Project 1

ENPM673

Vishnu Mandala
119452608

Problem 1:

In the given video, a red ball is thrown against a wall. Assuming that the trajectory of the ball follows the equation of a parabola:

1. **Detect and plot the pixel coordinates of the center point of the ball in the video.** ***(Hint: Read the video using OpenCV's inbuilt function. For each frame, filter the red channel)***

2. **Use Standard Least Squares to fit a curve to the extracted coordinates. For the estimated parabola you must,**
   a. **Print the equation of the curve.**
   b. **Plot the data with your best fit curve.**

3. **Assuming that the origin of the video is at the top-left of the frame as shown below, 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.**

Solution:

1. The following code analyzes video of a ball in motion and performs the following tasks:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

cap = cv2.VideoCapture('ball.mov')  # Read the video
```

- **import numpy as np** imports the NumPy library which is used for numerical computations.
- **import matplotlib.pyplot as plt** imports the Matplotlib library which is used for data visualization.
- **cap = cv2.VideoCapture('ball.mov')** initializes a video capture object **cap** that captures frames from the specified video file **'ball.mov'**.

```
#Create lists to store the x-coordinates and y-coordinates of the center of the ball
x = []
y = []
```

- **x** and **y** are empty lists initialized to store the x-coordinates and y-coordinates of the center of the ball for each frame of the video.

```
# Loop over the frames of the video
while cap.isOpened():
    ret, frame = cap.read()  # Read a frame
    if not ret:
        break
```

- **while cap.isOpened():** is a loop that runs until the end of the video.
- **ret, frame = cap.read()** reads a single frame from the video capture object **cap**. It returns a boolean **ret** indicating whether the frame was successfully read and the **frame** itself.

```
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)  # Convert frame to HSV color space
    mask = cv2.inRange(hsv, (0,150,130), (6,255,255))  # Filter the Red Channel
    ycoords, xcoords = np.where(mask > 0)   # Unpacks the non-zero indices obtained through the mask into
xcoords and ycoords
```

- **cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)** converts the frame from the default BGR color space to the HSV color space. This is because the HSV color space separates the color information into three components - Hue, Saturation, and Value - making it easier to isolate a specific color range. In contrast, the RGB or BGR color space represents colors using a combination of red, green, and blue values, which can be challenging to work with when trying to isolate a specific color range.
- **cv2.inRange(hsv, (0,150,130), (6,255,255))** creates a binary mask by thresholding the red color channel in the HSV color space. The resulting mask array has a value of 255 (white) at the pixels that fall within the red color range and 0 (black) at the pixels outside the range.

- **np.where(mask > 0)** returns a tuple of two 1-dimensional arrays that contain the row and column indices of the white pixels (i.e., pixels with a value greater than 0) in the mask array. These indices represent the x-y coordinates of the center of the ball in the frame. The xcoords array contains the column indices (i.e., the x-coordinates) and the ycoords array contains the row indices (i.e., the y-coordinates).

```
    # Find center point of the ball
    if len(ycoords) > 0 and len(xcoords) > 0:
        center  = (int(np.mean(xcoords)), int(np.mean(ycoords)))
        cv2.circle(frame, center, 3, (255, 0, 0), -1)  # Draws a blue, solid circle at the center of the ball
        x.append(center[0])
        y.append(center[1])
    # Display the frame
    cv2.imshow('Frame', frame)
    cv2.waitKey(1)
cap.release()
cv2.destroyAllWindows()
```

- **if len(ycoords) > 0 and len(xcoords) > 0:** checks if there are any non-zero pixels in the binary mask.
- **(int(np.mean(xcoords)), int(np.mean(ycoords)))** calculates the mean of the x-coordinates and y-coordinates of the non-zero pixels to get the center point of the ball.
- **cv2.circle(frame, center, 3, (255, 0, 0), -1)** draws a blue solid circle of radius 3 at the center of the ball in the original frame.
- **x.append(center[0])** and **y.append(center[1])** append the x-coordinate and y-coordinate to the lists
- **cv2.imshow('Frame, frame)** displays the current frame on the screen.
- **cv2.waitKey(1**) function waits for a specified delay (1 millisecond) for a keyboard event.
- **cap.release()** releases the video capture and frees up any associated resources
- **cv2.destroyAllWindows()** closes the window that is created by the program.

Now, we have the lists with x-coordinates and y-coordinates of the center of the ball in each frame.

2. **Standard Least Squares -**
We start by fitting a parabolic curve to the extracted x-y coordinates of the ball. The equation of a parabolic curve is given by:

$$y = ax^2 + bx + c$$

where a, b, and c are coefficients to be determined.

We can find these coefficients using the method of Standard Least Squares. The goal of this method is to find the coefficients that minimize the sum of the squares of the vertical distances between the data points and the curve i.e., we want to minimize the following objective function:

$$f(a,b,c) = \sum (y - (ax^2 + bx + c))^2$$

where the sum is taken over all the (x,y) pairs we have.

The values of a, b, and c that minimize this function will be the coefficients of the parabolic curve that best fits the data. We can find these coefficients using linear algebra. We first write our objective function in matrix form:

$$f(a,b,c) = (y - MX)^T(y - MX)$$

where X is a column vector containing the values of a, b, and c
M is a matrix containing the values of x^2, x, and 1 for each data point
M^T denotes the transpose of the M matrix.

We can expand this expression as follows:

$$f(a,b,c) = y^Ty - 2X^TM^Ty + X^TM^TMX$$

We want to find the values of X that minimize this expression. To do so, we take the derivative of f with respect to X, set it equal to zero, and solve for X. This gives us:

$$X = (M^TM)^{-1}M^Ty$$

where $(M^TM)^{-1}$ denotes the inverse of the matrix M^TM.

```
#Convert lists to numpy arrays
x = np.array(x)
y = np.array(y)

#Fit parabola to data using Standard Least Squares
M = np.array([x**2, x, np.ones(len(x))]).T                    #Matrix with 3 columns containing 'x square'
values, 'x' values and '1's is created and transposed
coeffs = np.linalg.inv(M.T.dot(M)).dot(M.T).dot(y)         #Co-efficients of x^2, x, constant for the equation
y=ax^2+bx+c are obtained
a, b, c = coeffs
print(f"\nEquation of the parabolic trajectory of the ball: y = {a}x^2 + {b}x + {c}")
```

- The code first converts the lists of x and y coordinates into numpy arrays.
- **np.array** creates the matrix M by stacking the vectors x^2, x, and 1s vertically.
- **np.dot** calculates dot product of M^TM and M^Ty.
- **np.linalg.inv** calculates the inverse of M^TM, and dot it with M^Ty to get the values of a, b, and c that minimize the objective function. These values are then assigned to the variables a, b, and c and the equation of the curve is printed

3. The following code computes the x-coordinate of the landing spot:

```
y_first = y[0]
y_land = y_first + 300
```

- **y_first** stores the first y-coordinate in the y array which is the initial point. We add 300 to this value to get an estimate of the y-coordinate where the ball will land.

