# PURDUE UNIVERSITY

# ECE 661 COMPUTER VISION

# HOMEWORK 3

SUBMISSION: ARJUN KRAMADHATI GOPI

**EMAIL:** akramadh@purdue.edu

### TASK 1.1 : POINT-TO-POINT CORRESPONDENCE METHOD

The task for the first part of the question is to remove distortion using a Point-to-Point Correspondence approach. For this we use the approach we adopted in the solution for homework 2 of this course.

#### SOLUTION

From the program's perspective, we can split the task into these separate tasks:

- 1. Write code to easily pick the four coordinates which collectively form the region of interest (ROI) in the image.
- 2. Form ROI using the given world plane measurements.
- 3. Calculate point-to-point homography using the two corresponding ROIs
- 4. Use the newly found mapping to determine new pixel value for the resulting image.

Once we know the broad tasks at hand, we can work on the logic for each part. The first task, then, would be to calculate the homography. Let the point **A** on the worl plane **PQRS** be denoted by the HC representation (x,y,1). That is to say that the point **A** has the coordinates (x,y) in the physical plane **PQRS**. Let the corresponding point **B** on the image plane **ABCD** be denoted by the HC representation (x',y',1). That is to say that the point **B** has the coordinates (x',y') in the physical image plane **ABCD**. We can say that for a particular homography **H** there exists the relation AH=B. Let us consider the general homography matrix representation:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} = 1 \end{bmatrix}$$

The last element is 1 because the homography matrix is homogeneous and non singular. By taking it as 1, we make sure the last row does not become (0,0,0) and also the ratio is maintained. So by taking it as 1 we preserve the information. From the equation AH=B we get:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Solving the above equation we get the following three equations:

$$a_{11}x + a_{12}y + a_{13} = x'$$
$$a_{21}x + a_{22}y + a_{23} = y'$$
$$a_{31}x + a_{32}y + 1 = 1$$

Dividing the first equation by 1 on both sides we get:

$$\frac{a_{11}x + a_{12}y + a_{13}}{1} = \frac{x'}{1}$$

This can be written as:

$$\frac{a_{11}x + a_{12}y + a_{13}}{a_{31}x + a_{32}y + 1} = \frac{x'}{1}$$

Because

 $a_{31}x + a_{32}y + a_{33} = 1$ 

Similarly for the second equation we get:

$$\frac{a_{21}x + a_{22}y + a_{23}}{a_{31}x + a_{32}y + 1} = \frac{y'}{1}$$

After simplification we get the following two equations to solve:

$$a_{11}x + a_{12}y + a_{13} = a_{31}xx' + a_{32}yx' + x'$$
$$a_{21}x + a_{22}y + a_{23} = a_{31}xy' + a_{32}yy' + y'$$

These can be written in the form:

$$x' = a_{11}x + a_{12}y + a_{13} - a_{31}xx' - a_{32}yx'$$
$$y' = a_{21}x + a_{22}y + a_{23} - a_{31}xy' - a_{32}yy'$$

A system with 8 unknowns needs at least 8 equations to solve. Let us take three more pairs of equations which describe the correspondence between the pair of points  $(x_1, y_1)and(x'_1, y'_1), (x_2, y_2)and(x'_2, y'_2), (x_3, y_3)and(x'_3, y'_3).$ 

Thus, we now have a total of 8 equations representing the correspondence between the points (x,y) and (x',y'),  $(x_1, y_1)$  and  $(x'_1, y'_1)$ ,  $(x_2, y_2)$  and  $(x'_2, y'_2)$ ,  $(x_3, y_3)$  and  $(x'_3, y'_3)$  Writing the 8 equations is the correspondence of the second s

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -xx' & yx' \\ 0 & 0 & 0 & x & y & 1 & -xy' & yy' \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & y_3y'_3 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ x'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{bmatrix}$$

By solving the above equation for the values of the H matrix we can then rearrange the terms to arrive at the final 3X3 H matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{bmatrix}$$

Once we have a way to map the pixels, all that is left is to find the actual pixel value for each newly mapped pixel. We know that each pixel has to be located at a specific integer coordinate value. For any point A located at (x,y) in the physical plane, we know that:

#### $x, y \in Integers$

For any point A (x,y) on the physical world plane **PQRS** we can find the corresponding coordinate on the image plane **ABCD** : B (x',y') using the relation:

$$AH = B$$

Unlike the previous solution (in homework 2), we use the inverse homography because we are mapping points from the image plane to the world plane. Therefore the final relation we are looking at is:

$$A = H^{-1}B$$

Note that we form the ROIs for the world image plane using the given measurements. The given measurements are in centimeters. For the purpose of this solution we assume that each pixel measures one centimeter in both height width. Therefore the ROI of the world image is formed in the following way:

- Point one = (0,0)
- Point two = (width,0)
- Point three = (0,height)
- Point four = (width,height)

#### WEIGHTED PIXEL VALUES

This was presented in the solution for homework 2. I am writing it here again because it is relevant for our solution for homework 3.

Once we find the mapping between the world image plane and the source image plane, we get the coordinates of the pixels whose pixel values we need to form the newly transformed image. It is highly likely that the resulting (x',y') value will be float values and not Integer values. But we cannot use the float value coordinates because such a location does not exist on the image plane **ABCD**. Consequently we cannot get the pixel value of such a point. A workaround for this is to find the weighted pixel value of the point using the pixel values of the surrounding pixels as reference values.

Consider four pixels

$$p_1, p_2, p_3, p_4$$

. The pixel values are

 $pv_1, pv_2, pv_3, pv_4$ 

The pixels are such that they form a square around the point (x',y'). That is to say that these four pixels are four of the closest pixels around point B (x',y') that form a square. Therefore, the coordinates of the pixels would be:

 $p_{1} : (floor(x'), floor(y'))$   $p_{2} : (floor(x'), ceil(y'))$   $p_{3} : (ceil(x'), ceil(y'))$   $p_{4} : (ceil(x'), floor(y'))$ 

Where floor() function floors the value of x' or y' to the highest Integer value less than x' or y'. Ceil function ceils the value of x' or y' to the lowest Integer value higher than x' or y'. Next, let us take

 $dist_1, dist_2, dist_3, dist_4$ 

as the distance between the pixels

$$p_1, p_2, p_3, p_4$$

from the point B at (x',y'). Then the weighted pixel value of the coordinate (x',y') is given by the equation:

$$pv_{(x',y')} = \frac{dist_1(pv_1) + dist_2(pv_2) + dist_3(pv_3) + dist_4(pv_4)}{dist_1 + dist_2 + dist_3 + dist_4}$$

Now, we can say that for every point  $\mathbf{A}$  at (x,y) on the plane **PQRS** we have corresponding point  $\mathbf{B}$  on the plane **ABCD** whose pixel value is

 $pv_{(x',y')}$ 

We then construct the new image pixel by pixel. If, the calculated (x',y') lies outside the plane **ABCD** then we assign a RGB value of [0,0,0] to that pixel (black). Else we calculate the weighted pixel value at (x',y') and use that value for the new pixel in the result image.

### TASK 1.2 - TWO-STEP METHOD

The two step approach we need to take involves the following tasks:

• Task a : Remove projective distortion using the vanishing line method. By removing projective distortion, we mean that we eliminate all the converging lines in the image which are supposed to be parallel in the world plane. We do this by mapping the vanishing line back to the line at infinity.

$$l_{vl} \to l_{\infty}$$

• Task b : Remove affine distortion using the cosine theta method. By removing the affine distortion we mean that we eliminate the angles between the parallel lines and make them orthogonal - just like how they are in the world image (reality). We use the known relation:

$$\cos(\theta) = \frac{L^T C_{\infty}^* M}{\sqrt{(L^T C_{\infty}^* L)(M^T C_{\infty}^* M)}}$$

TASK 1.2.A - REMOVING PROJECTIVE DISTORTION

To map the vanishing line back to the line at infinity, we first need to figure out a method to represent the vanishing line in equation. For this, we will need a total of two unique pairs of lines which strictly form two unique pairs of parallel lines in the real world. Because of projective distortion, we know that the original parallel lines in the real world will appear to be converging at a point (known as the vanishing point). Therefore, two such pairs will converge at two unique vanishing points. By knowing the two vanishing points, we have essentially found the vanishing line as all vanishing points have to lie on the vanishing line.

Let us consider two points  $p_1$  and  $p_2$  which lie on a line  $l_1$  in the image. We get the equation of the line  $l_1$  using the relation:

$$l_1 = p_1 X p_2$$

Similarly for two such points  $p_3$  and  $p_4$  on a 'seemingly' parallel line  $l_2$  we get the line using the relation:

$$l_2 = p_3 X p_4$$

The lines  $l_1$  and  $l_2$  converge at a point known as the vanishing point  $vp_1$  then we have:

$$vp_1 = l_1 X l_2$$

$$vp_2 = l_3 X l_4$$

Therefore, we can finally get the vanishing line representation using the relation:

$$l_{vl} = vp_1 X vp_2$$

If  $vl_1$ ,  $vl_2$  and  $vl_3$  are the parameters that represent the vanishing line  $l_{vl}$  then we have the homography matrix H which maps the vanishing line back to the line at infinity given by:

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ vl_1 & vl_2 & vl_3 \end{bmatrix}$$

Where we have  $l_{vl} = [vl_1 \ vl_2 \ vl_3]^T$ 

By obtaining the H matrix as shown above, we create an image with no projective distortion. Of course, we will need to use the inverse H matrix  $H^{-1}$  because we are mapping from the image plane to the world plane.

