Stelang@umd.edu

Homework 1

Problem 1

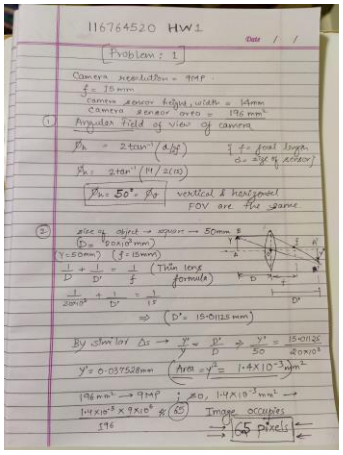
1) Angular Field of view = 2tan^{-1}(d/2f), where d = height of sensor and f = focal length.

Vertical and horizontal FOV are the same here, as the camera sensor is square of 14mm

f=15mm

Hence, **angular FOV = 2tan-1(14/30) = 50°**

2) **Image occupies 65 pixels** – Calculation is done in the file **calculation_pixel.jpg**



116764520 HW1

Date / /

Problem : 1

Camera resolution = 9MP

$f = 15$ mm

camera sensor height, width = 14mm

camera sensor area = 196 mm^2

(1) Angular field of view of camera

$\phi_h = 2\tan^{-1}(d/2f)$ { f = focal length, d = size of sensor }

$\phi_h = 2\tan^{-1}(14/2(15))$

$\boxed{\phi_h = 50° = \phi_v}$ vertical & horizontal FOV are the same

(2) size of object → square → 50mm

($D = 20\times10^3$ mm)

($Y = 50$mm) ($f = 15$mm)

$\frac{1}{D} + \frac{1}{D'} = \frac{1}{f}$ (Thin lens formula)

$\frac{1}{20\times10^3} + \frac{1}{D'} = \frac{1}{15}$

$\Rightarrow$ ($D' = 15.01125$ mm)

By similar Δs → $\frac{y'}{y} = \frac{D'}{D} \Rightarrow \frac{y'}{50} = \frac{15.01125}{20\times10^3}$

$y' = 0.037528$mm (Area $= y'^2 = 1.4\times10^{-3}$ mm^2)

196 mm^2 → 9MP ; so, 1.4×10^{-3} mm^2 →

$\frac{1.4\times10^{-3}\times9\times10^6}{196} \approx$ 65

Image occupies

→ $\boxed{65 \text{ pixels}}$ ←

Stelang@umd.edu

Problem. 2

Given a trajectory of a ball thrown on a white background, our aim is to find the equation of the curve, which would be a parabola.
This can be solved using fitting techniques, which are – Least square, Total least square and RANSAC.

Files:
curveFitting.py - includes LS, TLS, and RANSAC implementation using the below helper file - **myUtils.py**
# Functions:
# 1) calculate_svd - calculate SVD.
# 2) calculate_least_square - calculate least square coefficients.
# 3) calculate_total_least_square - calculate total least square coefficients.
# 4) calculate_RANSAC - calculate RANSAC coefficients.
# 5) centroid - calculate centroid of an image

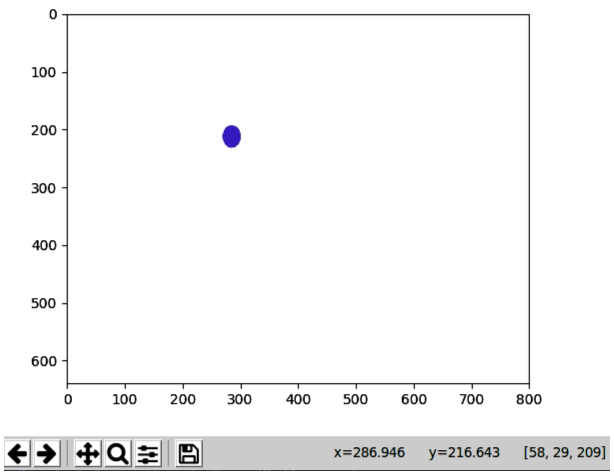
There are four major steps involved:
Step. 1: Find the coordinates / points
Step. 2: Form the matrix equation
Step. 3: Solve the matrix equation using eigen decomposition or SVD and find the coefficients
Step. 4: Plot the graph

Step.1:

We read the input video frame by frame using opencv function and then find the centroid of the red ball, which is thrown on the white background.

