Principal Component Analysis of Multiple Redundant Video Samples

Trent Yarosevich[1]

Abstract
Three redundant video recordings of a moving object are analyzed with Principal Component Analysis (PCA), a common multivariate statistical technique that can be use to identify lower order dynamics in high order systems. The analysis spans four sections which introduce new dynamics like transverse motion, rotation, and noise (camera shake). In each case, PCA will be used to try to identify the basic underlying physics by examining the dominant modes.

[1] *Department of Applied Mathematics, University of Washington, Seattle*

Contents

1. Introduction and Overview

https://github.com/tyarosevich/AMATH_582/blob/master/HW3/Final%20Version%20for%20Github%20Access/yarosevich_582_hw3_final.pdf

This investigation takes data from twelve videos (four groups of three videos), cleans up the data, and then analyzes it using PCA. The purpose of this exercise is to illustrate how the spring-mass dynamics can be uncovered with a low-order representation of the system. More specifically, we examine the behavior of just a few dominant modes, in the hopes that they will govern the majority of the dynamics in the data.

In each grouping of videos, the analysis will be identical, with each section involving new behavior. In the first group, we have a nice clean mass-spring object moving up and down; in the second group we introduce camera shake to each video; in the third we introduce horizontal displacement to the mass-spring object; and in the fourth we introduce *both* horizontal movement and rotation to the object (that is, it spins around as it moves). In each part, the analysis will attempt to identify these various dynamics in the dominant modes of the PCA.

2. Theoretical Background

2.1 The Singular Value Decomposition

The singular value decomposition, or SVD, is a famous and widely used matrix decomposition with countless applications, one common one being image analysis. Mathematically, the SVD is motivated by the observation that the image of the unit sphere under any *mxn* matrix is a hyperellipse [1], as in Figure 1. More intuitively, we can view this motivation as the

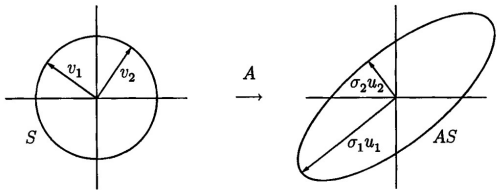

Figure 1. The SVD of a 2x2 matrix

decomposition of a matrix into three matrices that govern the transition in Figure 1. These matrices rotate, stretch, and then again rotate any matrix to which they are applied. Crucially, these rotation matrices are always unitary[2], and thus for a given rotation matrix R we have $R^T = R^{-1}$. Furthermore, the 'stretch' matrix is simply applying a scalar multiple to each dimension of the target matrix, and is thus diagonal. The SVD

[1] Trefethen, Lloyd N., and David Ill. Bau. Numerical Linear Algebra. SIAM Society for Industrial and Applied Mathematics, 2000, page 25,26.

[2] "Rotation Matrix." Wikipedia, Wikimedia Foundation, 20 Feb. 2020, en.wikipedia.org/wiki/Rotation_matrix.

is thus written as,

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

where $\hat{U}$ and V^* 'rotate' and are unitary, and Σ 'stretches' and is diagonal. In this case, we are referring to the 'reduced' SVD, which differs from the 'Full' SVD in two ways: first, it omits the silent columns of U that actually make it a unitary matrix, which is why it is written as $\hat{U}$; second it omits the additional zero singular values in Σ that correct for these silent columns in the final product. Generally speaking, the reduced SVD is used unless the unitary property is expressly required.

In the subsequent analysis, the SVD will be calculated numerically with the eponymous Matlab function. For this reason, I will not go into detail to the SVD's formal derivation, save to mention that it is related to the eigenvalue decomposition by two self-adjoint problems:

$$\begin{aligned} A^T AV &= V\Sigma^2 \\ AA^T &= U\Sigma^2 \end{aligned} \tag{2}$$

Consequently, this means that the non-zero singular values are related to the eigenvalues of A^*A and AA^*, which are hermitian, by $\sigma_j = \sqrt{\lambda_j}$.

In addition to the structure of these matrices, the SVD has several associated theorems[3]. I will state the following three as they are crucial to the subsequent analysis.

