ECE 661 Computer Vision: HW2

Qiuchen Zhai
qzhai@purdue.edu

September 22, 2020

1 Logic and Methodology

1.1 Point-to-point correspondences

The linear mapping on homogeneous 3-vectors is given by

$$\mathbf{x}' = \mathbf{H}\mathbf{x} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^3$ is the domain point, $\mathbf{x}' \in \mathbb{R}^3$ is the corresponding range point, and $\mathbf{H}$ is the HC matrix. Thus we have

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} h_{11}x_1 + h_{12}x_2 + h_{13}x_3 \\ h_{21}x_1 + h_{22}x_2 + h_{23}x_3 \\ h_{31}x_1 + h_{32}x_2 + x_3 \end{pmatrix} \tag{2}$$

Let $x = \frac{x_1}{x_3}$, $y = \frac{x_2}{x_3}$, $x' = \frac{x_1'}{x_3'}$, and $y' = \frac{x_2'}{x_3'}$. Then we get the following equations

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + 1} \tag{3}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + 1} \tag{4}$$

Therefore, the HC matrix could be solved by

$$\mathbf{h} = \mathbf{A}^{-1}\mathbf{b} \tag{5}$$

where $\mathbf{A} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x_3' & -y_3x_3' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y_3' & -y_3y_3' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x_4' & -y_4x_4' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y_4' & -y_4y_4' \end{pmatrix}$, $\mathbf{b} = \begin{pmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \\ x_4' \\ y_4' \end{pmatrix}$, and $\mathbf{h} = \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix}$.

One important fundamental invariant of the Projective Linear group PL(3) is that straight lines going to straight lines. Once the HC matrix is obtained, the image distortion could be removed by point-to-point correspondences.

1.2 Two-step method

1.2.1 Remove the projective distortion

It's been proved that, a homography $\mathbf{H}$ is affine if and only if the *line at infinity* $\mathbf{l}_\infty$ is mapped to $\mathbf{l}_\infty$. While the *vanishing line* the *line at infinity* mapped to a physical line through a general projective transform mapping, we could utilize the *vanishing line* to get rid of projective distortion. First, we need to choose two pairs of parallel lines in the physical world and compute their intersection points in the image with distortion. Assume the pixel coordinates of the corners of a rectangular frame formed by the two pairs of lines are p, q, s, r, the two pairs of lines and their intersection points are given by

$$l_{pq} = p \times q, l_{rs} = r \times s \Rightarrow x_1 = l_{pq} \times l_{rs} \tag{6}$$
$$l_{qs} = q \times s, l_{pr} = p \times r \Rightarrow x_2 = l_{qs} \times l_{pr} \tag{7}$$

Then, the vanish line could be obtained by computing the cross product of the two intersection points.

$$\mathbf{l}_{VL} = x_1 \times x_2 \tag{8}$$

After we get the vanishing line, the homography that sends $\mathbf{l}_{VL} = \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix}$ back to $\mathbf{l}_\infty = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ is

$$H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{pmatrix} \tag{9}$$

1.2.2 Remove the affine distortion

Since the affine distortion turns a 90° angle into some value θ, our correction should take θ back to the 90° angle. Assume θ is the angle between the lines $\mathbf{l} = \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix}$ and $\mathbf{m} = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix}$ in an image with affine distortion. Then we have

$$cos\theta = \frac{l_1 m_1 + l_2 m_2}{\sqrt{(l_1^2 + l_2^2)(m_1^2 + m_2^2)}} \tag{10}$$

Since $C_\infty^* = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$, we have

$$cos\theta = \frac{\mathbf{l}^T C_\infty^* \mathbf{m}}{\sqrt{(\mathbf{l}^T C_\infty^* \mathbf{l})(\mathbf{m}^T C_\infty^* \mathbf{m})}} \tag{11}$$

Now assume the angle θ is 90° in the original scene, we have the following constraint for estimating $\mathbf{H}$

$$\mathbf{l}'^T \mathbf{H} C_\infty^* \mathbf{H}^T \mathbf{l}' = \mathbf{0} \tag{12}$$

where

$$\mathbf{l}'^T \mathbf{H} C_\infty^* \mathbf{H}^T \mathbf{l}' = \begin{pmatrix} l_1' & l_2' & l_3' \end{pmatrix} \begin{pmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{A}^T & \mathbf{0} \\ \mathbf{t}^T & 1 \end{pmatrix} \begin{pmatrix} m_1' \\ m_2' \\ m_3' \end{pmatrix} = \begin{pmatrix} l_1' & l_2' & l_3' \end{pmatrix} \begin{pmatrix} \mathbf{A}\mathbf{A}^T & \mathbf{0} \\ \mathbf{t}^T & 0 \end{pmatrix} \begin{pmatrix} m_1' \\ m_2' \\ m_3' \end{pmatrix} \tag{13}$$

The above equation yields the following constraint

$$\begin{pmatrix} l_1^{'} & l_2^{'} \end{pmatrix} \mathbf{A}\mathbf{A}^T \begin{pmatrix} m_1^{'} \\ m_2^{'} \end{pmatrix} = \mathbf{0} \tag{14}$$

Let $\mathbf{S} = \mathbf{A}\mathbf{A}^T = \begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix}$, then

$$l_1^{'} m_1^{'} s_{11} + (l_1^{'} m_2^{'} + l_2^{'} m_1^{'}) s_{12} + l_2^{'} m_2^{'} s_{22} = 0 \tag{15}$$

There're three unknowns in the equation. Therefore, at least two pairs of orthogonal lines are needed to solve the $\mathbf{S}$. Assume $\mathbf{A}$ is positive definite , then we could use the singular value decomposition (SVD)

$$\mathbf{S} = \mathbf{A}\mathbf{A}^T = \mathbf{V}\mathbf{D}^2\mathbf{V}^T \tag{16}$$

Then

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^T \tag{17}$$

And the corresponding HC matrix is

$$\mathbf{H} = \begin{pmatrix} \mathbf{A} & 0 \\ 0 & 1 \end{pmatrix} \tag{18}$$

1.3 One-step method

In one-step method to remove both distortions, the projection of the $\mathbf{C}_\infty^*$ is

$$\mathbf{C}_\infty^{*^{'}} = \begin{pmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{pmatrix} \tag{19}$$

In HC matrix, we could set $f = 1$ since only the ratios matter. Therefore, we need to identify five pairs of orthogonal lines in world coordinates to estimate the five unknowns $a, b, c, d, and e$ by solving

$$\mathbf{l}^{'T} C_\infty^{*^{'}} \mathbf{m}^{'} = 0 \tag{20}$$

For each pair of orthogonal lines $\mathbf{l} = \begin{pmatrix} l_1^{'} \\ l_2^{'} \\ l_3^{'} \end{pmatrix}$ and $\mathbf{m} = \begin{pmatrix} m_1^{'} \\ m_2^{'} \\ m_3^{'} \end{pmatrix}$, we have

$$l_1^{'} m_1^{'} a + \frac{1}{2}(l_1^{'} m_2^{'} + l_2^{'} m_1^{'}) b + l_2^{'} m_2^{'} c + \frac{1}{2}(l_1^{'} m_3^{'} + l_3^{'} m_1^{'}) d + \frac{1}{2}(l_2^{'} m_3^{'} + l_3^{'} m_2^{'}) e = -l_3^{'} m_3^{'} \tag{21}$$

