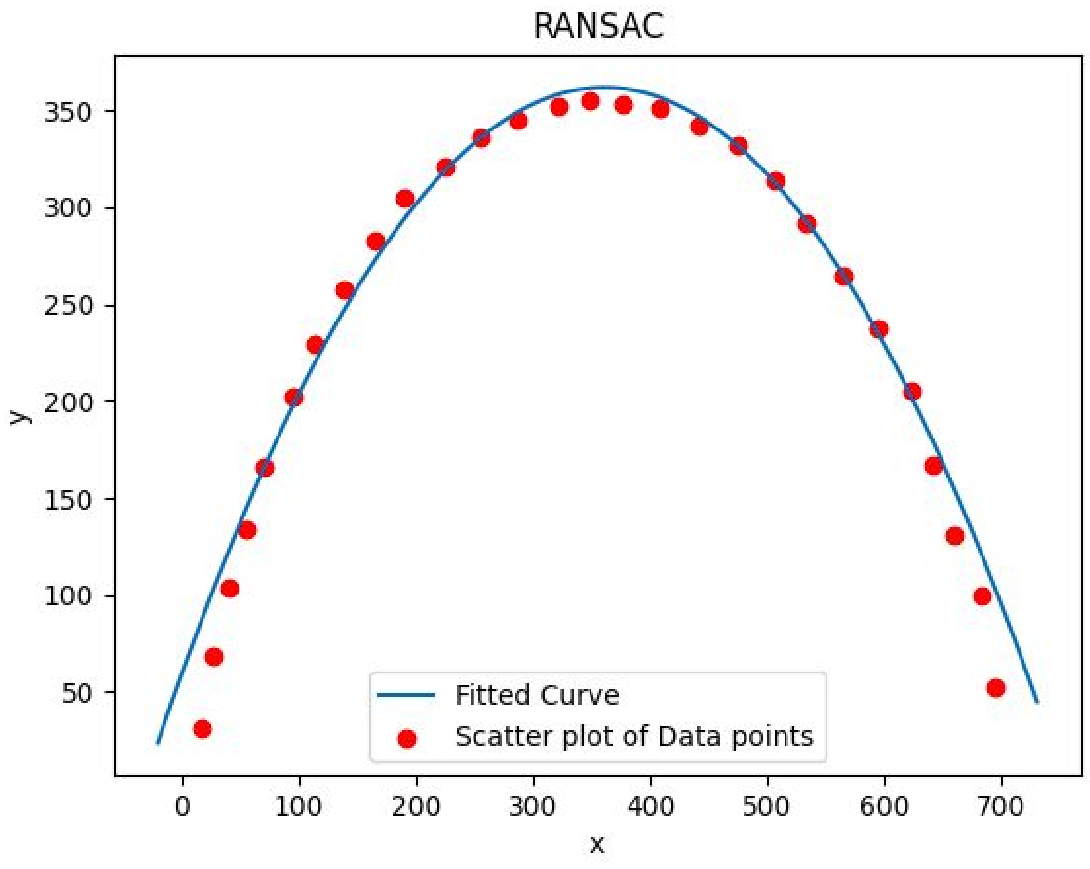
HW1 REPORT

ENPM673 - Perception For Autonomous Robots - Spring 21



Paras Savnani

15.02.2021

University Of Maryland College Park

Question 1:

Given details :

Camera Resolution : 9 MP (9.0* 10^6 pixels)

Camera sensor dimension(d) : 14mm*14mm

Focal length of Camera(f) : 15mm

Ans:

1. Field of View of Camera = ?

   $\phi = tan^{-1}(\frac{d}{2f})$ i.e $tan^{-1}(\frac{14}{2*15})$ = 0.4366 rad

   AFOV = $2 * \phi$ = **0.873 rad / 50.02°**

   FOV = $2 * D * tan(\frac{AFOV}{2})$, where **D** = Distance to the object

   Assuming D = 20m (20000mm) (taken from part 2 of question 1 as not given in part 1),

   FOV = **18.64m** in horizontal direction and **18.64m** in vertical direction.

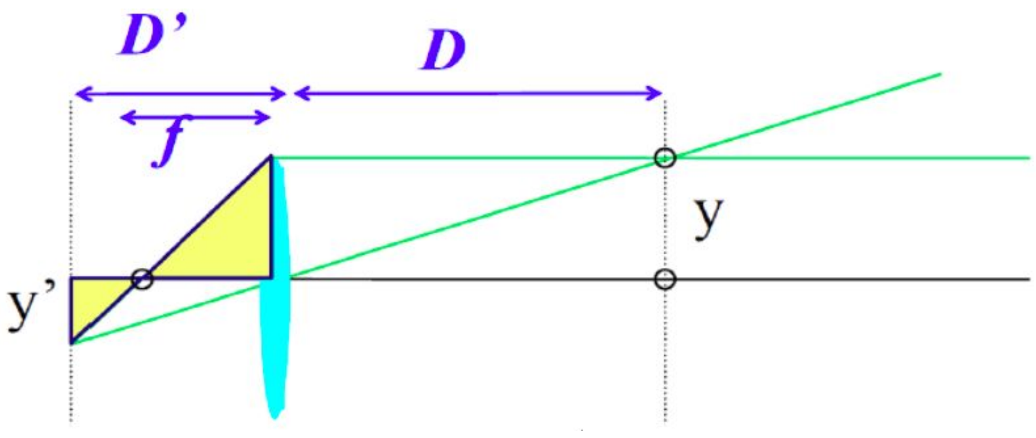
2. **Given:**

   Object size (y) : 50mm*50mm,

   Distance to the object(D) : 20m (20000mm),

   Minimum number of pixels the object will occupy in the camera = ?

Here, y is the object height and y^1 is the image height, also D is the distance of object from the lens and D^1 is the distance of the image from the lens:



Using the thin lens formula:

$\frac{1}{D} + \frac{1}{D^1} = \frac{1}{f}$ i.e $\frac{1}{20000} + \frac{1}{D^1} = \frac{1}{15}$

Solving for D^1, we get $D^1 = 15.01\text{mm}$

Using green similar triangles we get,

$\frac{y^1}{y} = \frac{D^1}{D}$ **,** Solving for y^1, we get $y^1 = 0.0375\text{mm}$

Therefore, image size = **0.0375mm*0.0375mm**.