Theorem 1: *Every Matrix A 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: *A is the sum of r rank one matrices, $A = \sum_{j=1}^{r} \sigma_j u_j v_j^*$.*

Theorem 3: *For any N such that $0 \leq N \leq r$ we define the partial sum $A_N = \sum_{j=1}^{N} \sigma_j u_j v_j^*$, and if $N = min\{m,n\}$, define $\sigma_{N+1} = 0$, then $\|A - A_N\| = \sigma_{N+1}$.*

While the proof of existence is obviously important, it is **Theorem 3** and its interpretation that is perhaps most important to PCA. In short, this theorem guarantees that any reduced rank representation of a matrix A using the SVD is *guaranteed* to be the best representation possible in an L-2 norm (or Frobenius) sense. This result underscores the power and viability of PCA.

2.2 Principal Component Analysis (PCA)

A useful way to introduce PCA is to consider the variance and covariance among a set of measurements (precisely what this investigation will do). Suppose we have a matrix of data, in which each row represents a measurement (or each pair of rows, etc.) vector:

$$X = \begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix} \tag{3}$$

The variance and covariance of these vectors are given by, e.g. $\sigma_a^2 = \frac{1}{1-n} x_a x_a^T$ and $\sigma_a^2 b = \frac{1}{1-n} x_a x_b^T$. Following from this we can represent an entire matrix of the variance and covariance terms called the Covariance Matrix, given by

$$C_x = \frac{1}{1-n} XX^T \tag{4}$$

in which the diagonal terms represent the variance, and the off-diagonal terms represent all possible covariance combinations between measurement vectors.

The connection with the SVD lies precisely with this covariance matrix. The SVD of X *diagonalizes* the covariance matrix by representing X in a new set of orthogonal bases, $\hat{U}$ and V^*, which remove all redundancy, that is to say covariance, from the matrix as well as ordering the variance values on the diagonal of Σ. This is easily illustrated by calculating the covariance matrix of X projected onto the unitary transformation U^* from the Full SVD, $Y = U^*X$:

$$C_Y = YY^T = \frac{1}{n-1}(U^*X)(U*X)^T = \frac{1}{n-1}\Sigma^2 \tag{5}$$

Thus since, by **Theorem 1** the SVD is guaranteed to exist, this removal of redundancy is always possible.

Once this redundancy is removed, several powerful analytical tools are opened up. For one, we can use this PCA method to examine the how much of the total energy or variance is captured by each mode by examining their relative weights, where σ_e is a vector representing the proportion of energy contained in each value, and σ_v is the total variance:

$$\begin{aligned} \sigma_e &= \frac{\sigma_j}{\sum_{j=1}^{m} \sigma_j} \\ \sigma_v &= \frac{\sigma_j^2}{\sum_{j=1}^{m} \sigma_j^2} \end{aligned} \tag{6}$$

We can then use these weights alongside an examination of the bases to make observations about the dynamics of the original data. For example, we can project the orignal data in the direction of the orthogonal spatial modes $\hat{U}$ and make observations about what the behavior of the first, dominant mode is.

We can also reconstruct an approximation of the original data using the *Principal Orthogonal Modes* (POD). For example, the first POD would be represented by the orthonormal basis functions given by the first column of U with its behavior in time given by the first column of V^*. These can then be used to re-construct the best rank-1 representation of the data possible by multiplying $u_1 \sigma_1 v_1^T$. The second POD would use the first two columns of each basis and first two singular

[3] Kutz, Jose Nathan. Data-Driven Modeling and Scientific Computation. Methods for Complex Systems and Big Data. Oxford University Press, 2013, page 392, 399.

values, etc. We can then take into consideration the relative weight of the singular values, and make observations about what the most dominant mode is doing through inspecting the behavior of its principal spatial and temporal modes. The end result is that in many cases, a very accurate approximation of a high-order system can be rendered with comparitively little information.

3. Algorithm Implementation and Development

The overall approach of the algorithm was to use Matlab Computer Vision toolbox to track the moving paint bucket in the video and determine its location in every other frame of the data, process this data to make it amenable to SVD, and then to perform PCA on it and obtain data for figures. The toolbox failed in the case of the fourth set of videos, and therefore those position datasets were manually captured. Before moving on to the analysis, I will review how these approaches were implemented.

3.1 Location Capture

3.1.1 Computer Vision Toolbox

The Matlab Computer Vision toolbox was used to iterate through each frame of each video and locate the moving bucket by tracking points of interest. These points are obtained by the *detectMinEigenFeatures()* function, which implements the minimum eigenvalue algorithm created by Shi and Tomasi[4], which in turn is based directly on the Harris corner detection algorithm[5]. The code for this implementation is attached below, but it is used in the following way:

(1) Create toolbox objects for reading and playing the video.
(2) Read the first frame of the vieo and use use input to capture a region of interest.
(3) Detect points of interest in the initial frame using the minimum eigenvalues function.
(4) Create and initialize a tracking object.
(5) Iterate through each frame, taking the average x and y axis values for the points of interest.