After estimating $\mathbf{C}_\infty^{*^{'}}$, the homography $\mathbf{H}$ that corrects both projective and affine distortion could be obtained by singular value decomposition of $\mathbf{C}_\infty^{*^{'}}$

$$\mathbf{C}_\infty^{*^{'}} = \mathbf{H} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \mathbf{H}^T \tag{22}$$

2 Results

2.1 Task1

2.1.1 Method 1: Point-to-point Correspondences

1. Point-to-point correspondences applied to the first image:

Figure 1: original Img1

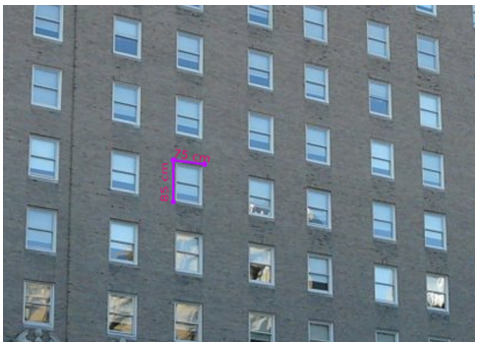

Figure 2: measurements of Img1

Figure 3: labelled points in Img1

Figure 4: Img1 after ptp mapping

2. Point-to-point correspondences applied to the second image:

Figure 5: original Img2

Figure 6: measurements of Img2

Figure 7: labelled points in Img2

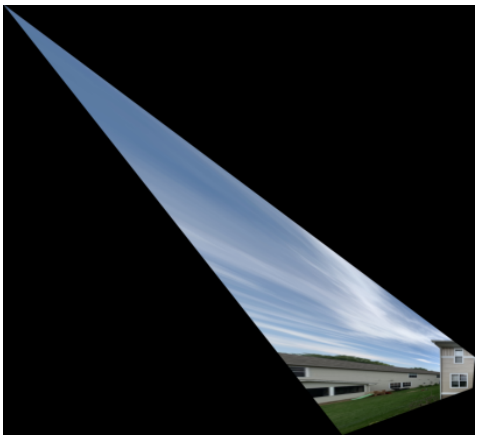

Figure 8: Img2 after ptp mapping

3. Point-to-point correspondences applied to the third image:

Figure 9: original Img3

Figure 10: measurements of Img3

Figure 11: labelled points in Img3

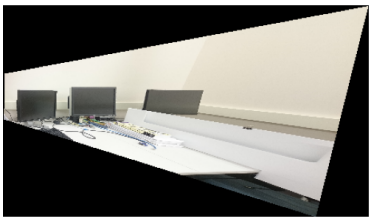

Figure 12: Img3 after ptp mapping

2.1.2 Method 2: Two-step Method

1. Two-step method applied to the first image:

Figure 13: original Img1

Figure 14: lines used for Img1

Figure 15: after removing projective distortion

Figure 16: after removing affine distortion

2. Two-step method Correspondences applied to the second image:

Figure 17: original Img2

Figure 18: lines used for Img2

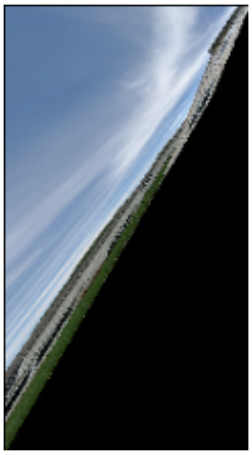

Figure 19: after removing projective distortion

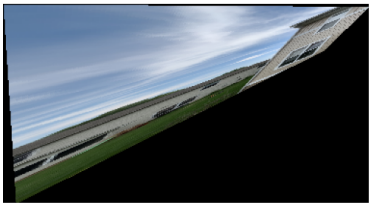

Figure 20: after removing affine distortion

3. Two-step method applied to the third image:

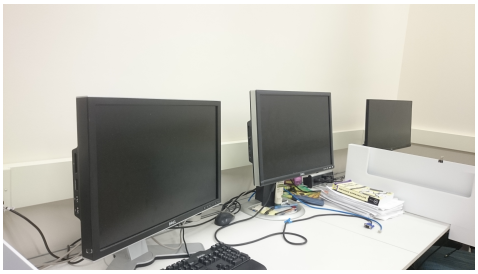

Figure 21: original Img3

Figure 22: lines used for Img3

Figure 23: after removing projective distortion

Figure 24: after removing affine distortion

Figure 25: after removing affine distortion (Enlarged version of Figure 24)