#### TASK 1.2.B - REMOVING AFFINE DISTORTION

Once we remove the projective distortion from the image, we know that we have restored parallelism in the image. That is to say that we have effectively mapped the vanishing line back to the line at infinity. Now, we are left with parallel lines but their angles are distorted. This means that there is affine distortion in the image. Orthogonal expansion leads to affine distortion. Our task, by removing affine distortion, is to restore the orthogonality of the scene in the image. We do this by using the cosine theta method. By using the earlier mentioned relation:

$$\cos(\theta) = \frac{L^T C_{\infty}^* M}{\sqrt{(L^T C_{\infty}^* L)(M^T C_{\infty}^* M)}}$$

We, in essence, trace our steps back to find the homography by setting the  $\theta$  value = 90 degrees. Therefore, we have  $\cos(90) = 0$  and hence the equation becomes:

$$\frac{L^T C^*_{\infty} M}{\sqrt{(L^T C^*_{\infty} L)(M^T C^*_{\infty} M)}} = 0$$

We know that for an affine homography H, the conic transforms in the following way:

$$C_{\infty}^{*'} = H C_{\infty}^{*} H^{T}$$

It is reasonable to say that in the cos equation, the numerator is equal to 0 since  $\cos(\theta) = 0$ . Therefore, we have:

$$L^{T'}C_{\infty}^{*'}M'=0$$

Using the transform relation for the conic, we get:

$$L^{T'}HC^*_{\infty}H^TM' = 0$$

Using the following relations:

$$C_{\infty}^* = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and

$$H = \begin{bmatrix} A & 0\\ 0 & 1 \end{bmatrix}$$

Also let us take the parameters of the line L as [a b c] and the parameters of the line M as [d e f]. Using the above relations, we simplify the equations to get:

$$HC_{\infty}^{*}H^{T} = \begin{bmatrix} AA^{T} & 0\\ 0 & 0 \end{bmatrix}$$

The complete equation becomes:

$$\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} AA^T & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} d \\ e \\ f \end{bmatrix} = 0$$

We will need to denote  $AA^T$  as matrix S which is

$$S = \begin{bmatrix} s_a & s_b \\ s_b & s_c = 1 \end{bmatrix}$$

Note that  $s_c$  is 1 because the information is in the ratios. Division by 1 preserves the information as it preserves the ratio. Using that, we simplify the equation to get the following equation to solve:

$$s_aad + s_b(ae + bd) + be = 0$$

The above equation has two variables:  $s_a$  and  $s_b$ . Therefore, we will need two equations, at least, to solve them. Hence, we will need to select two unique pairs of orthogonal lines. Using the two equations, we can calculate the matrix S. We know that  $S = AA^T$ . Since A is non-singular and positive definite, we can recover A by a SVD operation (singular value decomposition) where  $A = VDV^T$  From the lecture notes, we will be able to justify that:

$$S = V \begin{bmatrix} \lambda_1^2 & 0\\ 0 & \lambda_2^2 \end{bmatrix} V^T$$

Using this, we compute for A to finally form the matrix H which is:

$$H = \begin{bmatrix} A & 0\\ 0 & 1 \end{bmatrix}$$

We transform the image using this homography multiplied with the projective homography. The coordinates used to calculate the orthogonal lines for this are first calculated based on how they were transformed when we applied projective homography to transform the image.

#### TASK 1.3 ONE STEP APPROACH

The one step approach makes use of the fact that the dual degenerate conic is represented in the form:

$$C_{\infty}^{*'} = \begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f=1 \end{bmatrix}$$

Note that we have chosen to set the value of f as 1 because the information is in the ratios and by setting it to one, we preserve the ratio and hence the information. We now have the following variables to solve for: a, b, c, d, e. A total of 5 variables. Therefore, we will need to identify five orthogonal line pairs to solve for these 5 variables using the equation:

$$L^{T'}C_{\infty}^{*'}M' = 0$$

Further, we find the combined homography by a similar SVD operation of  $C_{\infty}^{*'}$  where the homography matrix H is given by:

$$H = \begin{bmatrix} A & 0\\ \mathbf{v}^T & 1 \end{bmatrix}$$

The method is the same as mentioned in the two step method. Here:

$$S = AA^T$$

further,

$$S = egin{bmatrix} a & \mathrm{b}/2 \ \mathrm{b}/2 & c \end{bmatrix}$$

Once we estimate the homography matrix H, we transform the image to get rid of both the projective and affine distortion in one go.

### RESULTS

The input images have been annotated with the points I used as inputs for the code. The yellow lines represent the points I used for the two step method and the one step method. The red lines represent the points I used for the Point-to-Point Correspondence method. We assume that one pixel is 1 cm for all purposes of this code. The measurements of the world plane are as follows:

- Input 1: Width 75cm, Height 85cm
- Input 2: Width 84cm, Height 74cm
- Input 3: Width 55cm, Height 36cm; I took only one of the three given measurements
- Input 4: Width 3.6cm, Height 3.6cm; For the purpose of scaling, I scaled it by a factor of 10
- Input 5: Width 40cm, Height 30cm;

REGARDING VECTORIZATION: In my source code I have clearly pointed out TWO instances where I have trid to implement some sort of vectorization to avoid the nested for loops. Both the attempts worked well. But the second instance consumed a lot of RAM. In the end I was forced to use the nested for loops to get the best results. But my code still has the functions where the vectorization attempts were made.

