PERCEPTION FOR AUTONOMOUS ROBOTS- ENPM673

PROJECT-1

PROJECT BY:

1. **VYSHNAV ACHUTHAN POONDI VENKATARAMANI (119304815)**

Question -1:

Given, a video containing a person throwing a red ball as a projectile. A series of questions pertains to the video.

Approach 1(a)

The problem statement was to detect and plot the pixel coordinates of the center point of the ball in the video.

Pipeline:

- As the problem required to plot the trajectory of the red ball thrown, I intended to create a mask over the video. Using basic CV2.VideoCapture(), initially the video was opened to analyze it framewise.
- Created two empty lists to store the X and Y coordinates of the centroid of the ball thrown.
- Before the creation of the mask, the initial color convention of the video was BGR, so in order to extract information on one color , convert BGR to HSV frame using cv2.cvtcolor function.
- As the image obtained in the mask was very noisy, a gaussian blur is applied with the size 5,5. The size was particular because the kernel width and height of the kernel should always be 1 when it's divided by 2. I tried with all values and only 5,5 fitted to the resolution of the video.
- To pick a specific HSV range for red color, the basic lower and upper range value of red can be from [0,50,50] to [10,255,255]. [1]. The values were adjusted according to the video mask displayed and found out that at lower range [0,180,100] and High range [9,255,255] was perfect to track red objects without the reflection of light on releasing hand(it also emits red). [refer a video for this : attached in bibliography]
- The cv2.inRange() function is used to create a binary mask of the pixels in the frame that fall within this range of colors. The values in range are higher red, lower red and the HSV frame.
- I used erode and dilate to get an even clearer mask video per frame and reduce disturbances as the frame moves.[2].

- The cv2.bitwise and() method is then called to generate a new picture rc that only displays the pixels in the original frame that relate to the red item. The binary mask mask is used to mask the original frame.
- When I printed the mask, I noticed it printed an array of pixel coordinates per frame, and there it also printed Nan and 0, when the red ball isn't in the frame.
- To remove those, np.count_nonzero is used so that it counts non zero. It runs into the loop when the count >0, and each time, a centroid call was made to take the mean of points column wise and append into X and Y lists created initially.
- This stores nonzero center points of the red ball as X and Y coordinates in the list. Plotted the graph of X and Y in relation to the mask created as max and min range of the graph.

Errors Faced:

- While setting random values for Gaussian blur faced this error "error: (-215:Assertion failed) ksize.width > 0 && ksize.width % 2 == 1 && ksize.height > 0 && ksize.height % 2 == 1 in function 'cv::createGaussianKernels".
- Faced error when I used np.isNan==False as it throwed a boolean error. This call didn't work for me as the video kept running with zero values, and the graph I got was way off.
- Graph didn't display as I coded plt.show() before the Cv2.destoryAllWindows() functions. I used stackoverflow to get an idea of where my error was. Faced this for a long time.

Results:

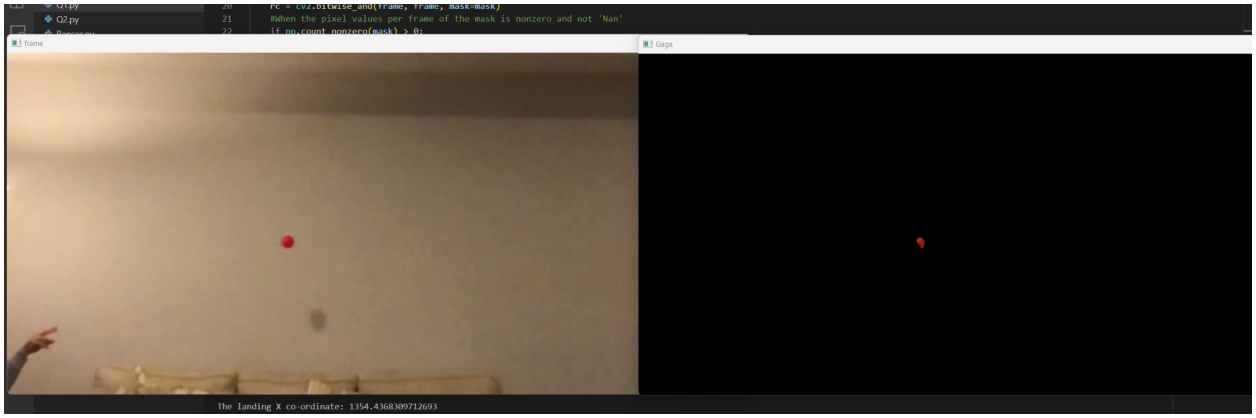


Fig1: Mask applied on the Video

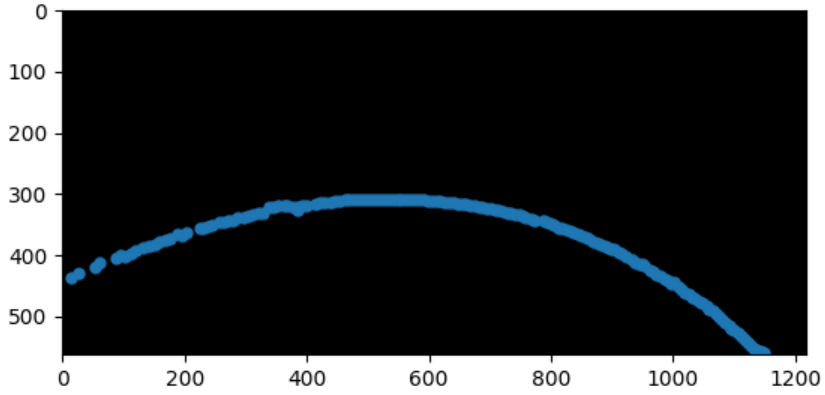


Fig1: Graph of the midpoint of the red ball

Approach 1(b)

The problem statement was to use Standard Least Squares to fit a curve to the extracted coordinates. For the estimated parabola Print the equation of the curve and Plot the data with the best fit curve.

Pipeline:

- The equation of the curve was taken $ax^2 + b * x + c$, which is the standard form of a one degree polynomial equation.
- To solve the equation for y, I took the matrix of solution as matrix of x^2 in the first column, x in the second and finally np.ones for the length of X list.
- As per the least squares method, the least square solution to the equation is given by $X^T X B = X^T Y$, where X is the matrix we calculated in the last step using np.vstack. Now finding B is easy.
- B is the solution of the least squares, which is given by $B = (X^T X)^{-1}.X^T Y$. To find the inverse of the matrix X, we use np.linalg.pinv() which finds the pseudo inverse.
- After finding the inverse of the $(X^T X)$, I proceeded to substitute it in the B formula to find the least squares, which is a matrix of 3 elements.
- To fit a curve to this, we need to find the curve equation using the least square values. As the curve equation is $ax^2 + b * x + c$, we find y using , $Y = (X^2 * lsq[0]) + X * lsq[1] + lsq[2]$. We then plot X and Y we found on top the plot we got in the previous section and print the equation.

Problems Faced:

- Since the problem mentioned parabola, I took the equation as $Y^2 = (X * 4 * a)$, which resulted in the plot going upwards.
- I plotted y,x instead of x,y, which resulted in upward plotting
- Faced error in obtaining the column stack as np.column_stack was a complex function to use in this case.

Results:

```
*****************************************Answer for the 2rd Part*******************************

The Curve equation is : 0.0006062430439800351 x**2 + -0.6184427016704945 x + 462.09793244091753
```

Fig3: Equation of Curve

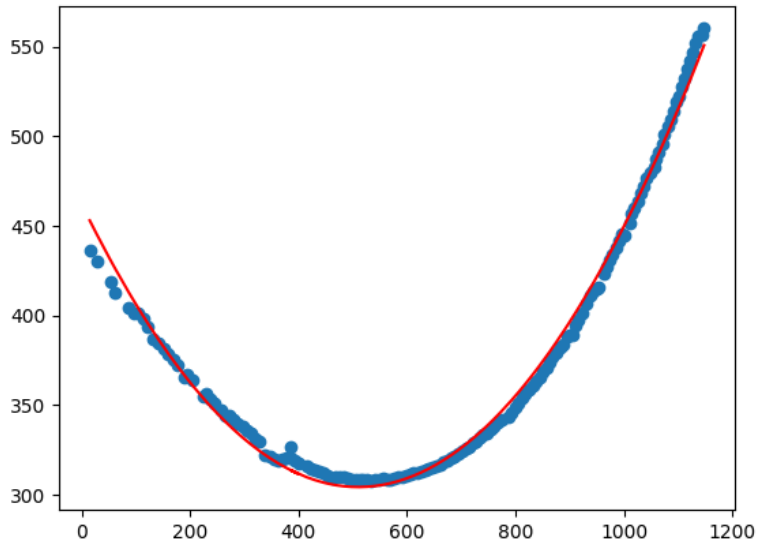


Fig4: Curve Fit(red) over the points obtained in Fig2

Approach 1(b)

The problem statement was to compute the x-coordinate of the ball's landing spot in pixels, if the y-coordinate of the landing spot is defined as 300 pixels greater than its first detected location.

