ECE 66100 HW3 Report

Zhengxin Jiang
(jiang839@purdue.edu)

September 18, 2022

1 Task 1.1: Point to Point Method

The point-to-point method removes the projective distortion by manually setting the corresponding points in the original scene, then calculating the homography using these points.

For a given area in the distorted image which the height h and width w are known to us, we can set the points in the original scene as $(0,0)$, $(0,h)$,(w,h) and $(w,0)$.

With the corresponding points, we use the linear system used in homework2 to solve the homography H:

$$PH = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' & x_1' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' & y_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' & x_2' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' & y_2' \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3x_3' & y_3x_3' & x_3' \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3y_3' & y_3y_3' & y_3' \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4x_4' & y_4x_4' & x_4' \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4y_4' & y_4y_4' & y_4' \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$H = P^{-1} * \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

To completely show the recovered image, we need to know the size and also the coordinate offset of the recovered image. Taking the four corners of the distorted image, we calculate their coordinates in original scene (x_0, y_0), (x_1, y_1),(x_2, y_2) (x_3, y_3) using the homography H. The size and offset are then given by:

$$width = max(X) - min(X)$$

$$height = max(Y) - min(Y)$$

$$offset = (min(X), min(Y)) - (0,0) = (min(X), min(Y))$$

Finally we map the distorted image back into the original scene using the equation with offset:

$$X' = H(X + offset)$$

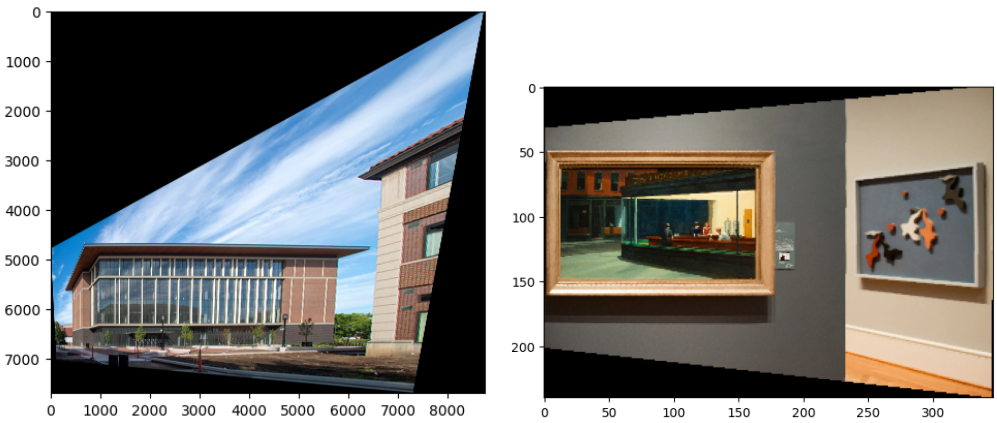

Figure 1: Result images of the point-to-point method

2 Task 1.2: The Two Step Method

In the two step Method, the first step is to remove the projective distortion and then the second step is to remove the affine distortion.

Remove projective distortion

To remove the projective distortion, we want to map the vanishing line back to l_∞. To find the vanishing line in the distorted image, we first find two pairs of orthogonal lines (l_1, l_2) and (l_3, l_4). Then we can calculate two points on the vanishing line

$$P_1 = l_1 \times l_2$$
$$P_2 = l_3 \times l_4$$

The vanishing line is then given by

$$vl = P_1 \times P_2$$

According to lecture notes, the homography that maps the vanishing line to l_{inf} is

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

The image recovery process is the same as described in task 1.1. One thing to notice is that we use H^{-1} since we are mapping from the image to the original scene.

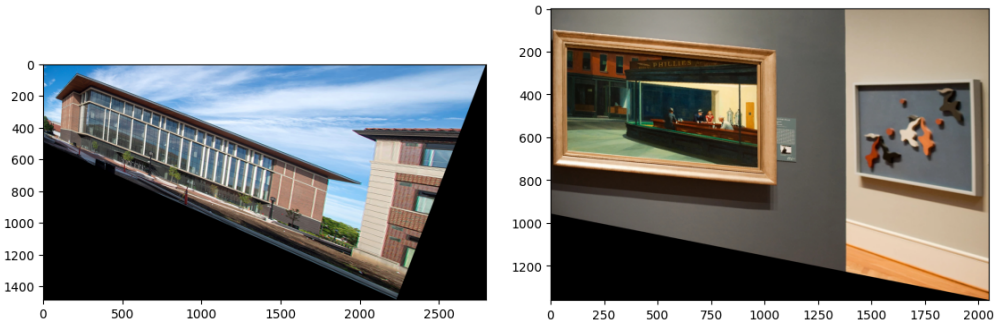

Figure 2: Result images after first step

Remove affine distortion

According to the lecture notes, we can express the angle of two lines in the form

