Homework 4: PCA with Cameras

Karl Marrett (kdmarrett@gmail.com)

Abstract

In this report we are given raw data from 3 cameras of raw footage of several dynamical systems. The dynamical systems analyzed involve the oscillations of a paint can on a spring in the following cases:

1. **Ideal case: This condition involves the case of the spring oscillating up and down (positive and negative z axis)**
2. **Noisy Case: This case is the same as the ideal only with notable noise in the recording devices on account of camera shake.**
3. **Horizontal Case: In this case, along with typical oscillation in the z axis, an additional horizontal displacement causes a pendulum like motion.**
4. **Horizontal and Rotation Case: In this case, along with typical oscillation and the pendulum motion: a rotation of target is introduced. This rotation is especially evident because there is a flashlight on top of the paint can.**

1 Introduction and Overview

The camera data and different cases represented in this report illustrate several important situations of dealing with high dimensional data in general. In these situations, since the physics of a mass on a spring is simple we could already predict that the dynamics of a mass on a spring is sufficient to describe in 1 dimension. In the past reports, we have used certain assumptions about our data to decide a reasonable basis to transform our data. For example in the analysis of sound files we used the convenient decomposition into the Fourier domain to create a low dimensional picture of the dynamics of the data.[1] However, when we instead don't know anything about the underlying structure of mechanism of the data we need a systematic approach. For example, given neural data of a spike train response of several neurons we may be able to explain structure in the data in a low dimensional Fourier space but the simplest model may in fact be described in an entirely different space. For problems of this nature, we want a data-driven approach for choosing the right basis to describe our data. Having a generic to high dimensional data of this sort allows equation-free modeling while also sidestepping our potentially misleading initial assumptions about the data.

The classic approach of choosing a basis from the data involves the technique known as Principal Component Analysis (PCA). The PCA finds a set of bases that maximally describe the variance of the data. This is performed via an Eigen decomposition of the covariance matrix. How and why this technique might describe a low dimensional bases for our function is described in detail in the next section.

2 Theoretical Background

2.1 Covariance Matrix

To decompose our data into modes of variance we can use the covariance matrix. The covariance matrix has certain special properties that allow it to be of use in our analysis. In this problem, we turn our image data into the x, y coordinates of a mass on a spring, thus we can consider the covariance matrix to be the following:

$$\text{Covar}\, M = \begin{pmatrix} \text{Var}\, x & \text{Covar}\, xy \\ \text{Covar}\, xy & \text{Var}\, y \end{pmatrix} \tag{1}$$

If we examine a common covariance matrix in this case in two dimensions we can note some properties. Namely, the diagonals describe the variance of the individual x and y and components whereas the off diagonals explain the amount of correlation between them. For example, if we had uncorrelated data of x and y, (imagine a scatter plot of x and y that forms a spherical blob), we would then have a covariance matrix where the off diagonal terms are near zero and where the diagonals describe the variance of the single variable. Having only two variables in the example is only for simplicity, the covariance will be the same regardless of the dimension and we will use 6 dimensions for the camera data. This is important for the discussion of the Eigen decomposition discussed in the next section.

2.2 Eigen Decomposition

Taking the example from above of a two dimensional spherical uncorrelated blob of x and y we can ask what would the Eigen values and Eigen vectors of the system be. If x and y were completely uncorrelated, then the off diagonal terms would be 0 and the matrix would be:

$$\text{Covar}\, M = \begin{pmatrix} \text{Var}\, x & \text{Covar}\, xy \\ \text{Covar}\, xy & \text{Var}\, y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \text{Var}\, x & 0 \\ 0 & \text{Var}\, y \end{pmatrix} \tag{2}$$

In this case, the Eigen vectors are simply the columns of the two by two identity matrix and the Eigen values describe the variances along those dimensions which in this case is simply $\text{Var}\, x$ and $\text{Var}\, y$. If the covariance matrix didn't happen to be diagonal like this ideal example, then our Eigen values would describe a new basis and the Eigen values would be the variance along those correspondent dimensions. Thus for any generic covariance matrix of size n, we will we have a corresponding number of Eigen vectors which describe a new basis to describe the covariance matrix.

Figure 1: As an exploration, the red, green, and blue parts of the image were separated to see which was best at discriminating the features of the paint can. We can see in the top left the original image changed to black and white, the top right the red values, lower left green, and lower right blue. In the end the gray values were used since the conversion from RGB to gray captured the most energy

2.3 Principal Component Analysis

From the Eigen decomposition of the covariance matrix we get a new set of bases functions and a corresponding set of Eigen values which describe the variance along these dimensions. These bases assume that the covariance was full rank but if it was not, in other words if there was redundancy between the different dimensions, then it would reveal itself as a low Eigen value for that Eigen vector. This could be very useful when we have a high dimensional data set that we intuit can be described in a lower dimensional space. This is exactly the approach we take in this report. For each of the four cases we have 3 cameras capturing the same dynamics of a paint at about the same times. Since we analyze the data as a set of x and y components, we essentially have a data set of a 6 dimensional space. However despite the data being in a multi-dimensional space recorded at different angles and locations we still expect that information about the variance is redundant.

The motivation for using PCA on this data set is that in these cases the camera data don't necessarily define the best bases for the dynamics of the system. What's more, we do not want to assume the ideal number of basis of the data. Thus running the PCA in the way described allows a data driven approach to the issue of dimension reduction.