```
d = b**2 - 4*a*(c - y_land)
```

- **d** represents the discriminant of the quadratic formula that we will use to solve for the x-coordinates where the ball lands. The quadratic formula is given by:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We substitute c - y_land for c to solve for the x-coordinates where the ball lands. The discriminant is used to determine whether there are real solutions to the equation. If d is negative, there are no real solutions

```
if d < 0:
    print("Error: No real solutions")
else:
    x1 = (-b + np.sqrt(d)) / (2*a)
    x2 = (-b - np.sqrt(d)) / (2*a)
    print(f"Possible landing spots: x = {x1}, {x2}")
    # Check if roots are negative
    if x1 > 0 and x2 < 0:
        print(f"The x-coordinate of the ball's landing spot is (As the other point is out of frame):
{x1}\nThe ball lands at ({x1},{y_land})\n")
        l = x1
    elif x1 < 0 and x2 > 0:
        print(f"The x-coordinate of the ball's landing spot is (As the other point is out of frame):
{x2}\nThe ball lands at ({x2},{y_land})\n")
        l = x2
    else:
        print(f"The ball does not land in frame\n")
```

- If there are real solutions, we solve for x1 and x2 using the quadratic formula. These correspond to the possible x-coordinates where the ball lands. If only one value is positive and the other is negative, we choose the positive value as the actual x-coordinate of the landing spot since the ball can't land outside the frame. We print the x-coordinate of the landing spot along with the y-coordinate (y_land) and store it in the variable l. If the ball does not land in frame, we print an appropriate message.

```
# Plot the data with the best fit curve
plt.scatter(x, y)
plt.scatter(l, y_land, marker='o', color='green')
plt.text(l+10, y_land+10, "Landing spot", color='green')
plt.plot(x, a*x**2 + b*x + c, 'r-')
plt.gca().invert_yaxis()
plt.xlabel('X-Axis')
plt.ylabel('Y-Axis')
plt.title('Trajectory of the Ball')
plt.show()
```

- **plt.scatter(x, y)** plots the scattered data points of the trajectory.
- **plt.scatter(l, y_land, marker='o', color='green')** plots the landing spot as a green circle marker on the plot.
- **plt.text(l+10, y_land+10, "Landing spot", color='green')** places a text label "Landing spot" next to the green circle marker at a distance of 10 units from the right and 10 units from the top.
- **plt.plot(x, a*x**2 + b*x + c, 'r-')** plots the best fit curve, which is a parabolic curve given by the equation $y = ax^2 + bx + c$, with the coefficients a, b, and c obtained from the previous step.
- **plt.gca().invert_yaxis**() inverts the y-axis of the plot so that the trajectory appears to be moving upward.
- **plt.xlabel('X-Axis')** and **plt.ylabel('Y-Axis')** add labels to the x-axis and y-axis respectively.
- **plt.title('Trajectory of the Ball')** adds a title to the plot.
- **plt.show()** displays the plot.

Full Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

cap = cv2.VideoCapture('ball.mov')                            # Read the video

'''-----------------Question 1. Finding the coordinates of the Center of the Ball------------------'''
#Create lists to store the x-coordinates and y-coordinates of the center of the ball
x = []
y = []

# Loop over the frames of the video
while cap.isOpened():
    ret, frame = cap.read()                                   # Read a frame
    if not ret:
        break
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)              # Convert frame to HSV color space
    mask = cv2.inRange(hsv, (0,150,130), (6,255,255))         # Filter the Red Channel
    ycoords, xcoords = np.where(mask > 0)                     #Unpacks the non-zero indices obtained through
the mask into xcoords and ycoords

    # Find center point of the ball
    if len(ycoords) > 0 and len(xcoords) > 0:
        center  = (int(np.mean(xcoords)), int(np.mean(ycoords)))
        cv2.circle(frame, center, 3, (255, 0, 0), -1)         #Draws a blue, solid circle at the center of
the ball
        x.append(center[0])
        y.append(center[1])
    # Display the frame
    cv2.imshow('Frame', frame)
    cv2.waitKey(1)

# Release the video capture and close the window
cap.release()
cv2.destroyAllWindows()

'''-----------------Question 2. Using Standard Least Squares Method to fit a curve to the extracted
coordinates------------------'''
#Convert lists to numpy arrays
x = np.array(x)
y = np.array(y)

#Fit parabola to data using Standard Least Squares
M = np.array([x**2, x, np.ones(len(x))]).T                    #Matrix with 3 columns containing 'x square'
values, 'x' values and '1's is created and transposed
coeffs = np.linalg.inv(M.T.dot(M)).dot(M.T).dot(y)            #Co-efficients of x^2, x, constant for the
equation y=ax^2+bx+c are obtained
a, b, c = coeffs
print(f"\nEquation of the parabolic trajectory of the ball: y = {a}x^2 + {b}x + {c}")

'''-----------------Question 3. Finding the x-coordinate of the ball's landing spot------------------'''
# Compute the x-coordinate of the landing spot
y_first = y[0]
y_land = y_first + 300
```

```
d = b**2 - 4*a*(c - y_land)
if d < 0:
    print("Error: No real solutions")
else:
    x1 = (-b + np.sqrt(d)) / (2*a)
    x2 = (-b - np.sqrt(d)) / (2*a)
    print(f"Possible landing spots: x = {x1}, {x2}")
    # Check if roots are negative
    if x1 > 0 and x2 < 0:
        print(f"The x-coordinate of the ball's landing spot is (As the other point is out of frame):
{x1}\nThe ball lands at ({x1},{y_land})\n")
        l = x1
    elif x1 < 0 and x2 > 0:
        print(f"The x-coordinate of the ball's landing spot is (As the other point is out of frame):
{x2}\nThe ball lands at ({x2},{y_land})\n")
        l = x2
    else:
        print(f"The ball does not land in frame\n")

# Plot the data with the best fit curve
plt.scatter(x, y)
plt.scatter(l, y_land, marker='o', color='green')
plt.text(l+10, y_land+10, "Landing spot", color='green')
plt.plot(x, a*x**2 + b*x + c, 'r-')
plt.gca().invert_yaxis()
plt.xlabel('X-Axis')
plt.ylabel('Y-Axis')
plt.title('Trajectory of the Ball')
plt.show()
```

Terminal Output:

```
PS C:\Users\manda\OneDrive - University of Maryland\Perception For Autonomous Robots\Project 1> &
"C:/Program Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of Maryland/Perception For
Autonomous Robots/Project 1/problem1.py"

Equation of the parabolic trajectory of the ball: y = 0.0005897003813826 97x^2 + -0.5977806132188792x +
455.14988106710655
Possible landing spots: x = 1372.9428397397155, -359.2405731458969
The x-coordinate of the ball's landing spot is (As the other point is out of frame): 1372.9428397397155
The ball lands at (1372.9428397397155,746)
```

Plot:

