Assignment 3: Principal Component Analysis
AMATH582 Data Analysis

Joseph David

February 24, 2020

Abstract

In this project, we determine the dynamics of an oscillator system by applying Principal Component Analysis on three videos of the oscillator from different angles. We diagonalize the covariance matrix of the video measurements from each camera to find the optimal coordinate system to view the oscillator's dynamics and project our measurements onto this coordinate system. This reduces the redundancy of our measurements and gives the simplest view of the system.

1 Introduction and Overview

In this project, we work with videos `cam1j`, `cam2j`, and `cam3j` for `j=1,2,3,4`. Each of these videos is of someone holding a vertical spring with a can attached to the bottom, and a light on top of the can. For each experiment, we are given three different sets of footage from different angles. The `j`'s correspond to four different sets, which we call experiments of footage.

1. Experiment I. (`j=1`) Ideal case. The videos' quality is good and the oscillation is only in the vertical direction.

2. Experiment II. (`j=2`) Noisy case. Cameras are slightly shaken so video resolution is poor. Oscillation is only in the vertical direction.

3. Experiment III. (`j=3`) Horizontal Displacement. Can has oscillations in the vertical as well as horizontal direction. Videos' quality is good.

4. Experiment IV. (`j=4`) Horizontal Displacement and Rotation. Can oscillates in vertical and horizontal directions and is given rotation as well. Videos' quality is good.

For each experiment, our first goal is to determine the paths of the can in the coordinate systems of each video, or, in other words, each video frame is a matrix of pixels and we must determine the position of the pixel containing the can at each frame of the video. This is the most crucial part of the project, as PCA analysis is only as good as the data we give it. We first summed the RGB components of each video, turning each video into a 3D as opposed to 4D array. Then, we tried four different methods for tracking the can.