- Resize the image frame to 480*640 (height, width) to ease the calculations
- Check the Red value inside the ball [GBR] (fig.1)
- Convert the image to grayscale (fig.2) - check value inside ball – set as threshold
- Use cv.threshold with THRESH_BINARY_INV – with this, all the values above 215, are converted to 255 and rest to Zero
- Now we have all the pixels of the circle which can be used to calculate the Moments and then the centroid

Stelang@umd.edu



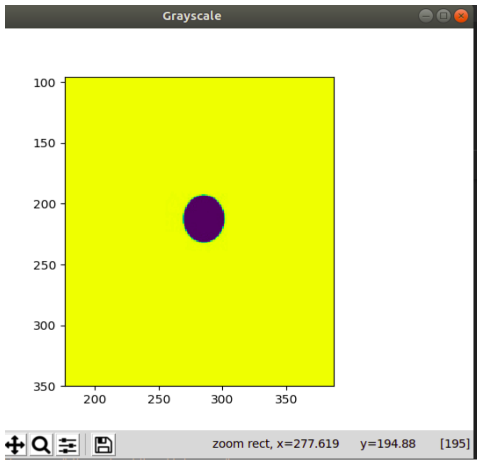


Fig. 1 Shows the various BGR values

Fig.2 shows the value inside the circle in Grayscale

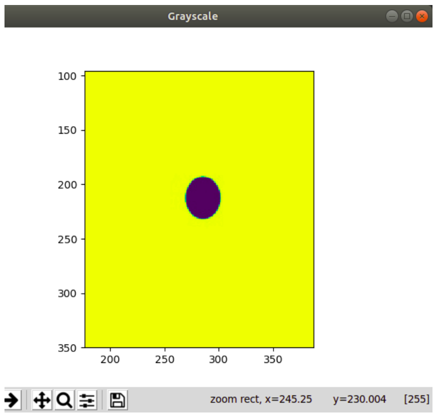
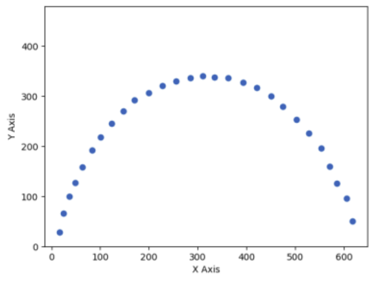
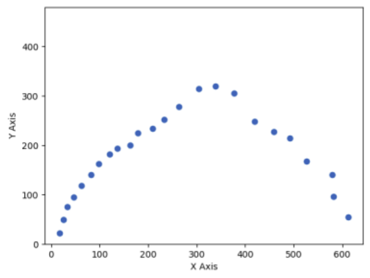


Figure 3. shows value outside the circle

Now that we have all the coordinates, we can plot a scatter plot to check.





(Fig. 4) Scatter plot without noise (Ball_travel_10fps.mp4)

(Fig. 5) Scatter plot with noise (Ball_travel_2_updated.mp4)

As we can see in Fig.4 and Fig.5, we have plot for two input videos – without and with noise. Our aim here is to fit this curve and derive its equation.

Stelang@umd.edu

1) Least Square Method:

Equation of Parabola: $y = ax^2 + bx + c$

Here, we have to find the parameters / coefficients – a, b, and c such that it minimizes the below error:

$E = \Sigma (y - ax^2 - bx - c)^2$

The matrix form can be written as E = Y – XB where X = [x^2 x 1] having each row corresponding to each point and B = [a b c $]^T$

$E = ||Y - XB||^2 = (Y-XB)^T (Y-XB) = Y^T Y - 2(XB)^T Y + (XB)^T (XB)$

Differentiating this with respect to B and equating to zero, we get ->

$$\mathbf{B = (X^TX)^{-1} (X^TY)}$$

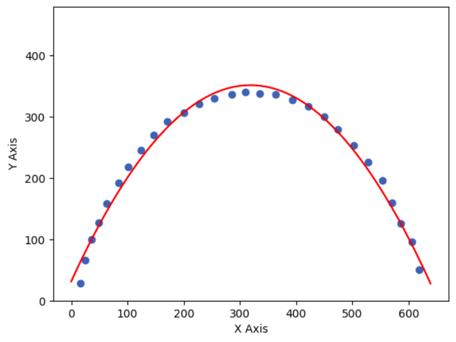
We solve the above equation to get the coefficients and use them to plot the graph.

