

PCA

Andrei Karavanov

Tuesday, Feb. 26, 2019

Abstract

In this paper I will explore different scenarios when one could use principal component analysis to get an insight into the state of system.

Introduction and Overview

Imagine we perform a scientific experiment. In it we use three cameras, which are placed at the different angles, to record motion of the mass-spring system. Our main goal is to use this data to determine a governing equation of motion. Since the data from each camera records the same mass, in our case can, we would expect to see a lot of redundant information. In this paper we will discuss four possible cases:

1. **Ideal Case:** Can moves only in z direction.
2. **Noisy Case:** Repeat ideal case, but introduce camera shake.
3. **Horizontal Displacement Case:** Can is released off-center so there will be motion both in the $x - y$ plane and z direction.
4. **Horizontal Displacement and Rotation Case:** Repeat horizontal displacement case, but introduce rotation.

Since we need at the max two sets of data as the output: z direction and the $x - y$ plane, we will approach this problem by using PCA. Specifically we will discuss whether low-dimensional reductions are appropriate to asses the dynamics of the mass-spring system.

Theoretical Background and Algorithm Implementation

From the introductory physics class about oscillations, we know that the governing equations for mass-spring system are:

$$\frac{d^2 f(t)}{dt^2} = -\omega^2 f(t) \tag{1}$$

If we will now solve this system, we will get the following solution:

$$f(t) = A \cos(\omega t + \omega_0) \quad (2)$$

which is a one-degree of freedom system. This solution is based on assumption that there is no movement in $x - y$ direction. This, however contradicts with case three and case four. Thus for cases three and four we don't know the governing equation prior the analysis of data. We however will assume that the result will be a two-degree of freedom system, since we would expect to see movement both in z direction and the $x - y$ plane.

Altogether, if we are correct with our assumptions and our governing equations are indeed one-degree of freedom and two-degrees of freedom systems, after applying PCA, we will get a low-dimensional systems, which will describe the motion of the system.

Now let us talk about our algorithmic approach.

First of all, let us talk about data ordering. In each case we will deal with three recorded footage. Each footage will consist of n number of frames. Each frame is represented by an RGB image in 2-D. That means our initial data set for each case is three 4-D matrices. Since it is not very practical to work with such structure we will use the following sudo-code to reorganize all of our data into a separate 2-D matrices:

Get the minimum amount of frames

For each video:

For each frame:

Move the frame from RGB to Grayscale

Cast each value to a double

Resize to the same size

Reshape into array

Store in data matrix

Now when we have a well structured matrices of each frame of each video, let us determine the location of the can at each frame:

For each video matrix:

For each row:

For each pixel:

If pixel is in the central region and is brighter than 240:

Store its coordinates in a matrix

Find average brightest point for each frame and store its coordinates in vectors

Return vectors of X and Y coordinates of the average brightest point for each frame for each video

Now, when we have the matrix of x and y coordinated (in pixels) of the can in each frame

in each video, we organize that data in the following matrix:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{bmatrix} \quad (3)$$

Now when our data has been organized, let us tackle our main goals: redundancy and noise (case two). To asses, which data is redundant we will use the co-variance matrix:

$$\mathbf{C}_X = \frac{1}{n-1} \mathbf{X} \mathbf{X}^T \quad (4)$$

where $\frac{1}{n-1}$ is our unbiased estimator. Each non-zero element located not on the diagonal shows that there exists redundancy in our data. Since, we want to eliminate this redundancy we want to diagonalize our co-variance matrix. Surprisingly enough, Singular Value Decomposition does exactly that. Here is how it works. Since our co-variance matrix is square matrix we can use eigenvalues decomposition to get the following:

$$\mathbf{X} \mathbf{X}^T = \mathbf{Q} \mathbf{\Sigma} \mathbf{Q}^{-1} \quad (5)$$

Where \mathbf{Q} is the matrix of eigenvectors and $\mathbf{\Sigma}$ is a diagonal matrix that represents distinct eigenvalues of \mathbf{C}_X . Since our co-variance matrix it is symmetric, the eigenvector columns should be orthogonal. In other words, it follows that $\mathbf{Q}^1 = \mathbf{Q}^T$. That means we can rewrite our equation five as the following:

$$\mathbf{X} \mathbf{X}^T = \mathbf{Q} \mathbf{\Sigma} \mathbf{Q}^T \quad (6)$$

These two facts suggest that we can try to work in the principal component basis.

$$\mathbf{Y} = \mathbf{Q}^T \mathbf{X} \quad (7)$$

For this new basis, we can then consider its co-variance:

$$\begin{aligned} \mathbf{C}_Y &= \frac{1}{n-1} \mathbf{Y} \mathbf{Y}^T = \frac{1}{n-1} (\mathbf{Q}^T \mathbf{X}) (\mathbf{Q}^T \mathbf{X})^T \\ &= \frac{1}{n-1} \mathbf{Q}^T (\mathbf{X} \mathbf{X}^T) \mathbf{Q} = \frac{1}{n-1} \mathbf{Q}^T \mathbf{Q} \mathbf{\Sigma} \mathbf{Q}^T \mathbf{Q} \\ &= \frac{1}{n-1} \mathbf{\Sigma} \end{aligned}$$