Now, as Camera has 9.0* 10^6 pixels in 14mm*14mm area, so 1mm*1mm area will have: $\frac{9*10^6}{196} = 45918.37$ pixels. So an area of size 0.0375mm*0.0375mm will have:

45918.3*0.001406 ~ **64 pixels**.

Question 2:

Objective: To fit the best curve for the ball's trajectory using the input from two different videos and compare different fitting methods on basis of efficiency, time of execution and goodness of fit etc.

CODE EXPLANATION AND PROCEDURE:

The main files contain all the function calls and the other files have all the function definitions required to execute the code.

video_process.py

This file has the code to capture the video frame by frame and process each frame. This file uses opencv to perform all image processing operations.

1. Initially the video file is read and the frames(images) are captured in a while 1 loop, moreover the video is resized to 30% to fit in a viewable screen.
2. Then, to separate the red channel from the background a threshold (value =127) is applied on the grayscale image, which reduces the image to black and white image.
3. Next, cv2.findContours is used to find the contours and cv2.moments to find the corresponding centroids of the contours. These contours and centroids are drawn to visualize the ball centre in each image.
4. Finally, the X and Y coordinates of the centroid are stored in 2 lists i.e data_points_x and data_points_y and returned to the main.py file.

Problems faced and solutions:

- (Not necessary to resolve this, but an interesting observation !)

  After loading the video using cv2, its playback speed was different from the original file when played in any external media player. This was happening because of the parameter given in the waitkey. For eg: if we give 10 in the waitkey, the player will wait for 10 ms between each frame(upto a lower limit, because processing also takes some overhead), so to control the playback speed a custom variable frametime was introduced(frametime = 100 ms, then 10fps is the playback speed).

- One cannot append the y coordinates directly because the origin in opencv is taken at the top left corner and while plotting in matplotlib it is in the bottom left corner. So for y coordinates we have to append (height-y) to the list.

sls.py

1. This file has the standard least square function and it uses numpy for matrix operations. To implement this function X and Y matrices are created using the data points found from the video.
2. As the curve is a parabola the equation to fit the points will be $y = ax^2 + bx + c$. Assuming n data points, the X matrix will be an n*3 matrix, having $x^2,\ x\ and\ 1$ values in the columns and matrix Y will be an n*1 matrix with y values.
3. To find the least squares solution, we need to minimize the error equation by equating its derivative to zero. So, we will end up the following equation:

$$X^T XB = X^T Y$$

$$B = (X^T X)^{-1} (X^T Y)$$

4. Finally we will return the B matrix to main.py.

tls.py

1. This file has the total least square function and it uses numpy and custom built svd function to calculate the tls solution. To implement this function X and Y matrices are created using the data points found from the video.
2. To fit the points using total least squares we will use $cy = ax^2 + bx + d$. Also, assuming the data is passing through origin we will drop the constant

value i.e the equation will be $cy = ax^2 + bx$. Assuming n data points, the X matrix will be an n*2 matrix having x^2 *and* x values in the columns and matrix Y will be an n*1 matrix with y values.

3. To find the solution we have to minimize the total least square error i.e we will equate its derivative to zero and will find the final equation.
4. Now to construct the final matrix we will combine X and Y matrices and assign them to Z, further we will find the Singular Value Decomposition(SVD) of the Z matrix, which will give the decomposition of combined matrix as:

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

5. Transpose the V^T matrix to get the coefficients matrix V. Now, separate the initial(a,b) and the last coefficient(c) from V. Finally, divide the initial coefficients by c to get the final coefficients. Return them to the main.py file.

Problems faced and solutions:

- If the data is not assumed to be passing through the origin and the constant d is considered in the equation, then the resulting curve has a very bad fit, thus d had to be dropped from the X matrix.

ransac.py

1. This file has the implementation of the RANSAC function and it uses numpy, math, random, scipy libraries along with sls.py file.
2. To implement the ransac function we need the following parameters:
    - p = desired probability that its a good sample (assumed 0.999999)
    - e = probability of a point being an outlier (#outliers/ #datapoints)
    - s = min number of points to fit the model (3 in our case)
    - t = threshold distance for inliners (decided using trial/error)
    - N = no. of iterations $\left(\frac{log(1-p)}{log(1-(1-e)^s)}\right)$
3. Algorithm:

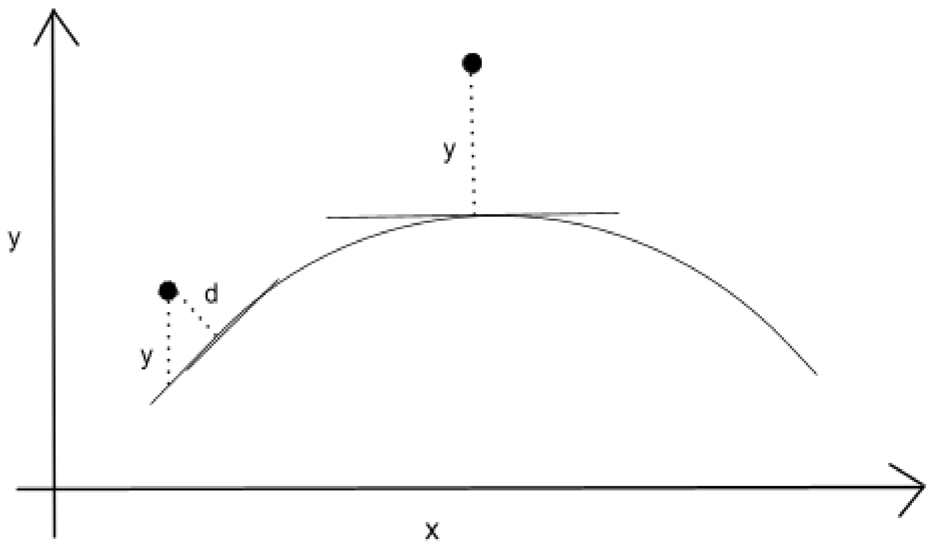
For **N iterations**:-

- Select randomly a subset of s data points from the list of all data points.
- Fit a model to those data points using std_least_square function(it will fit perfectly, because we have 3 points).
- After fitting the model, solve the constraint optimization problem using scipy for each point and find the distance from the point to the curve.
- After finding the distance compare it with the threshold(t), if the distance is less than t increment the inlier_count. Do this for all points for a single model.
- Now, if the new model has the higher inlier_count update the best_model and max_inlier_count variables with result and inlier_count respectively.
    - best_model ← result

repeat…..

