Homework-1 Report
for
Perception for Autonomous Robots (ENPM667)

Aditi Ramadwar UID: 117093575

Problem 1:

Sensor shape = 14mm
Focal length = f = 25mm
Resolution of camera = 5MP

1. The camera sensor is square shaped so the Field of View(FOV)(φ) will be same in horizontal and vertical directions. Using the formula,

$$\varphi = 2 \times \tan^{-1}\frac{d}{2f}$$
$$\varphi = 2 \times \tan^{-1}\frac{14}{2x25}$$
$$\varphi = 2 \times \tan^{-1}(0.28)$$
$$\varphi = 2 \times 15.6^{o} = 31.28^{o}$$

2. Object width = 5cm = 500mm
   Object distance = 20m = 20000mm

$$\frac{width\ of\ image}{Object\ width} = \frac{focal\ length}{object\ distance}$$
$$\frac{width\ of\ image}{500} = \frac{25}{20000}$$
$$width\ of\ image = \frac{25 * 500}{20000} = 0.0625$$

   Number of pixels that the image will occupy will be,

$$p = \frac{area\ of\ image}{area\ of\ sensor} * Resolution\ of\ camera$$
$$p = \frac{0.0625^2}{14^2} * 5 * 10^6 = 99.64 \approx 100\ pixels$$

Standard Least Square

The standard least square method is used to determine the position of the trend line, the trend line is the best fitting curve for the respective given data.

Consider a set of **n** values $(x_1, y_1), \dots, (x_n, y_n)$. Suppose we have to find **linear** relationship in the form $y = a + bx$ among the above set of x and y values:

The difference between observed and estimated values of y is called residual and is given by,

$$R_i = y_i - (a + bx_i) \quad (1)$$

In the least square method, we find a and b in such a way that $\sum R_i^2$ is minimum.

Using equation (1), let

$$T = \sum R_i^2 = \sum_i^n (y_i - (a + bx_i))^2 \quad (2)$$

The condition for T to be minimum is that $\frac{\partial T}{\partial a} = 0$ and $\frac{\partial T}{\partial b} = 0$ i.e.,

$$\frac{\partial T}{\partial a} = 2\sum_i^n (y_i - (a + bx_i))(-1) = 0 \quad (3.1)$$
$$\frac{\partial T}{\partial b} = 2\sum_i^n (y_i - (a + bx_i))(-x_i) = 0 \quad (3.2)$$

Rearranging equations (3.1) and (3.2), we get,

$$\sum_i^n y_i - \sum_i^n a - \sum_i^n bx_i = 0 \quad (4.1)$$

$$-\sum_{i}^{n} x_i y_i - \sum_{i}^{n} a x_i - \sum_{i}^{n} b x_i^2 = 0 \tag{4.2}$$

Since coefficients a and b are constants, the equation (4.1) and (4.2) can be rewritten and rearranged as,

$$\sum_{i}^{n} y_i = na + b\sum_{i}^{n} x_i \tag{5.1}$$

$$\sum_{i}^{n} x_i y_i = a\sum_{i}^{n} x_i + b\sum_{i}^{n} x_i^2 \tag{5.2}$$

Equations (5.1) and (5.2) are called normal equations. By solving these, we get a and b. Line of best fit can now be formed with these values obtained.

We can make a general set of equations that can be used for curve fitting using Standard Least square method

Let $y = a_1 + a_2 x + a_3 x_3 + \cdots + a_m a_{m-1}$ be the curve of best fit for the data set $(x_1, y_1), \ldots, (x_n, y_n)$

Using the Least Square Method, we can prove that the normal equations are:

$$\begin{aligned}
\sum_{i}^{n} y_i &= na_1 + a_2\sum_{i}^{n} x_i + a_3\sum_{i}^{n} x_i^2 + \cdots + a_m\sum_{i}^{n} x_i^{m-1} \\
\sum_{i}^{n} x_i y_i &= a_1\sum_{i}^{n} x_i + a_2\sum_{i}^{n} x_i^2 + a_3\sum_{i}^{n} x_i^3 + \cdots + a_m\sum_{i}^{n} x_i^m \\
&\vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
\sum_{i}^{n} x_i^{m-1} y_i &= a_1\sum_{i}^{n} x_i^{m-1} + a_2\sum_{i}^{n} x_i^m + \cdots + a_m\sum_{i}^{n} x_i^{2m-2}
\end{aligned} \tag{6}$$

Using these equations and solving for the a_i coefficients will give us the best fitting curve.

