ENPM673 Project 1

Brendan Neal

February 22, 2023

Contents

1 Introduction

This project has students use computer vision functions found in the OpenCV Library in order to identify a ball and characterize the ball's path in pixels. Additionally, this project has students use various estimation techniques learned in class in order to characterize noisy LIDAR point cloud data and compare which technique performs the best characterization.

2 Problem 1.1

2.1 Process

For Problem 1.1, I was tasked with detecting and plotting the ball's center coordinates for each frame in the video. To do this, I first loaded the video into my workspace as an RGB video. Next, I created two arrays, a min threshold and a max threshold. I then set the weights of the min and max thresholds to only capture red pixels. Each array was a 1 by 3 array including the blue channel, green channel, and red channel for the minimums and maximums. Next, I used the cv2.inRange() function to threshold out all of the pixels that did not fall between the minimum and maximum channels. This produced a binary thresholded video. After that, I located the indices where the binary thresholded frames were not equal to zero and saved them to an array. These indices are where the ball is found in each frame. Next, I took the mean of these x and y indices to find the center of the ball and append them into two new arrays. Finally, I plotted these x and y coordinates onto a scatter plot.

2.2 Problems Encountered

The only problem that occurred during this step was the fact that in an image, the coordinate frame is flipped (x is left to right, y is top to bottom from the top leftmost pixel in the image frame) so I had to negate all the y values before plotting to properly mimic the path of the ball in the video.

The other problem was that my thresholding was not perfect, and I could still detect parts of the hand in some of my frames, which threw off my center calculation for the ball. In order to solve this, I cropped the hand out of the image, and made the proper adjustments to account for this cropping when performing computations later.

2.3 Results

Pictured below in figure 1 is the plot of the balls path.

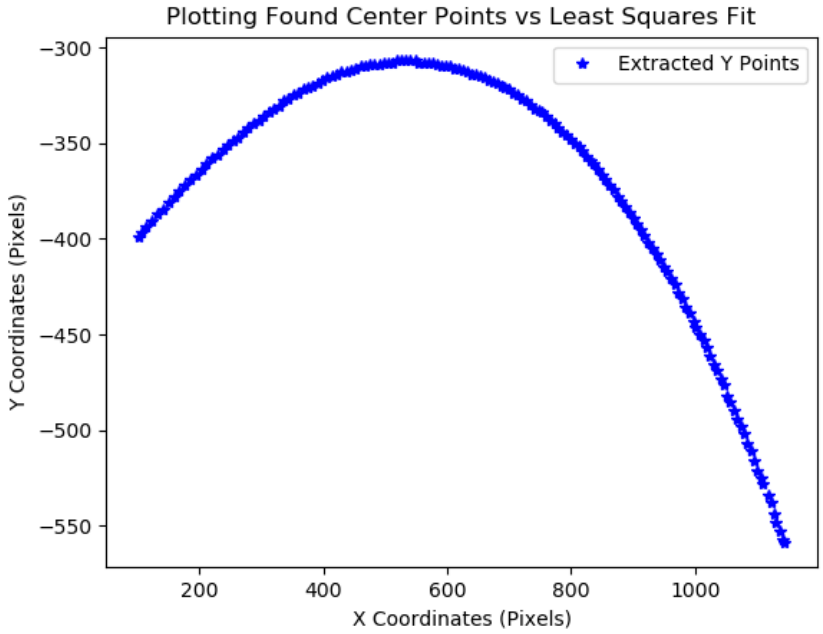

Figure 1: Extracted Center Points of Ball

3 Problem 1.2

Problem 1.2 had me use standard least squares to fit a curve to the extracted coordinates given that the ball travels in a parabolic path.