3 Algorithm Implementation and Development

A major focus of the PCA technique used in this report is cleaning the original video footage. We could use a PCA approach on the raw video footage but it would require an unrealistic

amount of computing resources for a personal computer especially if converting the *uint*8 format to the larger *double*. Instead, we will use a combination of techniques from the previous homeworks to get an x and y coordinate of the center of the paint can in each frame of the camera data for the different cases.

We start with movie files that are essentially an image (3 dimensional RGB matrix with each element corresponding to a pixel with an associated weight 0-255) for a set of time slices. Since the data that we are working with is camera data, it is inherently three dimensional with the extra dimension of time. The images were viewed in black and white across the original, red, green, and blue dimensions as shown in Figure 1. Red images showed to give the paint can the highest contrast over its surroundings. However, identifying the paint can in each image from value alone isn't discriminate enough from other features in the image. Since the paint can is the major source of motion across time slices, a more intelligent way to gain some traction on the target is to subtract image values from the next image. However, this does introduce new noise from other moving parts of the frame, producing edges around the other moving objects. Since these lines from other objects are generally less intense than the paint can, we can apply a light Gaussian filter across the entire frame to filter the smaller noise to basically zero. Another strategy which was tried takes advantage that the paint only appears in certain pixels of the frame for the different cameras. Knowing this information makes it easier to simply set all pixels out of this area to zero to lower the possibility that the paint can will be identified in another area . It turns out this strategy was not worth the effort of choosing the locations of the can in the image and it also overgeneralizes this data set. We want to rely on solutions that can perform behavior on a variety of types of situations by relying on the mathematical techniques developed rather than any specific adjustments to fit the data.

To preprocess the individual time slices we relied on several functions. The first step after loading all the camera data was to pass it to the function *dynamicObjLocation* which returned the estimated x, y coordinates of the paint can at each time step. Extracting the location required several steps, the first of which applied the function $imageflt$ which applies a Gaussian filter over the entire image. Shown below in Equation 3 describes the function applied over the image.

$$F(k_x, k_y) = e^{-D((k_x-a)^2+(k_x-b)^2)} \tag{3}$$

where a represents the width of the Gaussian and b represents how it is shifted across the time axis. Applying this essentially smooths over our image and allows it so that small features are less dominant. This is useful in lowering the values of some frames where there are other objects that are moving. In such frames, those images have a line around the moving objects because of the subtraction method used. By applying a global filter a lot of these small thin lines actually disappeared.

After a weak global filter and stronger local filter was applied with the function *smoothArea*. Within the function *smoothArea* a specified part of the frame is diffused with the function $imagedfs$. The process of diffusion of an image for the purpose of this report is very similar to the Gaussian filtering described above. A special technique when applying diffusion to a

small area was to choose a center from the previous point found for the target. A box size (denoted by the varible *pixSize*) was chosen by estimating the maximum area that the paint can traveled in one frame. The *smoothArea* function also slightly enhanced the brightness of the area that it diffused. This is nicely illustrated in Figure 2. We can treat this as essentially raising the probability that the current object location would be found in the same area as the previous one, helping the contiguity of the final data. Since, some frames had almost no information in them a more complex system for taking locations was developed. Basically, if no pixels were above some threshold value that frame's object location was flagged for later interpolation with the local points. After smoothing, the local area the entire frame was turned into a binary matrix where all values above were turned into 1 and all values below to 0. This created essentially blobs of motion across the frame. Using the *regionsProps* function, the centroid of all the blobs of the images was taken. Of all the centroids, the one with the largest area were estimated to be the location of the target. The final section of the *dynamicObjLoc* function was devoted to interpolating flagged null target locations. The results of this function were passed back to CamScript as a series of locations for each camera. The camera data was then stacked vertically on top of each other with the x and y location rows and the columns of each time point. This was the matrix needed to run the actual PCA analysis.

The pca analysis consisted of taking the covariance matri of the matrix of each case then calculating the Eigen vectors and associated Eigen values of that matrix. By exploring the results of the Eigen values, we can see what percentage of the variance could be described in a smaller dimensional space.

4 Computational Results

The results of the PCA analysis for each condition is shown in Table 1. The estimated ideal bases and the percentage variance that that basis would retain is listed in the table. Diagrams for the computed locations in the movies is shown for the first camera in case 1, 2, 3, 4 in Figures 3, 4, 5, 6, respectively. The number of ideal dimensions was estimated from the bar graphs of the Eigenvalues of each case.

4.1 Case 1

Since the paint can in this case follows a simple sine wave we know that it always varies with respect to one direction, the z axis. Although the cameras were placed at different angles and locations, the PCA technique recognized to some extent that the variance was mostly in 1 dimension. There was however quite a bit of energy left in the other dimensions as shown in Figure 7.