This toolbox is quite powerful, and when the orientation of the bucket didn't change, it generally tracked the object extremely well. In cases where camera shake caused large amounts of blur, it did have problems tracking points. Some of these issues could be resolved by loosening the Bi-directional Error (values used ranged from 1 to 50), which allows for a larger number of pixels to separate consecutive locations of interest-points, though in Part 4 even this did not seem to work.

[4]Shi, J., and C. Tomasi, "Good Features to Track," Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, June 1994, pp. 593–600.

[5]Harris, C., and M. Stephens, "A Combined Corner and Edge Detector," Proceedings of the 4th Alvey Vision Conference, August 1988, pp. 147-151.

3.1.2 Manual Capture

Manual data capture was quite simple, and merely iterated through each frame of the video, displaying the frame and prompting the user to click on a spot with the mouse, which then saved the click as a set of (x,y) coordinates. While somewhat tedious, this provided excellent data. In practice, an exploration of other types of object tracking libraries would likely render manual capture unnecessary.

3.2 Data Formatting

Before proceeding to PCA the data had to be manipulated to account for three issues. First, all the y values in the data set had to be adjusted to account for the fact that the *imshow()* function in Matlab uses y axes that increase *downward* from the top. This simply meant observing that the window height was 480 pixels, and thus every y value had to be subtracted from this value.

Second, I had to be sure every position dataset started and ended in the same place, since each video was being begun or terminated by a different person. This simply meant checking to see what the behavior of the bucket was in the beginning of each video, then looking at data set lengths and removing all points leading up the buckets first low point (or right-ward point in the horizontal cases). The trailing points of each dataset were then removed to make them consistent with whichever one was shortest.

Third, the data had to be centered in order to perform PCA, and so each positional vector had its average subtracted from it. Once this was done, the data from each video set simply had to be stacked in the following way, where x_n and y_n represent the coordinate vectors from each of the three videos:

$$X = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{bmatrix} \tag{7}$$

3.3 PCA

Once the data is captured and organized, its analysis is straightforward. For each part, an SVD is performed on the X matrix using matlab's built-in *svd()* function to obtain the basis functions and singular values. The total energy of each mode is calculated using the approach described by equation (6) in section (2.2). Additionally, for producing figures of the time bases and projections, the length of the domain is scaled using he observation that each frame represents 1/15 of a second (two frames at 30 FPS), in order to produce an accurate time domain for plotting.

4. Computational Results

In all cases, decent results were achieved and the dynamics of each dataset was captured comparitively well by a small

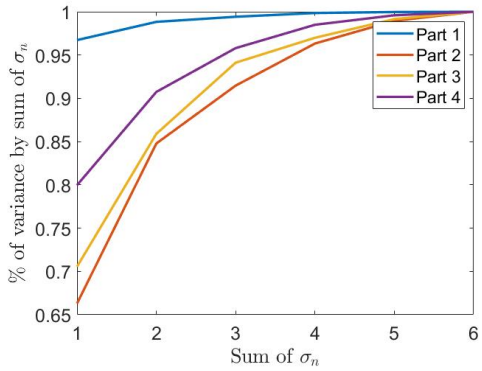

Figure 2. Comparison of cumulative variance across modes.

number of modes. Figure 2 shows the percentage of total variance captured by the first mode, first and second modes, etc., across each of the video sets. In all cases, over half the variance is captured by the first mode. Noting this, I will discuss each part individually.

4.1 Part 1 - Control Set

The first dataset serves as a control for the others, having little noise and little motion aside from the mass-spring behavior of the can. This is well-represented in the data, with about 74% of the total energy contained in the first mode. In particular, Figure 5, which shows the behavior of the basis over time, shows the first mode demonstrating well-defined oscillatory behavior. The other modes are less well-defined, but still appear to demonstrate some kind of oscillatory behavior, particularly the third mode which has some periodic resemblance to the first. This would lead me to speculate that this mode represents some aspect of the image data that is tied to the oscillation in some way, perhaps the changing impact of light sources on the bucket as it moves up and down. In general, there was a great deal of redundancy in the data, and the first three modes account for 91% of the energy, and over 99% of the variance in the data. Without prior knowledge of the phenomenon, one could look at the temporal oscillation and arrive at a somewhat accurate representation of the known physics of a spring-mass oscillator, namely $f(t) = Acos(\omega t + \omega_0)$, though one would want to add a damping term.