1. Method I. Take brightest pixel.

2. Method II. Take pixel with largest change in brightness in space.

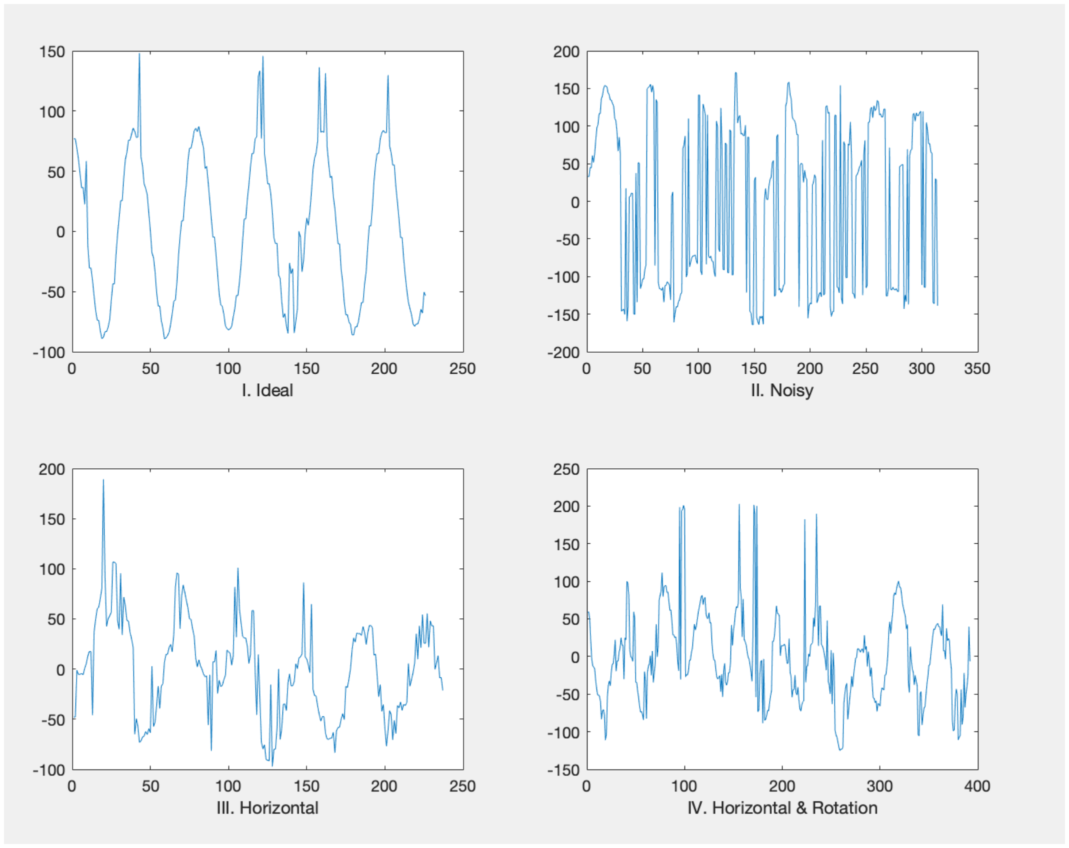

Figure 1: Projected measurements of `cam1j` y-direction using Method I, taking the brightest pixel.

3. Method III. Take pixel with largest change in brightness in time.

4. Method IV. Sum neighboring pixels together and take brightest.

For each experiment and method, we compile the x- and y-coordinates determined by that method for that experiment into rows of the measurement matrix X. Since each experiment contains three videos, each measurement matrix X will have six rows. From there, we compute its covariance matrix, diagonalize, and project our measurements onto a new coordinate system and plot it.

2 Theoretical Background

The theoretical background for this project comes from linear algebra. First, however, recall that if x and y are two vectors, their covariance σ_{xy} is defined by

$$\sigma_{xy} = \frac{1}{n-1} x \cdot y, \tag{1}$$

where $\cdot$ denotes the dot product. Recall that the six rows of the measurement matrix X contain the x- and y-coordinates for some experiment and method. For each X we construct the *covariance matrix* C_X defined by

$$C_X = \frac{1}{n-1} X X^T, \tag{2}$$

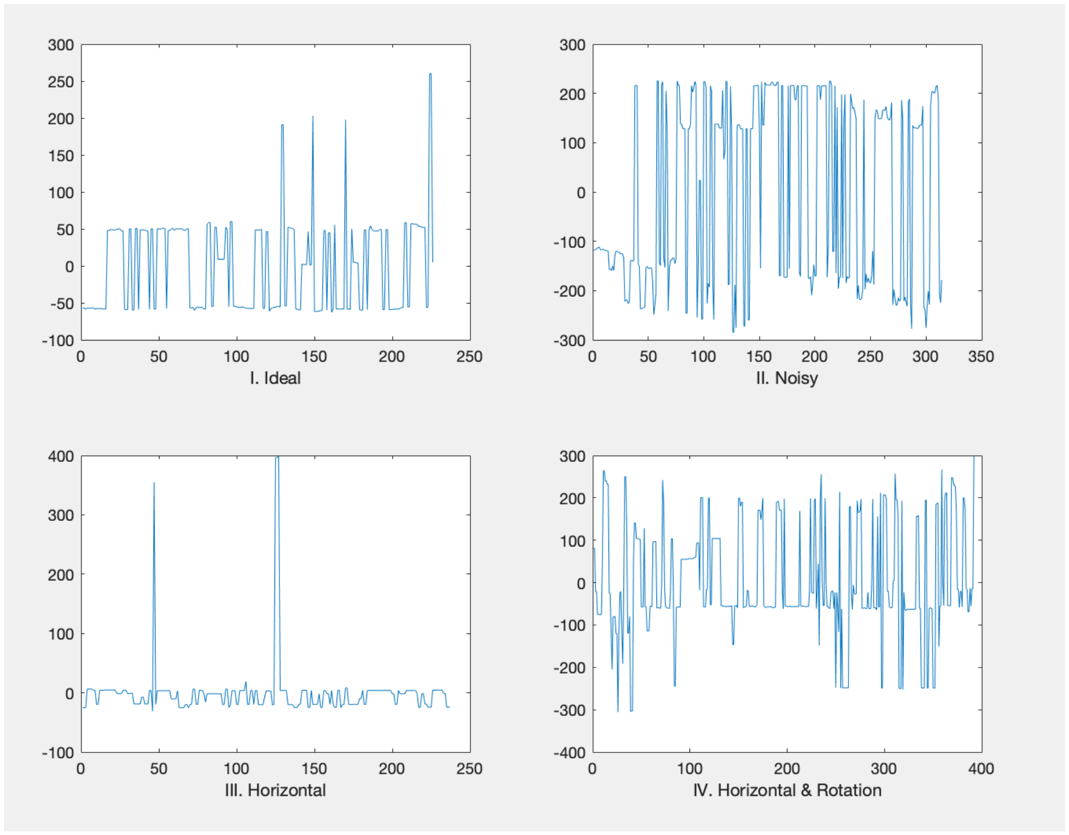

Figure 2: Projected measurements of `cam1j` y-direction using Method II.

