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$_1, y_1)and(x'_1, y'_1), (x_2, y_2)and(x'_2, y'_2), (x_3, y_3)and(x'_3, y'_3)Writingthe8equationsi$

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

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 **A** at (x,y) on the plane **PQRS** we have corresponding point **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.

$$l_{vl} \rightarrow l_{\infty}$$

- **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:

$$vp_1 = l_1 X l_2$$

The same can be applied to a set of four more points which lie on a pair (two each) of parallel lines (unique pair) l_3 and l_4 to get the second vanishing point vp_2. Where we have the relation:

$$vp_2 = l_3 X l_4$$

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:

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathrm{vl}_1 & \mathrm{vl}_2 & \mathrm{vl}_3 \end{bmatrix}$$

Where we have $l_{vl} = [\mathrm{vl}_1 \; vl_2 \; vl_3]^T$

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 θ 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:

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

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:

$$s_a ad + s_b(ae + bd) + be = 0$$

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:

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

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:

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

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

$$S = AA^T$$

further,

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

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 attemtps were made.

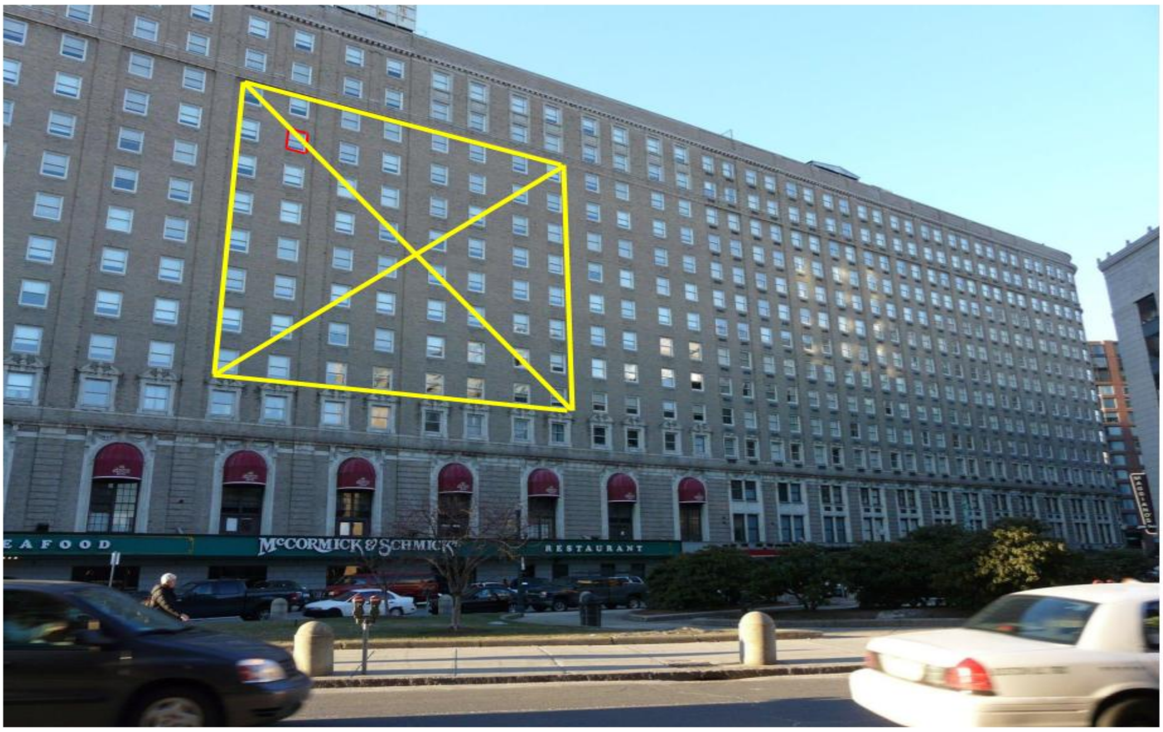

Figure 1: **Input Image**

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**

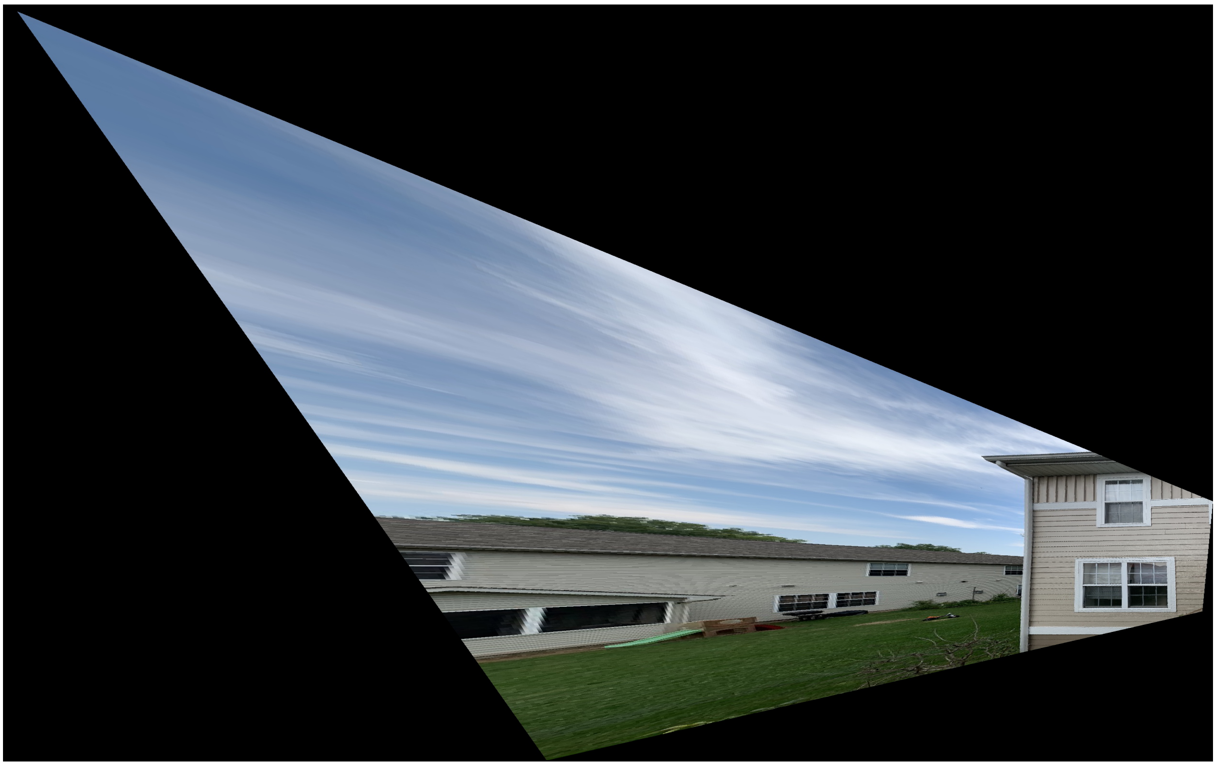

Figure 7: **Point to Point Correspondence Method**

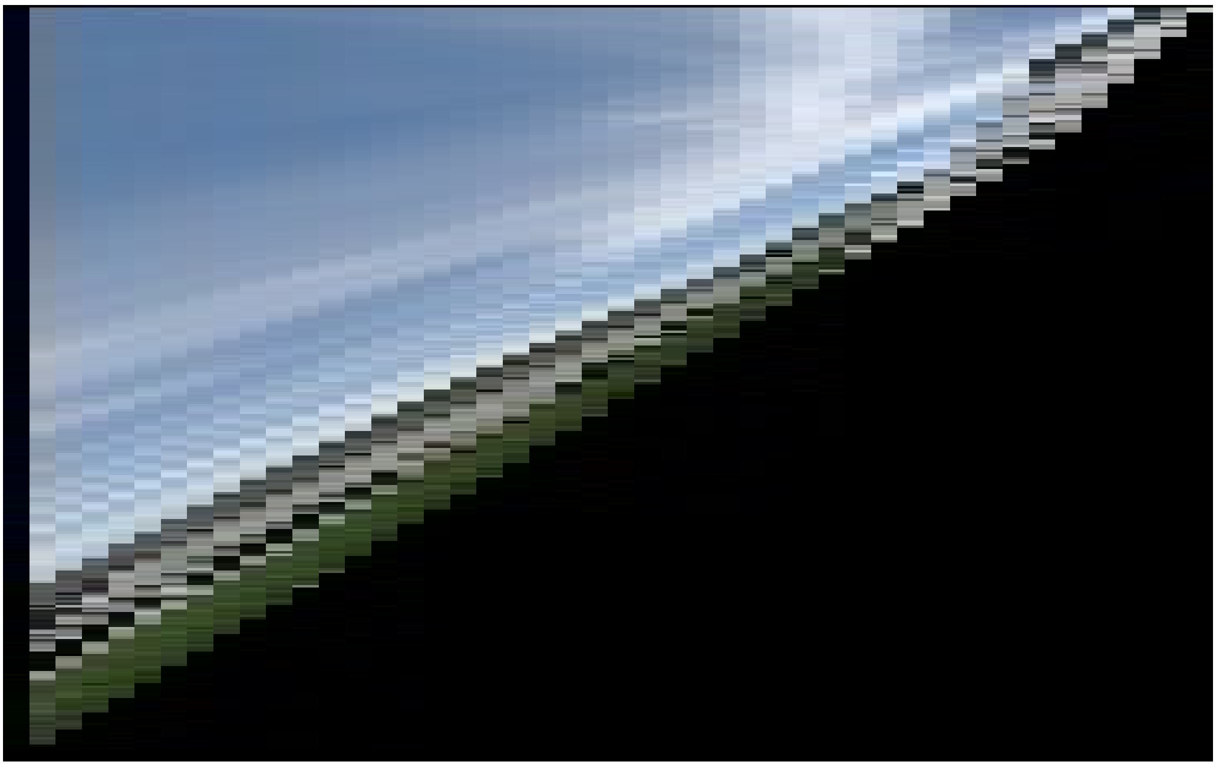

Figure 8: **Two Step Method - Removing Projective Distortion Alone**

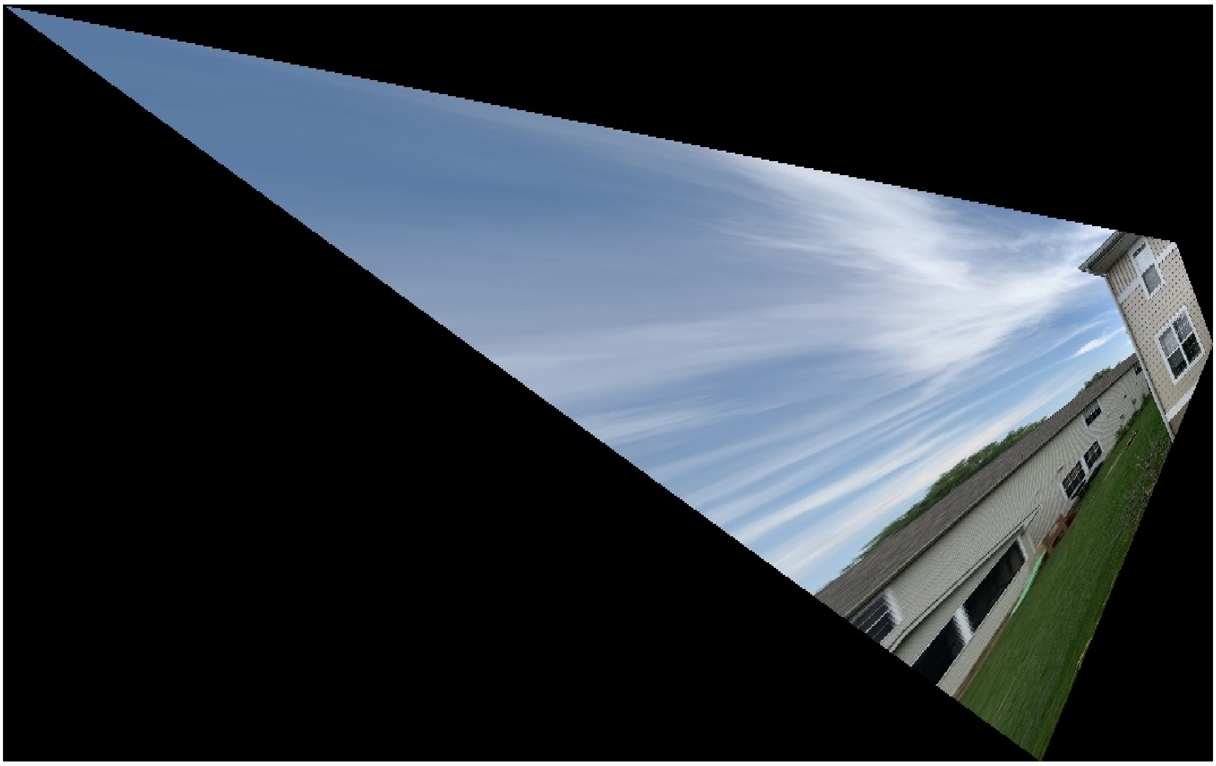

Figure 9: **Two Step Method - Removing both Projective and Affine Distortion**

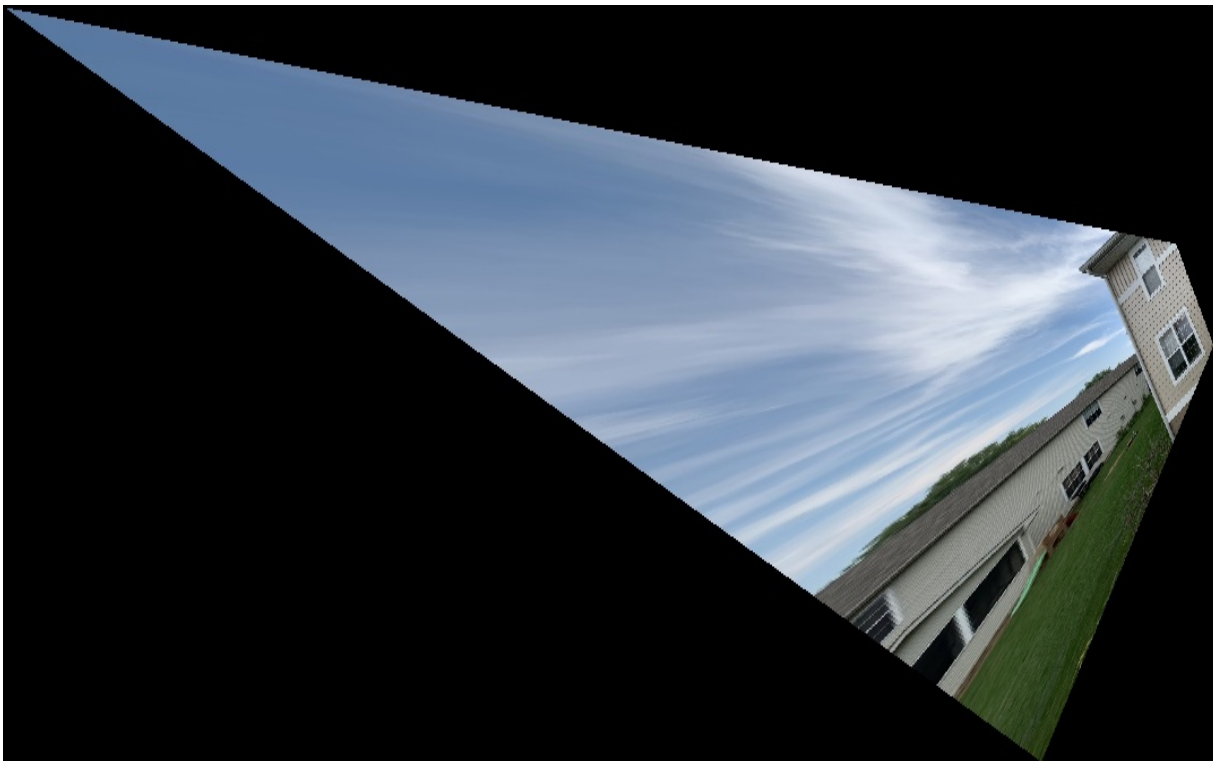

Figure 10: **One Step Method - Removing both Projective and Affine Distortion**

Figure 11: **Input Image**

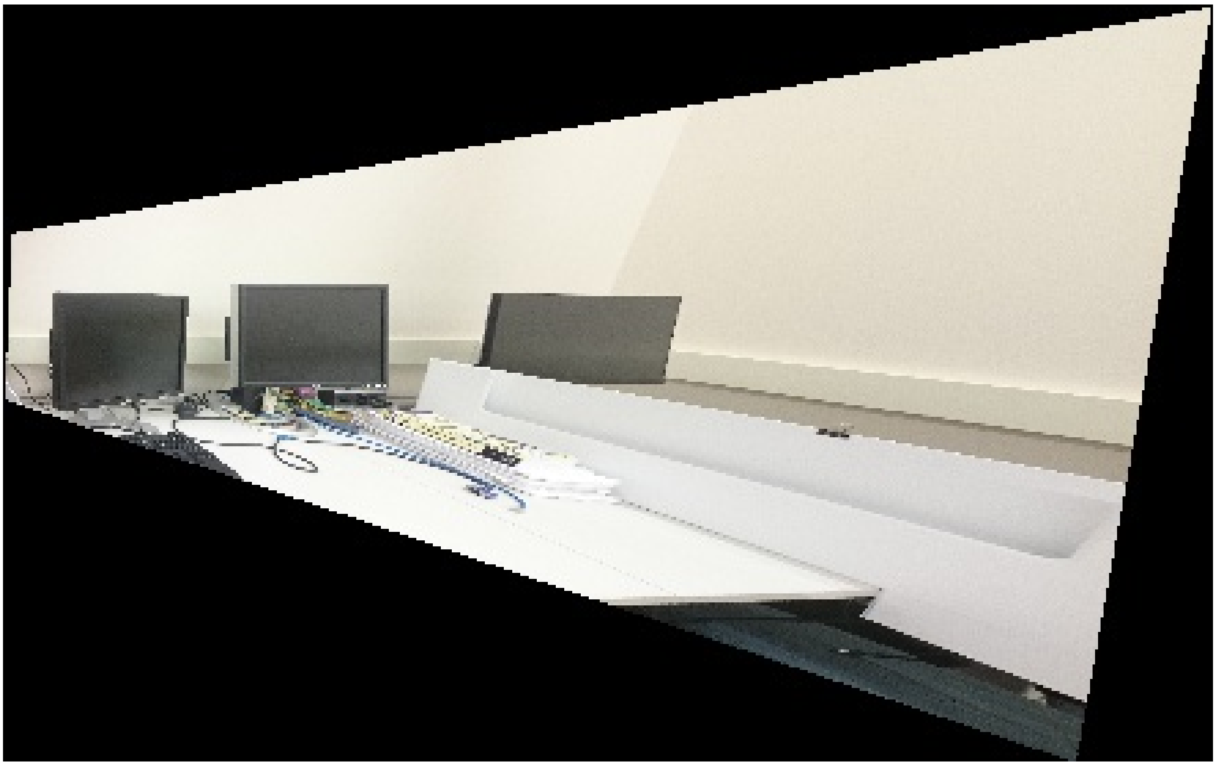

Figure 12: **Point to Point Correspondence Method**

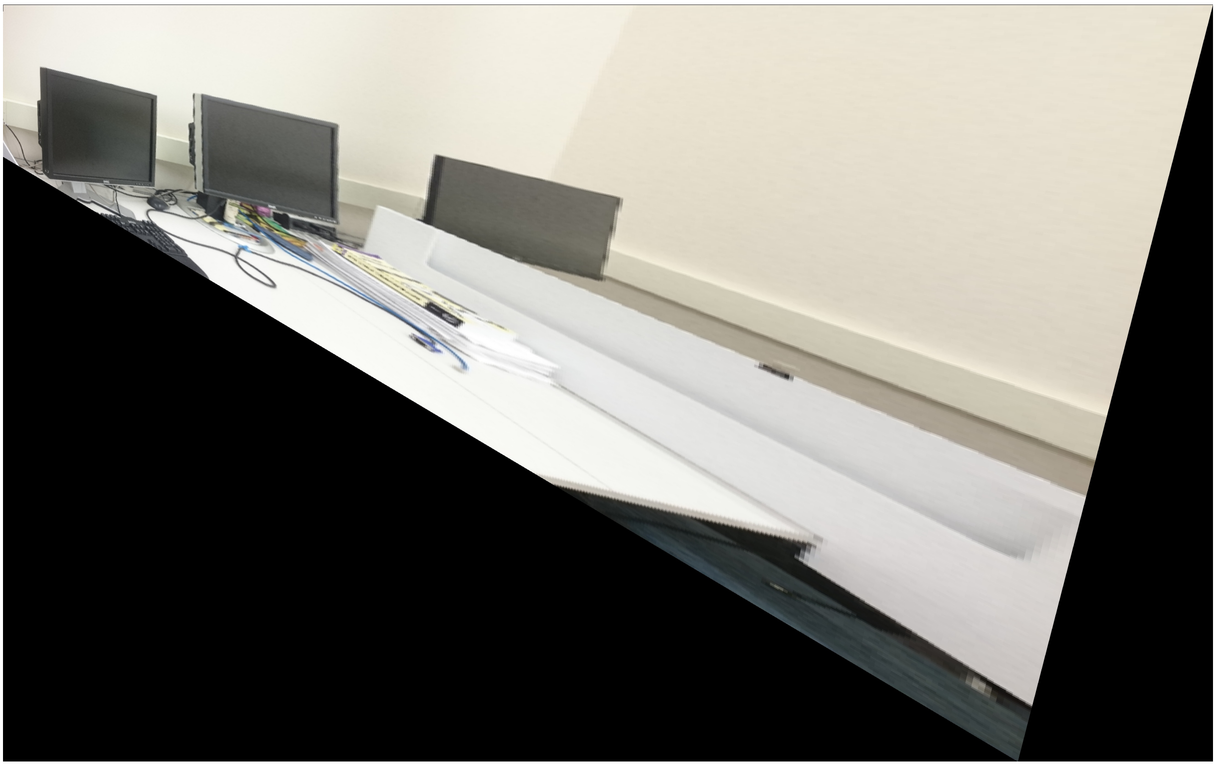

Figure 13: **Two Step Method - Removing Projective Distortion Alone**

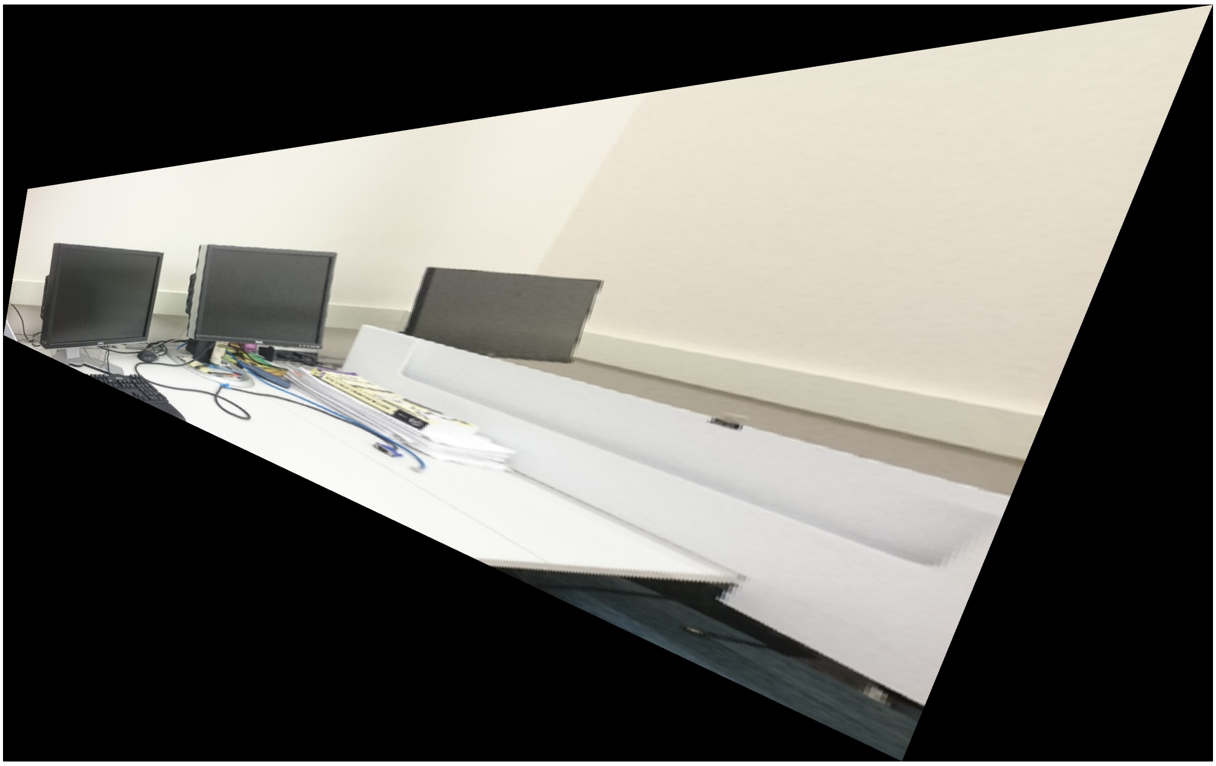

