2022

ENPM673: Homework 1

BHARADWAJ CHUKKALA || 118341705

Problem 1: [20 Points]

Assume that you have a camera with a resolution of 5MP where the camera sensor is square shaped with a width of 14mm. It is also given that the focal length of the camera is 25mm.

1. Compute the Field of View of the camera in the horizontal and vertical direction.
2. Assuming you are detecting a square shaped object with width 5cm, placed at a distance of 20 meters from the camera, compute the minimum number of pixels that the object will occupy in the image.

ANSWER:

Given:

We are given the values for focal length (f = 25mm), sensor size (s = 14mm) and the resolution (Res = 5mp) of the camera sensor and, the working distance (D = 20m) at which the object is placed.

Part 1:

From this we can calculate the field of view. The field of view is the extent of the observable world that is seen at any given moment.

To calculate the field of view we should know the angle of view of the corresponding sensor. We can calculate the angle of view by substituting the given in the formula

$$AOV = 2\tan^{-1}(\frac{s}{2f})$$

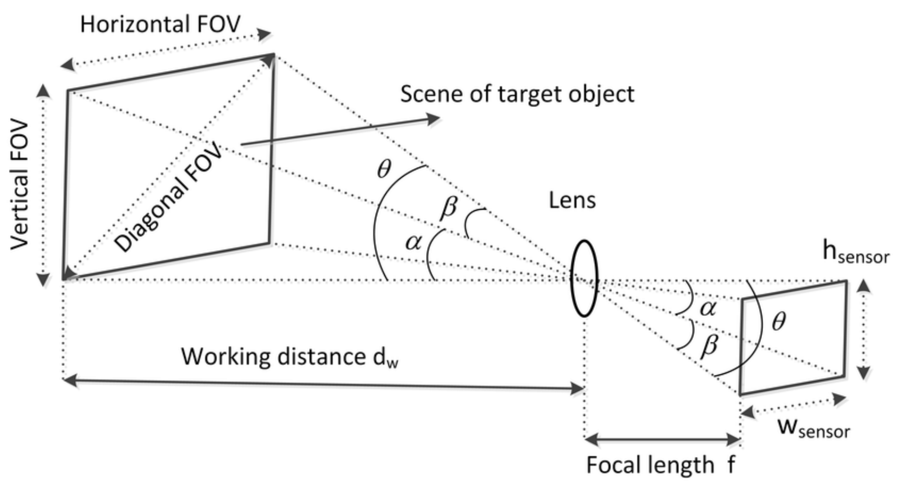
Now that we got the angle of view, we can calculate the fields of view. There are two fields of view, horizontal and vertical. For this case, the sensor being a squared shaped one, we get to the conclusion that $HFOV = VFOV$. This basically means that the aspect ratio of the image being captured by the camera is 1:1. To calculate the fields of view we can use the properties of similar triangles. We end up with the formula.

$$HFOV = VFOV = 2D(tan\left(\frac{AOV}{2}\right))$$

Upon substitution of the required fields, we get the Fields of view.

Part 2:

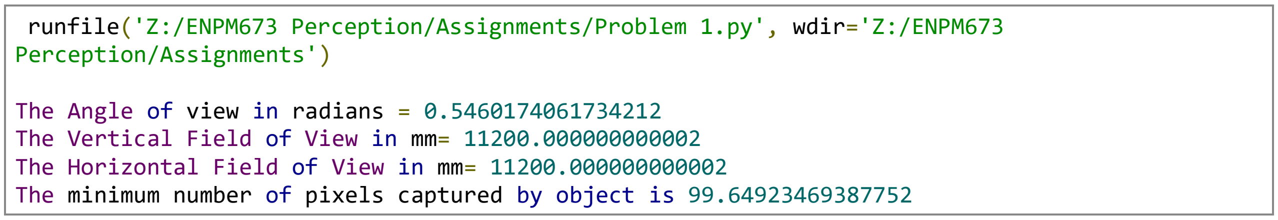
To find the pixels of the object in the image. I have firstly calculated the real time area that the sensor will span upon capturing an image. Now the image will have a resolution of 5MP which means the image can be encompassed in 5 million pixels. I have the object sizes which give the 2D area that the object occupies in real time. I get the pixel rate of how much area will one pixel cover. As I have the area of object and pixel rate. I can calculate the minimum number of pixels that the object occupies in the captured image.



Code Snippet:

```
import math

#Given data

Res = 5000000 #Resolution is 5Mp and it captures an image in 5 Million pixels
hs = 14 #height of sensor
ws = hs #width of sensor
f = 25 #focal length of camera
D = 20000 #Working distance is2 0m
ho = 50 #Object height
wo = 50 #Object width


#The vertical and Horizontal Angles of view are same because the sensor is squared shaped
Aov = 2*(math.atan(hs/(2*f)))
print("The Angle of view in radians =", Aov)

#To calculate field of view, we use the properties of similiar triangles
VFov =  2 * D * math.tan((Aov/2))
HFov = VFov

print("The Vertical Field of View in mm=", VFov)
print("The Horizontal Field of View in mm=", HFov)


#To calculate the number of pixels of occupied by object in the captured image
CArea = VFov * HFov #Realtime area of the image
ppmm = CArea/Res #Pixel rate

OArea = ho * wo
Obj_pix = OArea / ppmm  #Number of pixels occupied by the object in the image

print ("The minimum number of pixels captured by object is",Obj_pix)
```

Terminal window displaying the Results:



```
 runfile('Z:/ENPM673 Perception/Assignments/Problem 1.py', wdir='Z:/ENPM673
Perception/Assignments')

The Angle of view in radians = 0.5460174061734212
The Vertical Field of View in mm= 11200.000000000002
The Horizontal Field of View in mm= 11200.000000000002
The minimum number of pixels captured by object is 99.64923469387752
```

Problem 2: [30 Points]

A ball is thrown against a white background and a camera sensor is used to track its trajectory. We have a near perfect sensor tracking the ball in video1 and the second sensor is faulty and tracks the ball as shown in video2. Clearly, there is no noise added to the first video whereas there is significant noise in the second video. Assuming that the trajectory of the ball follows the equation of a parabola:

1. Use Standard Least Squares to fit curves to the given videos in each case. You have to plot the data and your best fit curve for each case. Submit your code along with the instructions to run it.