As we can see in principal component basis our co-variance matrix is diagonal. That means that all non-diagonal elements are zero, which means that we do not have any redundant information left! That is exactly what we wanted. The only thing left is to determine what information is important and what is not important. We can do that by looking at the singular values. The greater singular value, the greater "change" given principal component

represent. In other words, if we will now find significantly important singular values and then will project our original data matrix on corresponding principal component vectors, we will end up with set of vectors that represent the projections of our data onto the dominant directions.

We will implement PCA using the following pseudo-code:

De-mean our \mathbf{X} matrix: subtract the mean of each row from each element of that row

Create co-variance matrix

Find eigenvalues and eigenvectors of our co-variance matrix

Sort our eigenvalues in decreasing order

Project our original data onto the principal component basis

Plot either one (cases one and two) or two (cases three and four) projection vectors that correspond to one/two largest singular values. singular values.

Computational Results

1. **Ideal Case:** Looking at figure one we can identify that first singular value dominates the rest by two orders of magnitude. That means that the most of motion in the mass-spring system can be represented by the projection of our original data onto that principal component. The rest of singular values tend to be less significant. In the ideal case we would expect to see projections on the rest principal components to be equal to zero, since we are dealing with a one-degree of freedom system. In our case however, we can see that even though the rest of singular values are quite small they are not equal to zero. Same can be said about principal components. From the figure one we can see that even though second (blue), third (green) and fourth (yellow) principal components are not zero they do not provide any useful information about our system. The first principal component (red), in contrast, clearly shows that our system is experiencing simple harmonic oscillations.

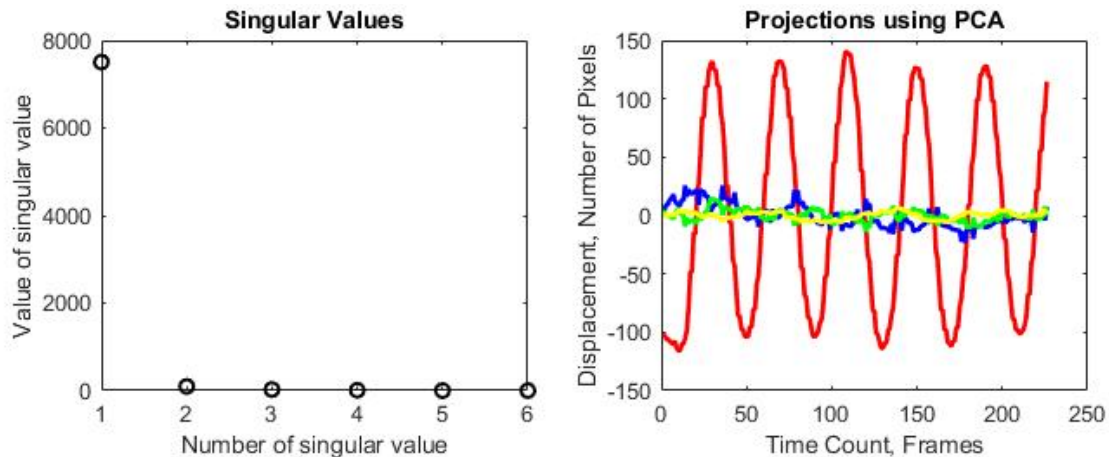


Figure 1: Results for the Ideal Case. Singular Values: [7503.5, 95.8, 24.9, 9.2, 0, 0]