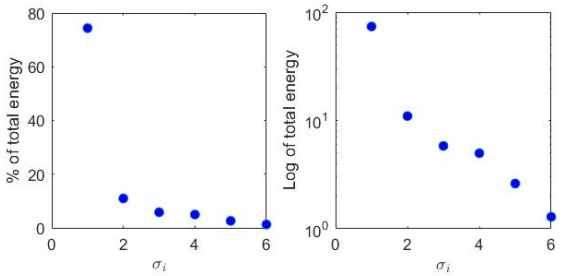

Figure 3. Percentage of total energy in sum of modes.

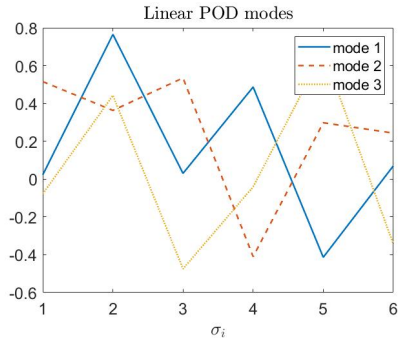

Figure 4. The first three spatial POD modes.

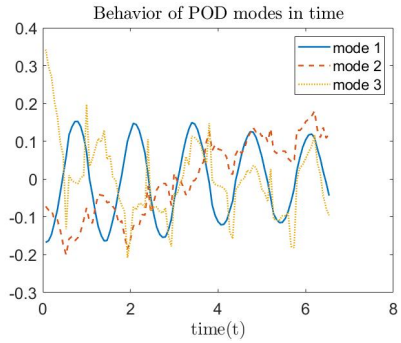

Figure 5. The first three temporal POD modes.

4.2 Part 2 - Camera Shake

In part 2 we had recordings in which there was significant camera shake. Not only did this introduce mostly horizontal noise, it also induced quite a bit of blurring in the recordings. In Figure 6 we see that the total energy accounted for in the first mode has dropped to about 41%, and in the temporal behavior of the modes we again have a dominant oscillatory mode, but now a second, fairly significant oscillating mode in Figure 8. This second oscillating mode probably corresponds to the back-and-forth of the camera shake, which was not particularly noisy, but actually fairly regular. Overall, this data was not particularly good as the Computer Vision Toolbox struggled with point-tracking due to the excessive blur, and tolerances had to be loosened to get the data. This is reflected by the fact that this set of videos captures the least amount of energy in the first 3 modes compared to the other sections.

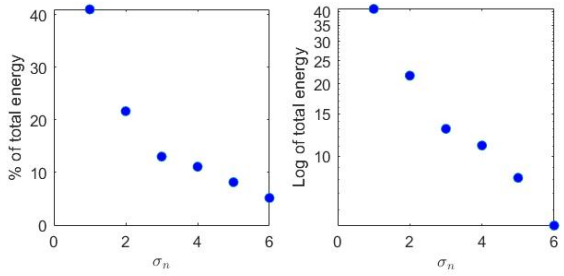

Figure 6. Part 2: Percentage of total energy in sum of modes.

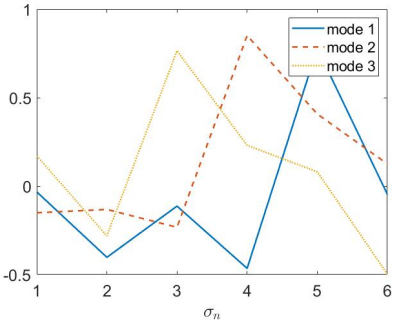

Figure 7. Part 2: The first three spatial POD modes.

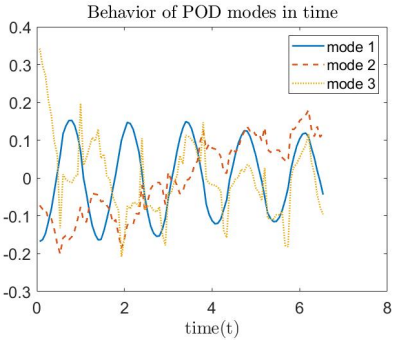

Figure 8. Part 2: The first three temporal POD modes.