   (Hint: Read the video frame by frame using OpenCV's inbuilt function. For each frame, filter the red channel for the ball and detect the topmost and bottom most colored pixel and store it as X and Y coordinates. Use this information to plot curves.)

ANSWER:

Conceptual understanding behind solving the problem

Here, our goal is to fit a parametric model that describes the equation of the parabola to a series of points labeled as $(x_1, y_2), (x_2, y_2), \dots, (x_n, y_n)$. If we model the line as $ax^2 + bx + c = y$, our goal is to estimate the parameters a, b, c to minimize the value of E (error). E is the sum of residuals or how far off our estimate is for each point. We can interpret this as choosing a line to minimize the distance between the line and the observed data points.

We can formulate this easily as a least squares problem. $E = \sum_{i=0}^{n}(y_i - ax^2 - bx - c)$ describes the objective function we are trying to minimize. We can re-write this equation in the form of $E = \left|\left|Y - XB\right|\right|^2$, where $\mathrm{B} = [\mathrm{a\ b\ c}]^{\mathrm{T}}$ and $X = [x^2\ x\ 1]$ and Y is the vector containing all y values of our data. Our goal is to find the parameters a, b, c that minimizes E. These can be found by taking the derivative of E with respect to B and equating it to zero.
$\frac{dE}{dB} = -2X^TY + 2X^TXB = 0$ Solving this equation allows to find B which is can be expressed as $B = (X^TX)^{-1}X^TY$. This solution can be interpreted as the optimal (in the least squares sense)

parameters a, b, c that minimize the residuals, or, equivalently, choosing a parabola that comes closest to best fitting most of the points.

Steps followed:

- Read the file
- Convert the image to grayscale
- Set thresholds to get a binary output
- Extract the black pixels
- Find the centroid of the blob
- Use the x, y values to plot the graph

Part 1: ballvideo1.mp4

```
 #Problem 2 part 1

import cv2
import matplotlib.pyplot as plt
import numpy as np
Video = cv2.VideoCapture("ball_video1.mp4")

success,frame = Video.read()
x= []
y= []

count = 0

while (Video.isOpened()):

    if success == False:
        break

    gray_image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(gray_image, 200, 255, cv2.THRESH_BINARY_INV)

    # calculate moments of binary image
    M = cv2.moments(thresh)

    # calculate x,y coordinate of center
    cX = int(M["m10"] / M["m00"])
    cY = -int(M["m01"] / M["m00"])
    x.append(cX)
    y.append(cY)
    success, frame = Video.read()
    print("Read and Captured", success)
    count +=1

x2 = np.power(x,2)
X = np.column_stack([x2 , x, np.ones(len(x))])
Y = np.row_stack(y)
E = np.matmul(X.T, X)
F= np.linalg.inv(E)
G = np.matmul(F, X.T)
B = np.matmul(G, Y) #weights

print(B)

line = np.matmul(X,B)
```

```
print(x)
print(y)
plt.scatter(x,y)
plt.plot(x, line, 'g')
plt.show()
```

Plot_1:

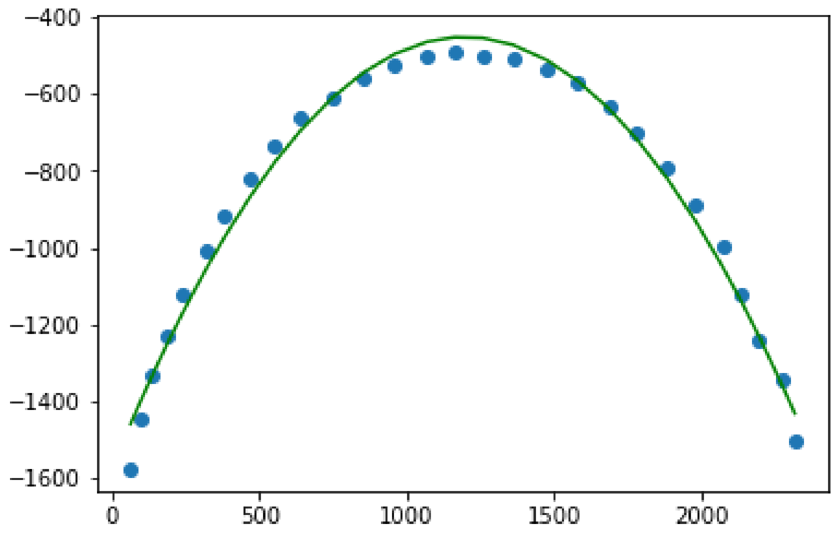


Fig 1.

Terminal window displaying the Results

```
runfile('Z:/ENPM673 Perception/Assignments/Problem 2.1.py', wdir='Z:/ENPM673Perception/Assignments')

The weights a,b,c are:
[[   -0.00077991]
 [    1.8698696 ]
 [-1571.67944213]]
```

Part 2: ballvideo2.mp4

```
 #Problem 2 part 2
import cv2
import matplotlib.pyplot as plt
import numpy as np
Video = cv2.VideoCapture("ball_video2.mp4")

success,frame = Video.read()
x= []
y= []

#Defining the parabola's Quadratic equation
#def func(params, x):
    #a,b,c = params
    #return a*x*x + b*x + c

#Defining the error function (Difference of fitted value and actual value)
#def error(params, x, y)
```

```
    #return func(params, x)-y

count = 0

while (Video.isOpened()):

    if success == False:
        break

    gray_image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # convert the grayscale image to binary image
    #blur = cv2.GaussianBlur(gray_image, (5, 5), cv2.BORDER_DEFAULT)
    ret, thresh = cv2.threshold(gray_image, 200, 255, cv2.THRESH_BINARY_INV)

    # calculate moments of binary image
    M = cv2.moments(thresh)

    # calculate x,y coordinate of center
    cX = int(M["m10"] / M["m00"])
    cY = -int(M["m01"] / M["m00"])
    x.append(cX)
    y.append(cY)
    success, frame = Video.read()
    print("Read and Captured", success)
    count +=1

x2 = np.power(x,2)
X = np.column_stack([x2 , x, np.ones(len(x))])
Y = np.row_stack(y)
E = np.matmul(X.T, X)
F= np.linalg.inv(E)
G = np.matmul(F, X.T)
B = np.matmul(G, Y) #weights

print(B)

line =  np.matmul(X,B)

print(x)
print(y)
plt.scatter(x,y)
plt.plot(x, line, 'g')
plt.show()
```

Plot_2:

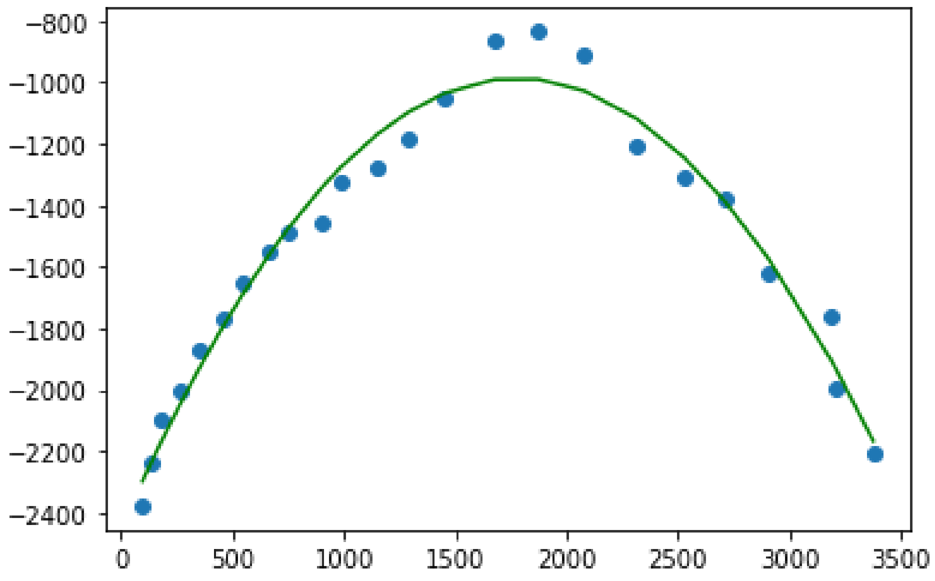


Fig 2.

Terminal Displaying the Results:

```
runfile('Z:/ENPM673 Perception/Assignments/Problem 2.2.py', wdir='Z:/ENPM673
Perception/Assignments')
The weights a,b,c are:
[[   -0.00046285]
 [    1.6468185 ]
 [-2450.74128375]]
```

Problem 3: [30 Points]

In the above problem, we used the least squares method to fit a curve. However, if the data is scattered, this might not be the best choice for curve fitting. In this problem, you are given data for health insurance costs based on the person's age. There are other fields as well, but you have to fit a line only for age and insurance cost data. The data is given in .csv file format and can be downloaded from here.

1. Compute the covariance matrix (from scratch) and find its eigenvalues and eigenvectors. Plot the eigenvectors on the same graph as the data. Refer to this article for better understanding. **[10]**
2. Fit a line to the data using linear least square method, total least square method and RANSAC. Plot the result for each method and explain drawbacks/advantages for each. **[15]**
3. Briefly explain all the steps of your solution and discuss which would be a better choice of outlier rejection technique for each case. **[5]**

ANSWER:

Part 1:

Conceptual understanding behind solving the problem

Variance measures the variation of a single random variable (like the height of a person in a population), whereas covariance is a measure of how much two random variables vary together (like the height of a person and the weight of a person in a population). The formula for variance is given by

$$\sigma_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where n is the number of samples (e.g. the number of people) and $\bar{x}$ is the mean of the random variable x (represented as a vector). The covariance $\sigma(x, y)$ of two random variables x and y is given by

$$\sigma(x, y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

with n samples. The variance σ_x^2 of a random variable x can be also expressed as the covariance with itself by $\sigma(x, x)$

Covariance Matrix: covariance matrix, which is a square matrix given by $C_{i,j} = \sigma(x_i, x_j)$

The Calculation of the Covariance matrix can be expressed as:

$$C = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})^T$$

Eigenvalues and Eigenvectors:

Eigenvalues are the special set of scalars associated with the system of linear equations that is used to transform an eigenvector. They can be calculated using the characteristic equation

$$Ax = \lambda x$$

Eigenvectors are the vectors (non-zero) that do not change the direction when any linear transformation is applied. It changes by only a scalar factor. In a brief, we can say, if A is a linear transformation from a vector space V and **x** is a vector in V, which is not a zero vector, then v is an eigenvector of A if A(X) is a scalar multiple of x, which is an eigenvector of A corresponding to eigenvalue, λ.

In this problem, we shall calculate the covariance matrix for the given data and then through that we plot the eigen vectors that give the directions of the given data.

Code snippet:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
data = pd.read_csv("datasetp3.csv")

x= data.age
y= data.charges
X= np.column_stack([x, y])

#To find covariances of a matrix
def cov(x,y):
    xbar, ybar = x.mean(), y.mean()
    return np.sum((x-xbar)*((y-ybar).T))/(len(x)-1)

#Covariance matrix
def cov_mat(X):
    Cmat= np.array([[cov(X[0],X[0]), cov(X[0], X[1])],
             [cov(X[1],X[0]), cov(X[1],X[1])]])
    return Cmat
#covi = np.cov(x,y)
eigval, eigvec = np.linalg.eig(cov_mat(X))

print('The Eigen values of the Matrix X:')
print(eigval)

print('The Corresponding Eigen vectors of X:')
print(eigvec)



O = [x.mean(),y.mean()]
s_eigvec = eigvec[:,0]
l_eigvec = eigvec[:,1]

#calculate Covariance matrix
print('The Covariance Matrix:')
print(cov_mat(X))

plt.scatter(x,y)
plt.quiver(*O, *s_eigvec, color=['r'], scale=6) #Eigenvector X
plt.quiver(*O, *l_eigvec, color=['b'], scale=6) #Eigenvector Y
plt.show
```

Plot:

Terminal window displaying the results:

```
runfile('Z:/ENPM673 Perception/Assignments/Problem 3 part 1.py', wdir='Z:/ENPM673
Perception/Assignments')

The Eigen values of the Matrix X:
[ 2.48286588e+08 -9.31322575e-10]
```

```
The Corresponding Eigen vectors of X:
[[ 0.98508111 -0.17209068]
 [ 0.17209068  0.98508111]]