2.1.3 Method 3: One-step Method

1. One-step method applied to the first image:

Figure 26: lines used for Img1

Figure 27: result of Img1

2. One-step method Correspondences applied to the second image:

Figure 28: lines used for Img2

Figure 29: result of Img2

3. One-step method applied to the third image:

Figure 30: lines used for Img3

Figure 31: result of Img3

2.2 Task2

2.2.1 Method 1: Point-to-point Correspondences

1. Point-to-point correspondences applied to the fourth image:

Figure 32: original Img4

Figure 33: measurements of Img4

Figure 34: labelled points in Img4

Figure 35: Img4 after ptp mapping

2. Point-to-point correspondences applied to the fifth image:

Figure 36: original Img5

Figure 37: measurements of Img5

Figure 38: labelled points in Img5

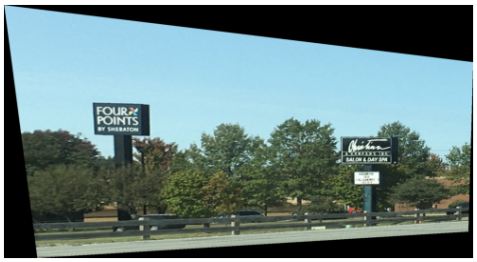

Figure 39: Img5 after ptp mapping

2.2.2 Method2: Two-step Method

1. Two-step method applied to the fourth image:

Figure 40: original Img4

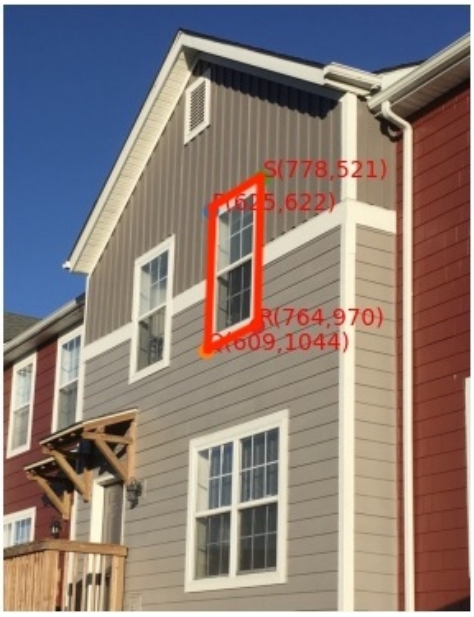

Figure 41: lines used for Img4

Figure 42: after removing projective distortion

Figure 43: after removing affine distortion

2. Two-step method applied to the fifth image:

Figure 44: original Img5

Figure 45: lines used for Img5

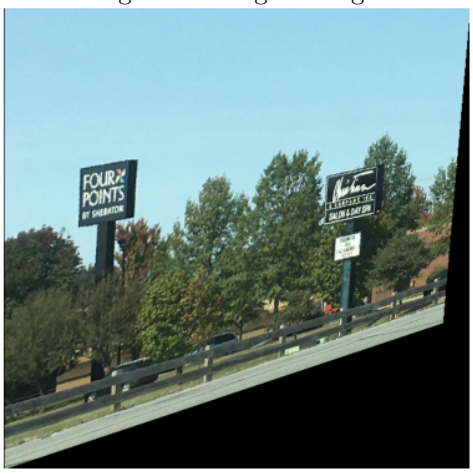

Figure 46: after removing projective distortion

Figure 47: after removing affine distortion

2.2.3 Method 3: One-step Method

1. One-step method applied to the fourth image:

Figure 48: lines used for Img4

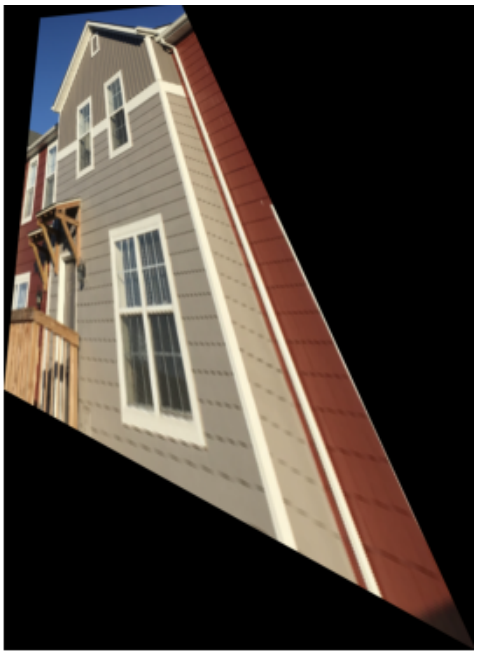

Figure 49: result of Img4

2. One-step method Correspondences applied to the fifth image:

Figure 50: lines used for Img5

Figure 51: result of Img5

2.3 Observations

- Both the point-to-point correspondences and one-step method are sensitive to the initial points. Thus we need to be careful while choosing the points and lines.
- The point-to-point correspondences method is accurate though sometimes we might not be able to retrieve the physical measurements in the world plane.
- The two-step method is robust and we can see the process of getting rid of projective distortion and affine distortion.
- The results from two-step method could still include distortions.
- The two-step method is fast.
- The one-step method is sensitive to the orthogonal lines. However, we could pick up more than five pairs of orthogonal lines in the image to facilitate the result.

3 Extra credits

The first image is utilized to demonstrate the length ratios. First, consider the window in the real-world shown in left with physical measurements. We could easily compute the angle values $\alpha = 41.423665625002656°$ and $\beta = 48.57633437499736°$.

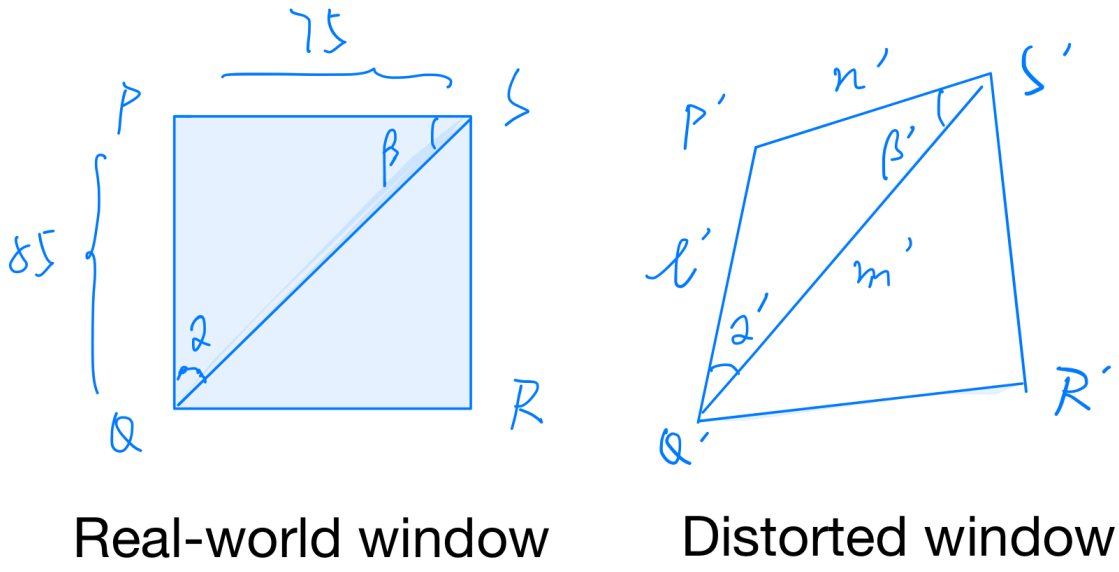

Figure 52: geometry between two coordinates

To compute the length ratios in the original scene using $\mathbf{C}_{\infty}^{*'}$, we need to find the corresponding points in the distorted image.

Figure 53: Point indication for image used

Then we estimate the angles by

$$cos\ \alpha = \frac{\mathbf{l}'^T C_\infty^* \mathbf{m}'}{\sqrt{(\mathbf{l}'^T C_\infty^* \mathbf{l}')(\mathbf{m}'^T C_\infty^* \mathbf{m}')}} \tag{23}$$

$$cos\ \beta = \frac{\mathbf{n}'^T C_\infty^* \mathbf{m}'}{\sqrt{(\mathbf{n}'^T C_\infty^* \mathbf{n}')(\mathbf{m}'^T C_\infty^* \mathbf{m}')}} \tag{24}$$

where $C_\infty^* = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$, $\mathbf{l}' = \mathbf{P}' \times \mathbf{Q}'$, $\mathbf{m}' = \mathbf{S}' \times \mathbf{Q}'$, $\mathbf{n}' = \mathbf{P}' \times \mathbf{S}'$.

Thus we get $\alpha_{est} = 41.51984067302296°$ and $\beta_{est} = 52.679921802611936°$. And the length ratio is then given by $sin\ \alpha_{est} : sin\ \beta_{est} \approx 0.8335367861149746$, which is close to the real length ratio 0.8823529411764706.

