ECE661 Computer Vision HW3

Chengjun Guo
guo456@purdue.edu

19 September 2022

1 Logic

The code used are mostly the same for both tasks. The only difference are the variable names and values for points and input images.

The functions used for all methods include: The point_mapping_complementary function is used for three questions to generate a transformed image. In this function, it is calculating the x length and y length by max - min and 0 - min as offset for both x and y. In this case, all points in the transformed polygon would be included in the included in the new image. The bilinear_interpolation function is defined for handling the points with float number as coordinates.

In the point to point method, the point_to_point_calculate_h is the function for calculating Homography in point to point method. It takes in points in both original image and transformed image. This function is based on the function created in hw2.

In the two-step method, there are two steps. The first one is to remove the projective distortion. send_VL_to_infinite_h_calculate uses 8 points holding 2 pairs of parallel lines in undistorted image and calculate the Homography by sending the vanishing line to infinity. After removing the projective distortion, the corresponding selected points in intermediate image can be calculated by calculate_transformed_points. The second step for this method is to remove the affine distortion with undistort_affine_h_calculate. This function would return the homography that would remove affine distortion.

In the one-step method, the function one_step_calculate_H will generate the homography based on the 5 points that have two orthogonal lines intersecting for each. This function is fully vectorized by handling the homography calculation in numpy functions. The vectorized operations replaced the for loops in function.

1.1 Mathematics for Homography calculation

1.1.1 point to point method

First we know that the homography for corresponding points fulfills that:

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1}$$

After simplication of the equation, each pair of corresponding points can result in two equations:

$$xh_{11} + yh_{12} + h_{13} - xx'h_{31} - yx'h_{32} = x'h_{33} \tag{2}$$

$$xh_{21} + yh_{22} + h_{23} - xy'h_{31} - yy'h_{32} = y'h_{33} \tag{3}$$

where $x = \frac{x_1}{x_3}, y = \frac{x_2}{x_3}$. Same to x' and y'.
Since only ratio matters in homography, h_{33} can be set to 1 manually and there are 8 unknowns left, 8 equations are needed to solve for the homography which means 4 pairs of corresponding points are needed. Then we can calculate the homography by the following system.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x_3' & -y_3x_3' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y_3' & -y_3y_3' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x_4' & -y_4x_4' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y_4' & -y_4y_4' \end{bmatrix} \times \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \\ x_4' \\ y_4' \end{bmatrix} \tag{4}$$

With this function, we can find the rest 8 entries of H by calculating the inverse of the 8 by 8 matrix multiplying the 8 by 1 column vector.

1.1.2 two-step method

Step1: Remove the projective distortion.

For two parallel lines l and m in transformed image, the vanishing point can be calculated by cross product. The connection of two vanishing points is vanishing line. In the projective distorted image, the vanishing line is not in infinity. We are going to send it back to l_∞ which is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$.

Assume the vanishing line is $\begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix}$.The homography $H^{-}T$ for line will be $\begin{bmatrix} 1 & 0 & -l_1/l_3 \\ 0 & 1 & -l_2/l_3 \\ 0 & 0 & 1/l_3 \end{bmatrix}$.

Step2: Remove affine distortion.

For the expression for $cos\theta$ in form of Dual Degenerate Conic, the lines forming the Degenerate Conic is l and m:

$$cos\theta = \frac{l^T C_\infty^* m}{\sqrt{(l^T C_\infty^* l)(m^T C_\infty^* m)}} \tag{5}$$

To make $cos\theta = 0$, we need the nominator of it to be 0. By substituting l and m by l' and m'. The equation would be ${l'}^T H C_\infty^* H^T m' = 0$. Assume $S = AA^T = \begin{bmatrix} s_{11} & s_{12} \\ s_{12} & s_{22} \end{bmatrix}$. Let $s_{22} = 1$. Substitute l' with $\begin{pmatrix} l'_1 \\ l'_2 \\ l'_3 \end{pmatrix}$. Same to m'

The simplified equation would be:

$$s_{11} l'_1 m'_1 + s_{12}(l'_1 m'_2 + l'_2 m'_1) = -l'_2 m'_2 \tag{6}$$

Since s_{22} are already set to be 1, two more unknowns are left. In this case, two equations are needed which means two more line are needed. Assume those lines to be p and q. The equation would be:

$$\begin{bmatrix} l'_1 m'_1 & l'_1 m'_2 + l'_2 m'_1 \\ p'_1 q'_1 & p'_1 q'_2 + p'_2 q'_1 \end{bmatrix} \begin{bmatrix} s_{11} \\ s_{12} \end{bmatrix} = \begin{bmatrix} -l'_2 m'_2 \\ -p'_2 q'_2 \end{bmatrix} \tag{7}$$

Apply SVD to A and we have:$A = VDV^T$. Substitute A with it to S, we will have:
$S = AA^T = VDV^T VDV^T = VD^2V^T = V \begin{bmatrix} \lambda_1^2 & 0 \\ 0 & \lambda_2^2 \end{bmatrix} V^T$

Then A can be calculated by applying SVD to S and A would be $V \begin{bmatrix} \sqrt{\lambda_1^2} & 0 \\ 0 & \sqrt{\lambda_2^2} \end{bmatrix} V^T$.

Thus, H would be constructed by $\begin{bmatrix} A & 0 \\ 0^T & 1 \end{bmatrix}$.

1.1.3 one-step method

The transformed dual degenerate conic can be denoted as: $C^{*'} = HC_\infty^* H^T$. Construct the $C^{*'}$ with $\begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{bmatrix}$.

