ECE661 Computer Vision HW5

Chengjun Guo
guo456@purdue.edu

12 October 2022

1 Theory question

1.1 Question1

Conceptually speaking, how do we differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

We randomly select several pairs of points in each trial. With these pairs of points we can find the homography H. With this given H, we calculate the error of the whole set pair of points. With the threshold δ, we define the pairs with error less than the threshold to be inliers. The homography with most inliers would be the expected homography. The outliers are the pairs with higher error than threshold δ.

1.2 Question2

As you will see in Lecture 13, the Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method but it can be numerically unstable. Explain in your own words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and numerically stable at the same time.

The Gradient-Descent method is slow because it is approaching the minimum with smaller and smaller step size. Theoretically, the minimum can never be reached. The Gauss-Newton method is numerically unstable because if the Jacobian is rank deficient, the J^TJ will be singular. When the solution approach the minimum, LM acts like GN. When the solution is far from the minimum, LM acts like GD. It is because the update of the damping coefficient μ is based

on the equation:

$$\mu_{k+1} = \mu_k \cdot max\{\frac{1}{3}, 1 - (2\rho_{k+1}^{LM} - 1)^3\}$$

where

$$\rho_{k+1}^{LM} = \frac{C(\overrightarrow{p}_k) - C(\overrightarrow{p}_{k+1})}{\overrightarrow{\delta}_p^T J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p}_k) + \overrightarrow{\delta}_p^T \mu_k I \overrightarrow{\delta}_p}$$

. The value of damping coefficient calculated using this formulation steers the LM in the direction of GD if max function choose the second argument or in the direction of GN if max function choose the first argument. With the damping coefficient, it is numerically stable and with the max function, it will act like GN when approaching minimum.

2 Logic

The code used are mostly the same for both tasks. The only difference are the image names. The coefficients are the same too.

The first main function is calculate_in_out. It is calling sift_match to find the corresponding points and it calls RANSAC function to get the Homography and the inliers. Then it call visualize_in_out to generate the inlier and outlier images.

The second main function is panorama. It first generate the H between the images. Then it use the H to calculate the coordinates in the canvas. Then it use different H to get the point value in the panorama.

The function RANSAC use the corresponding points to generate the H and inlier sets. The homography_calculate is the function used in the previous homework. find_inliers function is to loop through all the points to find the inliers with given threshold and H. The sift_match is the function used in the previous homework to generate the match points and the correspondence image.

2.1 Algorithms and implementation

2.1.1 RANSAC

We randomly select n pairs of points in each trial of N trials. With these n pairs of points we can find the homography H. With this given H, we calculate the error of the whole set pair of points. With the threshold δ, we define the pairs with error less than the threshold to be inliers. The homography with most inliers would be the expected homography.

The first step is to set up the coefficients. $\delta = 3$ is the decision of inlier threshold of 3 pixels. $p = 0.99$ means probability that at least one trial has no outliers.

$\epsilon = 0.25$ means 25% of outliers. $n = 10$ means we select 10 pairs to form a H in each trial. $N = \frac{\ln 1-p}{\ln 1-(1-\epsilon)^n}$ to be the trial number. $M = n_{total}(1-\epsilon)$ means the minimum size of the inlier set.

The second step is to form a for loop. For each trial we will find the homography by Linear Least Squares (in the next section) with the selected points. Then we generate calculated X' by $X' = HX$. The difference between the calcualted X' and the X' in the pairs would be $(\Delta x, \Delta y)$. It will be considered as inlier if $(\Delta x^2 + \Delta y^2) \leq \delta^2$.

Then we use the largest inlier set to compute the final homography by Linear Least Squares(in the next section). Then the Homography would be refined by non-linear Least Squares(in section 2.1.3) if the mode is set as LM.

2.1.2 Linear Least Squares

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

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1}$$

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'.
For each pair of points, we can have 2 equations. With n pairs of points, we will then have a 2n by 8 matrix A and a 8 by 1 vector h and a 2n by 1 vector b.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_nx'_n & -y_nx'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -x_ny'_n & -y_ny'_n \end{bmatrix} \times \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ \vdots \\ x'_n \\ y'_n \end{bmatrix} \tag{4}$$

For a least square approximation, $h = (A^TA)^{-1}A^Tb$. This is also known as pseudo inverse. With the vector h, we can form the 3 by 3 matrix by adding 1.