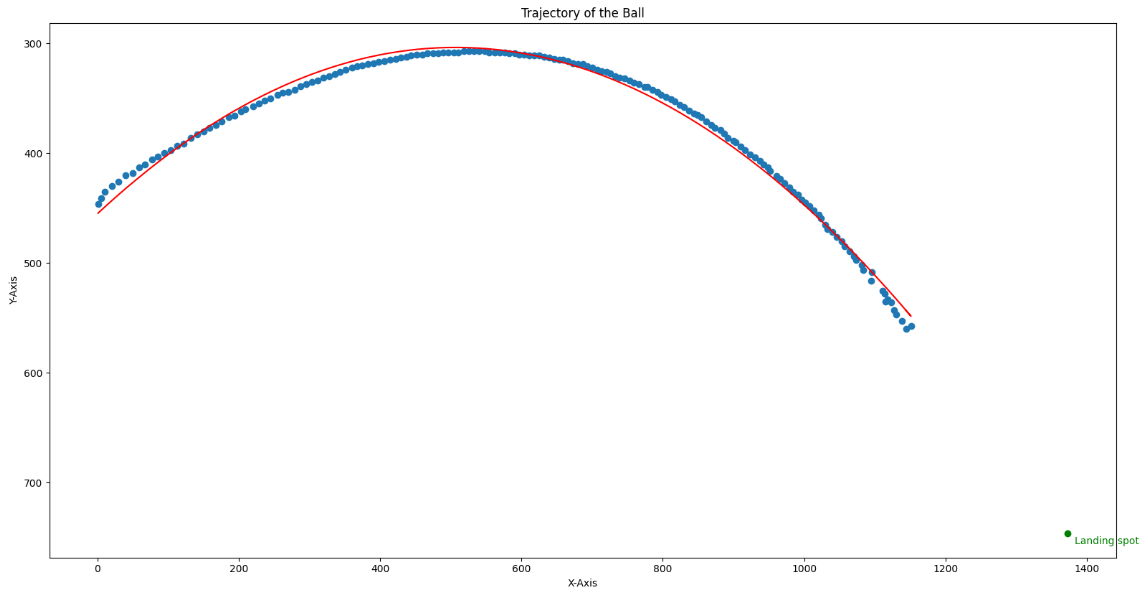

Figure 1. Trajectory of the Center of the Ball with the data points in Blue, Fitted Parabola in Red and Landing point in Green

Pipeline:

- **Import necessary libraries:** The code imports the OpenCV, NumPy, and Matplotlib libraries for image processing and plotting.
- **Read the video file:** The code reads a video file named "ball.mov" using the OpenCV VideoCapture() method.
- **Find the center of the ball:** The code uses a loop to iterate over the frames of the video. For each frame, it converts the image to HSV color space and applies a color filter to extract the red channel. It then finds the center of the ball in the frame by calculating the mean of the x and y coordinates of the non-zero indices obtained through the mask. The x and y coordinates of the center of the ball are appended to the lists 'x' and 'y'.
- **Fit a curve to the extracted coordinates:** The code uses the Standard Least Squares method to fit a parabolic curve to the x and y coordinates of the center of the ball. The x coordinates are squared and used to create a matrix M with 3 columns containing 'x^2' values, 'x' values, and '1's which is then transposed. The coefficients of x^2, x, constant for the equation $y=ax^2+bx+c$ are obtained using the least squares method.
- **Find the x-coordinate of the landing spot:** The code uses the equation of the parabolic curve to find the x-coordinate of the landing spot. It calculates the y-coordinate of the landing spot by adding a fixed value (300) to the y-coordinate of the first frame. It then finds the roots of the quadratic equation for the parabolic curve and uses them to calculate the possible landing spots. If one of the roots is negative, it is discarded, and the other root is taken as the x-coordinate of the landing spot. If both roots are negative, the ball is assumed to have landed out of frame.
- **Plot the data with the best fit curve:** The code plots the x and y coordinates of the center of the ball on a scatter plot. It also plots the best fit parabolic curve and the landing spot on the same plot. The plot is then displayed using the Matplotlib library.

Problem 2:

Given are 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.

1. Using pc1.csv:

a. Compute the covariance matrix.

b. Assuming that the ground plane is flat, use the covariance matrix to compute the magnitude and direction of the surface normal.

2. In this question, you will be required to implement various estimation algorithms such as Standard Least Squares, Total Least Squares and RANSAC.

a. 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.

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.

Solution:

```
import numpy as np
import matplotlib.pyplot as plt

# Load the data from the CSV files
pc1 = np.loadtxt('pc1.csv', delimiter=',')
pc2 = np.loadtxt('pc2.csv', delimiter=',')

data = np.concatenate([pc1, pc2])                                  # Combine the two dataframes into one
```

- **import numpy as np**: Imports the numpy library and renames it as **np**.
- **import matplotlib.pyplot as plt**: Imports the pyplot module from the matplotlib library and renames it as **plt**
- **pc1 = np.loadtxt('pc1.csv', delimiter=',')**: Loads the data from the **pc1.csv** file, assuming it is a comma-separated value (CSV) file. The resulting data is stored in a numpy array called **pc1**.
- **pc2 = np.loadtxt('pc2.csv', delimiter=',')**: Loads the data from the **pc2.csv** file, assuming it is a CSV file. The resulting data is stored in a numpy array called **pc2**.
- **data = np.concatenate([pc1, pc2])**: Concatenates the two numpy arrays **pc1** and **pc2** along their first (and only) axis, resulting in a single numpy array called **data** that contains all the data from both files.

A function is defined to plot the data and the fitted surface. The **plotting** function takes two parameters: f and i -
f represents the type of linear regression to be plotted
i represents the position of the plot within the overall figure

```
#Define a function to plot a graph
def plotting(f,i):
    print("Coefficients for" , f, ": ", n)                         # Print the coefficients of the fitted
plane
    ax = fig.add_subplot(1, 3, i, projection='3d')
    ax.scatter(data[:,0], data[:,1], data[:,2], c='r', marker='o')
    x_vals, y_vals = np.meshgrid(np.linspace(min(data[:,0]), max(data[:,0]), 10),
np.linspace(min(data[:,1]), max(data[:,1]), 10))
    # Plot the data and the fitted plane
    if f == 'Standard Least Squares':
        z_vals = n[0]*x_vals + n[1]*y_vals + n[2]
    elif f == 'Total Least Squares':
        z_vals = (-n[0]*x_vals - n[1]*y_vals - n[3]) / n[2]
    elif f == 'RANSAC':
        z_vals = (-n[3] - n[0]*x_vals - n[1]*y_vals) / n[2]
    ax.plot_surface(x_vals, y_vals, z_vals, alpha = 0.5)
    ax.set_xlabel('X-Coordinates')
    ax.set_ylabel('Y-Coordinates')
    ax.set_zlabel('Z-Coordinates')
    plt.title(f)
```

- **print("Coefficients for", f, ": ", n):** This command prints out the coefficients of the fitted plane. n is assumed to be defined outside of the function and contains the coefficients of the fitted plane.
- **ax = fig.add_subplot(1, 3, i, projection='3d'):** This command creates a 3D subplot within the overall figure. The subplot is positioned in the ith position of a 1x3 grid.
- **ax.scatter(data[:, 0], data[:, 1], data[:, 2], c='r', marker='o'):** This command plots the data as a scatter plot in the 3D subplot. data is assumed to be defined outside of the function and contains the data to be plotted. The first column of data is plotted on the x-axis, the second column is plotted on the y-axis, and the third column is plotted on the z-axis. The points are colored red and plotted as circles.
- **x_vals, y_vals = np.meshgrid(np.linspace(min(data[:, 0]), max(data[:, 0]), 10), np.linspace(min(data[:, 1]), max(data[:, 1]), 10)):** This command creates a grid of points to be used for plotting the fitted plane. np.meshgrid creates a 2D grid of points from 1D arrays created by np.linspace. The grid spans the range of the x-axis and y-axis values of the data and is divided into 10 equally spaced points along each axis.
- **f:** The type of estimation method f is checked using if-else statements and the corresponding z_vals are calculated using the meshgrid values of x and y axis in the plane equation. Further explanation about the calculation is given in the relevant section.
- **ax.plot_surface(x_vals, y_vals, z_vals, alpha=0.5):** This command plots the fitted plane with transparency 50% using the x, y, and z values stored in x_vals, y_vals, and z_vals.
- **ax.set_xlabel, ax.set_ylabel, ax.set_zlabel:** These commands label the axes.
- **plt.title(f):** Each subplot is given the Estimation Method name as its title.