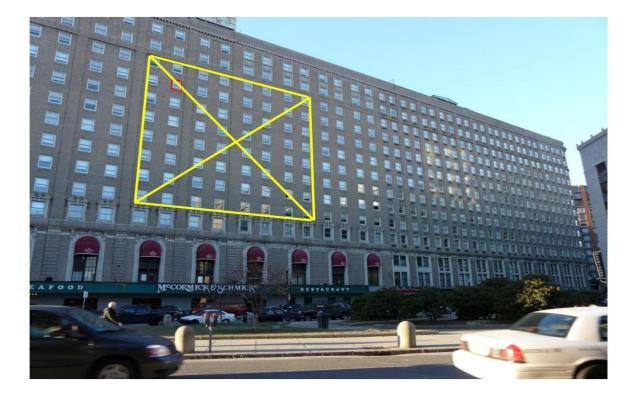


Figure 1: Input Image



Figure 2: Point to Point Correspondence Method



Figure 3: Two Step Method - Removing Projective Distortion Alone



Figure 4: Two Step Method - Removing both Projective and Affine Distortion



Figure 5: One Step Method - Removing both Projective and Affine Distortion



Figure 6: Input Image

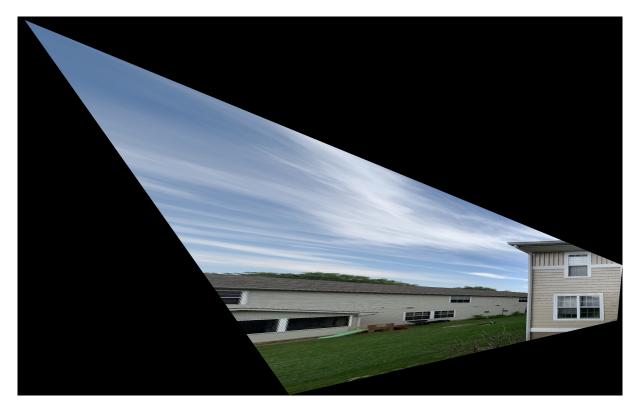


Figure 7: Point to Point Correspondence Method

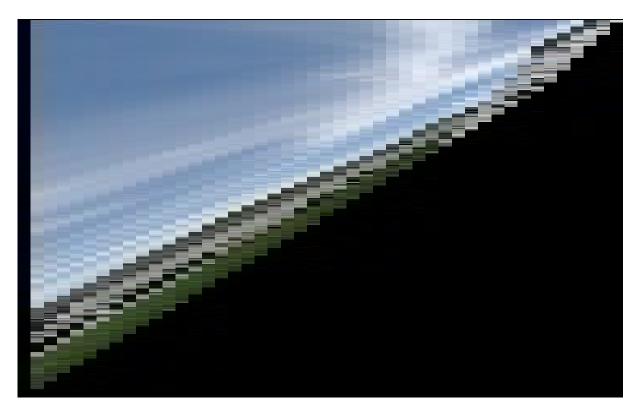


Figure 8: Two Step Method - Removing Projective Distortion Alone

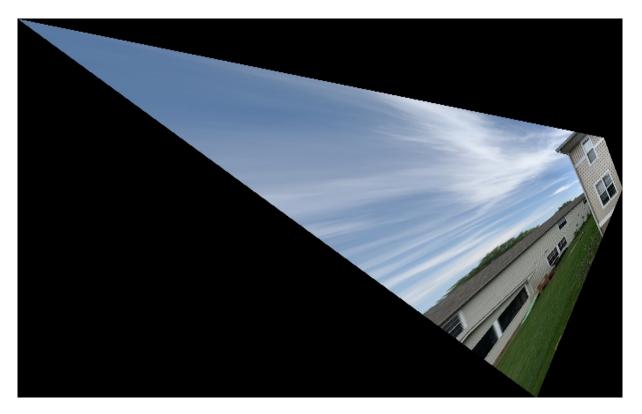


Figure 9: Two Step Method - Removing both Projective and Affine Distortion

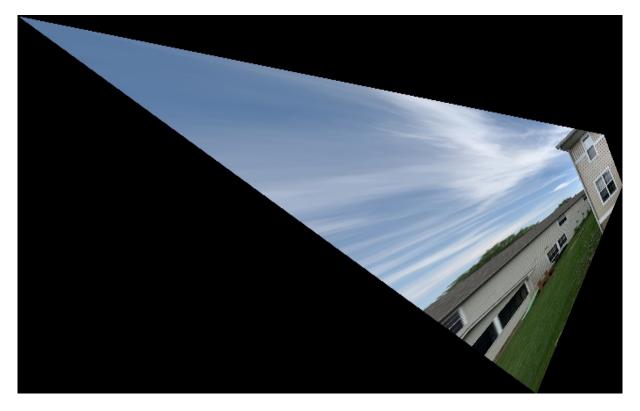


Figure 10: One Step Method - Removing both Projective and Affine Distortion

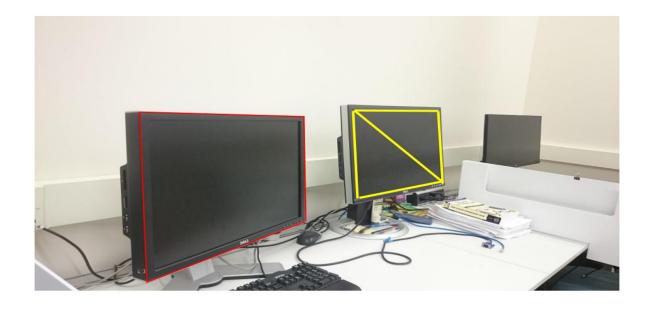


Figure 11: Input Image

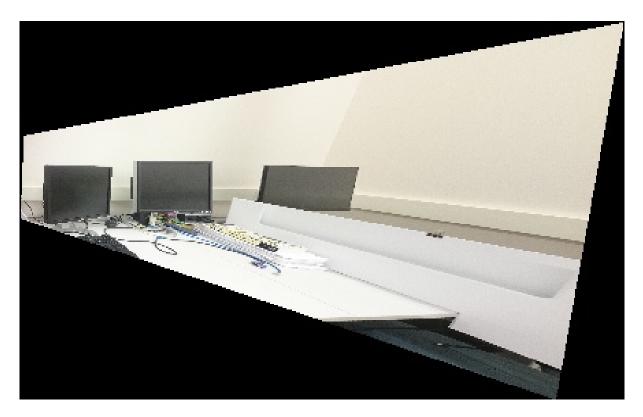


Figure 12: Point to Point Correspondence Method

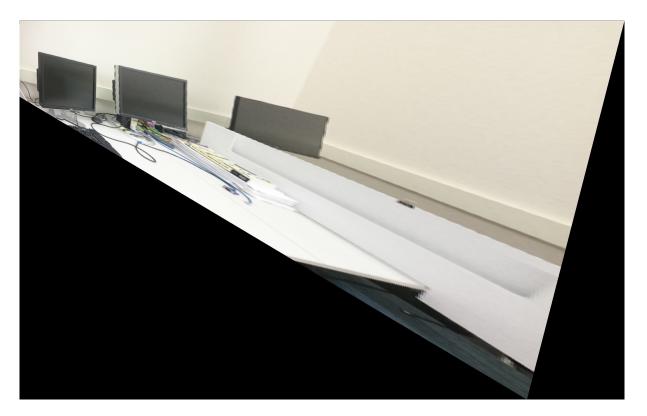


Figure 13: Two Step Method - Removing Projective Distortion Alone

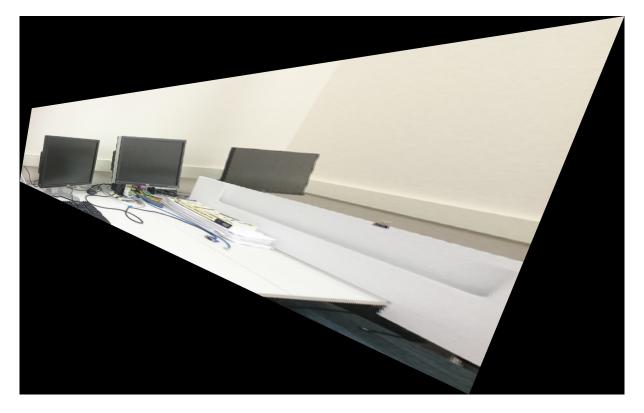


Figure 14: Two Step Method - Removing both Projective and Affine Distortion

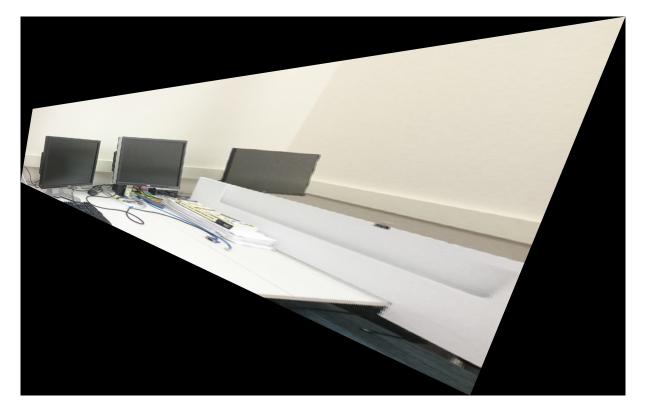


Figure 15: One Step Method - Removing both Projective and Affine Distortion

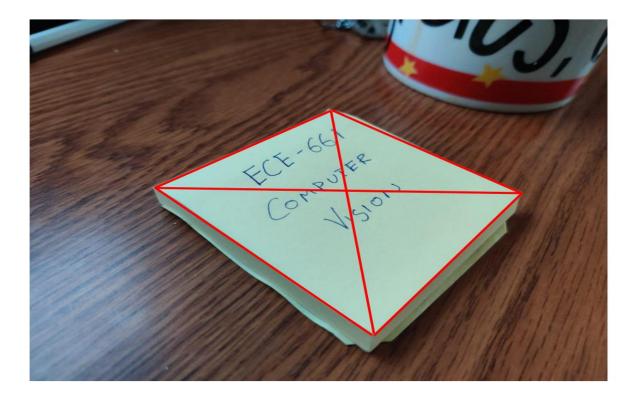


Figure 16: Input Image



Figure 17: Point to Point Correspondence Method

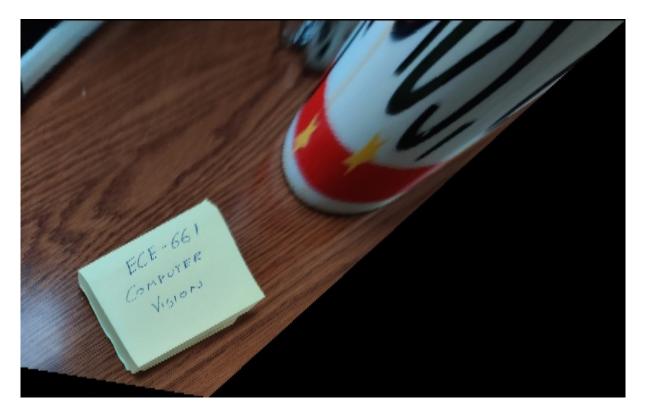


Figure 18: Two Step Method - Removing Projective Distortion Alone

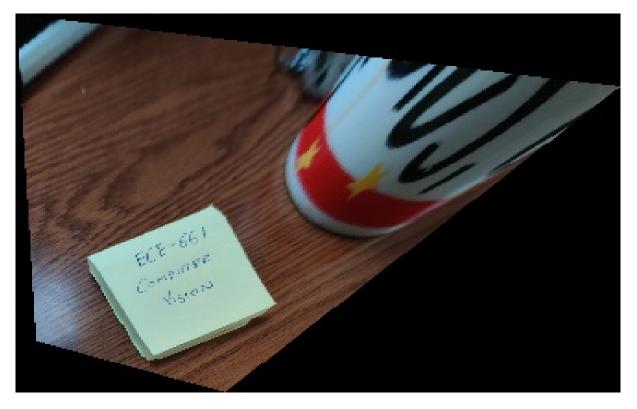


Figure 19: Two Step Method - Removing both Projective and Affine Distortion

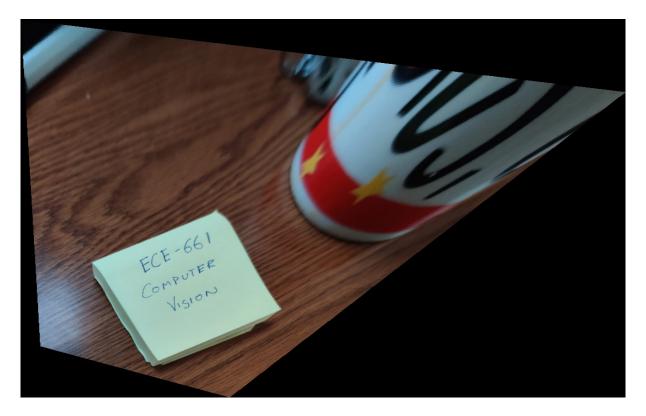


Figure 20: One Step Method - Removing both Projective and Affine Distortion



## Figure 21: Input Image



Figure 22: Point to Point Correspondence Method



Figure 23: Two Step Method - Removing Projective Distortion Alone



Figure 24: Two Step Method - Removing both Projective and Affine Distortion

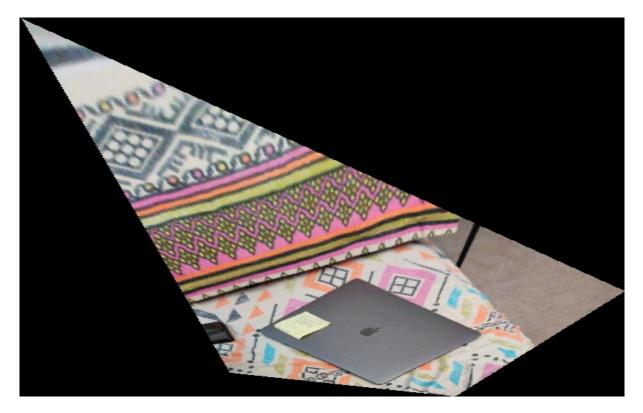


Figure 25: One Step Method - Removing both Projective and Affine Distortion

SOURCE CODE

```
1
   0.0.0
1
   Computer Vision - Purdue University - Homework 3
2
3
   Author : Arjun Kramadhati Gopi, MS-Computer & Information
4
      Technology, Purdue University.
   Date: September 21, 2020
5
6
7
   [TO RUN CODE]: python3 removeDistortion.py
8
9
   The code displays the pictures. The user will have to select the
      ROI points manually in the PQRS fashion.
10
   P ---- Q
11
   12
  13
   R ---- S
14
15
   Output:
16
17
       [jpg]: [Transformed images]
   0.0.0
18
19
20
  import cv2 as cv
21
   import math
22
   import numpy as np
23
   import time
24
```

```
25
26
   class removeDistortion:
27
28
       def __init__(self, image_addresses):
29
30
31
           self.image_addresses=image_addresses
32
           self.image_one = cv.imread(image_addresses[0])
           self.image_one = cv.resize(self.image_one,(int(self.
33
               image_one.shape[1]*0.5), int(self.image_one.shape
               [0]*0.5)))
34
           self.image_two = cv.imread(image_addresses[1])
           # self.image_two = cv.resize(self.image_two,(int(self.
35
               image_two.shape[1]*0.3), int(self.image_two.shape
               [0]*0.3)))
36
           self.image_three = cv.imread(image_addresses[2])
37
           self.image_three = cv.resize(self.image_three,(int(self.
               image_three.shape[1]*0.2), int(self.image_three.shape
               [0]*0.2)))
38
           self.images = [self.image_one,self.image_two,self.
               image_three]
39
           self.image_sizes = [(self.image_one.shape[0], self.
               image_one.shape[1]), (self.image_two.shape[0], self.
               image_two.shape[1]),(self.image_three.shape[0],self.
               image_three.shape[1])]
40
           self.image_sizes_corner_points_HC= []
           self.roiRealWorld = [[(0.0,0.0,1.0),(75.0,0.0,1.0)
41
               ,(0.0,85.0,1.0),(75.0,85.0,1.0)],[(0.0,0.0,1.0)
               (84.0, 0.0, 1.0), (0.0, 74.0, 1.0), (84.0, 74.0, 1.0)
               ], [(0.0, 0.0, 1.0), (55.0, 0.0, 1.0), (0.0, 36.0, 1.0)]
               (55.0, 36.0, 1.0)], [(0.0, 0.0, 1.0), (69.0, 0.0, 1.0)]
               ,(0.0,31.0,1.0),(69.0,31.0,1.0)]]
42
           self.roiCoordinates = []
43
           self.roiList = []
           self.homographies=[]
44
45
           self.resultImg = []
           self.xmin = 0
46
           self.ymin =0
47
           self.createImageCornerPointRepresentations()
48
49
50
51
       def createImageCornerPointRepresentations(self):
52
            [summary] This function creates HC representations of the
53
                corner points of the given original input images.
           ......
54
           templist = []
55
56
           for size in self.image_sizes:
                templist.append(np.asarray([0.0,0.0,1.0]))
57
                templist.append(np.asarray([float(size[1])
58
                   -1.0, 0.0, 1.0]))
                templist.append(np.asarray([0.0,float(size[0])
59
                   -1.0,1.0]))
```

```
templist.append(np.asarray([float(size[1])-1.0,float(
60
                    size[0])-1.0,1.0]))
                 self.image_sizes_corner_points_HC.append(templist)
61
62
                templist = []
63
64
65
        def append_points(self,event,x,y,flags,param):
66
67
            [This function is called every time the mouse left button
                is clicked - It records the (x,y) coordinates of the
               click location]
68
            0.0.0
69
            if event == cv.EVENT_LBUTTONDOWN:
70
71
                 self.roiCoordinates.append((float(x),float(y),1.0))
72
73
74
75
        def getROIFromUser(self):
            0.0.0
76
77
            [This function is responsible for taking the regions of
               interests from the user for all the 4 pictures in
               order]
78
            0.0.0
79
            self.roiList=[]
80
            cv.namedWindow('Select ROI')
81
82
            cv.setMouseCallback('Select ROI', self.append_points)
83
84
            for i in range(3):
                while(True):
85
                     cv.imshow('Select ROI', self.images[i])
86
87
                     k = cv.waitKey(1) \& 0xFF
                     if cv.waitKey(1) & 0xFF == ord('q'):
88
                         break
89
90
                self.roiList.append(self.roiCoordinates)
91
92
                 self.roiCoordinates = []
93
94
95
96
        def weightedPixelValue(self,rangecoordinates,objectQueue):
            0.0.0
97
            [This function calculates the weighted pixel value at the
98
                given coordinate in the target image]
99
100
            Args:
                rangecoordinates ([list]): [This is the coordinate of
101
                     the pixel in the target image]
                 objectQueue ([int]): [This is the index number of the
102
                     list which has the coordinates of the roI for the
                     Object picture]
103
```