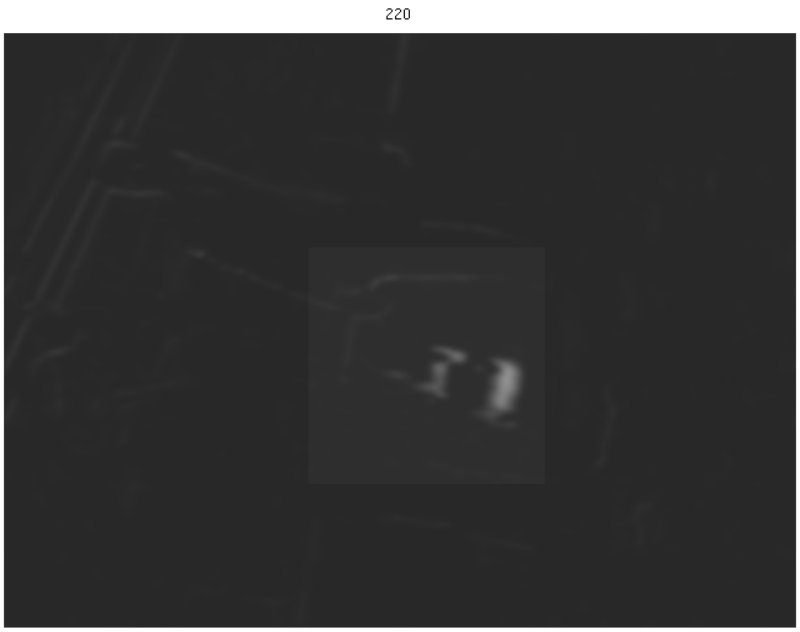

Figure 2: This figure illustrates some of the computation involved in estimating the position of the paint can. This image shows the black and white image of a frame that has had the previous frame subtracted to highlight movement. It has also undergone global filtering with a Gaussian filter. A faint gray square can also be seen, this is where the smoothArea function has blurred and brightened the data based off the previous reasonable location of the target.

4.2 Case 2

This case was the same as the previous except that noise was added. Presumably it would contain a primary Eigen value and much smaller secondary and tertiary values. The bar graph for this case actually has a stronger value in the primary Eigen vector than the non-noise case which shows that the denoising methods used for resolving the image location were not particularly effective as shown in Figure 8.

4.3 Case 3

This situation clearly had strong variance along two dimensions. The first being the z direction as before and the second being the new pendulum motion back and forth. From the bar graph we do see two stronger Eigen values but due to the high noise it would be unreasonable to argue that we would be able to pick this as a 2 dimensional decomposition from the bar graph alone as shown in Figure 9.

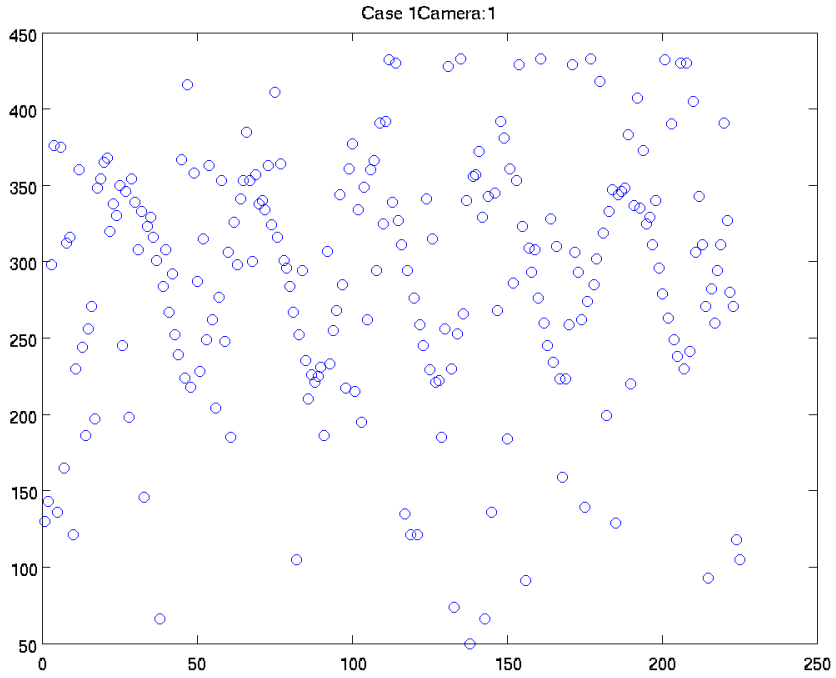

Figure 3: Object locations captured for the time points in Case 1 for camera 1

4.4 Case 4

In this case we would expect at least 3 strong Eigen values. Two for the spring and pendulum motion in case 3 and an extra for the dynamic flashing of the light. Indeed, the bar graph for this case does show three strong Eigen values in Figure 10. Given the data, whether the rotation of the flashlight was actually captured in the third Eigen value is highly debatable.

Table 1: Table showing the estimated ideal dimension to decompose the the case into along with the decimal percentage variance explained by that number of dimensions returned by PCA.

