ECE 661 Computer Vision: HW5

Qiuchen Zhai
qzhai@purdue.edu

October 6, 2020

1 Theory Question

1.1 How to differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem

The least-squares methods can tolerate noise of a few pixels in the localization of $\vec{x}'$ vis-a-vis to true location. However, it's very likely that the set of correspondences are false. While calculating the homography, the merely noisy $(\vec{x}, \vec{x}')$ correspondences constitutes the inliers, and the false $(\vec{x}, \vec{x}')$ correspondences constitutes the outliers.

When using RANSAC (Random Sample Consensus) for solving the homography estimation problem, the randomly-selected least amount of data would be used to construct an estimate and ascertain the extent of support that the rest of the data provides to the estimate. The estimate is accepted only if the support exceeds a threshold. Then the supporting data constitutes the inliers if an estimate made in this manner is accepted. The rest of the data then constitutes the outliers.

For example, the problem of linear regression estimates a linear relationship between an input random variable x and an output random variable y. With the goal of estimating an affine transformation between the two variables $y = ax + b$, at least two number of points are need to form the line estimate. RANSAC will randomly pick two data points and measure the support the rest of the data provides for the line constructed on the basis of the two points. The support is measured by the number of points that lie within a distance threshold δ of the line. Then the points outside the distance threshold constitute the outlier set and the points within the distance threshold constitute the inlier set.

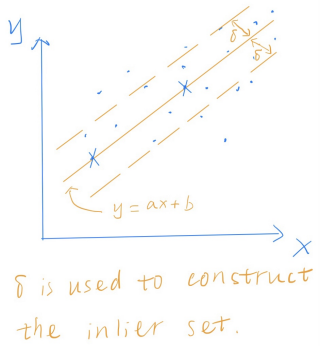

Figure 1: Linear Regression Example

1.2 How the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a reasonably fast and numerically stable method

The solution of Gauss-Newton method is given by

$$\vec{\delta}_p = (J_{\vec{f}}^T J_{\vec{f}})^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{1}$$

Then we can re-write the solution as

$$(J_{\vec{f}}^T J_{\vec{f}})\vec{\delta}_p = J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{2}$$

Note that the solution returned by GN would be same as by the gradient descent method (GD) if $J_{\vec{f}}^T J_{\vec{f}}$ were purely diagonal. So we heuristically extend the equation by the addition of damping component

$$(J_{\vec{f}}^T J_{\vec{f}} + \mu I)\vec{\delta}_p = J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{3}$$

With the damping coefficient μ, the solution returned by the Augmented Normal Equation would be close to the solution returned by GD if μ is much larger than the diagonal elements of $J_{\vec{f}}^T J_{\vec{f}}$. And the solution would be the same as GN if $\mu = 0$. The above equation could be re-written as

$$\vec{\delta}_p = (J_{\vec{f}}^T J_{\vec{f}} + \mu I)^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{4}$$

The method is referred as the Levenberg-Marquardt Method (LM). It's clear to see that LM behaves like GD if the starting point is far from the minimum and acts like GN as the solution point approaches the minimum. Therefore, LM gives a reasonably fast and numerically stable method by combing the best of GD and the best of GN.

2 Description of programming task

2.1 Interest Points and Descriptors

Given the five or more images,

1. First, SIFT algorithm from OpenCV library is used for extracting interest points between the pair of images.
2. Then NCC metric is used for finding the initial set of correspondences.

2.2 Outlier Rejection Using the RANSAC algorithm

The Random Sample Consensus Algorithm is used for rejecting the outlier which is constituted by the false pairings. The implementation of RANSAC algorithm involves the following steps:

1. First, we need to set up the following parameters
   - δ: the decision threshold to construct the inlier set. Ususally, it's assumed that the noise-induced displacement for the true location of a pixel would be modeled by the Gaussian $g(\Delta x, \Delta y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(\Delta x)^2+(\Delta y)^2}{2\sigma^2}}$. So $\delta = 3\sigma$ that captures 90% of the inliers where σ is set to a small number between 0.5 and 2. In this experiment, the parameter is set as $\delta = 8$.
   - ϵ: the probability that a correspdondence is an outlier. Assuming that roughly 10% of the correspdondences are false, the parameter is set as $\epsilon = 0.1$

- n: the smallest number of samples that at least one trial in this minimal set of the correspondences to construction the homography estimate does not include any outliers. In this experiment, the parameter is set as $n = 6$.
- p: the probability that at least one of the trials will be free of outliers in the calculation of homography estimate. In the experiment, the parameter is set as $p = 0.99$.
- N: the number of trials to construct which is determined by

$$N = \frac{ln(1-p)}{ln[1-(1-\epsilon)^n]} \tag{5}$$

- n_{total}: the total number of available correspondeces.
- M: A minimum value for the size of inlier set so that we can accept it. The parameter is set as $M = (1-\epsilon) \cdot n_{total}$.