3.1 1.2a Process

In order to print the equation of the curve, I had to build my own standard least squares function. My function takes two inputs, the x and y data, and outputs the least squares coefficients as well as the estimated y values. First, I created my A and Y matrices. The A matrix takes the form: $\begin{bmatrix} x_n^2 & x_n & 1 \end{bmatrix}$. Then, using that A matrix, I performed the following operations to solve for the least squares coefficients:

$$B = (A^T A)^{-1} \cdot (A^T) Y$$

Then I used this next equation to solve for the least squares estimated Y values:

$$Y_{est} = A \cdot B$$

After completing this, I then indexed the constants outputted from my function and printed the equation to the terminal.

3.2 1.2b Process

Using the outputs from the function mentioned before, I plotted the new estimated Y values with the same detected x values on top of the plot from before.

3.3 Problems Encountered

I am new to Python, so getting the indexing of matrices took some time, however I got it to work.

3.4 Results

The equation outputted by my Least Squares Function is:

$$y = -0.000622x^2 + 0.643667x - 470.0155$$

The least squares fit plotted over my original detection of the ball is shown in figure 2:

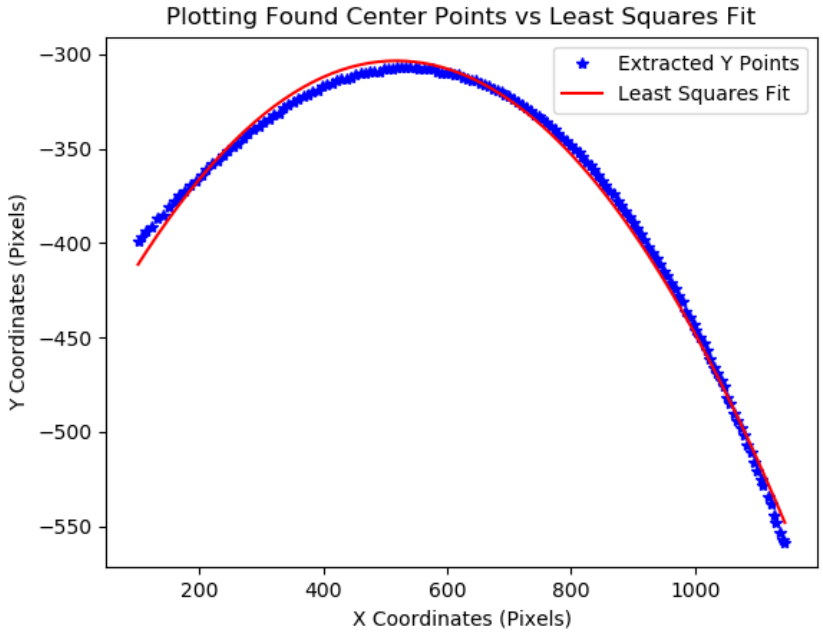

Figure 2: Least Squares Fit

4 Problem 1.3

Given the origin is the top left of the frame, I had to compute the x coordinate of the ball's landing spot knowing that the y coordinate of the ball's landing spot is 300 pixels greater than the original detected location.

4.1 Process

Because I now have my parabolic equation as well as the ball's final landing spot, I used the numpy.roots() function to calculate the final landing position. One root was negative, so I ignored that one and printed the result to the screen.

4.2 Problems Encountered

None in this section.

4.3 Results

I computed the final x position of the ball to be: x = 1315.26 pixels.

5 Problem 2.1

Problem 2.1 had me compute the covariance matrix of some noisy LIDAR point cloud data and compute the surface normal vector assuming the ground plane is flat.