Problems faced and solutions:

- To calculate the distance of a point from a parabola, we cannot use the standard error equation $E = (y - (ax^2 + bx + c))^2$ because it will only give vertical error (**error in y direction**), so we use the **fmin_cobyla** minimization approach to find the perpendicular distance between a point and the curve.
- As evident in the below figure, the **d** and **y** distances are much different if the outliers lie at the sides of the parabola but they are almost similar when outlier lies at the top of parabola.



svd.py

1. This file has the custom svd implementation. This is a generalized function which can be used to find the svd of any matrix. The function decomposes the X(m*n) matrix into following parts:

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

2. To calculate the V^T matrix, we will take the dot product of X^T and X matrix.

$$X^T X$$

3. Then, we find the eigenvalues and eigenvectors of the above matrix and sort both of them based on the eigenvalues. (**NOTE: Numpy does not return sorted eigenvalues and eigenvectors, so this step is necessary for implementation of svd.)**
4. And then we take the transpose of the eigenvectors matrix to get V^T matrix.

5. To find the sigma matrix, we stack up the square roots of the sorted eigenvalues along the diagonals to get the Σ matrix.
6. Similarly to get the U matrix we take the dot product of X and X^T matrix.
7. Then, we find the eigenvalues and eigenvectors and sort them based on the eigenvalues.
8. Finally, for the case where (m>n), U can have complex values, so we take the real part of the U matrix and return the U, Σ and V^T matrices.

main1.py and main2.py

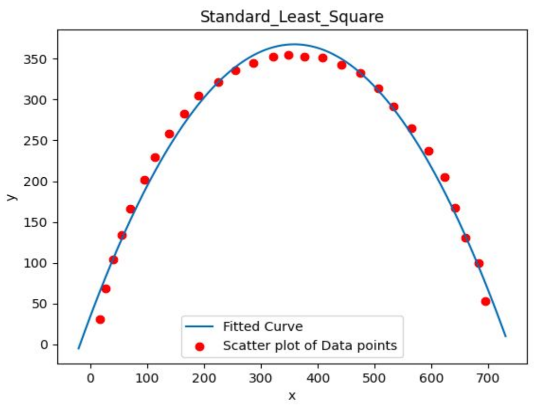
1. The main files contain all the function calls, error calculation and the code to plot the curves. main1.py executes the three methods for the first video and main2.py executes for the second video.
2. The error_cal function helps to calculate the goodness of the fit of the three methods for comparison. The method having the least error is preferable for fitting.
3. Also the time module is used to time the runtimes of the methods for comparison.
4. An x array of 1000 points is generated to plot the curve using np.linspace() and the corresponding y functions are written using the result matrices.
5. Finally, the data points and the curve are plotted to visualize the different methods and their fits.

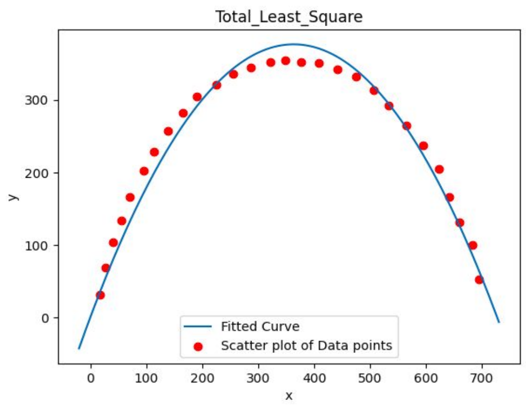
Problems faced and solutions:

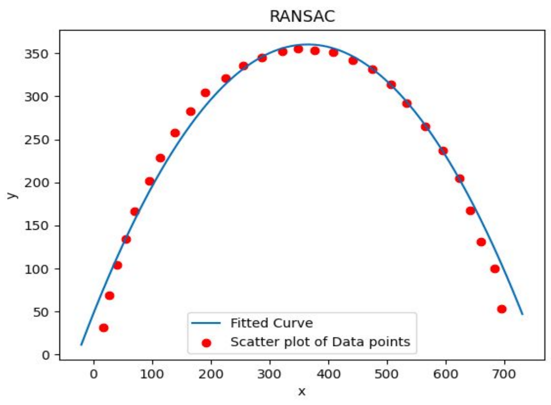
- Initially, the subplot function was used, but it squished the plots, so for better scaled plots they were plotted in different figures and saved locally for reference.