<pre>104 Returns: 105 [list]: [Weighted pixel value - RGB value] 106 """ 107 108 pointOne = (int(np.floor(rangecoordinates[1])),int(np. 109 fortTwre = (int(np.floor(rangecoordinates[1])),int(np. 109 ceil(rangecoordinates[0])) 100 pointTwre = (int(np.ceil(rangecoordinates[1])),int(np. 101 ceil(rangecoordinates[0])) 101 pointTwre = (int(np.ceil(rangecoordinates[1])),int(np. 102 ceil(rangecoordinates[0])) 102 jixelValueAtOne = self.images[objectQueue][pointOne[0]][ 103 pixelValueAtTwre = self.images[objectQueue][pointTwre[0]][ 104 pixelValueAtTree = self.images[objectQueue][pointTwre[0]][ 105 pixelValueAtTree = self.images[objectQueue][pointTwre[0]][ 106 pixelValueAtTree = self.images[objectQueue][pointFour 107 [0]][pointThre[1]] 108 veightAtTwre = 1/np.linalg.norm(pixelValueAtTwre- 109 rangecoordinates] 109 veightAtTwre = 1/np.linalg.norm(pixelValueAtTwre- 117 rangecoordinates] 120 veightAtTwre = 1/np.linalg.norm(pixelValueAtTwre- 121 rangecoordinates] 122 veightAtTwre = 1/np.linalg.norm(pixelValueAtTwre- 123 return ((veightAtDne+pixelValueAtTore+ 124 veightAtTwre = 1/np.linalg.norm(pixelValueAtTree) + 125 (veightAtTree + riselValueAtTore+ 126 veightAtThree + self.queueHonography,queuEmage): 127 """[summary] 128 This function is called to create the blank image. The 129 blank image is formed of an array - np.zeros. The size 120 of the blank image is cluulated 129 based on the homography matrix which is being used. The 120 original corner points are used to calculate the new 121 corner points in the new image. 122 [33] Args: 34] Args: 35] Returns: 34] 35] Returns:</pre>		
<pre>106 107 108 109 109 109 109 109 109 109 109 109 100 109 100 109 100 109 100 109 100 100</pre>	104	Returns:
<pre>106 107 108 109 109 109 109 109 109 109 109 109 100 109 100 109 100 109 100 109 100 100</pre>	105	[list]: [Weighted pixe] value - RGB value]
<pre>107 108 pointOne = (int(np.floor(rangecoordinates[1])),int(np. 109 floor(rangecoordinates[0])) 100 pointTre = (int(np.floor(rangecoordinates[1])),int(np. 101 ceil(rangecoordinates[0])) 101 pointTre = (int(np.ceil(rangecoordinates[1])),int(np. 102 ceil(rangecoordinates[0])) 102 pointTre = (int(np.ceil(rangecoordinates[1])),int(np. 103 floor(rangecoordinates[0])) 103 pixelValueAtOne = self.images[objectQueue][pointOne[0]][ 104 pointTwo[1]] 105 pixelValueAtThree = self.images[objectQueue][pointTwo[0]][ 105 pixelValueAtThree = self.images[objectQueue][pointTree 106 [0]][pointTree[1]] 106 pixelValueAtThree = self.images[objectQueue][pointFour 107 [0][[pointTree[1]] 107 veightAtOne = 1/np.linalg.norm(pixelValueAtTwo - 108 rangecoordinates) 109 veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo - 109 rangecoordinates) 120 veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo - 117 rangecoordinates) 121 veightAtTore = 1/np.linalg.norm(pixelValueAtTwo - 122 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* 123 pixelValueAtTwo) + (weightAtTree*pixelValueAtThree) + 124 (weightAtFour*pixelValueAtOne) + (weightAtTwo* 125 pixelValueAtTwo) + (weightAtTree*pixelValueAtThree) + 126 (veightAtTree+wightAtTwo+weightAtToo* 127 rangecoordinates) 128 This function is called to create the blank image. The 129 blank image is formed of an array - np.zeros. The size 120 of the blank image is calculated 121 based on the homography matrix which is being used. The 122 orner points in the new image. 133 Args: 134 queueHomography ([int]): [Index of the homography 134 matrix being used to calculate the new image size] 134 queueHomography ([int]): [Index of the image in the list 134 being used] 134</pre>		· ·
<pre>108 pointOne = (int(np.floor(rangecoordinates[1])),int(np. floor(rangecoordinates[0]))) 100 pointTwo = (int(np.ceil(rangecoordinates[1])),int(np. ceil(rangecoordinates[0]))) 111 pointFour = (int(np.ceil(rangecoordinates[1])),int(np. ceil(rangecoordinates[0]))) 112 113 114 pixelValueAtOne = self.images[objectQueue][pointOne[0]][ pointOne[1]] 115 pixelValueAtTwo = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] 116 pixelValueAtThree = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] 117 118 weightAtOne = 1/np.linalg.norm(pixelValueAtOne - rangecoordinates) 119 weightAtTwo = 1/np.linalg.norm(pixelValueAtThree - rangecoordinates) 120 weightAtTwo = 1/np.linalg.norm(pixelValueAtThree - rangecoordinates) 121 122 123 124 125 125 125 125 126 127 128 129 129 def createBlankImageArray(self,queueHomography,queueImage): 120 """[summary] 128 131 Args: 130 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 131 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 132 134 134 134 134 134 134 134 134 134 134</pre>		
<pre>floor(rangecoordinates[0])) pointTwo = (int(np.floor(rangecoordinates[1])),int(np. ceil(rangecoordinates[0])) pointThree = (int(np.ceil(rangecoordinates[1])),int(np. ceil(rangecoordinates[0])) pointFour = (int(np.ceil(rangecoordinates[1])),int(np. floor(rangecoordinates[0]))) pointTwo[1] pixelValueAtOne = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] pixelValueAtTwo = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] pixelValueAtTon = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] seget and the self.images[objectQueue][pointFour [0]][pointThree[1]] seget and the self.images[objectQueue][pointFour [0]][pointFour[1]] seget and the self.images[objectQueue][pointFour [10][pointFour[1]] seget and the self.images[objectQueue][pointFour [11] seget and the self.images[objectQueue][pointFour [12] seget and the self.images[objectQueue][pointFour [12] seget and the self.image is calculated based on the homography matrix which is being used. The [12] seget and the measing is calculate the new image size] seget and the self and</pre>		
<pre>109 109 109 109 109 109 109 109 109 109</pre>	108	
<pre>ceil(rangecoordinates[0])) pointThree = (int(np.ceil(rangecoordinates[1])),int(np.ceil(rangecoordinates[1])),int(np.floor(rangecoordinates[0])) pointFour = (int(np.ceil(rangecoordinates[1])),int(np.floor(rangecoordinates[0]))) pointToOne[1] pixelValueAtTow = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] pixelValueAtThree = self.images[objectQueue][pointTwe[0]][ pixelValueAtTow = self.images[objectQueue][pointTwo[0]][ pixelValueAtTow = self.images[objectQueue][pointTwo[0]][ pixelValueAtTow = self.images[objectQueue][pointTwo[0]][ pixelValueAtTow = self.images[objectQueue][pointFour [0]][pointFour[1]] veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo-rangecoordinates) veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo-rangecoordinates) veightAtTour = 1/np.linalg.norm(pixelValueAtThree-rangecoordinates) veightAtTour = 1/np.linalg.norm(pixelValueAtThree-rangecoordinates) veightAtTour = 1/np.linalg.norm(pixelValueAtThree) + (veightAtThree + ixelValueAtTour)/(veightAtTour+veightAtTwo+veightAtTour+veightAtT</pre>		÷
<pre>110 111 pointThree = (int(np.ceil(rangecoordinates[1])),int(np.ceil(rangecoordinates[0])) 112 113 pointFour = (int(np.ceil(rangecoordinates[1])),int(np.floor(rangecoordinates[0])) 114 pixelValueAtTone = self.images[objectQueue][pointTwo[0]][ 115 pixelValueAtTree = self.images[objectQueue][pointTree 116 [0]][pointTwo[1]] 117 118 veightAtTour = self.images[objectQueue][pointFour 101][pointFour[1]] 117 118 veightAtTour = self.images[objectQueue][pointFour 101][pointFour[1]] 117 118 veightAtTour = self.images[objectQueue][pointFour 101][pointFour[1]] 117 118 veightAtTour = 1/np.linalg.norm(pixelValueAtTwo- 118 rangecoordinates) 120 veightAtTree = 1/np.linalg.norm(pixelValueAtTree- 121 rangecoordinates) 122 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* 122 pixelValueAtTwo) + (weightAtThree*pixelValueAtTree+ 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* 124 veightAtThree+weightAtTwo*weightAtOne) 124 125 125 126 def createBlankImageArray(self,queueHonography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The 129 based on the homography matrix which is being used. The 129 original corner points are used to calculate the new 130 corner points in the new image. 131 132 queueHomography ([int]): [Index of the homography 133 matrix being used to calculate the new image size] 134 134 134 134 134 134 134 134 134 134</pre>	109	<pre>pointTwo = (int(np.floor(rangecoordinates[1])),int(np.</pre>
<pre>ceil(rangecoordinates[0])) pointFour = (int(np.ceil(rangecoordinates[1])),int(np.floor(rangecoordinates[0]))  iii pixelValueAtTone = self.images[objectQueue][pointOne[0]][ pointTone[1]]  iii pixelValueAtTore = self.images[objectQueue][pointThree [0]][pointThree[1]]  iii pixelValueAtTore = self.images[objectQueue][pointFour [0]][pointFour[1]]  iv eightAtOne = 1/np.linalg.norm(pixelValueAtTon- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore+ rangecoordinates)  def createBlankImageArray(self,queueHomography,queueImage):     """[summary]  def createBlankImageArray(self,queueHomography,queueImage):     """[summary]  This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.  Args: queueImage([int]): [Index of the homography matrix being used] </pre>		<pre>ceil(rangecoordinates[0])))</pre>
<pre>ceil(rangecoordinates[0])) pointFour = (int(np.ceil(rangecoordinates[1])),int(np.floor(rangecoordinates[0]))  iii pixelValueAtTone = self.images[objectQueue][pointOne[0]][ pointTone[1]]  iii pixelValueAtTore = self.images[objectQueue][pointThree [0]][pointThree[1]]  iii pixelValueAtTore = self.images[objectQueue][pointFour [0]][pointFour[1]]  iv eightAtOne = 1/np.linalg.norm(pixelValueAtTon- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtTree = 1/np.linalg.norm(pixelValueAtTore+ rangecoordinates)  def createBlankImageArray(self,queueHomography,queueImage):     """[summary]  def createBlankImageArray(self,queueHomography,queueImage):     """[summary]  This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.  Args: queueImage([int]): [Index of the homography matrix being used] </pre>	110	<pre>pointThree = (int(np.ceil(rangecoordinates[1])),int(np.</pre>
<pre>111 pointFour = (int(np.ceil(rangecoordinates[1])), int(np. floor(rangecoordinates[0]))) 112 113 pixelValueAtOne = self.images[objectQueue][pointTwo[0]][</pre>		
<pre>floor(rangecoordinates[0]))  floor(rangecoordinates[0]))  floor(rangecoordinates[0]))  find pointOne[1]  find pixelValueAtTore = self.images[objectQueue][pointTwo[0]][ pointTwo[1]]  find pixelValueAtTore = self.images[objectQueue][pointTree [0]][pointTree[1]]  find pixelValueAtTore = self.images[objectQueue][pointFour [0]][pointFour[1]]  find weightAtTore = 1/np.linalg.norm(pixelValueAtOne- rangecoordinates)  veightAtTore = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates)  veightAtTore = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates)  veightAtTore = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates)  veightAtFour = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates)  veightAtFour = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtFour = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtFour = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  veightAtFour = 1/np.linalg.norm(pixelValueAtTore- rangecoordinates)  def createBlankImageArray(self,queueHomography,queueImage):     """[summary]  def createBlankImageArray(self,queueHomography,queuEmage):     """[summary]  fhis function is called to create the blank image. The blank image is calculated based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.  Args: queueHomography ([int]): [Index of the homography matrix being used]  Args </pre>	111	•
<pre>112 113 114 115 114 114 115 114 115 115 115 116 116 117 116 117 117 117 117 118 117 118 117 118 119 119 119 119 119 119 119 119 120 120 120 120 121 121 12 12 12 12 12 12 12 12 12 12 1</pre>	111	
<pre>113 pixelValueAtOne = self.images[objectQueue][pointOne[0]][</pre>	110	11001 (langecooldinates [0])))
<pre>pointOne[1]] pixelValueAtTwo = self.images[objectQueue][pointTwo[0]][ pointTwo[1]] pixelValueAtThree = self.images[objectQueue][pointThree [0]][pointThree[1]] pixelValueAtThree = self.images[objectQueue][pointFour [0]][pointFour[1]] veightAtOne = 1/np.linalg.norm(pixelValueAtOne- rangecoordinates) veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo- rangecoordinates) veightAtTree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) veightAtTree = 1/np.linalg.norm(pixelValueAtTree- rangecoordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtTree- rangecoordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtTree+ rangecoordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtTree+ veightAtTree+ veightAtTree+</pre>		
<pre>114 pixelValueAtTwo = self.images[objectQueue][pointTwo[0]][</pre>	113	
<pre>pointTwo[1]] pixelValueAtThree = self.images[objectQueue][pointThree [0]][pointThree[1]] pixelValueAtFour = self.images[objectQueue][pointFour [0]][pointFour[1]]  veightAtOne = 1/np.linalg.norm(pixelValueAtOne- rangeccordinates) veightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangeccordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangeccordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtTwo* veightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangeccordinates) veightAtFour = 1/np.linalg.norm(pixelValueAtFour- veightAtFour = 1/np.linalg.norm(pixelValueAtFour- veightAtFour = 1/np.linal</pre>		-
<pre>115 pixelValueAtThree = self.images[objectQueue][pointThree [0]][pointThree[1]] 116 pixelValueAtFour = self.images[objectQueue][pointFour [0]][pointFour[1]] 117 118 weightAtOne = 1/np.linalg.norm(pixelValueAtOne-</pre>	114	pixelValueAtTwo = self.images[objectQueue][pointTwo[0]][