5.1 2.1a Process

To compute the covariance matrix, I created a function that would take in the x, y, and z data from the point cloud file and output the covariance matrix. Within the function, I first calculated the mean of the data. Next, I calculated the variance of the x, y, and z data. Below is the equation for the x data, but the same equation is applied for y and z:

$$S_x = \frac{1}{n} * \sum_i (x_i - \bar{x}) * (x_i - \bar{x})^T$$

Next I calculated the covariance between X/Y, X/Z, and Y/Z. The formula below is for the covariance between X and Y:

$$COV_{XY} = \frac{1}{n} * \sum i(x_i - \bar{x}) * (y_i - \bar{y})$$

Finally, I used Sx, Sy, Sz, COVXY, COVXZ, and COVYZ and formed the covariance matrix in the following form:

$$\begin{bmatrix} S_X & COV_{XY} & COV_{XZ} \\ COV_{XY} & S_Y & COV_{YZ} \\ COV_{XZ} & COV_{YZ} & S_Z \end{bmatrix}$$

5.2 2.1b Process

In order to calculate the surface normal vector for the plane, I created another function that would take the covariance matrix as an input then output the surface normal vector. The function finds the eigenvalues and eigenvectors of the covariance matrix, then finds the index of the minimum eigenvalue. Next, it finds the eigenvector associated with the minimum eigenvalue. Finally, it normalizes this eigenvector by dividing itself by the norm of itself. Then, the function outputs the surface normal vector.

5.3 Problems Encountered

I had to extend the covariance matrix equation from the lecture notes into three dimensions.

5.4 Results

I found the covariance matrix for PC1 to be:

$$\begin{bmatrix} 33.75 & -0.850 & -11.375 \\ -0.850 & 34.963 & -23.06 \\ -11.375 & -23.06 & 20.50 \end{bmatrix}$$

I found the surface normal vector for PC1 to be: $\begin{bmatrix} 0.2862 & 0.5396 & 0.7918 \end{bmatrix}$ and it's magnitude is 1.00000.

6 Problem 2.2

Problem 2.2 had me implement Standard Least Squares, Total Least Squares, and RANSAC in order to fit a plane to two different point clouds of LIDAR data.

6.1 2.2a Process

6.1.1 Standard Least Squares

For the standard least squares fit, I created a function that would take the x, y, and z data from the point cloud data and output the coefficients for the plane as well as the new z fit data. I first created my X matrix based on a rearranged version for the equation of the plane. Note that the coefficients a, b, and d are in reality (a/c), (b/c) and (d/c) but the calculations and the fit

remains the same since this form is a rearrangement of the actual equation of a plane: ax+by+cz+d = paragraph

$$z = -ax - by - d$$

Thus my X matrix looked like:

$$\begin{bmatrix} x_n & y_n & 1 \end{bmatrix}$$

My Y matrix was comprised of my z values from the data set. I then put these matrices through the following computations to get my standard least squares fit:

$$B = (X^T X)^{-1} \cdot (X^T) Y$$
$$Z_{est} = A \cdot B$$

Then, I built a function to generate the plotting surface for my standard least squares fit. This function took the least squares constants generated by the previous function and the estimated Z data as inputs and outputted 3 arrays used for plotting the data later.

6.1.2 Total Least Squares

For the total least square fit, I created a function that would take the x, y, and z data from the point cloud data and output the surface coefficients a,b,c and d. First, I calculated the mean of each of the data points and formed my U matrix. My U matrix looked like:

$$\begin{bmatrix} (x_i - \bar{x}) & (y_i - \bar{y}) & (z_i - \bar{z}) \end{bmatrix}$$

Next, I found my beta matrix by performing the following calculations:

$$beta = (U^T U)^T \cdot (U^T U)$$

Using the beta matrix, I then performed eigenvalue decomposition in order to output my coefficients a, b, and c. I used the following equation to get the d coefficient:

$$d = a * \bar{x} + b * \bar{y} + c * \bar{z}$$

Once I had a, b, c, and d, I created another function that would generate the surface using the coefficients as inputs and outputs the new X, Y, and Z data for the plane. The equation for the Z data came from the following equation:

$$Z = \frac{-1}{c} * (ax + by - d)$$

The outputs from this function are used in plotting the surface later.