	Case 1	Case 2	Case 3	Case 4
Dimension	1	1	2	3
Fractional variance	.37	.49	.67	.83

5 Summary and Conclusions

Although the final results are not ideal, using PCA in this way actually helped to indicate the effectiveness of our dynamicObjLoc function. Using the knowledge of physics and the covariance matrix we made predictions of what the final PCA should look like and evaluate the methods with the actual results. In this way, the main focus of this report, the technique

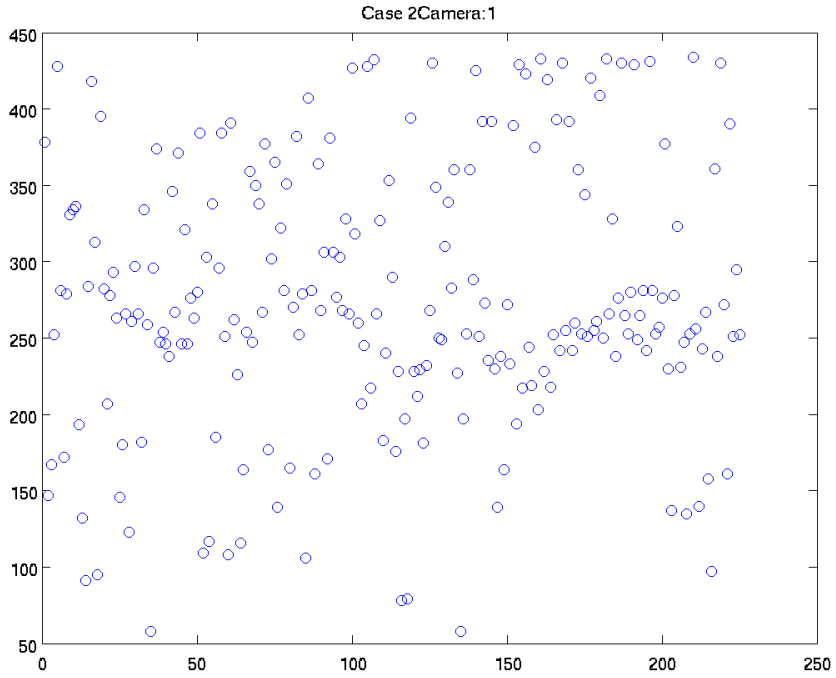

Figure 4: Object locations captured for the time points in Case 2 for camera 1

of PCA was achieved, although the method cleaning and preprocessing the data seemed largely ineffective. Through our results we find that the technique of Principal Component Analysis provides a useful method for analyzing high dimensional data. It is the first of approaches learned so far of allowing the decision of the choice of basis to be determined by the data rather than by the assumptions of the analysts. In this way, it is quite powerful in comparison to the techniques we have studied so far. Although variance is a reasonable way to infer structure in data there are perhaps more meaningful ways to pull out information in large datasets. Alternatives to relying on the covariance and the PCA technique will be examined in future homeworks.