The Covariance Matrix:
[[2.40933531e+08 4.20903556e+07]
 [4.20903556e+07 7.35305720e+06]]
```

Plot:

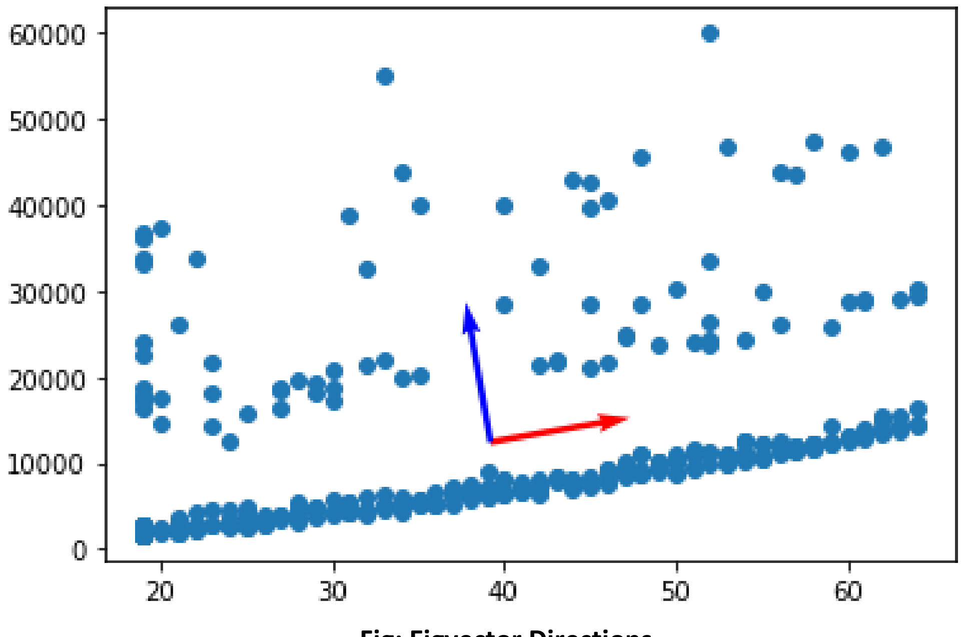


Fig: Eigvector Directions

Part 2 and 3:

Linear Least squares: (for this problem I have used the Ordinary least squares method)

Conceptual Understanding before solving the problem

Here, our goal is to fit a parametric model that describes the equation of the line to a series of points labeled as $(x_1, y_2), (x_2, y_2), \dots, (x_n, y_n)$. If we model the line as $y = mx + c$, our goal is to estimate the parameters m, c to minimize the value of E (error). E is the sum of residuals or how far off our estimate is for each point. We can interpret this as choosing a line to minimize the distance between the line and the observed data points.

We can formulate this easily as a least squares problem. $E = \sum_{i=0}^{n}(y_i - mx - c)$ describes the objective function we are trying to minimize. We can re-write this equation in the form of $E = ||Y - XB||^2$, where $\mathrm{B} = [\mathrm{m\ c}]^{\mathrm{T}}$ and $X = [x \ \ 1]$ and Y is the vector containing all y values of our data.

Our goal is to find the parameters m, c that minimizes E. These can be found by taking the derivative of E with respect to B and equating it to zero. $\frac{dE}{dB} = -2X^TY + 2X^TXB = 0$ Solving this equation allows to find B which can be expressed as $B = (X^TX)^{-1}X^TY$. This solution can be interpreted as the optimal (in the least squares sense) parameters a, b, c that minimize the residuals, or, equivalently, choosing a line that comes closest to best fitting most of the points.

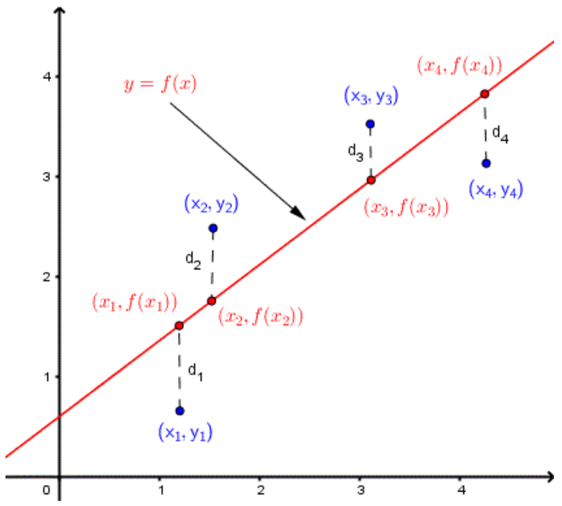


Fig 3: A generic LLS best fit model

Code snippet:

```
 #Librarures that are imported
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

#Reading data
data = pd.read_csv("datasetp3.csv")
x= data.age
y= data.charges
X= np.column_stack([x, y])


#Linear Least squares Implementation
A = np.column_stack([x, np.ones(len(x))]) #Matrix X
Y = np.row_stack(y) #Matrix Y
E = np.matmul(A.T, A)
F= np.linalg.inv(E)
G = np.matmul(F, A.T)
B = np.matmul(G, Y) #This is basically B = (((X^T*X)^-1) * X^T) * Y

#Calculating line equation
line =  np.matmul(A,B)
print('The values of m and c respectively:')
print(B)