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

With two orthogonal lines in the image plane we can write

$$l'^T H C_\infty^* H^T m' = 0$$

Let $H = \begin{bmatrix} A & t=0 \\ 0^T & 1 \end{bmatrix}$, The equation becomes

$$\begin{bmatrix} l'_1 & l'_2 & l'_3 \end{bmatrix} \begin{bmatrix} AA^T & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} = 0$$

We denote AA^T as $S = \begin{bmatrix} s_a & s_b \\ s_b & 1 \end{bmatrix}$. Since S has 2 DoF, we need two pairs of orthogonal lines to solve for S. Once S is calculated, A can be recovered by using SVD

$$S = UDU^T$$

$$A = U\sqrt{D}U^T$$

The image recovery process is the same as described in task 1.1.

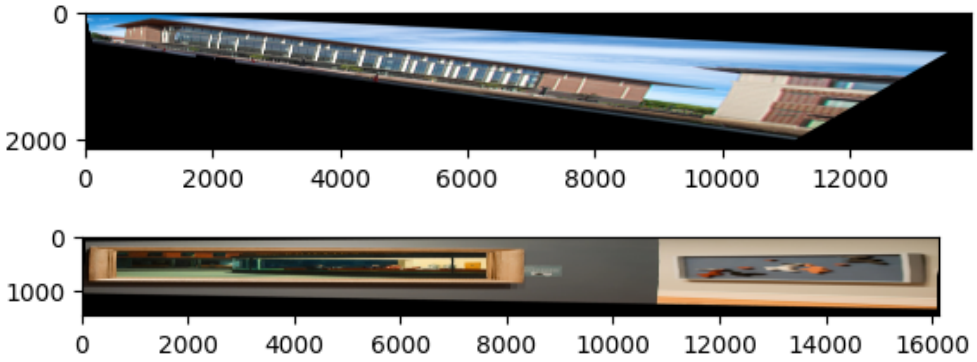

Figure 3: Result images after second step

3 Task 1.3: The One Step Method

In the one step method, we try to find the homography that maps $C^{*'}$ back to C^*_∞.

Taking the projection quation of the dual degenerate conic

$$C^{*'} = HC^*_\infty H^T$$

Substituting into the equation we derived in the previous section and we have

$$l'^T C^{*'} m' = 0$$

Let $C^{*'} = \begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & 1 \end{bmatrix}$, we write the equation in the form

$$\begin{bmatrix} l'_1 m'_1 & \frac{l'_1 m'_2 + l'_2 m'_1}{2} & l'_2 m'_2 & \frac{l'_1 m'_3 + l'_3 m'_1}{2} & \frac{l'_2 m'_3 + l'_3 m'_2}{2} & l'_3 m'_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ 1 \end{bmatrix} = 0$$

Since $C^{*'}$ has 5 DoF, we need 5 pairs of orthogonal lines to solve the linear system. I set the 5th pair of lines by selecting a square area in the original scene and use the two diagonal lines.

Once $C^{*'}$ is calculated, let $H = \begin{bmatrix} A & t = 0 \\ v^T & 1 \end{bmatrix}$ and we use the following relationships for solving A and v

$$AA^T = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}$$

$$Av = \begin{bmatrix} d/2 \\ e/2 \end{bmatrix}$$

The image recovery process is the same as described in task 1.1.

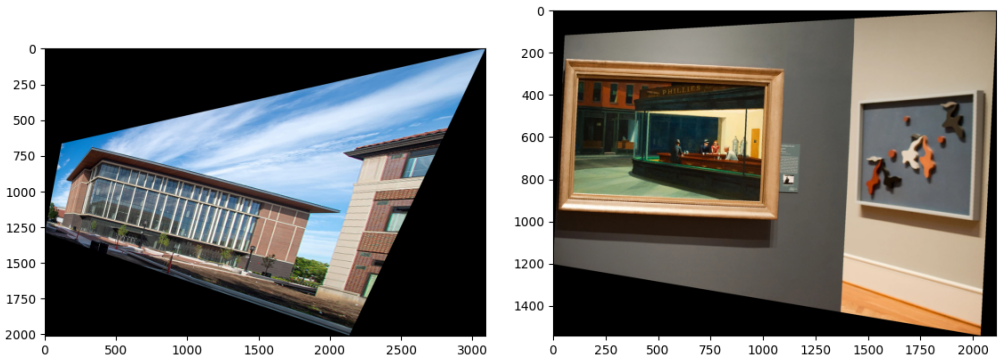

Figure 4: Result images of the one step method

4 Results of Task 2

For task 2 I took two pictures, one is a monitor and one is a Protect Purdue sticker.