Figure 14: **Two Step Method - Removing both Projective and Affine Distortion**

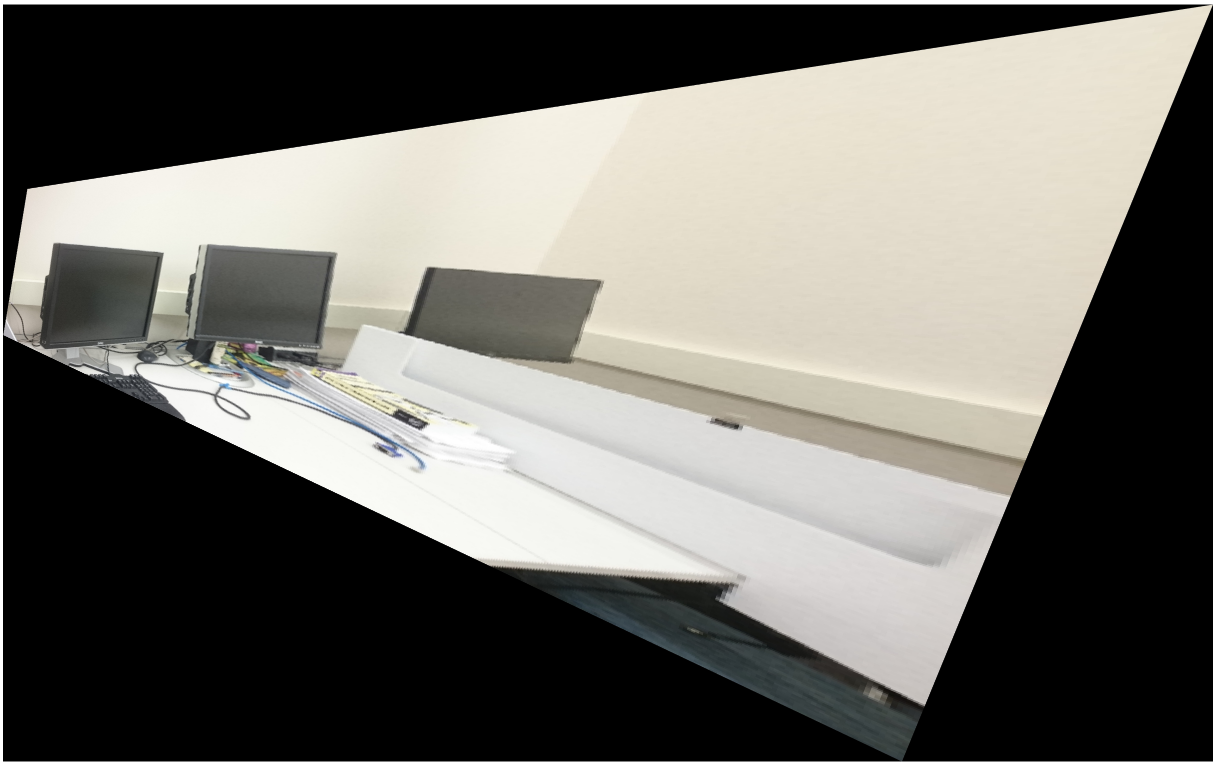

Figure 15: **One Step Method - Removing both Projective and Affine Distortion**

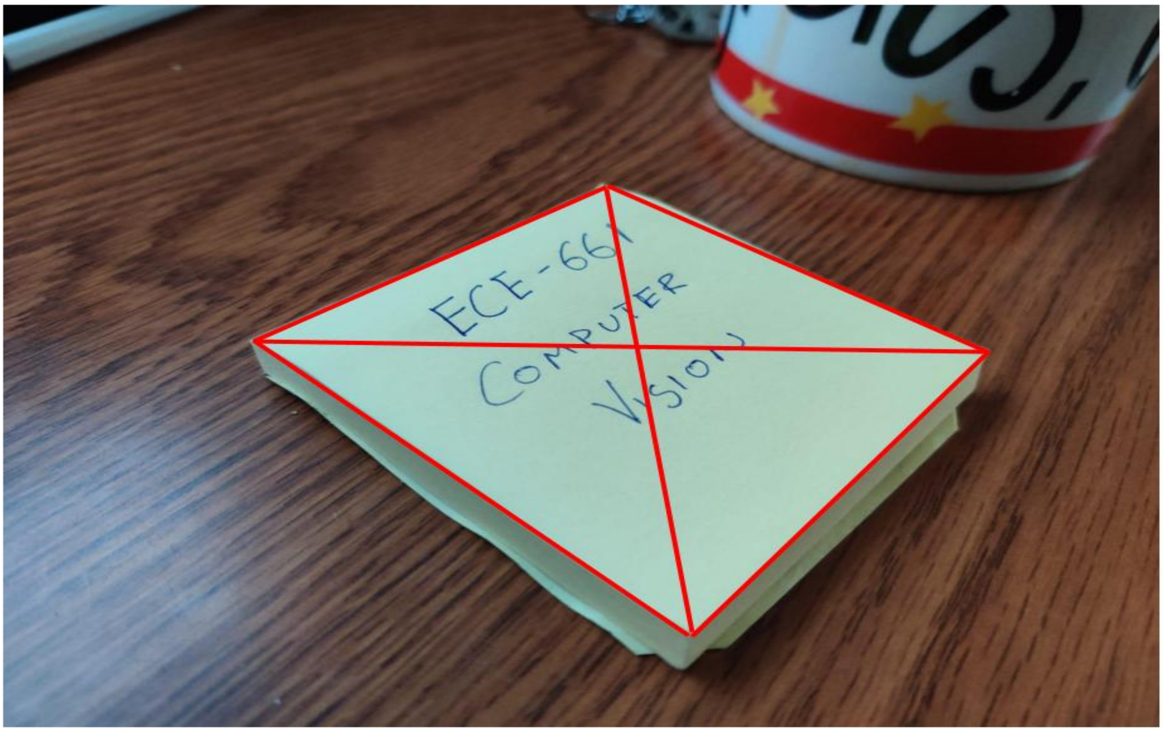

Figure 16: **Input Image**

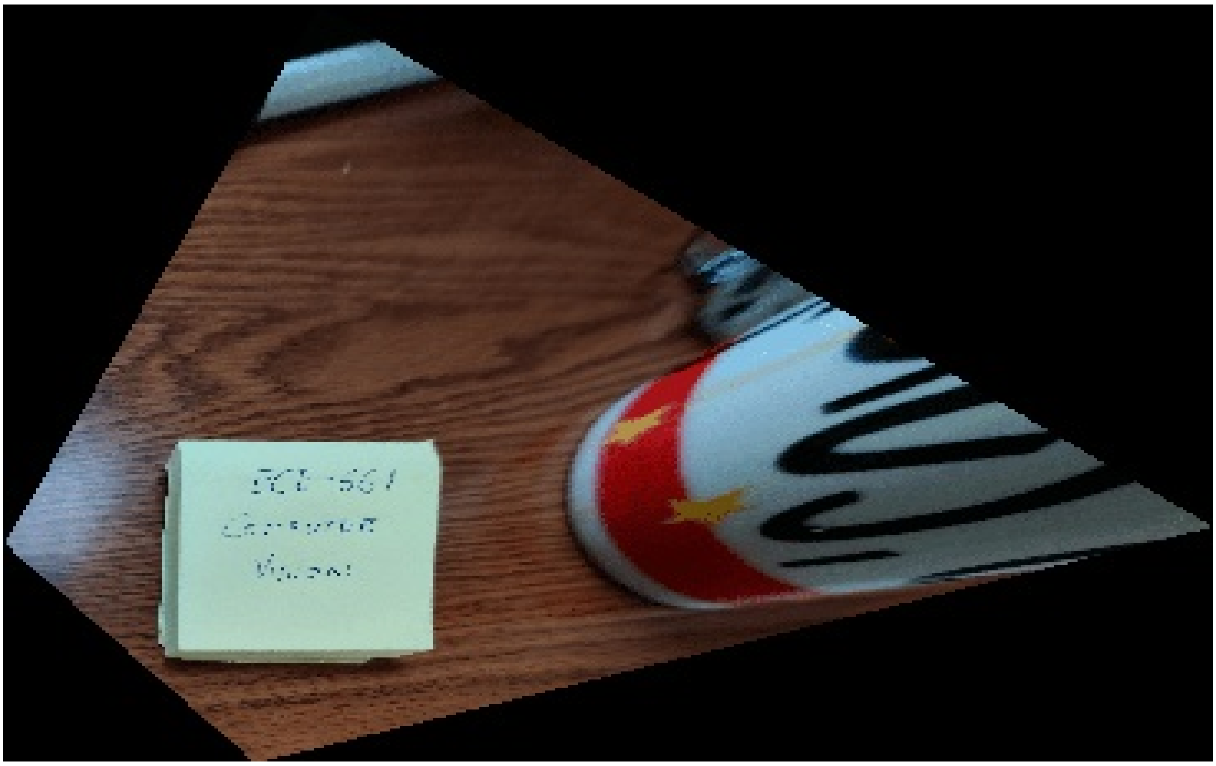

Figure 17: **Point to Point Correspondence Method**

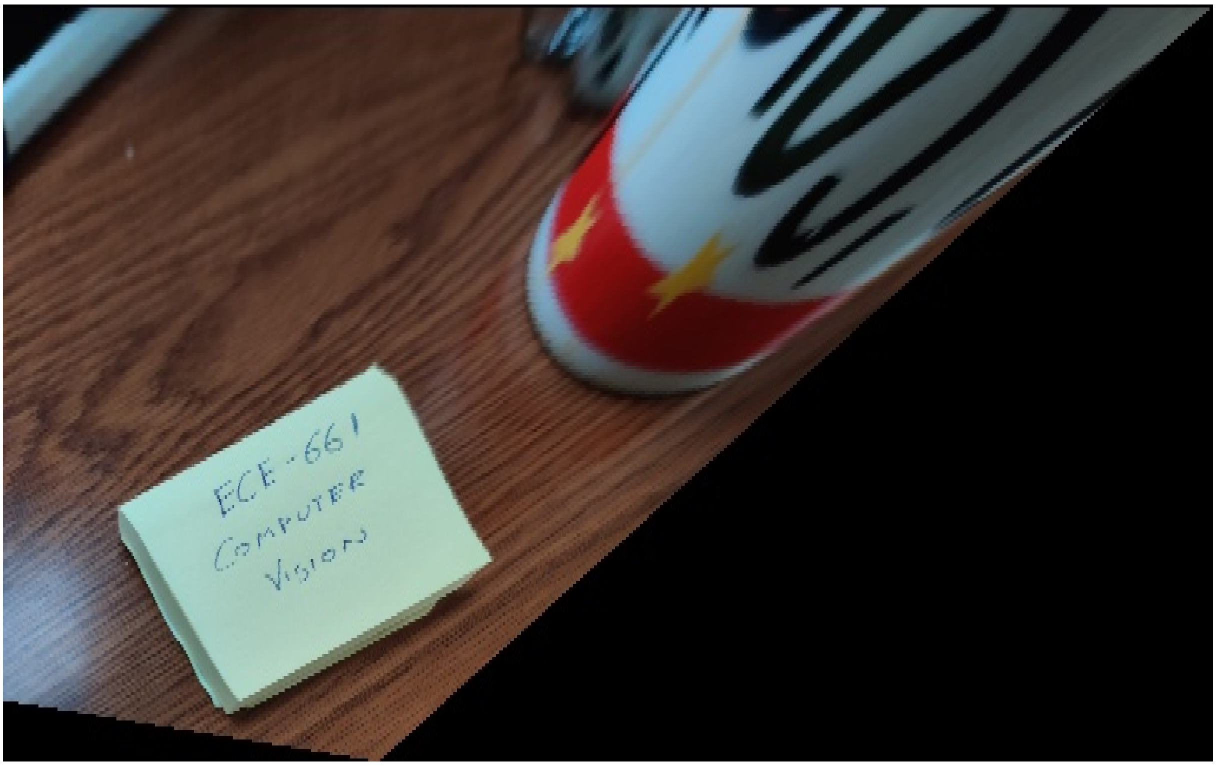

Figure 18: **Two Step Method - Removing Projective Distortion Alone**

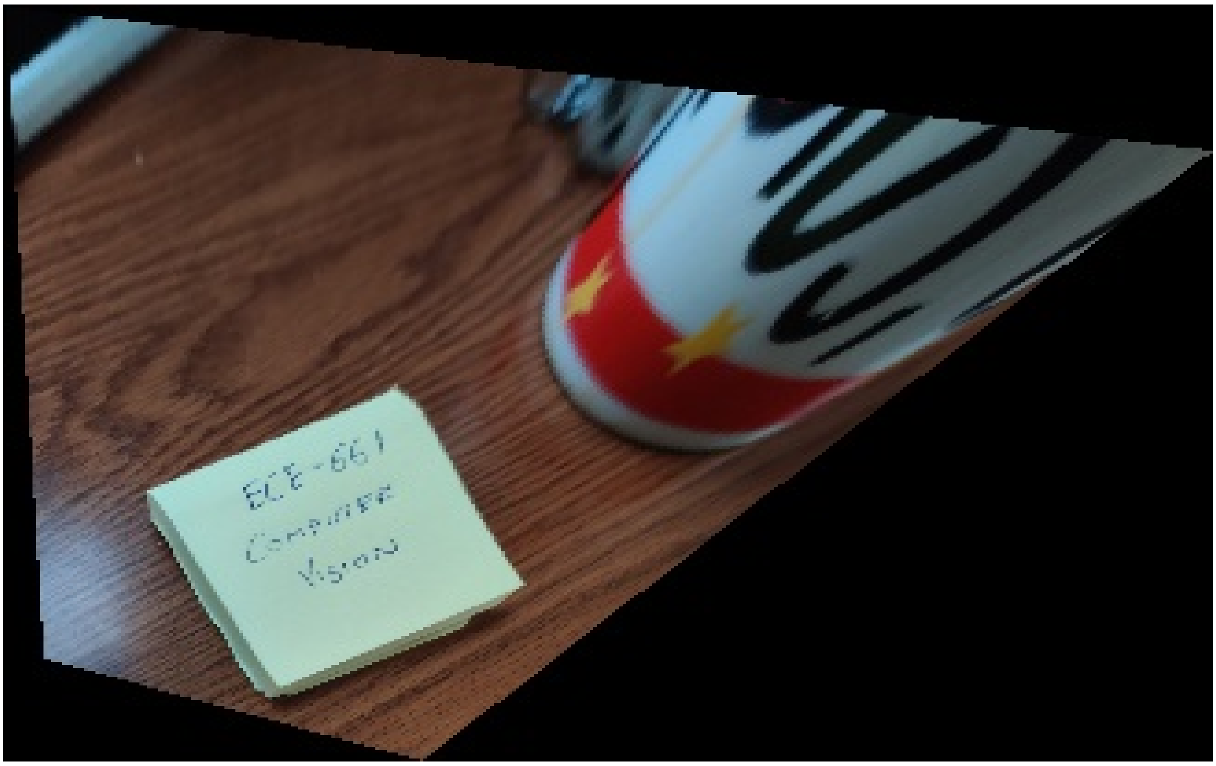

Figure 19: **Two Step Method - Removing both Projective and Affine Distortion**

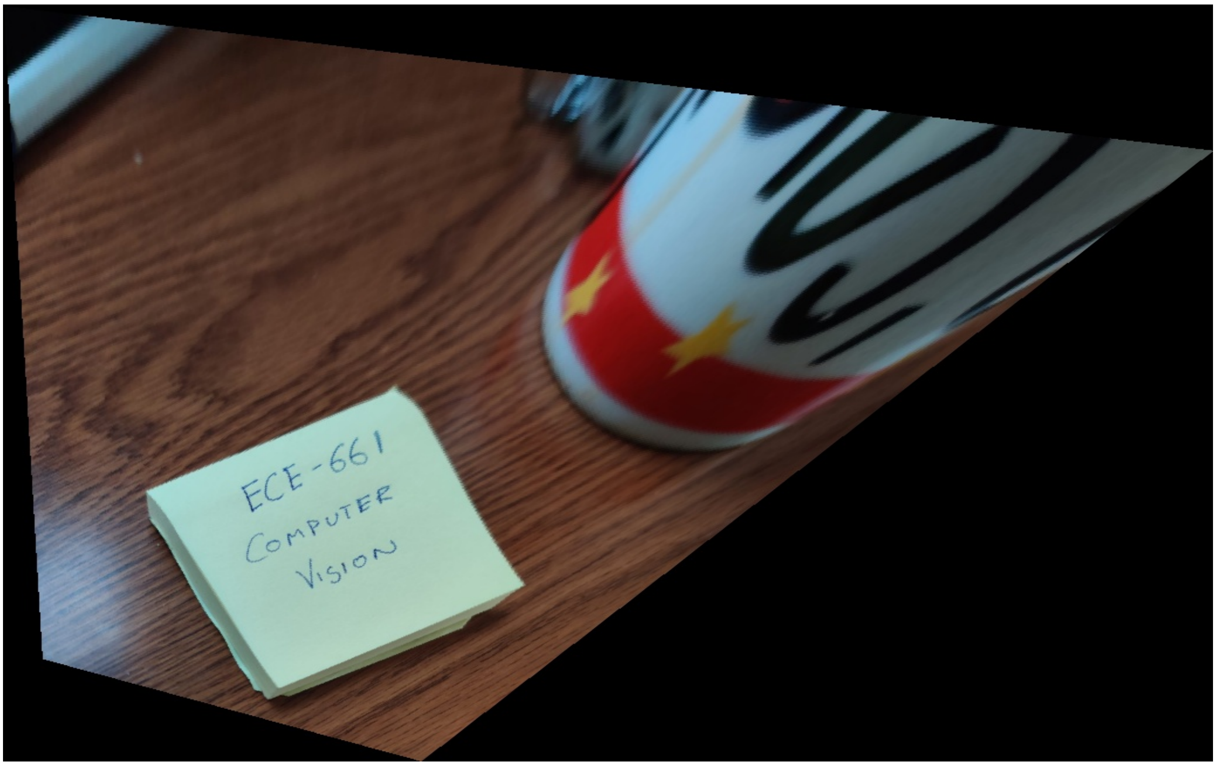

Figure 20: **One Step Method - Removing both Projective and Affine Distortion**

Figure 21: **Input Image**

Figure 22: **Point to Point Correspondence Method**

Figure 23: **Two Step Method - Removing Projective Distortion Alone**

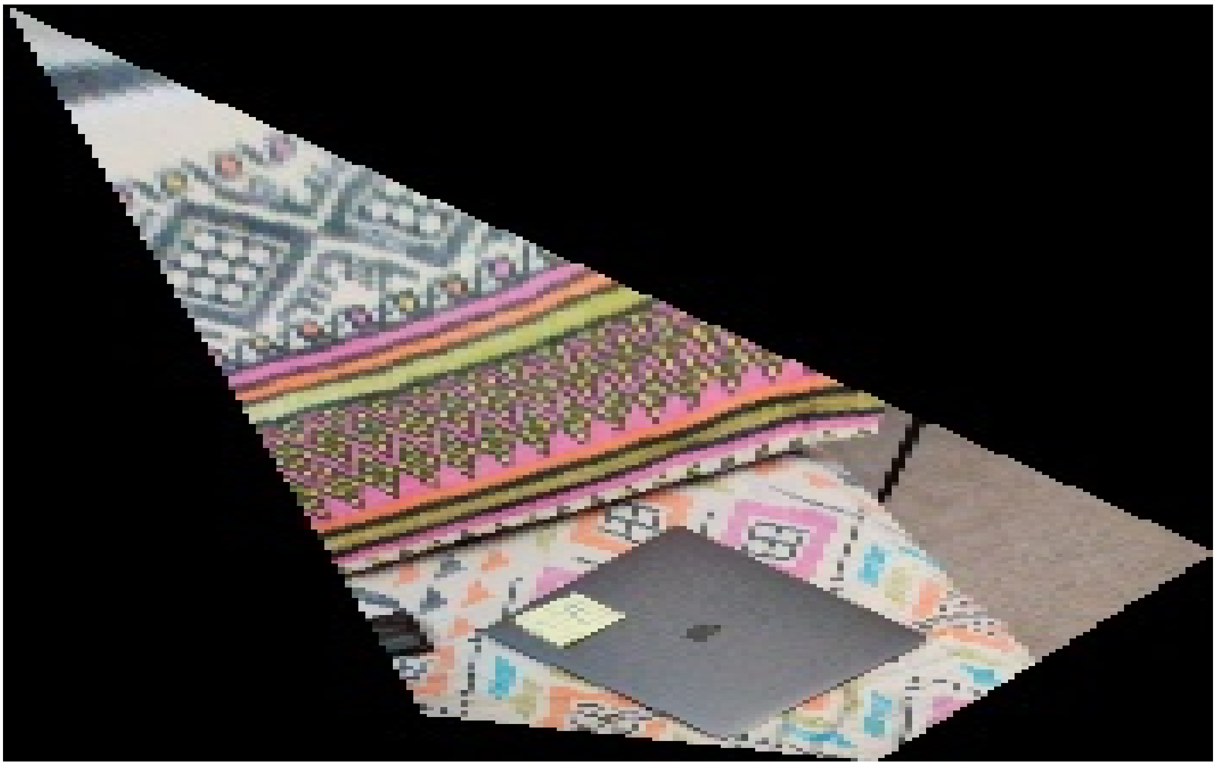

Figure 24: **Two Step Method - Removing both Projective and Affine Distortion**

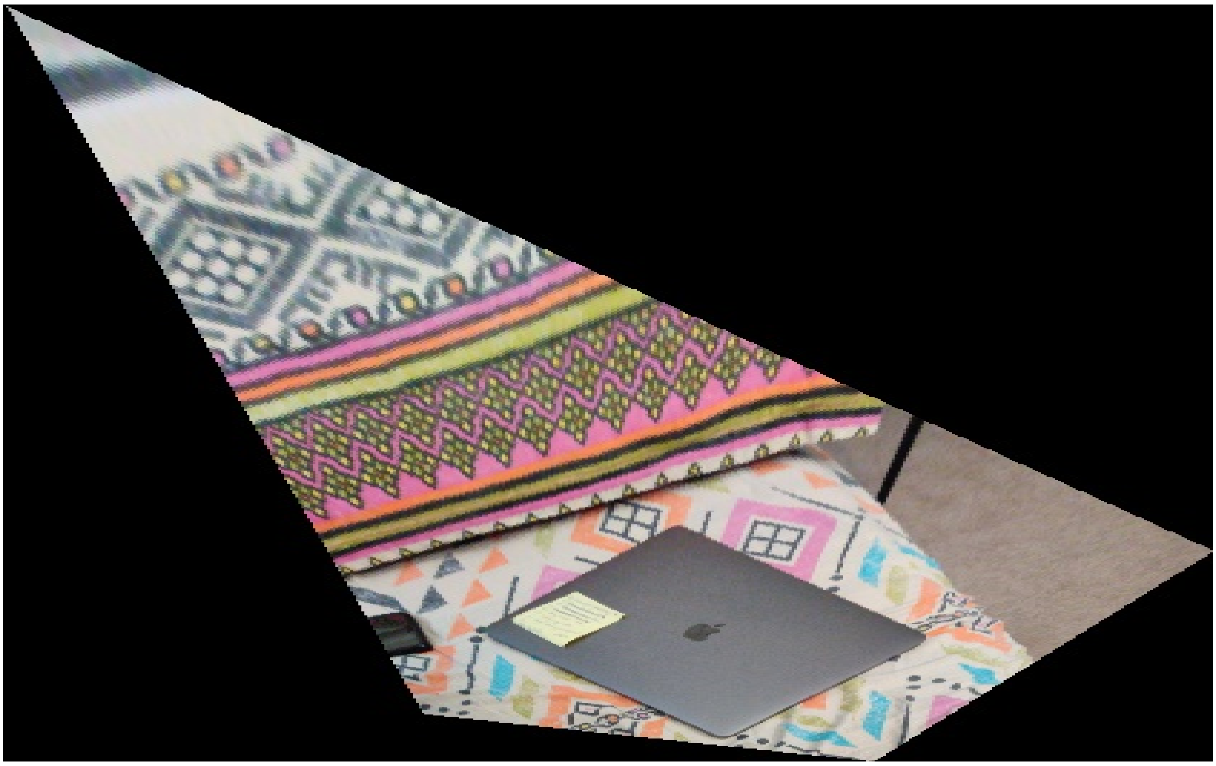

Figure 25: **One Step Method - Removing both Projective and Affine Distortion**

Source Code

```
"""
Computer Vision - Purdue University - Homework 3