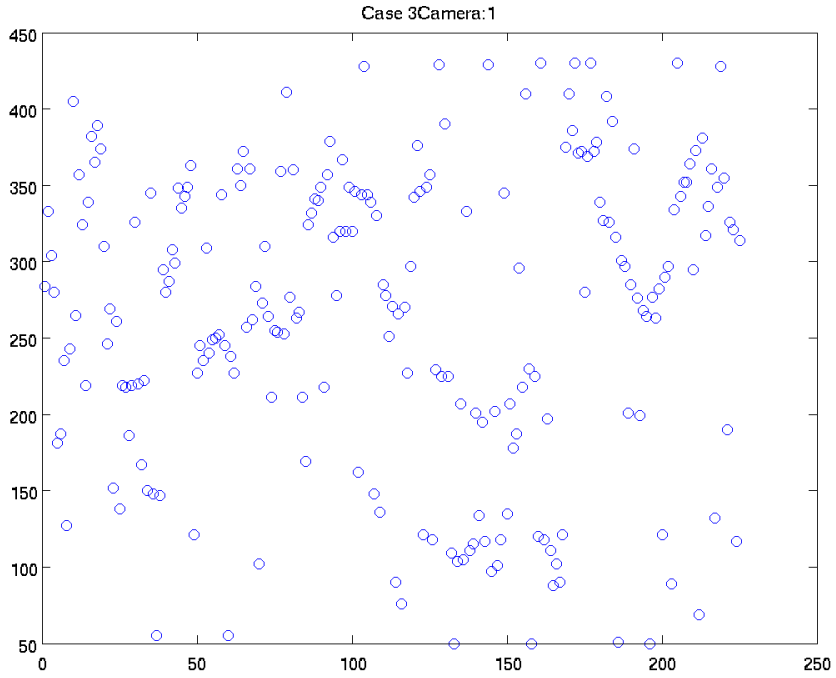

Figure 5: Object locations captured for the time points in Case 3 for camera 1

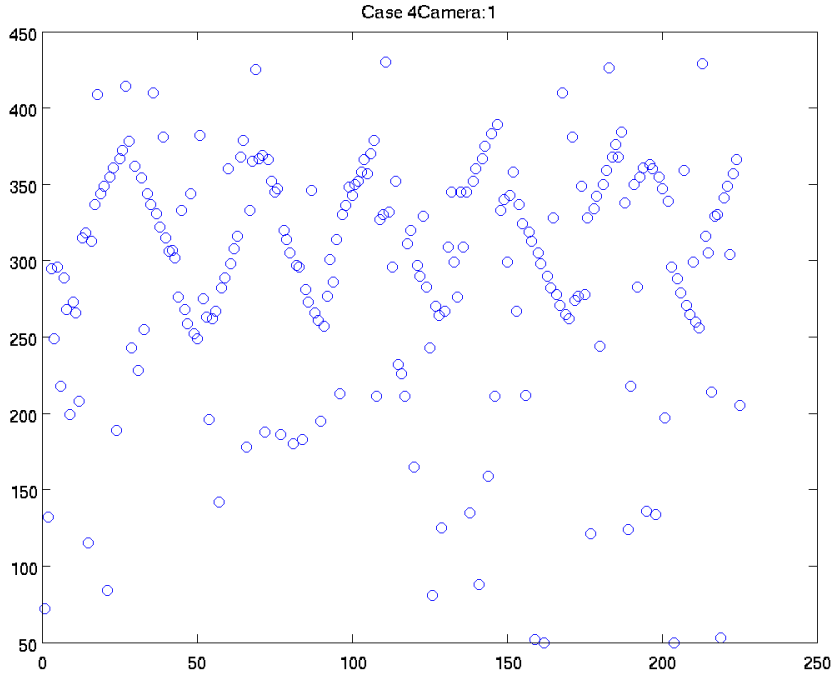

Figure 6: Object locations captured for the time points in Case 4 for camera 1

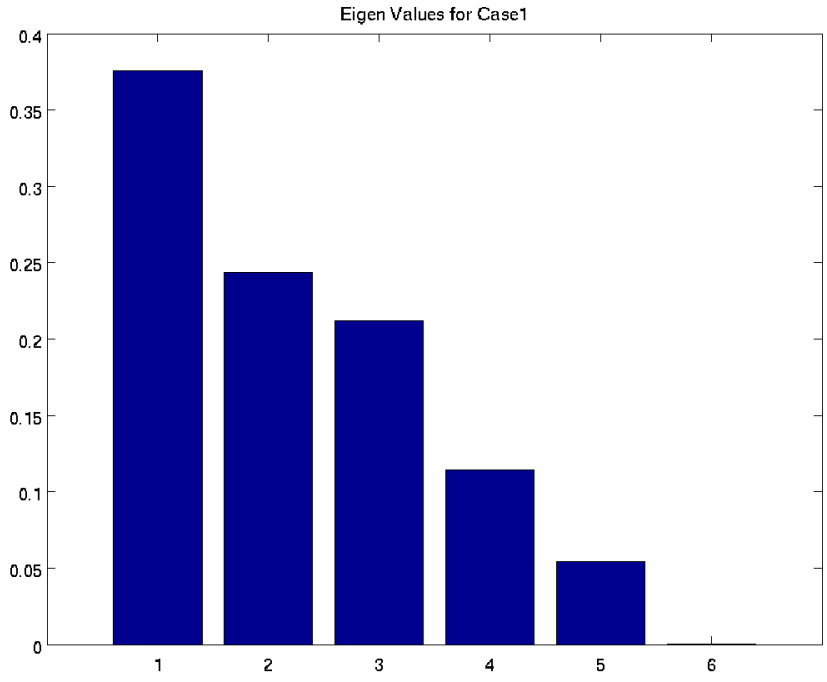

Figure 7: Case 1 Eigen Value decomposition: we would expect since this condition has oscillation in only one dimension that it would contain a major single Eigen values and other relatively small ones, but this graph shows other major contributions from the other axes indicating a high degree of noise. However virtually all variance can be represented in a 5 dimensional space

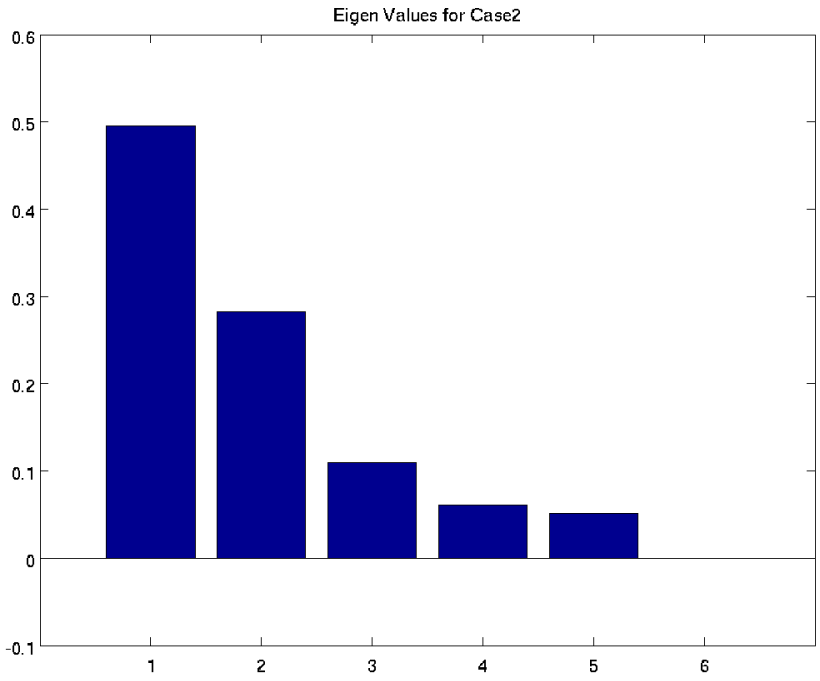

Figure 8: Case 2 Eigen Value decomposition: A strong single Eigen value is present as one would expect for the simple oscillation case, however noise also contributes to Eigen values in the other dimensions