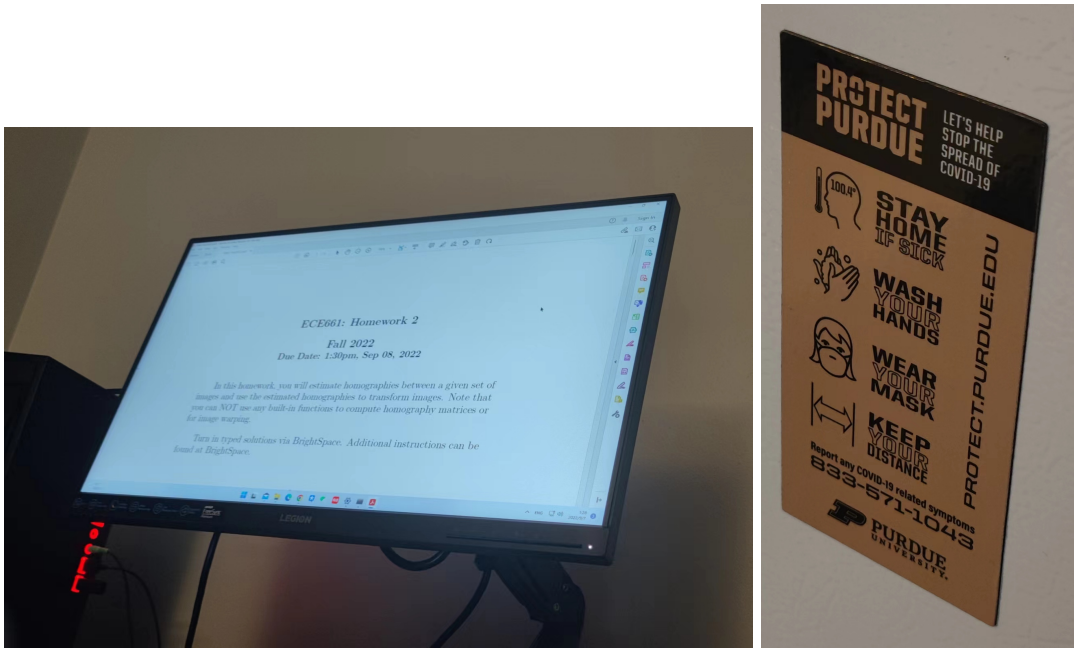

Figure 5: Two raw images for task2

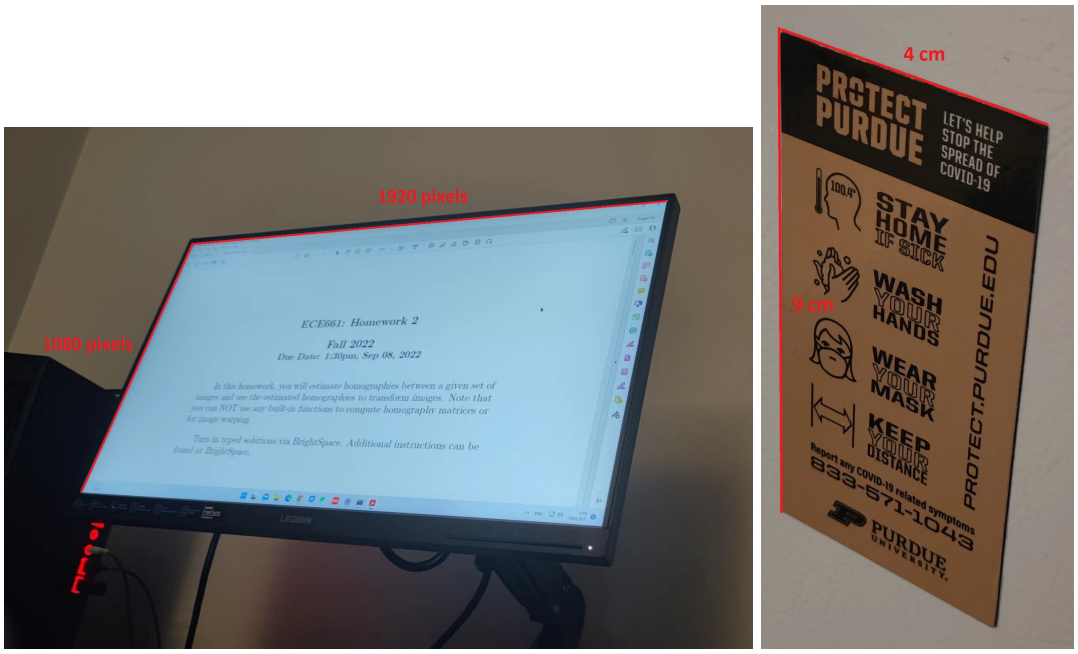

Figure 6: World coordinates of the two images

2.1 Point-to-point Method

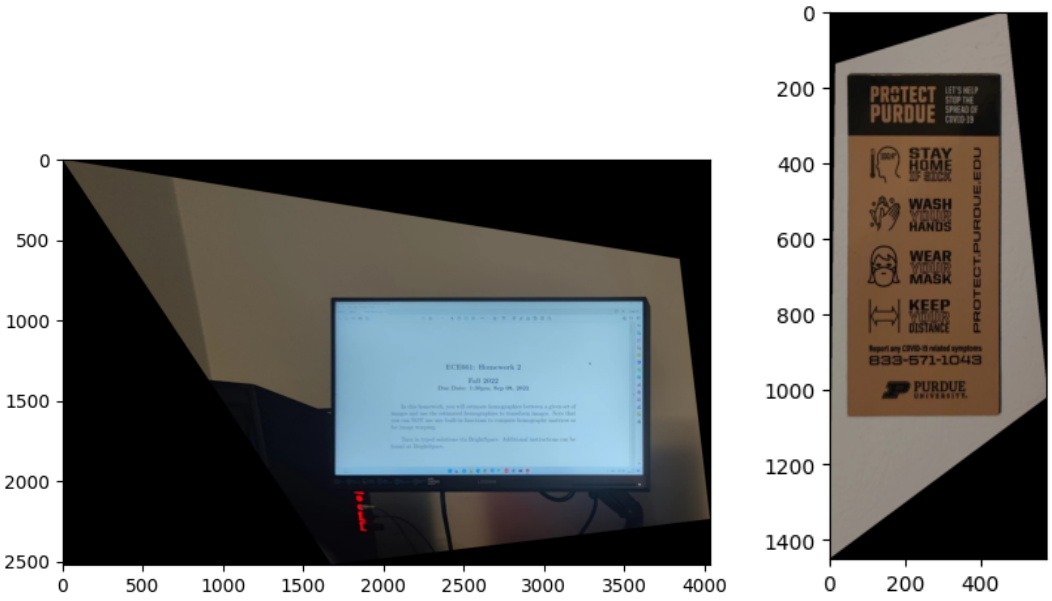

Figure 7: Result images of the point-to-point method

2.2 Two-step Method

