PCA and Spring-Mass Systems

AMATH582: Homework 3
Maple Tan

February 26, 2019

1 Abstract

The objective is to explore Principal Component Analysis (PCA) in a spring-mass experiment. We will obeserve how PCA can reduce redundancy in a system and also possible shortcomings.

2 Introduction and Overview

To illustrate how we may use PCA, suppose we have an ideal spring-mass system modeled by a flashlight on a paint can oscillating only in the z direction. We then set up three different cameras to record the oscillation from three different angles. Though logically we know that the paint can only oscillates in one dimension, our recordings will show the oscillations having two dimensions. Using PCA, we will show that there is only one dimension that our ideal spring oscillates in, even though we recorded six different dimensions. To perform PCA, we will store our measurements in a vector X that contains the six different measurements: two for each of the cameras A, B, and C.

$$X = \begin{bmatrix} \vec{x_A} \\ \vec{y_A} \\ \vec{x_B} \\ \vec{y_B} \\ \vec{x_C} \\ \vec{y_C} \end{bmatrix}$$

In addition to the ideal case, we will also look at different scenarios of oscillation. All the scenarios are listed below:

1. **Ideal Case**: A small displacement of the mass in the z direction. In this case, the entire motion is in the z directions with simple harmonic motion being observed.

2. **Noisy Case**: Repeat the ideal case experiment, but this time, introduce camera shake into the video recording.

3. **Horizontal Displacement**: In this case, the mass is released off-center so as to produce motion in the xy plane as well as the z direction. Essentially, it is a pendulum motion combined with a simple harmonic motion.

4. **Horizontal Displacement and Rotation**: Similar to case three but we add in a rotation motion as well.

3 Theoretical Background

Before we begin PCA, an important assumption is that the data must be mean-centered. So before we start, we must subtract the mean of each measurement from their respective measurement. Next, since PCA basically finds us the best coordinate system to remove redundancies, we will look at the covariance matrix of X:

$$\mathbf{C}_X = \frac{1}{n-1} XX^T = \begin{bmatrix} \sigma^2_{X_A X_A} & \sigma^2_{X_A Y_A} & \cdots \\ \sigma^2_{Y_A X_A} & \sigma^2_{Y_A Y_A} & \cdots \\ \vdots & & \end{bmatrix}$$

In a covariance matrix, the diagonal of the matrix is the variance of the individual measurements. We see that large variances correspond with dynamics of interests while small variances correspond with dynamics of little interest. While the off-diagonals are the covariances between different measurements. In this case, a large covariance represent correlated measurements and implies a degree of redundancy while in a similar vein, a small covariance implies that the measurements are statistically independent. To find a coordinate system that reduces redundancy, we must set the covariance between different measurements to 0, which we can do through diagonalization.

There are two ways to diagonalize the covariance matrix: one through the eigenvalues and eigenvectors and the other through the SVD decomposition. Starting with the eigenvalue and eigenvector method, we note that since XX^T is a square symmetric matrix, this means that there exists an eigendecomposition as follows:

$$XX^T = S\Lambda S^{-1} = S\Lambda S^T$$

Where S is a a matrix of the eigenvectors of XX^T arranged in a column and Λ is a diagonal matrix whose entries correspond with the distinct eigenvalues of XX^T. Note that since S is orthogonal, we can also write $S^{-1} = S^T$. This suggests that instead of working directly with our matrix X, we consider working with the transformed variables instead or in the principal component basis:

$$Y = S^T X$$

Then in this new basis, we can see that the covariance is:

$$\begin{aligned} C_Y &= \frac{1}{n-1} YY^T \\ &= \frac{1}{n-1} (S^T X)(S^T X)^T \\ &= \frac{1}{n-1} S^T X X^T S \\ &= \frac{1}{n-1} S^T (S \Lambda S^T) S \\ &= \frac{1}{n-1} \Lambda \end{aligned}$$

Thus the covariance matrix for Y is a diagonal matrix, meaning that its off diagonals equal 0, and so the covariance has been minimized.

