ECE 661: Computer Vision

Homework 3, Fall 2018

Arindam Bhanja Chowdhury
abhanjac@purdue.edu

1 Overview

This homework is about removing projective and affine distortion from two images (Fig 1 and 2) using *Homography* matrices. The original dimensions of the features of these images are also available, as shown in Fig 3 and 4.

Figure 1: Reference image 1 of homework.

Figure 2: Reference image 2 of homework.

Figure 3: Dimensions of features of the reference image 1 of homework.

Figure 4: Dimensions of features of the reference image 2 of homework.

2 METHOD 1 - Removing Distortion by Point-to-Point correspondence

This part has been done exactly in the same manner as was done in HW2. The points from the source (distorted) image is mapped to the target image and the *Homography* matrix is calculated to remove the distortions. So please refer to the previous homework (or the Appendix section) for the explanation of the theory. The only difference is that, there is no target image specified here. The points from the original image are to be mapped to some coordinates. Like the points around the window of the building in Fig 1 are to be mapped to an image at the pixel locations of $(0, 0), (60, 0), (0, 80)$ and $(60, 80)$. The pixel coordinates may not be exactly 60 or 80, but they should be some multiple or fraction of these numbers, so that the overall proportion of the image in the final corrected version is at par with the original dimensions. Same procedure was followed for the image in Fig 2 as well.

3 Procedure of METHOD 1

- 4 points are chosen from the vertices of the building as shown in Fig 5.
- These are mapped to the coordinates which are proportional to the true dimensions shown in Fig 3.

- Homography between the points of Fig 5 and the original Fig 1 is calculated.
- Some of the points are mapped to the negative coordinates. Hence the lowest value of the x and y coordinates that are encountered after the mapping is done, are subtracted from all the x and y coordinate values of all the points during mapping.
- Final image with the distortions removed is shown in Fig 7. It shows that lines like the edges of the building are parallel and straight.
- The same procedure is followed for removing the distortions of Fig 2. The resulting undistorted version is shown in Fig 8.

4 Results of METHOD 1

Figure 5: Image showing the points selected for removing the distortions from Fig 1 image.

Figure 6: Image showing the points selected for removing the distortions from Fig 2 image.

H matrix for Fig 1 image correction:

$$\begin{bmatrix} 2.16226150 \times 10^{-1} & -8.37511514 \times 10^{-2} & 6.61000000 \times 10^{2} \\ -2.55954273 & 4.37831905e+00 & 1.68900000 \times 10^{3} \\ -1.28323244 \times 10^{-3} & -1.26703709e-04 & 1.00000000 \end{bmatrix}$$

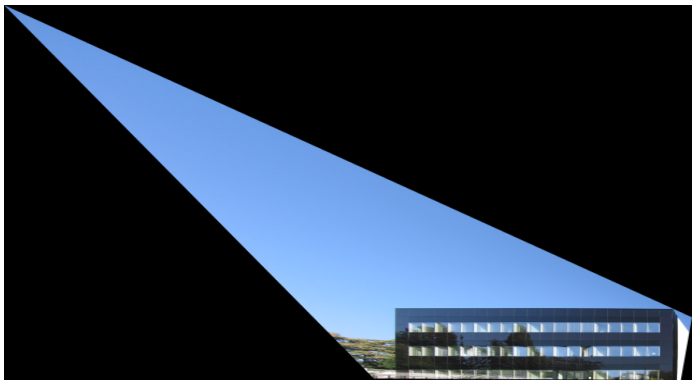

Figure 7: Undistorted version of Fig 1 image.

Figure 8: Undistorted version of Fig 2 image.

H matrix for Fig 2 image correction:

$$\begin{bmatrix} 3.08909749 & 5.35252073 \times 10^{-1} & 1.27000000 \times 10^{2} \\ 4.80605964 \times 10^{-1} & 1.80417477 & 1.20000000 \times 10 \\ 3.41594773 \times 10^{-3} & 1.14667845 \times 10^{-3} & 1.00000000 \end{bmatrix}$$

5 METHOD 2 - Removing Distortion by 2-Step Process

This method has two steps. First the projective distortion is removed which makes the parallel lines become parallel in the resulting image, and then the affine distortion is removed (from the output image of the previous step) which makes corrects the angles in the image, such that the perpendicular lines meet at right angles again in the resulting image. This corrects the distortions of the overall image.

5.1 Removing Projective Distortion

To remove projective distortion, first some pair of points are selected from the distorted image, such that the line joining these pair of points will be parallel in the undistorted version.

Let $P_1 = [p_{1x}, p_{1y}]^T$ and $P_2 = [p_{2x}, p_{2y}]^T$ be two points and $P_3 = [p_{3x}, p_{3y}]^T$ and $P_4 = [p_{4x}, p_{4y}]^T$ be two other points, such that the line $L_1 = [l_{11}, l_{12}, l_{13}]^T$ joining P_1 and P_2 and the line $L_2 = [l_{21}, l_{22}, l_{23}]^T$ joining P_3 and P_4 are parallel in the undistorted image. The formula for calculating

a line joining two points is show in 1. The points have to be in homogeneous coordinates in this equation.

$$L_1 = [l_{11}, l_{12}, l_{13}]^T = P_1 \times P_2 = [p_{1x}, p_{1y}, 1]^T \times [p_{2x}, p_{2y}, 1]^T \tag{1}$$

Similarly L_2 is also calculated. In the same manner another pair of distorted parallel lines L_3 and L_4 is calculated.

Now since the image is distorted, these pair of parallel lines do not meet at infinity. They meet at the *vanishingpoints*, which are the point of intersection of these pair of parallel lines. They can be calculated as shown in 2.

$$VP_1 = [VP_{1x}, VP_{1y}, VP_{1z}]^T = L_1 \times L_2 = [l_{11}, l_{12}, l_{13}]^T \times [l_{21}, l_{22}, l_{23}]^T \tag{2}$$

VP_1 is the vanishing point where L_1 and L_2 meets. Similarly, VP_2 is calculated where L_3 and L_4 meets.

Now, the line joining these two vanishing points or the *vanishingline* VL is also calculated using the same kind of equation as 1.

Now, in the undistorted image this VL will stay at infinity. Or VL should be equal to $[0, 0, 1]^T$. But because of the projective distortion, the VL has some different value $[VL_1, VL_2, VL_3]^T$. So, the $HomographyH$ that will transfer VL to the value of $[0, 0, 1]^T$ will rectify the projective distortion of this image.

$$H_{proj} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ VL_1 & VL_2 & VL_3 \end{bmatrix} \tag{3}$$

All the homogeneous coordinates of the points in the distorted image when multiplied with this H_{proj} matrix, gives the coordinates of the points in the undistorted image, that they will be mapped to.

5.2 Removing Affine Distortion

Now, if an image only has affine distortions, i.e. parallel lines are parallel, but the angle between perpendicular lines are not 90°.

To remove affine distortion, first some pair of points are selected from the distorted image, such that the line joining these pair of points will be perpendicular in the undistorted version.

In the same manner as in the earlier subsection, two pairs of points are selected from the distorted image, such that the line joining each of these pairs are perpendicular to each other.