<pre>115 pixelValueAtThree = self.images[objectQueue][pointThree [0]][pointThree[1]] 116 pixelValueAtFour = self.images[objectQueue][pointFour [0]][pointFour[1]] 117 118 weightAtOne = 1/np.linalg.norm(pixelValueAtOne-</pre>		pointTwo[1]]
<pre>[0]][pointThree[1]] 116     pixelValueAtFour = self.images[objectQueue][pointFour     [0]][pointFour[1]] 117 118     weightAtOne = 1/np.linalg.norm(pixelValueAtOne-     rangecoordinates) 119     weightAtTwe = 1/np.linalg.norm(pixelValueAtTwo-     rangecoordinates) 120     weightAtFour = 1/np.linalg.norm(pixelValueAtThree-     rangecoordinates) 121     weightAtFour = 1/np.linalg.norm(pixelValueAtFour-     rangecoordinates) 122 123     return ((weightAtOne*pixelValueAtOne) + (weightAtTwo*     pixelValueAtTwo) + (weightAtThree*pixelValueAtFour-</pre>	115	pixelValueAtThree = self.images[objectQueue][pointThree
<pre>116 pixelValueAtFour = self.images[objectQueue][pointFour [0]][pointFour[1]] 117 118 weightAtOne = 1/np.linalg.norm(pixelValueAtOne-</pre>		
<pre>[0]][pointFour[1]] 117 118 veightAtOne = 1/np.linalg.norm(pixelValueAtOne- rangecoordinates) 119 veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo- rangecoordinates) 120 veightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) 121 veightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used] 134 </pre>	116	-
<pre>117 118 118 118 119 119 119 119 119 119 119</pre>	110	
<pre>118 weightAtOne = 1/np.linalg.norm(pixelValueAtOne- rangecoordinates) 119 weightAtTwo = 1/np.linalg.norm(pixelValueAtTwo- rangecoordinates) 120 weightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) 121 weightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour)/(weightAtFour+ weightAtThree+weightAtTwo*weightAtOne) 124 125 def createBlankImageArray(self,queueHomography,queueImage): """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used] 134</pre>	117	
<pre>rangecoordinates) iiii veightAtTwo = 1/np.linalg.norm(pixelValueAtTwo- rangecoordinates) iveightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) iveightAtFour = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) iveightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) iveightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) iveightAtFour = 1/np.linalg.norm(pixelValueAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) iveightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) iveightAtThree+weightAtTwo+weightAtOne) iveightAtFour*pixelValueAtFour) iveightAtFour*pixelValueAtFour) iveightAtFour*pixelValueAtFour) iveightAtFour*pixelValueAtFour) iveightAtThree+weightAtTwo+weightAtOne) iveightAtThree+weightAtTwo+weightAtOne) iveightAtFour*pixelValueAtFour) iveightAtFour*pixelValueAt</pre>		
<pre>119 weightAtTwo = 1/np.linalg.norm(pixelValueAtTwo- rangecoordinates) 120 weightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) 121 weightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used] 134</pre>	118	
<pre>rangecoordinates) veightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) 121 veightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) +     (weightAtFour*pixelValueAtFour))/(weightAtFour+     weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The     blank image is formed of an array - np.zeros. The size     of the blank image is calculated 129 based on the homography matrix which is being used. The     original corner points are used to calculate the new     corner points in the new image. 130 131 Args: 132     queueHomography ([int]): [Index of the homography         matrix being used to calculate the new image size] 133 134 </pre>		•
<pre>120 weightAtThree = 1/np.linalg.norm(pixelValueAtThree- rangecoordinates) 121 weightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used] 134</pre>	119	weightAtTwo = 1/np.linalg.norm(pixelValueAtTwo-
<pre>rangecoordinates) 121 rangecoordinates) 121 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) +     (weightAtFour*pixelValueAtFour))/(weightAtFour+     weightAtFour*pixelValueAtFour))/(weightAtFour+     weightAtTree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The     blank image is formed of an array - np.zeros. The size     of the blank image is calculated 129 based on the homography matrix which is being used. The     original corner points are used to calculate the new     corner points in the new image. 130 131 Args: 132     queueHomography ([int]): [Index of the homography     matrix being used to calculate the new image size] 133     queueImage ([int]): [Index of the image in the list         being used] 134 </pre>		rangecoordinates)
<pre>121 weightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134</pre>	120	<pre>weightAtThree = 1/np.linalg.norm(pixelValueAtThree -</pre>
<pre>121 weightAtFour = 1/np.linalg.norm(pixelValueAtFour- rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134</pre>		rangecoordinates)
<pre>rangecoordinates) 122 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo* pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) +     (weightAtFour*pixelValueAtFour))/(weightAtFour+     weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The     blank image is formed of an array - np.zeros. The size     of the blank image is calculated 129 based on the homography matrix which is being used. The     original corner points are used to calculate the new     corner points in the new image. 130 131 Args: 132     queueHomography ([int]): [Index of the homography     matrix being used to calculate the new image size] 133     queueImage ([int]): [Index of the image in the list         being used] 134 </pre>	121	•
<pre>122 123 123 return ((weightAtOne*pixelValueAtOne) + (weightAtTwo*     pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) +         (weightAtFour*pixelValueAtFour))/(weightAtFour+         weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """"[summary] 128 This function is called to create the blank image. The         blank image is formed of an array - np.zeros. The size         of the blank image is calculated 129 based on the homography matrix which is being used. The         original corner points are used to calculate the new         corner points in the new image. 130 131 Args: 132         queueHomography ([int]): [Index of the homography         matrix being used to calculate the new image size] 133 134 </pre>		
<pre>123 123 124 125 124 125 126 126 127 128 128 129 128 129 129 129 130 131 131 131 131 131 131 131 131 131</pre>	122	141600014114000)
<pre>pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) +     (weightAtFour*pixelValueAtFour))/(weightAtFour+     weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The     blank image is formed of an array - np.zeros. The size     of the blank image is calculated 129 based on the homography matrix which is being used. The     original corner points are used to calculate the new     corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography     matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list     being used] 134</pre>		$r_{\text{oturn}}$ ((usight)+0 $r_{\text{oturn}}$ (usight)+ (usight)+ $r_{\text{ust}}$
<pre>(weightAtFour*pixelValueAtFour))/(weightAtFour+ weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used] 134</pre>	123	· · ·
<pre>weightAtThree+weightAtTwo+weightAtOne) 124 125 126 def createBlankImageArray(self,queueHomography,queueImage): 127 """[summary] 128 This function is called to create the blank image. The 129 blank image is formed of an array - np.zeros. The size 129 based on the homography matrix which is being used. The 130 original corner points are used to calculate the new 130 corner points in the new image. 132 queueHomography ([int]): [Index of the homography 133 queueImage ([int]): [Index of the image in the list 134</pre>		
<ul> <li>124</li> <li>125</li> <li>126 def createBlankImageArray(self,queueHomography,queueImage):</li> <li>127 """[summary]</li> <li>128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated</li> <li>129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.</li> <li>130</li> <li>131 Args:</li> <li>132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size]</li> <li>133 queueImage ([int]): [Index of the image in the list being used]</li> </ul>		
<ul> <li>def createBlankImageArray(self,queueHomography,queueImage): <ul> <li>"""[summary]</li> </ul> </li> <li>128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated</li> <li>129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.</li> <li>130 <ul> <li>131 Args:</li> <li>132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size]</li> <li>133 queueImage ([int]): [Index of the image in the list being used]</li> </ul> </li> </ul>		weightAtThree+weightAtTwo+weightAtOne)
<ul> <li>def createBlankImageArray(self,queueHomography,queueImage): """[summary]</li> <li>This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated</li> <li>based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.</li> <li>Args: 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used]</li> </ul>	124	
127 """[summary] 128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated 129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used]	125	
<ul> <li>127 """[summary]</li> <li>128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated</li> <li>129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.</li> <li>130</li> <li>131 Args:</li> <li>132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size]</li> <li>133 queueImage ([int]): [Index of the image in the list being used]</li> </ul>	126	<pre>def createBlankImageArray(self,queueHomography,queueImage):</pre>
<ul> <li>128 This function is called to create the blank image. The blank image is formed of an array - np.zeros. The size of the blank image is calculated</li> <li>129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image.</li> <li>130</li> <li>131 Args:</li> <li>132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list being used]</li> <li>134</li> </ul>	127	
blank image is formed of an array - np.zeros. The size of the blank image is calculated based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134		•
of the blank image is calculated based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134		-
129 based on the homography matrix which is being used. The original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134		
<pre>original corner points are used to calculate the new corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134</pre>	100	-
<pre>corner points in the new image. 130 131 Args: 132 queueHomography ([int]): [Index of the homography</pre>	129	
130 131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134		· ·
131 Args: 132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used]		corner points in the new image.
132 queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list being used] 134		
<pre>matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list</pre>	131	Args:
<pre>matrix being used to calculate the new image size] 133 queueImage ([int]): [Index of the image in the list</pre>	132	queueHomography ([int]): [Index of the homography
133 queueImage ([int]): [Index of the image in the list being used] 134		
being used] 134	133	· · · ·
134		
	134	
		Returns:
	100	

```
[numpy array]: [np.zeros of the size equal to the new
136
                     image size]
137
                 [int]: [Returns the xmin value of the new image - The
                     least x value amongst the four transformed corner
                     points]
                 [int]: [Returns the ymin value of the new image - The
138
                     least y value amongst the four transformed corner
                     points]
            0.0.0
139
140
            templist = []
141
142
            templistX=[]
            templistY=[]
143
            #print(self.image_sizes_corner_points_HC[queueImage])
144
            #print(self.homographies[queueHomography][0])
145
146
            for i in range(4):
147
                templist.append(np.dot(self.homographies[
                    queueHomography], self.image_sizes_corner_points_HC
                    [queueImage][i]))
            #print(templist)
148
149
150
            for i,element in enumerate(templist):
                templist[i] = element/element[2]
151
152
            for element in templist:
153
                templistX.append(element[0])
                templistY.append(element[1])
154
155
            breadth = int(math.ceil(max(templistX))) - int(math.floor
156
               (min(templistX)))
157
            height = int(math.ceil(max(templistY))) - int(math.floor(
               min(templistY)))
158
            return np.zeros((height, breadth, 3)), int(math.floor(min(
159
               templistX))),int(math.floor(min(templistY)))
160
161
162
        def createImage(self,queueHomography,queueImage):
163
            """[summary]
164
            This function is the function which creates the final
165
               result image. This function has the traditional but
               slow nested for loop approach to build the image.
166
            It begins by first getting the blank image of the size of
                the new image from the createBlankImageArray function
                above.
167
168
            Args:
                queueHomography ([int]): [Index of the homography
169
                   matrix being used to calculate the new image size]
                queueImage ([int]): [Index of the image in the list
170
                    being used]
171
172
            Returns:
```