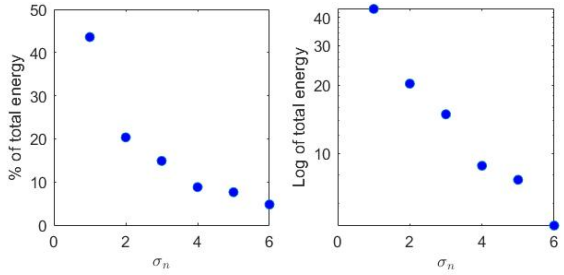

Figure 9. Part 3: Percentage of total energy in sum of modes.

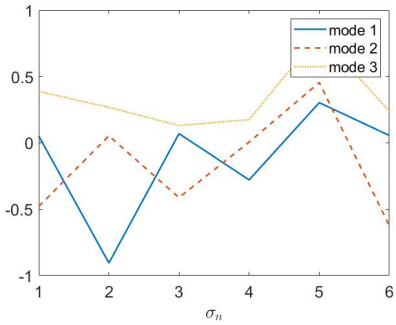

Figure 10. Part 3: The first three spatial POD modes.

4.3 Part 3 - Horizontal Displacement

In Part 3, the paint can has horizontal motion in the xy plane (if we are thinking of the up and down direction as z). This produced three fairly clear oscillations in the temporal displacement of the modes in Figure 11, the first of which still corresponds to the principle up and down motion. A quick glance at the video suggests that the second mode corresponds to the horizontal motion, which is periodic, and sometimes reaches its full amplitude at the same time as the up-and-down motion, and sometimes does not, which come across very precisely in the V modes. The third dominant mode is rather curious, as it seems to have the same period as the first, but starts slightly later. My initial speculation was that this mode corresponds to the person holding the camera bobbing up and down to instinctively track the moving object center-frame. Looking at the video confirms that there is indeed some motion of this kind, and is very small. If this is indeed the source, we could perhaps observe the ratio of the associated singular values to infer the size of the tracking motion relative to the paint can's motion.

4.4 Part 4 - Horizontal displacement and rotation

The data from Part 4 again produces a very clear dominant mode describing the spring-mass motion, and this is to be expected since this data set was manually collected and is very accurate (which feels like cheating given the miraculous capacity of the human brain to track objects). The second mode shows clear oscillations, and likly corresponds to horizontal swinging as in Part 3. However, looking at the video again, I would instead speculate that the second mode is in fact describing the spinning of the paint can, since it stops and reverses directions about 3 seconds into the video - precisely when the second mode momentarily flattens out before resuming. The same thing happens again toward the end of the video, and again the plot of the V^* modes corresponds to this.

5. Summary and Conclusions

In conclusion, the PCA of these redundant recordings produced precisely the sorts of results we would expect, and shows just how powerful the SVD is for eliminating redundancy and analyzing underlying dynamics in data. Part 1 in particular could be reduced to a rank 3 system - half the size of the oriinal data - and still retain nearly all the information. Furthermore, a rank 1 representation would have captured the majority of the dynamics we were interested in, namely the spring-mass dynamics.

The results from Parts 2 and 3 were also interesting, insofar as they showed how PCA can reveal and separate the different dynamics, but perhaps even more fascinating were the results from Part 4. If the interpretation above is corect, namely that the second mode captures the spinning o the bucket, it would seem the SVD can capture *three*-dimensional dynamics from a two-dimensional recording of a three-dimensional system. A control experiment of a freely spnning spring mass moving precisely up and down, and swinging precisely back and forth, would be fascinating, as I would surmise that PCA would very accurately describe each component with the first three modes.

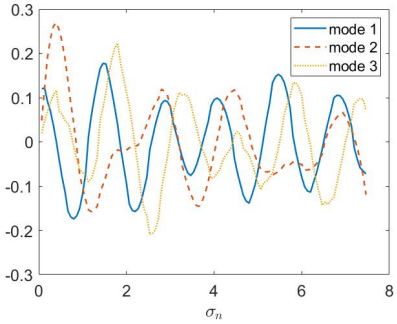

Figure 11. Part 3 The first three temporal POD modes.

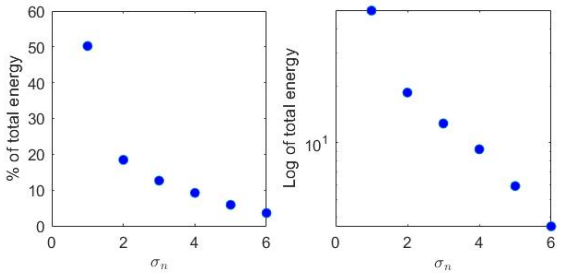

Figure 12. Part 4: Percentage of total energy in sum of modes.