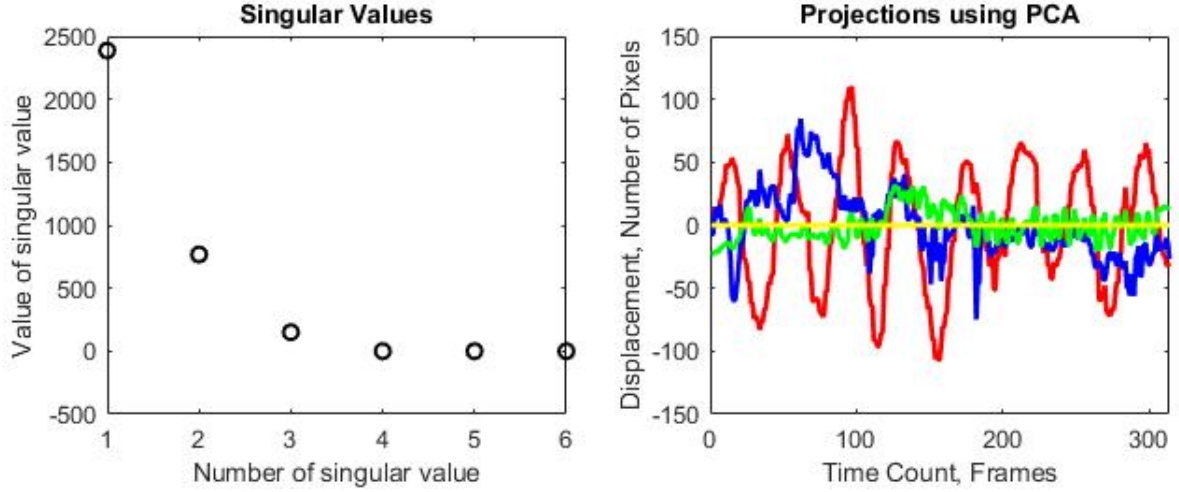


Figure 2: Results for the Noisy Case. Singular Values: [2386.6, 767.3, 150.5, 0, 0, 0]

2. **Noisy Case:** Unlike case one, case two introduces noise to our data. As we can see from the figure two, that leads to a different singular values as well as different projections. In this case we can see that the gap between the largest singular value and the second largest is not as drastic as it was in the case one. That comes from the fact that our algorithm had hard time finding redundant information in our initial data matrix. Nonetheless, if we will look at the vector projection on the first (red) principal component we will still see traits of harmonic motion. The rest of projections represent the noise that was coming from the shaking camera, and thus can be ignored.
3. **Horizontal Displacement Case:** In this case we release the can off-center, therefore we expect to see motion both in the $x - y$ plane and z direction. If we will look at the singular values plot on figure three we will see two singular values that clearly dominate the rest - first and second. It is important to notice that the greatest singular value is still five times greater than the second greatest. This behaviour can be explained by the fact that there was more motion in z direction rather motion in the $x - y$ plane. The rest singular values all stay around zero, which means that they are not significantly important. The same can be seen from the projection plot of figure three as well. We can clearly see how two projection on first (red) and second (blue) principal component look like oscillatory functions. The projections on third (green) and fourth (yellow) principal components, on the other hand, show no sign of oscillation and look closer to a constant line $y = 0$. Based on our observation of the projection behaviour, we indeed can conclude that given system is indeed a two-degree of freedom system.
4. **Horizontal Displacement and Rotation Case:** In this case we release the can off-center and give it some rotation. This case is similar to case three, except we will try to identify the rotational motion. From the singular values plot of figure four we can see that two singular values clearly dominate the rest - first and second. Similarly to previous case, we can notice that the greatest singular value is significantly greater than the second greatest. This behaviour, as well, can be explained by the fact that there

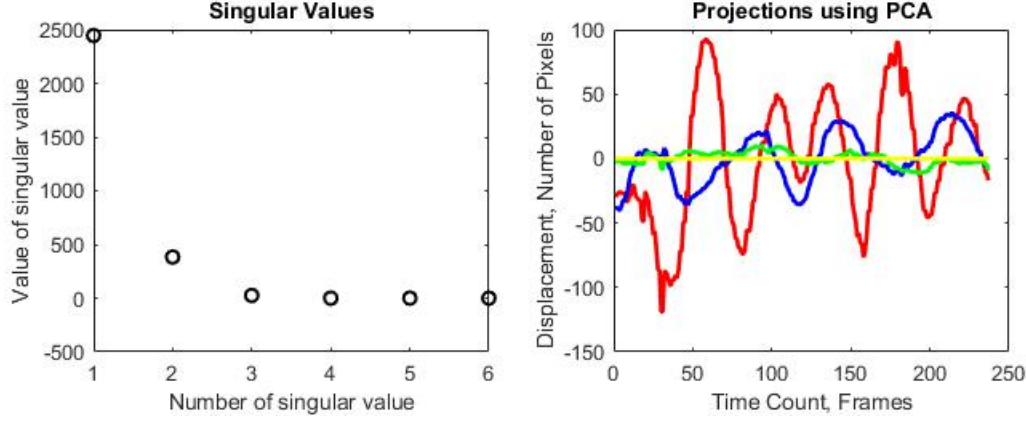


Figure 3: Results for the Horizontal Displacement Case. Singular Values: [2445.8, 383.0, 25.4, 0, 0, 0]

was more motion in z direction rather motion in the $x - y$ plane. The rest singular values all stay around zero, which means that they are not significantly important. Now, if we will look at the projection plot of figure four, we will clearly see how two projection on first (red) and second (blue) principal component look like oscillatory functions. Just like in the previous case, the projections on third (green) and fourth (yellow) principal components show no sign of oscillation and look closer to a constant line $y = 0$. Altogether, we do not have any direct evidence that rotation was introduced into our system. The only one thing that have change are the periods of the oscillatory functions. In case four our model oscillates more violently. May be that is caused by rotation, may be no - we have no way of knowing that for sure based on our data. One thing that we can conclude is that given system is indeed a two-degree of freedom system.