1. a. **Computing the Covariance of a Matrix -**

Covariance is a measure of how two variables are related to each other. Specifically, it measures the degree to which changes in one variable are associated with changes in another variable. Mathematically, the covariance between two variables X and Y is defined as:

$$cov(X,Y) = E[(X - E[X]) * (Y - E[Y])]$$

where E[X] and E[Y] are the expected values (i.e., means) of X and Y, respectively

In this formula, we subtract the mean of each variable from their respective values, multiply the differences together, and then take the expected value of the resulting product.

We define a function to calculate the covariance matrix so that it can later be used in Total Least Squares method.

```
def cov_mat(M):
    mean = np.mean(M,axis=0)                                    # Compute the mean of all dimensions
    center = M - mean                                           # Center the point cloud data around 
the mean

    # Compute and print the covariance matrix
    cov_matrix = np.zeros((3, 3))
    for i in range(3):
        for j in range(3):
            cov_matrix[i][j] = np.sum(center[:,i] * center[:,j]) / (len(center) - 1)
    return cov_matrix, mean

mat, _ = cov_mat(pc1)
print('\nCovariance Matrix for pc1.csv:\n', mat)
```

- **np.mean(M, axis=0)**: This calculates the mean of all dimensions (i.e. x, y, and z coordinates) of the point cloud data **M**.
- **center = M - mean**: This centers the point cloud data **M** around the mean.
- **cov_matrix = np.zeros((3, 3))**: This initializes an empty 3x3 matrix to store the covariance matrix.
- **for i in range(3), for j in range(3)**: This loops over each dimension (i.e. x, y, and z coordinates).
  cov_matrix[i][j] = np.sum(center[:,i] * center[:,j]) / (len(center) - 1): This calculates the covariance between dimensions **i** and **j**. The formula calculates the covariance between dimensions **i** and **j** by taking the sum of the element-wise products of the centered data in dimension i and the centered data in dimension j, and dividing by the number of data points minus 1. This process is repeated for all combinations of i and j, resulting in a full 3x3 covariance matrix.
- The Covariance matrix and mean are returned and the matrix for the data pc1.csv is printed

b. **Computing the Magnitude and Direction of Surface Normal –**

The eigenvectors correspond to the principal axes of the point cloud data and the eigenvalues represent the variances of the point cloud data along these axes. Eigenvalues and eigenvectors are calculated using matrix operations. Given a square matrix A, an eigenvector x and its corresponding eigenvalue λ satisfy the following equation:

$$Ax = \lambda x$$

This equation can be rearranged as:

$$(A - \lambda I)x = 0$$

where I is the identity matrix.

This equation has a non-trivial solution (i.e., $x \neq 0$) if and only if the determinant of (A - λ I) is equal to zero. Thus, to find the eigenvalues, we need to solve the equation:

$$det(A - \lambda I) = 0$$

This gives us a polynomial equation of degree n (where n is the size of the matrix A) that we can solve for the eigenvalues λ.

Once the eigenvalues are found, we can find the corresponding eigenvectors by solving the equation (A - λ I) x = 0 for each eigenvalue. The solution to this equation is a vector x that satisfies the equation, and the set of all such solutions for a given eigenvalue form a subspace of the vector space. We can then find a basis for this subspace (i.e., a set of linearly independent vectors that span the subspace) using methods such as Gaussian elimination or Gram-Schmidt orthogonalization. These basis vectors are the eigenvectors corresponding to the eigenvalue λ.
In Python, NumPy provides a convenient function, **np.linalg.eig()**, for computing the eigenvalues and eigenvectors of a matrix. The function takes a matrix as input and returns two arrays: the first array contains the eigenvalues, and the second array contains the corresponding eigenvectors as columns.
The eigenvector corresponding to the smallest eigenvalue is selected as the surface normal vector of the plane. This is because the smallest eigenvalue represents the direction with the least variance, which corresponds to the normal direction of the plane.

```
eig_vals, eig_vecs = np.linalg.eig(mat)                            # Compute the Eigenvectors and Eigenvalues of the Covariance Matrix
surface_normal = eig_vecs[:, np.argmin(eig_vals)]                  # Compute the Surface Normal as the Eigenvector corresponding to the smallest Eigenvalue
surface_normal_magnitude = np.linalg.norm(surface_normal)          # Compute the magnitude of the Surface Normal

print(f"Surface Normal to the flat, ground plane ----- Vector: {surface_normal} and Magnitude: {surface_normal_magnitude}\n")   # Print the Surface Normal Direction and its Magnitude
```

- **eig_vals, eig_vecs = np.linalg.eig(mat):** This calculates the Eigenvalues and Eigenvectors of the covariance matrix **mat**.
- **surface_normal = eig_vecs[:, np.argmin(eig_vals)]:** This selects the Eigenvector with the smallest corresponding Eigenvalue as the surface normal. The notation [:, np.argmin(eig_vals)] selects the column of the Eigenvector matrix corresponding to the index of the smallest Eigenvalue.
- **surface_normal_magnitude = np.linalg.norm(surface_normal):** This calculates the magnitude of the surface normal vector using the Euclidean norm. The notation np.linalg.norm(surface_normal) calculates the square root of the sum of the squares of the components of surface_normal.
- The Surface Normal vector and Magnitude for the covariance matrix ‘**mat**’ are printed

2. a. **Standard Least Squares –**

The Theory of Standard Least Squares is explained earlier in the report.

```
'''-----------------Standard Least Squares----------------------'''
A = np.column_stack((data[:,0], data[:,1], np.ones(len(data))))
b = data[:,2]

# Compute the Normal Equations Matrix
ATA = np.matmul(A.T, A)
ATb = np.matmul(A.T, b)

n = np.dot(np.linalg.inv(ATA), ATb)
fig = plt.figure()
plotting('Standard Least Squares',1)
```

- **A = np.column_stack((data[:,0], data[:,1], np.ones(len(data))))**: We create a matrix **A** by horizontally stacking three arrays: the first column of **data**, the second column of **data**, and an array of ones with length equal to the number of rows in **data**. This matrix **A** represents the coefficients of the plane we are fitting.
- **b = data[:,2]**: We create a vector **b** containing the third column of **data**. This vector represents the z-coordinates of the data points.
- **ATA = np.matmul(A.T, A)**: We compute the product of the transpose of matrix **A** and matrix **A**, which results in a 3x3 matrix.
- **ATb = np.matmul(A.T, b)**: We compute the product of the transpose of matrix **A** and vector **b**, which results in a 3x1 vector.
- **n = np.dot(np.linalg.inv(ATA), ATb)**: We solve the normal equations for the coefficients of the plane. Specifically, we invert the 3x3 matrix **ATA** and multiply it by the vector **ATb**. The resulting vector **n** contains the coefficients of the plane in the form **[a, b, c]** which represents the coefficients of x, y and constant terms respectively.
- **fig = plt.figure()**: We create a new figure for plotting the 3D data and the fitted plane.
- **z_vals = n[0]*x_vals + n[1]*y_vals + n[2]:** This is obtained from the plane equation z = ax + by + c where where n[0] is the coefficient a, n[1] is the coefficient b, and n[2] is the coefficient c.
- **plotting('Standard Least Squares',1)**: We use this function to plot the dataset and fitted plane in subplot 1

