AMATH 482 Assignment 3

Megan Freer

February 24, 2021

Abstract

One very powerful mathematical tool that makes use of linear algebra is the Singular Value Decomposition (SVD), as well as the closely related Principal Component Analysis (PCA). These allow for the decomposition of a system/matrix in order to further analyze the system, such as to find its rank. Spring-mass systems (meaning a mass attached to a spring, with the spring set into 1D motion) provide an interesting physical system that the SVD/PCA can be used to analyze. We can utilize the SVD and PCA on spring-mass position data from three different cameras in order to find if the mass is truly moving in 1D.

1 Introduction and Overview

Much of computational mathematics is based on linear algebra. For example, the Singular Value Decomposition (SVD) and the accompanying Principal Component Analysis (PCA) can be used to decompose a matrix/system and find more information about it, such as its rank.

As an example of using SVD/PCA, we will be looking at a spring-mass system. A paint can is suspended from a large spring and is set into motion so that the paint can is moving up and down vertically, meaning in a one-dimensional direction. We are then setting up three cameras around the spring-mass system and collecting videos from these different angles. In the frame of each video, there are two dimensions. These three videos are then recorded for four different test cases: the ideal case, noisy case, horizontal displacement, and horizontal displacement and rotation. These allow us to analyze differences in the SVD when the paint can is swinging side to side (as well as the typical up and down) and/or rotating, as well as when the camera is shaky.

We will be using MATLAB to find the position of the paint can at each video frame and to perform the Singular Value Decomposition function on this position data. This then allows us to analyze the differences between each of the four test case scenarios.

2 Theoretical Background

One very powerful mathematical tool that makes use of linear algebra is the singular value decomposition (SVD). Thinking about the SVD geometrically, it is important to recall that a matrix represents a linear transformation: it takes a vector and transforms it to another vector. In general, matrix multiplication will rotate and/or scale a vector. For example, the rotation matrix is given by:

$$R_\theta = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{1}$$

Finding $R_\theta x$ gives the original vector x rotated counterclockwise around the origin by an angle of θ. This gives an idea of how matrix multiplication can be used to change the locations of vectors in space. Another important property to keep in mind is that unitary and orthogonal matrices do not stretch or compress a vector. This means that the Euclidean norm (2-norm) of a vector $x \in \mathbb{R}^N$ given :

$$||x||_2 = \sqrt{\sum_{n=1}^{N} |x_n|^2} \tag{2}$$

is the same as the Euclidean norm of Ax for any unitary matrix. This means that $||x||_2 = ||Ax||_2$ when A is unitary. To scale a vector, we use another matrix of the form:

$$A = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} \tag{3}$$

where $\alpha > 0$. Thinking geometrically about multiplication of the unit circle by a matrix, we can label the principal semiaxes of the resultant ellipse as $\sigma_1 u_1$ and $\sigma_2 u_2$, where σ_1 and σ_2 are the lengths of the vectors and u_1 and u_2 are the unit vectors that point in the directions of the axes. There are then some unit vectors v_1 and v_2 such that:

$$Av_1 = \sigma_1 u_1, Av_2 = \sigma_2 u_2, \tag{4}$$

which in matrix form is equivalent to:

$$AV = U\Sigma \tag{5}$$

where V is a unitary matrix with v_1 and v_2 as its columns, U is a unitary matrix with u_1 and u_2 as its columns, and Σ is a diagonal matrix with σ_j on its diagonals. Since V is unitary and therefore invertible, we can isolate for A to get:

$$A = U\Sigma V^* \tag{6}$$

This idea generalizes to matrices of arbitrary size. The same equation (6) follows for the singular value decomposition of the matrix A, where $U \in \mathbb{R}^{mxm}$ and $V \in \mathbb{R}^{nxn}$ are unitary matrices, and $\Sigma \in \mathbb{R}^{mxn}$ is diagonal. The values σ_n on the diagonal of Σ are called the singular values of the matrix A, the vectors u_n that make up the columns of U are called the left singular vectors of A, and the vectors v_n that make up the columns of V are called the right singular vectors of A.