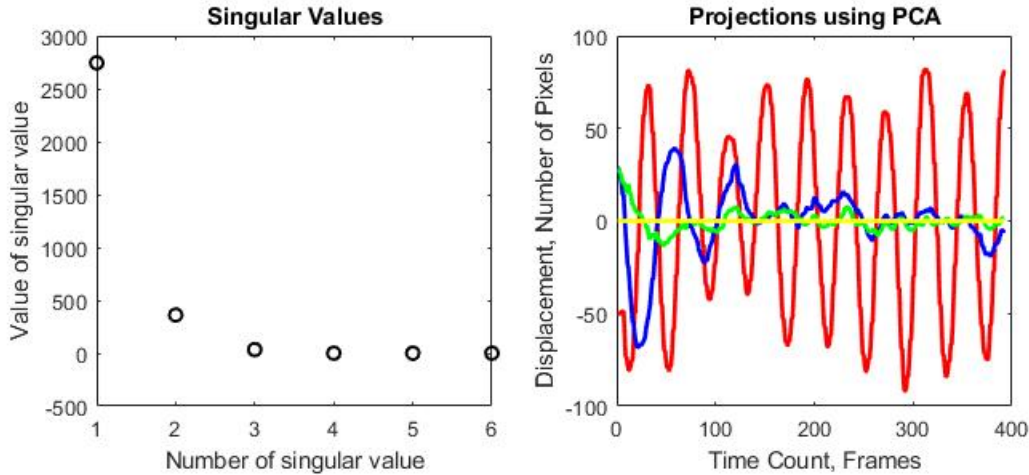


Figure 4: Results for the Horizontal Displacement and Rotation Case. Singular Values: [2744.5, 360.9, 33.7, 0, 0, 0]

Summary and Conclusions

In this paper we applied Principal Component Analysis to mass-spring system to determine a state of the system. We used four different scenarios to test whether PCA is the right tool or not. Here are our findings:

1. **Ideal Case:** In the ideal case the PCA performed extremely well. We were able to get a nice plot of the governing function of behaviour, which quite accurately represented the systems actual behaviour.
2. **Noisy Case:** The result of PCA was quite noisy. Although we were able to recognize the oscillatory behaviour in the system, we would not recommend using PCA on noisy data since the output will be noisy as well. If one have a noisy data and one wants to use PCA on it, one should de-noise it first if he/she wants to get a meaningful results out of PCA.
3. **Horizontal Displacement Case:** In the horizontal displacement case the PCA performed very well. It was able to identify both the movements in z direction and the $x - y$ plane.
4. **Horizontal Displacement and Rotation Case:** The result of PCA in this case left us with some question in our minds. For instance, we were not able to directly identify rotational movement of the system. Altogether, a bigger data set that will include 3-D mapping of objects will possibly deal with this uncertainty, however at the current state, usage of PCA on the given data-set is a 66-34 option. It successfully identifies movements in z direction and the $x - y$ plane, but misses the rotational movement.

To sum up all of the things, this paper a great insight into different scenarios when one could use principal component analysis to get an insight into the state of system. At the end we arrive to the conclusion whether the usage of PCA is effective or not for each specific scenario.

Appendices

A MATLAB commands

`load()`: Used to load video files into MatLab.
`zeros()`: Used to create empty matrices.
`rgb2gray()`: Used to move image from rgb space to grayscale space.
`double()`: Used to convert numbers to the double data type.
`min()`: Used to find the minimum element of the array.
`size()`: Used to find the size of the array.
`imresize()`: Used to resize images to the same dimensions.
`reshape()` : Used to reshape 2D matrix into 1D array.
`rem()`: Used to find remainder.
`mean()`: Used to find the mean of the data-set.
`cov()`: Used to find the co-variance matrix.
`eig()`: Used to find eigenvectors and eigenvalues.
`sort()`: Used to sort eigenvalues in decreasing order.
`diag()`: Used to find the diagonal of the matrix.