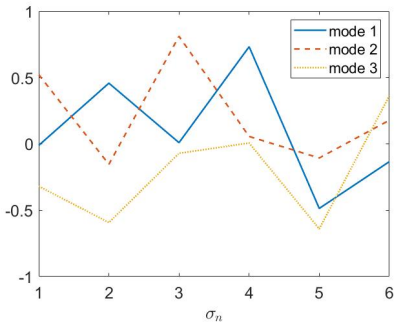

Figure 13. Part 4: The first three spatial POD modes.

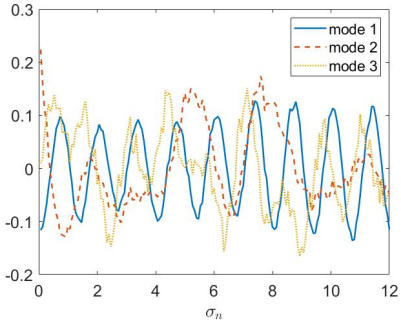

Figure 14. Part 4: The first three temporal POD modes.

Appendix 1 - Matlab Functions

Matlab Computer Vision Toolbox: The methods and objects used from this toolbox are taken directly from the example in the documentation at `https://www.mathworks.com/help/vision/ref/vision.pointtracker-system-object.html`. For the sake of brevity, and since I do not have an in-depth understanding of the method, I am giving this reference in lieu of an overview of the toolbox itself, which would likely be another paper unto itself.

imshow(a)* and *imread('filename': imread() reads in an image file whose name is passed as a string. The imshow() function then displays it. Note this must be an RBG or grayscale image, and the image is displayed with pixel axes, i.e. the y-axis extends down from the top.

readmatrix('filename.csv')* and *writematrix(var, 'filename.csv'): Reads and writes matrices to comma-separated value (csv) files.

cell(m,n): Creates a cell-array, which is a list-like object useful for storing objects of varying types, or matrices of varying dimensions.

diag(m): If passed a vector, returns a diagonal matrix with the vector on the diagonal; if passed a matrix, returns a vector of the matrix' diagonal values.

semilogy(x,y): Plots data with a log-scale on the y-axis (but not the x-axis).

cumsum(v): Returns a vector containing the cumulative sum of the argument. For example, if [a b c] is passed, it returns [a, a + b, a + b + c].