Author : Arjun Kramadhati Gopi, MS-Computer & Information
    Technology, Purdue University.
Date: September 21, 2020


[TO RUN CODE]: python3 removeDistortion.py
The code displays the pictures. The user will have to select the
    ROI points manually in the PQRS fashion.
P ------- Q
|         |
|         |
|         |
R ------- S

Output:
    [jpg]: [Transformed images]
"""

import cv2 as cv
import math
import numpy as np
import time

```

```

class removeDistortion:

    def __init__(self,image_addresses):


        self.image_addresses=image_addresses
        self.image_one = cv.imread(image_addresses[0])
        self.image_one = cv.resize(self.image_one,(int(self.
           image_one.shape[1]*0.5),int(self.image_one.shape
           [0]*0.5)))
        self.image_two = cv.imread(image_addresses[1])
        # self.image_two = cv.resize(self.image_two,(int(self.
           image_two.shape[1]*0.3),int(self.image_two.shape
           [0]*0.3)))
        self.image_three = cv.imread(image_addresses[2])
        self.image_three = cv.resize(self.image_three,(int(self.
           image_three.shape[1]*0.2),int(self.image_three.shape
           [0]*0.2)))
        self.images = [self.image_one,self.image_two,self.
           image_three]
        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])]
        self.image_sizes_corner_points_HC= []
        self.roiRealWorld = [[(0.0,0.0,1.0),(75.0,0.0,1.0)
           ,(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)]]
        self.roiCoordinates = []
        self.roiList = []
        self.homographies=[]
        self.resultImg = []
        self.xmin = 0
        self.ymin =0
        self.createImageCornerPointRepresentations()


    def createImageCornerPointRepresentations(self):
        """
        [summary] This function creates HC representations of the
            corner points of the given original input images.
        """
        templist = []
        for size in self.image_sizes:
            templist.append(np.asarray([0.0,0.0,1.0]))
            templist.append(np.asarray([float(size[1])
               -1.0,0.0,1.0]))
            templist.append(np.asarray([0.0,float(size[0])
               -1.0,1.0]))
```

```
            templist.append(np.asarray([float(size[1])-1.0,float(size[0])-1.0,1.0]))
            self.image_sizes_corner_points_HC.append(templist)
            templist = []


    def append_points(self,event,x,y,flags,param):
        """
        [This function is called every time the mouse left button is clicked - It records the (x,y) coordinates of the click location]

        """
        if event == cv.EVENT_LBUTTONDOWN:
            self.roiCoordinates.append((float(x),float(y),1.0))



    def getROIFromUser(self):
        """
        [This function is responsible for taking the regions of interests from the user for all the 4 pictures in order]

        """
        self.roiList=[]
        cv.namedWindow('Select ROI')

        cv.setMouseCallback('Select ROI',self.append_points)
        for i in range(3):
            while(True):
                cv.imshow('Select ROI',self.images[i])
                k = cv.waitKey(1) & 0xFF
                if cv.waitKey(1) & 0xFF == ord('q'):
                    break

            self.roiList.append(self.roiCoordinates)

            self.roiCoordinates = []


    def weightedPixelValue(self,rangecoordinates,objectQueue):
        """
        [This function calculates the weighted pixel value at the given coordinate in the target image]

        Args:
            rangecoordinates ([list]): [This is the coordinate of the pixel in the target image]
            objectQueue ([int]): [This is the index number of the list which has the coordinates of the roI for the Object picture]