where n is the number of frames in the videos. Thus, $(C_x)_{ij} = (1/n-1)X_i \cdot X_j$, which is the covariance between measurements X_i and X_j. Furthermore, we see that C_X is symmetric and so we may use the following theorem from linear algebra: There exist matrices S, Λ such that

$$C_X = S\Lambda S^{-1}, \tag{3}$$

where the columns of S are the eigenvectors of C_X and Λ is a diagonal matrix which contains the corresponding eigenvectors. Thus, the columns of S are a coordinate system in which the covariances between *distinct* measurements (the off-diagonal entries) are zero. In other words, a coordinate system where these measurements are not correlated. Thus, if we view the matrix $Y = S^T X$, which is the projection of our measurements into this new coordinate system, we may see which axes have the most motion.

3 Algorithm Implementation and Development

We use two main algorithms. The first is used to track the can. It takes a video as a parameter (ie a 3D array) and for each frame uses `max` to find the pixel with the largest value, then returns the x- and y-coordinates at each frame. This allows us to create different video arrays from the original and pass it to the function, in order to test different methods of tracking the can. The second algorithm computes the covariance matrix and its diagonalization using `eig`, then plots the measurements in this new coordinate system. It takes in the matrix of measurements X.

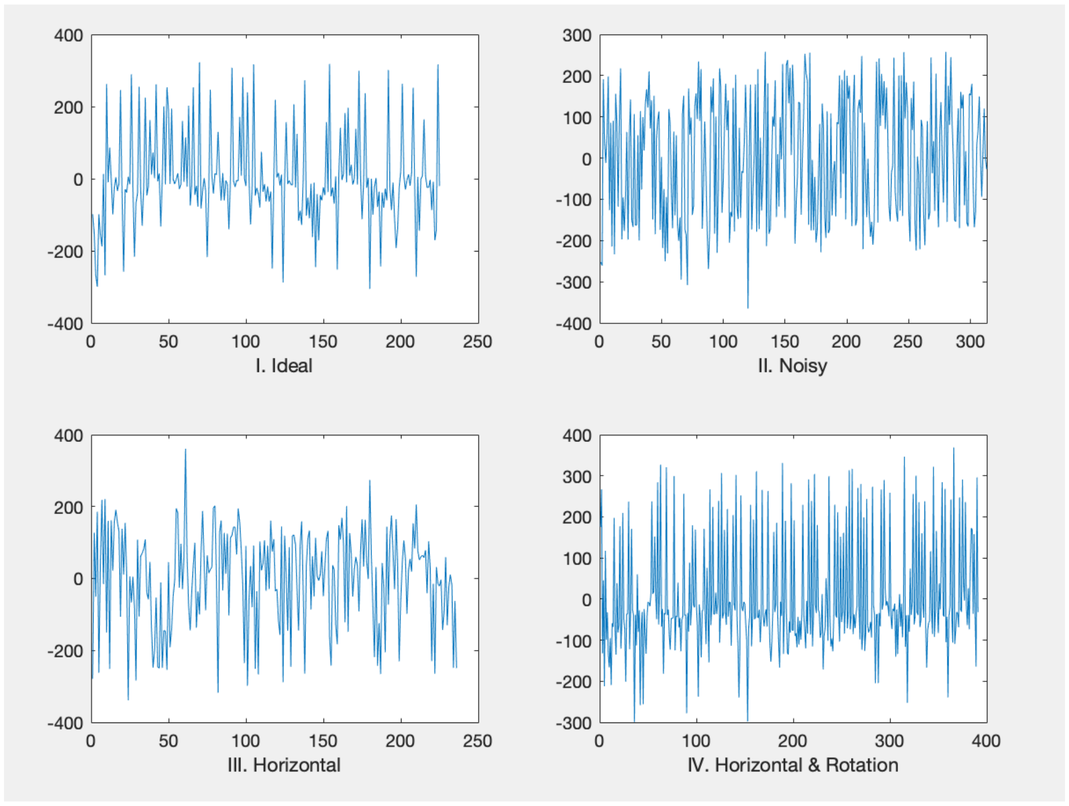

Figure 3: Projected measurements of `cam1j` y-direction using Method III.