Appendix 2 - Code Listing

Please Note the repeated code used to process each section of analysis was omitted as they are identical.

```
1
2  %% ***HW 3 ***
3  clc; clear all; close all;
4  % *** Part 1 ***
5
6  % This code loads all the matlab video files and
7  % exports them as .avi files to use the computer Vision
8  % package
9
10 load_format = 'cam%d_%d';
11 % no use this in the loop load_output = sprintf(load_format, i, j);
12 mat_format = 'vidFrames%d_%d';
13 % again use this in loop mat_output = sprintf(mat_format, i, j)
14 output_format = 'cam%d_%d';
15 for i = 1:4
16     for j = 1:4
17         load(sprintf(load_format, i, j))
18         v = VideoWriter(sprintf(output_format, i, j));
19         open(v)
20         writeVideo(v, eval(sprintf(mat_format, i, j)));
21         close(v)
22     end
23 end
24
25 %% Capture the Region To be Tracked
26 clc; clear all; close all
27 videoFileReader = vision.VideoFileReader('cam2_4.avi');
28 videoPlayer  = vision.VideoPlayer('Position', [100, 100, 680, 520]);
29
30 objectFrame = videoFileReader();
31
32 figure;
33 imshow(objectFrame);
34 objectRegion = round(getPosition(imrect));
35 %% Display the Object Frame with a red box
```

```
objectImage = insertShape(objectFrame,'Rectangle',objectRegion,'Color','red');
figure;
imshow(objectImage);
title('Red box shows object region');
%% Detect Points of Interest
points = detectMinEigenFeatures(rgb2gray(objectFrame),'ROI',objectRegion);
pointImage = insertMarker(objectFrame,points.Location,'+','Color','white');
figure;
imshow(pointImage);
title('Detected interest points');

%% Iterate through the data and capture the points

% Create The Tracker Object
tracker = vision.PointTracker('MaxBidirectionalError',50, 'MaxIterations', 100);
% Initialize the object
initialize(tracker,points.Location,objectFrame);
point_list = [];
count = 1;
while ~isDone(videoFileReader)
      count = count + 1
      frame = videoFileReader();
      [points,validity] = tracker(frame);
      out = insertMarker(frame,points(validity, :),'+');
      videoPlayer(out);
      if mod(count, 2) == 1
          x_ave = mean(points(:,1));
          y_ave = mean(points(:,2));
          point_list = [point_list; [x_ave y_ave]];
      end
end

%% Write out to a file
writematrix(point_list, 'coords2_2.csv')

%% Manual Capture For Problematic Videos
% I manually processed 1_4, 2_4, 3_4
clc; clear all; close all

load cam1_4
count = 1;
point_list = [];
t_length = size(vidFrames1_4);
for i = 1:t_length(4)
    count = count + 1;
    if mod(count, 2) == 1

        figure()
        imshow(vidFrames1_4(:,:,:,i)), title('Click a consistent spot')
        [x, y] = ginput(1);
        point_list = [point_list; [x y]];
        close all
    end

end

%% Plot to check data
close all
    figure(1)
    axis([0 640 0 480])
test_plot = plot(point_list(1,1), point_list(1,2));
for t = 1:length(point_list(:,1))
    figure(1)
    axis([0 640 0 480])
    hold on
    p = plot(point_list(t,1), point_list(t,2),'o');
    pause(0.02)
    delete(p)
end
%% Write to csv
```

```
writematrix(point_list, 'coords1_4_manual.csv')

%%
count = 1;
new_list = []
for i = 1:356
    count = count +1;
    if mod(count,2) == 1
     new_list = [new_list; point_list(i, :)];
    end

end


%% Process all videos and store all the data
clc; clear all; close all
% Cell array of all position data. 1-4 is part 1,
% 5-8 part 2 etc.
master_data_list = cell(16, 1);

% Formats for sprintf calls
load_format = 'cam%d_%d';
% no use this in the loop load_output = sprintf(load_format, i, j);
mat_format = 'vidFrames%d_%d';
% again use this in loop mat_output = sprintf(mat_format, i, j)
output_format = 'cam%d_%d.avi';

% Outer loop iterates through each part of the HW, i.e. i = 1
% corresponds to part 1.
index = 0;
for i = 1:4
    index = index + 1;
    % Inner loop iterates through each camera
    for j = 1:4
        videoFileReader = vision.VideoFileReader(sprintf(output_format, i, j));
        videoPlayer  = vision.VideoPlayer('Position', [100, 100, 680, 520]);

        objectFrame = videoFileReader();

        % Capture a box in which to track an object
        figure('units','normalized','outerposition',[0 0 1 1]);
        imshow(objectFrame);
        title('Please draw a box around the object')
        objectRegion = round(getPosition(imrect));
        close all

        objectImage = insertShape(objectFrame,'Rectangle',objectRegion,'Color','red');
        figure;
        imshow(objectImage);
        title('Red box shows object region');
        pause

        close all
        points = detectMinEigenFeatures(rgb2gray(objectFrame),'ROI',objectRegion);
        pointImage = insertMarker(objectFrame,points.Location,'+','Color','white');