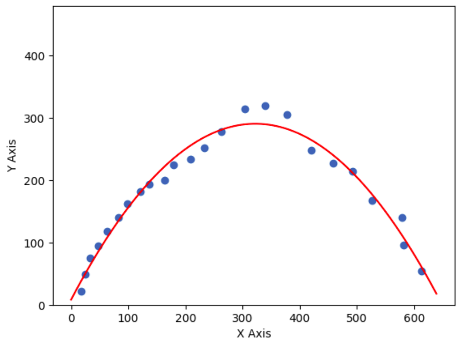
Running the code, the equation we get is:

$y = -0.003140959414214241 5x^2 + 2.004637908269215x + 31.755103053612615$
(Ball_travel_10fps.mp4)

$y = -0.0027055760328379996x^2 + 1.7466857255237707x + 8.834879101673156$
(Ball_travel_2_updated.mp4)

Below Figures show the final curve for both inputs – without noise and with noise using Least Square method.





The least square method holds good without noise, but if there is a noise/outliners in the system, it will fail. If there are more outliners, we would choose either least square with regularization, or RANSAC.

Stelang@umd.edu

2) Total Least Square Method:

As we see in least square method, the error was calculated in only 'x' direction and hence, it would not give us a proper fit if there is error / noise in the vertical 'y' direction.
Total least square method takes in the vertical distance from the point to the line/curve.

Equation formation for Parabola

Vertical distance: $ax^2 + bx + cy = d$

Error $E = \Sigma_i^n (ax_i^2 + bx_i + cy_i - d)^2$

$dE / dd = -2 \Sigma_i^n (ax_i^2 + bx_i + cy_i - d) = 0$

$$\Sigma_i^n d = a \Sigma_i^n x_i^2 + b \Sigma_i^n x_i + c \Sigma_i^n y_i$$

$d = a*(x^2)$ bar + b*(x)bar + c*(y)bar

from here, we can find the parameter d

Now, to find the rest of the coefficients, we will substitute d in the Error equation and the differentiate, equate to zero.

$E = \Sigma_i^n (a(x^2 - (x^2)bar) + b(x - xbar) + c(y - ybar)$

$E = ||ZN||^2$, where Z = row array of [$(x^2 - x^2$ bar) (x – xbar) (y – ybar)] for all points and N = $[a\ b\ c]^T$

$E = Z^TZ. N^2$

$dE/dN = 2(Z^TZ). N = 0$ which is of the form AX = 0

We use SVD to decompose Z^TZ and compute the unknowns; let $S = Z^TZ$
The eigen vector corresponding to the lowest or zero of S^TS will provide us the solution.
$S = UEV^T$ -> Here, as singular values are sorted in descending order, we need to take the last column of V OR last row of V^T , which will correspond to the zero or least eigen value to provide the coefficients of the equation

Then after, as the equation of parabola is $ax^2 + bx + cy = d$, we will reformulate it for y and plot graph for: **$y = (-ax^2 - bx + d) / c$**