Plot:

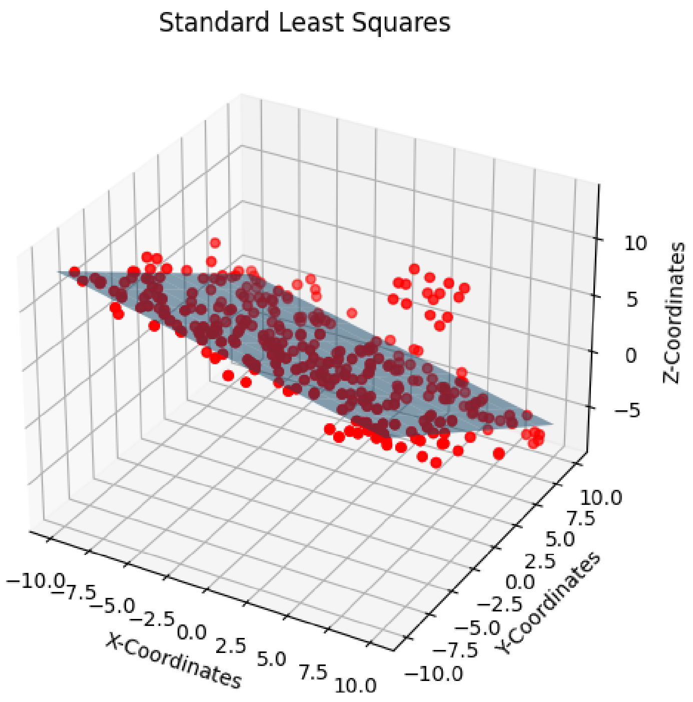

Figure 2. Plot with the fitted plane using Standard Least Squares

Total Least Squares –

Total Least Squares (TLS) is a regression analysis method that aims to minimize the distance between the observed data points and the fitted model, considering both the measurement errors in the dependent and independent variables. In other words, it is a technique for finding the best-fit line for data points when there is noise or measurement error in both the x and y variables.
In the context of 3D point clouds, TLS can be used to estimate the equation of a plane that best fits a set of noisy point data. The equation of the plane can be written in the form:

$$ax + by + cz + d = 0$$

where a, b, c are the components of the plane's normal vector
d is the offset from the origin along the normal vector.

The goal is to find the values of a, b, c, and d that best fit the data points in a least-squares sense.
The first step in TLS is to center the data points by subtracting their mean from each coordinate. This ensures that the plane passes through the centroid of the data points. Then, a covariance matrix is computed from the centered data points, as described in the standard least squares method.
Next, a singular value decomposition (SVD) is performed on the covariance matrix. SVD is a factorization of a matrix into three matrices,

$$A = U\Sigma V^T$$

where U and V are orthogonal matrices with left and right singular vectors respectively
Σ is a diagonal matrix of singular values

The singular vectors are the basis vectors that transform the input data into a new coordinate system that maximizes the variance in the first principal component, followed by the second, and so on. The singular values correspond to the magnitude of the variance along each principal component.

In TLS, we sort the singular values in decreasing order and take the last row of the right singular vectors as the normal vector of the plane. This is because the last row corresponds to the smallest singular value, which represents the dimension with the least variance and thus the most affected by noise. Taking this dimension as the normal vector ensures that the plane fits the data points in a least-squares sense, while also minimizing the effect of noise. After obtaining the normal vector, we compute the offset of the plane from the origin using the centroid of the data points and the normal vector.

```
'''-----------------Total Least Squares----------------------'''
_, c, m = cov_mat(data)

#Creating a Manual Partial Singular Value Decomposition Function
ATA = np.dot(c.T, c)
val, vec = np.linalg.eig(ATA)                                     # Compute the eigenvalues and
eigenvectors of A^T A
S = np.sqrt(val)                                                  # Compute the singular values
V = vec[:,np.argsort(-S)].T                                       # Sort the singular vectors in
descending order
n = V[-1, :]                                                      # Extract the last row of V to get the
normal vector
n = n / np.linalg.norm(n)                                         # Normalize the vector
# Translate the plane back to its original position
o = -np.dot(n, m)                                                 # Calculate offset of the plane
n = np.append(n, o)
plotting('Total Least Squares',2)
```

- **c, m = cov_mat(data)**: Computes the covariance matrix **c** and the mean vector **m** of the input data using the **cov_mat()** function.
- **ATA = np.dot(c.T, c)**: Computes the matrix $\mathbf{A^T A}$ (**A** is the design matrix) using the covariance matrix **c**.
- **vals, vecs = np.linalg.eig(c):** This calculates the Eigenvalues and Eigenvectors of the covariance matrix **c**.
- **S = np.sqrt(val)**: This computes the singular values of **A** from the eigenvalues of $\mathbf{A^T A}$.
- **V = vec[:,np.argsort(-S)].T**: This line sorts the singular vectors (columns of **V**) in descending order of their corresponding singular values (**S**) and transposes the resulting matrix.
- **n = V[-1, :]**: This line extracts the last row of **V** as the normal vector of the plane.

- **n = n / np.linalg.norm(n)**: This line normalizes the normal vector to have unit length.
- **o = -np.dot(n, m)**: This line calculates the offset of the plane by taking the dot product of the normal vector and the mean vector of the input data with a minus sign.
- **n = np.append(n, o)**: This line appends the offset value to the end of the normal vector to obtain the final equation of the plane.
- **z_vals = (-n[0]*x_vals - n[1]*y_vals - n[3]) / n[2]:** This is obtained from the plane equation ax + by + cz + d = 0 which is rearranged to z = (-ax - by - d) / c where n[0] is the coefficient a, n[1] is the coefficient b, n[2] is the coefficient c and n[3] is the coefficient of d.
- **plotting('Total Least Squares',2)**: We use this function to plot the dataset and fitted plane in subplot 2

Plot:

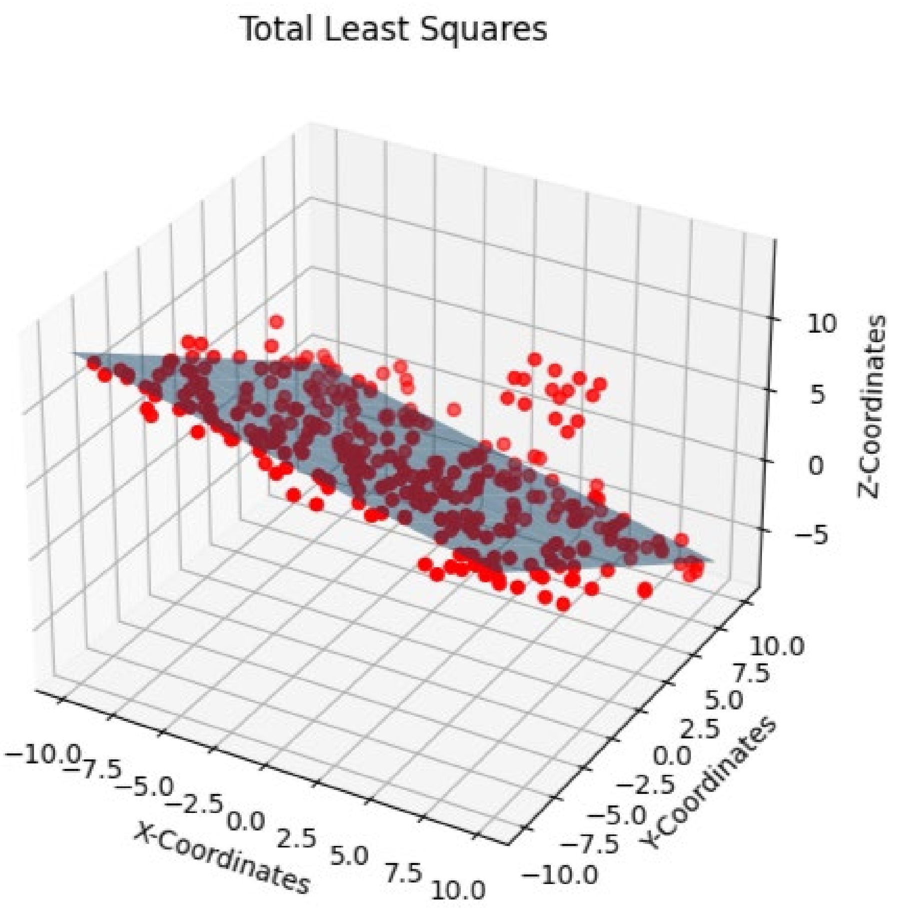

Figure 3. Plot with the fitted plane using Total Least Squares

b. **RANSAC –**

RANSAC (Random Sample Consensus) is used for estimating parameters of a mathematical model from a set of observed data. It is often used to estimate the parameters of a mathematical model from a set of noisy data, where the data contains outliers that may cause standard estimation techniques to fail.

RANSAC is an iterative algorithm that works as follows:

i. Randomly sample a subset of the data points to form an initial estimate of the model parameters.
ii. Use this initial estimate to fit a model to the entire dataset and identify the inliers, i.e., the data points that fit the model well.
iii. If the number of inliers is greater than some threshold, re-estimate the model parameters using all the inliers.
iv. Repeat steps i-iii a fixed number of times or until convergence to obtain the best model.

The RANSAC algorithm is useful in scenarios where the data has outliers, i.e., points that do not fit the model well. In these cases, standard techniques such as least-squares estimation can be highly sensitive to the presence of outliers, resulting in poor model estimates. RANSAC is more robust to outliers, as it iteratively discards outliers and only considers the inliers when estimating the model parameters.

```
'''-----------------RANSAC----------------------'''
threshold = 0.1
max_iters = 1000
best_inliers = []
i=1
while i <= max_iters:
    # Randomly sample 3 points from the data
    sample = data[np.random.choice(data.shape[0], size=3, replace=False)]

    # Fit a plane to the sampled points
    p1, p2, p3 = sample
    normal = np.cross(p2-p1, p3-p1)                                 # Using the normal vector obtained by
the cross-product of two vectors
    if np.linalg.norm(normal) < 1e-6:                               # Check if magnitude of normal vector
is very small
        continue
    o = -np.dot(normal, p1)
    plane = np.concatenate((n, [o]))
    distances = np.abs(np.dot(data, plane[:3]) + plane[3]) / np.linalg.norm(plane[:3])  # Calculate the
distance of all points from the plane
    inliers = np.where(distances <= threshold)[0]                   # Count the number of inliers (points
within the threshold distance from the plane)

    # Update the best model if the current model has more inliers
    if len(inliers) > len(best_inliers):
        n = plane
        best_inliers = inliers
    i += 1
plotting('RANSAC',3)
```

- **threshold = 0.1**: Threshold distance for inliers is taken as 0.1 (10% of the maximum distance)
- **max_iters = 1000**: Maximum number of iterations for RANSAC algorithm
- **best_inliers = []**: Array to store the best set of inliers obtained so far
- **i=1**: This line initializes a counter variable for the while loop that iterates over the RANSAC algorithm.
- **while i <= max_iters**: Start a loop that runs for at most **max_iters** iterations
- **sample = data[np.random.choice(data.shape[0], size=3, replace=False)]**: This selects the rows from the data array that correspond to 3 randomly chosen (without replacement) indices. The resulting sample array contains 3 points that will be used to fit a plane.
- **p1, p2, p3 = sample**: Assign the 3 sampled points to variables **p1**, **p2**, and **p3**
- **normal = np.cross(p2-p1, p3-p1)**: Calculate the normal vector of the plane passing through the 3 sampled points using cross-product
- **if np.linalg.norm(normal) < 1e-6: continue**: Check if magnitude of normal vector is very small, if so, skip the current iteration

- **o = -np.dot(normal, p1)**: Calculates the offset of the plane using the dot product of the normal vector and one of the points on the plane
- **plane = np.concatenate((n, [o]))**: Concatenate the normal vector and offset to create the plane model
- **distances = np.abs(np.dot(data, plane[:3]) + plane[3]) / np.linalg.norm(plane[:3])**: This line calculates the distances of all points in the data array from a plane specified by the plane equation, where plane[:3] is a normal vector to the plane and plane[3] is an offset from the origin.To compute the distances, the equation of the plane is used to calculate the value of plane[0]*x + plane[1]*y + plane[2]*z + plane[3] for each point in the data array. The absolute value of this value is then divided by the magnitude of the normal vector **np.linalg.norm(plane[:3]),** which gives the distance from the plane to the point. The resulting **distances** array contains the distances of all points in data from the plane specified by plane.
- **inliers = np.where(distances <= threshold)[0]**: Creates a boolean array of points that are within the threshold distance of the plane and stores their indices as inliers.
- **if len(inliers) > len(best_inliers): n = plane, best_inliers = inliers**: Update the best model and best set of inliers if the current model has more inliers
- **z_vals = (-n[0]*x_vals - n[1]*y_vals - n[3]) / n[2]:** This is obtained from the plane equation ax + by + cz + d = 0 which is rearranged to z = (-ax - by - d) / c where n[0] is the coefficient a, n[1] is the coefficient b, n[2] is the coefficient c and n[3] is the coefficient of d.
- **plotting('RANSAC',3)**: We use this function to plot the dataset and fitted plane in subplot 3

Plot:

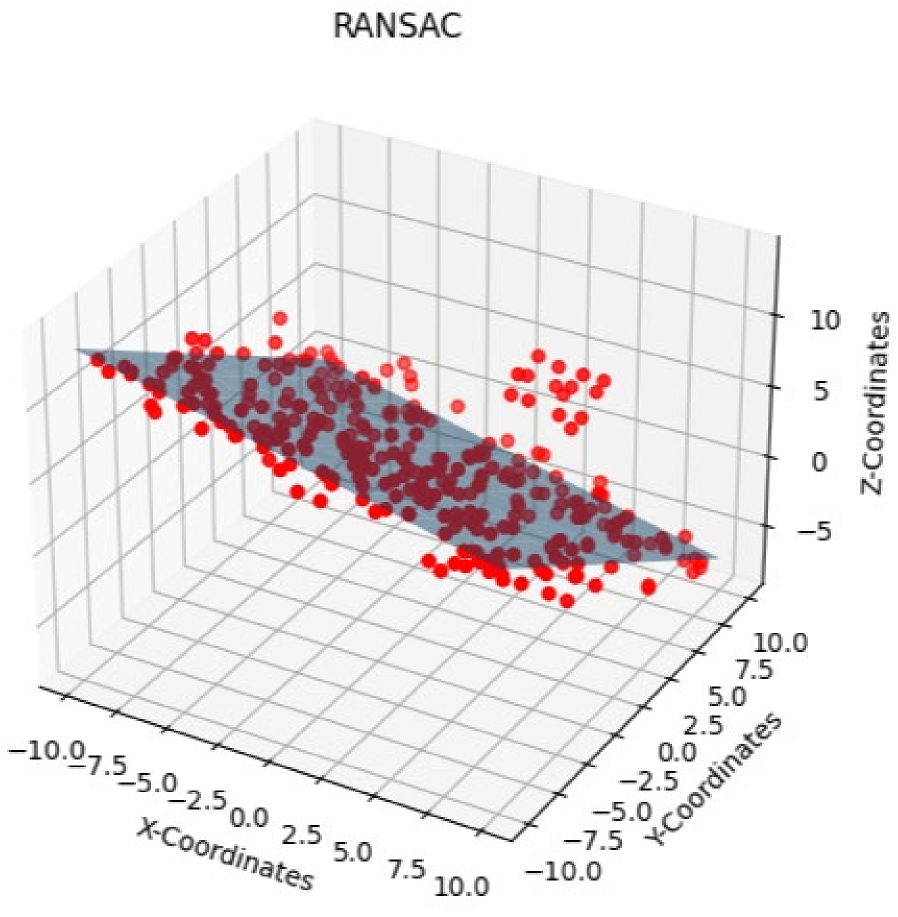

Figure 4. Plot with the fitted plane using RANSAC

Full Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Load the data from the CSV files
pc1 = np.loadtxt('pc1.csv', delimiter=',')
pc2 = np.loadtxt('pc2.csv', delimiter=',')

data = np.concatenate([pc1, pc2])                                   # Combine the two dataframes into one

#Define a function to plot a graph
def plotting(f,i):
    print("Coefficients for" , f, ": ", n)                          # Print the coefficients of the fitted
plane
    ax = fig.add_subplot(1, 3, i, projection='3d')
    ax.scatter(data[:,0], data[:,1], data[:,2], c='r', marker='o')
    x_vals, y_vals = np.meshgrid(np.linspace(min(data[:,0]), max(data[:,0]), 10),
np.linspace(min(data[:,1]), max(data[:,1]), 10))
    # Plot the data and the fitted plane
    if f == 'Standard Least Squares':
        z_vals = n[0]*x_vals + n[1]*y_vals + n[2]
    elif f == 'Total Least Squares':
        z_vals = (-n[0]*x_vals - n[1]*y_vals - n[3]) / n[2]
    elif f == 'RANSAC':
        z_vals = (-n[0]*x_vals - n[1]*y_vals - n[3]) / n[2]
    ax.plot_surface(x_vals, y_vals, z_vals, alpha = 0.5)
    ax.set_xlabel('X-Coordinates')
    ax.set_ylabel('Y-Coordinates')
    ax.set_zlabel('Z-Coordinates')
    plt.title(f)

'''--------------------------------------------------Question 1: For pc1.csv-------------------------------
----------------------------------------------------'''

'''----------------------a: Computing Covariance Matrix---------------------'''
#Define a function to calculate the Covariance of a Matrix
def cov_mat(M):
    mean = np.mean(M,axis=0)                                        # Compute the mean of all dimensions
    center = M - mean                                               # Center the point cloud data around
the mean

    # Compute and print the covariance matrix
    cov_matrix = np.zeros((3, 3))
    for i in range(3):
        for j in range(3):
            cov_matrix[i][j] = np.sum(center[:,i] * center[:,j]) / (len(center) - 1)
    return cov_matrix, mean

mat, _ = cov_mat(pc1)
print('\nCovariance Matrix for pc1.csv:\n', mat)

'''--------------b:  Computing Magnitude and Direction of Surface Normal using Covariance Matrix-----------
-------'''
eig_vals, eig_vecs = np.linalg.eig(mat)                             # Compute the Eigenvectors and
Eigenvalues of the Covariance Matrix
surface_normal = eig_vecs[:, np.argmin(eig_vals)]                   # Compute the Surface Normal as the
Eigenvector corresponding to the smallest Eigenvalue
surface_normal_magnitude = np.linalg.norm(surface_normal)           # Compute the magnitude of the Surface
Normal

print(f"Surface Normal to the flat, ground plane ----- Vector: {surface_normal} and Magnitude:
{surface_normal_magnitude}\n")   # Print the Surface Normal Direction and its Magnitude

'''-------------------------------------------------Question 2. For pc1.csv and pc2.csv ------------------
-----------------------------------------------------'''

'''----------------------Part (a)---------------------'''

'''-----------------Standard Least Squares----------------------'''
```

```
A = np.column_stack((data[:,0], data[:,1], np.ones(len(data))))
b = data[:,2]

# Compute the Normal Equations Matrix
ATA = np.matmul(A.T, A)
ATb = np.matmul(A.T, b)

n = np.dot(np.linalg.inv(ATA), ATb)
fig = plt.figure()
plotting('Standard Least Squares',1)


'''-----------------Total Least Squares----------------------'''
c, m = cov_mat(data)

#Creating a Manual Partial Singular Value Decomposition Function
ATA = np.dot(c.T, c)
val, vec = np.linalg.eig(ATA)                                     # Compute the eigenvalues and
eigenvectors of A^T A
S = np.sqrt(val)                                                  # Compute the singular values
V = vec[:,np.argsort(-S)].T                                       # Sort the singular vectors in
descending order
n = V[-1, :]                                                      # Extract the last row of V to get the
normal vector
n = n / np.linalg.norm(n)                                         # Normalize the vector
# Translate the plane back to its original position
o = -np.dot(n, m)                                                 # Calculate offset of the plane
n = np.append(n, o)
plotting('Total Least Squares',2)

'''----------------------Part (b)---------------------'''

'''-----------------RANSAC----------------------'''
threshold = 0.1
max_iters = 1000
best_inliers = []
i=1
while i <= max_iters:
    # Randomly sample 3 points from the data
    sample = data[np.random.choice(data.shape[0], size=3, replace=False)]

    # Fit a plane to the sampled points
    p1, p2, p3 = sample
    normal = np.cross(p2-p1, p3-p1)                               # Using the normal vector obtained by
the cross-product of two vectors
    if np.linalg.norm(normal) < 1e-6:                             # Check if magnitude of normal vector
is very small
        continue
    o = -np.dot(normal, p1)
    plane = np.concatenate((n, [o]))
    distances = np.abs(np.dot(data, plane[:3]) + plane[3]) / np.linalg.norm(plane[:3])  # Calculate the
distance of all points from the plane
    inliers = np.where(distances <= threshold)[0]                 # Count the number of inliers (points
within the threshold distance from the plane)