2. Then, n pairs of correspondences of interest points would be randomly selected and compute the homography estimate H by the Linear-Least Squares method. By applying the homography estimate to all the interest points, the correspondences within the decision threshold δ constitute the inlier and the other correspondences constitutes the outlier.
3. Repeat the above step for N times and the trial with maximum number of inliers would be retained for homography estimation and refinement.

2.3 Homography Estimation Using Least-Squares method

Since only the ratio is important, we consider the homography matrix with $h_{33} = 1$

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{pmatrix} \tag{6}$$

Given the image pair $(\overrightarrow{\mathbf{X}}_i, \overrightarrow{\mathbf{X}}_i^{'})$ where $\overrightarrow{\mathbf{X}}_i = \begin{pmatrix} x_i \\ y_i \\ w_i \end{pmatrix}$, $\overrightarrow{\mathbf{X}}_i^{'} = \begin{pmatrix} x_i^{'} \\ y_i^{'} \\ w_i^{'} \end{pmatrix}$, the homography could be computed by $\mathbf{A}\overrightarrow{\mathbf{h}} = \overrightarrow{\mathbf{b}}$ that

$$\begin{pmatrix} 0 & 0 & 0 & -w_1^{'}x_1 & -w_1^{'}y_1 & -w_1^{'}w_1 & y_1^{'}x_1 & y_1^{'}y_1 \\ w_1^{'}x_1 & w_1^{'}y_1 & w^{'}w_1 & 0 & 0 & 0 & -x_1^{'}x_1 & -x_1^{'}y_1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & -w_i^{'}x_i & -w_i^{'}y_i & -w_i^{'}w_i & y_i^{'}x_i & y_i^{'}y_i \\ w_i^{'}x_i & w_i^{'}y_i & w_i^{'}w_i & 0 & 0 & 0 & -x_i^{'}x_i & -x_i^{'}y_i \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & -w_n^{'}x_n & -w_n^{'}y_n & -w_n^{'}w_n & y_n^{'}x_n & y_n^{'}y_n \\ w_n^{'}x_n & w_n^{'}y_n & w_n^{'}w_n & 0 & 0 & 0 & -x_n^{'}x_n & -x_n^{'}y_n \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix} = \begin{pmatrix} -y_1^{'}w_1 \\ x_1^{'}w_1 \\ \cdots \\ -y_i^{'}w_i \\ x_i^{'}w_i \\ \cdots \\ -y_n^{'}w_n \\ x_n^{'}w_n \end{pmatrix} \tag{7}$$

The solution is then given by

$$\min \| \overrightarrow{\mathbf{b}} - \mathbf{A}\overrightarrow{\mathbf{h}} \| \tag{8}$$

The Least-squares solution gives

$$\overrightarrow{\mathbf{h}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{9}$$

2.4 Homography Refinement Using the LM Algorithm

The Levenburg-Marquardt combines the best of the Gradient-Descent method and the best of Gradient-Newtons Method and gives us a stable and fast method. The LM equation is given by

$$(J_{\vec{f}}^T J_{\vec{f}} + \mu I)\vec{\delta}_p = J_{\vec{f}}^T \vec{\epsilon}(\vec{p}) \tag{10}$$

where $f(\vec{p}) = [f_1^1(\vec{p}), f_2^1(\vec{p}), f_1^2(\vec{p}), f^2(\vec{p}), \cdots, f_1^N(\vec{p}), f_2^N(\vec{p})]^T$, $\vec{p} = [h_{11}, h_{12}, h13, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}]^T$, $\overrightarrow{\delta}_p$ denotes the change of $\overrightarrow{p}$ between neighboring iterations, μ is the damping coefficient, and the functions of vector $\vec{p}$ are given by

$$f_1^i(\vec{p}) = \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{11}$$

$$f_2^i(\vec{p}) = \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}} \tag{12}$$

The Jacobian matrix is given by

$$J_{\vec{f}} = \begin{pmatrix} \frac{\partial f_1}{\partial p_1} & \cdots & \frac{\partial f_1}{\partial p_n} \\ \vdots & \vdots & vdots \\ \frac{\partial f_m}{\partial p_1} & \cdots & \frac{\partial f_m}{\partial p_n} \end{pmatrix} \tag{13}$$

where m denotes the number of prediction and n denotes the number of parameter. The error of the estimation if given by

$$\vec{\epsilon}_{\vec{p}} = \vec{X} - \vec{f}_{\vec{p}} \tag{14}$$

To refine the linear matrix using LM method, we repeat the following steps until the result converges:

1. Initialization
   First, we need to initialize the damping coefficient $\mu_0 = \tau \times \max\{diag(J_{\vec{f}}^T J_{\vec{f}})\}$, where τ is a constant number between 0 and 1. In the experiment, we initialize $\tau = 0.5$.