Let the lines $L = [l_1, l_2, l_3]^T$ and $M = [m_1, m_2, m_3]^T$ be two such perpendicular lines. Their values are obtained in the same manner as shown in equation 1. Let the angle between these lines be θ which is not 90° in this distorted image. So the equation of $\cos\theta$ is obtained as the following,

$$\cos\theta = \frac{l_1 m_1 + l_2 m_2}{\sqrt{(l_1^2 + l_2^2)(m_1^2 + m_2^2)}} \tag{4}$$

Since the angle between the lines only depends on the slope, so there is no l_3 or m_3 in this equation 4.

Equation 4 can also be represented using the dual degenerate conic C^*_∞ as the following

$$\cos\theta = \frac{L^T C^*_\infty M}{\sqrt{(L^T C^*_\infty L)(M^T C^*_\infty M)}} \tag{5}$$

Conics transform by the equation $C' = H^{-T} C H$. Hence dual conics transform by the formula $C'^* = H C^* H^T$. So the degenerate conic also transforms in the same manner. Now if only the perpendicular angles are considered, then $\cos\theta = 0$. So the equation 5 can be rewritten with only the numerator as the following

$$L'^T H C^*_\infty H^T M = 0$$

Where $L' = H^{-T} L$ and $M' = H^{-T} M$ with H as the homography matrix for affine distortion given as,

$$H = \begin{bmatrix} A & \vec{t} \\ \vec{0} & 1 \end{bmatrix} \tag{6}$$

Where A is the affine part of H and t is the translation part of H.
Or,

$$[l'_1, l'_2, l'_3] H C^*_\infty H^T \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} = 0$$

Or,

$$[l'_1, l'_2, l'_3] \begin{bmatrix} A & \vec{t} \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} I & \vec{0} \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} A^T & \vec{0} \\ \vec{t}^T & 1 \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} = 0 \tag{7}$$

$$[l'_1, l'_2, l'_3] \begin{bmatrix} AA^T & \vec{0} \\ \vec{0} & 0 \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} \tag{8}$$

Equation 7 can be simplified to the following form by the matrix multiplications.

$$[l'_1, l'_2] \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \end{bmatrix} \tag{9}$$

The 2×2 matrix in equation 9 is referred to as $S = AA^T$.
Now, selecting points from the affine distorted image will give the l and m components. So the actual unknowns in the 9 are the s components. Now, because only ratios matter, so one of the s (here s_{22}) can be assumed to be 1. And, the S matrix is also symmetric, so $s_{12} = s_{21}$. So there are only 2 unknowns in the equation 9. So the 9 can be rewritten as follows in a more convenient form.

$$[l'_1 m'_1, l'_1 m'_2 + l'_2 m'_1] \begin{bmatrix} s_{11} \\ s_{12} \end{bmatrix} = -l'_2 m'_2 \tag{10}$$

Now, to solve for both the unknown s, another pair of distorted perpendicular line is calculated from the image ($[l_4, l_5, l_6]^T$ and $[m_4, m_5, m_6]^T$).
So now our equation 10 is like

$$\begin{bmatrix} l'_1 m'_1 & l'_1 m'_2 + l'_2 m'_1 \\ l'_4 m'_4 & l'_4 m'_5 + l'_5 m'_4 \end{bmatrix} \begin{bmatrix} s_{11} \\ s_{12} \end{bmatrix} = \begin{bmatrix} -l'_2 m'_2 \\ -l'_5 m'_5 \end{bmatrix} \tag{11}$$

Or,
The s components can be solved from this equation and the S matrix can be formed form them. Now if A is represented in its Singular Value Decomposition (SVD) form, as $AA^T = VDV^T$, then we can represent $S = AA^T = VDV^TVDV^T = VD^2V^T$. Which implies that if we take the SVD of $S = UEU^T$, then we can get back the A matrix as $A = U\sqrt{E}U^T$. Once we get A in this form, H can be obtained as shown in 6. The $\vec{t}$ can be just assumed to be $[0, 0]^T$ as the translation does not matter for affine transform.

5.3 Combining Affine and Projection Homography

Now, the H is the homography that distorts the image. So the $H_{aff} = H^{-1}$ is the homography that will correct that distortion. So, H_{aff} and the H_{proj} can be combined and applied together to the points of the distorted image to get the image with these kind of distortions removed. This homography matrix is given as $H_{aff}H_{proj}$.

6 Procedure of METHOD 2

- Points are selected from Fig 1 to find the distorted parallel lines. 4 points on the corners of rectangle are selected which gives altogether 2 pairs of parallel lines.
- The same points as shown in Fig 5 are used here.
- The homography matrix is obtained from the coefficients of the vanishing line. This is used to remove the projective distortion. The resulting image is shown in Fig 9.
- 8 points are selected to 2 pairs of perpendicular lines. These points are shown in the Fig 11.
- These perpendicular line coefficients are used to find the homography matrix that will remove affine distortion.
- The homography matrix for removing affine distortion is the combined with that of projective distortion to create an overall matrix that will undistort the Fig 1 image. The result is shown in Fig 13.
- The same procedure is followed for removing the distortions of Fig 2. The resulting undistorted version is shown in Fig 14.

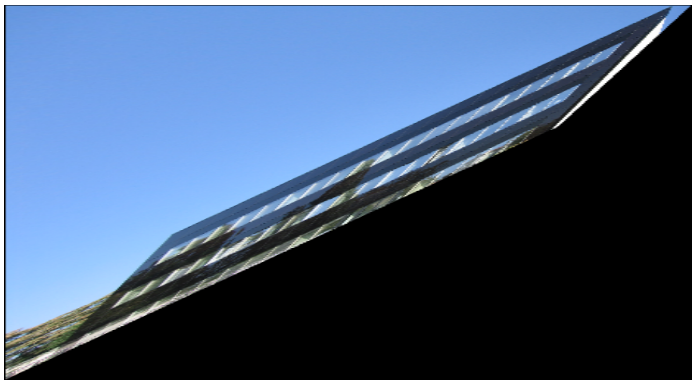

Figure 9: Fig 1 image with projective distortion removed.

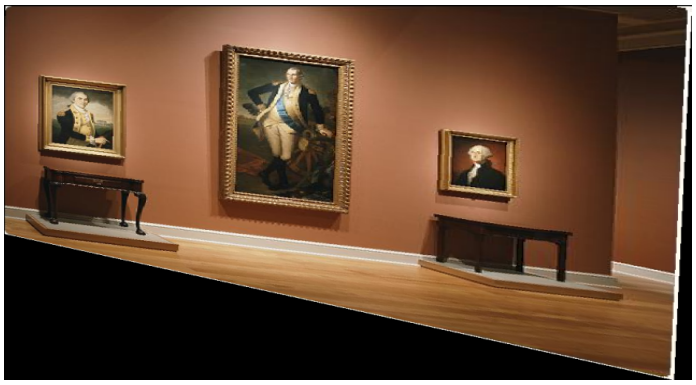

Figure 10: Fig 2 image with projective distortion removed.

Figure 11: Image showing the points selected for finding the distorted perpendicular lines from Fig 1 image.

7 Results of METHOD 2

H_{proj} to remove projective distortion from Fig 1 image:

$$\begin{bmatrix} 1.00000000 & 0.00000000 & 0.00000000 \\ 0.00000000 & 1.00000000 & 0.00000000 \\ 8.67829839 \times 10^{-3} & -5.25627385 \times 10^{-5} & 1.00000000 \end{bmatrix}$$

Figure 12: Image showing the points selected for finding the distorted perpendicular lines from Fig 2 image.