RESULTS

1. Analysis of the first video(Without noise):



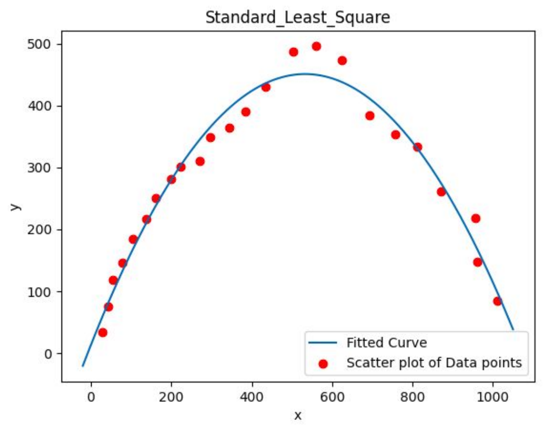


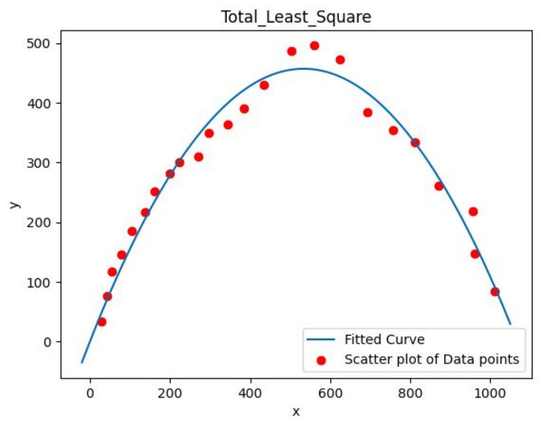


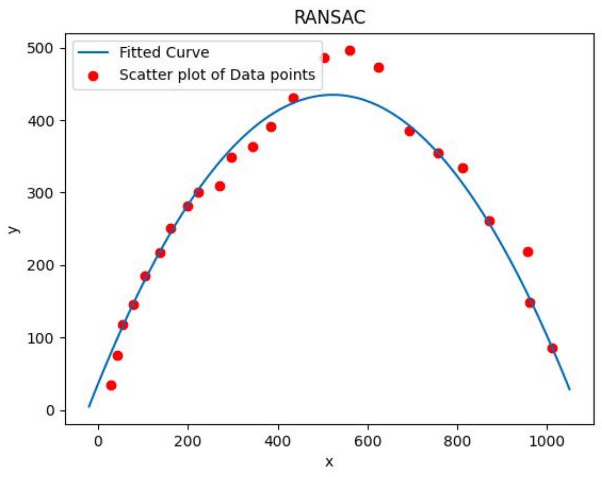
For the first video, standard least squares technique gives the least error(4500.34), and takes the least time to execute(0.0008s), whereas total least squares takes more time(0.0019) and has a significant error(9128.40). Also, as the RANSAC is a probabilistic model, its performance depends upon the number of iterations(i.e parameters), so it gives varying results, also it takes a significant amount of time to fit due to its high time complexity(~O(n^3)). So, for the first video **standard least squares** solution should be opted.

Parameters	Standard Least Square	Total Least Square	RANSAC (error and time depends upon no. of iterations)
Error	4500.34	9128.40	4708.51(varies)
Time of execution(s) (Machine dependent)	~0.0008	~0.0019	~0.615 (varies)
Code Complexity	Simple	Simple	Moderate

2. Analysis of the second video(With noise):







The second video contains a lot of noise, thus it has many outliers present. Now, standard least squares and total least squares methods don't take outliers into consideration and thus try to fit them into the curve, but the ransac method takes them into consideration and rejects the outliers, to fit a better curve. Here, although the RANSAC method takes some time to compute due to its high time complexity, but it has the least error(7430.49), which makes it desirable for these cases. (**NOTE: The error is calculated after removing the outliers.)** So, for the second video **RANSAC algorithm** should be opted.

DATA

Parameters	Standard Least Square	Total Least Square	RANSAC (error and time depends upon no. of iterations)
Error (after removing outliers)	8691.58	10762.75	7430.49(varies)
Time of execution(s) (Machine dependent)	~0.00075	~0.0010	~1.14 (varies)
Code Complexity	Simple	Simple	Moderate

Question 3:

Find the homography matrix (H) ?

Ans:

Homography maps images of points which lie on a world plane from one camera view to another. It is a projective relationship between source and destination points. You need at least 4 points to find the homography matrix between two views.

(Please refer **homography.py** and **svd.py** for question 3)

homography.py

1. To compute the homography matrix(H), we have to solve the homogeneous equation **AH=0**.
2. We can construct the A matrix from the source points and the destination points as given in the question.
3. Now, AH=0 is a homogeneous equation and to solve this we have to impose the constraint **|H| = 1,** to avoid the obvious solution of H being all zeroes. Then we find the svd of the A matrix and decompose it in U, Σ and V^T matrices.
4. We select the last singular vector of V^T as the solution and reshape it to 3*3 to get the homography matrix. Also, verify the solution with the numpy's inbuilt svd.

To Compute the SVD of any matrix follow the below mentioned steps. Suppose A is an **m*n** matrix.........

$$\mathbf{A} = U\Sigma V^T$$

1. To calculate V^T matrix we take the dot product of $A^T and\, A$. And then we find the eigenvalues and eigenvectors of the $A^T A$ matrix.
2. To do this we will make make the determinant of $A^T A - \lambda I$ as 0.

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

3. Solving this equation, we will get the eigenvalues and to get the eigenvectors we can find the null space vectors of $A^T A - \lambda I$ and normalize them by dividing with the magnitude.
4. Therefore, the V matrix will have these eigenvectors (sorted in decreasing order) as its column and V's transpose will give V^T matrix. It will be an (n*n) matrix
5. To get the Σ matrix, take the eigenvalues(sorted in decreasing order) and place them in the diagonal row in increasing order and fill all other elements with 0. This will be an (m*n) matrix.
6. To calculate the **U** matrix, we will take the the dot product of A and A^T. And, then similarly we will find the eigenvalues and eigenvectors of the A A^T matrix by making the $det(AA^T - \lambda I) = 0$.
7. After getting the eigenvalues and eigenvectors we will sort them in descending order and these will form the columns of the **U** matrix.

CALCULATION FOR QUESTION 3

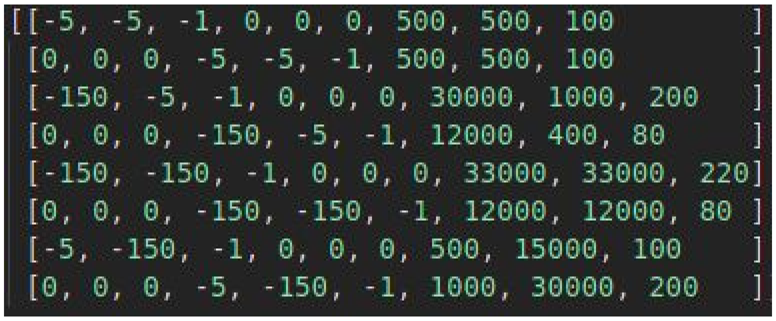
Given:

Source points: [(5, 5), (150, 5), (150, 150), (5, 150)]

Destination points: [(100, 100), (200, 80), (220, 80), (100, 200)]