2. Compute the $\vec{\delta}_p$
   Then we compute the $\vec{\delta}_p$ by

$$\vec{\delta}_{p_k} = (J_{\vec{f}}^T J_{\vec{f}} + \mu I)^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p_k}) \tag{15}$$

3. Compute the ratio ρ
   In order to check the sign of change of the cost function (the cost function for $\overrightarrow{p_k}$ is represented as $C(\overrightarrow{p_k}) = \|\overrightarrow{X} - \vec{f}_{\vec{p_k}}\|^2$), we use the following formula to compute the ratio:

$$\rho = \frac{C(\vec{p_k}) - C(\vec{p_{k+1}})}{\vec{\delta}_{p_k}^T J_{\vec{f}}^T \vec{\epsilon}_{\vec{p_k}} + \vec{\delta}_{p_k}^T \mu_k I \vec{\delta}_{p_k}} \tag{16}$$

4. Update the damping coefficient
   The value of ρ is used to calculate the damping coefficient for the next iteration

$$\mu \leftarrow \mu \times \max\{\frac{1}{3}, 1 - (2\rho - 1)^3\} \tag{17}$$

5. Obtain the refined nonliear homography matrix.

2.5 Image Mosaicing

Five images from different view are taken as input. The plane of the third image which is taken from the middle view is taken as the range plane. And other images are projected onto the range plane to be stitches together. We can compute the homography matrix between the image pairs and obtain $H_{12}, H_{23}, H_{34}, H_{45}$. Then the homography mappings from domain plane to range plane are given by