B MATLAB code

B.1 Main File

```
1 % HW3 – PCA
2 %% Test 1 – Initialization
3
4 clear all; close all; clc
5
6 disp("Getting Camera 1")
7 camOneRgb = load('cam1_1');
8 camOneRgb = camOneRgb.("vidFrames1_1");
9
10 disp("Getting Camera 2")
11 camTwoRgb = load('cam2_1');
12 camTwoRgb = camTwoRgb.("vidFrames2_1");
13
14 disp("Getting Camera 3")
15 camThreeRgb = load('cam3_1');
16 camThreeRgb = camThreeRgb.("vidFrames3_1");
17
18 % Test 1 – Setting-Up
19
20 close all; clc
```



```

21
22 timeSize = min([size(camOneRgb, 4) size(camTwoRgb, 4) size(
    camThreeRgb, 4)]);
23
24 xSize = min([size(camOneRgb, 1) size(camTwoRgb, 1) size(
    camThreeRgb, 1)]);
25
26 ySize = min([size(camOneRgb, 2) size(camTwoRgb, 2) size(
    camThreeRgb, 2)]);
27
28 camOneGrayMat = zeros(xSize, ySize, timeSize);
29 camTwoGrayMat = camOneGrayMat;
30 camThreeGrayMat = camOneGrayMat;
31
32 camOneGrayA = zeros(timeSize, xSize*ySize);
33 camTwoGrayA = camOneGrayA;
34 camThreeGrayA = camOneGrayA;
35
36 for i = 1:timeSize
37     camOneGrayMat(:, :, i) = imresize(double(rgb2gray(camOneRgb
        (:, :, :, i))), ...
38         [xSize, ySize]);
39     camTwoGrayMat(:, :, i) = imresize(double(rgb2gray(camTwoRgb
        (:, :, :, i))), ...
40         [xSize, ySize]);
41     camThreeGrayMat(:, :, i) = imresize(double(rgb2gray(camThreeRgb
        (:, :, :, i))), ...
42         [xSize, ySize]);
43     camOneGrayA(i, :) = reshape(camOneGrayMat(:, :, i), [1, xSize*
        ySize]);
44     camTwoGrayA(i, :) = reshape(camTwoGrayMat(:, :, i), [1, xSize*
        ySize]);
45     camThreeGrayA(i, :) = reshape(camThreeGrayMat(:, :, i), [1, xSize
        *ySize]);
46 end
47
48 allCamsGrayA = [camOneGrayA; camTwoGrayA; camThreeGrayA];
49
50 allCamsGrayMat = zeros(xSize, ySize, timeSize, 3);
51 allCamsGrayMat(:, :, :, 1) = camOneGrayMat;
52 allCamsGrayMat(:, :, :, 2) = camTwoGrayMat;
53 allCamsGrayMat(:, :, :, 3) = camThreeGrayMat;
54
55 leftBorders = 4*[70; 60; 60];
56 rightBorders = 4*[100; 90; 131];

```

```

57
58 [xCordinates, yCoordinates] = findCoordinates(allCamsGrayMat,
        leftBorders, ...
59         rightBorders, timeSize, xSize, ySize);
60
61 % Test 1 – PCA
62
63 %Creating X vector
64 xVec = zeros(6, timeSize);
65 for i = 1:6
66     if rem(i, 2) == 0
67         xVec(i, :) = yCoordinates(i/2, :);
68     else
69         xVec(i, :) = xCoordinates(rem(i,2),:);
70     end
71 end
72
73 [m,n] = size(xVec); % compute data size
74 mn = mean(xVec,2); % compute mean for each row
75 xVec = xVec - repmat(mn,1,n); % subtract mean
76 covarianceMat = cov(xVec'); % compute covariance
77
78 [V,D]=eig(covarianceMat); % eigenvectors(V)/eigenvalues(D)
79 lambda=diag(D); % get eigenvalue
80
81 [dummy,m_arrange]=sort(-1*lambda); % sort in decreasing order
82 lambda=lambda(m_arrange);
83 V=V(:,m_arrange);
84
85 Y = V'*xVec;
86
87 % Test 1 – Output
88
89 subplot(1,2,1)
90 plot(lambda, 'ko', 'Linewidth', [1.5])
91 title('Singular Values')
92 xlabel('Number of singular value')
93 ylabel('Value of singular value')
94 subplot(1,2,2)
95 plot([1:timeSize], Y(1,:), 'r', [1:timeSize], Y(2,:), 'b', [1:
        timeSize], ...
96         Y(3,:), 'g', [1:timeSize], Y(4,:), 'y', 'Linewidth', [2])
97 title('Projections using PCA')
98 xlabel('Time Count, Frames')
99 ylabel('Displacement, Number of Pixels')