Find the homography matrix (H)……

A =



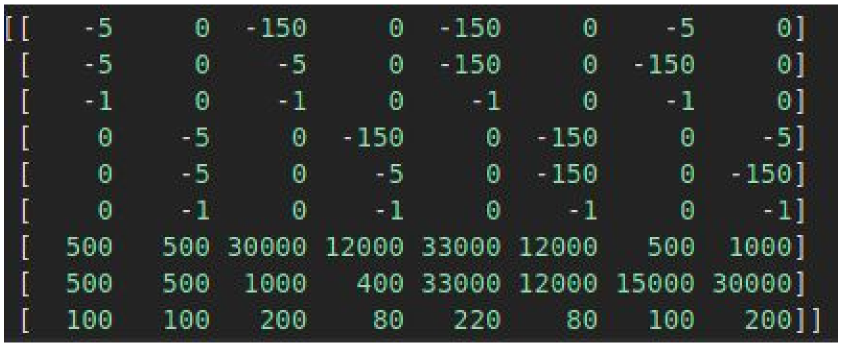
```
[[-5, -5, -1, 0, 0, 0, 500, 500, 100          ]
 [0, 0, 0, -5, -5, -1, 500, 500, 100          ]
 [-150, -5, -1, 0, 0, 0, 30000, 1000, 200    ]
 [0, 0, 0, -150, -5, -1, 12000, 400, 80       ]
 [-150, -150, -1, 0, 0, 0, 33000, 33000, 220]
 [0, 0, 0, -150, -150, -1, 12000, 12000, 80 ]
 [-5, -150, -1, 0, 0, 0, 500, 15000, 100     ]
 [0, 0, 0, -5, -150, -1, 1000, 30000, 200    ]
```

So we will calculate the SVD of A matrix first and use the least value column of V^T matrix to reshape it into 3*3 matrix to form the homography matrix.

A = $U\Sigma V^T$

1. To compute V^T matrix, first find A^T :

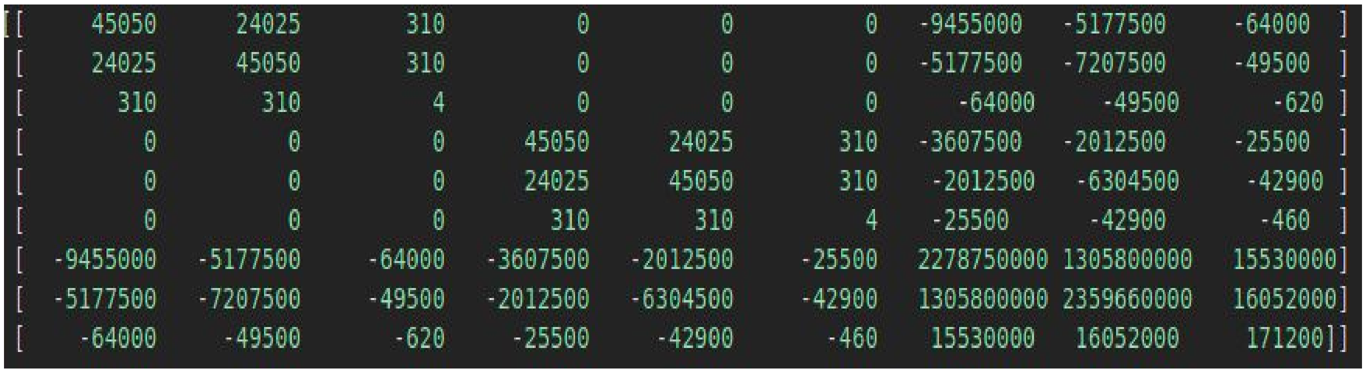
A^T =



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

2. Take the dot product of A^T and **A** :

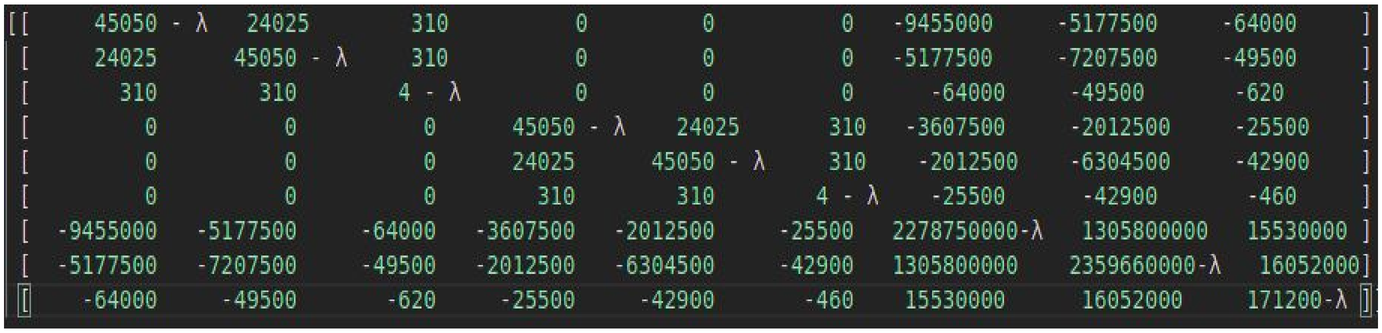
   A^T **A** =



```
[[     45050      24025        310          0          0          0   -9455000   -5177500     -64000   ]
 [     24025      45050        310          0          0          0   -5177500   -7207500     -49500   ]
 [       310        310          4          0          0          0     -64000     -49500       -620   ]
 [         0          0          0      45050      24025        310   -3607500   -2012500     -25500   ]
 [         0          0          0      24025      45050        310   -2012500   -6304500     -42900   ]
 [         0          0          0        310        310          4     -25500     -42900       -460   ]
 [  -9455000   -5177500     -64000   -3607500   -2012500     -25500 2278750000 1305800000   15530000]
 [  -5177500   -7207500     -49500   -2012500   -6304500     -42900 1305800000 2359660000   16052000]
 [    -64000     -49500       -620     -25500     -42900       -460   15530000   16052000     171200]]
```

3. Find the determinant of $A^T A - \lambda I$ and equate it to 0 :

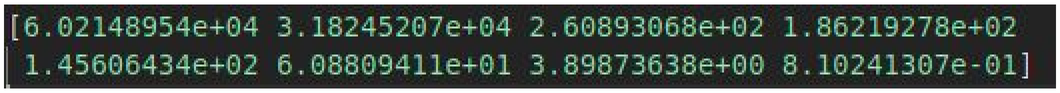
   A^T **A - $\boldsymbol{\lambda}$ I** =



```
[[     45050 - λ    24025         310          0           0          0   -9455000       -5177500       -64000       ]
 [     24025      45050 - λ       310          0           0          0   -5177500       -7207500       -49500       ]
 [       310         310         4 - λ         0           0          0     -64000         -49500         -620       ]
 [         0           0           0      45050 - λ     24025        310   -3607500       -2012500       -25500       ]
 [         0           0           0       24025      45050 - λ      310    -2012500       -6304500       -42900       ]
 [         0           0           0         310         310        4 - λ   -25500         -42900         -460       ]
 [  -9455000    -5177500      -64000    -3607500    -2012500     -25500   2278750000-λ    1305800000     15530000 ]
 [  -5177500    -7207500      -49500    -2012500    -6304500     -42900   1305800000     2359660000-λ   16052000]
 [    -64000      -49500        -620      -25500      -42900       -460    15530000        16052000     171200-λ ]]
```