$$H_{13} = H_{12} \times H_{23} \tag{18}$$

$$H_{33} = I(3) \tag{19}$$

$$H_{53} = H_{34}^{-1} \times H_{45}^{-1} \tag{20}$$

3 Implementation results

3.1 Input images

Figure 2: Input image 1

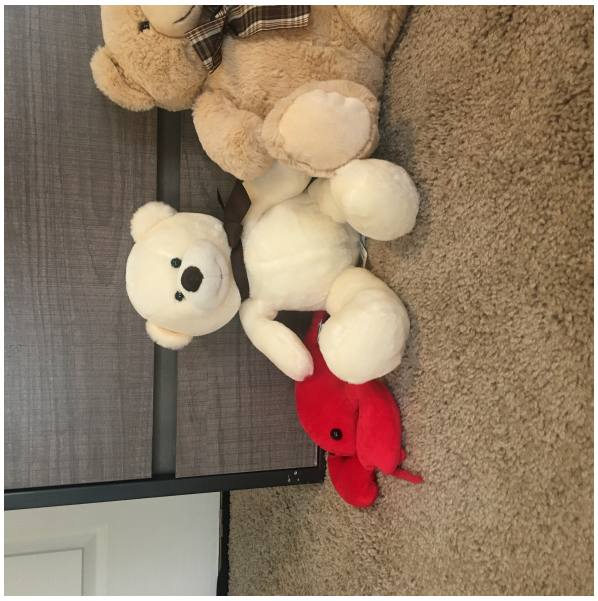

Figure 3: Input image 2

Figure 4: Input image 3

Figure 5: Input image 4

Figure 6: Input image 5

3.2 Extracted correspondences between the sets of adjacent images

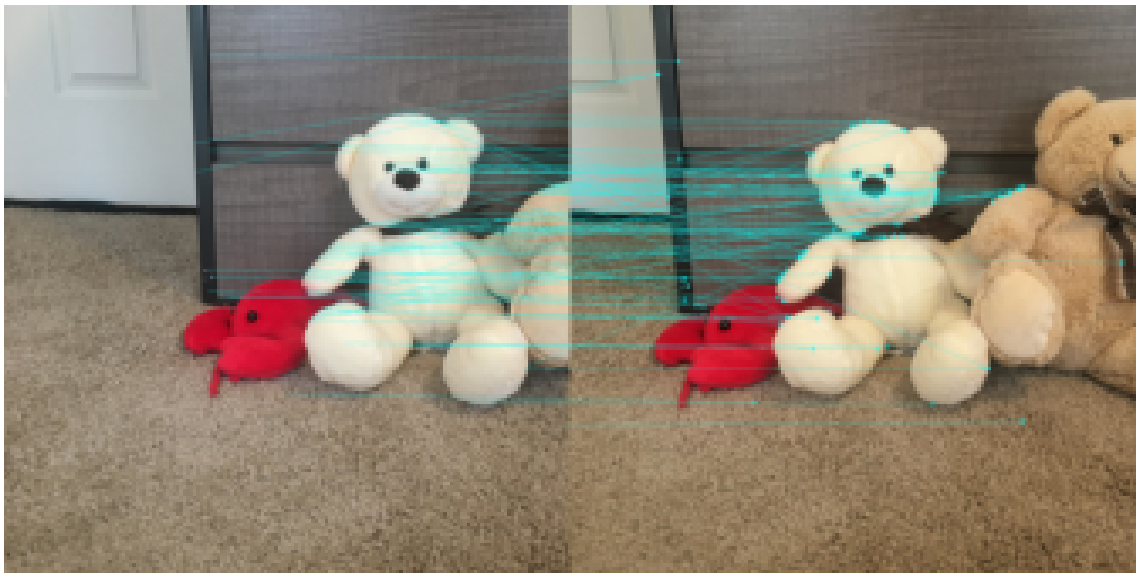

Figure 7: Interest points correspondences between img1 and img2

Figure 8: Interest points correspondences between img2 and img3

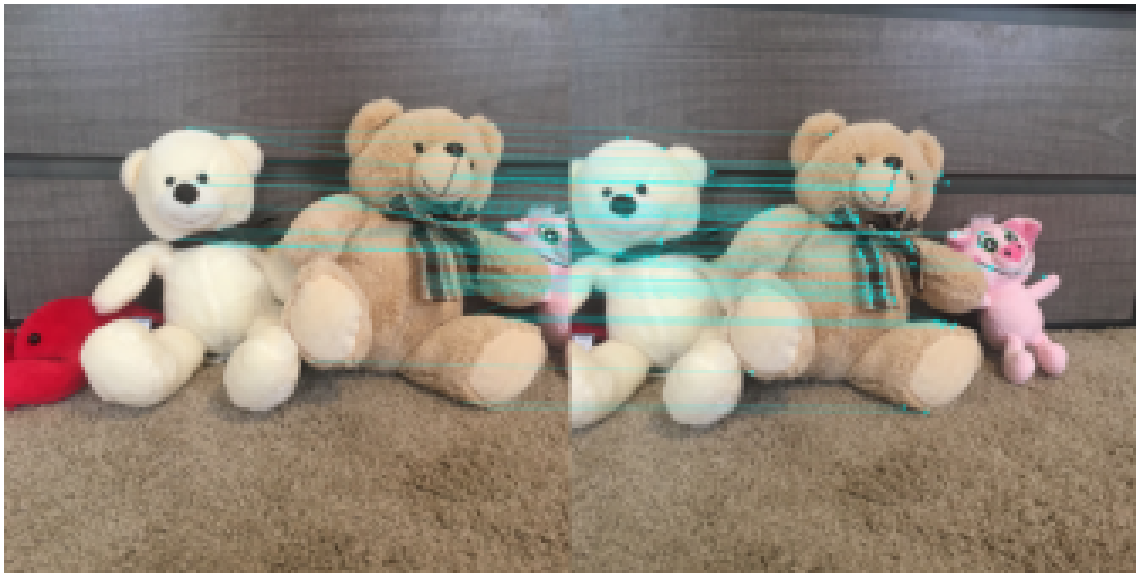

Figure 9: Interest points correspondences between img3 and img4

Figure 10: Interest points correspondences between img4 and img5

3.3 The sets of inlier and outlier

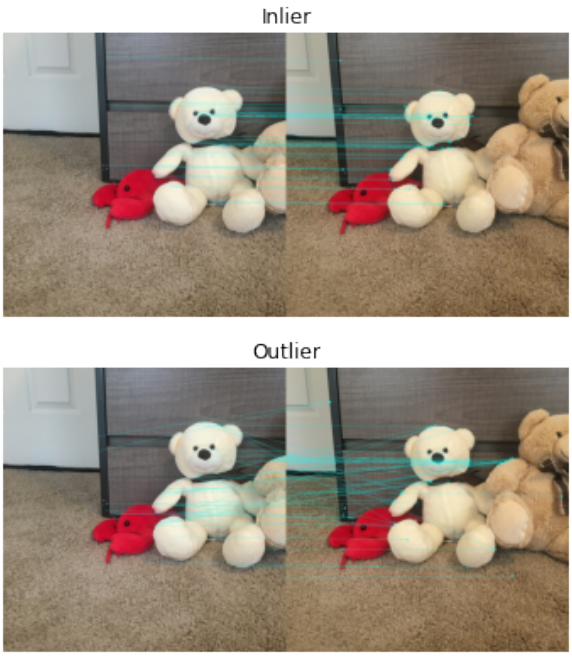

Figure 11: The inlier set and outlier set of first pair

Figure 12: The inlier set and outlier set of second pair

Figure 13: The inlier set and outlier set of third pair

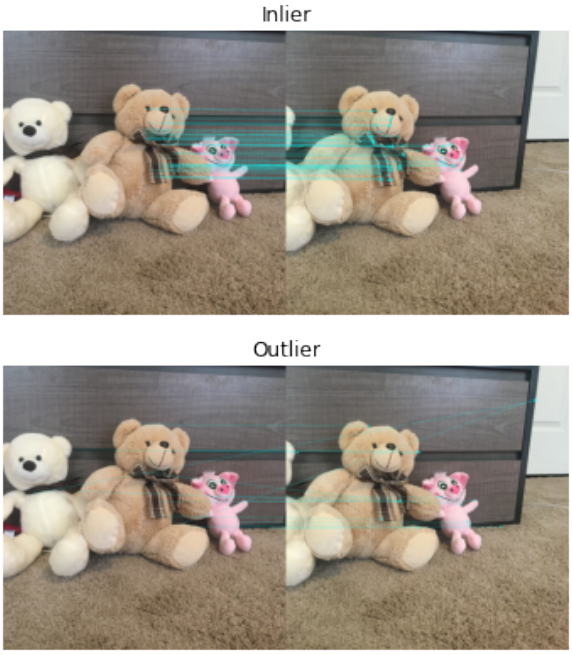