Some important properties of the SVD include that the singular values are uniquely determined and are always non-negative real numbers, and the singular values are always ordered from largest to smallest along the diagonal of Σ. Furthermore, the number of nonzero singular values is the rank of A. Letting r=rank(A), we have that $\{u_1, u_2, ..., u_r\}$ is a basis for the range of A and $\{v_r + 1, ..., v_n\}$ is a basis for the null space of A.

In order to understand the usefulness of the SVD particularly for low-dimensional approximations, we need to understand the theorem that if A is a matrix of rank r, then A is the sum of r rank 1 matrices:

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^* \tag{7}$$

The product $u_j v_j^*$ is called an outer product and is the same size as A. A matrix that comes from an outer product has rank 1. This representation of A by the sum of rank 1 matrices give a good method for approximating A, particularly when the rank r is large. We would not have to sum all the way up to r, but rather we could sum the first N terms to get the best rank N approximation that is possible for A. As a Theorem, for any N such that $0 \leq N \leq r$, we can define the partial sum:

$$A_N = \sum_{j=1}^{N} \sigma_j u_j v_j^* \tag{8}$$

This theorem means that out of all the matrices that are rank N or less, the best approximation of A is given by A_N. This allows us to find approximations of data, which is important because the SVD is able to tell how to keep the most amount of information while keeping the fewest number of dimensions to the data. The SVD does this by capturing the important trends in the data.

The SVD is just a way of factoring matrices, but it turns out to be very useful for many applications. One similar technique that ends up just being the SVD is Principal Component Analysis, or PCA. The SVD can be used to produce low-rank approximations of a data set, and Principal Component Analysis, or PCA, follows the same idea of the SVD, but has to do more with low-rank approximations. For example, when coding this in MATLAB, there is no use in obtaining the full SVD, which includes a number of zero rows

padded into the Σ matrix. When thinking about PCA, we can think about how we could compute all the variances and covariances between the rows of X with one matrix multiplication:

$$C_x = \frac{1}{n-1} X X^T \begin{bmatrix} \sigma_a^2 & \sigma_{ab}^2 & \sigma_{ac}^2 & \sigma_{ad}^2 \\ \sigma_{ba}^2 & \sigma_b^2 & \sigma_{bc}^2 & \sigma_{bd}^2 \\ \sigma_{ca}^2 & \sigma_{cb}^2 & \sigma_c^2 & \sigma_{cd}^2 \\ \sigma_{da}^2 & \sigma_{db}^2 & \sigma_{dc}^2 & \sigma_d^2 \end{bmatrix} \tag{9}$$

where X is:

$$X = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \tag{10}$$

In equation (9), we notice that C_X is a square symmetric matrix, which is called the covariance matrix. The goal of principal component analysis is to find a new set of coordinates (a change of basis) so that the variables are now uncorrelated. We would also want to know which variables have the largest variance because these contain the most important information about the data. Therefore, we want to diagonalize the matrix so that all off-diagonal elements (covariances) are zero:

$$C_X = V \Lambda V^{-1} \tag{11}$$

The basis of eigenvectors contained in V are called the principal components and are uncorrelated since they are orthogonal. The diagonal entries of Λ, the eigenvalues of C_x, are the variances of the new variables. This equation is very similar in form to that of the SVD.

One way of analyzing how "good" a certain rank approximation is, is to determine how much energy from the full system is contained in each mode. Assuming the full matrix is rank r, we measure the energy contained in the rank-N approximation by:

$$energy_N = \frac{\sigma_1^2 + \sigma_2^2 + ...\sigma_N^2}{\sigma_1^2 + \sigma_2^2 + ...\sigma_r^2} = \frac{||X_N||_F^2}{||X||_F^2} \tag{12}$$

We are then able to calculate the energies for each of the rank approximations to see how the energies change with the rank approximations.