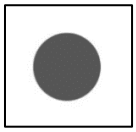
Problem 2:

The aim is to use Standard Least Squares to fit curves to datapoints of the given videos.
To plot the best fit curve using standard least squares to the given videos, I divided the problem in two different segments. One file (*2_data_points.py*) is for extracting the data points of the ball in each video and second file (*2_LS_plot_curve.py*) is for implementing the Standard Least Square method for fitting the curve based on the data points.

Step 1: Save the data points

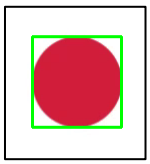
Using the OpenCV library, open the video for collecting the data points. The variable *vid_name* is used to select the video that we need to process.

i. Each frame is converted to a grayscale image using the OpenCV cvtColor()
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)



ii. Since the background of the ball is white, a mask can be used to separate the ball pixels from the background pixels. The pixels above a threshold, that is those that are not white are separated using the numpy.where() function. This function gives the index of each pixel in the mask.
b = where(gray<200)

iii. Using the indices, we can extract the top and bottom pixel of the ball at each frame by getting the minimum and maximum indexed x-y pixels.
y_max = max(b[0])
y_min = min(b[0])
x_max = max(b[1])
x_min = min(b[1])



iv. *Since* we now know where the ball is at each frame, we can get the centroid of it and store those points(x_mid,y_mid) in a dataset for further evaluation.
x_mid=int((x_min+x_max)/2)
y_mid=int((y_min+y_max)/2)

Step 2: Get the best fitting curve

Since we are fitting a parabola with three points, we establish the relationship between variables in the form of the equation $y = a_1 + a_2x + a_3x^2$.

So according to equation (6), where m=3, the equations we get for getting a best fitting parabolic curve are,

$$\sum_{i}^{n} y_{\mathrm{i}} = na_1 + a_2 \sum_{i}^{n} x_{\mathrm{i}} + a_3 \sum_{i}^{n} x_i^2$$

$$\sum_i^n x_i y_i = a_1 \sum_i^n x_i + a_2 \sum_i^n x_i^2 + a_3 \sum_i^n x_i^3$$
$$\sum_i^n x_i^2 y_i = a_1 \sum_i^n x_i^2 + a_2 \sum_i^n x_i^3 + a_3 \sum_i^n x_i^4$$

These equations can be rearranged in the form of AX=B, where,

$$A = \begin{bmatrix} n, & \sum_i^n x_i, & \sum_i^n x_i^2 \\ \sum_i^n x_i, & \sum_i^n x_i^2, & \sum_i^n x_i^3 \\ \sum_i^n x_i^2, & \sum_i^n x_i^3, & \sum_i^n x_i^4 \end{bmatrix}, X = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, B = \begin{bmatrix} y_i \\ x_i y_i \\ x_i^2 y_i \end{bmatrix}$$

To solve for X, we take inverse of A using the numpy.linalg.inv() function and multiply it by B,

A_inv = np.linalg.inv(A)

*# X = A^-1 * B*

X = np.dot(A_inv,B)

*X=A^-1 * B*

After obtaining $a_1, a_2 \; and \; a_3$, we can use the parabola equation to plot the best fitting curve

Results:

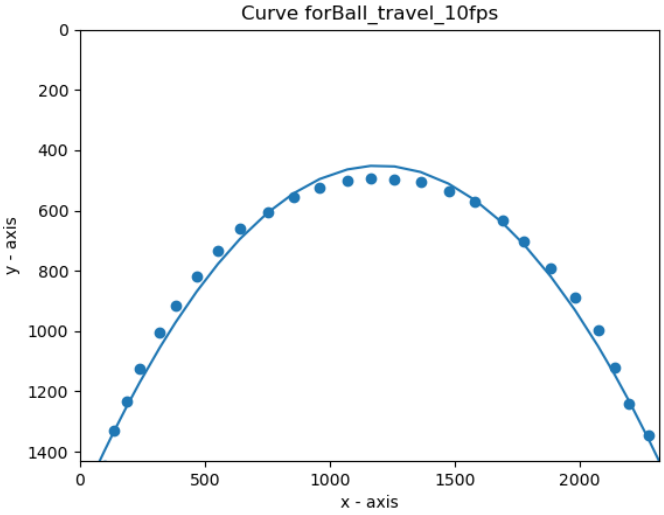


Figure (1)

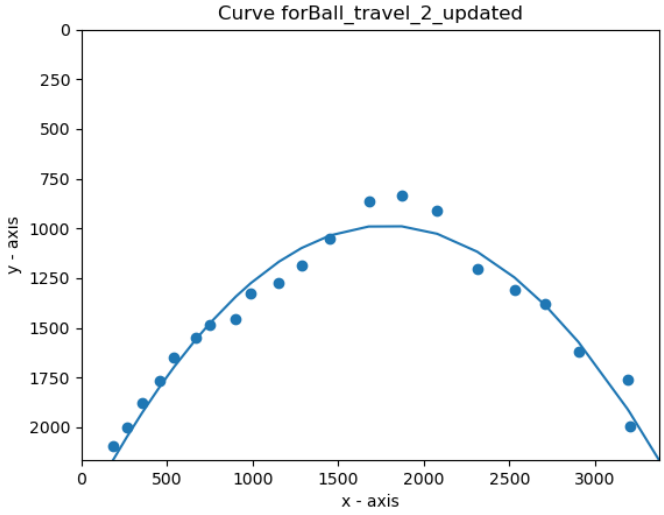
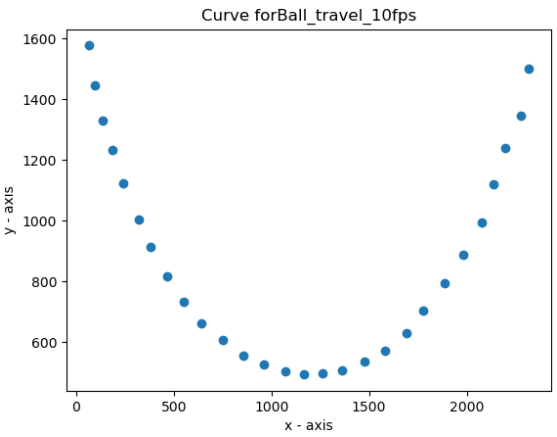


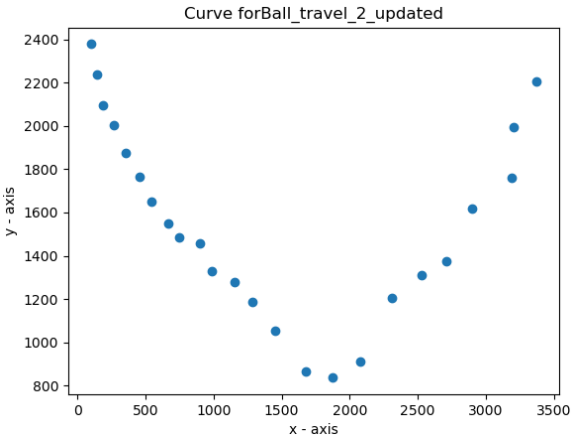
Figure (2)

- The figure (1) shows the best fitting curve for the near perfect sensor tracking of the ball
- The figure (2) shows the best fitting curve for the tracking of the ball using a faulty sensor.

Interesting Problem:

While plotting the data points (x, y) using matplotlib (*plt.scatter(x, y)*), the data points were inverted and an inverted parabola was being plotted.





After reading up on it more, I figured out that the origin on an image is slightly different than our standard coordinate system Figure (3). While the x-coordinate still originates from the left side of the image, the y-coordinate originates from the top left corner rather than the bottom left corner Figure (4).