Figure 14: The inlier set and outlier set of fourth pair

3.4 Final output panoramic view after stitching input images

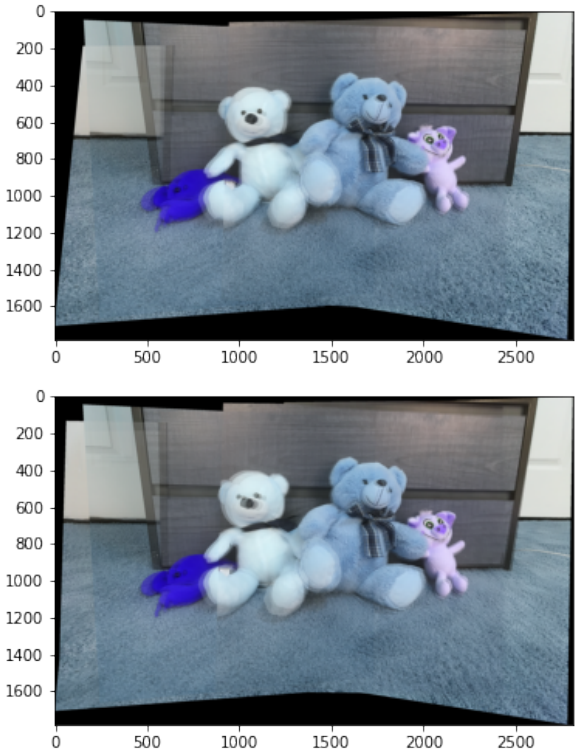

Figure 15: panoramic views of image before and after homography refinement

4 Source Code

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


# Read images
img1 = io.imread('1.jpeg')
img2 = io.imread('2.jpeg')
img2 = cv2.resize(img2, (img1.shape[1], img1.shape[0]), interpolation=cv2.INTER_AREA)
img3 = io.imread('3.jpeg')
img3 = cv2.resize(img3, (img1.shape[1], img1.shape[0]), interpolation=cv2.INTER_AREA)
img4 = io.imread('4.jpeg')
img4 = cv2.resize(img4, (img1.shape[1], img1.shape[0]), interpolation=cv2.INTER_AREA)
img5 = io.imread('5.jpeg')
img5 = cv2.resize(img5, (img1.shape[1], img1.shape[0]), interpolation=cv2.INTER_AREA)
# Converts image to gray images
gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
gray3 = cv2.cvtColor(img3, cv2.COLOR_BGR2GRAY)
gray4 = cv2.cvtColor(img4, cv2.COLOR_BGR2GRAY)
gray5 = cv2.cvtColor(img5, cv2.COLOR_BGR2GRAY)