$HC^*_\infty H^T$ can be expanded as $\begin{bmatrix} AA^T & Av \\ v^T A^T & v^T v \end{bmatrix}$. Then the relationship becomes:

$$AA^T = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix} \tag{8}$$

$$Av = \begin{bmatrix} d/2 \\ e/2 \end{bmatrix} \tag{9}$$

Assume two lines l and m orthogonal, then they will have the property mentioned in the previous section that the numerator is 0. Then we will have:

$$(l'_1\, l'_2\, 1) \begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & 1 \end{bmatrix} \begin{pmatrix} m'_1 \\ m'_2 \\ 1 \end{pmatrix} = 0 \tag{10}$$

Expand this equation, it can be simplified as:

$$\begin{bmatrix} l'_1 m'_1 & l'_1 m'_2 + l'_2 m'_1 & l'_2 m'_2 & l'_1 + m'_1 & l'_2 + m'_2 \end{bmatrix} \begin{pmatrix} a \\ b/2 \\ c \\ d/2 \\ e/2 \end{pmatrix} = -1 \tag{11}$$

With these five unknown variables, five pairs of lines would be needed to solve for $C^{*'}$. After it is solved, the relationship between A,v and $C^{*'}$ would construct the homography.

2 Task1 data and images: building and nighthawks

Input images:

(a) building (b) nighthawks

Figure 1: input for task1

2.1 point to point method

The point selection:

	building	nighthawks
P	(237, 196)	(75,179)
Q	(232, 370)	(78,654)
R	(295, 375)	(805,621)
S	(301, 213)	(804,219)

Table 1: Points

Points in the images:

(a) building (b) nighthawks

Figure 2: point selection of point to point method for task1

The output for point to point method:

(a) building (b) nighthawks

Figure 3: output of point to point for task1

2.2 two-step method

The line PQ,QR,RS,PS:

(a) building

(b) nighthawks

Figure 4: line selection of two step for task1

The intermediate output after projective remove:

(a) building

(b) nighthawks

Figure 5: intermediate output of two step for task1

The output after affine remove:

(a) building

(b) nighthawks

Figure 6: output of two step for task1

2.3 one-step method

The points PQRS and new QPS selection for degenerate conic at P,Q,R,S,new Q:

(a) building

(b) nighthawks

Figure 7: Degenerate conic of one step for task1

The output after one step:

(a) building

(b) nighthawks

Figure 8: output of one step for task1

3 Task2 data and images:building2 and painting

Input images:

(a) card1

(b) painting

Figure 9: input for task2

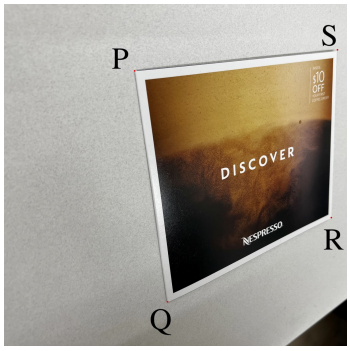
3.1 point to point method

The point selection:

	card1	painting
P	(487, 250)	(1090, 303)
Q	(610, 1114)	(1091, 662)
R	(1221, 799)	(1530, 700)
S	(1244, 171)	(1531, 275)

Table 2: Points

Points in the images:



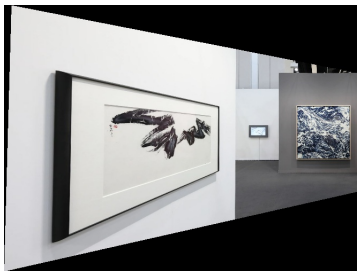
(a) card1

(b) painting

Figure 10: point selection of point to point method for task2

The output for point to point method:

(a) card1

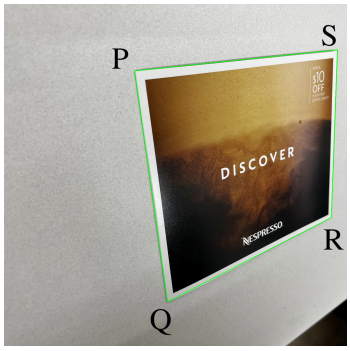


(b) painting

Figure 11: output of point to point for task2

3.2 two-step method

The line PQ,QR,RS,PS:



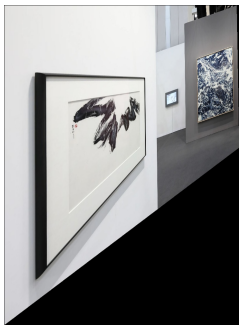
(a) card1

(b) painting

Figure 12: line selection of two step for task2

The intermediate output after projective remove:

(a) card1



(b) painting

Figure 13: intermediate output of two step for task2

The output after affine remove:

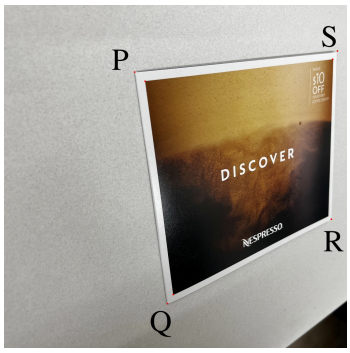
(a) card1

(b) painting

Figure 14: output of two step for task2

3.3 one-step method

The points PQRS and new QPS selection for degenerate conic at P,Q,R,S,new Q:



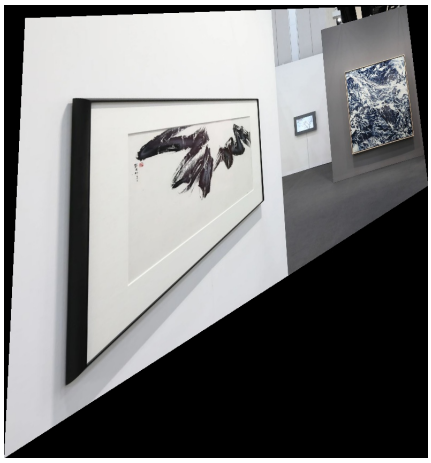
(a) card1

(b) painting

Figure 15: Degenerate conic of one step for task2

The output after one step:

(a) card1



(b) painting

Figure 16: output of one step for task2

4 Discussion and observations

4.1 Discussion about vectorization

In this homework, I fully vectorized the H calculation for one step. I modified the points with numpy functions that help doing more entry operation in matrices. For the other functions, they are partially vectorized.