        figure;
        imshow(pointImage);
        title('Detected interest points');
        close all

        % Create The Tracker Object
        tracker = vision.PointTracker('MaxBidirectionalError',1);
        % Initialize the object
        initialize(tracker,points.Location,objectFrame);
        point_list = [];

        while ~isDone(videoFileReader)
              frame = videoFileReader();
              [points,validity] = tracker(frame);
              point_list = [point_list; points];
```

```
176             end
177             master_data_list{index} = point_list;
178         end
179     end
180
181
182     %% HW 3 SVD an Analysis
183
184     % Process the computer vision captured values by converting
185     % the y-coords. They have to be changed from y to 480 - y.
186     clc; clear all; close all
187     load_format = 'coords%d_%d';
188
189     % The master_data_mat matrix will be organized like the two columns of:
190     % 1_1 2_1 3_1 1_2 2_2 3_2 1_3 2_3 3_3 1_4 2_4 3_4
191     master_data_mat = {};
192     count = 0;
193     for j = 1:4
194         for i = 1:3
195             count = count + 1;
196
197             % Switch the y-matrix from pixel form to
198             % standard y coordinates due to imshow() function.
199             mat = readmatrix(sprintf(load_format, i, j));
200             mat = [mat(:,1) 480 - mat(:,2)];
201             master_data_mat{1,count} = mat;
202
203         end
204     end
205     %% Part 1 Processing.
206     % Start each position vector at the bottom of the trajectory. For 1_1, 2_1
207     % this means a y min, and for 3_1 an x_max
208     mat = master_data_mat{1};
209     mat = mat(14:end,:);
210     master_data_mat{1,1} = mat;
211
212     mat = master_data_mat{2};
213     mat = mat(19:end,:);
214     master_data_mat{1,2} = mat;
215
216     mat = master_data_mat{3};
217     mat = mat(14:end,:);
218     master_data_mat{1,3} = mat;
219
220
221     %% Chop off the end to make them all the same lengths
222     for i = 1:3
223         mat = master_data_mat{i};
224         mat = mat(1:98, :);
225         master_data_mat{1, i} = mat;
226     end
227
228     %% To do: loop through the cell list and subtract the average from each measurement vector.
229
230     for i = 1:12
231         A = master_data_mat{i};
232         A(:,1) = A(:,1) - mean(A(:,1));
233         A(:,2) = A(:,2) - mean(A(:,2));
234         master_data_mat{1,i} = A;
235     end
236
237     %%% Section for plotting to check orientations
238     %
239     % point_list = master_data_mat{1};
240     % figure(1)
241     % axis([0 640 0 480])
242     % test_plot = plot(point_list(1,1), point_list(1,2));
243     % for t = 1:length(point_list(:,1))
244     %     figure(1)
245     %     axis([0 640 0 480])
```

```
%     hold on
%     p = plot(point_list(t,1), point_list(t,2),'o');
%     pause(0.05)
%     delete(p)
% end
%% Part 1 Analysis

A1 = [master_data_mat{1}';
      master_data_mat{2}';
      master_data_mat{3}'];
[U1, S1, V1] = svd(A1);
sig = diag(S1);

% Calculate time-step of each data point. The .avi files are all 30 FPS
% and I used every other frame, so each data point represents 1/15 seconds.
t = (1/15)*(1:length(A1(1,:)));
% energy and log energy plots
figure(1)
subplot(1,2,1)
plot((sig / sum(sig)) * 100,'o', 'MarkerFaceColor', 'b')
xlabel('$\sigma_i$', 'Interpreter', 'latex')
ylabel('% of total energy')
set(gca, 'Fontsize', [10])
axis('square')
subplot(1,2,2)
semilogy((sig / sum(sig)) * 100,'o', 'MarkerFaceColor', 'b')
xlabel('$\sigma_i$', 'Interpreter', 'latex')
ylabel('Log of total energy')
set(gca, 'Fontsize', [10])
axis('square')
sgtitle('Percentage of total energy in each $\sigma$', 'interpreter', 'latex')

e_1 = sig(1)/sum(sig); e_2 = sum(sig(1:2)) / sum(sig); e_3 = sum(sig(1:3))/sum(sig);
e_list1 = [e_1; e_2; e_3];

% project onto U
Y1_U = U1.'*A1;
% Plot result
figure(2)
plot(t, Y1_U(1,:), t, Y1_U(2,:), t, Y1_U(3,:))
xlabel('time(seconds)', 'Interpreter', 'latex')
legend('mode 1', 'mode 2', 'mode 3', 'Location', 'NorthEast')
set(gca, 'Fontsize', [15])
title('Projection of $\sigma_i$ onto $u_i$', 'interpreter', 'latex')

% Spatial Modes
figure(3)
x = 1:6;
plot(x, U1(:,1),'-', x, U1(:,2),'--', x, U1(:,3), ':', 'linewidth', 1.5)
xlabel('$\sigma_i$', 'Interpreter', 'latex')
legend('mode 1', 'mode 2', 'mode 3', 'Location', 'NorthEast')
set(gca, 'Fontsize', [15])
title('Linear POD modes', 'interpreter', 'latex')

% Time Behavior
figure(4)
plot(t, V1(:,1), '-', t, V1(:,2),'--', t, V1(:,3),':', 'linewidth', 1.5)
xlabel('time(t)', 'Interpreter', 'latex')
legend('mode 1', 'mode 2', 'mode 3', 'Location', 'NorthEast')
set(gca, 'Fontsize', [15])
title('Behavior of POD modes in time', 'interpreter', 'latex')
```