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

```python
# Extract SIFT feature
num_features = 2000
sift = cv2.xfeatures2d.SIFT_create(nfeatures=num_features)
kp1, des1 = sift.detectAndCompute(gray1, None)
kp2, des2 = sift.detectAndCompute(gray2, None)
kp3, des3 = sift.detectAndCompute(gray3, None)
kp4, des4 = sift.detectAndCompute(gray4, None)
kp5, des5 = sift.detectAndCompute(gray5, None)


def ncc_metric(img1, kp1, img2, kpts, k):

    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    # Initialization
    h, w = img1.shape
    n = int(k / 2)
    # Neighborhood of kp1
    img1_padded = np.zeros([h + 2 * n, w + 2 * n])
    img1_padded[n: n + h, n: n + w] = img1
    img2_padded = np.zeros([h + 2 * n, w + 2 * n])
    img2_padded[n: n + h, n: n + w] = img2
    # find the ncc between kp1 and kp2's in corner list of img2 and return the
    correspondence
    neighbor1 = img1_padded[int(kp1[1]): int(kp1[1]) + 2 * n, int(kp1[0]): int(kp1[0])
     + 2 * n]
    max_ncc = -1e6
    # index = int(len(corner2)-1)
    for i in range(len(kpts)):
        kp2 = kpts[i].pt
        neighbor2 = img2_padded[int(kp2[1]): int(kp2[1]) + 2 * n, int(kp2[0]): int(kp2
    [0]) + 2 * n]
        sum1 = np.sum(neighbor1 - np.mean(neighbor1) ** 2)
        sum2 = np.sum(neighbor2 - np.mean(neighbor2) ** 2)
        ncc = np.sum((neighbor1 - np.mean(neighbor1)) * (neighbor2 - np.mean(neighbor2
    ))) / np.sqrt(sum1 * sum2)
        if ncc > max_ncc:
            max_ncc = ncc
            index = i

    return index, max_ncc


def match_pt(img1, kp1, img2, kp2, num_best):
    """
    Use SSD to build points correspondences.
    """
    pts = []
    match_pts = []
    match_dist = []
    for i in range(len(kp1)):
        pt1 = kp1[i].pt
        pts.append(pt1)
        index, max_ncc = ncc_metric(img1, pt1, img2, kp2, k=21)
        match_pts.append(kp2[index].pt)
        match_dist.append(max_ncc)

    # Sort
    pts = [x for _, x in sorted(zip(match_dist, pts))]
    pts = np.array(pts)[0:num_best, :]
    match_pts = [y for _, y in sorted(zip(match_dist, match_pts))]
    match_pts = np.array(match_pts)[0:num_best, :]
```

```
    match_dist = sorted(match_dist)
    match_dist = np.array(match_dist)[0:num_best]

    # # Plot
    # img = np.hstack((img1, img2))
    # w, h, d = img1.shape
    # for i in range(num_best):
    #     pt1 = [int(pts[i][0]), int(pts[i][1])]
    #     pt2 = [int(match_pts[i][0] + h), int(match_pts[i][1])]
    #     cv2.line(img, tuple(pt1), tuple(pt2), (0, 255, 255), 1)
    #     cv2.circle(img, tuple(pt1), 4, (0, 255, 255), 2)
    #     cv2.circle(img, tuple(pt2), 4, (0, 255, 255), 2)
    # plt.imshow(img, cmap='gray')
    # plt.axis('off')
    # plt.show()
    # plt.clf()

    return pts, match_pts, match_dist


# Build points correspondences
pts1, match_pts12, match_dist12 = match_pt(img1, kp1, img2, kp2, num_best=200)
pts2, match_pts23, match_dist23 = match_pt(img2, kp2, img3, kp3, num_best=200)
pts3, match_pts34, match_dist34 = match_pt(img3, kp3, img4, kp4, num_best=200)
pts4, match_pts45, match_dist45 = match_pt(img4, kp4, img5, kp5, num_best=200)


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

# RANSAC


def random_miniset(pts1, pts2, num_pts):

    random_selection = np.random.permutation(len(pts1))
    selection1 = []
    selection2 = []
    for i in range(num_pts):
        selection1.append(pts1[random_selection[i]])
        selection2.append(pts2[random_selection[i]])

    return selection1, selection2


def compute_homography(pts1, pts2):

    A = np.zeros((2 * len(pts1), 8))
    b = np.zeros((2 * len(pts1), 1))

    for i in range(len(pts1)):
        pt1 = np.array([pts1[i][0], pts1[i][1], 1], dtype=float)
        pt2 = np.array([pts2[i][0], pts2[i][1], 1], dtype=float)
        A[2 * i] = [pt1[0], pt1[1], 1, 0, 0, 0, -pt1[0]*pt2[0], -pt1[1]*pt2[0]]
        A[2 * i + 1] = [0, 0, 0, pt1[0], pt1[1], 1, -pt1[0]*pt2[1], -pt1[1]*pt2[1]]
        b[2 * i] = pt2[0]
        b[2 * i + 1] = pt2[1]

    h = np.dot(np.linalg.pinv(A), b)
    homo_mat = np.append(h, 1)
    H = homo_mat.reshape((3, 3))

    return H
```

```
def get_liers(pts1, pts2, map_pts1, delta):

    inlier_pts1 = []
    inlier_pts2 = []
    outlier_pts1 = []
    outlier_pts2 = []

    for i in range(len(pts2)):
        if np.sqrt((pts2[i][0] - map_pts1[i][0])**2 + (pts2[i][1] - map_pts1[i][1])
    **2) < delta:
            inlier_pts1.append(pts1[i])
            inlier_pts2.append(pts2[i])
        else:
            outlier_pts1.append(pts1[i])
            outlier_pts2.append(pts2[i])

    return inlier_pts1, inlier_pts2, outlier_pts1, outlier_pts2


def apply_H(H, pts):

    pts = np.append(pts, np.ones([len(pts), 1]), 1)
    mapped_pts = np.zeros(pts.shape, dtype=float)
    for i in range(len(pts)):
        mapped_pts[i] = np.dot(H, pts[i]) / np.dot(H, pts[i])[2]

    return mapped_pts


def ransac_rejection(pts1, pts2, n, epsilon, p, delta):

    N = int( np.log(1-p) / np.log( 1 - (1 - epsilon)**n ) )
    n_total = len(pts1)
    M = int( (1 - epsilon) * n_total)
    for i in range(N):
        selected_pts1, selected_pts2 = random_miniset(pts1, pts2, num_pts=6)
        H = compute_homography(selected_pts1, selected_pts2)
        map_pts = apply_H(H, pts1)
        inlier_pt1, inlier_pt2, outlier_pt1, outlier_pt2 = get_liers(pts1, pts2,
    map_pts, delta)