We see something similar in the SVD method, first noting that every matrix has a SVD decomposition $X = U\Sigma V$ where U, V are unitary and Σ is a diagonal matrix. The SVD diagonalizes the matrix by working in the appropriate bases U and V. So in a similar vein as the previous method, we can define our transformed variable as follows:

$$Y = U^* X$$

Then calculating our covariance matrix, we get:

$$\begin{aligned} C_Y &= \frac{1}{n-1} YY^T \\ &= \frac{1}{n-1} (U^* X)(U^* X) T \\ &= \frac{1}{n-1} U^* X X^T U \\ &= \frac{1}{n-1} U^* (U \Sigma V^*)(U \Sigma V^*) T U \\ &= \frac{1}{n-1} \Sigma^2 \end{aligned}$$

Again, we can see that we have diagonalized our covariance matrix so that there are 0s in the nondiagonals. As stated in the notes, we should primarily be using the SVD decompostion as it is more robust.

4 Algorithm Implementation and Development

The main bulk of the code is dedicated to extracting the data from our video. The initial data is a 1×1 struct object that contains a 4D uint8 matrix. We use the `struct2cell` method to convert the struct into a cell object and then the `cell2mat` and `double` methods to convert that into a usable 4D matrix. The dimensions of this matrix contains three different 3D matrices containing a number from 0-255, representing values of red, green and blue at the pixel at that location. Each of these 3D matrices then contains information for a 480×640 size video with n frames, where n depends on the video length. We can also use `imshow` and for loop to preview the image anytime, remembering to convert the matrix back to uint8.

Now that we converted the data into a workable form, we next have to loop through each separate frame to track the movement of the our oscillatory object. To help with this, we create a filter that blacks out the areas of the image that is irrelevant to the oscillation. Using this filter, we element-wise multiply each of the three color frames by a 480×640 matrix that equals 1 where we want to keep the image pixels and 0s everywhere else to denote black. After applying the filter, we will choose a target color depending on the camera: the red-ish flash light, the white light of the flashlight or the light blue light of the flashlight. Each of these colors have a specific RGB value and we can calculate the distance between our target color and the color of each pixel in our frame using a distance formula and choose the pixel location closest to our target color. Since these colors are on the lighter side of the spectrum, the black that we have used for our filter, is never close enough to our target to be chosen. We then store the x and y location of our 'closest pixel' for our data.

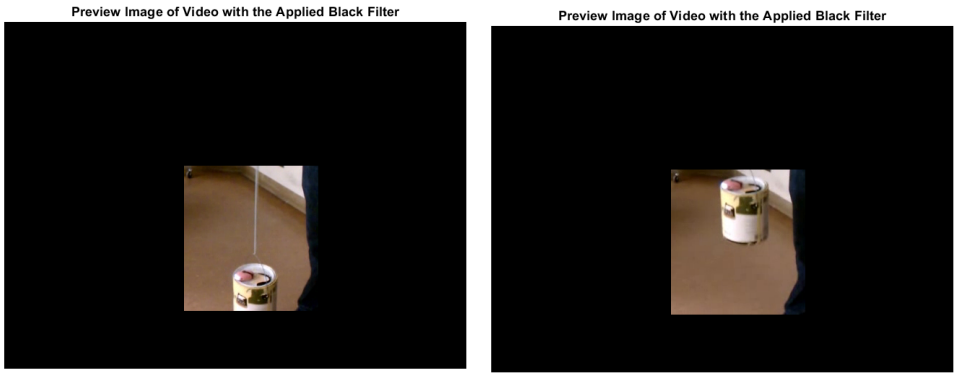

Figure 1: An example of a filter on the video. Note that the pinkish-red flash light that we are tracking in this case is clearly visible at both the top and bottom of the oscillation

After collecting our raw data, we then see the following patterns from each camera:

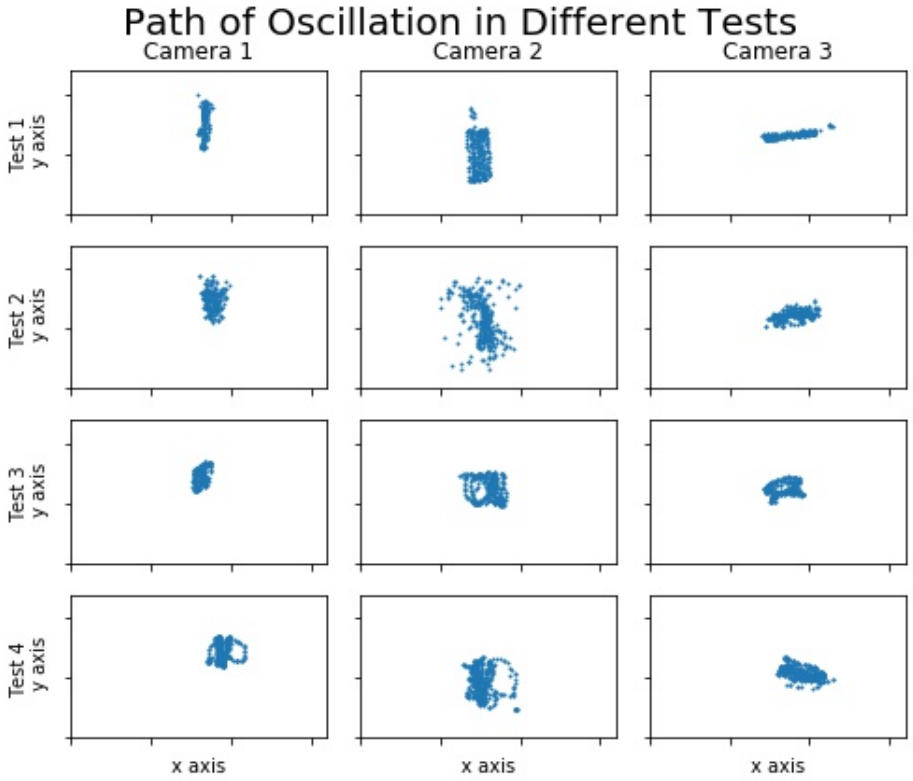

Figure 2: Visual of how the paint can oscillates in accordance to each camera. One interesting to note is that we do see circles in Test 3 and Test 4, indicating two dimensions rather than only one. Created using Python instead of MATLAB.

Before we can begin PCA, we also have to match the oscillation phase between measurements. This is due to the videos not starting the oscillations at the same time. To do this, we will define the starting position of the paint can as the position furthest away from the hand. We can find this desired frame by finding the frame which contains the paint can at the starting position, which can be seen as the min y in the camera 1 and 2 and the max x position during the first handful of frames and initialize the data at those frames. We use the `min` or `max` function in this case. Once properly aligned, we then use the SVD to find the principal components of our data during each of the tests. We do this by decomposing our data into a matrix of 6 different measurements X into an U, Σ, and V. Then multiplying U^*X, we get our data projected into our new coordinate system and which we will use to plot our principal components.

5 Summary and Conclusion

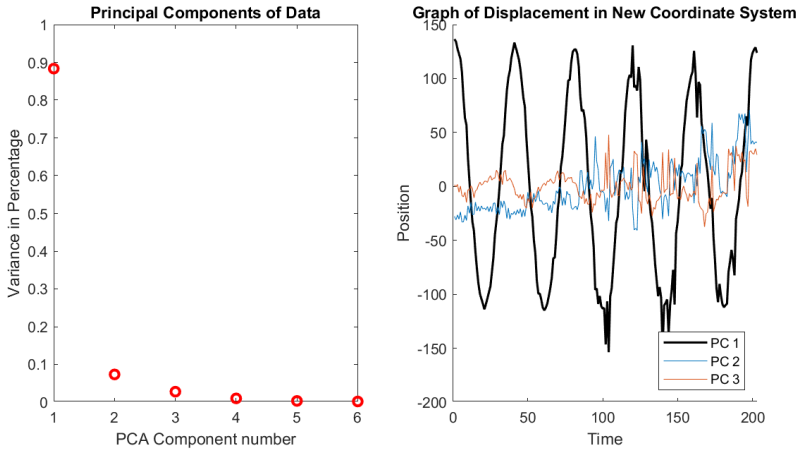

Figure 3: The right is a graph showing the amount of variation in each PCA component and the left is the graph of the first principal component. Notice how close to 90% of the variation is in the first principal component and we see that the second and third principal component mostly just contains numerical error round off. Through the first principal component, we can see that the object does oscillate only in one direction.