4.2 Discussion about the image choosing

For task2, the indices for downloaded images will reach limit during the calculation. I used the indices for the images of previous homework and it works fine. Consider an extreme situation, moving viewpoint from 90 degree to approximately 180 degree on the right. It is sending the point on the left to infinity and transferring the ideal point into the scope. Since our algorithm is mapping the new image point back to the original images, some original points indices will become extremely large.

4.3 Observation

Among all three methods outputs, take building as example, point to point method would not have the same point choosing problem since the points mapping is straight forward. For the other two methods, if I choose pairs of lines that is parallel to the other pairs, image could have some extremely bad shape. For example, in first step of two step method, if I choose two diagonal lines of a square and two edges of square, it will return a better result than choosing all four edges. Also, the point to point method doesn't have to rotate. The subgroup of affine is similarity. After removing projective and affine, the images still needs to remove the subgroup of affine transformation.
Other than the image quality, the point to point method has best execution time and memory efficiency from my observation.

5 Code for task1

```
import numpy as np
import cv2
import sys
import math


#   imported function from hw2 for corresponding points homography calculation
def point_to_point_calculate_h(X, X_prime):
    A = np.zeros((8, 8))
    B = np.zeros((8, 1))
    for i in range(4):
        A[2 * i, 0] = X[i][0]
        A[2 * i, 1] = X[i][1]
        A[2 * i, 2] = 1
        A[2 * i, 6] = -X[i][0] * X_prime[i][0]
```

```
        A[2 * i, 7] = -X[i][1] * X_prime[i][0]

        A[2 * i + 1, 3] = X[i][0]
        A[2 * i + 1, 4] = X[i][1]
        A[2 * i + 1, 5] = 1
        A[2 * i + 1, 6] = -X[i][0] * X_prime[i][1]
        A[2 * i + 1, 7] = -X[i][1] * X_prime[i][1]
        B[2 * i] = X_prime[i][0]
        B[2 * i + 1] = X_prime[i][1]

    H_vec = np.ndarray.flatten(np.dot(np.linalg.inv(A), B))
    print(H_vec)
    H = np.ones((3, 3))
    H[0, :] = H_vec[0:3]
    H[1, :] = H_vec[3:6]
    H[2, :2] = H_vec[6:8]
    return H


def bilinear_interpolation(y, x, image):
    # based on wikipedia formula
    X_Q11 = image[int(math.floor(y)), int(math.floor(x))]
    X_Q12 = image[int(math.floor(y)), int(math.ceil(x))]
    X_Q22 = image[int(math.ceil(y)), int(math.ceil(x))]
    X_Q21 = image[int(math.ceil(y)), int(math.floor(x))]
    X_x_y1 = (math.ceil(y) - y) * X_Q11 + (y - math.floor(y)) * X_Q21
    X_x_y2 = (math.ceil(y) - y) * X_Q12 + (y - math.floor(y)) * X_Q22

    result = (math.ceil(x) - x) * X_x_y1 + (x - math.floor(x)) * X_x_y2
    return result


def point_mapping(Image_X, Image_X_prime, H):
    # same function from previous hw, won't include all points transformed.
    H_inv = np.linalg.inv(H)
    for i in range(len(Image_X_prime)):
        for j in range(len(Image_X_prime[i])):
            point_in_x_prime = np.array([j, i, 1])
            point_in_x = np.dot(H_inv, point_in_x_prime)
            x = point_in_x[0] / point_in_x[2]
            y = point_in_x[1] / point_in_x[2]
            if y < 0:
                y = 0
            if y > len(Image_X) - 1:
                y = len(Image_X) - 1
            if x < 0:
```

```
                x = 0
            if x > len(Image_X[0]) - 1:
                x = len(Image_X[0]) - 1
            Image_X_prime[i, j] = bilinear_interpolation(y, x, Image_X)

    # cv2.imshow("image_X_prime filled", Image_X_prime)
    return Image_X_prime


def point_mapping_complementary(Image_X, Image_X_prime, H):
    # finishing the full picture, includes blank area
    H_inv = np.linalg.inv(H)
    tl_prime = np.dot(H, np.array([0, 0, 1]))
    tr_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, 0, 1]))
    bl_prime = np.dot(H, np.array([0, len(Image_X_prime) - 1, 1]))
    br_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, len(Image_X_prime) - 1, 1]))
    tl_point = [int(tl_prime[0] / tl_prime[2]), int(tl_prime[1] / tl_prime[2])]
    tr_point = [int(tr_prime[0] / tr_prime[2]), int(tr_prime[1] / tr_prime[2])]
    bl_point = [int(bl_prime[0] / bl_prime[2]), int(bl_prime[1] / bl_prime[2])]
    br_point = [int(br_prime[0] / br_prime[2]), int(br_prime[1] / br_prime[2])]
    x_max = max(tr_point[0], br_point[0])
    x_min = min(tl_point[0], bl_point[0])
    y_max = max(bl_point[1], br_point[1])
    y_min = min(tl_point[1], tr_point[1])
    print(x_max, x_min, y_max, y_min)
    x_offset = int(0 - x_min)
    y_offset = int(0 - y_min)
    shape = (int(y_max - y_min), int(x_max - x_min), Image_X_prime.shape[2])
    area = np.zeros(shape)
    points = [tl_point, bl_point, br_point, tr_point]
    for pt in points:
        pt[0] += x_offset
        pt[1] += y_offset
    image_to_fill = cv2.fillPoly(area, pts=[np.array(points, dtype=np.int32)], color=(255, 2
    print(tl_point, bl_point, br_point, tr_point)
    # cv2.imshow("image_X_prime filled", image_to_fill)
    for i in range(len(image_to_fill)):