2.1.3 non-linear Least Squares

We first add a damping coefficient to make the GN method numerically stable. In this case, we will have a similar equation for LM:

$$(J_{\overrightarrow{f}}^T J_{\overrightarrow{f}} + \mu I)\overrightarrow{\delta}_p = J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p}_k)$$

where

$$\overrightarrow{f}(\overrightarrow{p}) = \begin{pmatrix} \frac{h_{11}x_1+h_{12}y_1+h_{13}}{h_{31}x_1+h_{32}y_1+h_{33}} \\ \frac{h_{21}x_1+h_{22}y_1+h_{23}}{h_{31}x_1+h_{32}y_1+h_{33}} \\ \vdots \\ \frac{h_{11}x_N+h_{12}y_N+h_{13}}{h_{31}x_N+h_{32}y_N+h_{33}} \\ \frac{h_{21}x_N+h_{22}y_N+h_{23}}{h_{31}x_N+h_{32}y_N+h_{33}} \end{pmatrix} \tag{5}$$

,

$$\overrightarrow{p} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix}$$

. The first step is to define the initial value of the damping coefficient. $\mu_0 = \tau \cdot max\{diag(J_{\overrightarrow{f}}^T J_{\overrightarrow{f}})\}$ where τ is a constant between 0 and 1.

Then we start to loop until we have the cost of $\overrightarrow{p}$ stop decreasing. In side the loop:

we solve for

$$\overrightarrow{\delta}_p = (J_{\overrightarrow{f}}^T J_{\overrightarrow{f}} + \mu_k I)^{-1} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p}_k)$$

.

Then we update $p_{k+1} = p_k + \overrightarrow{\delta}_p$. we will calculate ρ by

$$\rho_{k+1}^{LM} = \frac{C(\overrightarrow{p}_k) - C(\overrightarrow{p}_{k+1})}{\overrightarrow{\delta}_p^T J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p}_k) + \overrightarrow{\delta}_p^T \mu_k I \overrightarrow{\delta}_p}$$

where cost function $C(\overrightarrow{p}_k) = \|\overrightarrow{X} - \overrightarrow{f}(\overrightarrow{p}_k)\|^2$.

The damping coefficient is updated by

$$\mu_{k+1} = \mu_k \cdot max\{\frac{1}{3}, 1 - (2\rho_{k+1}^{LM} - 1)^3\}$$

. End of loop.

2.1.4 Image Mosaicing

When generating panorama, the first step is to generate a list of H by the ransac methods mentioned in the previous sections between the images(0 to 1, 1 to 2 ...). With these Hs, we can have the H' toward the center image. For example, $H_{02} = H_{12}H_{01}$ Then we use this list to generate the transformed coordinates. Then we can have a empty panorama image needed for the 5 images. The next step would be generating the mask image with the coordinates. For each points in the mask, we use the H to find the corresponding points in each image.

3 Data and images:

Coefficients:

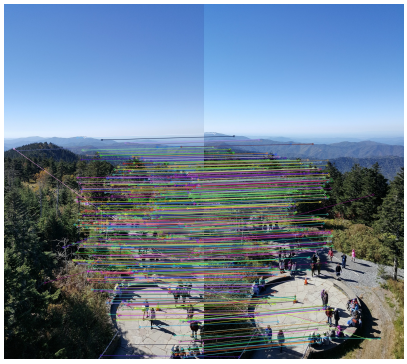
Parameter	Value
δ	3
n	10
ϵ	0.25
p	0.99

3.1 task1

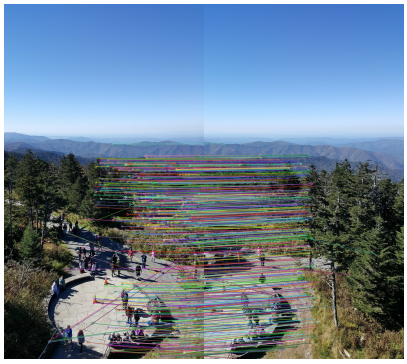
Input images:

(a) 0 (b) 1 (c) 2 (d) 3 (e) 4

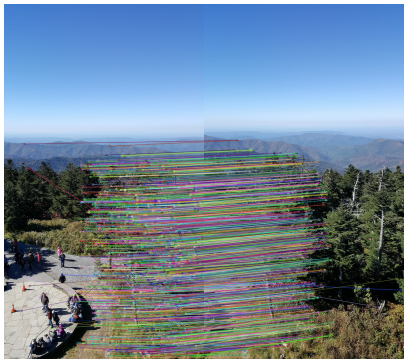
Correspondences between adjacent images:



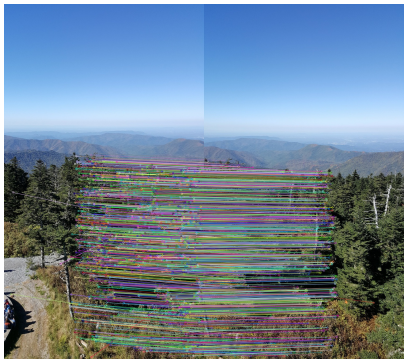
(a) 0_1 correspondences



(b) 1_2 correspondences

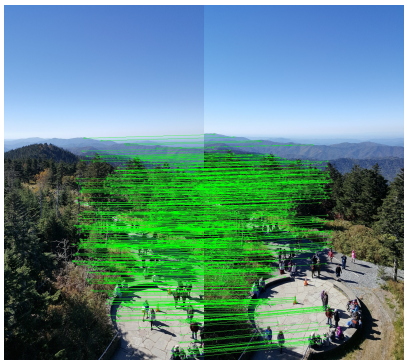


(c) 2_3 correspondences



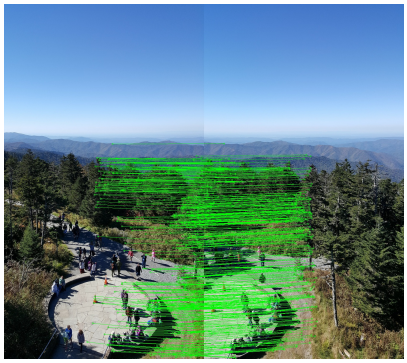
(d) 3_4 correspondences

Set of inliers and outliers without LM:

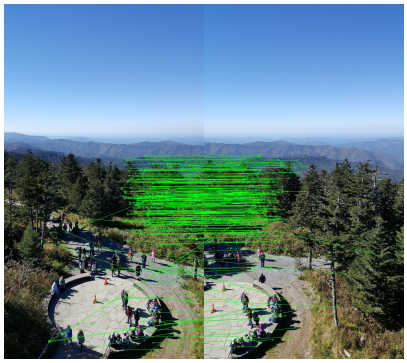


(a) 0_1 inliers

(b) 0_1 outliers

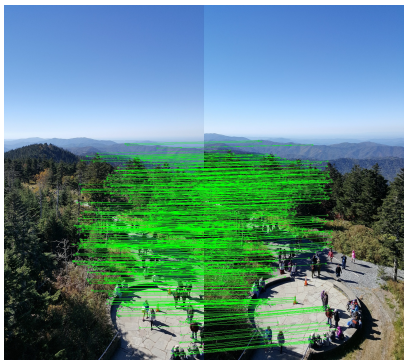


(c) 1_2 inliers



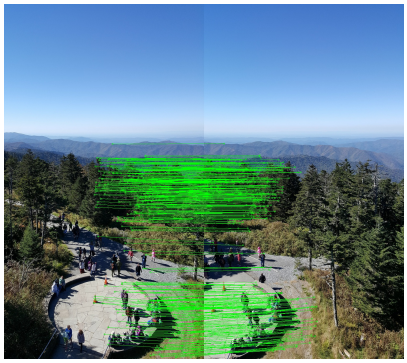
(d) 1_2 outliers

Set of inliers and outliers with LM:

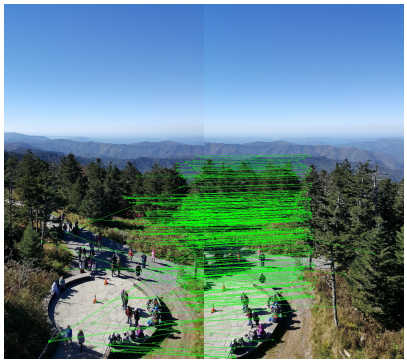


(a) 0_1 inliers

(b) 0_1 outliers



(c) 1_2 inliers



(d) 1_2 outliers

panorama:

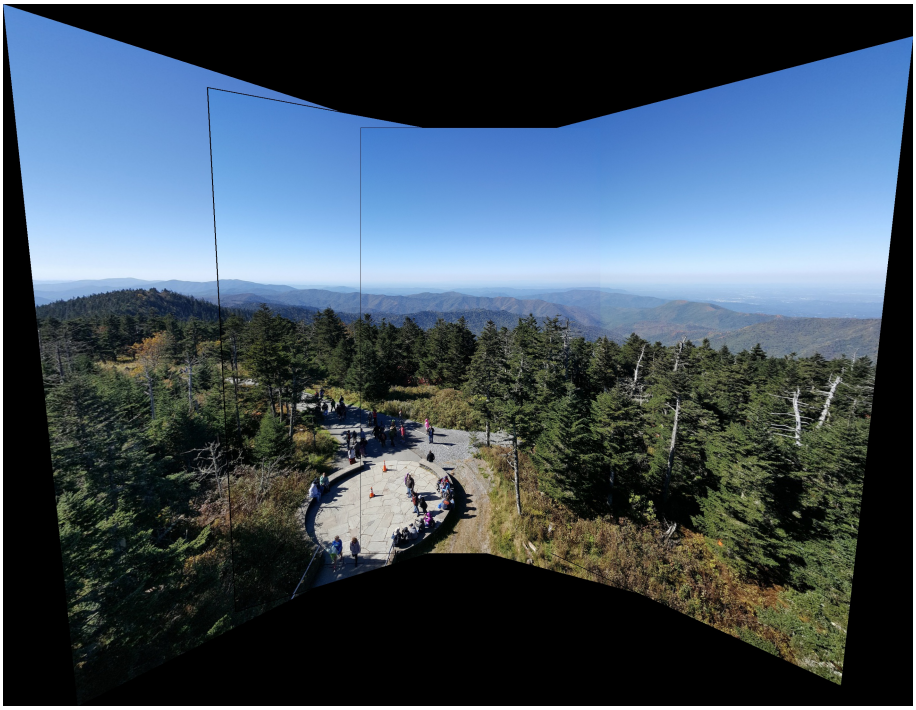
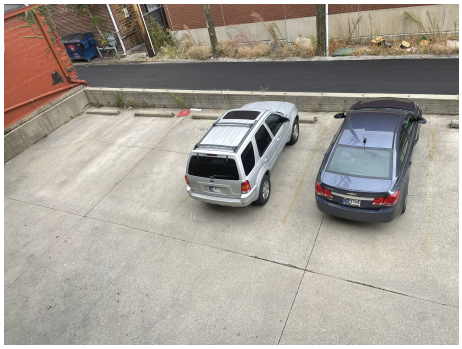


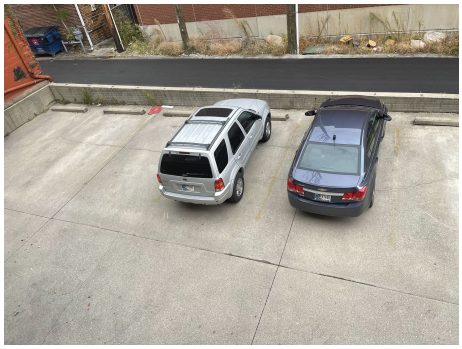
Figure 5: task1 output with LM

3.2 task2

Input images:

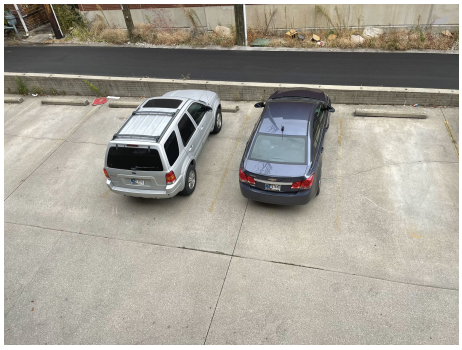


(a) 5

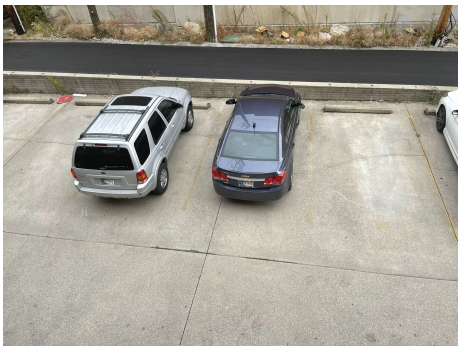


(b) 6

(c) 7



(d) 8



(e) 9

Correspondences between adjacent images:

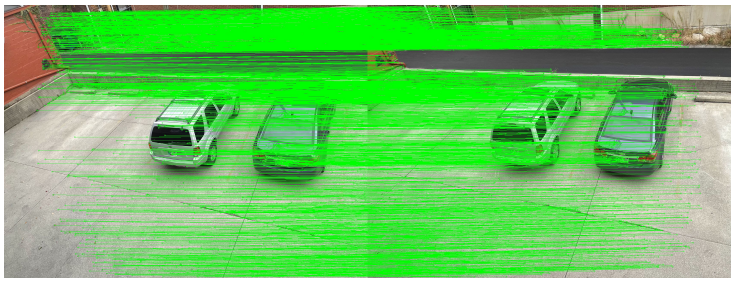
(a) 5_6 correspondences

(b) 6_7 correspondences

(c) 7_8 correspondences

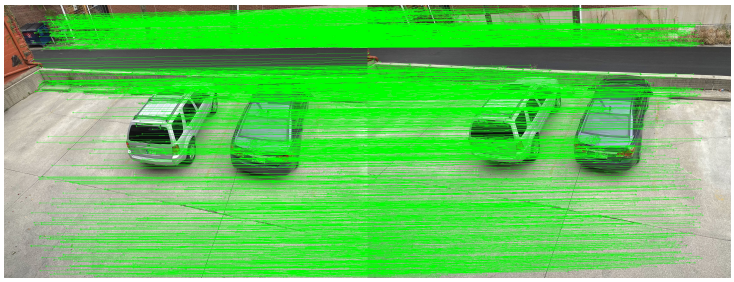
(d) 8_9 correspondences

Set of inliers and outliers without LM:

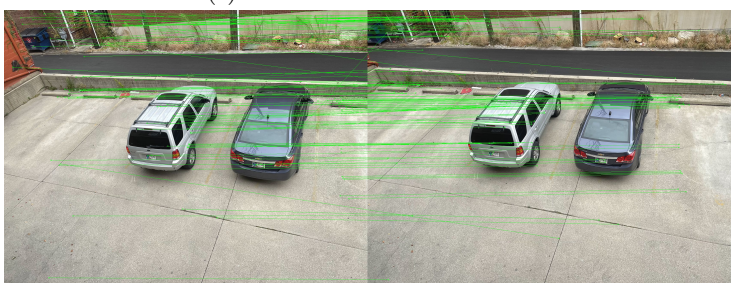


(a) 5_6 inliers

(b) 5_6 outliers

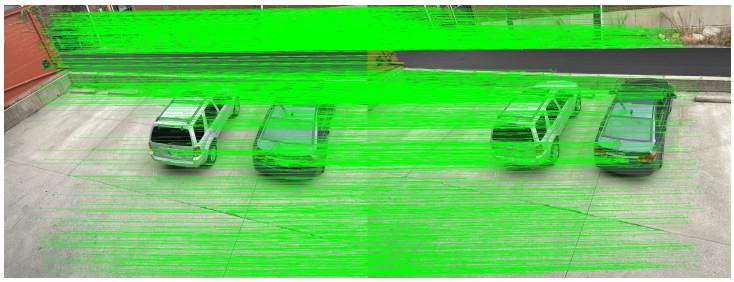


(c) 6_7 inliers

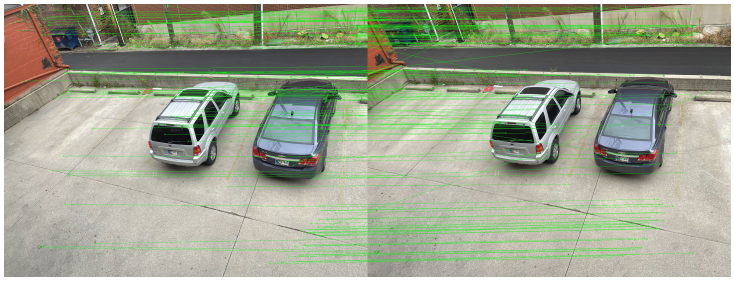


(d) 6_7 outliers

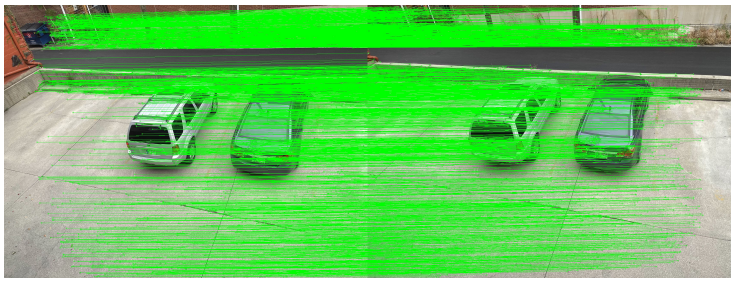
Set of inliers and outliers with LM:



(a) 5_6 inliers



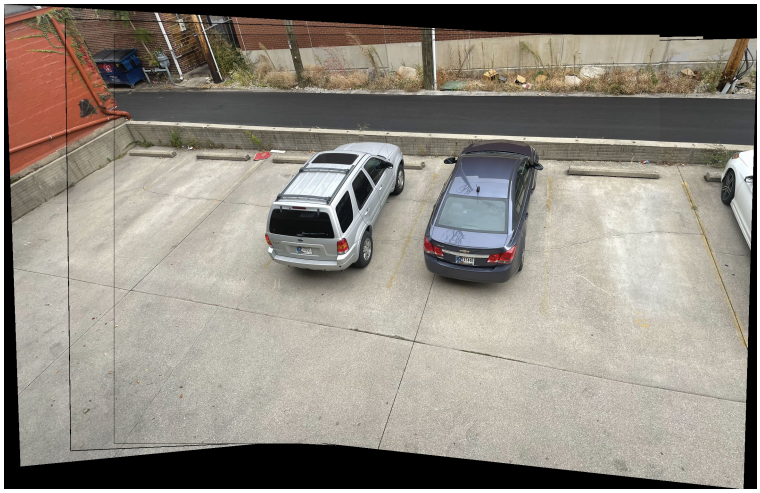
(b) 5_6 outliers



(c) 6_7 inliers

(d) 6_7 outliers

panorama:



4 Code

```
import numpy as np
import cv2
import sys
import math
import matplotlib.pyplot as plt
import random
from scipy.ndimage import map_coordinates
from scipy.optimize import least_squares


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 sift_match(img1, img2):
    # img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    # img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)
    bf = cv2.BFMatcher()
    # image1_kp = cv2.drawKeypoints(img1, kp1, outImage=np.array([]),\
        flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    matches = bf.knnMatch(des1, des2, k=2)
    good = []
    match_points = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good.append([m])
            img2_idx = m.trainIdx
            img1_idx = m.queryIdx
            (x1, y1) = kp1[img1_idx].pt
```

```
            (x2, y2) = kp2[img2_idx].pt
            # print("Comare %d to %d and %d to %d" % (x1, x2, y1, y2))
            match_points.append([(x1, y1), (x2, y2)])
    ans = cv2.drawMatchesKnn(img1, kp1, img2, kp2, good, None, flags=2)
    return ans, match_points


def homography_calculate(X, X_prime):
    A = np.zeros((2 * len(X), 8))
    B = np.zeros((2 * len(X), 1))
    for i in range(len(X)):
        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]

    A_plus = np.matmul(np.linalg.pinv(np.matmul(A.T,A)),A.T)
    H_vec = np.ndarray.flatten(np.dot(A_plus, B))
    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 RANSAC(points,mode='LM'):
    # constant variable
    n = 10
    epsilon = 0.25
    p = 0.99
    delta = 3
    n_total = len(points)
    N = int(np.log(1 - p) / np.log(1 - (1 - epsilon) ** n))
    M = int((1 - epsilon) * n_total)
    num_inliers = 0
    inlier_index = []
    #N trials
```

```
    X = [i[0] for i in points]
    X_prime = [i[1] for i in points]
    for i in range(N):
        pts = random.sample(points,n)
        domain_point = [i[0] for i in pts]
        range_point = [i[1] for i in pts]
        H = homography_calculate(domain_point,range_point)
        idx = find_inliers(H,X,X_prime,delta)
        if len(idx) > num_inliers:
            num_inliers = len(idx)
            inlier_index = idx
    if num_inliers < M:
        print('Minimum size of inlier not obtained')
    inliers = [points[i] for i in inlier_index]
    print(len(inliers))
    H_robust = homography_calculate([i[0] for i in inliers],[j[1] for j in inliers])
    #if LM
    if mode == 'LM':
        h = H_robust.flatten()
        res = least_squares(cost_func_H, h, args=([i[0] for i in inliers],\
            [j[1] for j in inliers]),method='lm')
        new = res.x
        H_robust = new.reshape((3,3))
    return H_robust,inliers

def find_inliers(H,X,X_prime,delta):
    X_vec = np.insert(X, 2, 1, axis=1)
    X_prime_cal = np.matmul(H,X_vec.T).T
    X_prime_cal = X_prime_cal / X_prime_cal[:,2].reshape(-1,1)
    X_prime_vec = np.insert(X_prime, 2, 1, axis=1)
    error = np.linalg.norm(X_prime_cal-X_prime_vec, axis = 1)
    idx = np.where(error <= delta)
    return idx[0]

def cost_func_H(H,X,X_prime):
    H = H.reshape((3,3))
    X_vec = np.insert(X, 2, 1, axis=1)
    X_prime_cal = np.matmul(H,X_vec.T).T
    X_prime_cal = X_prime_cal / X_prime_cal[:,2].reshape(-1,1)
    X_prime_vec = np.insert(X_prime, 2, 1, axis=1)
    error = np.linalg.norm(X_prime_cal-X_prime_vec, axis = 1)
    return (error**2)
```

```
def visualize_in_out(points,inliers,img1,img2,name):
    img_in = np.concatenate((img1, img2), axis=1)
    img_out = np.concatenate((img1, img2), axis=1)
    offset = img1.shape[1]
    for i in points:
        if i in inliers:
            pt1 = (int(i[0][0]),int(i[0][1]))
            pt2 = (int(i[1][0] + offset), int(i[1][1]))
            img_in = cv2.circle(img_in, pt1, 3, (0, 255, 0), 1)
            img_in = cv2.circle(img_in, pt2, 3, (0, 255, 0), 1)
            img_in = cv2.line(img_in, pt1, pt2, (0, 255, 0), 1)
        else:
            pt1 = (int(i[0][0]),int(i[0][1]))
            pt2 = (int(i[1][0] + offset), int(i[1][1]))
            img_out = cv2.circle(img_out, pt1, 3, (0, 255, 0), 1)
            img_out = cv2.circle(img_out, pt2, 3, (0, 255, 0), 1)