Thinking now about the Spring-Mass system that is being worked with for this assignment, we have a mass (paint can in our videos) that is hanging from a spring and the mass bobs up and down. The equation for the vertical displacement of the mass is given by:

$$z(t) = Acos(\omega t + \varphi) \tag{13}$$

where A and φ are determined by the initial displacement and velocity of the mass and the frequency ω can be found from the constants in the differential equation representing the system. The important note here is that the motion is one-dimensional, only going up and down in the z direction.

In terms of cameras, we are setting up three cameras around the spring-mass system and collecting videos. In the frame of each video, there are two dimensions. For camera 1, we can denote them (x_a, y_a) at each instance in time. If we do the same for camera 2 and 3, we can make vectors of all the position at each time to get the matrix:

$$X = \begin{bmatrix} x_a \\ y_a \\ x_b \\ y_b \\ x_c \\ y_c \end{bmatrix} \tag{14}$$

Using the SVD and this matrix, we will now be able to weed out redundancies. In this system, we expect that there should be only one nonzero singular value (representing the fact that the motion is 1D). But since no video recording or data collecting is perfect, we expect that there will be some noise, as well as noise due to movement in other directions, such as horizontally. Therefore we should really expect that there is one nonzero singular value that is significantly larger than the others, representing the dominant up-down motion of the paint can.

3 Algorithm Implementation and Development

Algorithms involving the SVD/PCA were used to find out more information about the spring-mass system. Starting with the algorithm for the ideal case data set, the code starts by loading in the camera names and video names for the three videos. The code also defines the number of cameras, 3, and initializes the cells where the position data from each camera will eventually be stored. Next, we loop through each of the cameras and load in the camera and corresponding video. We then find the number of frames in the video, as well as the size of the video in the x and y direction in terms of pixels. We next loop through each of the frames of the video and convert the image to a gray-scale image in order to make the entire video gray-scale. The code then creates a figure displaying the gray-scale image of the first frame, and the user is able to draw a rectangle on the image to define which area of the image they want to analyze. This eliminates noise from other areas of the image and allows the analysis to just focus on the area that the paint can is in over the course of the video.

The code then loops through each of the video frames (besides the last one) and finds the difference between that frame and the next frame in time in order to see the areas of the video that are moving the most. In theory, only the paint can would be moving up and down, so the current location of the paint can could be found from these areas that move the most. There are more areas that are moving from frame to frame though, such as the legs of the person who is holding the spring and slight shakiness from the camera. We next binarize the image of the differences between each frame based on a threshold in order to make the data easier to analyze. This resultant image is also cropped to the correct size based on the user-selected rectangle from earlier. In order to find the average position of the paint can (approximately the center of mass), we returned all the x coordinates and y coordinates that are 1 (meaning showing up white in the image), then averaged along those x coordinates and y coordinates to get an (x,y) position for the paint can in each frame. At this point, the black and white frame is plotted with a red star representing the calculated paint can position plotted on top. Since this is all within a loop through the frames, MATLAB quickly plots the frames on the same image so that the user can see visually that the program is finding approximately the correct position of the paint can.

At this point, many of the video frames now show a white area where the paint can is and a black background, but there are also many of the video frames that are completely black. In this case, no position data was able to be found for the paint can at this point. To work around this, we found the locations of these zeros in the position data and interpolated the position data to find the approximate location of the paint can, based on its location in the frames leading up to and after that frame. This was done to find the approximate position in both the x and y direction. The next issue is that each of the cameras is started/stopped at a slightly different time, meaning that each camera has a video with a different number of frames and each of the videos is slightly out of sync. In order to align these videos, we found the index at which the position data was at a minimum in the vertical direction, meaning the point when the paint can was at its lowest at the beginning of the recording. The position data was then cropped to start at this index. Next, we removed any NaN values and saved the position data for the camera.

Outside of that large camera loop, we find the minimum length of the arrays representing the position data for each camera, and then cropped all of the camera position data to that length. This allows us to further align the three cameras' data so that the position data is the same length and represents the same points in time. We next combined the (x,y) data for all three cameras in order to make an 6-by-n X matrix, where n is the number of frames in the video that were aligned. When making this matrix, we also subtract off the mean of that row. This is due to the fact that for a PCA to work properly, we must subtract the mean from each of the data dimensions. This also works to further align the position data from the three cameras in space (since the paint can was likely at slightly different pixel values in each camera's video). We were then able to perform the `svd` to find U, S/Σ, and V, as well as the resultant nonzero singular values. We next calculated the energy contained in the rank-r approximation for each rank from 1 to 6 (the highest possible for this system).