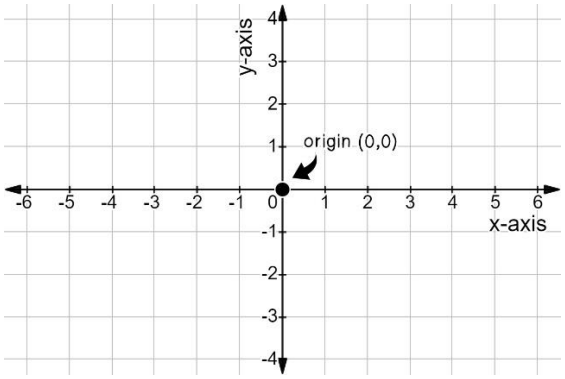


Figure (3)

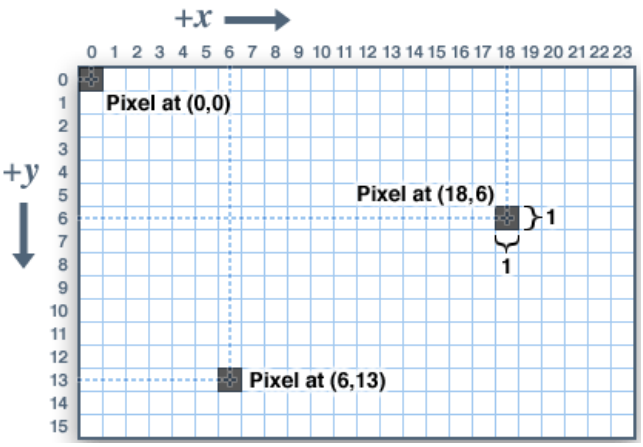
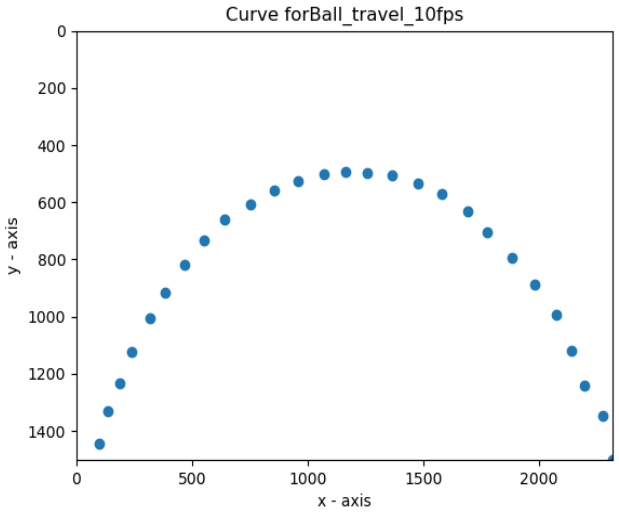


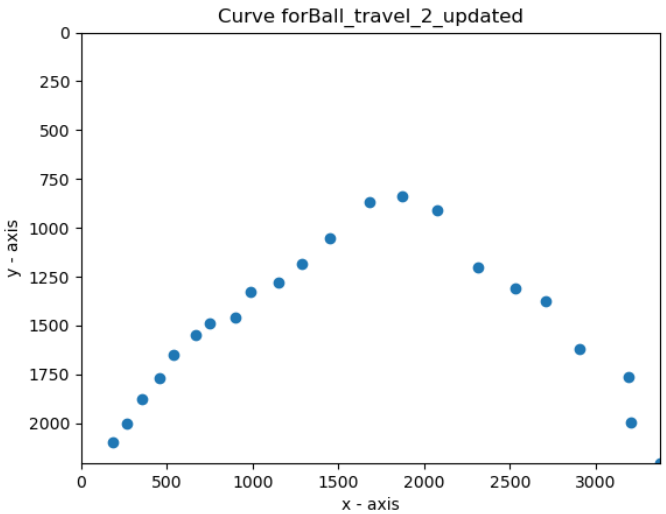
Figure (4)