4 Source Code

4.1 method1.py

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage import io


def compute_hc(src, dst):
    """
    The module computes the homography matrix by AH = b.
    :param src: source points.
    :param dst: mapped points.
    :return: homography matrix H.
    """

    A = np.zeros((8, 8))
    b = np.zeros((8, 1))

    for i in range(len(src)):
        A[2 * i] = [src[i][0], src[i][1], 1, 0, 0, 0, -src[i][0]*dst[i][0], -src[i
    ][1]*dst[i][0]]
        A[2 * i + 1] = [0, 0, 0, src[i][0], src[i][1], 1, -src[i][0]*dst[i][1], -src[i
    ][1]*dst[i][1]]
        b[2 * i] = dst[i][0]
        b[2 * i + 1] = dst[i][1]

    h = np.dot(np.linalg.pinv(A), b)
    homo_mat = np.append(h, 1)

    return homo_mat.reshape((3, 3))


def compute_pixel_val(img, loc):
    """
    This module returns the pixel value given by weighting average of neighbor pixels.
    :param img: the mapping img.
    :param loc: the coordinates of mapped points.
    :return: the pixel value.
    """

    loc0_f = np.int(np.floor(loc[0]))
    loc1_f = np.int(np.floor(loc[1]))
    loc0_c = np.int(np.ceil(loc[0]))
    loc1_c = np.int(np.ceil(loc[1]))
    a = img[loc0_f][loc1_f]
    b = img[loc0_f][loc1_c]
    c = img[loc0_c][loc1_f]
    d = img[loc0_c][loc1_c]
    dx = float(loc[0] - loc0_f)
    dy = float(loc[1] - loc1_f)
    Wa = 1 / np.linalg.norm([dx, dy])
    Wb = 1 / np.linalg.norm([1 - dx, dy])
    Wc = 1 / np.linalg.norm([dx, 1 - dy])
    Wd = 1 / np.linalg.norm([1 - dx, 1 - dy])
    output = (a * Wa + b * Wb + c * Wc + d * Wd) / (Wa + Wb + Wc + Wd)

    return output


def ptp_mapping(img, h_mat):
    # Determine the coordinates of domain plane
```

```
    m, n, d = np.shape(img)
    P = np.array([0, 0, 1])
    Q = np.array([0, m-1, 1])
    S = np.array([n-1, 0, 1])
    R = np.array([n-1, m-1, 1])
    # Compute the projection of corners
    locP = np.dot(h_mat, P) / np.dot(h_mat, P)[2]
    locQ = np.dot(h_mat, Q) / np.dot(h_mat, Q)[2]
    locS = np.dot(h_mat, S) / np.dot(h_mat, S)[2]
    locR = np.dot(h_mat, R) / np.dot(h_mat, R)[2]
    # Determin the size of mapped image
    x_min = np.int(np.floor(np.min([locP[1], locQ[1], locR[1], locS[1]])))
    x_max = np.int(np.ceil(np.max([locP[1], locQ[1], locR[1], locS[1]])))
    x_scale = x_max - x_min
    y_min = np.int(np.floor(np.min([locP[0], locQ[0], locR[0], locS[0]])))
    y_max = np.int(np.ceil(np.max([locP[0], locQ[0], locR[0], locS[0]])))
    y_scale = y_max - y_min

    # Compute scaling factor
    # if we fix width, then the scaling factor s_w = w_o / w_i
    scaling_w = y_scale / n
    # if we fix length, then the scaling factor s_h = h_o / h_i
    scaling_h = x_scale / m
    # s = max{s_w, s_h}
    s = np.maximum(scaling_w, scaling_h)
    if s < 1:
        s = 1
    # Determine the ourput size
    output = np.zeros((int(np.ceil(x_scale/s)), int(np.ceil(y_scale/s)), 3))
    # Compute the projection
    h_inv = np.linalg.pinv(h_mat) / np.linalg.pinv(h_mat)[2][2]
    # Create array
    y_pt, x_pt = np.meshgrid(np.arange(0, y_scale*s, 1*s), np.arange(0, x_scale*s, 1*s
    ))
    # flattened array along axis=0
    pts = np.vstack((y_pt.ravel(), x_pt.ravel())).T + np.array([[y_min, x_min]])
    # Add third coordinates
    pts = np.append(pts, np.ones([len(pts), 1]), 1)
    # Transformation
    translated_pts = (np.dot(h_inv, pts.T)).T
    # Normalization
    translated_pts = translated_pts[:, :2] / translated_pts[:, [-1]]
    for i in range(0, x_scale):
        for j in range(0, y_scale):
            loc0 = translated_pts[i*y_scale + j, 1]
            loc1 = translated_pts[i*y_scale + j, 0]
            if (loc0 > 0) and (loc1 > 0) and (loc0 < img.shape[0] - 1) and (loc1 < img
    .shape[1] - 1):
                output[i][j] = compute_pixel_val(img, [loc0, loc1])

    return output.astype(np.uint8)


point = ['P', 'Q', 'S', 'R']
# Image projection
img1 = io.imread('hw3_Task1_Images/Images/Img1.JPG')
PQSR1 = np.array([[642, 498], [642, 532], [666, 503], [666, 537]])
PQSR = np.array([[0, 0], [0, 85], [75, 0], [75, 85]])
# Plot
plt.imshow(img1)
plt.scatter(PQSR1[0][0], PQSR1[0][1])
plt.annotate(point[0] + '({},{})'.format(PQSR1[0][0], PQSR1[0][1]), (PQSR1[0][0]-50,
    PQSR1[0][1]-50), c='r')
plt.scatter(PQSR1[1][0], PQSR1[1][1])
```

```
plt.annotate(point[1] + '({},{})'.format(PQSR1[1][0], PQSR1[1][1]), (PQSR1[1][0]-100,
    PQSR1[1][1]+100), c='r')
plt.scatter(PQSR1[2][0], PQSR1[2][1])
plt.annotate(point[2] + '({},{})'.format(PQSR1[2][0], PQSR1[2][1]), (PQSR1[2][0]+100,
    PQSR1[2][1]-100), c='r')
plt.scatter(PQSR1[3][0], PQSR1[3][1])
plt.annotate(point[3] + '({},{})'.format(PQSR1[3][0], PQSR1[3][1]), (PQSR1[3][0]+50,
    PQSR1[3][1]+50), c='r')
plt.axis('off')
plt.savefig('img1_labelledpt.jpeg')
plt.show()
plt.clf()
# Computation
H = compute_hc(PQSR1, PQSR)
output = ptp_mapping(img1, H)
plt.imshow(output)
plt.axis('off')
plt.savefig("img1_results.jpeg")
plt.show()
plt.clf()

img2 = io.imread('hw3_Task1_Images/Images/Img2.jpeg')
PQSR2 = np.array([[480, 722], [481, 874], [600, 739], [606, 923]])
PQSR = np.array([[0, 0], [0, 74], [84, 0], [84, 74]])
# Plot
plt.imshow(img2)
for i in range(4):
    plt.scatter(PQSR2[i][0], PQSR2[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR2[i][0], PQSR2[i][1]), (PQSR2[i][0],
    PQSR2[i][1]), c='r')
plt.axis('off')
plt.savefig('img2_labelledpt.jpeg')
plt.show()
plt.clf()
# Computation
H2 = compute_hc(PQSR2, PQSR)
output = ptp_mapping(img2, H2)
plt.imshow(output)
plt.axis('off')
plt.savefig("img2_results.jpeg")
plt.show()
plt.clf()


img3 = io.imread('hw3_Task1_Images/Images/Img3.JPG')
PQSR3 = np.array([[2060, 700], [2092, 1483], [2666, 720], [2695, 1333]])
PQSR = np.array([[0, 0], [0, 36], [55, 0], [55, 36]])
# Plot
plt.imshow(img3)
for i in range(4):
    plt.scatter(PQSR3[i][0], PQSR3[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR3[i][0], PQSR3[i][1]), (PQSR3[i][0],
    PQSR3[i][1]), c='r')
plt.title('painting3.jpeg')
plt.axis('off')
plt.savefig('img3_labelledpt.jpeg')
plt.show()
plt.clf()
# Computation
H3 = compute_hc(PQSR3, PQSR)
output = ptp_mapping(img3, H3)
plt.imshow(output)
plt.axis('off')
plt.savefig("img3_results.jpeg")
```