    # Update the best model if the current model has more inliers
    if len(inliers) > len(best_inliers):
        n = plane
        best_inliers = inliers
    i += 1
plotting('RANSAC',3)

fig.suptitle('Fitting a surface to the data using various estimation Algorithms')
plt.show()
print('\n')
```

Plot:

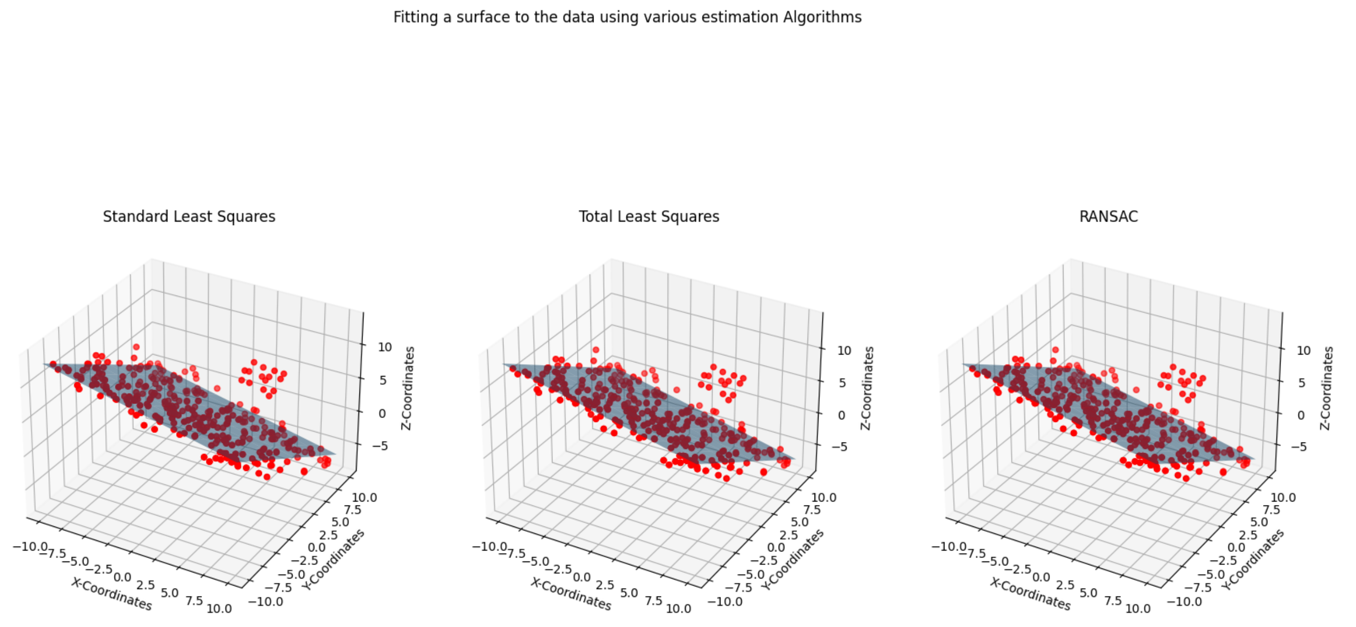

Figure 5. Plots with surfaces computed using various Estimation Algorithms

Terminal Output:

```
PS C:\Users\manda\OneDrive - University of Maryland\Perception For Autonmous Robots\Project 1> &
"C:/Program Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of Maryland/Perception For
Autonmous Robots/Project 1/problem2.py"

Covariance Matrix for pc1.csv:
 [[ 33.7500586   -0.82513692 -11.39434956]
 [ -0.82513692  35.19218154 -23.23572298]
 [-11.39434956 -23.23572298  20.62765365]]
Surface Normal to the flat, ground plane ----- Vector: [0.28616428 0.53971234 0.79172003] and Magnitude:
1.0

Coefficients for Standard Least Squares :  [-0.2997447  -0.67026719  3.44546813]
Coefficients for Total Least Squares :  [ 0.25336279  0.56455808  0.78554533 -2.70518836]
Coefficients for RANSAC :  [ 0.25336279  0.56455808  0.78554533 -2.70518836 22.53942768]
```

Interpretation:

From the outputs, we can see that the coefficients obtained from the **Standard Least Squares** method are [-0.2997447, -0.67026719, 3.44546813]. These coefficients define the plane equation obtained from the method. However, we can also observe that the residuals (i.e., the distances between the data points and the fitted plane) are relatively large, indicating that it is sensitive to outliers and noise in data. Therefore, the standard least squares method does not provide a particularly good fit to the data.

On the other hand, the coefficients obtained from the **Total Least Squares** method are [0.25336279, 0.56455808, 0.78554533, -2.70518836]. These coefficients define the plane equation obtained from the method. The coefficients for Total Least Squares also include an additional term, which represents the offset of the plane from the origin. The main advantage of the total least squares method is that it minimizes the orthogonal distances between the data points and the fitted plane, rather than just the vertical distances as in the standard least squares method. Therefore, it considers errors in both the input data and the measurement process, indicating that this method provides a better fit to the data compared to the standard least squares method.

Finally, we can see that the coefficients obtained from the **RANSAC** method are [0.25336279, 0.56455808, 0.78554533, -2.70518836, 15.64135487]. The additional coefficient corresponds to the offset of the plane, which was not included in the standard and total least squares methods. The RANSAC method is an iterative method that randomly selects subsets of data points and fits a plane to them. This process is repeated multiple times, and the plane with the largest number of inliers (i.e., data points that are within a certain distance of the fitted plane) is selected as the best fit. Therefore, the coefficients obtained from the RANSAC method correspond to the plane equation obtained from the best fit plane.

Conclusion: In this case, we can see that the coefficients obtained from the RANSAC method are the same as the coefficients obtained from the total least squares method, indicating that both methods provide a similar fit to the given data.

Pipeline:

- The code starts by importing necessary libraries including numpy and matplotlib.
- The point cloud data in CSV format is loaded into numpy arrays named pc1 and pc2 using the loadtxt() method.
- The two arrays pc1 and pc2 are concatenated into a single array called 'data' using the concatenate() method.
- A function named 'plotting' is defined to plot the graph of the fitted plane with the given coefficients. This function takes two parameters: f (the type of plane fitting method) and i (the subplot number).
- A function named 'cov_mat' is defined to compute the covariance matrix of the given matrix M. This function takes a single parameter M (the matrix) and returns the covariance matrix and the mean of all dimensions.
- The covariance matrix of the points in pc1.csv is computed using the cov_mat() function, and the result is printed to the console.
- The eigenvectors and eigenvalues of the covariance matrix are computed using the eig() method in numpy.
- The eigenvector corresponding to the smallest eigenvalue is considered as the normal vector to the ground plane.
- The magnitude of the surface normal vector is computed using the norm() method in numpy.
- The surface normal direction and magnitude are printed to the console.
- The code proceeds to perform plane fitting on the data using three different methods - standard least squares, total least squares, and RANSAC.
- For standard least squares plane fitting, a matrix A is constructed from the x and y coordinates of the data, and a vector b is constructed from the z coordinates.
- The normal equations matrix is computed as ATA = A.T * A and the right-hand side of the normal equations is computed as ATb = A.T * b.
- The coefficients of the fitted plane are computed using the equation n = inv(ATA) * ATb.
- The plotting() function is called with the type of plane fitting method and the subplot number as parameters to plot the fitted plane.
- For total least squares plane fitting, the data is centered and a covariance matrix 'c' is computed using the cov_mat() function.
- The partial singular value decomposition is performed manually on the covariance matrix 'c' to obtain the normal vector.
- The plane is translated back to its original position and plotted using the plotting() function.
- For RANSAC plane fitting, a threshold value and maximum number of iterations are set.
- The code enters a loop that will run for a maximum of 'max_iters' iterations.
- In each iteration, a random sample of 3 points is selected from the data.
- The normal vector of the plane passing through these points is computed using the cross-product of two vectors.
- The distance of each point from the fitted plane is computed, and points whose distances are less than the threshold value are considered as inliers.
- If the number of inliers is greater than the current best inlier count, the current set of inliers is stored as the best set.
- After the loop ends, the best set of inliers is used to fit a plane, and the coefficients of the fitted plane are plotted using the plotting() function.
- The code ends.