So, to rectify this misunderstanding, I added limits on the plot where I flipped the origin of y-axis:

plt.ylim(y_new[n-1], 0)

plt.xlim(0, x[n-1])





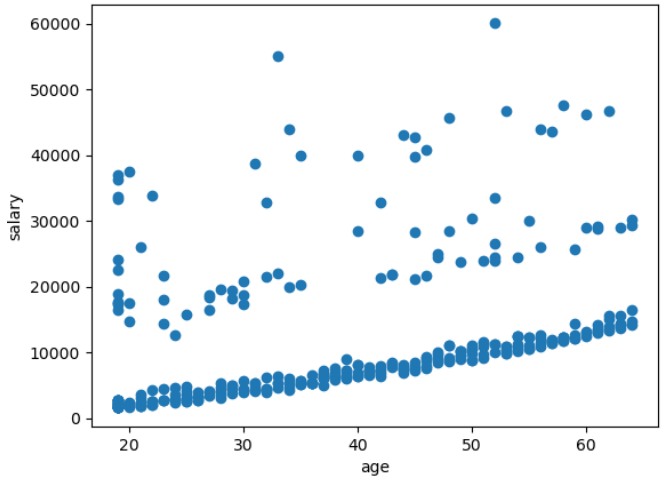
How to run the code:

1. To collect data points: *python3 2_data_points.py*
2. To get the best fitting curves: *python3 2_LS_plot_curve.py*

Problem 3:

The data from the .csv file is being read using the csv library.

```
csvreader = csv.reader(file)
age_og=[]
salary_og=[]
for row in csvreader:
    age_og=np.append(age_og,int(row[0]))
    salary_og=np.append(salary_og,float(row[6]))
file.close()
```



1. To compute the covariance matrix and plot the eigenvectors on the data:

A. Compute covariance matrix from scratch:

$$\text{Covariance matrix} = \begin{bmatrix} var(x), & cov(x,y) \\ cov(x,y), & var(y) \end{bmatrix} \tag{3.1}$$

Where,

$$var(x) = \frac{\sum_i^n (x_i - mean(x))^2}{n-1} \tag{3.2}$$

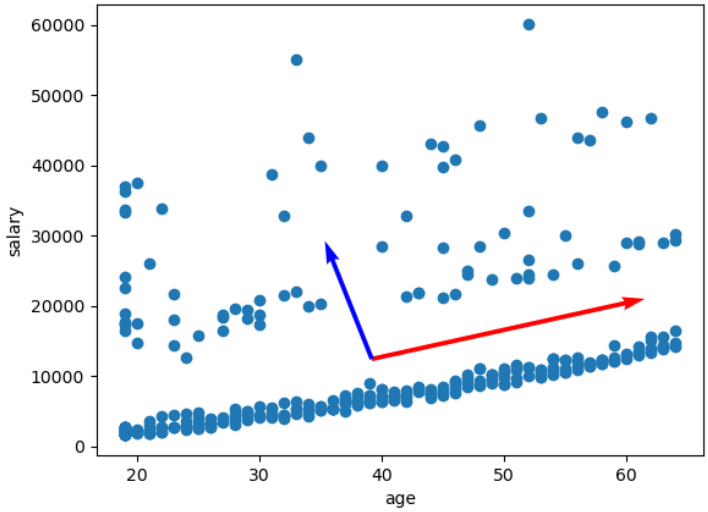
$$cov(x,y) = \frac{\sum_i^n \left((x_i - mean(x)) * (y_i - mean(y)) \right)}{n-1} \tag{3.3}$$

Covariance Matrix obtained from implementation:
[[0.09750555848662844, 0.019989873168443955],
[0.019989873168443955, 0.03594573020774471]]

B. The eigenvalues and eigenvectors of the Covariance Matrix were obtained by using the numpy.linalg.eig() function.

Eigen Values:
[0.10342712 0.03002417]
Eigen Vectors:
[[0.95881596 -0.28402809]
[0.28402809 0.95881596]]

C. Plot eigen vectors- PCA



Interesting problem:

To plot the eigen vectors properly, normalization of data was required since the two data sets given are not in the same range

2. Curve fitting

A. Linear least square method

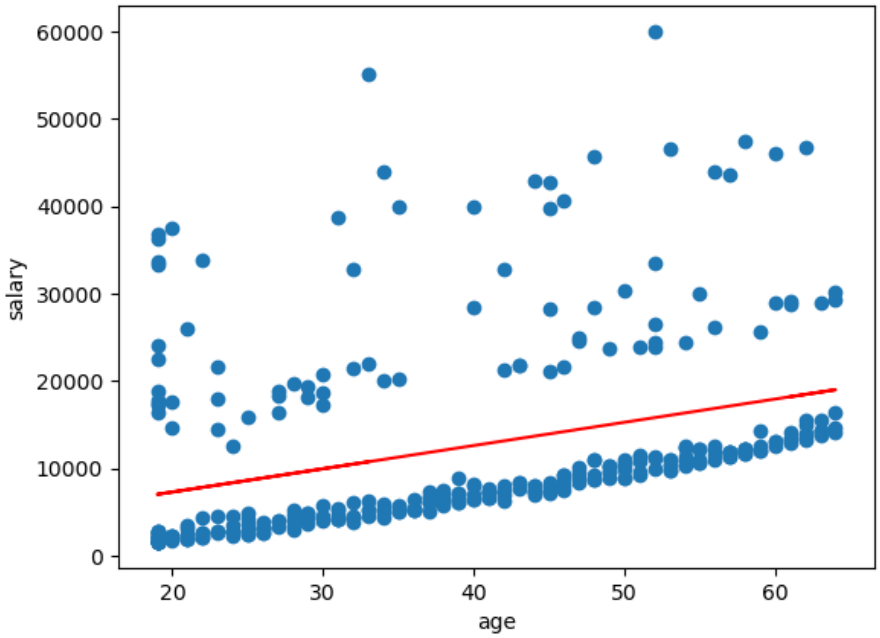
Since it is a linear curve, the equation of line y = ax+b can be used. We can use the equations (5.1) and (5.2) for carrying out the least square method.

We need to rearrange those equations to solve for a and b as,

$$a = \frac{n\sum(xy) - \sum x \sum y}{n\sum x^2 - \sum x^2}$$

$$b = \sum y - a \sum x$$

Using the values of a and b, we can now plot a line that best fits the data using standard least square method.



Advantages:

- The Standard Linear least square is very easy to comprehend as well as to implement. Many processes and datasets can be simply described by a linear model on which this method can easily be implemented.
- The estimated coefficients obtained from LS are the optimal estimates.

- Good results can be obtained with relatively small data sets.

Disadvantage:

- It is extra sensitive to outliers in the dataset which can generate major skewing of results
- For nonlinear processes, it becomes difficult to come up with a linear model that will give out the best fitting curve.

Problems: No problems arose

B. Total Least Square Method

While Standard Least square checks the error just in the y-axis, Total Least Square Method takes the unit norm of the points to the line to account for their error using a*x + b*y = d.

i. Normalized the given data in the range on (0,1)

ii. Arranged the data as

$$U = \begin{bmatrix} x_1 - mean(x), & y_1 - mean(y) \\ \vdots & \vdots \\ x_n - mean(x), & y_n - mean(y) \end{bmatrix}$$

$$A = U^T . U$$

When put this in AX=B format, where $X = \begin{bmatrix} a \\ b \end{bmatrix}$

U = np.vstack([x_n - x_mean, y_n - y_mean]).transpose()

A = np.dot(U.transpose(), U)

iii. Take SVD of A to obtain the left singular matrix V. The last column of V corresponds to X

U, S, VT = svd(A)