    return inlier_pt1, inlier_pt2, outlier_pt1, outlier_pt2


def show_liers(img1, img2, inlier_pt1, inlier_pt2, outlier_pt1, outlier_pt2):

    img = np.hstack((img1, img2))
    w, h, d = img1.shape

    for i in range(len(inlier_pt1)):
        pt1 = [int(inlier_pt1[i][0]), int(inlier_pt1[i][1])]
        pt2 = [int(inlier_pt2[i][0] + h), int(inlier_pt2[i][1])]
        cv2.line(img, tuple(pt1), tuple(pt2), (0, 255, 255), 1)
        cv2.circle(img, tuple(pt2), 4, (0, 255, 255), 2)
        cv2.circle(img, tuple(pt2), 4, (0, 255, 255), 2)
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.title('Inlier')
    plt.show()
    plt.clf()

    img = np.hstack((img1, img2))
    for i in range(len(outlier_pt1)):
```

```
        pt1 = [int(outlier_pt1[i][0]), int(outlier_pt1[i][1])]
        pt2 = [int(outlier_pt2[i][0] + h), int(outlier_pt2[i][1])]
        cv2.line(img, tuple(pt1), tuple(pt2), (0, 255, 255), 1)
        cv2.circle(img, tuple(pt2), 4, (0, 255, 255), 2)
        cv2.circle(img, tuple(pt2), 4, (0, 255, 255), 2)
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.title('Outlier')
    plt.show()
    plt.clf()


# RANSAC paramters
epsilon = 0.1
delta = 8
p = 0.99
n = 6


# the first pair
in1, in12, out1, out12 = ransac_rejection(pts1, match_pts12, n, epsilon, p, delta)
show_liers(img1, img2, in1, in12, out1, out12)
# the second pair
in2, in23, out2, out23 = ransac_rejection(pts2, match_pts23, n, epsilon, p,  delta)
show_liers(img2, img3, in2, in23, out2, out23)
# the third pair
in3, in34, out3, out34 = ransac_rejection(pts3, match_pts34, n, epsilon, p,  delta)
show_liers(img3, img4, in3, in34, out3, out34)
# the fourth pair
in4, in45, out4, out45 = ransac_rejection(pts4, match_pts45, n, epsilon, p,  delta)
show_liers(img4, img5, in4, in45, out4, out45)


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

# Linear Least Square


def homography_estimate(pts1, pts2):

    if len(pts1) % 2 == 1:
        pts1.append([0, 0])
        pts2.append([0, 0])
    pts1 = np.array(pts1, dtype=float)
    pts2 = np.array(pts2, dtype=float)
    A = np.zeros((2 * len(pts1), 8))
    b = np.zeros((2 * len(pts1), 1))

    for i in range(len(pts1)):
        pt1 = np.array([pts1[i][0], pts1[i][1], 1], dtype=float)
        pt2 = np.array([pts2[i][0], pts2[i][1], 1], dtype=float)
        A[2 * i] = [0, 0, 0, -pt2[2] * pt1[0], -pt2[2] * pt1[1], -pt2[2] * pt1[2], pt2
    [1] * pt1[0], pt2[1] * pt1[1]]
        A[2 * i + 1] = [pt2[2] * pt1[0], pt2[2] * pt1[1], pt2[2] * pt1[2], 0, 0, 0, -
    pt2[0] * pt1[0], -pt2[0] * pt1[1]]
        b[2 * i] = -pt2[1] * pt1[2]
        b[2 * i + 1] = pt2[0] * pt1[2]

    h = np.dot(np.linalg.pinv(A), b)
    homo_mat = np.append(h, 1)
    H = homo_mat.reshape((3, 3))

    return H
```

```
H_12 = homography_estimate(in1, in12)
H_23 = homography_estimate(in2, in23)
H_33 = np.identity(3)
H_34 = homography_estimate(in3, in34)
H_45 = homography_estimate(in4, in45)

H_43 = homography_estimate(in34, in3)
H_54 = homography_estimate(in45, in4)
H_13 = np.dot(H_12, H_23) / np.dot(H_12, H_23)[2][2]
H_53 = np.dot(H_54, H_43) / np.dot(H_54, H_43)[2][2]


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

# Create panoramic Image


def compute_boundary(img, H_mat):

    b = np.zeros((4, 3), dtype=float)
    b[0] = np.dot(H_mat, np.array([0, 0, 1])) / np.dot(H_mat, np.array([0, 0, 1]))[2]
    b[1] = np.dot(H_mat, np.array([img.shape[0], 0, 1])) / np.dot(H_mat, np.array([img
    .shape[0], 0, 1]))[2]
    b[2] = np.dot(H_mat, np.array([0, img.shape[1], 1])) / np.dot(H_mat, np.array([0,
    img.shape[1], 1]))[2]
    b[3] = np.dot(H_mat, np.array([img.shape[0], img.shape[1], 1])) / np.dot(H_mat, np
    .array([img.shape[0], img.shape[1], 1]))[2]

    return b


def init_pano_img(imgs, H_matrices):

    b1 = compute_boundary(imgs[0], H_matrices[0])[:, 0:2]
    b2 = compute_boundary(imgs[1], H_matrices[1])[:, 0:2]
    b3 = compute_boundary(imgs[2], H_matrices[2])[:, 0:2]
    b4 = compute_boundary(imgs[3], H_matrices[3])[:, 0:2]
    b5 = compute_boundary(imgs[4], H_matrices[4])[:, 0:2]
    x_min, y_min = np.amin(np.amin([b1, b2, b3, b4, b5], 0), 0)
    x_max, y_max = np.amax(np.amax([b1, b2, b3, b4, b5], 0), 0)
    x_scale = np.int(np.ceil(x_max)) - np.int(np.floor(x_min))
    y_scale = np.int(np.ceil(y_max)) - np.int(np.floor(y_min))
    output = np.zeros((x_scale, y_scale, 3))

    return output, x_min, y_min


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(panoramic_img, img, h_mat, x_min, y_min):

    x_scale, y_scale, d_scale = np.shape(panoramic_img)
    # # Determine the coordinates of domain plane
    # m, n, d = np.shape(img)
    # # 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
    s = 1
    panoramic_img = cv2.resize(panoramic_img, None, fx=1/s, fy=1/s)
    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):
                panoramic_img[i][j] = compute_pixel_val(img, [loc0, loc1])

    return panoramic_img.astype(np.uint8)


imgs = [img1, img2, img3, img4, img5]
H_matrices = [H_13, H_23, H_33, H_43, H_53]
pano_img, x_min, y_min = init_pano_img(imgs, H_matrices)
pano_img = img_map(pano_img, img1, H_13, x_min, y_min)
pano_img = img_map(pano_img, img5, H_53, x_min, y_min)
pano_img = img_map(pano_img, img2, H_23, x_min, y_min)
pano_img = img_map(pano_img, img4, H_43, x_min, y_min)
pano_img = img_map(pano_img, img3, H_33, x_min, y_min)
```