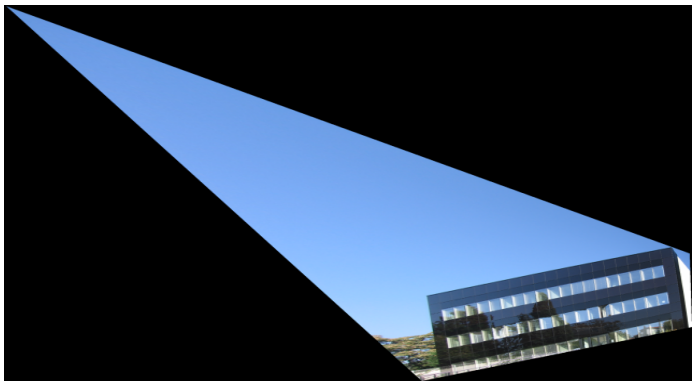

Figure 13: Undistorted version of Fig 1 image. (By method 2)

Figure 14: Undistorted version of Fig 2 image. (By method 2)

H affine to remove affine distortion from Fig 1 image:

$$\begin{bmatrix} 45.51047625 & 0.23736743 & 0.00000000 \\ 0.23736743 & 1.00125163 & 0.00000000 \\ 0.00000000 & 0.00000000 & 1.00000000 \end{bmatrix}$$

H for the overall distortion removal by Method 2 from Fig 1 image:

$$\begin{bmatrix} 4.55104762 \times 10^{1} & 2.37367430 \times 10^{-1} & 0.00000000 \\ 2.37367430 \times 10^{-1} & 1.00125163 & 0.00000000 \\ 8.67829839 \times 10^{-3} & -5.25627385 \times 10^{-5} & 1.00000000 \end{bmatrix}$$

H_{proj} to remove projective distortion from Fig 2 image:

$$\begin{bmatrix} 1.00000000 & 0.00000000 & 0.00000000 \\ 0.00000000 & 1.00000000 & 0.00000000 \\ -1.07102506 \times 10^{-3} & -2.97721070 \times 10^{-4} & 1.00000000 \end{bmatrix}$$

H affine to remove affine distortion from Fig 2 image:

$$\begin{bmatrix} 0.38730055 & -0.16760144 & 0.00000000 \\ -0.16760144 & 1.1621457 & 0.00000000 \\ 0.00000000 & 0.00000000 & 1.00000000 \end{bmatrix}$$

H for the overall distortion removal by Method 2 from Fig 2 image:

$$\begin{bmatrix} 3.87300551 \times 10^{-1} & -1.67601437 \times 10^{-1} & 0.00000000 \\ -1.67601437 \times 10^{-1} & 1.16214570 & 0.00000000 \\ -1.07102506 \times 10^{-3} & -2.97721070 \times 10^{-4} & 1.00000000 \end{bmatrix}$$

8 APPENDIX

8.1 Homography Calculation by Point Matching

Let $P = [p_x, p_y]^T$ be a point in planar coordinate of the source image and $P_1 = [p_{x1}, p_{y1}]^T$ be the corresponding planar point in the transformed image. Let H be the Homography matrix that does this transformation. And let P_h and P_{h1} are the homogeneous coordinates of the points P and P_1, then the following equation can be written.

$$P_{h1} = HP_h \tag{12}$$

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{13}$$

Now, in the homogeneous coordinate system, only the ratio of the coefficients matter. So any one of the elements of H can be considered as 1. Here, $h_{33} = 1$ is considered.

$$P_h = [p_{x1}, p_{y1}, 1]^T \tag{14}$$

$$P_{h1} = [p'_x, p'_y, p'_z]^T \tag{15}$$

So the P_{h1} can be converted to its planar form P_1 in the following manner.

$$P_1 = [p_{x1}, p_{y1}]^T = \left[\frac{p'_x}{p'_z}, \frac{p'_y}{p'_z}\right]^T \tag{16}$$

Expanding the equation (13) with the assumption of $h_{33} = 1$:

$$\begin{aligned} p'_x &= h_{11}p_x + h_{12}p_y + h_{13} \\ p'_y &= h_{21}p_x + h_{22}p_y + h_{23} \\ p'_z &= h_{31}p_x + h_{32}p_y + 1 \end{aligned} \tag{17}$$

Now, the equations for p_{x1} and p_{y1} can be written with the help of (7) as follows.

$$\begin{aligned} p_{x1} = \frac{p'_x}{p'_z} &= \frac{h_{11}p_x + h_{12}p_y + h_{13}}{h_{31}p_x + h_{32}p_y + 1} \\ p_{y1} = \frac{p'_y}{p'_z} &= \frac{h_{21}p_x + h_{22}p_y + h_{23}}{h_{31}p_x + h_{32}p_y + 1} \end{aligned}$$

which can be simplified as,

$$\begin{aligned} p_{x1} &= h_{11}p_x + h_{12}p_y + h_{13} - h_{31}p_x p_{x1} - h_{32}p_y p_{x1} \\ p_{y1} &= h_{21}p_x + h_{22}p_y + h_{23} - h_{31}p_x p_{y1} - h_{32}p_y p_{y1} \end{aligned} \tag{18}$$

Now, in this task, the points $P = [p_x, p_y]^T$ and $P_1 = [p_{x1}, p_{y1}]^T$ are already available from the given images (those have to be picked up manually). So the h elements of the Homography matrix are the actual unknowns. And there are altogether 8 of them. So, at least 8 equations like those in (18) are needed. For this at least 4 points are needed from the images. So 3 more points Q, R and S are also selected from each of the images. These points will also form the same derivation as above and the 8 equations obtained are

$$\begin{aligned} p_{x1} &= h_{11}p_x + h_{12}p_y + h_{13} - h_{31}p_x p_{x1} - h_{32}p_y p_{x1} \\ p_{y1} &= h_{21}p_x + h_{22}p_y + h_{23} - h_{31}p_x p_{y1} - h_{32}p_y p_{y1} \\ q_{x1} &= h_{11}q_x + h_{12}q_y + h_{13} - h_{31}q_x q_{x1} - h_{32}q_y q_{x1} \\ q_{y1} &= h_{21}q_x + h_{22}q_y + h_{23} - h_{31}q_x q_{y1} - h_{32}q_y q_{y1} \\ r_{x1} &= h_{11}r_x + h_{12}r_y + h_{13} - h_{31}r_x r_{x1} - h_{32}r_y r_{x1} \\ r_{y1} &= h_{21}r_x + h_{22}r_y + h_{23} - h_{31}r_x r_{y1} - h_{32}r_y r_{y1} \\ s_{x1} &= h_{11}s_x + h_{12}s_y + h_{13} - h_{31}s_x s_{x1} - h_{32}s_y s_{x1} \\ s_{y1} &= h_{21}s_x + h_{22}s_y + h_{23} - h_{31}s_x s_{y1} - h_{32}s_y s_{y1} \end{aligned} \tag{19}$$