```

```
        Returns:
            [list]: [Weighted pixel value - RGB value]
        """

        pointOne = (int(np.floor(rangecoordinates[1])),int(np.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])))

        pixelValueAtOne = self.images[objectQueue][pointOne[0]][pointOne[1]]
        pixelValueAtTwo = self.images[objectQueue][pointTwo[0]][pointTwo[1]]
        pixelValueAtThree = self.images[objectQueue][pointThree[0]][pointThree[1]]
        pixelValueAtFour = self.images[objectQueue][pointFour[0]][pointFour[1]]

        weightAtOne = 1/np.linalg.norm(pixelValueAtOne-rangecoordinates)
        weightAtTwo = 1/np.linalg.norm(pixelValueAtTwo-rangecoordinates)
        weightAtThree = 1/np.linalg.norm(pixelValueAtThree-rangecoordinates)
        weightAtFour = 1/np.linalg.norm(pixelValueAtFour-rangecoordinates)

        return ((weightAtOne*pixelValueAtOne) + (weightAtTwo*pixelValueAtTwo) + (weightAtThree*pixelValueAtThree) + (weightAtFour*pixelValueAtFour))/(weightAtFour+weightAtThree+weightAtTwo+weightAtOne)


    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:
            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]

        Returns:
```

```
            [numpy array]: [np.zeros of the size equal to the new image size]
            [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 least y value amongst the four transformed corner points]
        """

        templist = []
        templistX=[]
        templistY=[]
        #print(self.image_sizes_corner_points_HC[queueImage])
        #print(self.homographies[queueHomography][0])
        for i in range(4):
            templist.append(np.dot(self.homographies[queueHomography],self.image_sizes_corner_points_HC[queueImage][i]))
        #print(templist)

        for i,element in enumerate(templist):
            templist[i] = element/element[2]
        for element in templist:
            templistX.append(element[0])
            templistY.append(element[1])

        breadth = int(math.ceil(max(templistX))) - int(math.floor(min(templistX)))
        height = int(math.ceil(max(templistY))) - int(math.floor(min(templistY)))

        return np.zeros((height,breadth,3)),int(math.floor(min(templistX))),int(math.floor(min(templistY)))



    def createImage(self,queueHomography,queueImage):
        """[summary]
        This function is the function which creates the final result image. This function has the traditional but slow nested for loop approach to build the image.
        It begins by first getting the blank image of the size of the new image from the createBlankImageArray function above.

        Args:
            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]

        Returns:
```

```
173             [numpy ndarray]: [Returns the final resultant image in numpy.ndarray form.]
174         """
175         print("Processing...")
176         resultImg,xmin,ymin = self.createBlankImageArray(queueHomography,queueImage)
177 
178         for column in range(0,resultImg.shape[0]):
179             for row in range(0,resultImg.shape[1]):
180                 print("processing" + str(column) + " out of "+ str(resultImg.shape[0]))
181                 rangecoordinates = np.dot(self.homographies[queueHomography+1],(float(row+xmin),float(column+ymin),1.0))
182                 rangecoordinates = rangecoordinates/rangecoordinates[2]
183 
184                 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)):
185                      resultImg[column][row] = self.weightedPixelValue(rangecoordinates,queueImage)
186                 else:
187                      resultImg[column][row] = [0,0,0]
188 
189         return resultImg
190 
191     def createImageVectorised(self,queueHomography,queueImage):
192         """[summary]
193         ----------- Attempt #1 ---------------
194 
195         Vectorised numpy operation
196 
197         -------------------------------------
198 
199         This function is the function which creates the final result image. This was the first attempt towards writing a fully vectorised numpy pythonic operation.
200         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.
201         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
202         multiplication with the homograhy matrix H. I do this using the '@' operator. The resulting matrix has the corresponding pixel coordinates of the source image.
203         I extract the pixel values of each of these coordinates using a nested for loop. Basically, I was able to avoid the matrix multiplication being written inside the
```

```
        nexted for loop. I was able to get stable outputs much quicker - 40% faster.

        Args:
            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]

        Returns:
            [numpy ndarray]: [Returns the final resultant image in numpy.ndarray form.]
        """

        print("processing...")
        resultImg,xmin,ymin = self.createBlankImageArray(queueHomography,queueImage)
        column,row = np.mgrid[0:resultImg.shape[0],0:resultImg.shape[1]]
        vector = np.vstack((column.ravel(),row.ravel()))
        row = vector[1] + xmin
        column = vector[0] +ymin
        ones = np.ones(len(row))
        vector = np.array([column,row,ones])
        s=time.time()
        resultvector = self.homographies[queueHomography+1]@vector
        e=time.time()
        print("timetake",e-s)
        resultvector = resultvector/resultvector[2]
        # resultvector = resultvector[:2,:]
        for column in range(0,resultImg.shape[0]):
            for row in range(0,resultImg.shape[1]):
                print("processing" + str(column) + " out of "+str(resultImg.shape[0]))

                rangecoordinates=np.array([resultvector[1][(column*resultImg.shape[1])+row],resultvector[0][(column*resultImg.shape[1])+row],resultvector[2][(column*resultImg.shape[1])+row]])

                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)):
                     resultImg[column][row] = self.weightedPixelValue(rangecoordinates,queueImage)
                else:
                     resultImg[column][row] = [255.0,255.0,255.0]

        return resultImg
```

```python



    def buildImage(self,queueHomography,queueImage,row,column):
        """[summary]
        ----------- Attempt #2 ---------------

        Vectorised numpy operation

        -------------------------------------

        This function is the function which creates the final result image. This was the second attempt towards writing a fully vectorised numpy pythonic operation.
        This 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 vectorise operation. Using this entire function as a vector, I was able to successfully vectorise the
        whole image building process.

        Args:
            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]
            row ([int]) : [Row value of the pixel being considered]
            column ([int]) : [Column value of the pixel being considered]

        Returns:
            Does not return any value. It just updates the global image variable (self.resultImg).
        """


        rangecoordinates = np.matmul(self.homographies[queueHomography+1],(float(row+self.xmin),float(column+self.ymin),1.0))
        rangecoordinates = rangecoordinates/rangecoordinates/[2]
        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)):
            pointOne = (int(np.floor(rangecoordinates[1])),int(np.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])))

                pixelValueAtOne = self.images[queueImage][pointOne
                    [0]][pointOne[1]]
                pixelValueAtTwo = self.images[queueImage][pointTwo
                    [0]][pointTwo[1]]
                pixelValueAtThree = self.images[queueImage][
                    pointThree[0]][pointThree[1]]
                pixelValueAtFour = self.images[queueImage][pointFour
                    [0]][pointFour[1]]

                weightAtOne = 1/np.linalg.norm(pixelValueAtOne-
                    rangecoordinates)
                weightAtTwo = 1/np.linalg.norm(pixelValueAtTwo-
                    rangecoordinates)
                weightAtThree = 1/np.linalg.norm(pixelValueAtThree-
                    rangecoordinates)
                weightAtFour = 1/np.linalg.norm(pixelValueAtFour-
                    rangecoordinates)

                self.resultImg[column][row] = ((weightAtOne*
                    pixelValueAtOne) + (weightAtTwo*pixelValueAtTwo) +
                     (weightAtThree*pixelValueAtThree) + (weightAtFour
                    *pixelValueAtFour))/(weightAtFour+weightAtThree+
                    weightAtTwo+weightAtOne)
            else:

                self.resultImg[column][row] = [255.0,255.0,255.0]

    def vectoriseOperations(self,queueHomography,queueImage):
        """[summary]
        ----------- Attempt #2 Continued ---------------

        Vectorised numpy operation

        --------------------------------------

        This function is the extension of the above function -
           buildImage. This is the function which vectorises the
           entire buildImage function.
        In this function, I stack a list which contains all the
           pixel coordinates in the blank image. I feed this
           entire list to the vectorised function.
        This was a successful vectorisation operation however the
            RAM utilization peaked to a hundred percent. The
           laptop froze and I could not run this further.

        Args:
            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]
```

```
        Returns:
            [numpy ndarray]: [Returns the final resultant image
               in numpy.ndarray form.]
        """
        self.resultImg,self.xmin,self.ymin = self.
           createBlankImageArray(queueHomography,queueImage)
        length = self.resultImg.shape[0]*self.resultImg.shape[1]
        queueHomography = [queueHomography]*length
        queueImage = [queueImage]*length
        vectoriseOperation = np.vectorize(self.buildImage)
        row,column = np.mgrid[0:self.resultImg.shape[1],0:self.
           resultImg.shape[0]]
        point = np.vstack((row.ravel(),column.ravel()))
        row = point[0]
        column = point[1]
        #print(point)
        print("processing...")
        vectoriseOperation(queueHomography,queueImage,row,column)
        return self.resultImg






    def objectMatrixFunction(self,queue):
        """
        [We construct the B Matrix with dimension 8X1]

        Args:
            queue ([int]): [This is the index number of the list
               which has the coordinates of the roI for the
               object picture]
        """
        self.objectMatrix = np.zeros((8,1))

        for i in range(len(self.roiRealWorld[queue])):
            self.objectMatrix[(2*i)][0] = self.roiRealWorld[queue
               ][i][0]
            self.objectMatrix[(2*i)+1][0] = self.roiRealWorld[
               queue][i][1]

    def parameterMatrixFunction(self,queue,objectQueue):
        """
        [We construct the A Matrix with dimension 8X8 and then we
            calculate the inverse of A matrix needed for the
           homography calculation]

        Args:
            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 list which has the coordinates of the roI for the Object picture]
        """
        self.parameterMatrix=np.zeros((8,8))

        for i in range(4):
            self.parameterMatrix[2*i][0] = self.roiList[queue][i][0]
            self.parameterMatrix[2*i][1] = self.roiList[queue][i][1]
            self.parameterMatrix[2*i][2] = 1.0
            self.parameterMatrix[2*i][3] = 0.0
            self.parameterMatrix[2*i][4] = 0.0
            self.parameterMatrix[2*i][5] = 0.0
            self.parameterMatrix[2*i][6] = (-1)*(self.roiList[queue][i][0])*(self.roiRealWorld[objectQueue][i][0])
            self.parameterMatrix[2*i][7] = (-1)*(self.roiList[queue][i][1])*(self.roiRealWorld[objectQueue][i][0])
            self.parameterMatrix[(2*i) + 1][0] = 0.0
            self.parameterMatrix[(2*i) + 1][1] = 0.0
            self.parameterMatrix[(2*i) + 1][2] = 0.0
            self.parameterMatrix[(2*i) + 1][3] = self.roiList[queue][i][0]
            self.parameterMatrix[(2*i) + 1][4] = self.roiList[queue][i][1]
            self.parameterMatrix[(2*i) + 1][5] = 1.0
            self.parameterMatrix[(2*i) + 1][6] = (-1)*(self.roiList[queue][i][0])*(self.roiRealWorld[objectQueue][i][1])
            self.parameterMatrix[(2*i) + 1][7] = (-1)*(self.roiList[queue][i][1])*(self.roiRealWorld[objectQueue][i][1])

        self.parameterMatrixI = np.linalg.pinv(self.parameterMatrix)

    def calculateHomography(self):
        """
        [We calculate the homography matrix here. Once we have the values of the matrix, we rearrange them into a 3X3 matrix.]

        """
        homographyI = np.matmul(self.parameterMatrixI,self.objectMatrix)
        homography = np.zeros((3,3))

        homography[0][0]= homographyI[0]
        homography[0][1]= homographyI[1]
        homography[0][2]= homographyI[2]
```

```
        homography[1][0]= homographyI[3]
        homography[1][1]= homographyI[4]
        homography[1][2]= homographyI[5]
        homography[2][0]= homographyI[6]
        homography[2][1]= homographyI[7]
        homography[2][2]= 1.0
        self.homographies.append(homography)
        homography = np.linalg.pinv(homography)
        homography = homography/homography[2][2]
        self.homographies.append(homography)



    def projectiveDistortionHomography(self,queueImage):
        """[summary]
        Calculate the homography matrix to eliminate projective
           distortion

        Args:
            queueImage ([int]): [Index of the image in the list
               being used]

        Calculates the Homography matrix and appends it to the
           global homography list.
        """

        vanishingPointOne = np.cross(np.cross(self.roiList[
           queueImage][0],self.roiList[queueImage][1]),np.cross(
           self.roiList[queueImage][2],self.roiList[queueImage
           ][3]))
        vanishingPointTwo = np.cross(np.cross(self.roiList[
           queueImage][0],self.roiList[queueImage][2]),np.cross(
           self.roiList[queueImage][1],self.roiList[queueImage
           ][3]))

        vanishingLine = np.cross((vanishingPointOne/
           vanishingPointOne[2]),(vanishingPointTwo/
           vanishingPointTwo[2]))

        projectiveDHomography = np.zeros((3,3))
        projectiveDHomography[2] = vanishingLine/vanishingLine[2]
        projectiveDHomography[0][0] = 1
        projectiveDHomography[1][1] = 1
        self.homographies.append(projectiveDHomography)
        inverseH = np.linalg.pinv(projectiveDHomography)
        self.homographies.append(inverseH/inverseH[2][2])



    def affineDistortionHomography(self,queueImage):
        """[summary]
        Calculate the homography matrix to eliminate affine
           distortion
```

```

        Args:
            queueImage ([int]): [Index of the image in the list being used]

        Calculates the Homography matrix and appends it to the global homography list.
        """
        templist = []
        temppoints = []

        for i in range(4):
            tempvalue = np.dot(self.homographies[0],self.roiList[queueImage][i])
            tempvalue = tempvalue/tempvalue[2]
            temppoints.append(tempvalue)

        print(temppoints)
        ortholinePairOne = np.cross(temppoints[0],temppoints[1])
        ortholinePairTwo = np.cross(temppoints[0],temppoints[2])
        ortholinePairThree = np.cross(temppoints[0],temppoints[3])
        ortholinePairFour = np.cross(temppoints[1],temppoints[2])
        templist.append(ortholinePairOne)
        templist.append(ortholinePairTwo)
        templist.append(ortholinePairThree)
        templist.append(ortholinePairFour)

        for i,element in enumerate(templist):
            #print(element)
            #print(element[2])
            templist[i] = element/element[2]

        matrixAT = []
        matrixAT.append([templist[0][0]*templist[1][0],templist[0][0]*templist[1][1]+templist[0][1]*templist[1][0]])
        matrixAT.append([templist[2][0]*templist[3][0],templist[2][0]*templist[3][1]+templist[2][1]*templist[3][0]])
        matrixAT = np.asarray(matrixAT)
        matrixAT = np.linalg.pinv(matrixAT)
        matrixA = []
        matrixA.append([-templist[0][1]*templist[1][1]])
        matrixA.append([-templist[2][1]*templist[3][1]])
        matrixA = np.asarray(matrixA)

        matrixS = np.dot(matrixAT,matrixA)
        matrixSRearranged = np.zeros((2,2))

        matrixSRearranged[0][0] = matrixS[0]
        matrixSRearranged[0][1] = matrixS[1]
        matrixSRearranged[1][0] = matrixS[1]
        matrixSRearranged[1][1] = 1