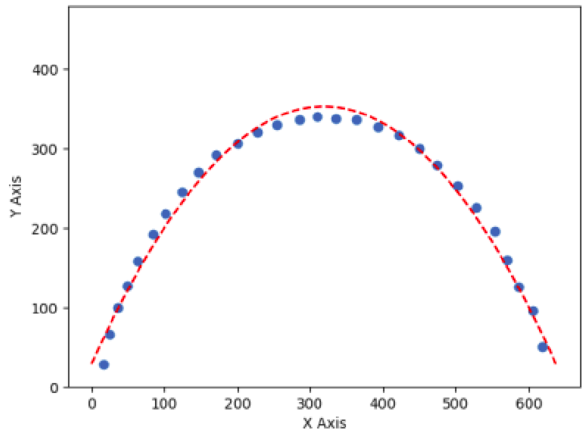
Stelang@umd.edu

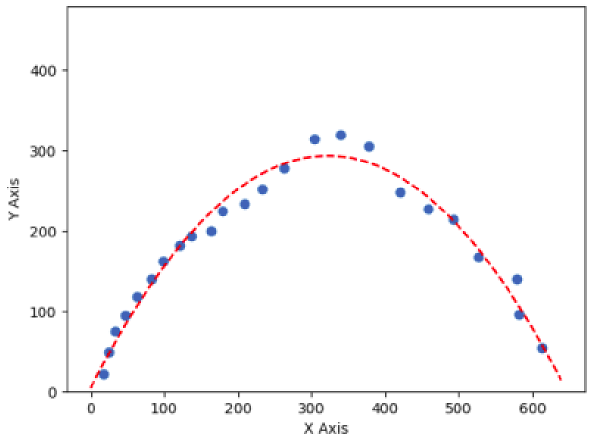
Running the code, the equation comes out to be:

$y = -0.0031793523107517627x^2 + 2.0300606931184926x + 12.856829170223563$ (Ball_travel_10fps.mp4)

$y = -0.0027744791570844226x^2 + 1.791196941362889x + 2.179175931103927$ (Ball_travel_2_updated.mp4)

On running the code, we get the plots for curve using total least square method which are shown below:





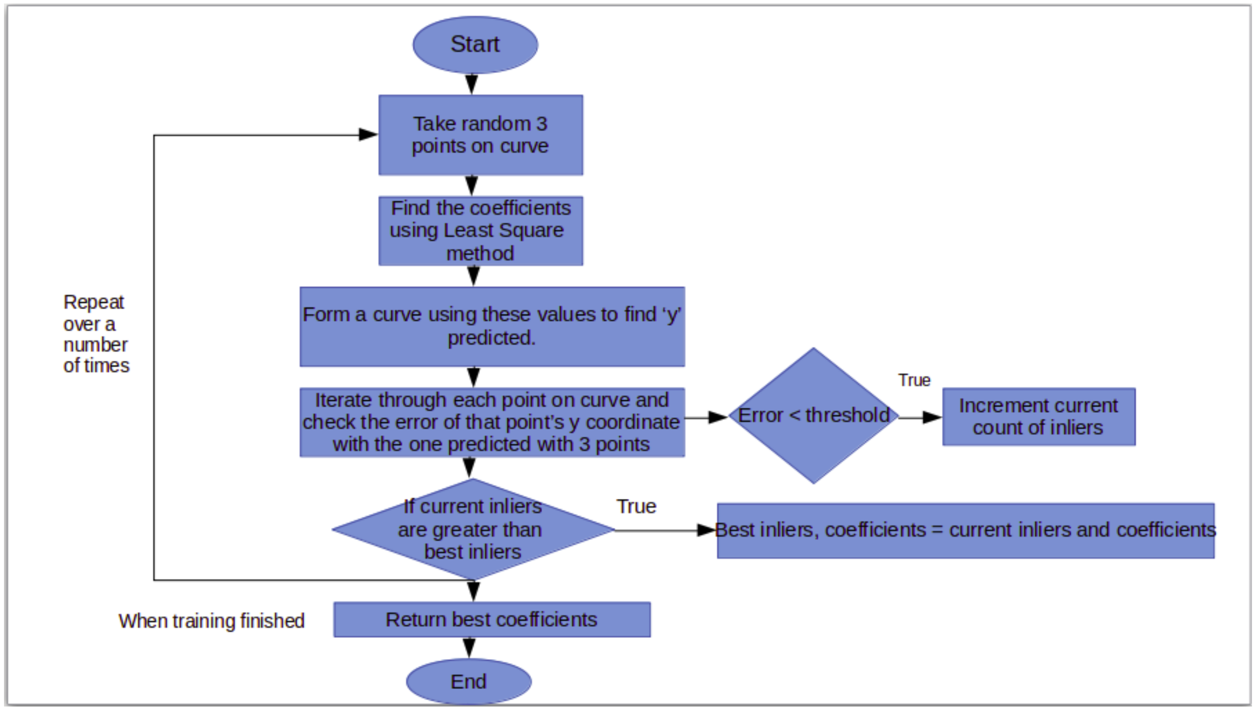
Above figure shows the curve for both the ball tracking's - Ball_travel_10fps.mp4(left) and Ball_travel_2_updated.mp4 (right).

As the above model takes into account the vertical direction, it will give a better prediction of the model having outliners. However, if there are several outliners, RANSAC can be used along with Total Least Square to predict the accurate model.