Pipeline:

- In this question, it is mentioned that the coordinates of the first landing point, which is y[0] , is shifted to more than 300 pixels. We know that the equation of the curve is $ax^2 + b * x + c$.
- The equation of the curve we got after the shift in first landing point is , $Y[0]+300 = 0.0006062430439800351x^2 - 0.6184427016704945x + 462.09793244091753$.
- Solving this equation , by using standard square roots will give us the x coordinates. We can use np.roots[], where we can feed the coefficients of each individual unknown in a quadratic equation.
- The end result is an array with two coordinates in negative x and positive x axis. We take the result in the positive x axis.

Errors Faced:

- Bit complexity faced in getting roots of quadratic equation, and hence used np.roots from numpy documentation.

Results:

```
****************************************Answer for the 3rd Part******************************

The landing X co-ordinate roots are: [1354.43683097 -334.31345337]
The landing X co-ordinate: 1354.4368309712693
```

Fig5: Landing Coordinates

Question -2:

Given two csv files, pc1.csv and pc2.csv, which contain noisy LIDAR point cloud data in the form of (x, y, z) coordinates of the ground plane.

Approach 2(a)

The problem statement is given in PC1.Csv, we need to compute the covariance matrix and Assuming that the ground plane is flat, use the covariance matrix to compute the magnitude and direction of the surface normal.

Pipeline:

- Using pandas dataframe,we first read the csv file, using pd.read_csv. Since , we don't have any column headers in our file, we can give headers=None inside the function call. We read both the csv files , as it is useful for the whole problem.
- In theory, the Covariance Matrix is a square matrix that calculates the covariance between two or more variables. The covariance matrix's entries indicate the covariance between two variables, whereas the diagonal elements represent the variance of each variable.
- We can calculate the covariance matrix using the formula

$$S = \frac{1}{n}\sum_{i}(x_i - \bar{x})(x_i - \bar{x})^T$$ where $\bar{x}$ is the mean of the set of points $\{x_i\}$ and S is the variance

- Now to calculate the x and mean of x, in our code, we first take the shape of df1 using x.shape. We also initialize the covariance matrix of column size as

it should have the same dimensions as the number of variables in our csv file which is the number of columns.

- Next, we take the mean of the data by either using np.mean or we can use the conventional method by looping for the number of columns and taking the average of the values , store it as a list.
- Once,we have X and X mean, we can now substitute the values in the formula. For this we use two for loops, to compute over each column and subtract value with its mean. For each value in the x or the first csv file, we then find a dot product of x value minus its average and x value minus its average transpose. Two for loops is to iterate through all columns of Df1.
- The dot product is stored in a variable called covs, where it is appended per each iteration. It is then divided by the number of rows in df1 which is x.shape[0].
- This is stored in a list covariance which can later be accessed in forthcoming operations.
- To calculate the magnitude and the direction of the resulting vector, we just need to calculate the eigenvalue and eigenvector of the covariance matrix.The eigenvectors of a matrix in linear algebra describe the directions in which the matrix affects by simply extending or compressing the vector without affecting its orientation. The appropriate eigenvalue represents the stretching or compressing factor.
- To calculate the eigenvalue and eigenvector, we can use np.linalg.eig() on the covariance matrix obtained.

Problems Faced:

- When I used np.mean to get mean and cross checked with np.cov, there was a big margin of error. Instead opted to use the conventional averaging method to find mean.
- When I used covs=covs/(x0), the resulting Eigenvectors and Eigenvalues were imaginary values, and there was a huge difference between np.cov and the covariance matrix obtained.
- Same error faced, when iterated only once in for loop to calculate covariance.