```
plt.show()
plt.clf()


img4 = io.imread('hw3_Task1_Images/Images/Img4.jpeg')
PQSR4 = np.array([[625, 622], [609, 1044], [778, 521], [764, 970]])
PQSR = np.array([[0, 0], [0, 120], [40, 0], [40, 120]])
# Plot
plt.imshow(img4)
for i in range(4):
    plt.scatter(PQSR4[i][0], PQSR4[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR4[i][0], PQSR4[i][1]), (PQSR4[i][0],
    PQSR4[i][1]), c='r')
plt.axis('off')
plt.savefig('img4_method1_labels.jpeg')
plt.show()
plt.clf()
# Computation
H4 = compute_hc(PQSR4, PQSR)
output = ptp_mapping(img4, H4)
plt.imshow(output)
plt.axis('off')
plt.savefig("img4_results.jpeg")
plt.show()
plt.clf()


img5 = io.imread('hw3_Task1_Images/Images/img5.jpeg')
PQSR5 = np.array([[573, 302], [572, 386], [692, 296], [692, 384]])
PQSR = np.array([[0, 0], [0, 100], [200, 0], [200, 100]])
# Plot
plt.imshow(img5)
for i in range(4):
    plt.scatter(PQSR5[i][0], PQSR5[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR5[i][0], PQSR5[i][1]), (PQSR5[i][0],
    PQSR5[i][1]), c='r')
plt.axis('off')
plt.savefig('img5_labelledpt.jpeg')
plt.show()
plt.clf()
# Computation
H5 = compute_hc(PQSR5, PQSR)
output = ptp_mapping(img5, H5)
plt.imshow(output)
plt.axis('off')
plt.savefig("img5_results.jpeg")
plt.show()
plt.clf()


img6 = io.imread('hw3_Task1_Images/Images/Img6.jpeg')
PQSR6 = np.array([[232, 63], [232, 1466], [693, 133], [692, 1453]])
PQSR = np.array([[0, 0], [0, 50], [20, 0], [20, 50]])
# Plot
plt.imshow(img6)
for i in range(4):
    plt.scatter(PQSR6[i][0], PQSR6[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR6[i][0], PQSR6[i][1]), (PQSR6[i][0],
    PQSR6[i][1]), c='r')
plt.axis('off')
plt.savefig('img6_method1_labels.jpeg')
plt.show()
plt.clf()
# Computation
```