At this point, we were able to plot all of these values that were calculated. The code makes a scatter plot of the horizontal position data over time, as well as a scatter plot of the vertical position data over time. The code also makes a bar chart of the various σ values in order, as well as the energies contained in each rank approximation. In order to find all of this for the next test case (such as the noisy case next), the user can

comment out the current test case at the top of the MATLAB code and uncomment the next test case. The same code can then be run again to produce the same figures and information for that new test case.

4 Computational Results

The MATLAB algorithm was able to produce position data in the horizontal and vertical directions for each of the four test conditions (Ideal Case, Noisy Case, Horizontal Displacement, and Horizontal Displacement and Rotation). The computational results for the vertical position data can be see in Figure 1, while the computational results for the horizontal position data can be seen in Figure 2. Particularly in the ideal case, we are able to see the oscillations in the vertical position for all three cameras. In comparison, the noisy case appears to have very noisy position data that can be seen to generally be oscillating when looking closer, but looks very much like noise at first glance. The three cameras also do not line up well in their position data in the noisy case. The horizontal displacement and the horizontal displacement and rotation cases both look better than the noisy case, as the oscillations are more distinct and synced up, but are not quite as clear as in the ideal case. Looking at the horizontal position data, we would generally expect to see a flat line at 0, meaning that the horizontal position of the paint can did not change over time, since in theory the paint can is only moving up and down vertically. This is relatively true for the ideal case, where there is some variation in horizontal position, but only in the range of ± 10 pixels. The noisy case appears to have even more changes in horizontal position (± 30 pixels), likely due to the shakiness of the camera side to side. The last two test cases where the paint can is swinging side to side horizontally (as well as up and down vertically) have even more horizontal displacement, in the range of ± 50 pixels. This is as to be expected, since the paint can can be seen to be moving horizontally in the video. The oscillations in the horizontal position can be seen for these two test cases, although they are not perfectly synced up between cameras, likely due to the different viewing angles.

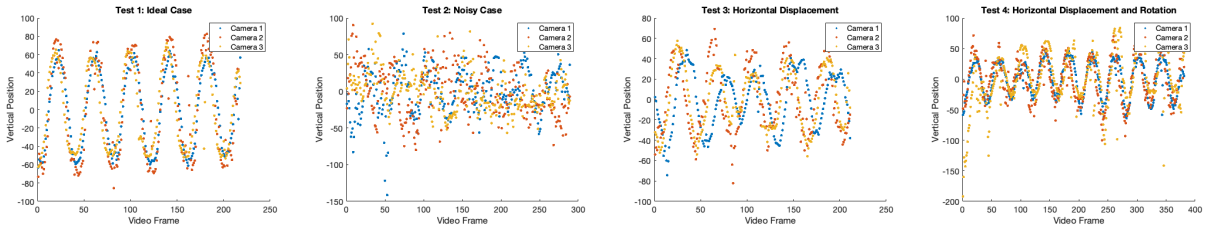

Figure 1: The vertical position of the paint can over the course of each video frame for the four test cases: ideal case, noisy case, horizontal displacement, and horizontal displacement and rotation. Each color represents a different camera.