4 Computational Results

We used four different methods for tracking the can in each experiment. Using Method I, we see in Figure 1 that for all experiments except the noisy case Experiment II, this method worked well in tracking the can. In the Ideal case Experiment I, the results are near perfect, and in Experiments III and IV we see very good oscillations.

For Method II (see Figure 2), we took the gradient of each frame in space and from there took the pixel with the largest gradient (defined as the sum of the absolute value of the gradients in the x and y directions). The idea was that the light would have much darker pixels around it. We found that this method worked much more poorly. In the ideal case, we see that the path essentially alternates between around -50 and 50,

Algorithm 1: Find x- and y-coordinates of can

```
Create vectors x and y which will contain the coordinates of the can.
Pass in video array.
for j = 1 : number of frames in video do
    Find entry with largest value in jth frame
    Store the indices of that entry in x(j) and y(j).
end for
Return x and y.
```

```
Algorithm 2: Plot measurement projections, comes from Kutz Textbook
    Pass in measurement matrix X.
    Compute covariance matrix Cx using formula (2).
    Diagonalize Cx and use eigenvector matrix S to project X onto.
    Plot projected measurements.
```

which is similar to the results of Method I. However in this case, the dynamics are much less smooth.

For Method III, we took the difference in pixel brightness in time. The idea was similar as in Method II, where the pixel with the light would likely have been much darker in the frame prior and hence would have the largest change in time. We see in Figure 3 that this was the worst method. The results appear to be random.

In Figure 4, we instead plotted the motion of the can in the x-direction as seen from the first camera. In the ideal case, we get some horizontal motion which we should not have seen. In Experiments III and IV where there is supposed to be some horizontal motion we do see some good results for this.

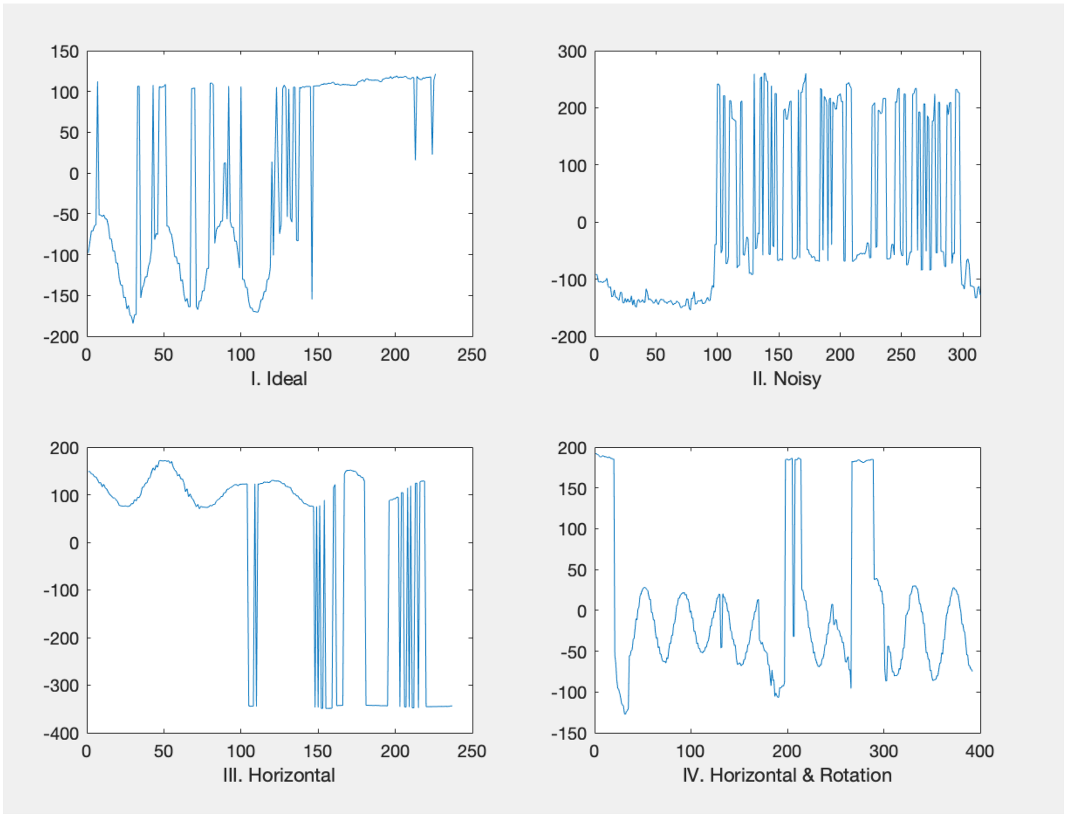

Figure 4: Projected measurements of `cam1j` x-direction using Method IV.