6.2 2.2b Process

This problem had me develop a RANSAC function to fit another best-fit model to the data. In order to accomplish this, I made two functions, a RANSAC-Fit and Calculate-RANSAC function. The RANSAC-Fit function takes in 4 inputs, a RANSAC array (The x and y data from the point cloud), the z data from the point cloud, the sample size, and the threshold. The function outputs the best model found via RANSAC operations. First, I initialize all of my parameters. I initialize my max iteration as 0, the current iteration as 0, the max inliers as 0, an empty best model, the probability of an outlier as 0, and the desired probability as 0.95. Next, I begin my RANSAC iterative operations by initializing a ransac-data array with the first two inputs (x, y, and z data). While the current iteration is less than the max iteration, I shuffle the data and pull out a random sample based on the set sample size. This random sample has the sampled x and y data in one array and the sampled z data in another. Then, I form the iteration model by using the standard least squares function I made from before. Next, I calculate the number of inliers with the following equation:

$$Inliers = RANSACArray \cdot IterationModel$$

After calculating the number of inliers, I then calculate the error using the following equation:

$$Error = abs(zdata - inliers^T)$$

Next, I have to update the parameters set above in order to keep searching for the best model fit. To do this, I used a series of logic operations. If the current iteration inlier count is greater than the current number of max inliers, set the max inliers variable to the current iteration inlier count and also set the best model variable to the current iteration model. Next, to update the maximum number of iterations, I calculate the probability of an outlier and use the following equation to update the number of max iterations:

$$Max_Iterations = \frac{log(1 - DesiredProbability)}{log(1 - (1 - OutlierProbabilty)^(SampleSize)}$$

Finally, we increase the iteration count and the function runs until the best fit is found (i.e. where the iteration number is greater than the max iterations).

Now, for the application of this function, I want to compute the outputs to use for plotting. This is where my Calculate-RANSAC function is used. The Calculate-RANSAC function takes in 3 arguments: the x, y, and z data from the point cloud. Within this function, I define my threshold for the RANSAC-fit function. I defined this threshold as the standard deviation of the z data divided by three, or

$$\frac{STD(Z)}{3}$$

Dividing the standard deviation by 5 also works equally well, but I chose 3. I also chose 3 samples per iteration. Then I use my RANSAC-fit function to output the best model, then calculate my outputted Z values by taking the dot product of the x and y data array by the model. Since the model was derived from standard least squares, I can use my build surface function for standard least squares to plot the plane fit in 3D space.

6.3 Problems Encountered

For me, this was the most challenging section of the project. At first, I tried to combine the purposes of both RANSAC functions into one, however I could not get everything to function properly, and splitting them into two allowed everything to work as intended. Additionally, my RANSAC-fit function would occasionally break if the probability of an outlier happened to be 0 during a trial run, so I had to add a line of logic when updating the max iterations that only ran that line if the probability of an outlier was greater than 0. Now, if the case where the probability of an outlier is 0, the max iterations would stay the same and the program would iterate through again.

6.4 Results

6.4.1 Standard Least Squares

Figure 3 depicts the standard least squares plane fit for the PC1 and PC2 data as a red plane.

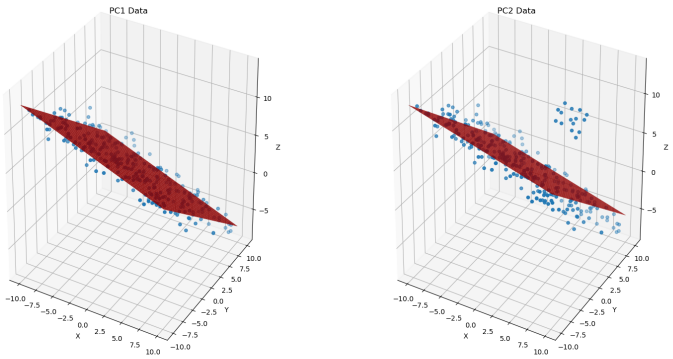

Figure 3: Standard Least Squares Fit

6.4.2 Total Least Squares

Figure 4 depicts the total least squares plane fit for the PC1 and PC2 data as a green plane.