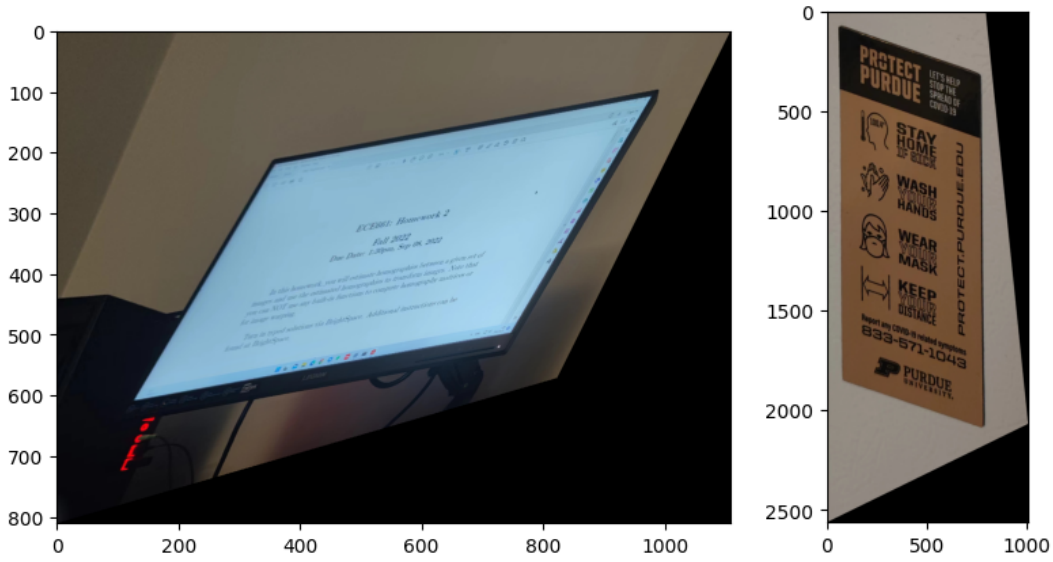

Figure 8: Result images after first step

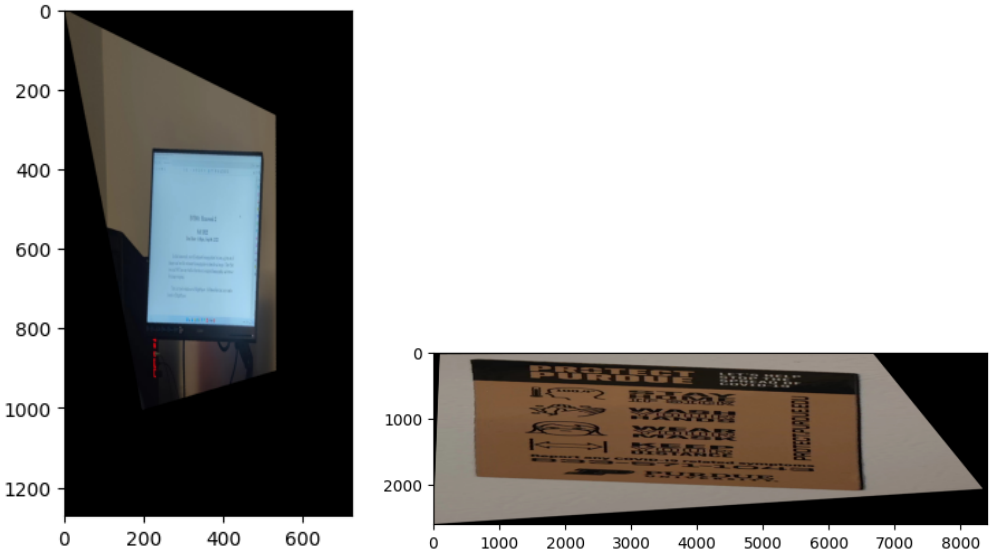

Figure 9: Result images after second step

2.3 One-step Method

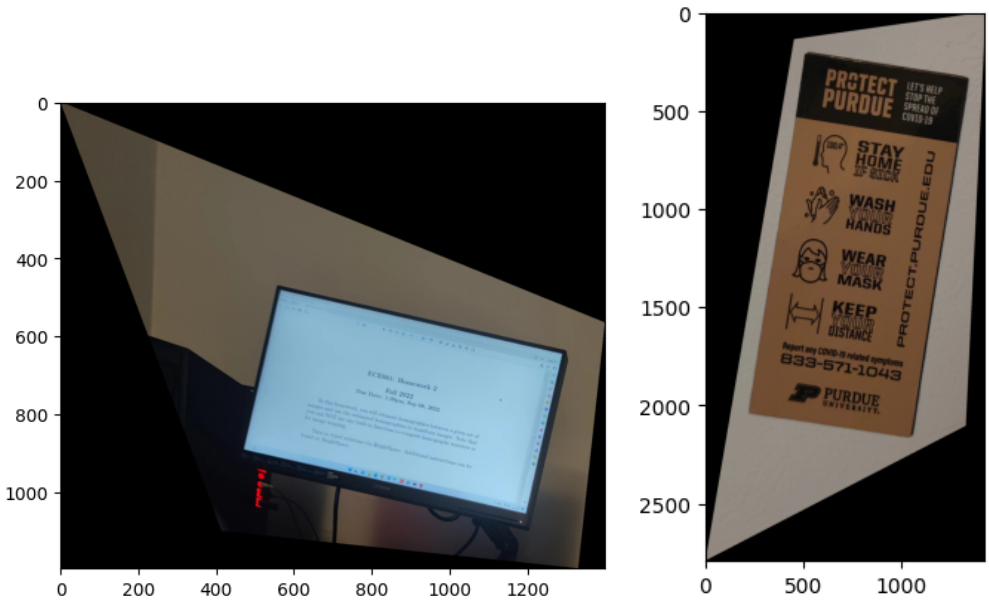

Figure 10: Result images of the one step method

5 Comments on the results

The results of the point-to-point method and the two-step method are very good. For the two-step method, the affine-remove step makes the objects squeezed. This can be that the affine-remove step just corrects the angles and does not directly map from image plane to world plane like other methods.