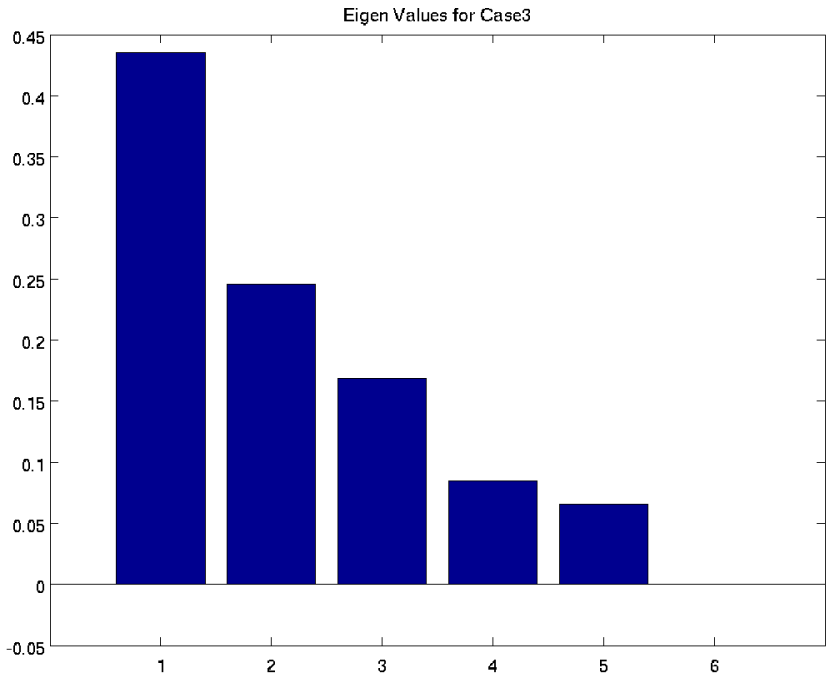

Figure 9: Two strong Eigen values for the pendulum and oscillation motion are observed. Although there is other variance in more dimensions, virtually all of the variance is included in 5 dimensions

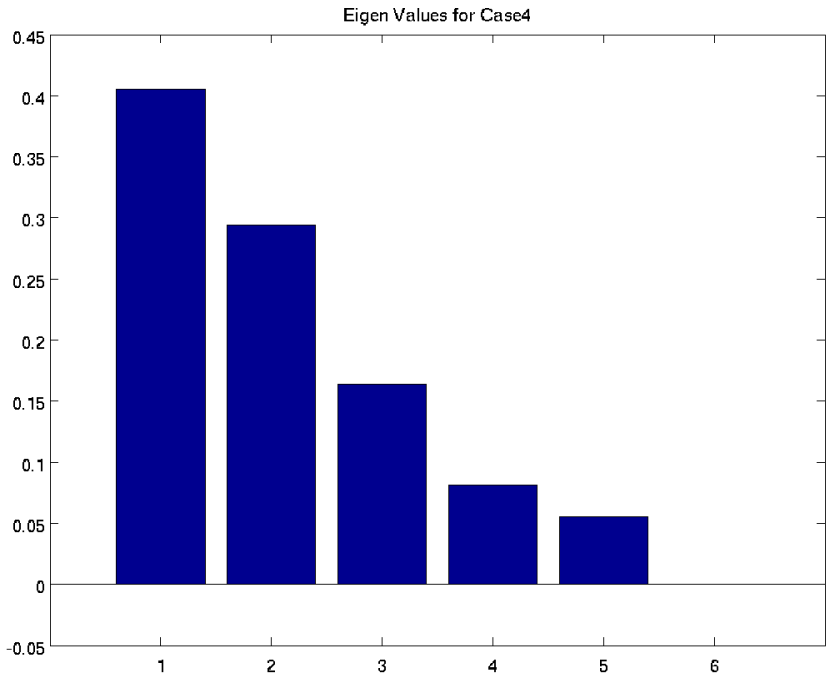

Figure 10: This bar graph shows a clear drop off between each potential dimension of the oscillation, the pendulum motion, and the rotation of the paint can.

A
EIGVEC, EIGVAL, ENERGY = pca(INPUT , RANK) function written for this assignment for taking the Principal Component Analysis of video data it can also return the PCA of only the first RANK vectors of the data. It also returns the total percent of variance captured by the RANK specified returned as a decimal
regionprops used to calculate the centroids and their area in a thresholded black and white image
rgb2gray used to convert the 3 dimensions of red green and blue to an approximation in a single 0 255 valued space
graythresh computes an accurate threshold for an image
smoothArea uses diffusion to denoise/blur a specific part of an image
imagefilt a Gaussian filter applied globally to an image
imagedfs a diffusion filter applied globally to an image