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

# Levenberg Marquardt Method


def LM(pts1, pts2, H_linear):

    pts1 = np.asarray(pts1)
    pts2 = np.asarray(pts2)

    def func(h):
        cost = []
        for i in range(len(pts1)):
            x = pts1[i][0] * h[0] + pts1[i][1] * h[1] + h[2] / (pts1[i][0] * h[6] +
    pts1[i][7] * h[4] + 1)
            y = pts1[i][0] * h[3] + pts1[i][4] * h[1] + h[5] / (pts1[i][0] * h[6] +
    pts1[i][7] * h[4] + 1)
            cost.append(pts2[i][0] - x)
            cost.append(pts2[i][1] - y)
        return np.asarray(cost)

    sol = least_squares(func, H_linear.squeeze(), method='lm')
    H_nonlinear = sol.x
    H_nonlinear.append(H_nonlinear, 1)

    return H_nonlinear.reshape((3, 3))


H_12_nonliear = LM(in1, in12, H_12)
H_23_nonliear = LM(in2, in23, H_23)
H_33_nonliear = np.identity(3)
H_34_nonliear = LM(in3, in34, H_34)
H_45_nonliear = LM(in4, in45, H_45)

H_43_nonliear = np.linalg.inv(H_34_nonliear)
H_54_nonliear = np.linalg.inv(H_45_nonliear)
H_13_nonliear = np.dot(H_12_nonliear, H_23_nonliear) / np.dot(H_12_nonliear,
    H_23_nonliear)[2][2]
H_53_nonliear = np.dot(H_54_nonliear, H_43_nonliear) / np.dot(H_54_nonliear,
    H_43_nonliear)[2][2]

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

# Create panoramic Image

imgs = [img1, img2, img3, img4, img5]
H_matrices = [H_13_nonliear, H_23_nonliear, H_33_nonliear, H_43_nonliear,
    H_53_nonliear]
pano_img, x_min, y_min = init_pano_img(imgs, H_matrices)
pano_img = img_map(pano_img, img1, H_13_nonliear, x_min, y_min)
pano_img = img_map(pano_img, img5, H_53_nonliear, x_min, y_min)
pano_img = img_map(pano_img, img2, H_23_nonliear, x_min, y_min)
pano_img = img_map(pano_img, img4, H_43_nonliear, x_min, y_min)
pano_img = img_map(pano_img, img3, H_33_nonliear, x_min, y_min)
```

hw5.py