plt.scatter(x,y) #Data plot
plt.plot(x, line, 'b-') #Linear Least Square fit
plt.show
```

Plot: (LLS)

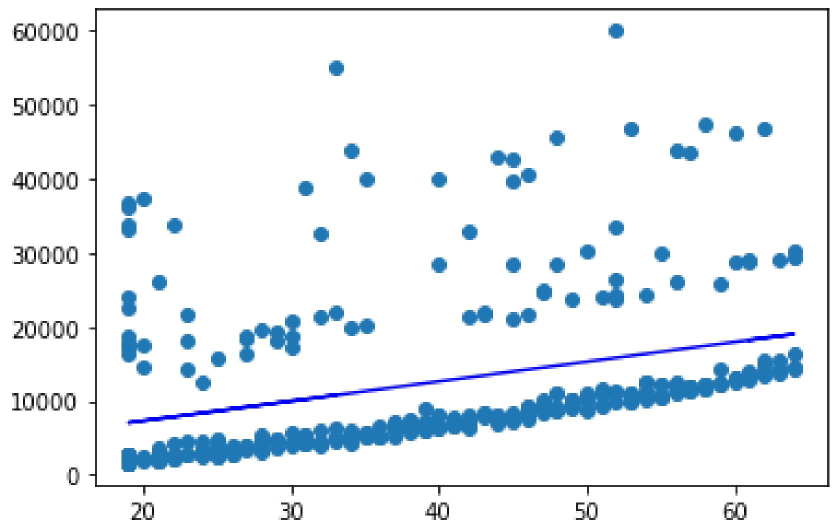


Fig 4.

Total Least Squares:

Conceptual Understanding before solving the problem

Total least squares is similar to OLS but instead of using vertical residuals (observed response value minus fitted response value), it uses diagonal residuals (shortest distant between observation point and fitted model). Here, our goal is to parametrize a model that describes the equation of the line to a series of points labeled as $(x_1, y_2), (x_2, y_2), \ldots, (x_n, y_n)$. If we model the line as $ax + by = d$, we happen to avoid the case of infinite slope. Here our goal is to estimate the parameters a, b, d to minimize the value of E (error). E is the sum of residuals or how far off our estimate is for each point. It can be shown that minimizing E with respect to a, b, d is equivalent to solving the homogenous system $Ah = 0$, where $h = [a\ b\ d]^T$and the matrix A collects all the coordinates of the data points. We can solve $Ah = 0$ using Singular Value Decomposition (SVD).

When we take the SVD of a matrix, we are separating the matrix into three pieces, U (directions of output), Σ (a diagonal matrix with scaling amounts for different directions), and V (directions of input).

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

In this case, input to the matrix means being multiplied by a vector on the right. We can interpret the SVD as, when we have an input to A that is in the direction of the nth column of V, it will get scaled by the nth value of Σ, and be output as the nth column of U. Thus, when the matrix of scaling factors Σ is sorted, the last column of V dictates the input direction of A that will have the least output. Therefore, that is the direction of input to A that will get the output closest to zero and is the vector that will minimize Ah. We minimize $||Ah||$ subject to $||h|| = 1$. h would be the last column of V.

Here $U = AA^T$, $V = A^T A$ and $\Sigma = diag(\sigma_1, \sigma_2 \ldots)$. Where $\sigma = \sqrt{\lambda}$ and λ is the eigen value of AA^T or $A^T A$ (both yield the same result).

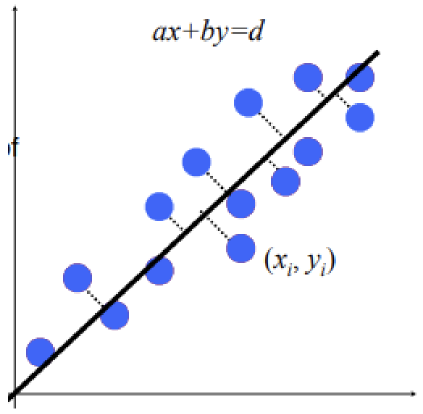


Fig 5. A TLS best fit model

In this case, I have first calculated the covariance matrix of the data which simplified my A matrix and then after doing that I calculated the SVD of the A matrix and thereby found the parameters a, b, d which I then used to substitute and calculate the line equation.

Code Snippet:

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

data = pd.read_csv("datasetp3.csv")

x= data.age
y= data.charges

x = np.row_stack(x)
y = np.row_stack(y)
X = [x,y]

xbar = x.mean()
ybar = y.mean()

def cov(x,y):
    xbar, ybar = x.mean(), y.mean()
    return np.sum((x-xbar)*((y-ybar).T))/(len(x)-1)

def cov_mat(X):
    Cmat= np.array([[cov(X[0],X[0]), cov(X[0], X[1])],
             [cov(X[1],X[0]), cov(X[1],X[1])]])
    return Cmat

A = cov_mat(X)

def calc_SVD(A):
    AT= A.T
    ATA= AT.dot(A)
    eigval_U, eigvec_U = np.linalg.eig(ATA)
    sort_eigval_U =  eigval_U.argsort()[::-1] #sorted eigenvalues of eigval_U
    U = eigvec_U[:, sort_eigval_U]
    H = U[:,1]
    return H

B = calc_SVD(A)
d = B[0]*xbar + B[1]*ybar
line = (d - (B[0]*x))/B[1]

plt.scatter(x,y) #Data plot
plt.plot(x, line, 'y-')
plt.show
```

Plot: (TLS)

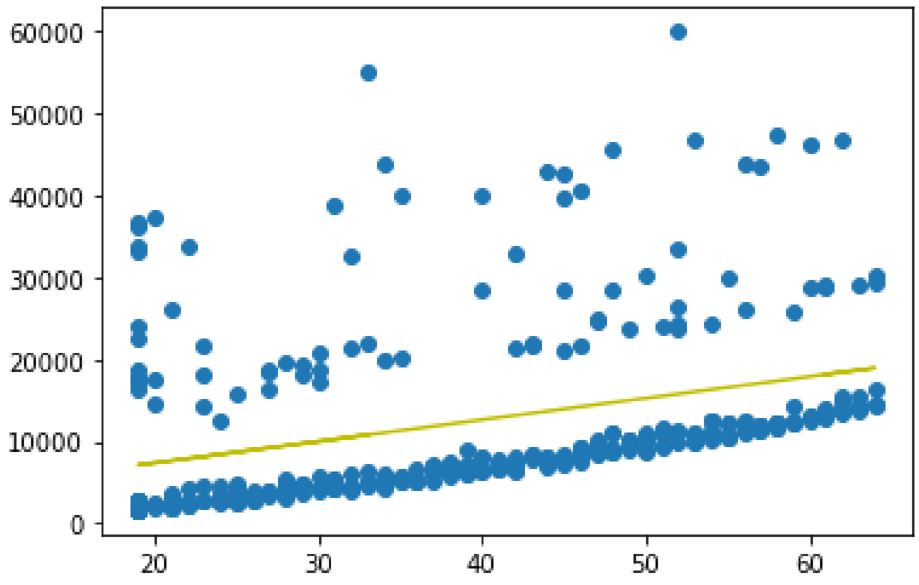


Fig 6:

RANSAC:

Random sample consensus, or RANSAC, is an iterative method for estimating a mathematical model from a data set that contains outliers. The RANSAC algorithm works by identifying the outliers in a data set and estimating the desired model using data that does not contain outliers. RANSAC is designed to be robust to outliers and missing data and follows two major assumptions:

1. Noisy data points will not vote consistently for any single model ("few" outliers)
2. There are enough data points to agree on a good model ("few" missing data)

We can formulate this problem as follows:
We want to find the best partition of points in the inlier set P and outlier set O such that we keep as many inlier points as necessary. So, to do that we give a threshold of a certain value

above and below the best fit to encompass all the points in the threshold range and consider them as inliers. The threshold is nothing but the tolerance limit we implement to get inliers.