V = VT.T

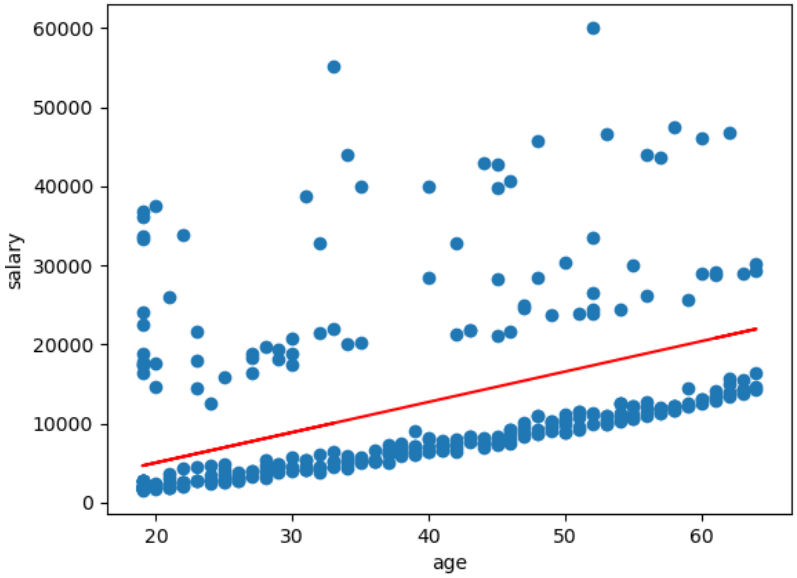
X = V[:, V.shape[1]-1]

a = X[0]

b = X[1]

iv. After a and b are obtained, d is then calculated as a*mean(x) + b*mean(y) = d

v. After rearranging the above equation in the form of y = mx + c, we get,
m = -a/b and c = d/b



Advantages: It takes into consideration the non-independent samples as well, which is helpful

Disadvantages: It takes into account the outliers weightage which can lead to inaccurate curve fitting

Problems: No particular problems faced

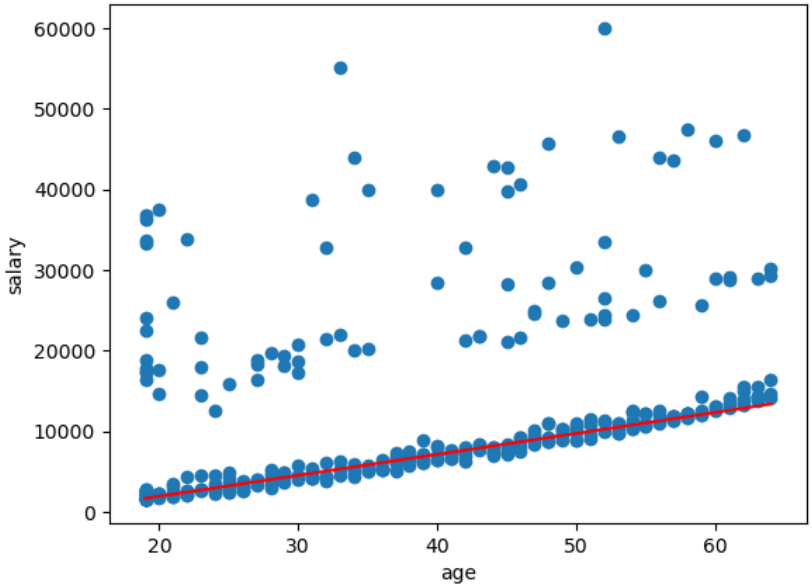
C. RANSAC: RANdom SAmple Consensus

Step 1: Choose two random set of points from the dataset.
Step 2: Find the best fitting curve for those two points using standard linear least square method.
Step 3: Check the error between each point in the dataset with respect to the best fitting curve.
Step 4: Keep a count *inlier_cnt* of the number of points that have an error less than a particular error threshold.
Step 5: Repeat step 1 through 4 for n number of iteration and store the curve with the highest *inlier_cnt*. This curve is our best fitting curve using RANSAC method
The error threshold is set by taking the standard deviation of the y- axis data points.