5 Summary and Conclusions

Principal Component Analysis (PCA) provides a very effective way of transforming data in high dimensions into a new coordinate system in which the majority of the information is captured in only a few dimensions.

For this project, we tried to capture the path of the can in multiple ways and applied PCA to each one. We then plotted our projected measurements in order to see if the dynamics of our system could be determined from these measurements. We found that our simplest method, taking the brightest pixel was the best way of tracking the can.

Due to space limitations, we did not analyze the results of our methods on all of the measurements. For instance, we simply plotted the y-direction of one of the cameras, but it is possible that other measurements would have given us more insight into the dynamics. A more in-depth analysis would do this.

Appendix A MATLAB Functions

Add your important MATLAB functions here with a brief implementation explanation. This is how to make an **unordered** list:

- `load camjk.mat` loads video into 4D arrray `vidFramesjk`.
- `[height width rbg numframes] = size(vidFrames)` stores the height, width, and number of frames in `vidFrames` as well as `rgb=3`, because the videos are in color.
- `im2double(A)` takes in an image matrix `A` and converts its entries from type `uint8` to type `double`.
- `gradient(X)` returns the gradient of the array `X` in all directions. For example, if `X` is a matrix then `gradient` returns two matrices that correspond to $\partial/\partial x$ and $\partial/\partial y$.
- `[m,i] = max(X,[],'all','linear')` returns the max `m` over the entire array `X` as well as the linear index `i` of the position where the maximum occurs.
- `ind2sub` converts linear indices to array indices.

Appendix B MATLAB Code

Contains two MATLAB scripts, `hw3functions.m` and `main.m`. The former contains the functions that are used to do the computations. The latter contains the preliminary data ingestion and cleaning.

```
% --------------- function definitions -------------------------------

%method 1: this function sums the RGB axis into one
function x = sumRGB(vidFrames,height,width,num_frames)
    x = zeros(height,width,num_frames);
    for j =1 : num_frames
        x(:,:,j) = sum(vidFrames(:,:,:,j),3);
    end
end

%method 2: this function returns the gradient
function x = grad_space(vidFrames, height, width, num_frames)
    x = zeros(height, width, num_frames);
    for j = 1 : num_frames
        [vFx, vFy] = gradient(vidFrames(:,:,j));
        x(:,:,j) = abs(vFx)+abs(vFy);
    end
end

%method 3: change in time
function x = grad_time(vidFrames, height, width, num_frames)
    x = zeros(height, width, num_frames-1);
    for j = 2 : num_frames
        x(:,:,j-1) = vidFrames(:,:,j) - vidFrames(:,:,j-1);
    end
end

%method 4: sum neighbors
function x = sum_neighbors(vidFrames, height, width, num_frames)
    x = zeros(height-1,width-1,num_frames);
    for j = 1 : num_frames
        for i = 1 : height-1
            for k = 1 : width-1
                x(i,k,j) = vidFrames(i,k,j) + vidFrames(i+1,k,j) + vidFrames(i,k+1,j);
            end
        end
    end
end

%find position of brightest pixel
function [x,y] = findlight(vidFrames,height,width,num_frames)
    x = zeros(1,num_frames); y = zeros(1,num_frames);
    for j = 1 : num_frames
        [~, i] = max(vidFrames(:,:,j),[],'all','linear');
        [y(j),x(j)] = ind2sub([height width], i);
    end
end
```

Listing 1: Function definitions

```
%perform diagonalization and projection
function Y = PCA(x1,y1,x2,y2,x3,y3,k,r)
    X = [x1; y1; x2; y2; x3; y3];
    [~,n] = size(X);
    mn = mean(X,2);
    X = X - repmat(mn,1,n);

    Cx = (1/(n-1))*X*X';
    [V,D] = eig(Cx);
    lambda = diag(D);

    [~, m_arrange] = sort(-1*lambda);
    lambda = lambda(m_arrange);
    V=V(:,m_arrange);

    Y=V'*X;

    figure(k)
    subplot(2,2,r), plot(Y(2,:))

   if r == 1
       xlabel('I. Ideal')
   elseif r == 2
       xlabel('II. Noisy')
   elseif r == 3
       xlabel('III. Horizontal')
   else
       xlabel('IV. Horizontal & Rotation')
   end

end
```