Three major steps of doing ransac:

- We select a random sample of the minimum required size to fit the model. In this case, a line is determined by at least two points. Thus, to fit a model of a line, we select two points at random which constitutes the random sample.
- We calculate a model from this sample set. Using the two points that we have already randomly sampled; we calculate the line that these two represent.
- We see how much of our entire set of 2D points agrees with this model up to a threshold δ.
- We repeat these steps until we find the parametric model that fits the greatest number of inliers in the threshold.

The number of iterations, N, is chosen high enough to ensure that the probability p (usually set to 0.99) that at least one of the sets of random samples does not include an outlier. Let u represent the probability that any selected data point is an inlier and $v = 1 - u$ is the probability of observing an outlier. Now N iterations of the minimum number of points (m) are required. Where,

$$1 - p = ((1 - u)^m)^N$$

Now after doing some manipulation, we can write the equation in the manner of getting N

$$N = \frac{\log(1 - p)}{\log(1 - (1 - v)^m)}$$

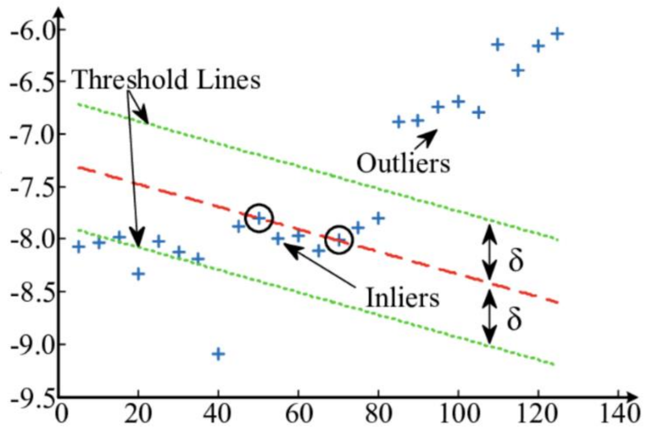


Fig 7: Line fitting using RANSAC

Code snippet:

```
#RANSAC
import random
import math
import matplotlib.pyplot as plt
import pandas as pd
```

```
#Reading the data
data = pd.read_csv("datasetp3.csv")

x= data.age
y= data.charges

#Function for selecting two random points from the sample
def randpts(i, data):
    x1, y1 = random.choice(list(zip(x,y)))
    x2, y2 = random.choice(list(zip(x,y)))

    try:
        m = (y2-y1)/(x2-x1) #slope of line
    except:
        m = 0
        return 0, 0, 0

    C = y1 - (m*x1)

    #Now we have to find the perpendicular distance between line and point
    # ax+by+c/(a^2 + b^2)^0.5

    a = -m #slope
    b = 1
    c = -C #Y-intercept

    #Defining a threshold to separate inliers and outliers
    Thresh = 2

    inlier_count = 0
    x_inliers = []
    y_inliers = []
    for d in list(zip(x,y)):
        i = d[0]
        j= d[1]
        dist = abs(a*i + b*j + c)/math.sqrt(a**2 + b**2)
        if dist < Thresh:
            x_inliers.append(x)
            y_inliers.append(y)
            inlier_count += 1

    return m, C, inlier_count

if __name__ == "__main__":

    fin_inlier_count = 0
    fin_m = []
    fin_c = []

    for i in range(100):

        m, C, inlier_count = randpts(i, data)

        if inlier_count > fin_inlier_count:
            fin_inlier_count = inlier_count #Most number of inliers possible for the given threshold
value
            fin_m = m #final slope
            fin_c = C #final intercept


    plt.scatter(x,y)
    xmin = min(x)
    xmax = max(x)

    ymin = fin_m*xmin + fin_c
    ymax = fin_m*xmax + fin_c

    plt.plot([xmin, xmax], [ymin, ymax], 'r-')
```

Plot: (RANSAC)

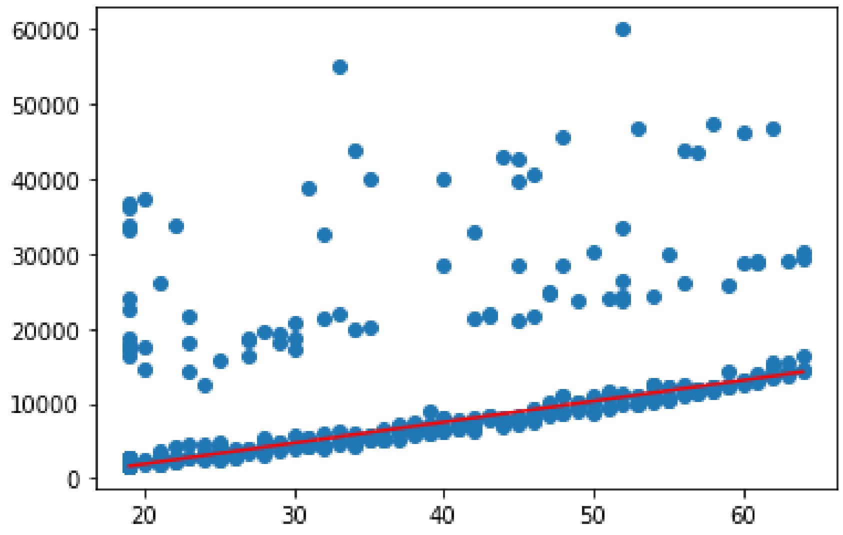


Fig 8

Part 2 & 3: (Advantages/Disadvantages and which method is more appropriate)

The conceptual understanding behind each of the methods is clearly stated above with the process, so in this section I will be writing about the advantages and limitations of each process, if they have any, and conclude what I think is a better method to use for line fitting a given data set.

Linear Least Squares:

This method has earned its place as the primary tool for process modeling because of its effectiveness and completeness. Though there are types of data that are better described by functions that are nonlinear in the parameters, many processes in science and engineering are well-described by linear models. This is because either the processes are inherently linear or because, over short ranges, any process can be well-approximated by a linear model. Practically speaking, linear least squares method makes very efficient use of the data. Good results can be obtained with relatively small data sets. The theory associated with linear least squares is well-understood and allows for construction of different types of easily interpretable statistical intervals for predictions, calibrations, and optimizations. These statistical intervals can then be used to give clear answers to scientific and engineering questions.