Results:

```
The Covariance Matrix:
 [[ 33.75161048  -0.83199927 -11.26419876]
 [ -0.83199927  35.22252652 -23.81124371]
 [-11.26419876 -23.81124371  31.54293801]]
```

Fig6: The Covariance Matrix Obtained

```
Eigen Values are [Magnitude]: [ 6.52888902 34.65841253 59.32977347]
Eigen Vectors are [Direction]:
 [[ 0.31860592  0.9052664  -0.28103917]
 [ 0.61077306 -0.42279807 -0.66947596]
 [ 0.72487691 -0.04164786  0.68761829]]
```

Fig7: The EigenValue and Vector Obtained

Approach 2(b)

In this question, you will be required to implement various estimation algorithms such as Standard Least Squares, Total Least Squares and RANSAC. Using pc1.csv and pc2, fit a surface to the data using the standard least square method and the total least square method. Plot the results (the surface) for each method and explain your interpretation of the results

Pipeline:

- As we loaded the csv using pandas dataframes, for our simplicity, assign one column to one variable for both the csvs.
- We can use the method used in question 1 to implement the least squares. The general formula to find the least squares is given as $X^TXB = X^TY$, where we can calculate the solution B using the formula $B = (X^TX)^{-1}.X^TY$.
- The equation of the curve was taken $ax + by + c = z$ as this is the general equation of a line in a cartesian form.Similar to problem 1, we calculated the X vector for both the files.
- To find the inverse of the matrix X, we use np.linalg.pinv() which finds the pseudo inverse.

- After finding the inverse of the $(X^T X)$, I proceeded to substitute it in the B formula to find the least squares, which is a matrix of 3 elements. We do this for both the files simultaneously
- Once we get LSQ1 and LSQ2 for files, we then find the z value, by substituting LSQ values and multiplying with the X column variables and Y column variables.
- We then plot a 3D figure using ax=plt.axes(projection='3d) using X1,Y1,Z1 and fit the curve on it by plotting matplotlib tri_surf function to plot a suitable surface over it. We repeat this process for the second file.

Approach for Total Least Squares

- In theory Total least squares is an upgrade over the OLS method, as OLS only considers the errors along one axis at a time,whereas TLS minimizes the sum of squared residuals in both the x and y variables simultaneously, taking into account the errors in both axes.
- Mathematically, this is expressed as

$$E = \sum_{i=1}^{n} (a(x_i - \bar{x}) + b(y_i - \bar{y}))^2$$

Where, a and b are coefficients of the line drawn.

- In our approach, we already used the difference between value and its mean in finding the covariance matrix. Mathematically, the total least squares problem can be solved by either taking SVD or eigenvalue decomposition of the matrix

$$: \left\| \begin{bmatrix} x_1 - \bar{x} & y_1 - \bar{y} \\ \vdots & \vdots \\ x_n - \bar{x} & y_n - \bar{y} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \right\|^2$$

Where, the above matrix is the covariance matrix representation of a given matrix.

- The approach to find total least squares is quite similar to finding covariance. As we already found covariance for one file, we can use the

same method used in approach 2(a) to find covariance and eigenvectors of the second file.

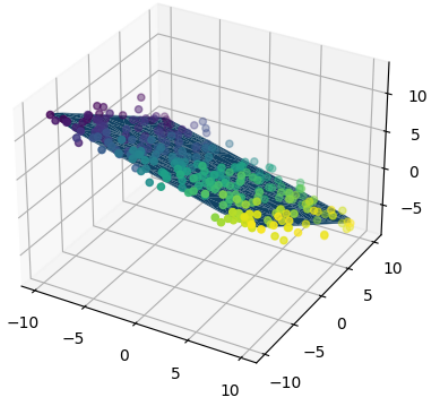
- Now the equation in this case is that we take the cartesian equation of the plane. This is done because the goal of TLS is to reduce the error between the independent and dependent variables at the same time. This signifies that the mistake is in the independent variables as well as the dependent variable .Hence, the TLS equation must account for errors in all variables, and it is written as $ax + by + cz = d$.
- We then use the eigenvectors to solve for this quadratic equation to obtain a plane equation. To find z , we rewrite the equation as $z = (1/c).((-ax + by) + (a * \bar{x} + b * \bar{y} + \bar{z})$, where we substitute a as eigenvector[0][0] , b as eigenvector[1][0] and c as eigenvector[2][0]. We take the first columns as solutions.
- We then plot a 3D figure using ax=plt.axes(projection='3d) using X1,Y1,Z1 and fit the curve on it by plotting matplotlib tri_surf function to plot a suitable surface over it. We repeat this process for the second file.

Problems Faced:

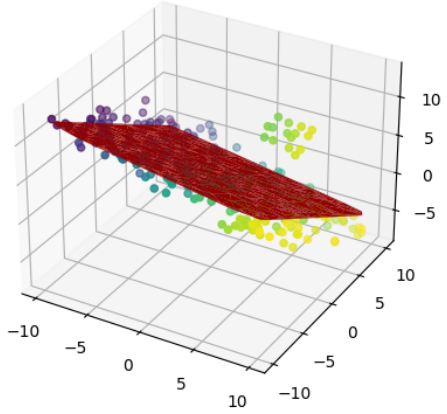
- Calculating covariance again for the second file
- The equation kept throwing a size error as I couldn't use the same equation over two files, hence was forced to create another variable.
- The curve fitting wasn't possible with other column eigen vectors as the curve result was way away from the points.