6 Use of Vetorized Operations

Some basic vectorized operations include the transformations between lines and points. The np.cross() method is used on two vectors of points or lines. One further improvement can be on the image recovery process. Since we will apply the homography on every pixels in the image, we can put the image pixels in to a matrix instead looping through the two axes of the image.

7 Source code

```
# ECE661 HW3
# Zhengxin Jiang
# jiang839

import cv2 as cv
import numpy as np
import matplotlib . pyplot as plt


### function definetion ###



# The function takes two arrays of points and return the homography
def findHomograpyMatrix_P2P(points_1, points_2):

    # create a linear system to solve H
    P = np.zeros((9,9))

    for i in range(4):

        P[2*i, 0] = -points_1[i][0]
        P[2*i, 1] = -points_1[i][1]
        P[2*i, 2] = -1
        P[2*i, 6] = points_1[i][0]*points_2[i][0]
        P[2*i, 7] = points_1[i][1]*points_2[i][0]
        P[2*i, 8] = points_2[i][0]

        P[2*i+1, 3] = -points_1[i][0]
        P[2*i+1, 4] = -points_1[i][1]
        P[2*i+1, 5] = -1
        P[2*i+1, 6] = points_1[i][0]*points_2[i][1]
        P[2*i+1, 7] = points_1[i][1]*points_2[i][1]
        P[2*i+1, 8] = points_2[i][1]

    P[8, 8] = 1

    # take the last col of P inverse
    H = np.linalg.inv(P)[:, 8]
    H = np.reshape(H, (3,3))

    return H


# The function finds the homography that creates projective distortion based on the given
    point set
def findHomograpyMatrix_Projective(points):

    l1, l2 = np.cross(points[0], points[3]), np.cross(points[1], points[2])
    l3, l4 = np.cross(points[0], points[1]), np.cross(points[3], points[2])
    p, q = np.cross(l1, l2), np.cross(l3, l4)
    p, q = p/p[2], q/q[2]
```

```
    vl = np.cross(p, q)
    vl = vl/vl[2]

    H = np.identity(3)
    H[2] = vl

    # H maps vl to vl_inf, we want the inverse H
    return np.linalg.inv(H)


# The function finds the homography that creates affine distortion based on the given
    point set
def findHomograpyMatrix_Affine(points):

    l1 = np.cross(points.T[0], points.T[1])
    l2 = np.cross(points.T[0], points.T[3])
    l3 = np.cross(points.T[1], points.T[2])
    l1 = l1/l1[2]
    l2 = l2/l2[2]
    l3 = l3/l3[2]

    # create a linear system to solve S
    A_ = np.zeros((2,2))
    A_[0] = [l1[0]*l2[0], l1[0]*l2[1]+l1[1]*l2[0]]
    A_[1] = [l1[0]*l3[0], l1[0]*l3[1]+l1[1]*l3[0]]

    b = [-l1[1]*l2[1], -l1[1]*l3[1]]

    S_ = np.linalg.solve(A_,b)

    # reshape S into 2x2 matrix
    S = np.ones((2,2))
    S[0][0] = S_[0]
    S[0][1] = S_[1]
    S[1][0] = S_[1]

    # solve A from S
    u , d, v = np.linalg.svd(S)
    d = np.sqrt(d)
    lamda = np.diag(d)
    A = np.dot(np.dot(u,lamda),np.transpose(u))

    # turn A into the 3x3 matrix H
    H_affine = np.zeros((3,3))
    H_affine[:2, :2] = A
    H_affine[2][2] = 1

    return H_affine


# The function finds the homography using one step method based on the given point set
def findHomograpyMatrix_Onestep(points, points2):

    l1 = np.cross(points[0], points[1])
```

```
    l2 = np.cross(points[1], points[2])
    l3 = np.cross(points[2], points[3])
    l4 = np.cross(points[3], points[0])
    l5 = np.cross(points2[0], points2[2])
    l6 = np.cross(points2[1], points2[3])

    # 5 orthogonal line pairs
    ls = [(l1,l2), (l2,l3), (l3,l4), (l4,l1), (l5,l6)]

    A_ = np.zeros((5,5))
    b = np.zeros(5)

    # create a linear system Ax = b to solve the degenerate conic