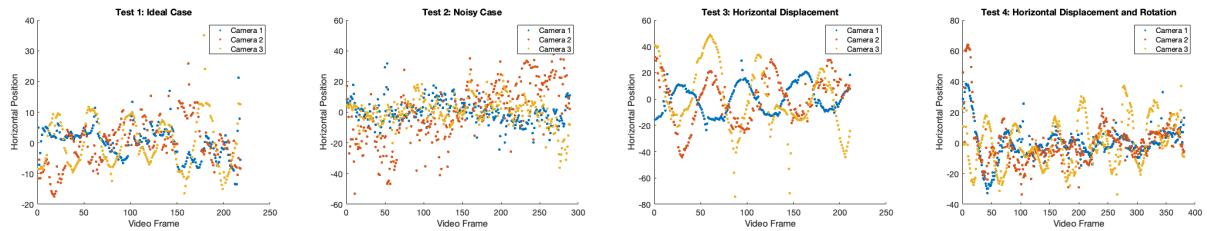

Figure 2: The horizontal position of the paint can over the course of each video frame for the four test cases: ideal case, noisy case, horizontal displacement, and horizontal displacement and rotation. Each color represents a different camera.

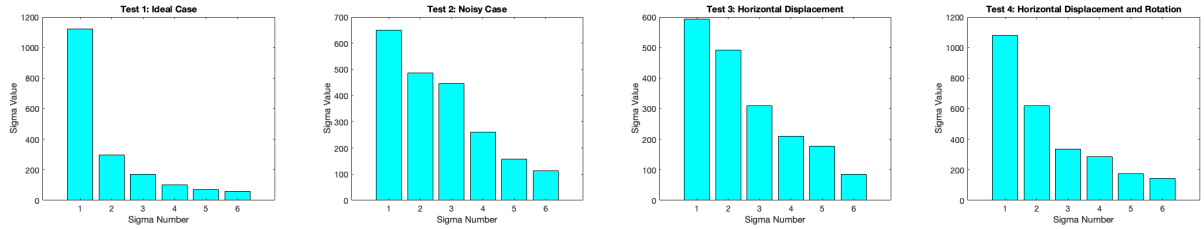

Figure 3: The σ values (nonzero singular values) for each σ_n for the four test cases: ideal case, noisy case, horizontal displacement, and horizontal displacement and rotation.

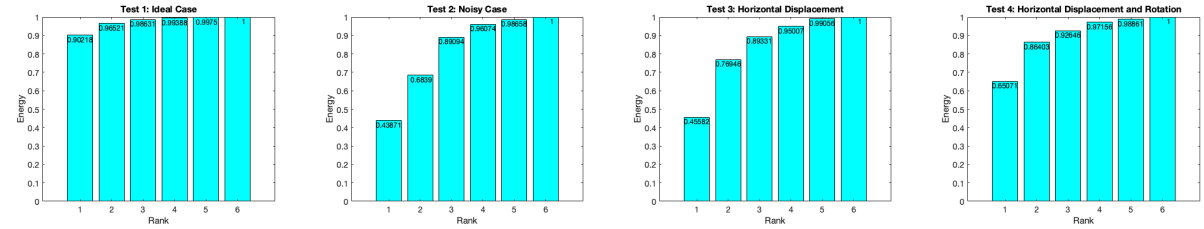

Figure 4: The cumulative energy accounted for by each rank, from 1 to 6, for the four test cases: ideal case, noisy case, horizontal displacement, and horizontal displacement and rotation. The higher the energy, the more of the energy of the system that is accounted for by that rank approximation.

This position data was then able to produce computational results for the U, Σ, and V matrices, which could be used to find the σs and energies, as can be seen in Figures 3 and 4. We are expecting only one nonzero singular value, because in theory the system should have a rank of 1, because the paint can is moving in a 1D motion (only up and down). However, we know that this will likely not be the case due to camera shakiness, errors in finding the paint can position, and the swinging side to side of the paint can as it moves up and down. This leads us to the believe that there will be one dominant value for σ, while the others are significantly smaller, representing the dominant up-down motion of the paint can. This is true in the ideal case, as can be seen in Figure 3, where σ_1 is significantly higher than the other σ values. However, you can also see in Figure 3 that for the next three test cases, the difference between the first σ and the next σ gets smaller. The case with the shaky camera has the least difference between σ_1 and σ_2. A similar pattern follows with the energies, with Test 1: Ideal Case having the highest energy for σ_1, then Test 4: Horizontal Displacement and Rotation with the next highest energy for σ_1, then Test 3: Horizontal Displacement, then followed closely by Test 2: Noisy Data. These energy values represents how much energy from the full system is contained in the 1st mode/the energy contained in the rank-1 approximation. This means that the ideal case data set is the closest to a rank 1 system. This seems true, because the ideal system did not have the purposeful camera shakiness or horizontal displacement of the paint can that the other test scenarios did. Therefore, this system was the closest to 1D motion of the paint can and therefore the closest to rank 1.