173	[numpy ndarray]: [Returns the final resultant image
	in numpy.ndarray form.]
174	
175	<pre>print("Processing")</pre>
176	resultImg,xmin,ymin = <i>self</i> .createBlankImageArray(
	queueHomography,queueImage)
177	
178	<pre>for column in range(0,resultImg.shape[0]):</pre>
179	<pre>for row in range(0,resultImg.shape[1]):</pre>
180	<pre>print("processing" + str(column) + " out of "+</pre>
	<pre>str(resultImg.shape[0]))</pre>
181	<pre>rangecoordinates = np.dot(self.homographies[</pre>
	<pre>queueHomography+1],(float(row+xmin),float(</pre>
	column+ymin),1.0))
182	rangecoordinates = rangecoordinates/
	rangecoordinates [2]
183	<b>0</b>
184	<pre>if ((rangecoordinates[0]&gt;0) and (rangecoordinates</pre>
	[0] < self.image_sizes[queueImage][1]-1)) and ((
	rangecoordinates [1] >0) and (rangecoordinates
	[1] < self.image_sizes[queueImage][0]-1)):
185	resultImg[column][row] = self.
	weightedPixelValue(rangecoordinates,
	queueImage)
186	else:
187	resultImg[column][row] = [0,0,0]
188	
188 189	return resultImg
189	return resultImg
189 190	
189	<pre>def createImageVectorised(self,queueHomography,queueImage):</pre>
189 190 191 192	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]</pre>
189 190 191	<pre>def createImageVectorised(self,queueHomography,queueImage):</pre>
189 190 191 192 193	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1</pre>
189 190 191 192 193 194	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]</pre>
189 190 191 192 193 194 195	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1</pre>
189 190 191 192 193 194 195 196	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1</pre>
<ol> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> </ol>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1</pre>
189 190 191 192 193 194 195 196 197 198	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation  This function is the function which creates the final</pre>
189 190 191 192 193 194 195 196 197 198	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation  This function is the function which creates the final     result image. This was the first attempt towards</pre>
189 190 191 192 193 194 195 196 197 198	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation  This function is the function which creates the final</pre>
189 190 191 192 193 194 195 196 197 198 199	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation.     Here, I first arrange the coordinates of each pixel in a</pre>
189 190 191 192 193 194 195 196 197 198 199	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation.</pre>
189 190 191 192 193 194 195 196 197 198 199	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation. Here, I first arrange the coordinates of each pixel in a     vertical stack (Line 205 - 207). Then I add xmin and y     min vallues to each of the X values and Y values.</pre>
<ol> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> </ol>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation. Here, I first arrange the coordinates of each pixel in a     vertical stack (Line 205 - 207). Then I add xmin and y     min vallues to each of the X values and Y values. Then I add a third row of just ones to make them into</pre>
<ol> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> </ol>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation      This function is the function which creates the final         result image. This was the first attempt towards         writing a fully vectorised numpy pythonic operation.     Here, I first arrange the coordinates of each pixel in a         vertical stack (Line 205 - 207). Then I add xmin and y         min vallues to each of the X values and Y values.     Then I add a third row of just ones to make them into         individual 3X1 vectors. Using these stacked vectors of</pre>
<ol> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> </ol>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation      This function is the function which creates the final         result image. This was the first attempt towards         writing a fully vectorised numpy pythonic operation.     Here, I first arrange the coordinates of each pixel in a         vertical stack (Line 205 - 207). Then I add xmin and y         min vallues to each of the X values and Y values.     Then I add a third row of just ones to make them into         individual 3X1 vectors. Using these stacked vectors of         individual pixel coordinates, I perform a vector</pre>
<ul> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> <li>201</li> </ul>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation. Here, I first arrange the coordinates of each pixel in a     vertical stack (Line 205 - 207). Then I add xmin and y     min vallues to each of the X values and Y values. Then I add a third row of just ones to make them into     individual 3X1 vectors. Using these stacked vectors of     individual pixel coordinates, I perform a vector     multiplication with the homograhy matrix H. I do this</pre>
<ul> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> <li>201</li> </ul>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation      This function is the function which creates the final         result image. This was the first attempt towards         writing a fully vectorised numpy pythonic operation.     Here, I first arrange the coordinates of each pixel in a         vertical stack (Line 205 - 207). Then I add xmin and y         min vallues to each of the X values and Y values.     Then I add a third row of just ones to make them into         individual 3X1 vectors. Using these stacked vectors of         individual pixel coordinates, I perform a vector     multiplication with the homograhy matrix H. I do this         using the '0' operator. The resulting matrix has the </pre>
<ul> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> <li>201</li> </ul>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation. Here, I first arrange the coordinates of each pixel in a     vertical stack (Line 205 - 207). Then I add xmin and y     min vallues to each of the X values and Y values. Then I add a third row of just ones to make them into     individual 3X1 vectors. Using these stacked vectors of     individual pixel coordinates, I perform a vector multiplication with the homograhy matrix H. I do this     using the '@' operator. The resulting matrix has the     corresponding pixel coordinates of the source image.</pre>
<ul> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> <li>201</li> <li>202</li> </ul>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation. Here, I first arrange the coordinates of each pixel in a     vertical stack (Line 205 - 207). Then I add xmin and y     min vallues to each of the X values and Y values. Then I add a third row of just ones to make them into     individual 3X1 vectors. Using these stacked vectors of     individual pixel coordinates, I perform a vector multiplication with the homograhy matrix H. I do this     using the '@' operator. The resulting matrix has the     corresponding pixel coordinates of the source image.     I extract the pixel values of each of these coordinates</pre>
<ul> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> <li>201</li> <li>202</li> </ul>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation      This function is the function which creates the final         result image. This was the first attempt towards         writing a fully vectorised numpy pythonic operation.     Here, I first arrange the coordinates of each pixel in a         vertical stack (Line 205 - 207). Then I add xmin and y         min vallues to each of the X values and Y values.     Then I add a third row of just ones to make them into         individual 3X1 vectors. Using these stacked vectors of         individual pixel coordinates, I perform a vector     multiplication with the homograhy matrix H. I do this         using the '@' operator. The resulting matrix has the         corresponding pixel coordinates of the source image.     I extract the pixel values of each of these coordinates         using a nested for loop. Basically, I was able to </pre>
<ul> <li>189</li> <li>190</li> <li>191</li> <li>192</li> <li>193</li> <li>194</li> <li>195</li> <li>196</li> <li>197</li> <li>198</li> <li>199</li> <li>200</li> <li>201</li> <li>202</li> </ul>	<pre>def createImageVectorised(self,queueHomography,queueImage):     """[summary]     Attempt #1     Vectorised numpy operation     This function is the function which creates the final     result image. This was the first attempt towards     writing a fully vectorised numpy pythonic operation. Here, I first arrange the coordinates of each pixel in a     vertical stack (Line 205 - 207). Then I add xmin and y     min vallues to each of the X values and Y values. Then I add a third row of just ones to make them into     individual 3X1 vectors. Using these stacked vectors of     individual pixel coordinates, I perform a vector multiplication with the homograhy matrix H. I do this     using the '@' operator. The resulting matrix has the     corresponding pixel coordinates of the source image.     I extract the pixel values of each of these coordinates</pre>

```
204
            nexted for loop. I was able to get stable outputs much
               quicker - 40% faster.
205
206
            Args:
                 queueHomography ([int]): [Index of the homography
207
                    matrix being used to calculate the new image size]
208
                 queueImage ([int]): [Index of the image in the list
                    being used]
209
210
            Returns:
                 [numpy ndarray]: [Returns the final resultant image
211
                    in numpy.ndarray form.]
            .....
212
213
214
            print("processing...")
215
            resultImg,xmin,ymin = self.createBlankImageArray(
               queueHomography, queueImage)
            column,row = np.mgrid[0:resultImg.shape[0],0:resultImg.
216
               shape[1]]
            vector = np.vstack((column.ravel(),row.ravel()))
217
            row = vector[1] + xmin
218
219
            column = vector[0] +ymin
220
            ones = np.ones(len(row))
            vector = np.array([column,row,ones])
221
222
            s=time.time()
223
            resultvector = self.homographies[queueHomography+1]
               @vector
224
            e=time.time()
225
            print("timetake",e-s)
226
            resultvector = resultvector/resultvector[2]
            # resultvector = resultvector[:2,:]
227
            for column in range(0, resultImg.shape[0]):
228
229
                 for row in range(0, resultImg.shape[1]):
                     print("processing" + str(column) + " out of "+
230
                        str(resultImg.shape[0]))
231
                     rangecoordinates=np.array([resultvector[1][(
232
                        column*resultImg.shape[1])+row],resultvector
                        [0][(column*resultImg.shape[1])+row],
                        resultvector[2][(column*resultImg.shape[1])+
                        row]])
233
234
                     if ((rangecoordinates [0] >0) and (rangecoordinates
                        [0] < self.image_sizes[queueImage][1]-1)) and ((
                        rangecoordinates[1]>0) and (rangecoordinates
                        [1] < self.image_sizes[queueImage][0]-1)):</pre>
235
                         resultImg[column][row] = self.
                            weightedPixelValue(rangecoordinates,
                            queueImage)
236
                     else:
237
                         resultImg[column][row] = [255.0,255.0,255.0]
238
239
            return resultImg
```