    for i in range(5):
        A_[i] = [ls[i][0][0]*ls[i][1][0], (ls[i][0][0]*ls[i][1][1]+ls[i][0][1]*ls[i
            ][1][0])/2,
                ls[i][0][1]*ls[i][1][1], (ls[i][0][0]*ls[i][1][2]+ls[i][0][2]*ls[i][1][0])
                    /2,
                (ls[i][0][1]*ls[i][1][2]+ls[i][0][2]*ls[i][1][1])/2]
        b[i] = -ls[i][0][2]*ls[i][1][2]

    C_ = np.linalg.solve(A_,b)
    C_ = C_/np.max(C_)

    # Solve A and V
    S = np.zeros((2,2))
    S[0][0] = C_[0]
    S[0][1] = C_[1]/2
    S[1][0] = C_[1]/2
    S[1][1] = C_[2]

    u , d_square, v = np.linalg.svd(S)
    d = np.sqrt(d_square)
    D = np.diag(d)
    A = np.dot(np.dot(u,D),np.transpose(u))

    b2 = [C_[3]/2, C_[4]/2]
    V = np.linalg.solve(A,b2)

    H = np.zeros((3,3))
    H[:2, :2] = A
    H[2, :2] = V
    H[2][2] = 1

    return H


# The function creates an blank image with the same size as the input image
def getBlankImage(width, height):

    blankimg = np.zeros((min(height, 50000), min(width, 50000), 3), dtype=np.uint8)

    return blankimg
```

```
# recover an image using a given homography
def imageRecovery(img, H):

    # use the origin image to calculate the size of the recovered image
    maxcoord_distort = np.array([[0, 0], [0, img.shape[0]], [img.shape[1], img.shape[0]],
         [img.shape[1], 0]])
    maxcoord_distort = np.append(maxcoord_distort, np.ones((4,1)), axis=1)

    maxcoord_world = np.linalg.inv(H).dot(maxcoord_distort.T)
    maxcoord_world = (maxcoord_world/maxcoord_world[-1]).astype(int)

    # offset in the recovered image coordinates
    offset_x = min(maxcoord_world[0])
    offset_y = min(maxcoord_world[1])

    # calculated size of the recovered image
    new_width = max(maxcoord_world[0]) - min(maxcoord_world[0])
    new_height = max(maxcoord_world[1]) - min(maxcoord_world[1])

    new_img = getBlankImage(new_width, new_height)

    # pixel replacement
    for i in range(new_img.shape[1]):
        for j in range(new_img.shape[0]):

            x = i + offset_x
            y = j + offset_y

            proj_coord = H.dot([x, y, 1])
            x_proj = round(proj_coord[0]/proj_coord[2])
            y_proj = round(proj_coord[1]/proj_coord[2])

            # replace the projected pixel
            if 0 <= x_proj and x_proj < img.shape[1] and 0 <= y_proj and y_proj < img.
                shape[0]:
                new_img[j, i] = img[y_proj, x_proj]

# return new_img
    return new_img


# The function recovery a distort image back into world coordinate
# using the point to point method
def pointToPointRecovery(img, coord_distort, coord_world):

    H = findHomograpyMatrix_P2P(coord_world, coord_distort)

    new_img = imageRecovery(img, H)

    return new_img


# The function recovery a distort image back into world coordinate
# using the two step method
```

```
def twoStepRecovery(img, coord_distort):


    coord_distort = np.append(coord_distort, np.ones((4,1)), axis=1)

    # remove projection distortion
    H_projective = findHomograpyMatrix_Projective(coord_distort)

    new_img_step1 = imageRecovery(img, H_projective)

    # remove affine distortion
    coord_distort_no_projective = np.linalg.inv(H_projective).dot(coord_distort.T)
    coord_distort_no_projective = coord_distort_no_projective/coord_distort_no_projective
        [2]

    H_affine = findHomograpyMatrix_Affine(coord_distort_no_projective)