```
H6 = compute_hc(PQSR6, PQSR)
output = ptp_mapping(img6, H6)
plt.imshow(output)
plt.axis('off')
plt.savefig("img6_results.jpeg")
plt.show()
plt.clf()
```

method1.py

4.2 method2.py

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage import io


def compute_pixel_val(img, loc):
    """
    This module returns the pixel value given by weighting average of neighbor pixels.
    :param img: the mapping img.
    :param loc: the coordinates of mapped points.
    :return: the pixel value.
    """

    loc0_f = np.int(np.floor(loc[0]))
    loc1_f = np.int(np.floor(loc[1]))
    loc0_c = np.int(np.ceil(loc[0]))
    loc1_c = np.int(np.ceil(loc[1]))
    a = img[loc0_f][loc1_f]
    b = img[loc0_f][loc1_c]
    c = img[loc0_c][loc1_f]
    d = img[loc0_c][loc1_c]
    dx = float(loc[0] - loc0_f)
    dy = float(loc[1] - loc1_f)
    Wa = 1 / np.linalg.norm([dx, dy])
    Wb = 1 / np.linalg.norm([1 - dx, dy])
    Wc = 1 / np.linalg.norm([dx, 1 - dy])
    Wd = 1 / np.linalg.norm([1 - dx, 1 - dy])
    output = (a * Wa + b * Wb + c * Wc + d * Wd) / (Wa + Wb + Wc + Wd)

    return output


def img_map(img, h_mat):
    # Determine the coordinates of domain plane
    m, n, d = np.shape(img)
    P = np.array([0, 0, 1])
    Q = np.array([0, m-1, 1])
    S = np.array([n-1, 0, 1])
    R = np.array([n-1, m-1, 1])
    # Compute the projection of corners
    locP = np.dot(h_mat, P) / np.dot(h_mat, P)[2]
    locQ = np.dot(h_mat, Q) / np.dot(h_mat, Q)[2]
    locS = np.dot(h_mat, S) / np.dot(h_mat, S)[2]
    locR = np.dot(h_mat, R) / np.dot(h_mat, R)[2]
    # Determin the size of mapped image
    x_min = np.int(np.floor(np.min([locP[1], locQ[1], locR[1], locS[1]])))
    x_max = np.int(np.ceil(np.max([locP[1], locQ[1], locR[1], locS[1]])))
    x_scale = x_max - x_min
    y_min = np.int(np.floor(np.min([locP[0], locQ[0], locR[0], locS[0]])))
    y_max = np.int(np.ceil(np.max([locP[0], locQ[0], locR[0], locS[0]])))
```

```
    y_scale = y_max - y_min

    # Compute scaling factor
    # if we fix width, then the scaling factor s_w = w_o / w_i
    scaling_w = y_scale / n
    # if we fix length, then the scaling factor s_h = h_o / h_i
    scaling_h = x_scale / m
    # s = max{s_w, s_h}
    s = np.maximum(scaling_w, scaling_h)
    if s < 1:
        s = 1
    # Determine the ourput size
    output = np.zeros((int(np.ceil(x_scale/s)), int(np.ceil(y_scale/s)), 3))
    # Compute the projection
    h_inv = np.linalg.pinv(h_mat) / np.linalg.pinv(h_mat)[2][2]
    # Create array
    y_pt, x_pt = np.meshgrid(np.arange(0, y_scale*s, 1*s), np.arange(0, x_scale*s, 1*s
    ))
    # flattened array along axis=0
    pts = np.vstack((y_pt.ravel(), x_pt.ravel())).T + np.array([[y_min, x_min]])
    # Add third coordinates
    pts = np.append(pts, np.ones([len(pts), 1]), 1)
    # Transformation
    translated_pts = (np.dot(h_inv, pts.T)).T
    # Normalization
    translated_pts = translated_pts[:, :2] / translated_pts[:, [-1]]
    for i in range(0, x_scale):
        for j in range(0, y_scale):
            loc0 = translated_pts[i*y_scale + j, 1]
            loc1 = translated_pts[i*y_scale + j, 0]
            if (loc0 > 0) and (loc1 > 0) and (loc0 < img.shape[0] - 1) and (loc1 < img
    .shape[1] - 1):
                output[i][j] = compute_pixel_val(img, [loc0, loc1])

    return output.astype(np.uint8)


def H_projective(pts):
    l1 = np.cross(pts[0], pts[1])
    l2 = np.cross(pts[2], pts[3])
    int_pt12 = np.cross(l1, l2) / np.cross(l1, l2)[2]

    l3 = np.cross(pts[0], pts[2])
    l4 = np.cross(pts[1], pts[3])
    int_pt34 = np.cross(l3, l4) / np.cross(l3, l4)[2]

    l_VL = np.cross(int_pt12, int_pt34) / np.cross(int_pt12, int_pt34)[2]
    H = np.array([[1, 0, 0], [0, 1, 0], [l_VL[0], l_VL[1], l_VL[2]]])

    return H


def H_affine(pts):
    l1 = np.cross(pts[0], pts[1]) / np.cross(pts[0], pts[1])[2]
    m1 = np.cross(pts[0], pts[2]) / np.cross(pts[0], pts[2])[2]
    l2 = np.cross(pts[1], pts[3]) / np.cross(pts[1], pts[3])[2]
    m2 = np.cross(pts[2], pts[3]) / np.cross(pts[2], pts[3])[2]

    A = np.array([[l1[0] * m1[0], l1[0] * m1[1] + l1[1] * m1[0]], [l2[0] * m2[0], l2
    [0] * m2[1] + l2[1] * m2[0]]])
    b = np.array([[-l1[1] * m1[1]], [-l2[1] * m2[1]]])
    S = np.zeros((2, 2), dtype=float)
    S[0][0] = np.dot(np.linalg.pinv(A), b)[0]
    S[0][1] = np.dot(np.linalg.pinv(A), b)[1]
```

```
    S[1][0] = np.dot(np.linalg.pinv(A), b)[1]
    S[1][1] = 1

    U, D, V = np.linalg.svd(S, full_matrices=True)
    sol = np.dot(np.dot(U, np.sqrt(np.diag(D))), U.transpose())
    H = np.array([[sol[0][0], sol[0][1], 0], [sol[1][0], sol[1][1], 0], [0, 0, 1]])

    return H


point = ["P", "Q", "S", "R"]
# Remove distortion of img1
img1 = io.imread('hw3_Task1_Images/Images/Img1.JPG')
PQSR1 = np.array([[462, 165], [431, 730], [1415, 482], [1462, 810]])
# Plot
plt.imshow(img1)
for i in range(4):
    plt.scatter(PQSR1[i][0], PQSR1[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR1[i][0], PQSR1[i][1]), (PQSR1[i][0],
    PQSR1[i][1]), c='r')
plt.axis('off')
plt.savefig('img1_method2_labelled1.jpeg')
plt.show()
plt.clf()
# Computation
pts = np.array([[462, 165, 1], [431, 730, 1], [1415, 482, 1], [1462, 810, 1]])
H1 = H_projective(pts)
output1 = img_map(img1, H1)
plt.imshow(output1, cmap='gray')
plt.axis('off')
plt.savefig("img1_method2_projective.jpeg")
plt.show()
plt.clf()
H2 = H_affine(pts)
output2 = img_map(img1, np.dot(np.linalg.pinv(H2), H1))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img1_method2_affine.jpeg")
plt.show()
plt.clf()



# Remove distortion of img2
img2 = io.imread('hw3_Task1_Images/Images/Img2.jpeg')
PQSR2 = np.array([[368, 552], [362, 853], [621, 510], [642, 974]])
# Plot
plt.imshow(img2)
for i in range(4):
    plt.scatter(PQSR2[i][0], PQSR2[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR2[i][0], PQSR2[i][1]), (PQSR2[i][0],
    PQSR2[i][1]), c='r')
plt.axis('off')
plt.savefig('img2_method2_labels.jpeg')
plt.show()
plt.clf()
# Computation
pts = np.array([[368, 552, 1], [362, 853, 1], [621, 510, 1], [642, 974, 1]])
H1 = H_projective(pts)
output1 = img_map(img2, H1)
plt.imshow(output1, cmap='gray')
plt.axis('off')
plt.savefig("img2_method2_projective.jpeg")
plt.show()
```

```
plt.clf()
H2 = H_affine(pts)
output2 = img_map(img2, np.dot(np.linalg.pinv(H2), H1))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img2_method2_affine.jpeg")
plt.show()
plt.clf()


# Remove distortion of img3
img3 = io.imread('hw3_Task1_Images/Images/Img3.JPG')
PQSR3 = np.array([[2060, 700], [2092, 1483], [2666, 720], [2695, 1333]])
# Plot
plt.imshow(img3, cmap='gray')
for i in range(4):
    plt.scatter(PQSR3[i][0], PQSR3[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR3[i][0], PQSR3[i][1]), (PQSR3[i][0],
    PQSR3[i][1]), c='r')
plt.axis('off')
plt.savefig('img3_method2_labels.jpeg')
plt.show()
plt.clf()
# Computation
pts = np.array([[2060, 700, 1], [2092, 1483, 1], [2666, 720, 1], [2695, 1333, 1]])
H1 = H_projective(pts)
output1 = img_map(img3, H1)
plt.imshow(output1, cmap='gray')
plt.axis('off')
plt.savefig("img3_method2_projective.jpeg")
plt.show()
plt.clf()
H2 = H_affine(pts)
output2 = img_map(img3, np.dot(np.linalg.pinv(H2), H1))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img3_method2_affine.jpeg")
plt.show()
plt.clf()


# Remove distortion of img4
img4 = io.imread('hw3_Task1_Images/Images/Img4.jpeg')
PQSR4 = np.array([[625, 622], [609, 1044], [778, 521], [764, 970]])
# Plot
plt.imshow(img4)
for i in range(4):
    plt.scatter(PQSR4[i][0], PQSR4[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR4[i][0], PQSR4[i][1]), (PQSR4[i][0],
    PQSR4[i][1]), c='r')
plt.axis('off')
plt.savefig('img4_method2_labels.jpeg')
plt.show()
plt.clf()
# Computation
pts = np.array([[625, 622, 1], [609, 1044, 1], [778, 521, 1], [764, 970, 1]])
H1 = H_projective(pts)
output1 = img_map(img4, H1)
plt.imshow(output1, cmap='gray')
plt.axis('off')
plt.savefig("img4_method2_projective.jpeg")
plt.show()
plt.clf()
H2 = H_affine(pts)
```

```
output2 = img_map(img4, np.dot(np.linalg.pinv(H2), H1))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img4_method2_affine.jpeg")
plt.show()
plt.clf()

# Remove distortion of img 5
img5 = io.imread('hw3_Task1_Images/Images/img5.jpeg')
PQSR5 = np.array([[573, 302], [572, 386], [692, 296], [692, 384]])
# Plot
plt.imshow(img5)
for i in range(4):
    plt.scatter(PQSR5[i][0], PQSR5[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR5[i][0], PQSR5[i][1]), (PQSR5[i][0],
    PQSR5[i][1]), c='r')
plt.axis('off')
plt.savefig('img5_method2_labels.jpeg')
plt.show()
plt.clf()
# Computation
pts = np.array([[573, 302, 1], [572, 386, 1], [692, 296, 1], [692, 384, 1]])
H1 = H_projective(pts)
output1 = img_map(img5, H1)
plt.imshow(output1, cmap='gray')
plt.axis('off')
plt.savefig("img5_method2_projective.jpeg")
plt.show()
plt.clf()
H2 = H_affine(pts)
output2 = img_map(img5, np.dot(np.linalg.pinv(H2), H1))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img5_method2_affine.jpeg")
plt.show()
plt.clf()

# Remove distortion of img6
img6 = io.imread('hw3_Task1_Images/Images/Img6.jpeg')
PQSR6 = np.array([[232, 63], [232, 1466], [693, 133], [692, 1453]])
# Plot
plt.imshow(img6)
for i in range(4):
    plt.scatter(PQSR6[i][0], PQSR6[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR6[i][0], PQSR6[i][1]), (PQSR6[i][0],
    PQSR6[i][1]), c='r')
plt.axis('off')
plt.savefig('img6_method2_labels.jpeg')
plt.show()
plt.clf()
# Computation
pts = np.array([[232, 63, 1], [232, 1466, 1], [693, 133, 1], [692, 1453, 1]])
H1 = H_projective(pts)
output1 = img_map(img6, H1)
plt.imshow(output1, cmap='gray')
plt.axis('off')
plt.savefig("img6_method2_projective.jpeg")
plt.show()
plt.clf()
H2 = H_affine(pts)
output2 = img_map(img6, np.dot(np.linalg.pinv(H2), H1))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img6_method2_affine.jpeg")
```

```
plt.show()
plt.clf()
```