To decide the number of iterations(n) required to get the best fitting curve is derived from this,

$$n = \frac{\log 1 - p}{\log(1 - (1 - e)^2)}$$

Where, p is the probability of success (set as 95%) and e is the outlier ratio (set as 50%)



Advantages:
- robust estimation of the model parameters, i.e., it can estimate the parameters with a high degree of accuracy even when a significant number of outliers are present in the data set

Disadvantages:
- there is no upper bound on the time it takes to compute these parameters. When the number of iterations computed is limited the solution obtained may not be optimal, and it may not even be one that fits the data in a good way.
- it requires the setting of problem-specific thresholds

Problems: The error threshold that needed to be set for checking the inliers and the outliers was first being done through trial and error and a slight variation in the threshold gave out more variations in the resulting curve.

Solution: The best curve fit was achieved when the standard deviation was considered and measured. This always gave the best result for this method.

3. All the steps of my solution as explained above. After looking at each result, it is fairly evident that RANSAC is the better choice for outlier rejection and accurate curve fitting. The iterative process of RANSAC makes the method give out a curve that fits way more accurately than other methods that were carried out above.

Problem 4:

1. Singular value decomposition:

Let's try to get the singular value decomposition components of a matrix A.

We need to find the left singular matrix(U), diagonal singular value matrix (Sigma) and right singular matrix(V^T) such that,

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

i. We first calculate the $A.A^T$
ii. Calculate the eigen values and eigen vector of the $A.A^T$ matrix.
iii. Sort the eigen values and their corresponding eigen vectors in descending order.
iv. The square root of the sorted eigen values will get us the singular values that form the Sigma matrix
v. We then calculate $A^T.A$
vi. Calculate the eigen values and eigen vector of the $A^T.A$ matrix and sort them like before.
vii. The eigen vectors of $A^T.A$ give the matrix V. The transpose of V gives us the right singular matrix.
viii. We now calculate the left singular matrix U by using the formula $U_i = \frac{A*V_i^T}{Sigma_i}$
ix. For verification, multiply the three matrices obtained, the resulting matrix should be A.
x. Homography matrix was constructed by taking the last column of $V^T$and reshaping the vector in a 3x3 matrix.

2.The result:

U:

```
[[-0. -0. -0.  0. -0.  0.  0. -1.]
 [-0. -0. -0.  0. -0.  0.  1.  0.]
 [-0. -1.  0.  0.  0. -0. -0.  0.]
 [-0. -0. -0. -0. -1.  1. -0.  0.]
 [-1. -0.  0. -0.  0.  0.  0. -0.]
 [-0. -0. -1. -0.  0. -1.  0. -0.]
 [-0.  0.  0.  0. -1. -0. -0.  0.]
 [-0.  1. -0.  0.  0.  0. -0.  0.]]
```

sigma:

[6.0215e+04, 3.1825e+04, 2.6100e+02, 1.8600e+02, 1.4600e+02 , 6.1000e+01, 4.0000e+00,1.0000e+00]

V.T:

```
[[ 0.  0.  0.  0.  0.  0. -1. -1. -0.]
 [ 0. -0.  0.  0. -0. -0. -1.  1.  0.]
 [-0. -0. -0.  1.  1.  0. -0.  0. -0.]
 [-0.  0. -0.  0. -0. -0. -0. -0.  1.]
 [-0.  1.  0.  1. -0.  0.  0.  0. -0.]
 [ 0.  1.  0. -0.  1. -0. -0.  0.  0.]
 [ 1. -0.  0.  0. -0. -0.  0. -0.  0.]
 [-0. -0.  1. -0.  0. -1. -0.  0. -0.]
 [ 0. -0.  1.  0. -0.  1.  0. -0.  0.]]
```

The homography Matrix is:

```
[[ 6.96774195e+00, -6.45161292e-01,  8.06451612e+01]
 [ 2.32258065e+00, -5.16129034e-01,  1.03225806e+02]
 [ 3.09677420e-02, -6.45161292e-03,  1.00000000e+00]]
```

Problems:

Obtaining U matrix was first being done through the eigen vectors of the $A.A^T$ matrix but the direction of those vectors were coming out to be a different which while verification resulted in a matrix that didn't match the original.

Solution:

Hence to keep the direction of the vectors intact to obtain A, $\sigma_i * U_i = A * V_i^T$ formula was used which gave out the perfect result.