(10) in matrix form is the following

$$\begin{bmatrix} p_{x1} \\ p_{y1} \\ q_{x1} \\ q_{y1} \\ r_{x1} \\ r_{y1} \\ s_{x1} \\ s_{y1} \end{bmatrix} = \begin{bmatrix} p_x & p_y & 1 & 0 & 0 & 0 & -p_x p_{x1} & -p_y p_{x1} \\ 0 & 0 & 0 & p_x & p_y & 1 & -p_x p_{y1} & -p_y p_{y1} \\ q_x & q_y & 1 & 0 & 0 & 0 & -q_x q_{x1} & -q_y q_{x1} \\ 0 & 0 & 0 & q_x & q_y & 1 & -q_x q_{y1} & -q_y q_{y1} \\ r_x & r_y & 1 & 0 & 0 & 0 & -r_x r_{x1} & -r_y r_{x1} \\ 0 & 0 & 0 & r_x & r_y & 1 & -r_x r_{y1} & -r_y r_{y1} \\ s_x & s_y & 1 & 0 & 0 & 0 & -s_x s_{x1} & -s_y s_{x1} \\ 0 & 0 & 0 & s_x & s_y & 1 & -s_x s_{y1} & -s_y s_{y1} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} \tag{20}$$

Which can be also represented as

$$b_{8\times 1} = A_{8\times 8} h_{8\times 1} \tag{21}$$

So,

$$h_{8\times 1} = A^{-1}_{8\times 8} b_{8\times 1} \tag{22}$$

From this equation all the h is evaluated and then H can be obtained from the (13).

9 Evaluation of Pixel Values

Now after the points have been mapped from one image to another, some of the pixel coordinates do not map to exact integral coordinates in the second image. Suppose a point G is mapped to a location between the junction of four points C, D, E and F. Then the pixel values of all these points are summed up to form a weighted averaged value, which is assigned as the value for G. The weights are the euclidean distances of the C, D, E and F points from the location of G. The following formula summarizes that.