```

```

100
101 %% Test 2 – Initilization
102
103 clear all; close all; clc
104
105 disp(" Getting Camera 1")
106 camOneRgb = load('cam1_2');
107 camOneRgb = camOneRgb.(" vidFrames1_2");
108
109 disp(" Getting Camera 2")
110 camTwoRgb = load('cam2_2');
111 camTwoRgb = camTwoRgb.(" vidFrames2_2");
112
113 disp(" Getting Camera 3")
114 camThreeRgb = load('cam3_2');
115 camThreeRgb = camThreeRgb.(" vidFrames3_2");
116
117 % Test 2 – Setting-Up
118
119 close all; clc
120
121 timeSize = min([ size(camOneRgb, 4) size(camTwoRgb, 4) size(
    camThreeRgb, 4)]);
122
123 xSize = min([ size(camOneRgb, 1) size(camTwoRgb, 1) size(
    camThreeRgb, 1)]);
124
125 ySize = min([ size(camOneRgb, 2) size(camTwoRgb, 2) size(
    camThreeRgb, 2)]);
126
127 camOneGrayMat = zeros(xSize, ySize, timeSize);
128 camTwoGrayMat = camOneGrayMat;
129 camThreeGrayMat = camOneGrayMat;
130
131 camOneGrayA = zeros(timeSize, xSize*ySize);
132 camTwoGrayA = camOneGrayA;
133 camThreeGrayA = camOneGrayA;
134
135 for i = 1:timeSize
136     camOneGrayMat(:, :, i) = imresize(double(rgb2gray(camOneRgb
        (:, :, :, i))), ...
        [xSize, ySize]);
137     camTwoGrayMat(:, :, i) = imresize(double(rgb2gray(camTwoRgb
        (:, :, :, i))), ...
        [xSize, ySize]);
138     camThreeGrayMat(:, :, i) = imresize(double(rgb2gray(camThreeRgb
        (:, :, :, i))), ...
        [xSize, ySize]);
139

```

```

140     camThreeGrayMat(:, :, i) = imresize(double(rgb2gray(camThreeRgb
        (:, :, :, i))), ...
141         [xSize, ySize]);
142     camOneGrayA(i, :) = reshape(camOneGrayMat(:, :, i), [1, xSize*
        ySize]);
143     camTwoGrayA(i, :) = reshape(camTwoGrayMat(:, :, i), [1, xSize*
        ySize]);
144     camThreeGrayA(i, :) = reshape(camThreeGrayMat(:, :, i), [1, xSize
        *ySize]);
145 end
146
147 allCamsGrayA = [camOneGrayA; camTwoGrayA; camThreeGrayA];
148
149 allCamsGrayMat = zeros(xSize, ySize, timeSize, 3);
150 allCamsGrayMat(:, :, :, 1) = camOneGrayMat;
151 allCamsGrayMat(:, :, :, 2) = camTwoGrayMat;
152 allCamsGrayMat(:, :, :, 3) = camThreeGrayMat;
153
154 leftBorders = 4*[70; 50; 60];
155 rightBorders = 4*[100; 100; 131];
156
157 [xCoordinates, yCoordinates] = findCoordinates(allCamsGrayMat,
        leftBorders, ...
158         rightBorders, timeSize, xSize, ySize);
159
160 % Test 2 – PCA
161
162 %Creating X vector
163 xVec = zeros(6, timeSize);
164 for i = 1:5
165     if rem(i, 2) == 0
166         xVec(i, :) = yCoordinates(i/2, :);
167     else
168         xVec(i, :) = xCoordinates(rem(i, 2), :);
169     end
170 end
171
172 [m, n] = size(xVec); % compute data size
173 mn = mean(xVec, 2); % compute mean for each row
174 xVec = xVec - repmat(mn, 1, n); % subtract mean
175 covarianceMat = cov(xVec'); % compute covariance
176
177 [V, D] = eig(covarianceMat); % eigenvectors (V)/eigenvalues (D)
178 lambda = diag(D); % get eigenvalue
179

```

```

180 [dummy, m_arrange]=sort(-1*lambda); % sort in decreasing order
181 lambda=lambda(m_arrange);
182 V=V(:, m_arrange);
183
184 Y = V'*xVec;
185
186 % Test 2 – Output
187
188 subplot(1,2,1)
189 plot(lambda, 'ko', 'Linewidth', [1.5])
190 title('Singular Values')
191 xlabel('Number of singular value')
192 ylabel('Value of singular value')
193 subplot(1,2,2)
194 plot([1:timeSize], Y(1,:), "r", [1:timeSize], Y(2,:), "b", [1:
    timeSize], ...
195      Y(3,:), "g", [1:timeSize], Y(4,:), "y", "Linewidth", [2])
196 title('Projections using PCA')
197 xlabel('Time Count, Frames')
198 ylabel('Displacement, Number of Pixels')
199
200 %% Test 3 – Initilization
201
202 clear all; close all; clc
203
204 disp("Getting Camera 1")
205 camOneRgb = load('cam1_3');
206 camOneRgb = camOneRgb.("vidFrames1_3");
207
208 disp("Getting Camera 2")
209 camTwoRgb = load('cam2_3');
210 camTwoRgb = camTwoRgb.("vidFrames2_3");
211
212 disp("Getting Camera 3")
213 camThreeRgb = load('cam3_3');
214 camThreeRgb = camThreeRgb.("vidFrames3_3");
215
216 % Test 3 – Setting-Up
217
218 close all; clc
219
220 timeSize = min([size(camOneRgb, 4) size(camTwoRgb, 4) size(
    camThreeRgb, 4)]);
221