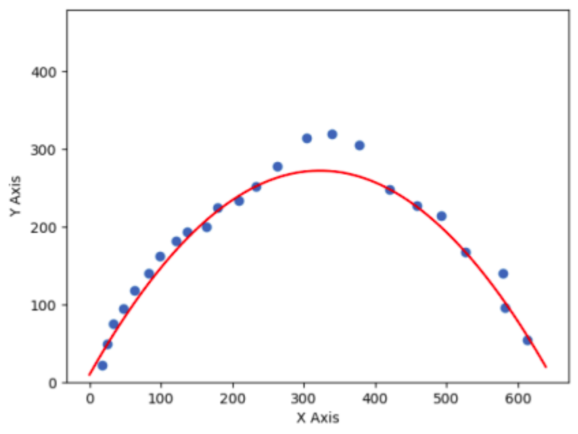
Stelang@umd.edu

3) RANSAC

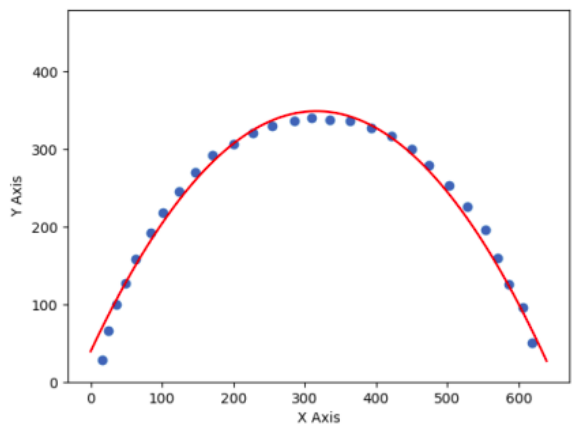
The RANSAC procedure in code is described by the below flow chart


Start
Take random 3 points on curve
Find the coefficients using Least Square method
Form a curve using these values to find 'y' predicted.
Iterate through each point on curve and check the error of that point's y coordinate with the one predicted with 3 points
Error < threshold
True
Increment current count of inliers
If current inliers are greater than best inliers
True
Best inliers, coefficients = current inliers and coefficients
Repeat over a number of times
When training finished
Return best coefficients
End

The RANSAC method is useful when there are outliners in our data. The curve with and without outliners is shown in below figures.



RANSAC with outliners



RANSAC without outliners

RANSAC involves selecting any three random points on the curve, drawing the model with that and checking for inliers count and iterating it over a number of times till we get the best model. This can be combined with Standard least square or least squares to predict a more accurate model.

However, there are a few points that have to be considered here:

- *If there are numerous points, RANSAC will take a lot of time to give the best fit.*
- *The model predicted by any three random points may not necessarily be a curve – it can be a line also. Hence, we need to apply more conditions to check if the 3-point model is actually a curve or not.*
- *The above point can be checked with the coefficients value in comparison with the LS / TLS and the model can be rejected. However, this would require additional checks, inspection and regression.*

Conclusion

Fitting method selection requires analysis of the data. If the data has negligible noise and outliners, Standard least squares will do a good job. However, if there is noise associated with the system, Total Least Squares and RANSAC will give a more accurate prediction of the data. In our case, there are two inputs – without noise and with noise.
For without noise, we see a perfect curve obtained by Least square.
With noise, both RANSAC and Total Least square predict a good model, but Total Least Square predicts better. This is because the number of points are less.
If there were a large data set with outliners, we would have to use RANSAC or a combination of TLS with RANSAC to eliminate the outliners and predict the accurate or best model.

Problem 3

To find the SVD of a matrix, we need to follow below method:

$S = U . E . V^T$

S – be the matrix to be decomposed
U – matrix formed by stacking eigen vectors of of $A.A^T$ in columns

V - matrix formed by stacking eigen vectors of $A^T.A$ in columns

E – Sigma matrix

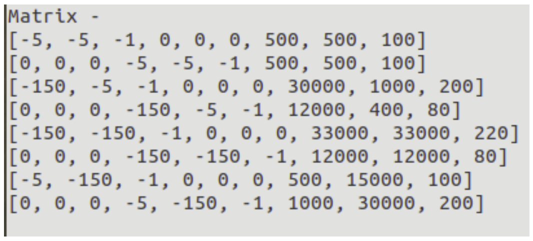
- Find the eigen values corresponding to $A^T.A$
- Take the square root of each value
- Arrange in descending order.
- Form a diagonal matrix of the above values; rest all elements except diagonal are zero

Stelang@umd.edu

It is important to note here that the U and V which have eigen vectors in "i"th column should correspond to their eigen value row in sigma matrix, which would be in descending order.