$$g = \frac{c \times Dist_{xg} + d \times Dist_{dg} + e \times Dist_{eg} + f \times Dist_{fg}}{Dist_{xg} + Dist_{dg} + Dist_{eg} + Dist_{fg}} \tag{23}$$

```
#!/usr/bin/env python

import numpy as np, cv2, os, time

#============================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 3
#============================================================================

#============================================================================
# FUNCTIONS CREATED IN HW2.
#============================================================================

# Global variables that will mark the points in the image by mouse click.
ix, iy = -1, -1

#============================================================================

def markPoints( event, x, y, flags, params ):
    '''
    This is a function that is called on mouse callback.
    '''
    global ix, iy
    if event == cv2.EVENT_LBUTTONDOWN:
        ix, iy = x, y

#============================================================================

def selectPts( filePath=None ):
    '''
    This function opens the image and lets user select the points in it.
    These points are returned as a list.
    If the image is bigger than 640 x 480, it is displayed as 640 x 480. But
    the points are mapped and stored as per the original dimension of the image.
    The points are clicked by mouse on the image itself and they are stored in
    the listOfPts.
    '''

    global ix, iy

    img = cv2.imread( filePath )
    h, w = img.shape[0], img.shape[1]

    w1, h1, wRatio, hRatio, resized = w, h, 1, 1, False
    print( 'Image size: {}x{}'.format(w, h) )

    if w > 640:
        w1, resized = 640, True
        wRatio = w / w1
    if h > 480:
        h1, resized = 480, True
        hRatio = h / h1

    if resized:     img = cv2.resize( img, (w1, h1), \
                                      interpolation=cv2.INTER_AREA )

    cv2.namedWindow( 'Image' )
    cv2.setMouseCallback( 'Image', markPoints )  # Function to detect mouseclick
    key = ord('`')

#----------------------------------------------------------------------------

    listOfPts = []      # List to collect the selected points.
```

```
    while key & 0xFF != 27:         # Press esc to break.

        imgTemp = np.array( img )      # Temporary image.

        # Displaying all the points in listOfPts on the image.
        for i in range( len(listOfPts) ):
            cv2.circle( imgTemp, tuple(listOfPts[i]), 3, (0, 255, 0), -1 )

        # After clicking on the image, press any key (other than esc) to display
        # the point on the image.

        if ix > 0 and iy > 0:
            print( 'New point: ({}, {}). Press \'s\' to save.'.format(ix, iy) )
            cv2.circle( imgTemp, (ix, iy), 3, (0, 0, 255), -1 )
            # Since this point is not saved yet, so it is displayed on the
            # temporary image and not on actual img1.

        cv2.imshow( 'Image', imgTemp )
        key = cv2.waitKey(0)

        # If 's' is pressed then the point is saved to the listOfPts.
        if key == ord('s'):
            listOfPts.append( [ix, iy] )
            cv2.circle( imgTemp, (ix, iy), 3, (0, 255, 0), -1 )
            img1 = imgTemp
            ix, iy = -1, -1
            print( 'Point Saved.' )

        elif key == ord('d'):   ix, iy = -1, -1  # Delete point by pressing 'd'.

    # Map the selected points back to the size of original image using the
    # wRatio and hRatio (if they we resized earlier).
    if resized:   listOfPts = [ [ int( p[0] * wRatio ), int( p[1] * hRatio ) ] \
                                                        for p in listOfPts ]

    return listOfPts

#================================================================================

# Converts an input point (x, y) into homogeneous format (x, y, 1).
homogeneousFormat = lambda pt: [ pt[0], pt[1], 1 ]

#================================================================================

# Converts an input point in homogeneous coordinate (x, y, z) into planar form.
planarFormat = lambda pt: [ int(pt[0] / pt[2]), int(pt[1] / pt[2]) ]

#================================================================================

# Converts an input point in homogeneous coordinate (x, y, z) into planar form,
# But does not round them into integers, keeps them as floats.
planarFormatFloat = lambda pt: [ pt[0] / pt[2], pt[1] / pt[2] ]

#================================================================================

def homography( srcPts=None, dstPts=None ):
    '''
    The srcPts and dstPts are the list of points from which the 3x3 homography
    matrix (H) is to be decuced. The equation will be dstPts = H * srcPts.
    The srcPts and dstPts should be a list of at least 8 points (each point is
    in the homogeneous coordinate form, arranged as a sublist of 3 elements
    [x, y, 1]).
    The homography matrix is 3x3 but since only the ratios matter in homography,
    so one of the element can be taken as 1 and so there are only 8 unknowns.
    Here the last element of H is considered to be 1.
```

```
    '''

    # Number of source and destination points should be the same and should have
    # a one to one correspondence.
    if len(srcPts) != len(dstPts):
        print( 'srcPts and dstPts have different number of points. Aborting.' )
        return

    nPts = len( srcPts )

    # Creating matrix A and X (for the equation A * h = X).
    A, X = [], []
    for i in range( nPts ):
        # With each pair of points from srcPts and dstPts, 2 rows of A are made.
        xs, ys, xd, yd = srcPts[i][0], srcPts[i][1], dstPts[i][0], dstPts[i][1]
        row1 = [ xs, ys, 1, 0,  0,  0, -xs*xd, -ys*xd ]
        row2 = [ 0,  0,  0, xs, ys, 1, -xs*yd, -ys*yd ]
        A.append( row1 )
        A.append( row2 )
        X.append( xd )
        X.append( yd )
    #print( A )
    #print( X )

    # Converting A into nPts x nPts array and X into a nPts x 1 array.
    A, X = np.array( A ), np.reshape( X, ( nPts*2, 1 ) )
    Ainv = np.linalg.inv( A )
    h = np.matmul( Ainv, X )        # A * h = X, so h = Ainv * X.
    #print(h)

    # Appending a 1 for last element
    h = np.insert( h, nPts*2, 1 )
    H = np.reshape( h, (3,3) )      # Reshaping to 3x3.

    return H

#================================================================================

# The input is a 2 element list of the homography matrix H and the point pt.
# [H, pt]. Applies H to the point pt. pt is in homogeneous coordinate form.
applyHomography = lambda HandPt: np.matmul( HandPt[0], \
                                            np.reshape( HandPt[1], (3,1) ) )

#================================================================================

# Functions to find the min and max x and y coordinates from a list of points.
maxX = lambda listOfPts: sorted( listOfPts, key=lambda x: x[0] )[-1][0]
minX = lambda listOfPts: sorted( listOfPts, key=lambda x: x[0] )[0][0]
maxY = lambda listOfPts: sorted( listOfPts, key=lambda x: x[1] )[-1][1]
minY = lambda listOfPts: sorted( listOfPts, key=lambda x: x[1] )[0][1]

#================================================================================

def rectifyColor( pt=None, img=None ):
    '''
    This function takes in a point which is in float (not int) and an image and
    gives out a weighted average of the color of the surrounding pixels to
    which the point may be mapped to when converted from float to int.
    It is meant for those points which gets mapped to subpixel locations instead
    of mapping perfectly in an integer location in the given img.

    Format of pt is (x, y), like opencv.
    Returns the values as a numpy array.
    '''
    x1, y1 = np.floor( pt[0] ), np.floor( pt[1] )
```

```
    x2, y2 = np.ceil( pt[0] ),  np.floor( pt[1] )
    x3, y3 = np.floor( pt[0] ), np.ceil( pt[1] )
    x4, y4 = np.ceil( pt[0] ),  np.ceil( pt[1] )

    x, y = int( pt[0] ), int( pt[1] )   # Location where pt is to be mapped.

    # Distances of the potential location from the final location (x, y).
    # The + 0.0000001 is to prevent division by 0.
    dX1 = np.linalg.norm( np.array( [ x-x1, y-y1 ] ) ) + 0.0000001
    dX2 = np.linalg.norm( np.array( [ x-x2, y-y2 ] ) ) + 0.0000001
    dX3 = np.linalg.norm( np.array( [ x-x3, y-y3 ] ) ) + 0.0000001
    dX4 = np.linalg.norm( np.array( [ x-x4, y-y4 ] ) ) + 0.0000001

#-------------------------------------------------------------------------------

    h, w = img.shape[0], img.shape[1]

    #print( x1, y1, x2, y2, x3, y3, x4, y4 )

    # This is done so that while taking int(), the x, y dont go out of bound.
    x1 = int(x1) if int(x1) < w else w-1
    x2 = int(x2) if int(x2) < w else w-1
    x3 = int(x3) if int(x3) < w else w-1
    x4 = int(x4) if int(x4) < w else w-1

    y1 = int(y1) if int(y1) < h else h-1
    y2 = int(y2) if int(y2) < h else h-1
    y3 = int(y3) if int(y3) < h else h-1
    y4 = int(y4) if int(y4) < h else h-1

    # Color values at the above locations.
    C1, C2, C3, C4 = img[ int(y1) ][ int(x1) ], img[ int(y2) ][ int(x2) ], \
                     img[ int(y3) ][ int(x3) ], img[ int(y4) ][ int(x4) ]

#-------------------------------------------------------------------------------

    # Final color. So the farther the potential location is from the final
    # location, more is the corresponding distance, and hence less will be the
    # effect of the color of that location on the final color. In the above