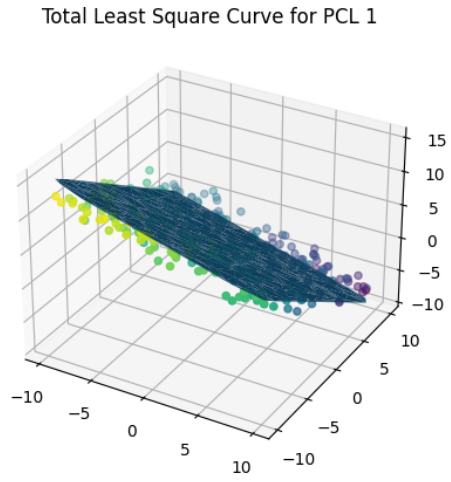
Results:

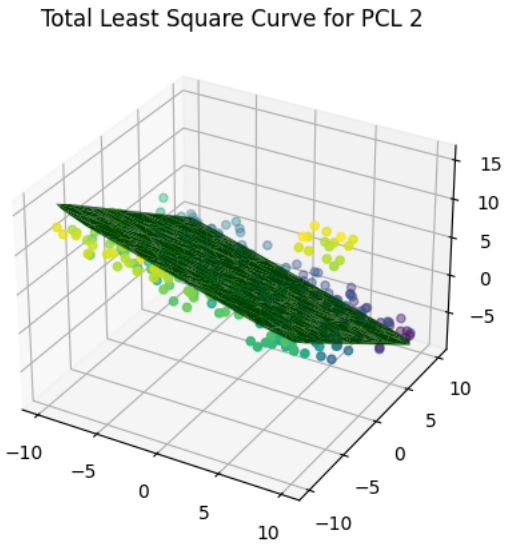
Curve for PC 1



Curve for PC 2







Approach 2(b)(b)

Additionally, fit a surface to the data using RANSAC. You will need to write RANSAC code from scratch. Briefly explain all the steps of your solution, and the parameters used. Plot the output surface on the same graph as the data. Discuss which graph fitting method would be a better choice of outlier rejection.

Pipeline:

- As we loaded the csv using pandas dataframes, for our simplicity, assign one column to one variable for both the csvs.
- The RANSAC method selects a small group of data points at random and fits a model to them. Based on a user-defined threshold, this model is then used to categorize the remaining data points as inliers or outliers. If the number of inliers exceeds a given threshold, the model is deemed valid and re-estimated using all inliers. This process is performed a certain number of times, and the model with the most outliers is chosen as the best estimate.
- For the same purpose in our case we take the following parameters:
    - Threshold - Zero-mean Gaussian noise with std. dev. σ: $t^2=3.84\sigma^2$
    - Number of Samples : $N = \log(1-p)/\log\left(1-(1-e)^s\right)$, where p = 0.95 and e as 0.5. We take s to be as 5 in our case
- Once we initiated the initial parameters for both files, for number of samples,we run a for loop. In this case we use the cartesian equation of a plane given by

$$Plane\ Equation\ in\ Cartesian\ form$$
$$ax + by + cz + d = 0$$

    - , where using this we can calculate the the constants a,b,c using the formula

$$a = [(y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_2)]$$

$$b = [(z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_2)]$$

$$c = [(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_2)]$$

- After finding the constants for both the files, we can now set the number of random samples to be taken at a moment using np.random.choice. We then append all the constants in a coefficient matrix.
- We iterate through each column of a 3D dataset df1, estimating the perpendicular distance from each point (x,y,z) to the plane ax + by + cz + d = 0, where a, b, c, and d are estimated coefficients.

- The distance is calculated by dividing the plane's equation ax + by + cz + d by the magnitude of the plane's normal vector, which is provided by

np.sqrt(a**2 + b**2 + c**2). As a result, the point and plane are separated by a scalar distance dis1.