The main disadvantages of linear least squares are limitations in the shapes that linear models can assume over long ranges, possibly poor extrapolation properties, and sensitivity to outliers. It is very difficult to predict curves for linear models with nonlinear parameters. There is a case where if the parametrized model has infinite slope, this method does not work, hence completely failing for vertical lines. The model is not robust in that case. While the method of least squares often gives optimal estimates of the unknown parameters, it is very sensitive to the presence of unusual data points in the data used to fit a model. One or two outliers can sometimes seriously skew the results of a least squares analysis resulting in erroneous results.

Total Least squares:

This method is very similar to Linear Least Squares method, with a small change to the fitting process. The TLS method calculates the perpendicular distance of the point from the line whereas the Linear Least squares method calculates the distance between its own y coordinate and the y coordinate that touches the line when vertically dropped to the line. Thus, the robustness of the Total Least squared method is better than Linear Least squares method, it does not fail when the slope is infinite. The fitting accuracy of the Total Least Squares method is higher compared to Linear Least Squares.

RANSAC:

RANSAC is an easily implementable method to estimate a model that often works well in practice. Unlike conventional sampling techniques that use as much of the data as possible to obtain an initial solution and then proceed to prune outliers, RANSAC uses the smallest set possible and proceeds to enlarge this set with consistent data points

However, there are many parameters to tune; it may take a long time to get to the accuracy that you need and requires the inlier to outlier ratio to be reasonable. Computational time grows quickly with fraction of outliers and number of parameters in such cases.

RANSAC is robust to outliers, but empirical studies show that RANSAC doesn’t work well if the inlier/outlier ratio is greater than 50%. RANSAC is also not known for getting multiple fits of a data.

Conclusion: ***The appropriate choice of fitting that I would choose is RANSAC*** because this data has a lot of outliers and LLS and TLS are not robust enough and the fitting is skewed because of that. Whereas when we compare the inlier and outlier count, the ratio seems to be way above 50% which makes RANSAC a great parametric model to fit this data.

Problem 4: [20 Points]

The concept of homography in Computer Vision is used to understand, explain and study visual perspective, and specifically, the difference in appearance of two plane objects viewed from different points of view. This concept will be taught in more detail in the coming lectures. For now, you just need to know that given 4 corresponding points on the two different planes, the homography between them is computed using the following system of equations Ax = 0, where A is given by

$$A = \begin{bmatrix} -x1 & -y1 & -1 & 0 & 0 & 0 & x1*xp1 & y1*xp1 & xp1 \\ 0 & 0 & 0 & -x1 & -y1 & -1 & x1*yp1 & y1*yp1 & yp1 \\ -x2 & -y2 & -1 & 0 & 0 & 0 & x2*xp2 & y2*xp2 & xp2 \\ 0 & 0 & 0 & -x2 & -y2 & -1 & x2*yp2 & y2*yp2 & yp2 \\ -x3 & -y3 & -1 & 0 & 0 & 0 & x3*xp3 & y3*xp3 & xp3 \\ 0 & 0 & 0 & -x3 & -y3 & -1 & x3*yp3 & y3*yp3 & yp3 \\ -x4 & -y4 & -1 & 0 & 0 & 0 & x4*xp4 & y4*xp4 & xp4 \\ 0 & 0 & 0 & -x4 & -y4 & -1 & x4*yp4 & y4*yp4 & yp4 \end{bmatrix}, x = \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \\ H_{33} \end{bmatrix}$$

For the given point correspondences,

	x	y	xp	yp
1	5	5	100	100
2	150	5	200	80
3	150	150	220	80
4	5	150	100	200

Find the homography matrix:

$$H = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix}$$

1. Show mathematically how you will compute the Singular Value Decomposition (SVD) for the matrix A. **[10]**
2. Write python code to compute the SVD. **[10]**

ANSWER:

Singular Value decomposition is a generalized version of eigen decomposition where we factorize an mxn matrix A of rank r as $A = U\Sigma V^T$

Here

- columns of U are orthogonal eigenvectors of AA^T. It is an mxm square orthonormal basis function
- columns of V are orthogonal eigenvectors of A^TA. It is an nxn square orthornormal basis function.
- $\Sigma = diag(\sigma_1, \sigma_2 \dots)$. Where $\sigma = \sqrt{\lambda}$ and λ is the eigen value of AA^T or A^TA (both yield the same result). Σ is an m×n diagonal rectangular matrix which acts as a scaling matrix.

When we take the SVD of a matrix, we are separating the matrix into three pieces, U (directions of output), Σ (a diagonal matrix with scaling amounts for different directions), and V (directions of input). Note that to implement SVD A has to be a positive semi definite matrix, meaning that $A \geq 0$ or all eigenvalues have to be non negative.

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

In this case, input to the matrix means being multiplied by a vector on the right. We can interpret the SVD as, when we have an input to A that is in the direction of the nth column of V, it will get scaled by the nth value of Σ, and be output as the nth column of U. Thus, when the matrix of scaling factors Σ is sorted, the last column of V dictates the input direction of A that will have the least output. Therefore, that is the direction of input to A that will get the output closest to zero and is the vector that will minimize Ah. We minimize $||Ah||$ subject to $||h|| = 1$. h would be the last column of V^T .

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

$$A = (AA^T)(\Sigma)(A^TA)^T$$

Algorithm for applying SVD to a homogeneous system of equations:

- Substitute the given x, y, xp, yp values in the matrix A
- Calculate AA^T and find its eigen vectors.
- Sort the eigenvectors which will then be the columns of U
- Calculate A^TA and find its eigen vectors and sort them as well
- These eigen vectors then be the columns of V
- Calculate the Σ matrix, to do that find σ_i which is $\sigma = \sqrt{\lambda}$
- $\Sigma = diag(\sigma_1, \sigma_2 \dots)$ substitute the σ_i in the matrix.
- To find the elements of the homography matrix, all we must do is single out the last column of V^T which corresponds to the least eigen value and is hence the least eigen vector.

Here we use the SVD method to calculate the homography matrix fo $Ax = 0$ because $Ax = 0$ is an underdetermined homogeneous system of equations.