method2.py

4.3 method3.py

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage import io


import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage import io


def compute_pixel_val(img, loc):
    """
    This module returns the pixel value given by weighting average of neighbor pixels.
    :param img: the mapping img.
    :param loc: the coordinates of mapped points.
    :return: the pixel value.
    """

    loc0_f = np.int(np.floor(loc[0]))
    loc1_f = np.int(np.floor(loc[1]))
    loc0_c = np.int(np.ceil(loc[0]))
    loc1_c = np.int(np.ceil(loc[1]))
    a = img[loc0_f][loc1_f]
    b = img[loc0_f][loc1_c]
    c = img[loc0_c][loc1_f]
    d = img[loc0_c][loc1_c]
    dx = float(loc[0] - loc0_f)
    dy = float(loc[1] - loc1_f)
    Wa = 1 / np.linalg.norm([dx, dy])
    Wb = 1 / np.linalg.norm([1 - dx, dy])
    Wc = 1 / np.linalg.norm([dx, 1 - dy])
    Wd = 1 / np.linalg.norm([1 - dx, 1 - dy])
    output = (a * Wa + b * Wb + c * Wc + d * Wd) / (Wa + Wb + Wc + Wd)

    return output


def img_map(img, h_mat):
    # Determine the coordinates of domain plane
    m, n, d = np.shape(img)
    P = np.array([0, 0, 1])
    Q = np.array([0, m-1, 1])
    S = np.array([n-1, 0, 1])
    R = np.array([n-1, m-1, 1])
    # Compute the projection of corners
    locP = np.dot(h_mat, P) / np.dot(h_mat, P)[2]
    locQ = np.dot(h_mat, Q) / np.dot(h_mat, Q)[2]
    locS = np.dot(h_mat, S) / np.dot(h_mat, S)[2]
    locR = np.dot(h_mat, R) / np.dot(h_mat, R)[2]
    # Determin the size of mapped image
    x_min = np.int(np.floor(np.min([locP[1], locQ[1], locR[1], locS[1]])))
    x_max = np.int(np.ceil(np.max([locP[1], locQ[1], locR[1], locS[1]])))
    x_scale = x_max - x_min
    y_min = np.int(np.floor(np.min([locP[0], locQ[0], locR[0], locS[0]])))
```

```
    y_max = np.int(np.ceil(np.max([locP[0], locQ[0], locR[0], locS[0]])))
    y_scale = y_max - y_min

    # Compute scaling factor
    # if we fix width, then the scaling factor s_w = w_o / w_i
    scaling_w = y_scale / n
    # if we fix length, then the scaling factor s_h = h_o / h_i
    scaling_h = x_scale / m
    # s = max{s_w, s_h}
    s = np.maximum(scaling_w, scaling_h)
    if s < 1:
        s = 1
    # Determine the ourput size
    output = np.zeros((int(np.ceil(x_scale/s)), int(np.ceil(y_scale/s)), 3))
    # Compute the projection
    h_inv = np.linalg.pinv(h_mat) / np.linalg.pinv(h_mat)[2][2]
    # Create array
    y_pt, x_pt = np.meshgrid(np.arange(0, y_scale*s, 1*s), np.arange(0, x_scale*s, 1*s
    ))
    # flattened array along axis=0
    pts = np.vstack((y_pt.ravel(), x_pt.ravel())).T + np.array([[y_min, x_min]])
    # Add third coordinates
    pts = np.append(pts, np.ones([len(pts), 1]), 1)
    # Transformation
    translated_pts = (np.dot(h_inv, pts.T)).T
    # Normalization
    translated_pts = translated_pts[:, :2] / translated_pts[:, [-1]]
    for i in range(0, x_scale):
        for j in range(0, y_scale):
            loc0 = translated_pts[i*y_scale + j, 1]
            loc1 = translated_pts[i*y_scale + j, 0]
            if (loc0 > 0) and (loc1 > 0) and (loc0 < img.shape[0] - 1) and (loc1 < img
    .shape[1] - 1):
                output[i][j] = compute_pixel_val(img, [loc0, loc1])

    return output.astype(np.uint8)


def compute_coe(array1, array2):
    output = np.array([array1[0] * array2[0], (array1[0]*array2[1]+array1[1]*array2
    [0])/2,
                       array1[1]*array2[2], (array1[0]*array2[2]+array1[2]*array2[0])
    /2,
                       (array1[1]*array2[2]+array1[2]*array2[1])/2])

    return output


def H_matrix(pts):
    # PQ \ PS
    l1 = np.cross(pts[0], pts[1]) / np.max(np.cross(pts[0], pts[1]))
    m1 = np.cross(pts[0], pts[2]) / np.max(np.cross(pts[0], pts[2]))
    # PS \ SR
    l2 = np.cross(pts[0], pts[2]) / np.max(np.cross(pts[0], pts[2]))
    m2 = np.cross(pts[2], pts[3]) / np.max(np.cross(pts[2], pts[3]))
    # SR \ QR
    l3 = np.cross(pts[2], pts[3]) / np.max(np.cross(pts[2], pts[3]))
    m3 = np.cross(pts[1], pts[3]) / np.max(np.cross(pts[1], pts[3]))
    # PQ \ QR
    l4 = np.cross(pts[0], pts[1]) / np.max(np.cross(pts[0], pts[1]))
    m4 = np.cross(pts[1], pts[3]) / np.max(np.cross(pts[1], pts[3]))
    # AB \ BC
    l5 = np.cross(pts[4], pts[5]) / np.max(np.cross(pts[4], pts[5]))
    m5 = np.cross(pts[5], pts[6]) / np.max(np.cross(pts[5], pts[6]))
```