        for j in range(len(image_to_fill[i])):
            if np.all(image_to_fill[i, j] == 255):
                point_in_x_prime = np.array([j - x_offset, i - y_offset, 1])
                point_in_x = np.dot(H_inv, point_in_x_prime)
                x = point_in_x[0] / point_in_x[2]
                y = point_in_x[1] / point_in_x[2]
                if y < 0:
                    y = 0
                if y > len(Image_X) - 1:
```

```
                    y = len(Image_X) - 1
                if x < 0:
                    x = 0
                if x > len(Image_X[0]) - 1:
                    x = len(Image_X[0]) - 1
                image_to_fill[i, j] = bilinear_interpolation(y, x, Image_X)

    return image_to_fill


def send_VL_to_infinite_h_calculate(points):
    # points hold 8 points for 4 lines, 0,1 for a line 2,3 for a line, same to the rest
    p = np.zeros((8, 3))
    for i in range(8):
        p[i] = np.array([points[i][0], points[i][1], 1])
    line1 = np.cross(p[0], p[1])
    line2 = np.cross(p[2], p[3])
    line3 = np.cross(p[4], p[5])
    line4 = np.cross(p[6], p[7])
    VP1 = np.cross(line1, line2)
    VP2 = np.cross(line3, line4)
    VL = np.cross(VP1, VP2)
    H = np.zeros((3, 3))
    H[0, 0] = 1
    H[1, 1] = 1
    H[2, :] = VL / VL[2]
    return H


def calculate_transformed_points(points, Image_X, Image_X_prime, H):
    #calculate the point from original image after step1
    tl_prime = np.dot(H, np.array([0, 0, 1]))
    tr_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, 0, 1]))
    bl_prime = np.dot(H, np.array([0, len(Image_X_prime) - 1, 1]))
    br_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, len(Image_X_prime) - 1, 1]))
    tl_point = [int(tl_prime[0] / tl_prime[2]), int(tl_prime[1] / tl_prime[2])]
    tr_point = [int(tr_prime[0] / tr_prime[2]), int(tr_prime[1] / tr_prime[2])]
    bl_point = [int(bl_prime[0] / bl_prime[2]), int(bl_prime[1] / bl_prime[2])]
    br_point = [int(br_prime[0] / br_prime[2]), int(br_prime[1] / br_prime[2])]
    x_max = max(tr_point[0], br_point[0])
    x_min = min(tl_point[0], bl_point[0])
    y_max = max(bl_point[1], br_point[1])
    y_min = min(tl_point[1], tr_point[1])
    print(x_max, x_min, y_max, y_min)
    x_offset = int(0 - x_min)
    y_offset = int(0 - y_min)
```

```
    result_points = []
    for point in points:
        point_in_x = np.array([point[0], point[1], 1])
        point_in_x_prime = np.dot(H, point_in_x)
        x = int(point_in_x_prime[0] / point_in_x_prime[2]) + x_offset
        y = int(point_in_x_prime[1] / point_in_x_prime[2]) + y_offset
        result_points.append([x, y])
    return np.array(result_points)


def undistort_affine_h_calculate(points):
    p = np.zeros((4, 3))
    for i in range(4):
        p[i] = np.array([points[i][0], points[i][1], 1])
    line1 = np.cross(p[0], p[1])
    line2 = np.cross(p[1], p[2])
    line3 = np.cross(p[0], p[2])
    line4 = np.cross(p[1], p[3])
    line1 = line1 / line1[2]
    line2 = line2 / line2[2]
    line3 = line3 / line3[2]
    line4 = line4 / line4[2]
    A = np.zeros((2, 2))
    A[0, 0] = line1[0] * line2[0]
    A[0, 1] = line1[0] * line2[1] + line1[1] * line2[0]
    A[1, 0] = line3[0] * line4[0]
    A[1, 1] = line3[0] * line4[1] + line3[1] * line4[0]
    B = np.zeros((2, 1))
    B[0] = - line1[1] * line2[1]
    B[1] = - line3[1] * line4[1]
    S_vec = np.ndarray.flatten(np.dot(np.linalg.pinv(A), B))
    S = np.ones((2, 2))
    S[0, 0] = S_vec[0]
    S[0, 1] = S_vec[1]
    S[1, 0] = S_vec[1]
    # S constructed
    V, D_square, V_T = np.linalg.svd(S)
    D = np.sqrt(np.diag(D_square))  # dimension wrong for D, do diag manually
    A_matrix = np.dot(np.dot(V, D), V.transpose())  # do transpose for V, SVD is not returni
    H = np.zeros((3, 3))
    H[0, 0:2] = A_matrix[0]
    H[1, 0:2] = A_matrix[1]
    H[2, 2] = 1
    H = np.linalg.inv(H)  # H is affine transformation, inverse to find affine removing H
    return H
```

```
def one_step_calculate_H(points):
    # points PQRS, new_PQR 7 points total, vectorized operation in this step
    points = np.asarray(points)
    points = np.insert(points,2,1,axis=1)
    PQ = np.cross(points[0], points[1])
    QR = np.cross(points[1], points[2])
    RS = np.cross(points[2], points[3])
    PS = np.cross(points[3], points[0])
    PQ_new = np.cross(points[4], points[5])
    QR_new = np.cross(points[5], points[6])
    l_lines = np.asarray([PQ,QR,RS,PS,PQ_new])
    m_lines = np.asarray([QR,RS,PS,PQ,QR_new])
    l_lines = np.divide(l_lines.T,l_lines[:,2]).T  # calculate the l lines with z = 1
    m_lines = np.divide(m_lines.T,m_lines[:,2]).T  # calculate the m lines with z = 1
    A = np.zeros((5,5))
    A[:,0] = np.multiply(l_lines[:,0],m_lines[:,0])
    A[:,1] = np.multiply(m_lines[:,1],l_lines[:,0]) + np.multiply(m_lines[:,0],l_lines[:,1])
    A[:,2] = np.multiply(l_lines[:,1], m_lines[:,1])
    A[:,3] = l_lines[:,0] + m_lines[:,0]
    A[:,4] = l_lines[:,1] + m_lines[:,1]
    DC_vec = np.ndarray.flatten(np.dot(np.linalg.pinv(A),-np.ones((5,1))))
    DC_vec = DC_vec/np.max(DC_vec)
    DC = np.ones((3, 3))
    DC[0,0:2] = DC_vec[0:2]
    DC[0,2] = DC_vec[3]
    DC[1,0:2] = DC_vec[1:3]
    DC[1,2] = DC_vec[4]