5 Summary and Conclusions

In conclusion, Singular Value Decomposition and Principal Component Analysis are useful tools, particularly for analyzing spring-mass position data from various cameras. These mathematical tools were able to help us reduce dimensionality and find the approximate rank of the system. We were able to demonstrate how MATLAB can be used as a way to implement these algorithms in order to analyze position data for a spring-mass system from various cameras. The ideas of SVD and PCA can be applied to many other applications in order to keep the most amount of information while keeping the fewest number of dimensions, unlocking the possibility for faster analysis in many future applications.

Appendix A MATLAB Functions

The following MATLAB functions are implemented in this code:

- `load(filename)` loads variables in the MAT-File into the MATLAB workspace to load in the camera data.
- `eval(expression)` evaluates the MATLAB code in expression in order to load in and play the video.
- `szdim = size(A,dim)` returns the length of dimension dim in the matrix A.
- `length(obj)` returns the length of an object, such as the length of an array.
- `X = zeros(sz1,...,szN,'uint8')` returns an sz1-by-...-by-szN matrix of zeros of data type `uint8`.
- `I = rgb2gray(RGB)` converts the truecolor image RGB to the grayscale image I.
- `imshow(I)` displays the grayscale image I in a figure.
- `rect = getrect` lets the user select a rectangle in the current axes using the mouse. When the user finishes selecting the rectangle, `getrect` returns information about the position and size of the rectangle in `rect` in the form `[xmin ymin width height]`.
- `Z = imabsdiff(X,Y)` subtracts each element in array Y from the corresponding element in array X and returns the absolute difference in the corresponding element of the output array Z. This is used to find the parts of the image that are moving the most from frame to frame.
- `y = round(x)` rounds x to the nearest integer. This is used to round the pixels selected by the user to integer values.
- `BW = imbinarize(I,T)` creates a binary image from image I using the threshold value T. This was used to further extract the moving paint can and make the position easier to analyze.
- `[row, col] = find(X)` returns the row and column subscripts of each nonzero element in array X, or those that evaluate to True from the expression in parentheses.
- `M = mean(A)` returns the mean of the elements of A along the first array dimension whose size does not equal 1.
- `plot(X, Y, LineSpec)` creates a 2D line plot of the data in Y versus the corresponding values in X. `LineSpec` represents the ability to change the marker size and style for clearer graphs.
- `drawnow` updates image to each video frame in order to see changes between frames immediately.
- `v = nonzeros(A)` returns a full column vector of the nonzero elements in A. This is used to find the frames that did not fully evaluate to 0.
- `vq = interp1(x, v, xq)` returns the interpolated values of a 1D function at specific query points using linear interpolation. Vector x contains the sample points, v contains the corresponding values, and xq contains the coordinates of the query points. This was used to find the estimated paint can position for the frames that evaluated to 0.
- `[min_value, min_index] = min(A)` returns the minimum value in the matrix A and the index into A that corresponds to this value.
- `R = rmmissing(A)` removes missing entries, such as NaN, from an array in order for the paint can position data to be analyzed correctly.
- `scatter(x,y)` creates a scatter plot with circles at the locations specified by the vectors x and y. The markers can also be modified to be different shapes and sizes.

- `[U,S,V] = svd(X, 'econ')` produces an economy-size singular value decomposition of matrix X in order to find the matrices that evaluate to X=U*S*V'.
- `sig = diag(S)` returns a column vector of the main diagonal elements of S.
- `sum(A)` returns the sum of the elements of A along the first array dimension whose size does not equal 1.
- `bar(x,y)` creates a bar graph with one bar for each element in y, with the bars drawn at the locations specified by x. The color of the bars can also be specified.
- `text(x,y,txt)` adds a text description to data points in the current axes using the text specified by `txt`. This was used to label the energy values on the energy bar chart.