241242 243def buildImage(self,queueHomography,queueImage,row,column): 244"""[summary] 245246----- Attempt #2 ------247248 Vectorised numpy operation 249250251This function is the function which creates the final 252result image. This was the second attempt towards writing a fully vectorised numpy pythonic operation. 253This function is pretty much the same as the createImage function. The ket difference here is that this function does not have the nester for loop. Instead, I vectorise this entire function using the numpy 254vectorise operation. Using this entire function as a vector, I was able to successfully vectorise the 255whole image building process. 256257Args: 258queueHomography ([int]): [Index of the homography matrix being used to calculate the new image size] queueImage ([int]): [Index of the image in the list 259being used] row ([int]) : [Row value of the pixel being 260considered] column ([int]) : [Column value of the pixel being 261considered] 262263Returns: Does not return any value. It just updates the global 264image variable (self.resultImg). 0.0.0 265266267268rangecoordinates = np.matmul(self.homographies[ queueHomography+1],(float(row+self.xmin),float(column+ self.ymin),1.0)) 269rangecoordinates = rangecoordinates/rangecoordinates/[2] if ((rangecoordinates[0]>0) and (rangecoordinates[0]<self 270.image\_sizes[queueImage][1]-1)) and ((rangecoordinates [1]>0) and (rangecoordinates [1] < self.image\_sizes [ queueImage][0]-1)): pointOne = (int(np.floor(rangecoordinates[1])), int(np 271.floor(rangecoordinates[0]))) pointTwo = (int(np.floor(rangecoordinates[1])),int(np 272.ceil(rangecoordinates[0]))) 273pointThree = (int(np.ceil(rangecoordinates[1])),int( np.ceil(rangecoordinates[0])))