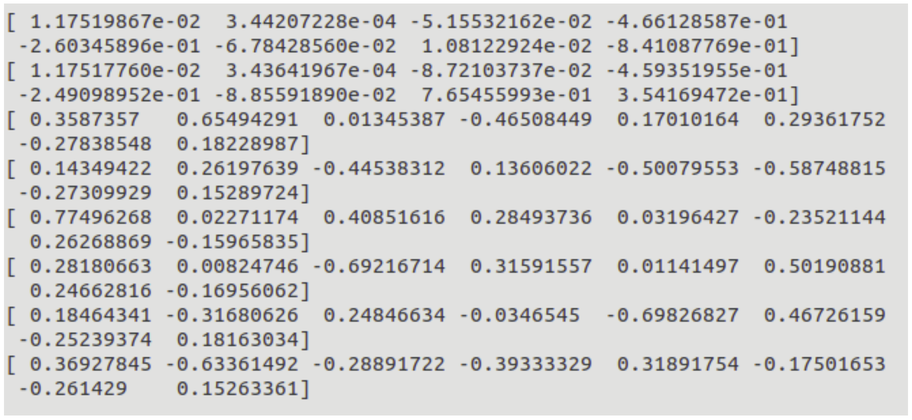
The python code to calculate SVD is in the file: myUtils.py - function – calculate_svd

Matrix to be computed for SVD



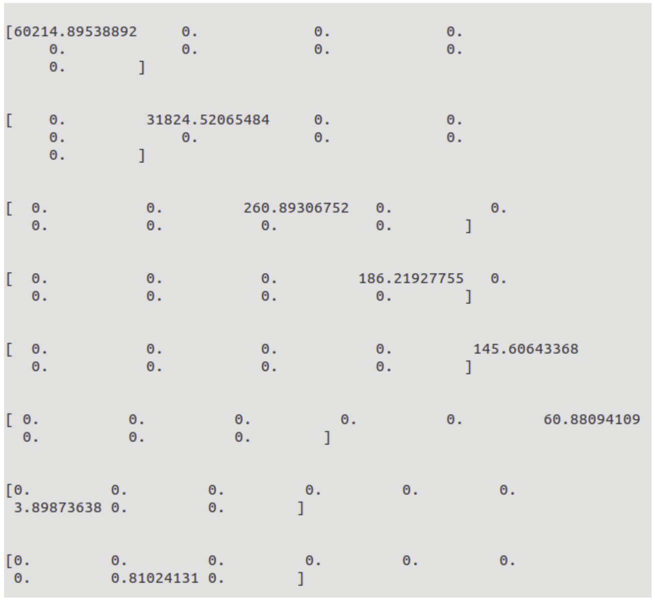
```
Matrix -
[-5, -5, -1, 0, 0, 0, 500, 500, 100]
[0, 0, 0, -5, -5, -1, 500, 500, 100]
[-150, -5, -1, 0, 0, 0, 30000, 1000, 200]
[0, 0, 0, -150, -5, -1, 12000, 400, 80]
[-150, -150, -1, 0, 0, 0, 33000, 33000, 220]
[0, 0, 0, -150, -150, -1, 12000, 12000, 80]
[-5, -150, -1, 0, 0, 0, 500, 15000, 100]
[0,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

Matrix 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]
[ 0.3587357   0.65494291  0.01345387 -0.46508449  0.17010164  0.29361752
 -0.27838548  0.18228987]
[ 0.14349422  0.26197639 -0.44538312  0.13606022 -0.50079553 -0.58748815
 -0.27309929  0.15289724]
[ 0.77496268  0.02271174  0.40851616  0.28493736  0.03196427 -0.23521144
  0.26268869 -0.15965835]
[ 0.28180663  0.00824746 -0.69216714  0.31591557  0.01141497  0.50190881
  0.24662816 -0.16956062]
[ 0.18464341 -0.31680626  0.24846634 -0.0346545  -0.69826827  0.46726159
 -0.25239374  0.18163034]
[ 0.36927845 -0.63361492 -0.28891722 -0.39333329  0.31891754 -0.17501653
 -0.261429    0.15263361]
```