# print(H_affine)

    new_img_step2 = imageRecovery(new_img_step1, H_affine)

    return new_img_step1, new_img_step2


# The function recovery a distort image back into world coordinate
# using the two step method
def oneStepRecovery(img, coord_distort, coord_distort2):

    coord_distort = np.append(coord_distort, np.ones((4,1)), axis=1)
    coord_distort2 = np.append(coord_distort2, np.ones((4,1)), axis=1)

    H = findHomograpyMatrix_Onestep(coord_distort, coord_distort2)

    new_img = imageRecovery(img, H)

    return new_img


if __name__ == '__main__' :

    # images for task 1
    img_a = cv.imread('hw3images/building.jpg')
    img_b = cv.imread('hw3images/nighthawks.jpg')
    img_a=cv.cvtColor(img_a,cv.COLOR_BGR2RGB)
    img_b=cv.cvtColor(img_b,cv.COLOR_BGR2RGB)

    corners_a = np.array([[240, 200], [236, 369], [295, 374], [297, 215]])
    corners_b = np.array([[75, 179], [78, 652], [805, 620], [803, 219]])
    corners_a_world = np.array([[0, 0], [0, 90], [30, 90], [30, 0]])
    corners_b_world = np.array([[0, 0], [0, 85], [150, 85], [150, 0]])


    ### Task 1.1 ###
    img_a_1_1 = pointToPointRecovery(img_a, corners_a , corners_a_world)
    img_b_1_1 = pointToPointRecovery(img_b, corners_b , corners_b_world)
```

```
plt.imshow(img_a_1_1)
plt.figure()
plt.imshow(img_b_1_1)
plt.figure()


### Task 1.2 ###
img_a_1_2_proj, img_a_1_2_aff = twoStepRecovery(img_a, corners_a)
img_b_1_2_proj, img_b_1_2_aff = twoStepRecovery(img_b, corners_b)

plt.imshow(img_a_1_2_proj)
plt.figure()
plt.imshow(img_a_1_2_aff)
plt.figure()
plt.imshow(img_b_1_2_proj)
plt.figure()
plt.imshow(img_b_1_2_aff)
plt.figure()


### Task 1.3 ###

# one more set of points for the 5th orthogonal line pair
corners_a2 = np.array([[240, 200], [238, 262], [296, 271], [297, 215]])
corners_b2 = np.array([[75, 179], [78, 652], [555, 632], [552, 204]])

img_a_1_3 = oneStepRecovery(img_a, corners_a , corners_a2)
img_b_1_3 = oneStepRecovery(img_b, corners_b , corners_b2)

plt.imshow(img_a_1_3)
plt.figure()
plt.imshow(img_b_1_3)
plt.figure()


# images for task 2
img_a = cv.imread('hw3images/monitor.jpg')
img_b = cv.imread('hw3images/sticker.jpg')
img_a=cv.cvtColor(img_a,cv.COLOR_BGR2RGB)
img_b=cv.cvtColor(img_b,cv.COLOR_BGR2RGB)

corners_a = np.array([[464, 284], [186, 895], [1470, 979], [1630, 180]])
corners_b = np.array([[56, 78], [61, 1505], [694, 1861], [848, 360]])
corners_a_world = np.array([[0, 0], [0, 1080], [1920, 1080], [1920, 0]])
corners_b_world = np.array([[0, 0], [0, 900], [400, 900], [400, 0]])


### Task 2.1 ###
img_a_1_1 = pointToPointRecovery(img_a, corners_a , corners_a_world)
img_b_1_1 = pointToPointRecovery(img_b, corners_b , corners_b_world)

plt.imshow(img_a_1_1)
plt.figure()
```

```
plt.imshow(img_b_1_1)
plt.figure()


### Task 2.2 ###
img_a_1_2_proj, img_a_1_2_aff = twoStepRecovery(img_a, corners_a)
img_b_1_2_proj, img_b_1_2_aff = twoStepRecovery(img_b, corners_b)

plt.imshow(img_a_1_2_proj)
plt.figure()
plt.imshow(img_a_1_2_aff)
plt.figure()
plt.imshow(img_b_1_2_proj)
plt.figure()
plt.imshow(img_b_1_2_aff)
plt.figure()


### Task 2.3 ###

# one more set of points for the 5th orthogonal line pair
corners_a2 = np.array([[890, 247], [667, 925], [1470, 979], [1630, 180]])
corners_b2 = np.array([[56, 78], [61, 808], [770, 1106], [848, 360]])

img_a_1_3 = oneStepRecovery(img_a, corners_a , corners_a2)
img_b_1_3 = oneStepRecovery(img_b, corners_b , corners_b2)

plt.imshow(img_a_1_3)
plt.figure()
plt.imshow(img_b_1_3)
plt.figure()
```