```

```python
        v,lambdamatrix,q = np.linalg.svd(matrixSRearranged)

        lambdavalue = np.sqrt(np.diag(lambdamatrix))
        Hmatrix = np.dot(np.dot(v,lambdavalue),v.transpose())

        affineHomography=np.zeros((3,3))
        affineHomography[0][0] = Hmatrix[0][0]
        affineHomography[0][1] = Hmatrix[0][1]
        affineHomography[1][0] = Hmatrix[1][0]
        affineHomography[1][1] = Hmatrix[1][1]
        affineHomography[2][2] = 1


        inverseH = np.linalg.pinv(affineHomography)
        inverseH = np.dot(inverseH,self.homographies[0])
        self.homographies.append(inverseH)
        inverseH = np.linalg.pinv(inverseH)
        self.homographies.append(inverseH/inverseH[2][2])

    def oneStepDistortionHomography(self,queueImage):
        """[summary]
        Calculate the homography matrix to eliminate both projective and affine distortion

        Args:
            queueImage ([int]): [Index of the image in the list being used]

        Calculates the Homography matrix and appends it to the global homography list.
        """
        matrixA=[]
        matrixAT = []
        templist=[]
        templist.append(np.cross(self.roiList[queueImage][0],self.roiList[queueImage][1]))
        templist.append(np.cross(self.roiList[queueImage][1],self.roiList[queueImage][3]))
        templist.append(np.cross(self.roiList[queueImage][1],self.roiList[queueImage][3]))
        templist.append(np.cross(self.roiList[queueImage][3],self.roiList[queueImage][2]))
        templist.append(np.cross(self.roiList[queueImage][3],self.roiList[queueImage][2]))
        templist.append(np.cross(self.roiList[queueImage][2],self.roiList[queueImage][0]))
        templist.append(np.cross(self.roiList[queueImage][2],self.roiList[queueImage][0]))
        templist.append(np.cross(self.roiList[queueImage][0],self.roiList[queueImage][1]))
        templist.append(np.cross(self.roiList[queueImage][0],self.roiList[queueImage][3]))
        templist.append(np.cross(self.roiList[queueImage][1],self
```

```
            .roiList[queueImage][2]))

        for i,element in enumerate(templist):
            templist[i] = element/element[2]

        for i in range(0,10,2):
            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]])

        matrixAT = np.asarray(matrixAT)
        matrixA = np.asarray(matrixA)
        matrixS = np.dot(np.linalg.pinv(matrixAT),matrixA)
        matrixS = matrixS/np.max(matrixS)

        matrixSRearranged = np.zeros((2,2))
        matrixSRearranged[0][0] = matrixS[0]
        matrixSRearranged[0][1] = matrixS[1] * 0.5
        matrixSRearranged[1][0] = matrixS[1] * 0.5
        matrixSRearranged[1][1] = matrixS[2]
        matrixST = np.array([matrixS[3]*0.5,matrixS[4]*0.5])
        v,lambdamatrix,q = np.linalg.svd(matrixSRearranged)
        lambdavalue = np.sqrt(np.diag(lambdamatrix))
        Hmatrix = np.dot(np.dot(v,lambdavalue),v.transpose())
        Vmatrix = np.dot(np.linalg.pinv(Hmatrix),matrixST)

        onestepHomography =np.zeros((3,3))
        onestepHomography[0][0] = Hmatrix[0][0]
        onestepHomography[0][1] = Hmatrix[0][1]
        onestepHomography[1][0] = Hmatrix[1][0]
        onestepHomography[1][1] = Hmatrix[1][1]
        onestepHomography[2][0] = Vmatrix[0]
        onestepHomography[2][1] = Vmatrix[1]
        onestepHomography[2][2]=1

        inverseH = np.linalg.pinv(onestepHomography)
        self.homographies.append(inverseH)
        inverseH = np.linalg.pinv(inverseH)
        self.homographies.append(inverseH/inverseH[2][2])





if __name__ == "__main__":

        """
        The code begins here. Make sure the input image paths are
            properly inserted.
```

```

        """

    tester = removeDistortion(['hw3_Task1_Images/Images/1.jpg','hw3_Task1_Images/Images/2.jpg','hw3_Task1_Images/Images/3.jpg'])
    tester.getROIFromUser()
    for i in range(0,3):
        tester.objectMatrixFunction(i)
        tester.parameterMatrixFunction(i,i)
        tester.calculateHomography()
        resultImg = tester.createImage(0,i)
        cv.imwrite("ptp" +str(i)+".jpg",resultImg)

    tester.getROIFromUser()

    for i in range(0,3):
        tester.projectiveDistortionHomography(i)
        resultImg = tester.createImage(0,i)
        # resultImg = tester.createImageVectorised(0,0)
        cv.imwrite('1' +str(i)+'.jpg',resultImg)
        tester.affineDistortionHomography(i)
        resultImg = tester.createImage(2,i)
        cv.imwrite('2' +str(i)+'.jpg',resultImg)
        tester.oneStepDistortionHomography(i)
        resultImg = tester.createImage(4,i)
        cv.imwrite('3' +str(i)+'.jpg',resultImg)

    ######Custom Input Images########

    tester = removeDistortion(['hw3_Task1_Images/Images/sn.jpg','hw3_Task1_Images/Images/laptop.jpg'])
    tester.getROIFromUser()
    for i in range(0,2):
        tester.objectMatrixFunction(i)
        tester.parameterMatrixFunction(i,i)
        tester.calculateHomography()
        resultImg = tester.createImage(0,i)
        cv.imwrite("ptp" +str(i)+".jpg",resultImg)

    tester.getROIFromUser()

    for i in range(0,2):
        tester.projectiveDistortionHomography(i)
        resultImg = tester.createImage(0,i)
        # resultImg = tester.createImageVectorised(0,0)
        cv.imwrite('1' +str(i)+'.jpg',resultImg)
        tester.affineDistortionHomography(i)
        resultImg = tester.createImage(2,i)
        cv.imwrite('2' +str(i)+'.jpg',resultImg)
        tester.oneStepDistortionHomography(i)
        resultImg = tester.createImage(4,i)
        cv.imwrite('3' +str(i)+'.jpg',resultImg)
```