Appendix B MATLAB Code

```
% Megan Freer
% AMATH 482
% Assignment 3

clear all; close all; clc;

% Note: Uncomment a section to create plots for that test set

% Test 1: Ideal Case
camera_names = ["cam1_1.mat", "cam2_1.mat", "cam3_1.mat"];
video_names = ["vidFrames1_1", "vidFrames2_1", "vidFrames3_1"];
test_title = 'Test 1: Ideal Case';

% % Test 2: Noisy Case
% camera_names = ["cam1_2.mat", "cam2_2.mat", "cam3_2.mat"];
% video_names = ["vidFrames1_2", "vidFrames2_2", "vidFrames3_2"];
% test_title = 'Test 2: Noisy Case';

% % Test 3: Horizontal Displacement (also swings side to side)
% camera_names = ["cam1_3.mat", "cam2_3.mat", "cam3_3.mat"];
% video_names = ["vidFrames1_3", "vidFrames2_3", "vidFrames3_3"];
% test_title = 'Test 3: Horizontal Displacement';

% % Test 4: Horizontal Displacement and Rotation
% camera_names = ["cam1_4.mat", "cam2_4.mat", "cam3_4.mat"];
% video_names = ["vidFrames1_4", "vidFrames2_4", "vidFrames3_4"];
% test_title = 'Test 4: Horizontal Displacement and Rotation';

number_cameras = 3;
positions = cell(1, 3);

for i = 1:number_cameras
    % Load in video
    load(camera_names(i));
    video = eval(video_names(i));

    % Finding number of frames (4th dimension of video)
    number_frames = size(video,4);
```

```
% Find size in x and y direction (# pixels)
pixels_x = size(video,1);
pixels_y = size(video,2);

% Initialize matrix to store grey scale frames
grey_scale = zeros(pixels_x, pixels_y, number_frames, 'uint8');

% Loop through frames and convert to grey scale
for j = 1:number_frames
    grey_scale(:,:,j) = rgb2gray(video(:,:,:,j));
end

% Have user select part of image to be studied
figure()
imshow(grey_scale(:,:,1))
rect = getrect;

% Initialize difference image and position data
difference = zeros(pixels_x, pixels_y, 'uint8');
position = zeros(2, number_frames-1);

% Loop through each frame
for j = 1:number_frames-1
    % Calculate difference between this frame and next frame to find
    % moving parts
    difference(:,:) = imabsdiff(grey_scale(:,:,j), grey_scale(:,:,j+1));

    % Extract information from rect
    x_min = round(rect(1));
    y_min = round(rect(2));
    x_max = round(rect(1)+rect(3));
    y_max = round(rect(2)+rect(4));

    % Binarize difference image
    binarized = imbinarize(difference(y_min:y_max, x_min:x_max), 0.3);

    % Returns all the (x, y) coordinates on the image that are white/1
    [x_coordinates, y_coordinates] = find(binarized);
    if length(x_coordinates) >= 1
        % Save the average location of the white space
        position(:, j) = [mean(x_coordinates), mean(y_coordinates)];
    end

    % Plot calculated position on top of image
    imshow(binarized)
    hold on
    plot(position(2,j), position(1,j), 'rp', 'MarkerFaceColor', 'red',...
         'MarkerSize', 24)
    hold off
    drawnow
end

% At this point, there are many zeros still in the position data
% (black frames) -> need to replace these zeros
```

```
    frames_array = 1:length(position);
    zero_positions = find(position(1,:) == 0); % where to find estimates
    nonzero_positions = find(position(1,:) ~= 0);
    position_no0 = zeros(2, length(nonzeros(position(1,:)))); % resets size for each loop
    position_no0(1,:) = nonzeros(position(1,:));
    position_no0(2,:) = nonzeros(position(2,:));

    % need to remove 0s before entering into interp1