4. Solving for λ in **det(**A^T **A - $\boldsymbol{\lambda}$ I) = 0**, we get:

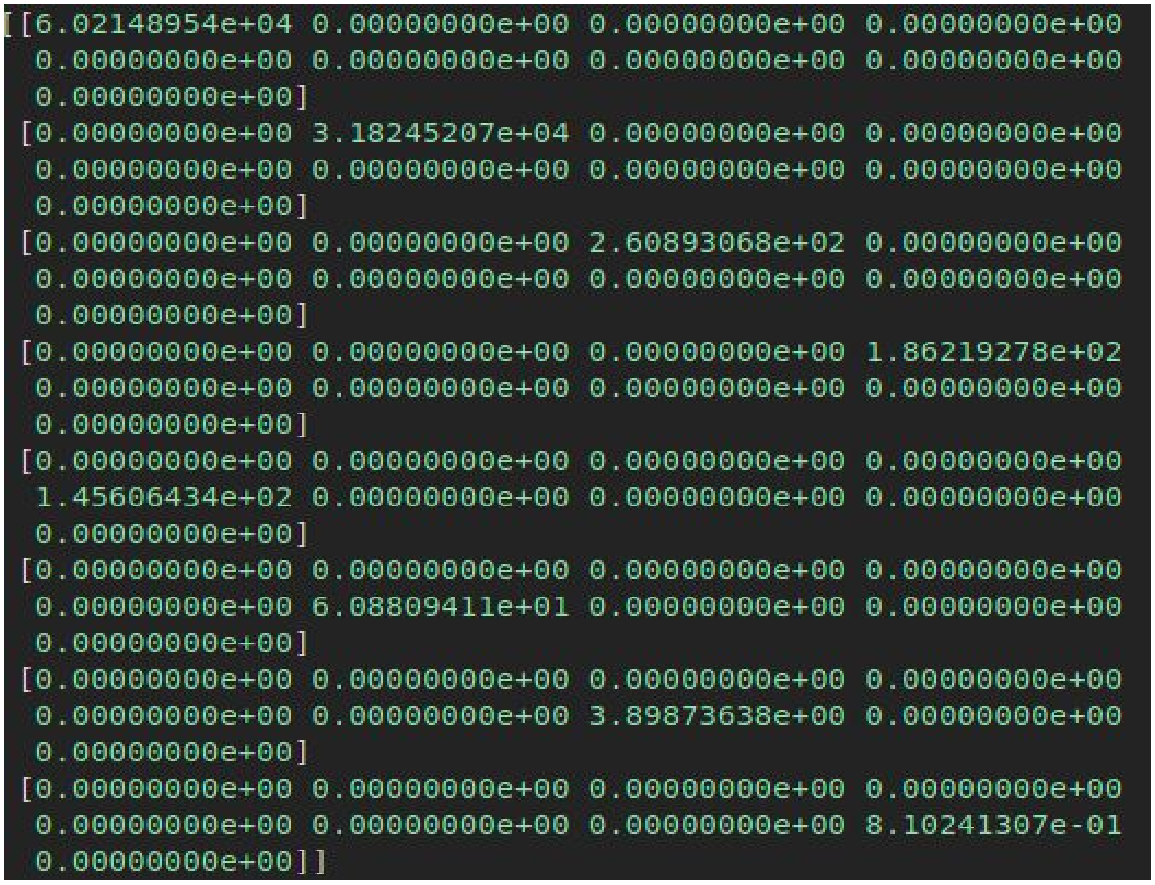
   λ =



```
[6.02148954e+04 3.18245207e+04 2.60893068e+02 1.86219278e+02
 1.45606434e+02 6.08809411e+01 3.89873638e+00 8.10241307e-01]
```

We can put these values in the diagonals to get the sigma (Σ) matrix and leave other elements as zeroes.

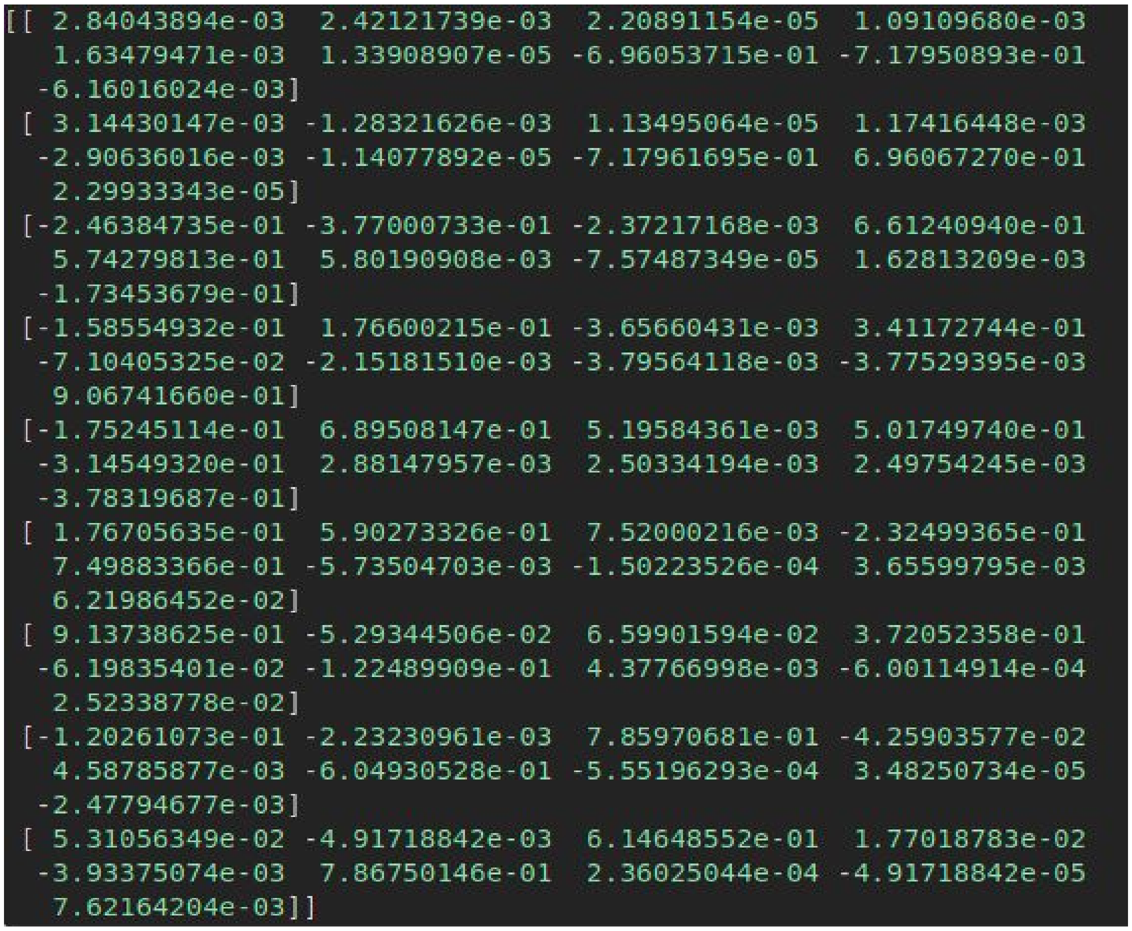
Σ =



```
[[6.02148954e+04 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 3.18245207e+04 0.00000000e+00 0.00000000e+00
  0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 2.60893068e+02 0.00000000e+00
  0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 1.86219278e+02
  0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  1.45606434e+02 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00 6.08809411e+01 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00 0.00000000e+00 3.89873638e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00 0.00000000e+00 0.00000000e+00 8.10241307e-01
  0.00000000e+00]]
```