            img_out = cv2.line(img_out, pt1, pt2, (0, 255, 0), 1)

    cv2.imwrite(name + '_in.jpg', img_in)
    cv2.imwrite(name + '_out.jpg', img_out)


def calculate_in_out():
    mode = 'none'
    image_num = 5
    for i in range(image_num - 1):
        i = i + 5
        img1 = cv2.imread(str(i) + '.jpg')
        img2 = cv2.imread(str(i + 1) + '.jpg')
        pic, points = sift_match(img1, img2)
        # cv2.imwrite(str(i) + '_' + str(i + 1) + '.jpg', pic)
        # for pair_point in points:
        #     img1 = cv2.circle(img1, (int(pair_point[0][0]), int(pair_point[0][1])), 3, (0,
        #     img2 = cv2.circle(img2, (int(pair_point[1][0]), int(pair_point[1][1])), 3, (0,
        # cv2.imwrite(str(i) + '_prime.jpg', img1)
        # cv2.imwrite(str(i + 1) + '_prime.jpg', img2)
        cv2.imwrite(str(i) + '_' + str(i + 1) + '.jpg', pic)
        H, inliers = RANSAC(points,mode)
        visualize_in_out(points,inliers,img1,img2,str(i) + '_' + str(i + 1) + '_'+mode)


def panorama():
    mode = 'LM'
    image_num = 5
    imgs = []
    for i in range(image_num):
```

```
    i = i + 5
    img = cv2.imread(str(i) + '.jpg')
    imgs.append(img)
Hs = []
for i in range(image_num-1):
    pic, points = sift_match(imgs[i],imgs[i+1])
    H, inliers = RANSAC(points,mode)
    Hs.append(H)
H_to_centers = [np.matmul(Hs[1],Hs[0]),Hs[1],np.eye(3),np.linalg.inv(Hs[2]),np.linalg.in
x_max = float('-inf')
x_min = float('inf')
y_max = float('-inf')
y_min = float('inf')
corners = []
for i in range(image_num):
    H = H_to_centers[i]
    tl_prime = np.dot(H, np.array([0, 0, 1]))
    tr_prime = np.dot(H, np.array([len(imgs[i][0]) - 1, 0, 1]))
    bl_prime = np.dot(H, np.array([0, len(imgs[i]) - 1, 1]))
    br_prime = np.dot(H, np.array([len(imgs[i][0]) - 1, len(imgs[i]) - 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])]
    # print(tl_point,tr_point,bl_point,br_point)
    corners.append([tl_point, bl_point, br_point, tr_point])
    x_max = max(tr_point[0], br_point[0], x_max)
    x_min = min(tl_point[0], bl_point[0], x_min)
    y_max = max(bl_point[1], br_point[1], y_max)
    y_min = min(tl_point[1], tr_point[1], y_min)
# 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), 3)

area = np.zeros(shape)
for count, corner_of_one in enumerate(corners):
    for corner_pt in corner_of_one:
        corner_pt[0] += x_offset
        corner_pt[1] += y_offset
    image_to_fill = cv2.fillPoly(area, pts=[np.array(corner_of_one, dtype=np.int32)], co
# print(corners)
# cv2.imwrite('empty_mask.jpg',image_to_fill)
H_inv = [np.linalg.inv(i) for i in H_to_centers]
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])
                p_x = j - x_offset
                p_y = i - y_offset
                if p_y > -1 and p_y < len(imgs[2]) and p_x > -1 and p_x < len(imgs[2][0]):
                    if p_y < 0:
                        p_y = 0
                    if p_y > len(imgs[2]) - 1:
                        p_y = len(imgs[2]) - 1
                    if p_x < 0:
                        p_x = 0
                    if p_x > len(imgs[2][0]) - 1:
                        p_x = len(imgs[2][0]) - 1
                    image_to_fill[i, j] = imgs[2][p_y,p_x]
                else:
                    for count in range(len(H_inv)):
                        point_in_x = np.dot(H_inv[count], point_in_x_prime)
                        x = point_in_x[0] / point_in_x[2]
                        y = point_in_x[1] / point_in_x[2]
                        if y > -2 and y < len(imgs[count]) + 1 and x > -2 and x < len(imgs[c
                            if y < 0:
                                y = 0
                            if y > len(imgs[count]) - 1:
                                y = len(imgs[count]) - 1
                            if x < 0:
                                x = 0
                            if x > len(imgs[count][0]) - 1:
                                x = len(imgs[count][0]) - 1
                            image_to_fill[i, j] = bilinear_interpolation(y, x, imgs[count])
    cv2.imwrite('56789_panorama.jpg',image_to_fill)


if __name__ == '__main__':
    calculate_in_out()
    # panorama()
```