ECE 661 Homework 5

Weichen Xu

xu1363@purdue.edu

Theory Questions

1 How do we differentiate between the inliers and the outliers using RANSAC?

Usually inliers are correspondences of points with merely noise, outliers are only false correspondences. Since using Linear Least-Square method to estimate homography only tolerates inliers, the outliers should be removed.

To view the problem of differentiating between the inliers and the outliers as the linear regression problem. Consider x and y as two random variables, linear regression estimates a 1-D affine transformation $y = ax + b$, since at least two points are needed to form a line, RANSAC will randomly pick two points constructing a line, then the support is measure by the number of points that lie within certain distance threshold of the line. The points within the threshold distance of the line are inliers, points outside the threshold distance of the line are outliers.

Applying to the case of 2-D image, at least four pairs of correspondences are randomly selected, then using the linear least-square method to estimate the homography. We calculate the Euclidean distance between transformed point and its original corresponding point, if the value is below certain threshold, the pair is viewed as inliers, otherwise is outliers.

2 How Levenberg-Marquardt (LM) algorithm combines the best of Gradient-Descent (GD) and Gauss-Newton (GN)?

If the starting point is far from the minimum, LM algorithm behaves like GD, as the solution point approaches the minimum, it behaves like GN.

For GN's solution, it is given by

$$\overrightarrow{\delta_p} = \left(J_{\vec{f}}^T J_{\vec{f}}\right)^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p})$$

We can rewrite it as

$$\left(J_{\vec{f}}^T J_{\vec{f}}\right)\overrightarrow{\delta_p} = J_{\vec{f}}^T \vec{\epsilon}(\vec{p})$$

If $J_{\vec{f}}^T J_{\vec{f}}$ were purely diagonal, the solution of GN will be the same as GD. We can heuristically extend the equation to the following form

$$\overrightarrow{\delta_p} = \left(J_{\vec{f}}^T J_{\vec{f}} + \mu I\right)^{-1} J_{\vec{f}}^T \vec{\epsilon}(\vec{p})$$

The LM method add a damping coefficient μ, making it equal to GN if $\mu = 0$, close to GD if μ is much larger than elements of $J_{\vec{f}}^T J_{\vec{f}}$.

Scale Invariant Feature Transform (SIFT)

Since we have covered the SIFT method from the last homework, here the details of SIFT method is left out. Initially the SIFT algorithm will extract 1000 feature points with corresponding descriptor vectors, the correspondences are built through choosing pairs with least Euclidean distance. Here the first 100 pairs of correspondences of least Euclidean distances are kept as the valid correspondence pairs.

Random Sample Consensus (RANSAC)

The correspondence pairs return by the SIFT algorithm still have some false matching pairs, which are called outliers. Since the next step Linear Least-Square method requires a relatively correct point location to estimate homography, the outliers should be removed to improve the homography. The implementation procedure of RANSAC is following:

1. Select the probability ϵ that the ratio of outliers in the initial correspondences, here I choose $\epsilon = 0.1$. Then the total trials of conduct N is expressed as $N = \frac{\ln(1-p)}{\ln[1-(1-\epsilon)^n]}$, where p is the probability of at least one of the N trials has no outliers in the following calculation. n is the number of correspondences used to estimate the homography for trials. The minimum number of accepted inliers considered as a successful trial $M = (1-\epsilon) * n_{all}$, where n_{all} is the number of all initial correspondences.
2. For repetitive N trials, a set of random 4 correspondences are selected, thus an estimated homography H can be obtained. Transform all points in the correspondences in domain 1 to the domain 2.
3. Calculating the distance between the transformed point in domain 2 with its corresponding point in domain 2. Only if the distance is less than the threshold δ, such correspondence pair can be viewed as inliers, otherwise outliers.
4. After N trials, the set resulting at the greatest number of inliers pairs is returned to be the result with inliers of RANSAC (If there are sets that has the number of inliers larger than M)

Linear Least-Square Method to Estimate Homography

Recall on previous homework, the transformation of a point by homography matrix H can be expressed as

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = H \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

While

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$$

We can then transform to the following equations

$$x' = h_{11}x + h_{12}y + h_{13} - h_{31}xx' - h_{32}yx'$$
$$y' = h_{21}x + h_{22}y + h_{23} - h_{31}xy' - h_{32}yy'$$

For one correspondence pair, we have two equations for eight unknowns, if we have the correspondence pairs that are more than 4, we will have more than eight equations for eight unknowns. In other words, we are solving an over-determined system.