    # calculations basically the following equation is evaluated.
    C = ( (C1 / dX1) + (C2 / dX2) + (C3 / dX3) + (C4 / dX4) ) / ( \
                     (1 / dX1) + (1 / dX2) + (1 / dX3) + (1 / dX4) )

    C = np.asarray( C, dtype=np.uint8 )

    return C

#===============================================================================

def mapImages( sourceImg=None, targetImg=None, pqrs=None, H=None ):
    '''
    This function takes in a source image, a target image and the four
    corner points (pqrs) of the target that defines the region where the source
    image is to be projected. It also takes in homography matrix from the target
    to the source image. Returns the projected image.

    pqrs should be a list of points. Each point in the list is a sublist of x
    and y coordinate of the point. These should be in planar (not homogeneous)
    coordinate.
    '''

    # Drawing a black polygon to make all the pixels in the target region = 0,
    # before mapping.
    targetImg = cv2.fillPoly( targetImg, np.array( [ pqrs ] ), (0,0,0) )
```

```
    sourceH, sourceW = sourceImg.shape[0], sourceImg.shape[1]

    processingTime = time.time()

    # Mapping the points.
    # Scanning only the region between the points p, q, r, s.
    for r in range( minY( pqrs ), maxY( pqrs ) ):
        for c in range( minX( pqrs ), maxX( pqrs ) ):

            if targetImg[r][c].all() == 0:
                # If point is in the 0 polygon.
                # which indicates that the point is in the black polygon where
                # the source image is to be projected.
                targetPt = homogeneousFormat( [ c, r ] )
                sourcePt = planarFormatFloat( applyHomography( [H, targetPt] ) )

                if sourcePt[0] < sourceW and sourcePt[1] < sourceH and \
                   sourcePt[0] > 0 and sourcePt[1] > 0:

                    # Mapping the source point pixel to the target point pixel.
                    targetImg[r][c] = sourceImg[ int( sourcePt[1] ) ][ \
                                                 int( sourcePt[0] ) ]
                    #targetImg[r][c] = rectifyColor( pt=sourcePt, img=sourceImg )
                    pass

    print( 'Time taken: {}'.format( time.time() - processingTime ) )

    return targetImg    # Returning target image after mapping target into it.

#================================================================================
# FUNCTIONS CREATED IN HW3.
#================================================================================

def dimsOfModifiedImg( img=None, H=None ):
    '''
    This function takes in an entire image and a homography matrix and returns
    what the dimensions of the output image (after applying this homography)
    should be, to accomodate all the modified point locations along with the
    min and max x and y coordinates of the modified image.
    '''

    imgH, imgW, _ = img.shape

    # The output image will be of a different dimension than the input image.
    # But the corner points will always stay contained inside the final image.
    # In order to find the dimensions of the final image we find out where the
    # corners of the input image will be mapped.

    tlc = [ 0, 0, 1 ]   # Top left corner of the image. (x, y, 1 format)
    trc = [ imgW, 0, 1 ]   # Top right corner of the image. (x, y, 1 format)
    brc = [ imgW, imgH, 1 ]   # Bottom right corner of the image. (x, y, 1 format)
    blc = [ 0, imgH, 1 ]   # Bottom left corner of the image. (x, y, 1 format)

    # Applying homography.
    tlcNew = applyHomography( [H, tlc] )    # Top left corner in new image.
    tlcNew = planarFormat( tlcNew )
    trcNew = applyHomography( [H, trc] )    # Top right corner in new image.
    trcNew = planarFormat( trcNew )
    brcNew = applyHomography( [H, brc] )    # Bottom right corner in new image.
    brcNew = planarFormat( brcNew )
    blcNew = applyHomography( [H, blc] )    # Bottom left corner in new image.
    blcNew = planarFormat( blcNew )

    # Making a list of the x and y coordinates of the corner locations in new image.
    listOfX = [ tlcNew[0], trcNew[0], brcNew[0], blcNew[0] ]
```

```
    listOfY = [ tlcNew[1], trcNew[1], brcNew[1], blcNew[1] ]

    maxX, maxY = max( listOfX ), max( listOfY )
    minX, minY = min( listOfX ), min( listOfY )

    # Now so that the minX and minY map to 0, 0 in the modified image, so those
    # locations should be subtracted from the max locations.
    # If maxX is 5 and minX is -2, then image should be between 0 and 5 - (-2) = 7.
    # If maxX is 5 and minX is 1, then image should be between 0 and 5 - (1) = 4.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    dimX, dimY = maxX - minX + 1, maxY - minY + 1

    return [ dimX, dimY, maxX, maxY, minX, minY ]

#================================================================================

def createLAandLB( line1ProjRemovedPerp=None, line2ProjRemovedPerp=None, \
                   line3ProjRemovedPerp=None, line4ProjRemovedPerp=None ):
    '''
    This function creates the LA and LB matrices, such that LA * [s11, s12]^T = LB,
    that will be used to find the S matrix later.
    '''
    l1, l2 = line1ProjRemovedPerp[0], line1ProjRemovedPerp[1]
    m1, m2 = line2ProjRemovedPerp[0], line2ProjRemovedPerp[1]
    l3, l4 = line3ProjRemovedPerp[0], line3ProjRemovedPerp[1]
    m3, m4 = line4ProjRemovedPerp[0], line4ProjRemovedPerp[1]

    LA = [ [ l1*m1, l1*m2 + l2*m1 ], [ l3*m3, l3*m4 + l4*m3 ] ]
    LB = [ -1*l2*m2, -1*l4*m4 ]

    return np.array(LA), np.array(LB)

#================================================================================

def modifyImg3( sourceImg=None, H=None ):
    '''
    This function takes in a source image, and a homography of the target to source.
    Then it applies the inverse of this homography to the source (which is a
    deformed image) and finds the location where the corner points will be mapped
    when the H is applied to the source image. Thencreates a target image with
    these dimensions. Now it takes all the locations of this blank target image
    one by one and applies the homography to them to find the location of the
    corresponding points in the source image. Then takes the pixel values of these
    points from the source image and replace the points in the target image.
    It also shifts the locations of the points mapped to the negative coordinates.
    '''

    # Drawing a black polygon to make all the pixels in the target region = 0,
    # before mapping.

    tgtW, tgtH, tgtMaxX, tgtMaxY, tgtMinX, tgtMinY = \
            dimsOfModifiedImg( sourceImg, H=np.linalg.inv( H ) )

    #print( tgtW, tgtH, tgtMaxX, tgtMaxY, tgtMinX, tgtMinY )

    targetImg = np.zeros( (tgtH, tgtW, 3), dtype=np.uint8 )

    sourceH, sourceW = sourceImg.shape[0], sourceImg.shape[1]

    processingTime = time.time()

    # Mapping the points.
    for r in range( tgtMinY, tgtMaxY + 1 ):
        for c in range( tgtMinX, tgtMaxX + 1 ):
```

```
            targetPt = homogeneousFormat( [ c, r ] )
            sourcePt = planarFormatFloat( applyHomography( [H, targetPt] ) )

            if sourcePt[0] < sourceW and sourcePt[1] < sourceH and \
               sourcePt[0] > 0 and sourcePt[1] > 0:

                # Mapping the source point pixel to the target point pixel.
                targetImg[ r - tgtMinY ][ c - tgtMinX ] = sourceImg[ \
                               int( sourcePt[1] ) ][ int( sourcePt[0] ) ]
                #targetImg[r][c] = rectifyColor( pt=sourcePt, img=sourceImg )
                pass

    print( 'Time taken: {}'.format( time.time() - processingTime ) )

    return targetImg    # Returning target image after mapping target into it.

#================================================================================


if __name__ == '__main__':

    # TASK 1.1.

    filePath = './HW3Pics'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[238, 1121], [2207, 20], [2332, 1401], [105, 1753]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[239, 58], [338, 73], [336, 276], [236, 282]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

    pqrsFig1Orig = [[0, 0], [1260, 0], [1260, 560], [0, 560]]
    #print( 'Points of {}: {}\n'.format( filename1Orig, pqrsFig1Orig ) )

    pqrsFig2Orig = [[0, 0], [160, 0], [160, 320], [0, 320]]
    #print( 'Points of {}: {}\n'.format( filename2Orig, pqrsFig2Orig ) )

#--------------------------------------------------------------------------------

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]
    pqrsHomFmt1Orig = [ homogeneousFormat( pt ) for pt in pqrsFig1Orig ]
    pqrsHomFmt2Orig = [ homogeneousFormat( pt ) for pt in pqrsFig2Orig ]

#--------------------------------------------------------------------------------

    # Finding the homography.

    # Homography between fig1 and Original Fig1.
    Hbetw1OrigTo1 = homography( srcPts=pqrsHomFmt1Orig, dstPts=pqrsHomFmt1 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename1, Hbetw1OrigTo1 ) )

    # Homography between fig2 and Original Fig2.
    Hbetw2OrigTo2 = homography( srcPts=pqrsHomFmt2Orig, dstPts=pqrsHomFmt2 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename2, Hbetw2OrigTo2 ) )
```

```
#-------------------------------------------------------------------------------

    # Implanting the target into fig 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw1OrigTo1