```

```

222 xSize = min([size(camOneRgb, 1) size(camTwoRgb, 1) size(
    camThreeRgb, 1)]);
223
224 ySize = min([size(camOneRgb, 2) size(camTwoRgb, 2) size(
    camThreeRgb, 2)]);
225
226 camOneGrayMat = zeros(xSize, ySize, timeSize);
227 camTwoGrayMat = camOneGrayMat;
228 camThreeGrayMat = camOneGrayMat;
229
230 camOneGrayA = zeros(timeSize, xSize*ySize);
231 camTwoGrayA = camOneGrayA;
232 camThreeGrayA = camOneGrayA;
233
234 for i = 1:timeSize
235     camOneGrayMat(:, :, i) = imresize(double(rgb2gray(camOneRgb
        (:, :, :, i))), ...
236         [xSize, ySize]);
237     camTwoGrayMat(:, :, i) = imresize(double(rgb2gray(camTwoRgb
        (:, :, :, i))), ...
238         [xSize, ySize]);
239     camThreeGrayMat(:, :, i) = imresize(double(rgb2gray(camThreeRgb
        (:, :, :, i))), ...
240         [xSize, ySize]);
241     camOneGrayA(i, :) = reshape(camOneGrayMat(:, :, i), [1, xSize*
        ySize]);
242     camTwoGrayA(i, :) = reshape(camTwoGrayMat(:, :, i), [1, xSize*
        ySize]);
243     camThreeGrayA(i, :) = reshape(camThreeGrayMat(:, :, i), [1, xSize
        *ySize]);
244 end
245
246 allCamsGrayA = [camOneGrayA; camTwoGrayA; camThreeGrayA];
247
248 allCamsGrayMat = zeros(xSize, ySize, timeSize, 3);
249 allCamsGrayMat(:, :, :, 1) = camOneGrayMat;
250 allCamsGrayMat(:, :, :, 2) = camTwoGrayMat;
251 allCamsGrayMat(:, :, :, 3) = camThreeGrayMat;
252
253 leftBorders = 4*[70; 60; 60];
254 rightBorders = 4*[100; 90; 131];
255
256 [xCoordinates, yCoordinates] = findCoordinates(allCamsGrayMat,
    leftBorders, ...
257     rightBorders, timeSize, xSize, ySize);

```

```

258
259 % Test 3 – PCA
260
261 %Creating X vector
262 xVec = zeros(6, timeSize);
263 for i = 1:5
264     if rem(i, 2) == 0
265         xVec(i, :) = yCoordinates(i/2, :);
266     else
267         xVec(i, :) = xCoordinates(rem(i,2),:);
268     end
269 end
270
271 [m,n] = size(xVec); % compute data size
272 mn = mean(xVec,2); % compute mean for each row
273 xVec = xVec - repmat(mn,1,n); % subtract mean
274 covarianceMat = cov(xVec'); % compute covariance
275
276 [V,D]=eig(covarianceMat); % eigenvectors(V)/eigenvalues(D)
277 lambda=diag(D); % get eigenvalue
278
279 [dummy,m_arrange]=sort(-1*lambda); % sort in decreasing order
280 lambda=lambda(m_arrange);
281 V=V(:, m_arrange);
282
283 Y = V'*xVec;
284
285 % Test 3 – Output
286
287 subplot(1,2,1)
288 plot(lambda, 'ko', 'Linewidth', [1.5])
289 title('Singular Values')
290 xlabel('Number of singular value')
291 ylabel('Value of singular value')
292 subplot(1,2,2)
293 plot([1:timeSize], Y(1,:), 'r', [1:timeSize], Y(2,:), 'b', [1:
    timeSize], ...
294     Y(3,:), 'g', [1:timeSize], Y(4,:), 'y', 'Linewidth', [2])
295 title('Projections using PCA')
296 xlabel('Time Count, Frames')
297 ylabel('Displacement, Number of Pixels')
298
299 %% Test 4 – Initilization
300
301 clear all; close all; clc