    DC[2,0:2] = DC_vec[3:5]
    H = np.zeros((3, 3))
    V, D_square, V_T = np.linalg.svd(DC[0:2,0:2])
    D = np.sqrt(np.diag(D_square))
    A_matrix = np.dot(np.dot(V, D), V.transpose())
    H[0:2,0:2] = A_matrix
    v = np.dot(np.linalg.pinv(A_matrix),DC[0:2,2])
    H[2,0:2] = v.transpose()
    H[2,2] = 1
    H = np.linalg.inv(H)
    return H


if __name__ == "__main__":
    # Task1
    building = cv2.imread('building.jpg')
    nighthawks = cv2.imread('nighthawks.jpg')
    # point to point
```

```
#   building [237,196],[232,370],[295,375],[301,213]
#   undistorted building [237,196],[237,375],[295,375],[295,196]
#
building_PQRS = np.array([[237,196],[232,370],[295,375],[301,213]])
building_undistorted_PQRS = np.array([[237,196],[237,370],[295,370],[295,196]])
H_building = point_to_point_calculate_h(building_PQRS,building_undistorted_PQRS)
undistorted_building = point_mapping_complementary(building,building.copy(),H_building)
# cv2.imshow("undistorted building",undistorted_building)
# cv2.imwrite("undistorted_building_p2p.jpg",undistorted_building)
image = building.copy()
for i in [(237, 196), (232, 370), (295, 375), (301, 213)]:
    image = cv2.circle(image,i, radius=0, color=(0, 0, 255), thickness=6)
cv2.imwrite("building_point_to_point_PQRS.jpg",image)

  # nighthawks [75,179],[78,654],[805,621],[804,219]
  # undistorted nighthawks [75,179],[75,593],[805,593],[805,179]

nighthawks_PQRS = np.array([[75,179],[78,654],[805,621],[804,219]])
nighthawks_undistorted_PQRS = np.array([[75,179],[75,593],[805,593],[805,179]])
H_nighthawks = point_to_point_calculate_h(nighthawks_PQRS,nighthawks_undistorted_PQRS)
undistorted_nighthawks = point_mapping_complementary(nighthawks,nighthawks.copy(),H_nigh
# cv2.imwrite("undistorted_nighthawks_p2p.jpg",undistorted_nighthawks)
image = nighthawks.copy()
for i in [(75,179),(78,654),(805,621),(804,219)]:
    image = cv2.circle(image,i, radius=0, color=(0, 0, 255), thickness=6)
cv2.imwrite("nighthawks_point_to_point_PQRS.jpg",image)

# two-step method
building_2_pair_parallel_line_points = np.array(
    [[237, 196], [232, 370], [295, 375], [301, 213], [237, 196], [301, 213], [232, 370],

H_two_step_1_building = send_VL_to_infinite_h_calculate(building_2_pair_parallel_line_po
building_two_step_1 = point_mapping_complementary(building, building.copy(), H_two_step_
# cv2.imwrite("building_two_step_mid.jpg", building_two_step_1)
# transformed_building_points = calculate_transformed_points([[237, 196], [232, 254], [2
# print("transformed building points: ",transformed_building_points)
# H_two_step_2_building = undistort_affine_h_calculate(transformed_building_points)
# building_by_two_step = point_mapping_complementary(building,building.copy(),np.matmul(
# cv2.imwrite("building_two_step.jpg", building_by_two_step)
image = building.copy()
for i in range(4):
    image = cv2.line(image, tuple(building_2_pair_parallel_line_points[2*i]), tuple(buil
cv2.imwrite("building_two_step_parallel_lines.jpg", image)

nighthawks_2_pair_parallel_line_points = np.array(
    [[75, 179], [78, 654], [805, 621], [804, 219], [75, 179], [804, 219], [78, 654], [80
```

```
H_two_step_1_nighthawks = send_VL_to_infinite_h_calculate(nighthawks_2_pair_parallel_lin
nighthawks_two_step_1 = point_mapping_complementary(nighthawks, nighthawks.copy(), H_two
# cv2.imwrite("nighthawks_two_step_mid.jpg", nighthawks_two_step_1)
# transformed_nighthawks_points = calculate_transformed_points([[75, 179], [78, 654], [4
# print("transformed nighthawks points: ", transformed_nighthawks_points)
# H_two_step_2_nighthawks = undistort_affine_h_calculate(transformed_nighthawks_points)
# nighthawks_by_two_step = point_mapping_complementary(nighthawks,nighthawks.copy(),np.m
# cv2.imwrite("nighthawks_two_step.jpg", nighthawks_by_two_step)
image = nighthawks.copy()
for i in range(4):
    image = cv2.line(image, tuple(nighthawks_2_pair_parallel_line_points[2*i]), tuple(ni
cv2.imwrite("nighthawks_two_step_parallel_lines.jpg", image)


# one-step method
building_one_step_points = [[237, 196], [232, 370], [295, 375], [301, 213],[232,378],[24
# H_one_step_building = one_step_calculate_H(building_one_step_points)
# building_one_step = point_mapping_complementary(building,building.copy(),H_one_step_bu
# cv2.imwrite("building_one_step.jpg",building_one_step)
image = building.copy()
for i in [(237, 196), (232, 370), (295, 375), (301, 213),(232,378),(240,124),(716,291)]:
    image = cv2.circle(image, i, radius=0, color=(0, 0, 255), thickness=6)
cv2.imwrite("building_one_step_PQRSQPS.jpg",image)

nighthawks_one_step_points = [[75,179],[78,654],[805,621],[804,219],[13,730],[11,99],[86
# H_one_step_nighthawks = one_step_calculate_H(nighthawks_one_step_points)
# nighthawks_one_step = point_mapping_complementary(nighthawks,nighthawks.copy(),H_one_s
# cv2.imwrite("nighthawks_one_step.jpg",nighthawks_one_step)
image = nighthawks.copy()
for i in [[75,179],[78,654],[805,621],[804,219],[13,730],[11,99],[863,160]]:
    image = cv2.circle(image, tuple(i), radius=0, color=(0, 0, 255), thickness=6)
cv2.imwrite("nighthawks_one_step_PQRSQPS.jpg",image)

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

6 Code for task2