    position(1,zero_positions) = interp1(nonzero_positions, position_no0(1,:),...
            zero_positions);
    position(2,zero_positions) = interp1(nonzero_positions, position_no0(2,:),...
            zero_positions);

    % Make all position arrays start at the same point in time
    if i == 1 || i == 2
        % Find index of minimum in first 25 frames
        [starting_point, starting_index] = min(position(1,1:25));
    end

    % Third video is rotated -> find the minimum vertical position from
    % other dimension
    if i == 3
        % Find index of minimum in first 25 frames
        [starting_point, starting_index] = min(position(2,1:25));
    end

    % Crop the position data to the correct starting point
    position = position(:, starting_index:length(position));

    % Remove any NaN values
    position = rmmissing(position, 2); % means removing column

    % Save data adjusted to correct start point to a cell
    positions{i} = position;
end

% Find minimum length of the previous position arrays
minimum_length = min(cellfun('size', positions, 2));

% Loop through each camera/position array and make the same length
for i = 1:number_cameras
    positions{i} = positions{i}(:, 1:minimum_length);
end

% Make X (matrix with all (x,y) data for 3 cameras)
% For PCA to work properly, subtract mean from each of the data dimensions
X = zeros(6, minimum_length);
X(1,:) = positions{1}(2,:) - mean(positions{1}(2,:)); % x1
X(2,:) = positions{1}(1,:) - mean(positions{1}(1,:)); % y1
X(3,:) = positions{2}(2,:) - mean(positions{2}(2,:)); % x2
X(4,:) = positions{2}(1,:) - mean(positions{2}(1,:)); % y2
X(5,:) = positions{3}(2,:) - mean(positions{3}(2,:)); % x3
X(6,:) = positions{3}(1,:) - mean(positions{3}(1,:)); % y3
```

```
% Plot horizontal positions
figure()
scatter(1:length(X), X(1,:), 100, '.'); hold on
scatter(1:length(X), X(3,:), 100, '.'); hold on
scatter(1:length(X), X(6,:), 100, '.'); hold on
legend('Camera 1', 'Camera 2', 'Camera 3')
xlabel('Video Frame', 'FontSize', 18)
ylabel('Horizontal Position', 'FontSize', 18)
title(test_title, 'FontSize', 18)
set(gca, 'FontSize', 14)

% Plot vertical positions
figure()
scatter(1:length(X), X(2,:), 100, '.'); hold on
scatter(1:length(X), X(4,:), 100, '.'); hold on
scatter(1:length(X), X(5,:), 100, '.'); hold on
legend('Camera 1', 'Camera 2', 'Camera 3')
xlabel('Video Frame', 'FontSize', 18)
ylabel('Vertical Position', 'FontSize', 18)
title(test_title, 'FontSize', 18)
set(gca, 'FontSize', 14)

% Actual PCA part
[U,S,V] = svd(X, 'econ');

% Analyzing sigmas
sig = diag(S);

% Calculate energy for each rank/sigma
energy1 = sig(1)^2/sum(sig.^2);
energy2 = sum(sig(1:2).^2)/sum(sig.^2);
energy3 = sum(sig(1:3).^2)/sum(sig.^2);
energy4 = sum(sig(1:4).^2)/sum(sig.^2);
energy5 = sum(sig(1:5).^2)/sum(sig.^2);
energy6 = sum(sig(1:6).^2)/sum(sig.^2);
energies = [energy1, energy2, energy3, energy4, energy5, energy6];

% Plot sigmas
figure()
bar(1:6, sig, 'c')
title(test_title, 'FontSize', 18)
xlabel('Sigma Number', 'FontSize', 18)
ylabel('Sigma Value', 'FontSize', 18)
set(gca, 'FontSize', 14)

% Plot energies
figure()
bar(1:6, energies, 'c')
ylim([0 1])
text(1:length(energies),energies,num2str(energies'),'vert','top','horiz',...
    'center', 'FontSize', 12);
title(test_title, 'FontSize', 18)
xlabel('Rank', 'FontSize', 18)
ylabel('Energy', 'FontSize', 18)
```

```
set(gca, 'FontSize', 14)
```