5. Substitute the eigenvalues in the previous equation and find the corresponding eigenvectors and stack them as columns to form the V matrix and take its transpose to form V^T matrix:

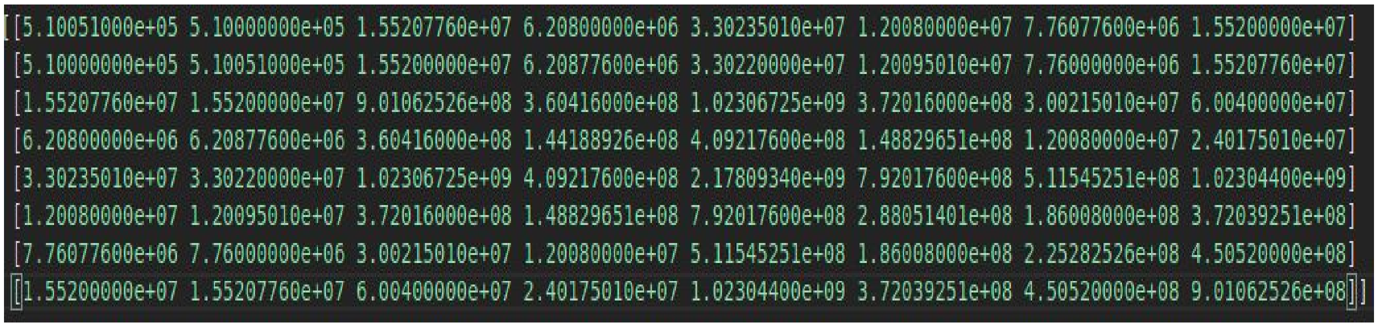
V^T =



```
[[ 2.84043894e-03  2.42121739e-03  2.20891154e-05  1.09109680e-03
   1.63479471e-03  1.33908907e-05 -6.96053715e-01 -7.17950893e-01
  -6.16016024e-03]
 [ 3.14430147e-03 -1.28321626e-03  1.13495064e-05  1.17416448e-03
  -2.90636016e-03 -1.14077892e-05 -7.17961695e-01  6.96067270e-01
   2.29933343e-05]
 [-2.46384735e-01 -3.77000733e-01 -2.37217168e-03  6.61240940e-01
   5.74279813e-01  5.80190908e-03 -7.57487349e-05  1.62813209e-03
  -1.73453679e-01]
 [-1.58554932e-01  1.76600215e-01 -3.65660431e-03  3.41172744e-01
  -7.10405325e-02 -2.15181510e-03 -3.79564118e-03 -3.77529395e-03
   9.06741660e-01]
 [-1.75245114e-01  6.89508147e-01  5.19584361e-03  5.01749740e-01
  -3.14549320e-01  2.88147957e-03  2.50334194e-03  2.49754245e-03
  -3.78319687e-01]
 [ 1.76705635e-01  5.90273326e-01  7.52000216e-03 -2.32499365e-01
   7.49883366e-01 -5.73504703e-03 -1.50223526e-04  3.65599795e-03
   6.21986452e-02]
 [ 9.13738625e-01 -5.29344506e-02  6.59901594e-02  3.72052358e-01
  -6.19835401e-02 -1.22489909e-01  4.37766998e-03 -6.00114914e-04
   2.52338778e-02]
 [-1.20261073e-01 -2.23230961e-03  7.85970681e-01 -4.25903577e-02
   4.58785877e-03 -6.04930528e-01 -5.55196293e-04  3.48250734e-05
  -2.47794677e-03]
 [ 5.31056349e-02 -4.91718842e-03  6.14648552e-01  1.77018783e-02
  -3.93375074e-03  7.86750146e-01  2.36025044e-04 -4.91718842e-05
   7.62164204e-03]]
```

6. To Compute **U** matrix we will follow the same procedure as for V matrix, first we compute the dot product of A and A^T.