```
import numpy as np
import cv2
import sys
import math


#   imported function from hw2 for corresponding points homography calculation
def point_to_point_calculate_h(X, X_prime):
    A = np.zeros((8, 8))
    B = np.zeros((8, 1))
    for i in range(4):
        A[2 * i, 0] = X[i][0]
        A[2 * i, 1] = X[i][1]
        A[2 * i, 2] = 1
        A[2 * i, 6] = -X[i][0] * X_prime[i][0]
        A[2 * i, 7] = -X[i][1] * X_prime[i][0]

        A[2 * i + 1, 3] = X[i][0]
        A[2 * i + 1, 4] = X[i][1]
        A[2 * i + 1, 5] = 1
        A[2 * i + 1, 6] = -X[i][0] * X_prime[i][1]
        A[2 * i + 1, 7] = -X[i][1] * X_prime[i][1]
        B[2 * i] = X_prime[i][0]
        B[2 * i + 1] = X_prime[i][1]

    H_vec = np.ndarray.flatten(np.dot(np.linalg.inv(A), B))
    print(H_vec)
    H = np.ones((3, 3))
    H[0, :] = H_vec[0:3]
    H[1, :] = H_vec[3:6]
    H[2, :2] = H_vec[6:8]
    return H


def bilinear_interpolation(y, x, image):
    # based on wikipedia formula
    X_Q11 = image[int(math.floor(y)), int(math.floor(x))]
    X_Q12 = image[int(math.floor(y)), int(math.ceil(x))]
    X_Q22 = image[int(math.ceil(y)), int(math.ceil(x))]
    X_Q21 = image[int(math.ceil(y)), int(math.floor(x))]
    X_x_y1 = (math.ceil(y) - y) * X_Q11 + (y - math.floor(y)) * X_Q21
    X_x_y2 = (math.ceil(y) - y) * X_Q12 + (y - math.floor(y)) * X_Q22

    result = (math.ceil(x) - x) * X_x_y1 + (x - math.floor(x)) * X_x_y2
    return result
```

```
def point_mapping(Image_X, Image_X_prime, H):
    # same function from previous hw, won't include all points transformed.
    H_inv = np.linalg.inv(H)
    for i in range(len(Image_X_prime)):
        for j in range(len(Image_X_prime[i])):
            point_in_x_prime = np.array([j, i, 1])
            point_in_x = np.dot(H_inv, point_in_x_prime)
            x = point_in_x[0] / point_in_x[2]
            y = point_in_x[1] / point_in_x[2]
            if y < 0:
                y = 0
            if y > len(Image_X) - 1:
                y = len(Image_X) - 1
            if x < 0:
                x = 0
            if x > len(Image_X[0]) - 1:
                x = len(Image_X[0]) - 1
            Image_X_prime[i, j] = bilinear_interpolation(y, x, Image_X)

    # cv2.imshow("image_X_prime filled", Image_X_prime)
    return Image_X_prime


def point_mapping_complementary(Image_X, Image_X_prime, H):
    # finishing the full picture, includes blank area
    H_inv = np.linalg.inv(H)
    tl_prime = np.dot(H, np.array([0, 0, 1]))
    tr_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, 0, 1]))
    bl_prime = np.dot(H, np.array([0, len(Image_X_prime) - 1, 1]))
    br_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, len(Image_X_prime) - 1, 1]))
    tl_point = [int(tl_prime[0] / tl_prime[2]), int(tl_prime[1] / tl_prime[2])]
    tr_point = [int(tr_prime[0] / tr_prime[2]), int(tr_prime[1] / tr_prime[2])]
    bl_point = [int(bl_prime[0] / bl_prime[2]), int(bl_prime[1] / bl_prime[2])]
    br_point = [int(br_prime[0] / br_prime[2]), int(br_prime[1] / br_prime[2])]
    x_max = max(tr_point[0], br_point[0])
    x_min = min(tl_point[0], bl_point[0])
    y_max = max(bl_point[1], br_point[1])
    y_min = min(tl_point[1], tr_point[1])
    print(x_max, x_min, y_max, y_min)
    x_offset = int(0 - x_min)
    y_offset = int(0 - y_min)
    shape = (int(y_max - y_min), int(x_max - x_min), Image_X_prime.shape[2])
    area = np.zeros(shape)
    points = [tl_point, bl_point, br_point, tr_point]
    for pt in points:
```

```
            pt[0] += x_offset
            pt[1] += y_offset
        image_to_fill = cv2.fillPoly(area, pts=[np.array(points, dtype=np.int32)], color=(255, 2
        print(tl_point, bl_point, br_point, tr_point)
        # cv2.imshow("image_X_prime filled", image_to_fill)
        for i in range(len(image_to_fill)):
            for j in range(len(image_to_fill[i])):
                if np.all(image_to_fill[i, j] == 255):
                    point_in_x_prime = np.array([j - x_offset, i - y_offset, 1])
                    point_in_x = np.dot(H_inv, point_in_x_prime)
                    x = point_in_x[0] / point_in_x[2]
                    y = point_in_x[1] / point_in_x[2]
                    if y < 0:
                        y = 0
                    if y > len(Image_X) - 1:
                        y = len(Image_X) - 1
                    if x < 0:
                        x = 0
                    if x > len(Image_X[0]) - 1:
                        x = len(Image_X[0]) - 1
                    image_to_fill[i, j] = bilinear_interpolation(y, x, Image_X)

        return image_to_fill


    def send_VL_to_infinite_h_calculate(points):
        # points hold 8 points for 4 lines, 0,1 for a line 2,3 for a line, same to the rest
        p = np.zeros((8, 3))
        for i in range(8):
            p[i] = np.array([points[i][0], points[i][1], 1])
        line1 = np.cross(p[0], p[1])
        line2 = np.cross(p[2], p[3])
        line3 = np.cross(p[4], p[5])
        line4 = np.cross(p[6], p[7])
        VP1 = np.cross(line1, line2)
        VP2 = np.cross(line3, line4)
        VL = np.cross(VP1, VP2)
        H = np.zeros((3, 3))
        H[0, 0] = 1
        H[1, 1] = 1
        H[2, :] = VL / VL[2]
        return H


    def calculate_transformed_points(points, Image_X, Image_X_prime, H):
        #calculate the point from original image after step1
```