Listing 2: Function definitions.

```
%load all video data as 4D arrays vidFramesi_j
load cam1_1.mat;
load cam1_2.mat;
load cam1_3.mat;
load cam1_4.mat;
load cam2_1.mat;
load cam2_2.mat;
load cam2_3.mat;
load cam2_4.mat;
load cam3_1.mat;
load cam3_2.mat;
load cam3_3.mat;
load cam3_4.mat;
%
%this code shortens videos in each experiment to length of shortest
%video
[height width rgb num_frames1_1] = size(vidFrames1_1);
[height width rgb num_frames2_1] = size(vidFrames2_1);
[height width rgb num_frames3_1] = size(vidFrames3_1);
num_frames1 = min([num_frames1_1 num_frames2_1 num_frames3_1]);
vidFrames1_1 = im2double(vidFrames1_1(:,:,:,1:num_frames1));
vidFrames2_1 = im2double(vidFrames2_1(:,:,:,1:num_frames1));
vidFrames3_1 = im2double(vidFrames3_1(:,:,:,1:num_frames1));

[height width rgb num_frames1_2] = size(vidFrames1_2);
[height width rgb num_frames2_2] = size(vidFrames2_2);
[height width rgb num_frames3_2] = size(vidFrames3_2);
num_frames2 = min([num_frames1_2 num_frames2_2 num_frames3_2]);
vidFrames1_2 = im2double(vidFrames1_2(:,:,:,1:num_frames2));
vidFrames2_2 = im2double(vidFrames2_2(:,:,:,1:num_frames2));
vidFrames3_2 = im2double(vidFrames3_2(:,:,:,1:num_frames2));

[height width rgb num_frames1_3] = size(vidFrames1_3);
[height width rgb num_frames2_3] = size(vidFrames2_3);
[height width rgb num_frames3_3] = size(vidFrames3_3);
num_frames3 = min([num_frames1_3 num_frames2_3 num_frames3_3]);
vidFrames1_3 = im2double(vidFrames1_3(:,:,:,1:num_frames3));
vidFrames2_3 = im2double(vidFrames2_3(:,:,:,1:num_frames3));
vidFrames3_3 = im2double(vidFrames3_3(:,:,:,1:num_frames3));

[height width rgb num_frames1_4] = size(vidFrames1_4);
[height width rgb num_frames2_4] = size(vidFrames2_4);
[height width rgb num_frames3_4] = size(vidFrames3_4);
num_frames4 = min([num_frames1_4 num_frames2_4 num_frames3_4]);
vidFrames1_4 = im2double(vidFrames1_4(:,:,:,1:num_frames4));
vidFrames2_4 = im2double(vidFrames2_4(:,:,:,1:num_frames4));
vidFrames3_4 = im2double(vidFrames3_4(:,:,:,1:num_frames4));
```

Listing 3: Data ingestion and cleaning

```
%sum RGB axes into one
vidFrames1_1 = sumRGB(vidFrames1_1,height,width,num_frames1);
vidFrames2_1 = sumRGB(vidFrames2_1,height,width,num_frames1);
vidFrames3_1 = sumRGB(vidFrames3_1,height,width,num_frames1);

vidFrames1_2 = sumRGB(vidFrames1_2,height,width,num_frames2);
vidFrames2_2 = sumRGB(vidFrames2_2,height,width,num_frames2);
vidFrames3_2 = sumRGB(vidFrames3_2,height,width,num_frames2);

vidFrames1_3 = sumRGB(vidFrames1_3,height,width,num_frames3);
vidFrames2_3 = sumRGB(vidFrames2_3,height,width,num_frames3);
vidFrames3_3 = sumRGB(vidFrames3_3,height,width,num_frames3);

vidFrames1_4 = sumRGB(vidFrames1_4,height,width,num_frames4);
vidFrames2_4 = sumRGB(vidFrames2_4,height,width,num_frames4);
vidFrames3_4 = sumRGB(vidFrames3_4,height,width,num_frames4);
```

Listing 4: Data ingestion and cleaning.