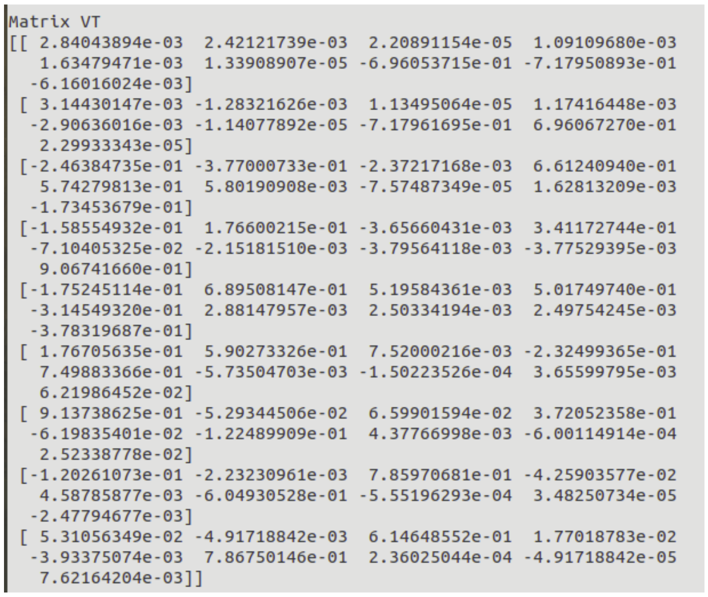
Matrix E ->



```
[60214.89538892     0.             0.             0.
     0.             0.             0.             0.
     0.        ]


[    0.         31824.52065484     0.             0.
     0.             0.             0.             0.
     0.        ]


[  0.           0.         260.89306752   0.           0.
   0.           0.           0.           0.        ]


[  0.           0.           0.         186.21927755   0.
   0.           0.           0.           0.        ]


[  0.           0.           0.           0.         145.60643368
   0.           0.           0.           0.        ]


[ 0.          0.          0.          0.          0.         60.88094109
  0.          0.          0.        ]


[0.         0.         0.         0.         0.         0.
 3.89873638 0.         0.        ]


[0.         0.         0.         0.         0.         0.
 0.         0.81024131 0.        ]
```

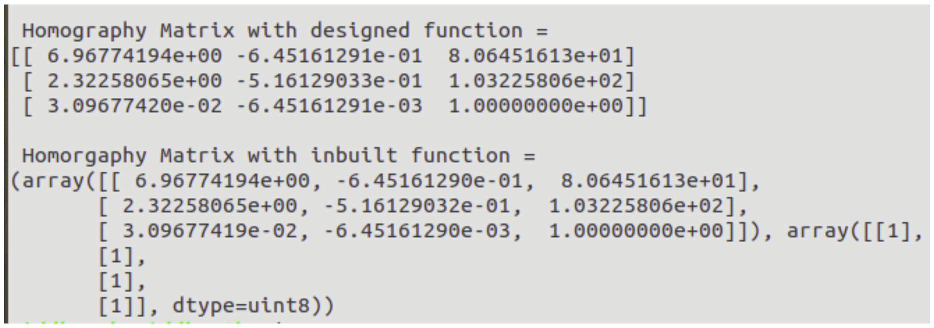
Stelang@umd.edu

Matrix V^T



```
Matrix VT
[[ 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]]
```

Homography Matrix calculated by self-designed function and inbuilt funtion:



```
 Homography Matrix with designed function =
[[ 6.96774194e+00 -6.45161291e-01  8.06451613e+01]
 [ 2.32258065e+00 -5.16129033e-01  1.03225806e+02]
 [ 3.09677420e-02 -6.45161291e-03  1.00000000e+00]]

 Homorgaphy Matrix with inbuilt function =
(array([[ 6.96774194e+00, -6.45161290e-01,  8.06451613e+01],
       [ 2.32258065e+00, -5.16129032e-01,  1.03225806e+02],
       [ 3.09677419e-02, -6.45161290e-03,  1.00000000e+00]]), array([[1],
       [1],
       [1],
       [1]], dtype=uint8))
```