```
    tl_prime = np.dot(H, np.array([0, 0, 1]))
    tr_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, 0, 1]))
    bl_prime = np.dot(H, np.array([0, len(Image_X_prime) - 1, 1]))
    br_prime = np.dot(H, np.array([len(Image_X_prime[0]) - 1, len(Image_X_prime) - 1, 1]))
    tl_point = [int(tl_prime[0] / tl_prime[2]), int(tl_prime[1] / tl_prime[2])]
    tr_point = [int(tr_prime[0] / tr_prime[2]), int(tr_prime[1] / tr_prime[2])]
    bl_point = [int(bl_prime[0] / bl_prime[2]), int(bl_prime[1] / bl_prime[2])]
    br_point = [int(br_prime[0] / br_prime[2]), int(br_prime[1] / br_prime[2])]
    x_max = max(tr_point[0], br_point[0])
    x_min = min(tl_point[0], bl_point[0])
    y_max = max(bl_point[1], br_point[1])
    y_min = min(tl_point[1], tr_point[1])
    print(x_max, x_min, y_max, y_min)
    x_offset = int(0 - x_min)
    y_offset = int(0 - y_min)
    result_points = []
    for point in points:
        point_in_x = np.array([point[0], point[1], 1])
        point_in_x_prime = np.dot(H, point_in_x)
        x = int(point_in_x_prime[0] / point_in_x_prime[2]) + x_offset
        y = int(point_in_x_prime[1] / point_in_x_prime[2]) + y_offset
        result_points.append([x, y])
    return np.array(result_points)


def undistort_affine_h_calculate(points):
    p = np.zeros((4, 3))
    for i in range(4):
        p[i] = np.array([points[i][0], points[i][1], 1])
    line1 = np.cross(p[0], p[1])
    line2 = np.cross(p[1], p[2])
    line3 = np.cross(p[0], p[2])
    line4 = np.cross(p[1], p[3])
    line1 = line1 / line1[2]
    line2 = line2 / line2[2]
    line3 = line3 / line3[2]
    line4 = line4 / line4[2]
    A = np.zeros((2, 2))
    A[0, 0] = line1[0] * line2[0]
    A[0, 1] = line1[0] * line2[1] + line1[1] * line2[0]
    A[1, 0] = line3[0] * line4[0]
    A[1, 1] = line3[0] * line4[1] + line3[1] * line4[0]
    B = np.zeros((2, 1))
    B[0] = - line1[1] * line2[1]
    B[1] = - line3[1] * line4[1]
    S_vec = np.ndarray.flatten(np.dot(np.linalg.pinv(A), B))
```

```
    S = np.ones((2, 2))
    S[0, 0] = S_vec[0]
    S[0, 1] = S_vec[1]
    S[1, 0] = S_vec[1]
    # S constructed
    V, D_square, V_T = np.linalg.svd(S)
    D = np.sqrt(np.diag(D_square))  # dimension wrong for D, do diag manually
    A_matrix = np.dot(np.dot(V, D), V.transpose())  # do transpose for V, SVD is not returni
    H = np.zeros((3, 3))
    H[0, 0:2] = A_matrix[0]
    H[1, 0:2] = A_matrix[1]
    H[2, 2] = 1
    H = np.linalg.inv(H)  # H is affine transformation, inverse to find affine removing H
    return H


def one_step_calculate_H(points):
    # points PQRS, new_PQR 7 points total, vectorized operation in this step
    points = np.asarray(points)
    points = np.insert(points,2,1,axis=1)
    PQ = np.cross(points[0], points[1])
    QR = np.cross(points[1], points[2])
    RS = np.cross(points[2], points[3])
    PS = np.cross(points[3], points[0])
    PQ_new = np.cross(points[4], points[5])
    QR_new = np.cross(points[5], points[6])
    l_lines = np.asarray([PQ,QR,RS,PS,PQ_new])
    m_lines = np.asarray([QR,RS,PS,PQ,QR_new])
    l_lines = np.divide(l_lines.T,l_lines[:,2]).T  # calculate the l lines with z = 1
    m_lines = np.divide(m_lines.T,m_lines[:,2]).T  # calculate the m lines with z = 1
    A = np.zeros((5,5))
    A[:,0] = np.multiply(l_lines[:,0],m_lines[:,0])
    A[:,1] = np.multiply(m_lines[:,1],l_lines[:,0]) + np.multiply(m_lines[:,0],l_lines[:,1])
    A[:,2] = np.multiply(l_lines[:,1], m_lines[:,1])
    A[:,3] = l_lines[:,0] + m_lines[:,0]
    A[:,4] = l_lines[:,1] + m_lines[:,1]
    DC_vec = np.ndarray.flatten(np.dot(np.linalg.pinv(A),-np.ones((5,1))))
    DC_vec = DC_vec/np.max(DC_vec)
    DC = np.ones((3, 3))
    DC[0,0:2] = DC_vec[0:2]
    DC[0,2] = DC_vec[3]
    DC[1,0:2] = DC_vec[1:3]
    DC[1,2] = DC_vec[4]
    DC[2,0:2] = DC_vec[3:5]
    H = np.zeros((3, 3))
    V, D_square, V_T = np.linalg.svd(DC[0:2,0:2])
```