```

```

302
303 disp(" Getting Camera 1")
304 camOneRgb = load( 'cam1_4' );
305 camOneRgb = camOneRgb.( " vidFrames1_4" );
306
307 disp(" Getting Camera 2")
308 camTwoRgb = load( 'cam2_4' );
309 camTwoRgb = camTwoRgb.( " vidFrames2_4" );
310
311 disp(" Getting Camera 3")
312 camThreeRgb = load( 'cam3_4' );
313 camThreeRgb = camThreeRgb.( " vidFrames3_4" );
314
315 % Test 4 – Setting-Up
316
317 close all; clc
318
319 timeSize = min([ size(camOneRgb, 4) size(camTwoRgb, 4) size(
    camThreeRgb, 4)]);
320
321 xSize = min([ size(camOneRgb, 1) size(camTwoRgb, 1) size(
    camThreeRgb, 1)]);
322
323 ySize = min([ size(camOneRgb, 2) size(camTwoRgb, 2) size(
    camThreeRgb, 2)]);
324
325 camOneGrayMat = zeros(xSize, ySize, timeSize);
326 camTwoGrayMat = camOneGrayMat;
327 camThreeGrayMat = camOneGrayMat;
328
329 camOneGrayA = zeros(timeSize, xSize*ySize);
330 camTwoGrayA = camOneGrayA;
331 camThreeGrayA = camOneGrayA;
332
333 for i = 1:timeSize
334     camOneGrayMat(:, :, i) = imresize(double(rgb2gray(camOneRgb
        (:, :, :, i))), ...
335         [xSize, ySize]);
336     camTwoGrayMat(:, :, i) = imresize(double(rgb2gray(camTwoRgb
        (:, :, :, i))), ...
337         [xSize, ySize]);
338     camThreeGrayMat(:, :, i) = imresize(double(rgb2gray(camThreeRgb
        (:, :, :, i))), ...
339         [xSize, ySize]);

```



```

340     camOneGrayA(i,:) = reshape(camOneGrayMat(:,:,i), [1, xSize*
        ySize]);
341     camTwoGrayA(i,:) = reshape(camTwoGrayMat(:,:,i), [1, xSize*
        ySize]);
342     camThreeGrayA(i,:) = reshape(camThreeGrayMat(:,:,i), [1, xSize
        *ySize]);
343 end
344
345 allCamsGrayA = [camOneGrayA; camTwoGrayA; camThreeGrayA];
346
347 allCamsGrayMat = zeros(xSize, ySize, timeSize, 3);
348 allCamsGrayMat(:,:,1) = camOneGrayMat;
349 allCamsGrayMat(:,:,2) = camTwoGrayMat;
350 allCamsGrayMat(:,:,3) = camThreeGrayMat;
351
352 leftBorders = 4*[70; 60; 60];
353 rightBorders = 4*[100; 90; 131];
354
355 [xCoordinates, yCoordinates] = findCoordinates(allCamsGrayMat,
        leftBorders, ...
356         rightBorders, timeSize, xSize, ySize);
357
358 % Test 4 – PCA
359
360 %Creating X vector
361 xVec = zeros(6, timeSize);
362 for i = 1:5
363     if rem(i, 2) == 0
364         xVec(i, :) = yCoordinates(i/2, :);
365     else
366         xVec(i, :) = xCoordinates(rem(i,2),:);
367     end
368 end
369
370 [m,n] = size(xVec); % compute data size
371 mn = mean(xVec,2); % compute mean for each row
372 xVec = xVec - repmat(mn,1,n); % subtract mean
373 covarianceMat = cov(xVec'); % compute covariance
374
375 [V,D]=eig(covarianceMat); % eigenvectors(V)/eigenvalues(D)
376 lambda=diag(D); % get eigenvalue
377
378 [dummy,m_arrange]=sort(-1*lambda); % sort in decreasing order
379 lambda=lambda(m_arrange);
380 V=V(:, m_arrange);

```

```

381
382 Y = V'*xVec;
383
384 % Test 4 – Output
385
386 subplot(1,2,1)
387 plot(lambda, 'ko', 'Linewidth', [1.5])
388 title('Singular Values')
389 xlabel('Number of singular value')
390 ylabel('Value of singular value')
391 subplot(1,2,2)
392 plot([1:timeSize], Y(1,:), 'r', [1:timeSize], Y(2,:), 'b', [1:
    timeSize], ...
393       Y(3,:), 'g', [1:timeSize], Y(4,:), 'y', 'Linewidth', [2])
394 title('Projections using PCA')
395 xlabel('Time Count, Frames')
396 ylabel('Displacement, Number of Pixels')

```

B.2 Script that finds coordinates

```

1 function [xCoordinates, yCoordinates] = findCoordinates(data,
    leftBorder, ...
2     rightBorder, timeSize, xSize, ySize)
3
4 xCoordinates = zeros(3,timeSize);
5 yCoordinates = zeros(3,timeSize);
6 counts = zeros(3,timeSize);
7
8 for i = 1:timeSize
9     for j = 1:xSize
10        for k = 1:ySize
11            for l = 1:3
12                if data(j,k,i,l) >= 240 && k >= leftBorder(l) && k
                    <= rightBorder(l)
13                    xCoordinates(l, i) = xCoordinates(l, i) + j;
14                    yCoordinates(l, i) = yCoordinates(l, i) + k;
15                    counts(l, i) = counts(l, i) + 1;
16                end
17            end
18        end
19    end
20
21    for j = 1:3
22        xCoordinates(j, i) = xCoordinates(j, i)/counts(j, i);
23        yCoordinates(j, i) = yCoordinates(j, i)/counts(j, i);

```

```
24         end
25     end
26 end
```