B

```
%Karl Marrett
%AMATH 482
%Homework 4 PCA
%Due Feb 26th 2015

close all; clear all;

cams = 3; % tot 3
caseno = 1; % tot 4
framesAnalyzed = 225;
minFrames = framesAnalyzed + 1;
smoothSpan = .1;

% LOAD/ARRANGE DATA
% files named by camera recorded from '_' case number
for k = 1 : caseno
  case_specifier = strcat('case', int2str(k));
  camData.(case_specifier) = [];
  for l = 1 : cams
    file_specifier = strcat(int2str(l), '_', int2str(k));
    name = strcat('cam', file_specifier, '.mat');
    vid = load(name);
    frameField = strcat('vidFrames', file_specifier);
    % ensure constant number of frames
    vidDat = vid.(frameField)(:,:,:, 1:(minFrames));
    % [a,b,c,frames] = size(vidDat);
    % assert(frames == minFrames, 'Mismatched frame numbers');

    % Get x;y pixel location of moving objection
    locations = dynamicObjLoc(vidDat);
    % save data stacking cameras vertically
    camData.(case_specifier) = ...
      [camData.(case_specifier); locations];
  end

% colorscheme = {'bo', 'ro', 'go'};
% for k = 1: cams
% figure(k)
%   plot(1:length(camData.case4(1*k,:)), camData.case4(2*k,:), ...
%     colorscheme{k});
```

```
%   title(strcat('Case 4 ', 'Camera: ', int2str(k)));
% saveas(gcf, strcat('case4cam', int2str(k)), 'png');
% end


% plot(camData.case1(1, :), camData.case1(2, :), 'ro');
% plot(1:length(camData.case1(1,:)), camData.case1(1,:), 'bo');

% figure(1)
% subplot(2,2,1)
% % red
% imshow(camData.case1(:,:,1,1));
% subplot(2,2,2)
% % green
% imshow(camData.case1(:,:,1,1));
% subplot(2,2,3)
% % blue
% imshow(camData.case1(:,:,1,1));
% subplot(2,2,4)
% % grayed
% imshow(rgb2gray(camData.case1(:,:,:,1)));
% saveas(1, 'colors', 'png');

[eigVec, eigVal, energy] = pca(camData.(case_specifier), 1);
energy
figure(k)
%bar(eigVal/sum(eigVal));
%title(strcat('Eigen Values for Case ', int2str(k)));
%saveas(k, strcat('eigvals', int2str(k), 'full'), 'png');

end
function [locations] = dynamicObjLoc(frames)
% Returns center locations of the major moving
% object in a frame

% Misc. Parameters
[rows,cols,~,minFrames] = size(frames);
tau=0.0005; % time of diffusion
D=0.0025; % diffusion coefficient
strength = .0004;
% square width of target paint can
pixSize = 250;
```

```
% crop image around these pixels
cropBool = 0;
background = uint8(zeros(rows, cols));
minX = 290;
maxX = 450;
minY = 40; maxY = 450;
pausetime = 3;
thresh = 40;
binaryBool = 1;
baselineEnergy = 0;

locations = [];
for m = 2:(minFrames)
  % get gray time slice for processing
  slice = rgb2gray(frames(:,:,:,m));

  % crop around known locations of paint can
  if cropBool
    crop = background;
    crop(minX:maxX, minY:maxY) = slice(minX:maxX, minY:maxY);
  else
    crop = slice;
  end
  % build matrix of changed pixels
  changeFrame = uint8(crop - rgb2gray(frames(:,:,:, m -1)));
  changeFrame = changeFrame + baselineEnergy;

  % global Gaussian filter
  changeFrame = imageflt(changeFrame, strength);
  tryIndex = 2;
  while (true)
    try
      lastLoc = locations(:, (m - tryIndex));
      tryIndex = tryIndex + 1;
    catch
      lastLoc = [cols / 2; rows / 2];
    end
    if (sum(lastLoc) ~= 0)
      break;
    end
  end
  % Diffuse local area of paint can
```

```
% if (m > 10)
%   imshow(changeFrame); title('before smootharea');pause(.4);
% end
changeFrame = smoothArea(changeFrame, lastLoc(1), lastLoc(2), ...
  pixSize, tau, D ); if binaryBool
  testafter = length(find(changeFrame) > thresh);
  if testafter
    binaryChange = im2bw(changeFrame,...
       graythresh(changeFrame));
    % if (m > 10)
    %   imshow(binaryChange); title('binary');pause(2)
    % end
    region = regionprops(binaryChange, 'Centroid',...
      'Area', 'PixelIdxList');
    [maxValue, index] = max([region.Area]);
    row = region(index).Centroid(1);
    col = region(index).Centroid(2);
    %centroids = cat(1, region(index).Centroid);
    %row = centroids(1, 1);
    %col = centroids(1, 2);
    %[row, col] = ind2sub(size(binaryChange), ...
    % mean(region(index).PixelIdxList));
    % Visualize
    % if (m > 10)
    %   % imshow(binaryChange);
    %   hold on;
    %   temp = background;
    %   temp(floor(row),floor(col)) = 255;
    %   title('final decision');
    %   plot(temp,'r.','MarkerSize',20);
    %   pause(2);
    %   hold off;
    % end
    % alternate
    % centroids = cat(1, region.Centroid);
    % xCentroids = centroids(:, 1);
    % yCentroids = centroids(:, 2);
    % Remove any NaN measururements
    % x = mean(xCentroids(~any(isnan(xCentroids))));
    % y = mean(yCentroids(~any(isnan(yCentroids))));
  else
    x = 0; y = 0;  % mark interpolated max later
```

```
    end
    % append locations horizontally by time slice
    locations(:, (m - 1)) = [ floor(row); floor(col) ];
  else
    % get subscript of max value