For a homogeneous system of equations, there is a unique trivial solution $x = 0$. We are trying to avoid getting a trivial solution; therefore, we need to impose a constraint $||x|| = 1$ to avoid it. Now we have to find x such that $||Ax||^2$ is minimized. $||Ax||^2 = xAA^Tx$

Code Snippet:

```
import numpy as np

x1, x2, x3, x4 = 5,150,150,5
y1, y2, y3, y4 = 5,5,150,150
xp1, xp2, xp3, xp4 = 100, 200, 220, 100
yp1, yp2, yp3, yp4 = 100, 80, 80, 200

A =np.array( [[-x1,-y1,-1,0,0,0,x1*xp1,y1*xp1,xp1],
             [0,0,0,-x1,-y1,-1,x1*yp1,y1*yp1,yp1],
             [-x2,-y2,-1,0,0,0,x2*xp2,y2*xp2,xp2],
             [0,0,0,-x2,-y2,-1,x2*yp2,y2*yp2,yp2],
             [-x3,-y3,-1,0,0,0,x3*xp3,y3*xp3,xp3],
             [0,0,0,-x3,-y3,-1,x3*yp3,y3*yp3,yp3],
             [-x4,-y4,-1,0,0,0,x4*xp4,y4*xp4,xp4],
             [0,0,0,-x4,-y4,-1,x4*yp4,y4*yp4,yp4]])

print(A)

def calc_SVD(A):
    AT= A.T
    AAT= A.dot(AT)
    ATA= AT.dot(A)
    eigval_U, eigvec_U = np.linalg.eig(AAT)
    eigval_V, eigvec_V = np.linalg.eig(ATA)
    sort_eigval_U =  eigval_U.argsort()[::-1] #sorted eigenvalues of eigval_U
    sort_eigval_V =  eigval_V.argsort()[::-1] #sorted eigenvalues of eigval_V
    U = eigvec_U[:, sort_eigval_U]
    V = eigvec_V[:, sort_eigval_V]
    VT = V.T
    sigma =  np.sqrt(eigval_U)
    E = np.diag(sigma)
    H = VT[:,8]
    H = np.reshape(H,(3,3))
    return VT, U, E, H


#Return decomposed values of A
VT, U, E, H = calc_SVD(A)

np.set_printoptions(suppress = True)
print("V Transpose =", VT)
print("U=", U)
print("Sigma=", E)
print("Homography Matrix=",H)


#print(eigval_U)
#print(eigvec_U)
```

Terminal window displaying the results:

```
 runfile('Z:/ENPM673 Perception/Assignments/Problem 4.py', wdir='Z:/ENPM673 Perception/Assignments')

The A Matrix is
[[   -5    -5    -1     0     0     0   500   500   100]
 [    0     0     0    -5    -5    -1   500   500   100]
 [ -150    -5    -1     0     0     0 30000  1000   200]
 [    0     0     0  -150    -5    -1 12000   400    80]
 [ -150  -150    -1     0     0     0 33000 33000   220]
 [    0     0     0  -150  -150    -1 12000 12000    80]
 [   -5  -150    -1     0     0     0   500 15000   100]
 [    0     0     0    -5  -150    -1  1000 30000   200]]
```

```
 V Transpose =
[[ 0.37376641  0.31698352  0.00300085  0.14598349  0.21804997  0.00185801
-0.02813455 -0.02889245 -0.8302556 ]
 [-0.38540978 -0.33481509 -0.0031922   0.57961717  0.63425663  0.00553984
   0.00055821 -0.00028679 -0.03285329]
 [-0.43658932  0.17171976 -0.00076267 -0.72733205  0.48318723 -0.00019319
   0.00091061 -0.00076944 -0.13197819]
 [-0.31426555  0.8576514   0.00130812  0.32992417 -0.02745482  0.00004854
   0.00020557 -0.00050437  0.23678156]
 [ 0.64970575  0.14894284 -0.00298254 -0.069421    0.56201275 -0.0064375
  -0.00027001 -0.00021289  0.48473746]
 [ 0.00350322  0.00337763 -0.81364209 -0.00255301 -0.00292733  0.58133276
  -0.00000037 -0.00000201  0.00000917]
 [ 0.00432192  0.00030943  0.58133996 -0.00319797  0.00183738  0.81361944
   0.00000005  0.00000074  0.00590597]
 [-0.01534412 -0.01307963 -0.00011932 -0.00589392 -0.00883095 -0.00007233
  -0.69550285 -0.71739059  0.03327553]
 [-0.00097023  0.00039597 -0.0000035  -0.00036233  0.00089679  0.00000352
  -0.71797143  0.69607111 -0.00000708]]
```

```
 U=
[[ 0.0171751   0.00105765  0.17278434 -0.01719854  0.62536604 -0.262681
  -0.71373613 -0.00710167]
 [ 0.01717511  0.00105712  0.13409755  0.00303011  0.64912154 -0.26620293
   0.69959943 -0.0071011 ]
 [ 0.51469903  0.6770349   0.42426639  0.17552678 -0.12628942 -0.02104738
   0.01117163 -0.2222166 ]
 [ 0.20588268  0.27081369 -0.42153477 -0.41033528  0.35391304  0.63635731
  -0.01002825 -0.08888214]
 [ 0.38154617  0.01329261  0.00000767 -0.00000291 -0.00000717  0.0000011
   0.          0.9242542 ]
 [ 0.41180511  0.02534806 -0.67325891 -0.02883031 -0.09261359 -0.58115859
  -0.01790773 -0.17035849]
 [ 0.27457279 -0.30576469  0.3689511  -0.79523062 -0.16730485 -0.14888341
   0.02392208 -0.108957   ]
 [ 0.54914126 -0.61153102  0.07118583  0.40900783  0.09892877  0.30794398
  -0.00480104 -0.21789772]]
```

```
 Sigma=
[[          nan     0.              0.              0.
      0.            0.              0.              0.        ]
 [    0.        46716.0060638      0.              0.
      0.            0.              0.              0.        ]
 [    0.            0.          31804.12647532     0.
      0.            0.              0.              0.        ]
 [    0.            0.              0.            221.61041666
      0.            0.              0.              0.        ]
 [    0.            0.              0.              0.
      2.13798962    0.              0.              0.        ]
 [    0.            0.              0.              0.
      0.           47.22988149     0.              0.        ]
 [    0.              0.              0.              0.
      0.              0.            136.31872867      0.        ]
 [    0.              0.              0.              0.
      0.              0.              0.            147.08420569]]
```

```
 Homography Matrix=
[[-0.8302556  -0.03285329 -0.13197819]
 [ 0.23678156  0.48473746  0.00000917]
 [ 0.00590597  0.03327553 -0.00000708]]
```