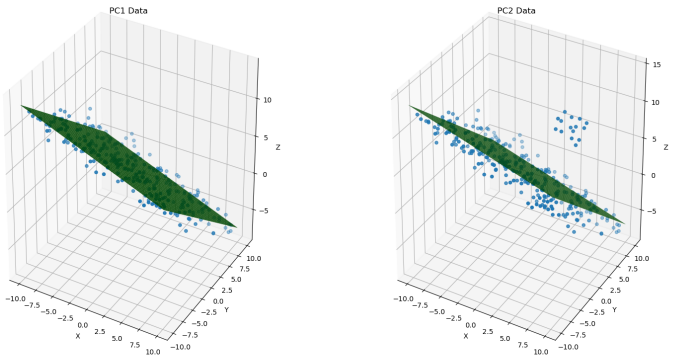

Figure 4: Total Least Squares Fit

6.4.3 RANSAC

Figure 5 depicts the standard least squares plane fit for the PC1 and PC2 data as a yellow plane.

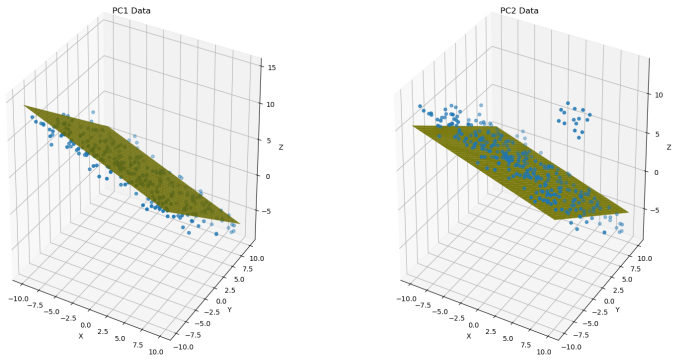

Figure 5: RANSAC Fit

6.4.4 Three Methods Compared

Figure 6 depicts all three fits for the PC1 and PC2 data in the same color scheme outlined above.

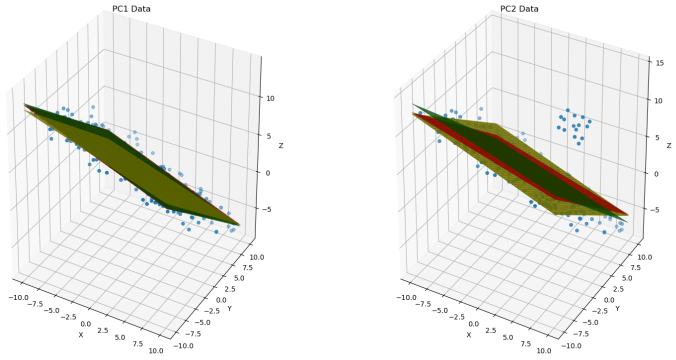

Figure 6: All Three Fits Compared

6.4.5 Analysis and Comparison of Results - SLS and TLS

The total least squares and standard least squares fit for PC1 are almost identical, since there are not nearly as many outliers. However, PC2 shows a

difference between the two fits. Since there exists a cloud of outliers in PC2, the standard least squares fit appears to lose some accuracy in the presence of these outliers. This is because the Total Least Squares takes into account error from both the independent and dependent variables in the surface function, and so it gives less weight to data points we consider outliers.

6.5 Standard Least Squares vs. Total Least Squares vs. RANSAC Comparison

For this particular comparison of the three estimation methods, I got that total least squares fitted the data the best in both the PC1 data and PC2 data. Total Least Squares is more robust in the presence of outliers, and the comparison of the three results in figure 6 shows that the total least squares produces the best fit. There is, however, a random chance that the RANSAC function will produce a more accurate fit, but that depends on the trial. The most consistent case comes from the total least squares fit.