```
    # Compute S_mat
    A = []
    A.append(compute_coe(l1, m1))
    A.append(compute_coe(l2, m2))
    A.append(compute_coe(l3, m3))
    A.append(compute_coe(l4, m4))
    A.append(compute_coe(l5, m5))
    A = np.asarray(A)
    b = np.array([[-l1[2] * m1[2]], [-l2[2] * m2[2]], [-l3[2] * m3[2]], [-l4[2] * m4
    [2]], [-l5[2] * m5[2]]])
    sol = np.dot(np.linalg.pinv(A), b) / np.max(np.abs(np.dot(np.linalg.pinv(A), b)))
    S = np.zeros((2, 2), dtype=float)
    S[0][0] = sol[0]
    S[0][1] = sol[1] / 2
    S[1][0] = sol[1] / 2
    S[1][1] = sol[2]
    # Compute SVD of S
    S_mat = np.array(S, dtype=float)
    U, D, V = np.linalg.svd(S_mat, full_matrices=True)
    temp = np.dot(np.dot(U, np.sqrt(np.diag(D))), U.transpose())
    v_vec = np.dot(np.linalg.pinv(temp), np.array([sol[3] / 2, sol[4] / 2]))
    H = np.array([[temp[0][0], temp[0][1], 0], [temp[1][0], temp[1][1], 0], [v_vec[0],
     v_vec[1], 1]], dtype=float)

    return H


# # Method 3 for image1
# img1 = io.imread('hw3_Task1_Images/Images/Img1.JPG')
# PQSR1 = np.array([[462, 165], [431, 730], [1415, 482], [1462, 810], [165, 69], [98,
    876], [425, 883]])
# point1 = ["P", "Q", "S", "R", "A", "B", "C"]
# # Plot
# plt.imshow(img1)
# for i in range(7):
#     plt.scatter(PQSR1[i][0], PQSR1[i][1])
#     plt.annotate(point1[i] + '({},{})'.format(PQSR1[i][0], PQSR1[i][1]), (PQSR1[i
    ][0], PQSR1[i][1]), c='r')
# plt.axis('off')
# plt.savefig('img1_method3_labels.jpeg')
# plt.show()
# plt.clf()
# pts = np.array([[462, 165, 1], [431, 730, 1], [1415, 482, 1], [1462, 810, 1], [165,
    69, 1], [98, 876, 1], [425, 883, 1]])
# # Computation
# H1 = H_matrix(pts)
# output1 = img_map(img1, np.linalg.inv(H1))
# plt.imshow(output1, cmap='gray')
# plt.axis('off')
# plt.savefig("img1_method3_result.jpeg")
# plt.show()
# plt.clf()

# Method 3 for image2
img2 = io.imread('hw3_Task1_Images/Images/Img2.jpeg')
PQSR2 = np.array([[368, 552], [362, 853], [621, 510], [642, 974], [309, 342], [291,
    896], [496, 1023]])
point2 = ["P", "Q", "S", "R", "A", "B", "C"]
# Plot
plt.imshow(img2)
for i in range(7):
    plt.scatter(PQSR2[i][0], PQSR2[i][1])
    plt.annotate(point2[i] + '({},{})'.format(PQSR2[i][0], PQSR2[i][1]), (PQSR2[i][0],
     PQSR2[i][1]), c='r')
```

```
plt.axis('off')
plt.savefig('img2_method3_labels.jpeg')
plt.show()
plt.clf()
pts = np.array([[368, 552, 1], [362, 853, 1], [621, 510, 1], [642, 974, 1], [309, 342,
     1], [291, 896, 1], [496, 1023, 1]])
# Computation
H2 = H_matrix(pts)
output2 = img_map(img2, np.linalg.inv(H2))
plt.imshow(output2, cmap='gray')
plt.axis('off')
plt.savefig("img2_method3_result.jpeg")
plt.show()
plt.clf()

# Method 3 for image3
point = ['P', 'Q', 'S', 'R', 'A', 'B', 'C']
img3 = io.imread('hw3_Task1_Images/Images/Img3.JPG')
PQSR3 = np.array([[2060, 700], [2092, 1483], [2666, 720], [2695, 1333], [689, 750],
    [741, 2088], [1774, 1621]])
plt.imshow(img3, cmap='gray')
for i in range(7):
    plt.scatter(PQSR3[i][0], PQSR3[i][1])
    plt.annotate(point[i] + '({},{})'.format(PQSR3[i][0], PQSR3[i][1]), (PQSR3[i][0],
    PQSR3[i][1]), c='r')
plt.axis('off')
plt.savefig('img3_method3_labels.jpeg')
plt.show()
plt.clf()
pts3 = np.array([[2060, 700, 1], [2092, 1483, 1], [2666, 720, 1], [2695, 1333, 1],
    [689, 750, 1], [741, 2088, 1], [1774, 1621, 1]])
H3 = H_matrix(pts3)
output3 = img_map(img3, np.linalg.inv(H3))
plt.imshow(output3, cmap='gray')
plt.axis('off')
plt.savefig("img3_method3_result.jpeg")
plt.show()
plt.clf()


# Method 3 for image4
img4 = io.imread('hw3_Task1_Images/Images/Img4.jpeg')
PQSR4 = np.array([[625, 622], [609, 1044], [778, 521], [764, 970], [396, 776], [378,
    1153], [497, 1094]])
point4 = ["P", "Q", "S", "R", "A", "B", "C"]
# Plot
plt.imshow(img4)
for i in range(7):
   plt.scatter(PQSR4[i][0], PQSR4[i][1])
   plt.annotate(point4[i] + '({},{})'.format(PQSR4[i][0], PQSR4[i][1]), (PQSR4[i][0],
    PQSR4[i][1]), c='r')
plt.axis('off')
plt.savefig('img4_method3_labels.jpeg')
plt.show()
plt.clf()
pts4 = np.array([[625, 622, 1], [609, 1044, 1], [778, 521, 1], [764, 970, 1], [396,
    776, 1], [378, 1153, 1], [497, 1094, 1]])
H4 = H_matrix(pts4)
output4 = img_map(img4, np.linalg.pinv(H4))
plt.imshow(output4, cmap='gray')
plt.axis('off')
plt.savefig("img4_method3_result.jpeg")
plt.show()
plt.clf()
```

```
1230
1232 # Method 3 for image5
     img5 = io.imread('hw3_Task1_Images/Images/img5.jpeg')
1234 PQSR5 = np.array([[573, 302], [572, 386], [692, 296], [692, 384], [110, 230], [106,
         316], [184, 227], [181, 313]])
     point5 = ["P", "Q", "S", "R", "A", "B", "C", "D"]
1236 plt.imshow(img5)
     for i in range(len(point5)):
1238    plt.scatter(PQSR5[i][0], PQSR5[i][1])
        plt.annotate(point5[i] + '({},{})'.format(PQSR5[i][0], PQSR5[i][1]), (PQSR5[i][0],
         PQSR5[i][1]), c='r')
1240 plt.axis('off')
     plt.savefig('img5_method3_labels.jpeg')
1242 plt.show()
     plt.clf()
1244 pts = np.array([[573, 302, 1], [572, 386, 1], [692, 296, 1], [692, 384, 1], [110, 230,
          1], [106, 316, 1], [184, 227, 1], [181, 313, 1]])
     H = H_matrix(pts)
1246 output5 = img_map(img5, np.linalg.inv(H))
     plt.imshow(output5, cmap='gray')
1248 plt.axis('off')
     plt.savefig("img5_method3_result.jpeg")
1250 plt.show()
     plt.clf()
```

method3.py