$$\underbrace{\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_nx_n' & -y_nx_n' \\ 0 & 0 & 0 & x_n & y_n & 1 & -x_ny_n' & -y_ny_n' \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix}}_{\vec{h}} = \underbrace{\begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ \vdots \\ x_n' \\ y_n' \end{bmatrix}}_{\vec{b}}$$

To look for the h, where h is the vector form of the homography H, that minimizes $\left\|A\vec{h} - \vec{b}\right\|$

$$\vec{h} = (A^T A)^{-1} A^T \vec{b}$$

Homography Refinement with LM

The Levenberg-Marquardt (LM) algorithm combines the best of Gradient-Descent (GD) and Gauss-Newton (GN), its equation is given by

$$\left(J_{\vec{f}}^T J_{\vec{f}} \ + \mu I\right) \overrightarrow{\delta_p} = J_{\vec{f}}^T \vec{\epsilon}(\vec{p})$$

While

$$\vec{f}(\vec{p}) = \begin{bmatrix} f_1^1(\vec{p}) \\ f_2^1(\vec{p}) \\ \vdots \\ f_1^N(\vec{p}) \\ f_2^N(\vec{p}) \end{bmatrix} = \begin{bmatrix} h_{11}x_1 + h_{12}y_1 + h_{13}/h_{31}x_1 + h_{32}y_1 + h_{33} \\ h_{21}x_1 + h_{22}y_1 + h_{23}/h_{31}x_1 + h_{32}y_1 + h_{33} \\ \vdots \\ h_{11}x_N + h_{12}y_N + h_{13}/h_{31}x_N + h_{32}y_N + h_{33} \\ h_{21}x_N + h_{22}y_N + h_{23}/h_{31}x_N + h_{32}y_N + h_{33} \end{bmatrix}$$

$$\vec{p} = [h_{11}\ h_{12}\ h_{13}\ h_{21}\ h_{22}\ h_{23}\ h_{31}\ h_{32}\ h_{33}]^T$$

The optimization goal is to minimize the error between the loss in the estimation $\left\|\vec{X}-\vec{f}(\vec{p})\right\|$

Following are the steps for the LM algorithm:

1. First estimate the initial damping coefficient μ_0, it is given by $\mu_0=\tau\cdot\max\left\{diag\left(J_{\vec{f}}^T J_{\vec{f}}\right)\right\}$ where $\tau\in(0,1]$.
2. Compute $\overrightarrow{\delta_p}$ according to the following equation $\overrightarrow{\delta_p}=\left(J_{\vec{f}}^T J_{\vec{f}}+\mu I\right)^{-1}J_{\vec{f}}^T\vec{\epsilon}(\vec{p})$
3. Compute the ratio $\rho=\frac{C(\vec{p}_k)-C(\vec{p}_{k+1})}{\overrightarrow{\delta_p}^T J_{\vec{f}}^T\vec{\epsilon}(\vec{p}_k)+\overrightarrow{\delta_p}^T\mu_k I\overrightarrow{\delta_p}}$, where $C(\vec{p}_k)=\left\|\vec{X}-\vec{f}(\overrightarrow{p_k})\right\|$
4. The updated damping coefficient for the next iteration is given by $\mu_{k+1}=\mu_k\cdot\max\{\frac{1}{3},1-(2\rho-1)^3\}$
5. Iterate until the loss $C(\vec{p}_k)$ stop decreasing.

Stitching Images

We have five images with overlapping areas, to project them into a common frame, we need to assign the destination frame is the center image Img_3, while Img_i denotes the ith image. Then we can get the homography matrix from the previous steps, let denote H_{ij} is the homography matrix between Img_i and Img_j.

Since H_{23}, H_{43} can directly obtained from the previous steps, we also need to get the H_{13}, H_{53}, which can be calculated as

$$H_{13}=H_{12}H_{23}$$

$$H_{53}=H_{54}H_{43}$$

Then we can map each image Img_i with its corresponding matrix H_{i3} to the frame of image 3. It's noted that the stitched image has the larger size than original image 3, and the image 3 should be centered on the stitched image.