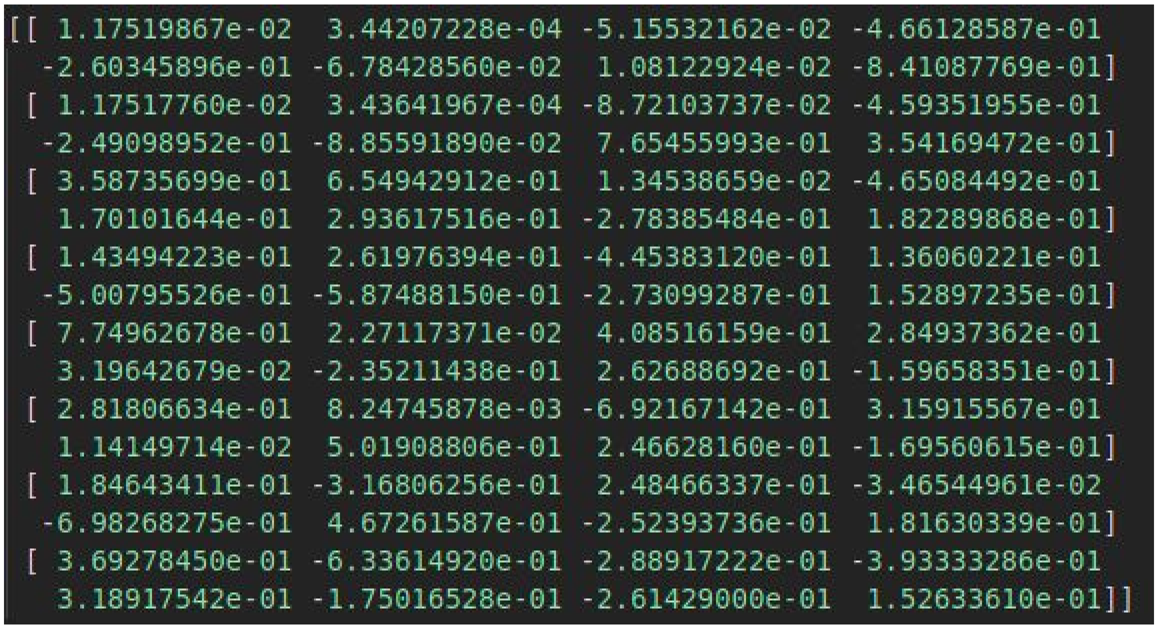
   A A^T =



```
[[5.10051000e+05 5.10000000e+05 1.55207760e+07 6.20800000e+06 3.30235010e+07 1.20080000e+07 7.76077600e+06 1.55200000e+07]
 [5.10000000e+05 5.10051000e+05 1.55200000e+07 6.20877600e+06 3.30220000e+07 1.20095010e+07 7.76000000e+06 1.55207760e+07]
 [1.55207760e+07 1.55200000e+07 9.01062526e+08 3.60416000e+08 1.02306725e+09 3.72016000e+08 3.00215010e+07 6.00400000e+07]
 [6.20800000e+06 6.20877600e+06 3.60416000e+08 1.44188926e+08 4.09217600e+08 1.48829651e+08 1.20080000e+07 2.40175010e+07]
 [3.30235010e+07 3.30220000e+07 1.02306725e+09 4.09217600e+08 2.17809340e+09 7.92017600e+08 5.11545251e+08 1.02304400e+09]
 [1.20080000e+07 1.20095010e+07 3.72016000e+08 1.48829651e+08 7.92017600e+08 2.88051401e+08 1.86008000e+08 3.72039251e+08]
 [7.76077600e+06 7.76000000e+06 3.00215010e+07 1.20080000e+07 5.11545251e+08 1.86008000e+08 2.25282526e+08 4.50520000e+08]
 [1.55200000e+07 1.55207760e+07 6.00400000e+07 2.40175010e+07 1.02304400e+09 3.72039251e+08 4.50520000e+08 9.01062526e+08]]
```

7. Finally, we will calculate the eigenvalues and eigenvectors of the $\mathbf{A}A^T$ matrix by equating the determinant with 0 and solving the equation, therefore:

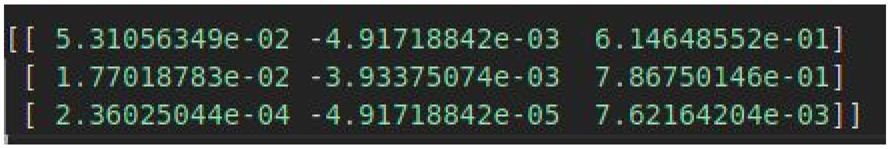
   U =



```
[[ 1.17519867e-02  3.44207228e-04 -5.15532162e-02 -4.66128587e-01
  -2.60345896e-01 -6.78428560e-02  1.08122924e-02 -8.41087769e-01]
 [ 1.17517760e-02  3.43641967e-04 -8.72103737e-02 -4.59351955e-01
  -2.49098952e-01 -8.85591890e-02  7.65455993e-01  3.54169472e-01]
 [ 3.58735699e-01  6.54942912e-01  1.34538659e-02 -4.65084492e-01
   1.70101644e-01  2.93617516e-01 -2.78385484e-01  1.82289868e-01]
 [ 1.43494223e-01  2.61976394e-01 -4.45383120e-01  1.36060221e-01
  -5.00795526e-01 -5.87488150e-01 -2.73099287e-01  1.52897235e-01]
 [ 7.74962678e-01  2.27117371e-02  4.08516159e-01  2.84937362e-01
   3.19642679e-02 -2.35211438e-01  2.62688692e-01 -1.59658351e-01]
 [ 2.81806634e-01  8.24745878e-03 -6.92167142e-01  3.15915567e-01
   1.14149714e-02  5.01908806e-01  2.46628160e-01 -1.69560615e-01]
 [ 1.84643411e-01 -3.16806256e-01  2.48466337e-01 -3.46544961e-02
  -6.98268275e-01  4.67261587e-01 -2.52393736e-01  1.81630339e-01]
 [ 3.69278450e-01 -6.33614920e-01 -2.88917222e-01 -3.93333286e-01
   3.18917542e-01 -1.75016528e-01 -2.61429000e-01  1.52633610e-01]]
```

8. After getting all the required matrices, we will take the V^T matrix and take its last column and reshape it to (3*3) matrix to get the homography matrix.

   H =



```
[[ 5.31056349e-02 -4.91718842e-03  6.14648552e-01]
 [ 1.77018783e-02 -3.93375074e-03  7.86750146e-01]
 [ 2.36025044e-04 -4.91718842e-05  7.62164204e-03]]
```

REFERENCES

- https://math.stackexchange.com/questions/494238/how-to-compute-homography-matrix-h-from-corresponding-points-2d-2d-planar-homog
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin_cobyla.html
- https://math.berkeley.edu/~hutching/teach/54-2017/svd-notes.pdf
- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_table_of_contents_video/py_table_of_contents_video.html#py-table-of-content-video
- http://people.duke.edu/~hpgavin/SystemID/CourseNotes/TotalLeastSquares.pdf
- https://www.youtube.com/watch?v=UKhh_MmGIjM