```
    D = np.sqrt(np.diag(D_square))
    A_matrix = np.dot(np.dot(V, D), V.transpose())
    H[0:2,0:2] = A_matrix
    v = np.dot(np.linalg.pinv(A_matrix),DC[0:2,2])
    H[2,0:2] = v.transpose()
    H[2,2] = 1
    H = np.linalg.inv(H)
    return H


if __name__ == "__main__":
    # Task1
    card1 = cv2.imread('card1.jpeg')
    painting = cv2.imread('painting.jpg')
    # point to point
    #   card1 [237,196],[232,370],[295,375],[301,213]
    #   undistorted card1 [237,196],[237,375],[295,375],[295,196]
    #
    card1_PQRS = np.array([[487, 250], [610, 1114], [1221, 799], [1244, 171]],np.float32)
    card1_undistorted_PQRS = np.array([[0,0],[0,100],[100,100],[100,0]],np.float32)
    H_card1 = point_to_point_calculate_h(card1_PQRS,card1_undistorted_PQRS)
    undistorted_card1 = point_mapping_complementary(card1,card1.copy(),H_card1)
    # cv2.imshow("undistorted card1",undistorted_card1)
    cv2.imwrite("undistorted_card1_p2p.jpg",undistorted_card1)
    image = card1.copy()
    for i in [[487, 250], [610, 1114], [1221, 799], [1244, 171]]:
        image = cv2.circle(image,tuple(i), radius=0, color=(0, 0, 255), thickness=6)
    cv2.imwrite("card1_point_to_point_PQRS.jpg",image)

      # painting [75,179],[78,654],[805,621],[804,219]
      # undistorted painting [75,179],[75,593],[805,593],[805,179]

    painting_PQRS = np.array([[1090, 303], [1091, 662], [1530, 700], [1531, 275]])
    painting_undistorted_PQRS = np.array([[0,0],[0,100],[100,100],[100,0]])
    H_painting = point_to_point_calculate_h(painting_PQRS,painting_undistorted_PQRS)
    undistorted_painting = point_mapping_complementary(painting,painting.copy(),H_painting)
    cv2.imwrite("undistorted_painting_p2p.jpg",undistorted_painting)
    image = painting.copy()
    for i in [[1090, 303], [1091, 662], [1530, 700], [1531, 275]]:
        image = cv2.circle(image,tuple(i), radius=0, color=(0, 0, 255), thickness=6)
    cv2.imwrite("painting_point_to_point_PQRS.jpg",image)

    # two-step method
    card1_2_pair_parallel_line_points = np.array(
        [[487, 250], [610, 1114], [1221, 799], [1244, 171], [487, 250], [1244, 171], [610, 1
```

```
H_two_step_1_card1 = send_VL_to_infinite_h_calculate(card1_2_pair_parallel_line_points)
card1_two_step_1 = point_mapping_complementary(card1, card1.copy(), H_two_step_1_card1)
cv2.imwrite("card1_two_step_mid.jpg", card1_two_step_1)
transformed_card1_points = calculate_transformed_points([[487, 250], [610, 1114], [1221,
print("transformed card1 points: ",transformed_card1_points)
H_two_step_2_card1 = undistort_affine_h_calculate(transformed_card1_points)
card1_by_two_step = point_mapping_complementary(card1,card1.copy(),np.matmul(H_two_step_
cv2.imwrite("card1_two_step.jpg", card1_by_two_step)
image = card1.copy()
for i in range(4):
    image = cv2.line(image, tuple(card1_2_pair_parallel_line_points[2*i]), tuple(card1_2
cv2.imwrite("card1_two_step_parallel_lines.jpg", image)

painting_2_pair_parallel_line_points = np.array(
    [[1090, 303], [1091, 662], [1530, 700], [1531, 275], [1090, 303], [1531, 275], [1091
H_two_step_1_painting = send_VL_to_infinite_h_calculate(painting_2_pair_parallel_line_po
painting_two_step_1 = point_mapping_complementary(painting, painting.copy(), H_two_step_
cv2.imwrite("painting_two_step_mid.jpg", painting_two_step_1)
transformed_painting_points = calculate_transformed_points([[1090, 303], [1091, 662], [1
print("transformed painting points: ", transformed_painting_points)
H_two_step_2_painting = undistort_affine_h_calculate(transformed_painting_points)
painting_by_two_step = point_mapping_complementary(painting,painting.copy(),np.matmul(H_
cv2.imwrite("painting_two_step.jpg", painting_by_two_step)
image = painting.copy()
for i in range(4):
    image = cv2.line(image, tuple(painting_2_pair_parallel_line_points[2*i]), tuple(pair
cv2.imwrite("painting_two_step_parallel_lines.jpg", image)


# one-step method
card1_one_step_points = [[487, 250], [610, 1114], [1221, 799], [1244, 171],[631,1080],[5
H_one_step_card1 = one_step_calculate_H(card1_one_step_points)
card1_one_step = point_mapping_complementary(card1,card1.copy(),H_one_step_card1)
cv2.imwrite("card1_one_step.jpg",card1_one_step)
image = card1.copy()
for i in [[487, 250], [610, 1114], [1221, 799], [1244, 171], [631,1080],[526,284],[1226,
    image = cv2.circle(image, tuple(i), radius=0, color=(0, 0, 255), thickness=6)
cv2.imwrite("card1_one_step_PQRSQPS.jpg",image)

painting_one_step_points = [[1090, 303], [1091, 662], [1530, 700], [1531, 275],[970,742]
H_one_step_painting = one_step_calculate_H(painting_one_step_points)
painting_one_step = point_mapping_complementary(painting,painting.copy(),H_one_step_pair
cv2.imwrite("painting_one_step.jpg",painting_one_step)
image = painting.copy()
for i in [[1090, 303], [1091, 662], [1530, 700], [1531, 275],[970,742],[972,101],[1601,0
    image = cv2.circle(image, tuple(i), radius=0, color=(0, 0, 255), thickness=6)
```

```
cv2.imwrite("painting_one_step_PQRSQPS.jpg",image)

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