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # Implanting the target into fig 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw2OrigTo2

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    print( tgtH, tgtW )
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#===============================================================================

    # TASK 2.1.

    filePath = './PicsSelf'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[661, 1689], [965, 1577], [972, 2409], [661, 2296]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    #pqrsFig2 = [[1456, 492], [2873, 599], [2905, 1564], [1547, 2023]]
    pqrsFig2 = [[127, 12], [429, 62], [432, 182], [169, 205]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

    pqrsFig1Orig = [[0, 0], [209, 0], [209, 130], [0, 130]]
    #print( 'Points of {}: {}\n'.format( filename1Orig, pqrsFig1Orig ) )

    pqrsFig2Orig = [[0, 0], [186, 0], [186, 123], [0, 123]]
```

```
    #print( 'Points of {}: {}\n'.format( filename2Orig, pqrsFig2Orig ) )

#-------------------------------------------------------------------------------

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]
    pqrsHomFmt1Orig = [ homogeneousFormat( pt ) for pt in pqrsFig1Orig ]
    pqrsHomFmt2Orig = [ homogeneousFormat( pt ) for pt in pqrsFig2Orig ]

#-------------------------------------------------------------------------------

    # Finding the homography.

    # Homography between fig1 and Original Fig1.
    Hbetw1OrigTo1 = homography( srcPts=pqrsHomFmt1Orig, dstPts=pqrsHomFmt1 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename1, Hbetw1OrigTo1 ) )

    # Homography between fig2 and Original Fig2.
    Hbetw2OrigTo2 = homography( srcPts=pqrsHomFmt2Orig, dstPts=pqrsHomFmt2 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename2, Hbetw2OrigTo2 ) )

#-------------------------------------------------------------------------------

    # Implanting the target into fig 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw1OrigTo1

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # Implanting the target into fig 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw2OrigTo2

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    print( tgtH, tgtW )
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
```

```
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#================================================================================

    # TASK 1.2.1.   Remove projective distortion.

    filePath = './HW3Pics'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[2029, 405], [251, 1251], [2085, 1441], [149, 1745], [149, 1745], [251, 1251], [2085, 1441],
[2029, 405]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[46, 121], [139, 128], [209, 355], [344, 339], [237, 282], [237, 57], [498, 267], [495, 26]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

#--------------------------------------------------------------------------------

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

#--------------------------------------------------------------------------------

    # Removing projective distortion fig 1.

    # Finding the line joining the points selected in the figure 1.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig1Para = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Para = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Para = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Para = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Calculating vanishing points.
    VP1fig1 = np.cross( line1Fig1Para, line2Fig1Para )
    VP2fig1 = np.cross( line3Fig1Para, line4Fig1Para )

    # The magnitude of the vanishing points are too large. But since they are
    # the physical points in homogeneous coordinates, so we can always make the
    # magnitude smaller by dividing by one of the coordinate values and express
    # them equvalently in [x, y, 1] format (instead of having some
    # [very_large_X, very_large_Y, large_number] format).

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig1 = VP1fig1 / VP1fig1[2]
    VP2fig1 = VP2fig1 / VP2fig1[2]

    # Calculating vanishing line.
    VLfig1 = np.cross( VP1fig1, VP2fig1 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig1 = VLfig1 / VLfig1[2]

    # Calculating the Homography matrix.
    Hbetw1OrigTo1Proj = np.eye( 3 )
    Hbetw1OrigTo1Proj[2] = VLfig1

    print( 'Hbetw1OrigTo1Proj: \n{}\n'.format( Hbetw1OrigTo1Proj ) )
```

```
#-------------------------------------------------------------------------------

    # Removing projective distortion fig 2.

    # Finding the line joining the points selected in the figure 2.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig2Para = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Para = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Para = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Para = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Calculating vanishing points.
    VP1fig2 = np.cross( line1Fig2Para, line2Fig2Para )
    VP2fig2 = np.cross( line3Fig2Para, line4Fig2Para )

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig2 = VP1fig2 / VP1fig2[2]
    VP2fig2 = VP2fig2 / VP2fig2[2]

    # Calculating vanishing line.
    VLfig2 = np.cross( VP1fig2, VP2fig2 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig2 = VLfig2 / VLfig2[2]

    # Calculating the Homography matrix.
    Hbetw2OrigTo2Proj = np.eye( 3 )
    Hbetw2OrigTo2Proj[2] = VLfig2

    print( 'Hbetw2OrigTo2Proj: \n{}\n'.format( Hbetw2OrigTo2Proj ) )

#-------------------------------------------------------------------------------

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )
```

```
    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]
```

```
    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # TASK 1.2.2.   Remove affine distortion.

    # Removing affine distortion figure 1.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[238, 1117], [2215, 16], [2361, 1802], [2219, 125], [1255, 765], [1668, 1300], [1668, 571],
[1210, 1413]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[237, 58], [338, 74], [336, 276], [338, 86], [245, 72], [326, 174], [326, 82], [245, 167]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

#-------------------------------------------------------------------------------

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

#-------------------------------------------------------------------------------

    # Removing affine distortion from figure 1.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig1Perp = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Perp = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Perp = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Perp = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw1OrigTo1Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig1ProjRemovedPerp = np.matmul( HinvT, line1Fig1Perp )
    line2Fig1ProjRemovedPerp = np.matmul( HinvT, line2Fig1Perp )
    line3Fig1ProjRemovedPerp = np.matmul( HinvT, line3Fig1Perp )
    line4Fig1ProjRemovedPerp = np.matmul( HinvT, line4Fig1Perp )

    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig1ProjRemovedPerp /= line1Fig1ProjRemovedPerp[2]
    line2Fig1ProjRemovedPerp /= line2Fig1ProjRemovedPerp[2]
    line3Fig1ProjRemovedPerp /= line3Fig1ProjRemovedPerp[2]
    line4Fig1ProjRemovedPerp /= line4Fig1ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
```

```
    LA, LB = createLAandLB( line1Fig1ProjRemovedPerp, line2Fig1ProjRemovedPerp, \
                      line3Fig1ProjRemovedPerp, line4Fig1ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )

    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
    Hbetw1OrigTo1Aff = [ [ A[0,0], A[1,0], 0 ], [ A[0,1], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw1OrigTo1Aff = np.array( Hbetw1OrigTo1Aff )
    Hbetw1OrigTo1Aff = np.linalg.inv( Hbetw1OrigTo1Aff )
    print( 'Hbetw1OrigTo1Aff: \n{}\n'.format( Hbetw1OrigTo1Aff ) )

    Hbetw1OrigTo1 = np.matmul( Hbetw1OrigTo1Aff, Hbetw1OrigTo1Proj )
    print( 'Hbetw1OrigTo1: \n{}\n'.format( Hbetw1OrigTo1 ) )

#--------------------------------------------------------------------------------

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
```

```
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # Removing affine distortion figure 2.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig2Perp = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Perp = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Perp = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Perp = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw2OrigTo2Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig2ProjRemovedPerp = np.matmul( HinvT, line1Fig2Perp )
    line2Fig2ProjRemovedPerp = np.matmul( HinvT, line2Fig2Perp )
    line3Fig2ProjRemovedPerp = np.matmul( HinvT, line3Fig2Perp )
    line4Fig2ProjRemovedPerp = np.matmul( HinvT, line4Fig2Perp )

    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig2ProjRemovedPerp /= line1Fig2ProjRemovedPerp[2]
    line2Fig2ProjRemovedPerp /= line2Fig2ProjRemovedPerp[2]
    line3Fig2ProjRemovedPerp /= line3Fig2ProjRemovedPerp[2]
    line4Fig2ProjRemovedPerp /= line4Fig2ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
    LA, LB = createLAandLB( line1Fig2ProjRemovedPerp, line2Fig2ProjRemovedPerp, \
                        line3Fig2ProjRemovedPerp, line4Fig2ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )

    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
```

```
    Hbetw2OrigTo2Aff = [ [ A[0,0], A[0,1], 0 ], [ A[1,0], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw2OrigTo2Aff = np.array( Hbetw2OrigTo2Aff )
    Hbetw2OrigTo2Aff = np.linalg.inv( Hbetw2OrigTo2Aff )
    print( 'Hbetw2OrigTo2Aff: \n{}\n'.format( Hbetw2OrigTo2Aff ) )

    Hbetw2OrigTo2 = np.matmul( Hbetw2OrigTo2Aff, Hbetw2OrigTo2Proj )
    print( 'Hbetw2OrigTo2: \n{}\n'.format( Hbetw2OrigTo2 ) )

#-------------------------------------------------------------------------------

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#===============================================================================
```

```
    # TASK 2.2.1.   Remove projective distortion.

    filePath = './PicsSelf'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    #pqrsFig1 = [[2029, 405], [251, 1251], [2085, 1441], [149, 1745], [149, 1745], [251, 1251], [2085, 1441],
[2029, 405]]
    #pqrsFig1 = [[661, 1689], [965, 1577], [972, 2409], [661, 2296], [661, 2296], [965, 1577], [972, 2409],
[661, 1689]]
    pqrsFig1 = [[661, 1672], [965, 1560], [665, 2279], [972, 2409], [1144, 1100], [1129, 3172], [1868, 606],
[1820, 3839]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    #pqrsFig2 = [[46, 121], [139, 128], [209, 355], [344, 339], [237, 282], [237, 57], [498, 267], [495, 26]]
    #pqrsFig2 = [[1456, 492], [2873, 599], [2905, 1564], [1547, 2023], [1547, 2023], [2873, 599], [2905,
1564], [1456, 492]]
    #pqrsFig2 = [[1456, 497], [2860, 599], [1547, 2013], [2899, 1564], [1592, 580], [1670, 1896], [2827,
648], [2827, 1535]]
    pqrsFig2 = [[128, 12], [427, 63], [167, 206], [431, 183], [129, 25], [166, 195], [428, 78], [431, 169]]
    #pqrsFig2 = [[234, 52], [780, 268], [307, 892], [789, 784], [235, 116], [303, 840], [782, 333], [789,
710]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

#-------------------------------------------------------------------------------

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

#-------------------------------------------------------------------------------

    # Removing projective distortion fig 1.

    # Finding the line joining the points selected in the figure 1.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig1Para = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Para = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Para = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Para = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Calculating vanishing points.
    VP1fig1 = np.cross( line1Fig1Para, line2Fig1Para )
    VP2fig1 = np.cross( line3Fig1Para, line4Fig1Para )

    # The magnitude of the vanishing points are too large. But since they are
    # the physical points in homogeneous coordinates, so we can always make the
    # magnitude smaller by dividing by one of the coordinate values and express
    # them equvalently in [x, y, 1] format (instead of having some
    # [very_large_X, very_large_Y, large_number] format).

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig1 = VP1fig1 / VP1fig1[2]
    VP2fig1 = VP2fig1 / VP2fig1[2]

    # Calculating vanishing line.
    VLfig1 = np.cross( VP1fig1, VP2fig1 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig1 = VLfig1 / VLfig1[2]
```

```
    # Calculating the Homography matrix.
    Hbetw1OrigTo1Proj = np.eye( 3 )
    Hbetw1OrigTo1Proj[2] = VLfig1

    print( 'Hbetw1OrigTo1Proj: \n{}\n'.format( Hbetw1OrigTo1Proj ) )

#-------------------------------------------------------------------------------

    # Removing projective distortion fig 2.

    # Finding the line joining the points selected in the figure 2.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig2Para = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Para = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Para = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Para = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Calculating vanishing points.
    VP1fig2 = np.cross( line1Fig2Para, line2Fig2Para )
    VP2fig2 = np.cross( line3Fig2Para, line4Fig2Para )

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig2 = VP1fig2 / VP1fig2[2]
    VP2fig2 = VP2fig2 / VP2fig2[2]

    # Calculating vanishing line.
    VLfig2 = np.cross( VP1fig2, VP2fig2 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig2 = VLfig2 / VLfig2[2]

    # Calculating the Homography matrix.
    Hbetw2OrigTo2Proj = np.eye( 3 )
    Hbetw2OrigTo2Proj[2] = VLfig2

    print( 'Hbetw2OrigTo2Proj: \n{}\n'.format( Hbetw2OrigTo2Proj ) )

#-------------------------------------------------------------------------------

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.
```

```
    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
```

```
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # TASK 2.2.2.   Remove affine distortion.

    # Removing affine distortion figure 1.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[665, 1672], [968, 1560], [961, 2435], [965, 1715], [1137, 1109], [1831, 3007], [1872, 606],
[1140, 2608]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[129, 12], [427, 63], [432, 181], [429, 78], [169, 204], [356, 187], [355, 187], [338, 47]]
    #pqrsFig2 = [[129, 13], [428, 63], [431, 182], [430, 73], [170, 205], [359, 188], [359, 188], [339, 47]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

#-------------------------------------------------------------------------------

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

#-------------------------------------------------------------------------------

    # Removing affine distortion from figure 1.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig1Perp = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Perp = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Perp = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Perp = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw1OrigTo1Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig1ProjRemovedPerp = np.matmul( HinvT, line1Fig1Perp )
    line2Fig1ProjRemovedPerp = np.matmul( HinvT, line2Fig1Perp )
    line3Fig1ProjRemovedPerp = np.matmul( HinvT, line3Fig1Perp )
    line4Fig1ProjRemovedPerp = np.matmul( HinvT, line4Fig1Perp )
```

```
    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig1ProjRemovedPerp /= line1Fig1ProjRemovedPerp[2]
    line2Fig1ProjRemovedPerp /= line2Fig1ProjRemovedPerp[2]
    line3Fig1ProjRemovedPerp /= line3Fig1ProjRemovedPerp[2]
    line4Fig1ProjRemovedPerp /= line4Fig1ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
    LA, LB = createLAandLB( line1Fig1ProjRemovedPerp, line2Fig1ProjRemovedPerp, \
                      line3Fig1ProjRemovedPerp, line4Fig1ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )

    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
    Hbetw1OrigTo1Aff = [ [ A[0,0], A[1,0], 0 ], [ A[0,1], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw1OrigTo1Aff = np.array( Hbetw1OrigTo1Aff )
    Hbetw1OrigTo1Aff = np.linalg.inv( Hbetw1OrigTo1Aff )
    print( 'Hbetw1OrigTo1Aff: \n{}\n'.format( Hbetw1OrigTo1Aff ) )

    Hbetw1OrigTo1 = np.matmul( Hbetw1OrigTo1Aff, Hbetw1OrigTo1Proj )
    print( 'Hbetw1OrigTo1: \n{}\n'.format( Hbetw1OrigTo1 ) )

#--------------------------------------------------------------------------------

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1
```

```
    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

#-------------------------------------------------------------------------------

    # Removing affine distortion figure 2.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig2Perp = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Perp = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Perp = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Perp = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw2OrigTo2Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig2ProjRemovedPerp = np.matmul( HinvT, line1Fig2Perp )
    line2Fig2ProjRemovedPerp = np.matmul( HinvT, line2Fig2Perp )
    line3Fig2ProjRemovedPerp = np.matmul( HinvT, line3Fig2Perp )
    line4Fig2ProjRemovedPerp = np.matmul( HinvT, line4Fig2Perp )

    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig2ProjRemovedPerp /= line1Fig2ProjRemovedPerp[2]
    line2Fig2ProjRemovedPerp /= line2Fig2ProjRemovedPerp[2]
    line3Fig2ProjRemovedPerp /= line3Fig2ProjRemovedPerp[2]
    line4Fig2ProjRemovedPerp /= line4Fig2ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
    LA, LB = createLAandLB( line1Fig2ProjRemovedPerp, line2Fig2ProjRemovedPerp, \
                        line3Fig2ProjRemovedPerp, line4Fig2ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )
```

```
    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
    Hbetw2OrigTo2Aff = [ [ A[0,0], A[0,1], 0 ], [ A[1,0], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw2OrigTo2Aff = np.array( Hbetw2OrigTo2Aff )
    Hbetw2OrigTo2Aff = np.linalg.inv( Hbetw2OrigTo2Aff )
    print( 'Hbetw2OrigTo2Aff: \n{}\n'.format( Hbetw2OrigTo2Aff ) )

    Hbetw2OrigTo2 = np.matmul( Hbetw2OrigTo2Aff, Hbetw2OrigTo2Proj )
    print( 'Hbetw2OrigTo2: \n{}\n'.format( Hbetw2OrigTo2 ) )

#-------------------------------------------------------------------------------

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
```

```
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```