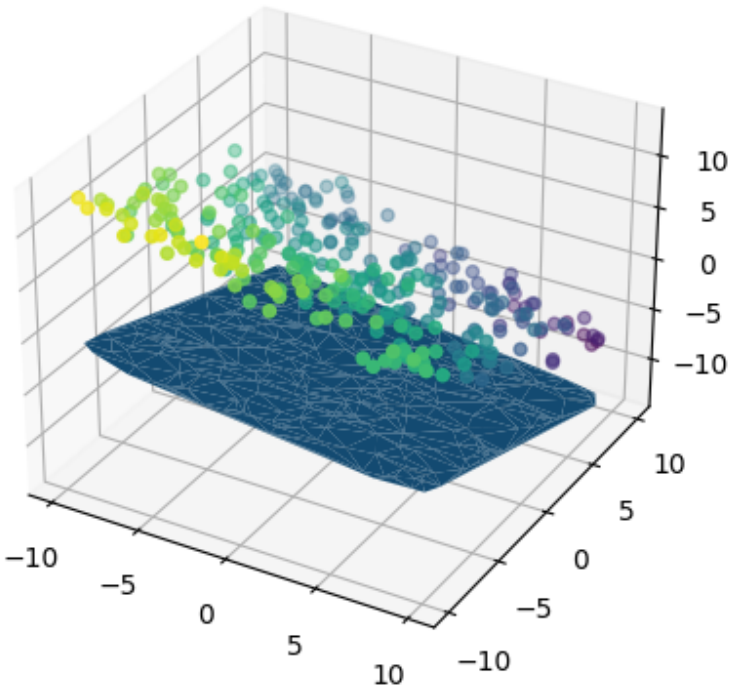
- We then check to see whether the distance dis1 is less than a certain threshold value threshold. If the distance between two points is smaller than the threshold, the point is deemed an inlier, and the inline1 counter is incremented.
- We then take the maximum inlier points and then return the coordinates of the points from the coefficients array.
- We then do the curve fitting for the plane equation $z = (1/c).((-ax + by) + (a * \bar{x} + b * \bar{y} + \bar{z})$, and plot the z points for the particular iterations which is 3. Note , the plots can always vary as per the iteration as the model gives the best fit.

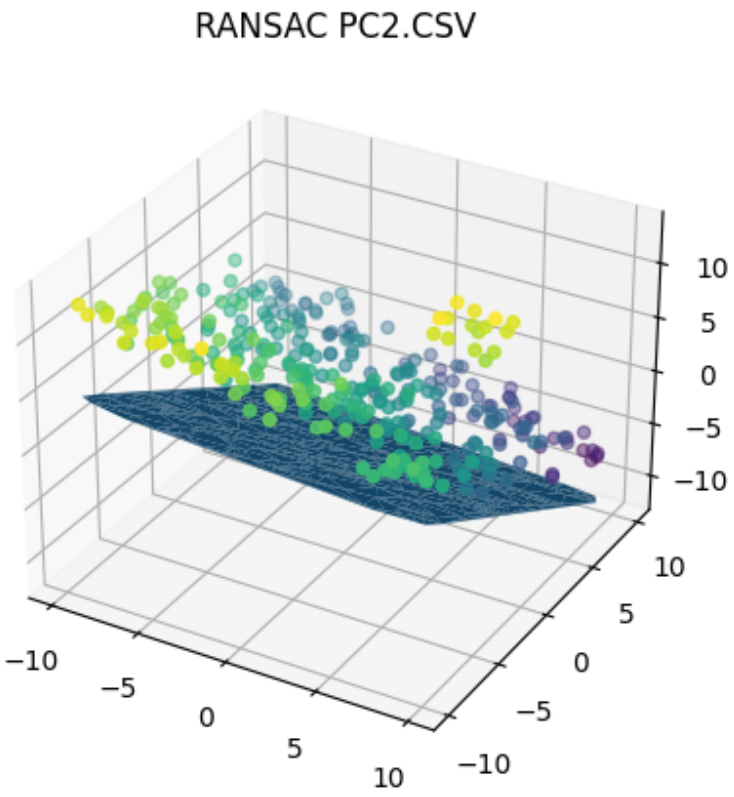
Problems Faced:

- Implementing two separate files in one single code block using functions.
- The method to find coefficients was complex.
- Took the wrong plane equation

RANSAC is a robust fitting approach that can deal with data that has a considerable quantity of outliers. It works by picking a random subset of data points repeatedly, fitting a model to that subset, and then assessing the model's goodness of fit on the remaining points. This process is performed several times, and the model with the most inliers is selected as the best match.

Results:


RANSAC PC1.CSV
10
5
0
−5
−10
10
5
0
−5
−10
−10
−5
0
5
10



Bibliography:

1- https://www.analyzemath.com/line/equation-of-line.html