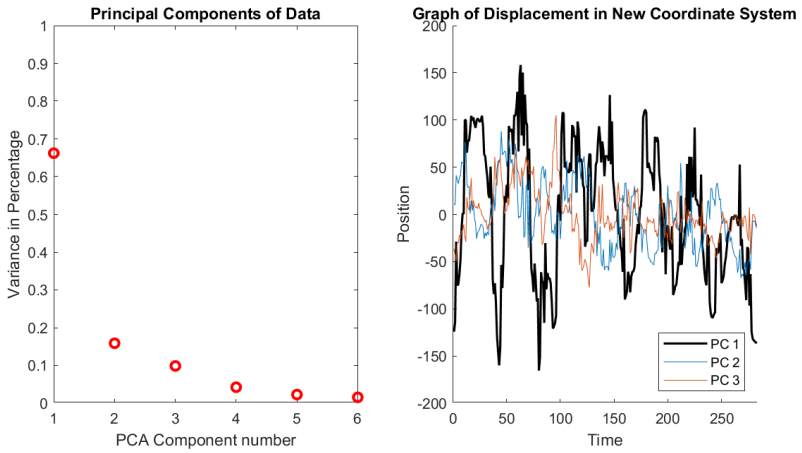

Figure 4: We see here that adding noise through camera shaking does lower the variance in the first principal component but overall as seen in the right graph, we can still see the primary oscillation from the first principal component, even if it is noticeably more noisy.

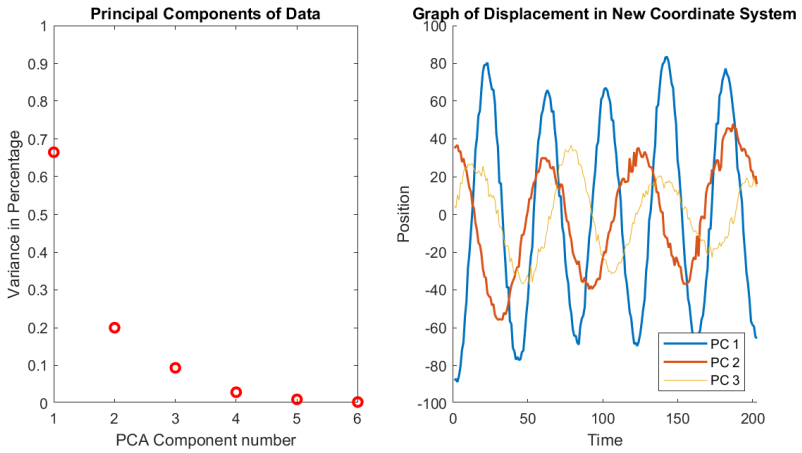

Figure 5: Here, though the variation in the first principal component is similar to test 2, we see a little more variation in the second principal component, implying another dimension in the movement, which would match the releasing of the mass off-center.

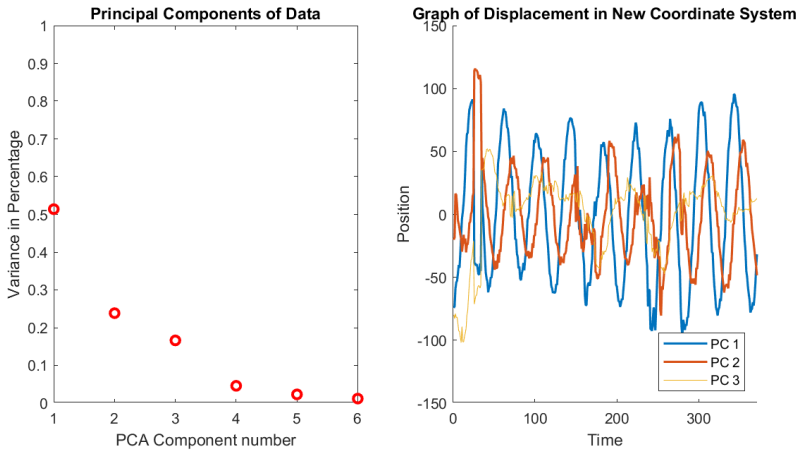

Figure 6: We can see that adding the rotation greatly increased the variation in the second principal component. As such, we see that when we graph the first and second principal components, we can see that