274	<pre>pointFour = (int(np.ceil(rangecoordinates[1])),int(np</pre>
	.floor(rangecoordinates[0])))
275	
276	<pre>pixelValueAtOne = self.images[queueImage][pointOne</pre>
	[0]][pointOne[1]]
277	pixelValueAtTwo = <i>self</i> .images[queueImage][pointTwo
	[0]][pointTwo[1]]
278	pixelValueAtThree = <i>self</i> .images[queueImage][
	pointThree[0]][pointThree[1]]
279	pixelValueAtFour = <i>self</i> .images[queueImage][pointFour
	[0]][pointFour[1]]
280	
281	<pre>weightAtOne = 1/np.linalg.norm(pixelValueAtOne -</pre>
	rangecoordinates)
282	<pre>weightAtTwo = 1/np.linalg.norm(pixelValueAtTwo-</pre>
	rangecoordinates)
283	<pre>weightAtThree = 1/np.linalg.norm(pixelValueAtThree -</pre>
	rangecoordinates)
284	<pre>weightAtFour = 1/np.linalg.norm(pixelValueAtFour -</pre>
	rangecoordinates)
285	
286	<pre>self.resultImg[column][row] = ((weightAtOne*</pre>
	<pre>pixelValueAtOne) + (weightAtTwo*pixelValueAtTwo) +</pre>
	(weightAtThree*pixelValueAtThree) + (weightAtFour
	<pre>*pixelValueAtFour))/(weightAtFour+weightAtThree+</pre>
	weightAtTwo+weightAtOne)
287	else:
288	
289	self.resultImg[column][row] = [255.0,255.0,255.0]
290	
291	<pre>def vectoriseOperations(self,queueHomography,queueImage):</pre>
292	"""[summary]
293	Attempt #2 Continued
294	
295	Vectorised numpy operation
296	
297	
298 200	This function is the extension of the phone function
299	This function is the extension of the above function -
	buildImage. This is the function which vectorises the
300	entire buildImage function. In this function, I stack a list which contains all the
300	
	pixel coordinates in the blank image. I feed this entire list to the vectorised function.
301	This was a successful vectorisation operation however the
301	RAM utilization peaked to a hundred percent. The
	laptop froze and I could not run this further.
302	raptop 11026 and 1 courd not full this fulther.
$\frac{302}{303}$	Args:
$303 \\ 304$	queueHomography ([int]): [Index of the homography
001	matrix being used to calculate the new image size]
305	queueImage ([int]): [Index of the image in the list
000	being used]

```
307
            Returns:
                 [numpy ndarray]: [Returns the final resultant image
308
                    in numpy.ndarray form.]
            .....
309
            self.resultImg,self.xmin,self.ymin = self.
310
                createBlankImageArray(queueHomography,queueImage)
            length = self.resultImg.shape[0]*self.resultImg.shape[1]
311
312
            queueHomography = [queueHomography]*length
313
            queueImage = [queueImage]*length
            vectoriseOperation = np.vectorize(self.buildImage)
314
            row,column = np.mgrid[0:self.resultImg.shape[1],0:self.
315
                resultImg.shape[0]]
            point = np.vstack((row.ravel(),column.ravel()))
316
317
            row = point[0]
318
            column = point[1]
319
            #print(point)
320
            print("processing...")
321
            vectoriseOperation(queueHomography,queueImage,row,column)
            return self.resultImg
322
323
324
325
326
327
328
329
        def objectMatrixFunction(self,queue):
            0.0.0
330
            [We construct the B Matrix with dimension 8X1]
331
332
333
            Args:
                 queue ([int]): [This is the index number of the list
334
                    which has the coordinates of the roI for the
                    object picture]
            .....
335
336
            self.objectMatrix = np.zeros((8,1))
337
            for i in range(len(self.roiRealWorld[queue])):
338
339
                 self.objectMatrix[(2*i)][0] = self.roiRealWorld[queue
                    ][i][0]
340
                 self.objectMatrix[(2*i)+1][0] = self.roiRealWorld[
                    queue][i][1]
341
342
        def parameterMatrixFunction(self,queue,objectQueue):
343
            0.0.0
            We construct the A Matrix with dimension 8X8 and then we
344
                 calculate the inverse of A matrix needed for the
                homography calculation]
345
346
            Args:
347
                 queue ([int]): [This is the index number of the list
                    which has the coordinates of the roI for the
                    destination picture]
```

```
objectQueue ([int]): [This is the index number of the
348
                     list which has the coordinates of the roI for the
                     Object picture]
            0.0.0
349
            self.parameterMatrix=np.zeros((8,8))
350
351
352
            for i in range(4):
                self.parameterMatrix[2*i][0] = self.roiList[queue][i
353
                    ][0]
                self.parameterMatrix[2*i][1] = self.roiList[queue][i
354
                   ][1]
355
                 self.parameterMatrix[2*i][2] = 1.0
                 self.parameterMatrix[2*i][3] = 0.0
356
                self.parameterMatrix[2*i][4] = 0.0
357
                 self.parameterMatrix[2*i][5] = 0.0
358
359
                self.parameterMatrix[2*i][6] = (-1)*(self.roiList[
                    queue][i][0])*(self.roiRealWorld[objectQueue][i
                    ][0])
                 self.parameterMatrix[2*i][7] = (-1)*(self.roiList[
360
                    queue][i][1])*(self.roiRealWorld[objectQueue][i
                    ][0])
361
                 self.parameterMatrix[(2*i) + 1][0] = 0.0
                 self.parameterMatrix[(2*i) + 1][1] = 0.0
362
363
                 self.parameterMatrix[(2*i) + 1][2] = 0.0
364
                 self.parameterMatrix[(2*i) + 1][3] = self.roiList[
                    queue][i][0]
365
                self.parameterMatrix[(2*i) + 1][4] = self.roiList[
                    queue][i][1]
366
                 self.parameterMatrix[(2*i) + 1][5] = 1.0
367
                 self.parameterMatrix[(2*i) + 1][6] = (-1)*(self.
                    roiList[queue][i][0])*(self.roiRealWorld[
                    objectQueue][i][1])
368
                 self.parameterMatrix[(2*i) + 1][7] = (-1)*(self.
                    roiList[queue][i][1])*(self.roiRealWorld[
                    objectQueue][i][1])
369
            self.parameterMatrixI = np.linalg.pinv(self.
370
               parameterMatrix)
371
372
        def calculateHomography(self):
            0.0.0
373
374
            [We calculate the homography matrix here. Once we have
               the values of the matrix, we rearrange them into a 3X3
                matrix.]
375
            .....
376
            homographyI = np.matmul(self.parameterMatrixI,self.
377
               objectMatrix)
            homography = np.zeros((3,3))
378
379
380
            homography[0][0] = homographyI[0]
            homography[0][1] = homographyI[1]
381
382
            homography[0][2] = homographyI[2]
```

```
383
            homography[1][0] = homographyI[3]
            homography[1][1] = homographyI[4]
384
385
            homography[1][2] = homographyI[5]
386
            homography[2][0] = homographyI[6]
            homography[2][1] = homographyI[7]
387
            homography [2] [2] = 1.0
388
389
            self.homographies.append(homography)
            homography = np.linalg.pinv(homography)
390
391
            homography = homography/homography[2][2]
392
            self.homographies.append(homography)
393
394
395
        def projectiveDistortionHomography(self,queueImage):
396
            """[summary]
397
398
            Calculate the homography matrix to eliminate projective
                distortion
399
400
            Args:
401
                 queueImage ([int]): [Index of the image in the list
                    being used]
402
            Calculates the Homography matrix and appends it to the
403
               global homography list.
            .....
404
405
            vanishingPointOne = np.cross(np.cross(self.roiList[
406
                queueImage][0], self.roiList[queueImage][1]), np.cross(
                self.roiList[queueImage][2], self.roiList[queueImage
                1[3]))
407
            vanishingPointTwo = np.cross(np.cross(self.roiList[
                queueImage][0], self.roiList[queueImage][2]), np.cross(
                self.roiList[queueImage][1], self.roiList[queueImage
                ][3]))
408
            vanishingLine = np.cross((vanishingPointOne/
409
                vanishingPointOne[2]),(vanishingPointTwo/
                vanishingPointTwo[2]))
410
411
            projectiveDHomography = np.zeros((3,3))
            projectiveDHomography[2] = vanishingLine/vanishingLine[2]
412
413
            projectiveDHomography[0][0] = 1
            projectiveDHomography[1][1] = 1
414
            self.homographies.append(projectiveDHomography)
415
416
            inverseH = np.linalg.pinv(projectiveDHomography)
            self.homographies.append(inverseH/inverseH[2][2])
417
418
419
420
421
        def affineDistortionHomography(self,queueImage):
422
            """[summary]
423
            Calculate the homography matrix to eliminate affine
                distortion
```

```
425
            Args:
426
                 queueImage ([int]): [Index of the image in the list
                    being used]
427
            Calculates the Homography matrix and appends it to the
428
               global homography list.
            .....
429
430
            templist = []
            temppoints = []
431
432
433
            for i in range(4):
434
                 tempvalue = np.dot(self.homographies[0], self.roiList[
                    queueImage][i])
435
                 tempvalue = tempvalue/tempvalue[2]
436
                 temppoints.append(tempvalue)
437
438
            print(temppoints)
            ortholinePairOne = np.cross(temppoints[0],temppoints[1])
439
            ortholinePairTwo = np.cross(temppoints[0],temppoints[2])
440
            ortholinePairThree = np.cross(temppoints[0],temppoints
441
                [3])
            ortholinePairFour = np.cross(temppoints[1],temppoints[2])
442
443
            templist.append(ortholinePairOne)
444
            templist.append(ortholinePairTwo)
            templist.append(ortholinePairThree)
445
            templist.append(ortholinePairFour)
446
447
            for i,element in enumerate(templist):
448
449
                 #print(element)
                 #print(element[2])
450
                 templist[i] = element/element[2]
451
452
453
            matrixAT = []
            matrixAT.append([templist[0][0]*templist[1][0],templist
454
                [0][0]*templist[1][1]+templist[0][1]*templist[1][0]])
            matrixAT.append([templist[2][0]*templist[3][0],templist
455
                [2][0]*templist[3][1]+templist[2][1]*templist[3][0]])
            matrixAT = np.asarray(matrixAT)
456
457
            matrixAT = np.linalg.pinv(matrixAT)
            matrixA = []
458
459
            matrixA.append([-templist[0][1]*templist[1][1]])
            matrixA.append([-templist[2][1]*templist[3][1]])
460
            matrixA = np.asarray(matrixA)
461
462
            matrixS = np.dot(matrixAT,matrixA)
463
            matrixSRearranged = np.zeros((2,2))
464
465
            matrixSRearranged[0][0] = matrixS[0]
466
            matrixSRearranged[0][1] = matrixS[1]
467
468
            matrixSRearranged[1][0] = matrixS[1]
469
            matrixSRearranged[1][1] = 1
470
```

```
471
            v,lambdamatrix,q = np.linalg.svd(matrixSRearranged)
472
473
            lambdavalue = np.sqrt(np.diag(lambdamatrix))
474
            Hmatrix = np.dot(np.dot(v,lambdavalue),v.transpose())
475
            affineHomography=np.zeros((3,3))
476
477
            affineHomography[0][0] = Hmatrix[0][0]
            affineHomography[0][1] = Hmatrix[0][1]
478
            affineHomography[1][0] = Hmatrix[1][0]
479
            affineHomography[1][1] = Hmatrix[1][1]
480
481
            affineHomography[2][2] = 1
482
483
            inverseH = np.linalg.pinv(affineHomography)
484
485
            inverseH = np.dot(inverseH, self.homographies[0])
486
            self.homographies.append(inverseH)
487
            inverseH = np.linalg.pinv(inverseH)
488
            self.homographies.append(inverseH/inverseH[2][2])
489
490
        def oneStepDistortionHomography(self,queueImage):
            """[summary]
491
            Calculate the homography matrix to eliminate both
492
               projective and affine distortion
493
494
            Args:
                 queueImage ([int]): [Index of the image in the list
495
                   being used]
496
            Calculates the Homography matrix and appends it to the
497
               global homography list.
            .....
498
            matrixA = []
499
            matrixAT = []
500
            templist=[]
501
            templist.append(np.cross(self.roiList[queueImage][0],self
502
                .roiList[queueImage][1]))
            templist.append(np.cross(self.roiList[queueImage][1],self
503
               .roiList[queueImage][3]))
            templist.append(np.cross(self.roiList[queueImage][1],self
504
               .roiList[queueImage][3]))
            templist.append(np.cross(self.roiList[queueImage][3], self
505
               .roiList[queueImage][2]))
            templist.append(np.cross(self.roiList[queueImage][3],self
506
               .roiList[queueImage][2]))
            templist.append(np.cross(self.roiList[queueImage][2], self
507
                .roiList[queueImage][0]))
            templist.append(np.cross(self.roiList[queueImage][2],self
508
                .roiList[queueImage][0]))
            templist.append(np.cross(self.roiList[queueImage][0],self
509
               .roiList[queueImage][1]))
            templist.append(np.cross(self.roiList[queueImage][0], self
510
               .roiList[queueImage][3]))
511
            templist.append(np.cross(self.roiList[queueImage][1],self
```

```
.roiList[queueImage][2]))
512
513
            for i,element in enumerate(templist):
514
                templist[i] = element/element[2]
515
            for i in range(0,10,2):
516
517
                matrixAT.append([templist[i][0]*templist[i+1][0],(
                    templist[i][0]*templist[i+1][1]+templist[i][1]*
                    templist[i+1][0])/2,templist[i][1]*templist[i
                    +1][1],(templist[i][0]*templist[i+1][2]+templist[i
                    ][2]*templist[i+1][0])/2,(templist[i][1]*templist[
                    i+1][2]+templist[i][2]*templist[i+1][1])/2])
                matrixA.append([-templist[i][2]*templist[i+1][2]])
518
519
520
            matrixAT = np.asarray(matrixAT)
521
            matrixA = np.asarray(matrixA)
522
            matrixS = np.dot(np.linalg.pinv(matrixAT),matrixA)
            matrixS = matrixS/np.max(matrixS)
523
524
525
            matrixSRearranged = np.zeros((2,2))
            matrixSRearranged[0][0] = matrixS[0]
526
527
            matrixSRearranged[0][1] = matrixS[1] * 0.5
            matrixSRearranged[1][0] = matrixS[1] * 0.5
528
529
            matrixSRearranged[1][1] = matrixS[2]
530
            matrixST = np.array([matrixS[3]*0.5,matrixS[4]*0.5])
            v, lambdamatrix, q = np.linalg.svd(matrixSRearranged)
531
            lambdavalue = np.sqrt(np.diag(lambdamatrix))
532
            Hmatrix = np.dot(np.dot(v,lambdavalue),v.transpose())
533
            Vmatrix = np.dot(np.linalg.pinv(Hmatrix),matrixST)
534
535
            onestepHomography =np.zeros((3,3))
536
            onestepHomography[0][0] = Hmatrix[0][0]
537
            onestepHomography[0][1] = Hmatrix[0][1]
538
            onestepHomography[1][0] = Hmatrix[1][0]
539
            onestepHomography[1][1] = Hmatrix[1][1]
540
            onestepHomography[2][0] = Vmatrix[0]
541
            onestepHomography[2][1] = Vmatrix[1]
542
            onestepHomography[2][2]=1
543
544
545
            inverseH = np.linalg.pinv(onestepHomography)
            self.homographies.append(inverseH)
546
            inverseH = np.linalg.pinv(inverseH)
547
            self.homographies.append(inverseH/inverseH[2][2])
548
549
550
551
552
553
    if __name__ == "__main__":
554
555
            0.0.0
556
            The code begins here. Make sure the input image paths are
557
                properly inserted.
```

```
.....
559
560
        tester = removeDistortion(['hw3_Task1_Images/Images/1.jpg','
561
           hw3_Task1_Images/Images/2.jpg','hw3_Task1_Images/Images/3.
           ipg'])
562
        tester.getROIFromUser()
        for i in range(0,3):
563
            tester.objectMatrixFunction(i)
564
            tester.parameterMatrixFunction(i,i)
565
566
            tester.calculateHomography()
567
            resultImg = tester.createImage(0,i)
            cv.imwrite("ptp" +str(i)+".jpg",resultImg)
568
569
570
        tester.getROIFromUser()
571
572
        for i in range(0,3):
573
            tester.projectiveDistortionHomography(i)
            resultImg = tester.createImage(0,i)
574
575
            # resultImg = tester.createImageVectorised(0,0)
            cv.imwrite('1' +str(i)+'.jpg',resultImg)
576
            tester.affineDistortionHomography(i)
577
578
            resultImg = tester.createImage(2,i)
579
            cv.imwrite('2' +str(i)+'.jpg',resultImg)
580
            tester.oneStepDistortionHomography(i)
            resultImg = tester.createImage(4,i)
581
582
            cv.imwrite('3' +str(i)+'.jpg',resultImg)
583
        ######Custom Input Images########
584
585
        tester = removeDistortion(['hw3_Task1_Images/Images/sn.jpg','
586
           hw3_Task1_Images/Images/laptop.jpg'])
587
        tester.getROIFromUser()
        for i in range(0,2):
588
            tester.objectMatrixFunction(i)
589
590
            tester.parameterMatrixFunction(i,i)
            tester.calculateHomography()
591
592
            resultImg = tester.createImage(0,i)
            cv.imwrite("ptp" +str(i)+".jpg",resultImg)
593
594
        tester.getROIFromUser()
595
596
        for i in range(0,2):
597
            tester.projectiveDistortionHomography(i)
598
            resultImg = tester.createImage(0,i)
599
600
            # resultImg = tester.createImageVectorised(0,0)
            cv.imwrite('1' +str(i)+'.jpg',resultImg)
601
602
            tester.affineDistortionHomography(i)
            resultImg = tester.createImage(2,i)
603
            cv.imwrite('2' +str(i)+'.jpg',resultImg)
604
605
            tester.oneStepDistortionHomography(i)
            resultImg = tester.createImage(4,i)
606
607
            cv.imwrite('3' +str(i)+'.jpg',resultImg)
```