(sig);
 energy4 = sum(sig(1:4))/sum(sig);
```

```
subplot(2, 1, 1), plot(sig, 'ko', 'Linewidth', [1.5])
axis([0 10 0 80])
set(gca, 'Fontsize', [13], 'Xtick', [0 2 4 6 8 10 ])

x = linspace(0,10,384);
subplot(2,1,2)
plot(x, U(:,1), 'k',x, U(:,2), 'b',x, U(:,3), 'r', 'Linewidth', [1])
set(gca, 'Fontsize', [5])
legend('mode1', 'mode2', 'mode3', 'Location', 'NorthWest')
```

Appendix D Github page

https://github.com/HyesLee99/Data-analysis/tree/master/HW3