A MATLAB CODE: Data Extraction

```
clear; close all; clc;

% Load Data
file_name = 'cam1_1.mat';
camm = load(file_name);

% Convert the video data into a matrix of numeric values
% Note that cell2mat will return a 4D uint matrix where the
%   there is a 480 x 640 pixel video that has 226 (in the
%   cam1_1.mat case) frames and then 3 other matrices that
%   represents the RGB value of the picture
cam = cell2mat(struct2cell(camm));
cam_dim = size(cam);
imsize = cam_dim(1:2);
imx = imsize(2);
imy = imsize(1);
time = cam_dim(4);

% % Play first k frames
% k = 50;
% for n = 1:k
%     frame = cam(:,:,:,n);
%     imshow(uint8(frame))
%     hold on
%     pause(0.1)
% end

%%
% Initialize Filter Bounds

% Camera 1 Filter Bounds
fxmin = 250;
fxmax = 435;
fymin = 200;
fymax = 400;

% % Camera 2-1, 2-2 Filter Bounds
% fxmin = 200;
```

```
% fxmax = 400;
% fymin = 50;
% fymax = 400;

% % Camera 2-3 Filter Bounds
% fxmin = 200;
% fxmax = 400;
% fymin = 150;
% fymax = 400;

% % Camera 2-4 Filter Bounds
% fxmin = 200;
% fxmax = 390;
% fymin = 95;
% fymax = 350;

% % Camera 3 Filter Bounds
% fxmin = 250;
% fxmax = imy;
% fymin = 150;
% fymax = 350;

% Create filter
filter = zeros(imy, imx);
filter(fymin:fymax, fxmin:fxmax) = ones((fymax-fymax)+1, (fymax-fymax)+1);
minyy = zeros(1,time);      % Store object's y location
minxx = zeros(1,time);      % Store object's x location
threshold = 30;

% Note the target choice for each different camera is as follows:
%   Camera 1: The Reddish-Hue of the Flashlight in all cases
%   Camera 2: 2-1 White, 2-2 Blue-White, 2-3 and 2-4 Reddish Hue
%   Camera 3: 3-1 3-2 3-3 White
for n = 1:time
    % Change frame values into doubles
    frame = double((cam(:,:,:,n)));

    % Show Filter on Video as necessary
    frame(:,:,1) = frame(:,:,1).*filter;
    frame(:,:,2) = frame(:,:,2).*filter;
```

```
    frame(:,:,3) = frame(:,:,3).*filter;
    imshow(uint8(frame));
    pause(0.1)

    % Choose Target
    target = [223 154 151];       % Reddish Hue of Flashlight
    % target = [255 255 255];        % White
    % target = [243 254 254];        % Blue-White emitted from flashlight

    % Find Location in Filtered Frame that's the closest
    %   to our Target color
    dist = colordiff(frame,filter, target);
%     dist2 = colordiff(r,g,b,[255 255 255]);
%     if min(dist(:)) > threshold
%         dist = dist2;
%     end
    [min_val, min_ind] = min(dist(:));
    [miny, minx] = ind2sub(imsize, min_ind);

    % Record x and y location of 'closest pixel'
    minyy(n) = miny;
    minxx(n) = minx;
end

% Save x and y locations into a .mat file
% save('camera1_1.mat', 'minxx','minyy')
%%

% The following code will check path of recorded
%   against the original video
for n = 1:time
    frame = cam(:,:,:,n);
    framebw = rgb2gray(frame);
    imshow(uint8(frame))
    hold on
    plot(minxx(n), minyy(n), 'r.', 'markersize',20)
    pause(0.1)
end

% Plots the path of the oscillating object.
```

```
figure(2)
plot(minxx, imy-minyy, 'r.', 'markersize', 20)
xlim([0 imx])
ylim([0 imy])


% Receives a frame and target color and then calculates
%   the difference between the target color and each pixel
%   of the frame and returns a matrix of differences
function dist = colordiff(frame, filter, target)
    r = frame(:,:,1).*filter;
    g = frame(:,:,2).*filter;
    b = frame(:,:,3).*filter;
    dist = sqrt((r-target(1)).^2 + (g-target(2)).^2 + (b-target(3)).^2);
end
```