    [row,col] = ind2sub(size(changeFrame), ...
    find(changeFrame == max(max(changeFrame))));
    region = regionprops(changeFrame, 'WeightedCentroid');
    locations(:, (m - 1)) = [ row; col ];
  end


  % Visualize
  % if (m > 10)
  %   imshow(changeFrame);
  %   hold on;
  %   temp = background;
  %   temp(locations(1, (m - 1)),locations(2, (m - 1))) = 255;
  %   title('final decision');
  %   plot(temp,'r.','MarkerSize',20);
  %   pause(2);
  %   hold off;
  % end
end

zeroCols = find(sum(locations) == 0);

for l = 1:length(zeroCols)
  try
    locations(1, zeroCols) = (locations(1, zeroCols - 1) + ...
      locations(1, zeroCols + 1)) /2;
    locations(2, zeroCols) = (locations(2, zeroCols - 1) + ...
      locations(2, zeroCols + 1)) /2;
  catch
    try
      locations(1, zeroCols) = ...
        (locations(1, zeroCols - 1));
      locations(2, zeroCols) = ...
        (locations(2, zeroCols - 1));
    catch
      % dummy first point
      x = cols / 2;
```

```
      y = rows / 2;
    end
  end
end

assert(length(find(sum(locations) == 0)) == 0, 'some zeros remain');
end

function [smoothImage] = smoothArea(image, ...
  centerX, centerY, pixSize, tau, D)
% Takes a uint8 matrix image, center pixel
% location in x y coordinates and a pixel size
% of a local area of an image to clean
% Takes doubles tau and diffusion constant D
% for diffusion filtering
% cleans the specific area of the image and
% returns the new matrix in unint8 format

pixelTrimFactor = 5;
brightEnhance = 2;
[x,y,z] = size(image);
hWid = floor(pixSize / 2);
centerX = floor(centerX);
centerY = floor(centerY);
% must be ints:
lowX = centerX - hWid ;
highX = centerX + hWid;
lowY = centerY - hWid;
highY = centerY + hWid;

% handle bleed over
if lowX < 1
  bleedunderx = lowX - 1;
  lowX = 1;
  % fprintf('Warning: area under index');
end
if lowY < 1
  bleedundery = lowX - 1;
  lowY = 1;
  % fprintf('Warning: area under index');
end
if highX > x
```

```
  bleedoverx = highX - x;
  highX = x;
  % fprintf('Warning: area over index');
end
if highY > y
  bleedovery = highY - y;
  highY = y;
  % fprintf('Warning: area over index');
end
xspan = highX - lowX;
yspan = highY - lowY;

if ((highX < 10)  && (highY < 10))

  % Cut patch out of image
  temp = image(lowX:highX, lowY:highY, :);
  % create higher dimensional copies for time slices
  for k = 1:length(tau)
    smoothImage(:,:,:, k) = image(:,:,:);
  end

  % Diffuse locally on that patch
  smoothTemp = imagedfs(temp, tau, D);
  [m,n,o,p] = size(smoothTemp);
  % trim out edges
  % add energy of brightEnhance to compensate for diffusion
  trimTemp = smoothTemp(((1 + pixelTrimFactor): ...
    (m - pixelTrimFactor)), ((1 + pixelTrimFactor):(n - ...
    pixelTrimFactor)), :, :) + brightEnhance;
  %place back into higher dimensional copies
  finalLowX = (lowX + pixelTrimFactor);
  finalLowY = (lowY + pixelTrimFactor);
  xspan = xspan - 2* pixelTrimFactor;
  yspan = yspan - 2* pixelTrimFactor;
  smoothImage(finalLowX :(finalLowX + xspan), ...
   finalLowY:(finalLowY + yspan), :, :) = trimTemp;
  % old version
  % smoothImage((lowX + pixelTrimFactor):(highX - pixelTrimFactor), ...
  % (lowY + pixelTrimFactor):(highY - pixelTrimFactor), :, :) = trimTemp;
else
  smoothImage = image;
end
```

```
end
function [dfsImage] = imagedfs(image, tau, D)
  % Takes a uint8 2 or 3 dimensional matrix and
  % an int strength (which parameterizes the width
  % of the filter in the spectral domain) and applies
  % Gaussian filtering in the 2nd dimension
  % returning the filtered uint8 matrix back

  [x,y,z]=size(image);
  L = buildL(x,y);
  for k=1:z
    % convert to double place in vector form
    u = reshape(double(image(:,:,k)), x*y, 1);
    % evolve through diffusion/heat equation
    [t, usol] = ode113('dfsODE', tau, u, [], L, D);
    % place back into dfs image
    for l = 1:length(tau);
      % create an extra dimension l for each tau value
      dfsImage(:, :, k, l) = uint8(reshape(usol(l,:),x,y));
    end
  end

end

function [filImage] = imageflt(image, strength)
  % Takes a uint8 2 or 3 dimensional matrix and
  % an int strength (which parameterizes the width
  % of the filter in the spectral domain) and applies
  % Gaussian filtering in the 2nd dimension
  % returning the filtered uint8 matrix back

  [x,y,z]=size(image);
  kx = 1:x; ky = 1:y;
  [Kx, Ky] = meshgrid(kx,ky);
  centerx = ceil(x/2);
  centery = ceil(y/2);
  % transposed Gaussian filter
  fil = (exp(-strength*(Kx - centerx).^2 - strength*(Ky - centery).^2))';
  for k=1:z
    imageFFT = fftshift(fft2(double(image(:,:,k))));
    temp = imageFFT.*fil;
```

```
    filImage(:, :, k) = uint8(abs(ifft2(ifftshift(temp))));
  end

end
```

References

[1] J N Kutz. *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*. OUP Oxford, 2013.