B MATLAB AND PYTHON CODE: PCA Analysis and Figure Creation

B.1 MATLAB Code:

```
clear; clc; close all;

% Load Data
test1 = load_dat('camera1_1.mat','camera2_1.mat','camera3_1.mat');
test2 = load_dat('camera1_2.mat','camera2_2.mat','camera3_2.mat');
test3 = load_dat('camera1_3.mat','camera2_3.mat','camera3_3.mat');
test4 = load_dat('camera1_4.mat','camera2_4.mat','camera3_4.mat');


%%
X = test3;
[~,n] = size(X);

mn = mean(X, 2);
X=X-repmat(mn,1,n);
[u,s,~]=svd(X/sqrt(n-1),0); % perform the SVD
lambda=diag(s).^2; % produce diagonal variances
```

```
figure(1)
clf;
% Plot Variance in Principal Components
subplot(1,2,1)
plot(lambda/sum(lambda), 'ro', 'linewidth', 2)
title('Principal Components of Data')
xlabel('PCA Component number')
ylabel('Variance in Percentage')
ylim([0 1])

% Plot Displacement in New Coordinate System
subplot(1,2,2)
project = u'*X;
hold on
plot(1:length(X), project(1:2,:), 'linewidth', 1.5)
plot(1:length(X), project(3,:), 'linewidth', 0.5)
title('Graph of Displacement in New Coordinate System')
xlabel('Time')
ylabel('Position')
xlim([0, length(X)])
legend('PC 1','PC 2', 'PC 3', 'location', 'southeast')
print(gcf, '-dpng', 'test3_PCA.png')


% This Function will take in the three different file names for each
%   of the tests and will create a 6 row matrix that contains the data
%   for each camera. It will also align the phases of the data.
function test = load_dat(mat1, mat2, mat3)
    cam1 = cell2mat(struct2cell(load(mat1, 'min*')));
    cam2 = cell2mat(struct2cell(load(mat2, 'min*')));
    cam3 = cell2mat(struct2cell(load(mat3, 'min*')));

    [~, ind1] = min(cam1(2,1:40));
    [~, ind2] = min(cam2(2,1:40));
    [~, ind3] = max(cam3(1,1:40));

    cam1_al = cam1(:,ind1:length(cam1));
    cam2_al = cam2(:, ind2:length(cam2));
    cam3_al = cam3(:, ind3:length(cam3));
```

```
    len = min([length(cam1_al),length(cam2_al),length(cam3_al)]);

    test = [cam1_al(:, 1:len); cam2_al(:,1:len); cam3_al(:,1:len)];
end
```

B.2 Python Code:

```
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt


cameras = np.arange(1, 4)
rectype = np.arange(1, 5)

file_names = []

for t1 in rectype:
    for t2 in cameras:
        fname = 'camera'+ str(t2)
        fname = fname + '_' + str(t1) + '.mat';
        file_names.append(fname)

test1 = []
test2 = []
test3 = []
test4 = []
for c in np.arange(0,3):
    x = loadmat(file_names[c])
    print(file_names[c])
    test1.append(x['minxx'][0])
    test1.append(x['minyy'][0])
    x = loadmat(file_names[3+c])
    print(file_names[3+c])
    test2.append(x['minxx'][0])
    test2.append(x['minyy'][0])
    x = loadmat(file_names[6+c])
    print(file_names[6+c])
    test3.append(x['minxx'][0])
    test3.append(x['minyy'][0])
```

```
    x = loadmat(file_names[9+c])
    print(file_names[9+c])
    test4.append(x['minxx'][0])
    test4.append(x['minyy'][0])

fig, axs = plt.subplots(4,3, figsize=(7,6), constrained_layout=True)
axs[0][0].set_title('Camera 1')
axs[0][1].set_title('Camera 2')
axs[0][2].set_title('Camera 3')
axs[0][0].set_ylabel('Test 1 \n y axis')
axs[1][0].set_ylabel('Test 2 \n y axis')
axs[2][0].set_ylabel('Test 3 \n y axis')
axs[3][0].set_ylabel('Test 4 \n y axis')
axs[3][0].set_xlabel('x axis')
axs[3][1].set_xlabel('x axis')
axs[3][2].set_xlabel('x axis')
r = 0
fig.suptitle('Path of Oscillation in Different Tests', fontsize = 20)
for c in np.arange(0,3):
    axs[r][c].scatter(test1[0+(2*c)],test1[1+(2*c)], s=1)
    axs[r][c].set_xlim([0, 640])
    axs[r][c].set_ylim([0, 480])
    axs[r][c].set_xticklabels([])
    axs[r][c].set_yticklabels([])

    axs[r+1][c].scatter(test2[0+(2*c)],test2[1+(2*c)], s=1)
    axs[r+1][c].set_xlim([0, 640])
    axs[r+1][c].set_ylim([0, 480])
    axs[r+1][c].set_xticklabels([])
    axs[r+1][c].set_yticklabels([])

    axs[r+2][c].scatter(test3[0+(2*c)],test3[1+(2*c)], s=1)
    axs[r+2][c].set_xlim([0, 640])
    axs[r+2][c].set_ylim([0, 480])
    axs[r+2][c].set_xticklabels([])
    axs[r+2][c].set_yticklabels([])

    axs[r+3][c].scatter(test4[0+(2*c)],test4[1+(2*c)], s=1)
    axs[r+3][c].set_xlim([0, 640])
    axs[r+3][c].set_ylim([0, 480])
```

```
    axs[r+3][c].set_xticklabels([])
    axs[r+3][c].set_yticklabels([])


fig.tight_layout()
fig.subplots_adjust(top=0.9)
```