Experiment Results

Figure 1: Input image 1

Figure 2: Input image 2

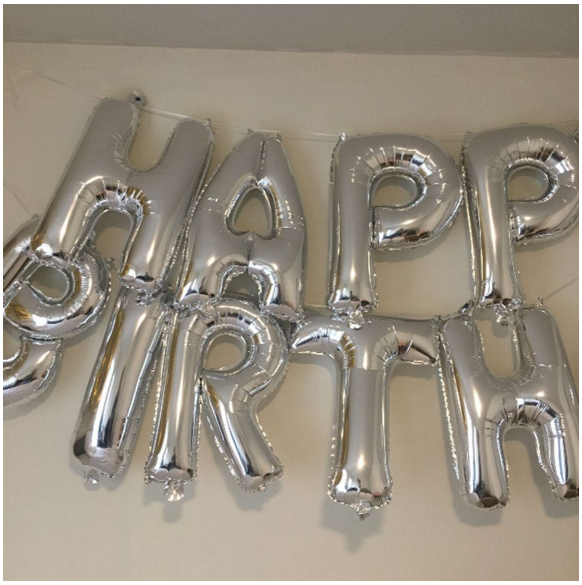

Figure 3: Input image 3

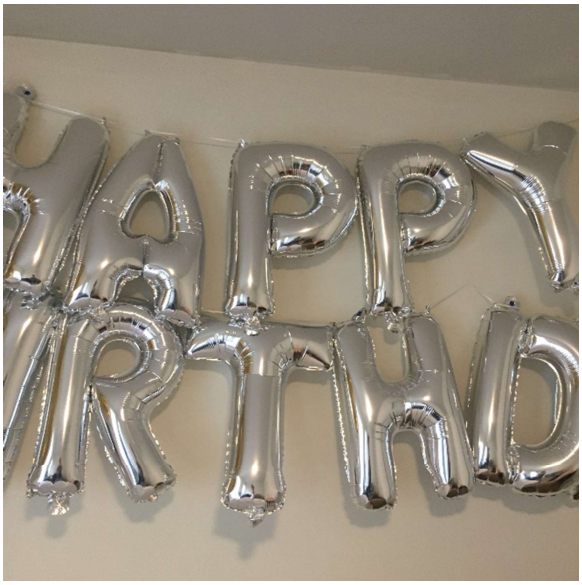

Figure 4: Input image 4

Figure 5: Input image 5

Figure 6: Correspondences between image 1 and image 2, while the yellow lines are inliers, the red lines are outliers

Figure 7: Correspondences between image 2 and image 3, while the yellow lines are inliers, no outliers is detected in this case

Figure 8: Correspondences between image 3 and image 4, while the yellow lines are inliers, the red lines are outliers

Figure 9: Correspondences between image 4 and image 5, while the yellow lines are inliers, the red lines are outliers

Figure 10: Stitched panorama image

Optimal Parameters

SIFT	# of features	1000
SSD	Threshold of # of best matching pairs	100
RANSAC	Probability of outliers in all pairs ϵ	0.1
	# of pairs used to estimate H	4
	Probability of at least one trial has no outliers p	0.99

Code

```
#!/usr/bin/env python

# coding: utf-8



import numpy as np

import cv2

import matplotlib.pyplot as plt

from scipy import signal


def GetSSD(patch1, patch2):

    # calculate the SDD value from the formula

    sum = np.sum(np.sum(np.square(patch1 - patch2)))

    return sum


def Correspondence_SSD (img1, corners1, img2, corners2, window_size = 21):

    # get the relevant correspondent pairs with SSD

    value_min = []

    index_min = []

    h = int(window_size / 2)


    corners1 = GetValidCorners(corners1, img1, window_size)

    corners2 = GetValidCorners(corners2, img2, window_size)
```

```
for pt1 in corners1:
    x1 = pt1[0]
    y1 = pt1[1]
    patch1 = img1[y1 - h: y1 + h, x1 - h: x1 + h]

    SSD_all = []
    for pt2 in corners2:
        x2 = pt2[0]
        y2 = pt2[1]
        patch2 = img2[y2 - h: y2 + h, x2 - h: x2 + h]
        SSD_all.append(GetSSD(patch1, patch2))

    # calculate the current image patch's minimum SSD value
    idx = np.argmin(SSD_all)
    value_min.append(SSD_all[idx])
    index_min.append(idx)

# set the threshold of keeping pairs
threshold = np.min(value_min) * 3
#print(np.min(value_min))
#print(np.max(value_min))
corners1_corr = []
corners2_corr = []

for i in range(len(value_min)):
    if value_min[i] < threshold:
        corners1_corr.append(corners1[i])
        corners2_corr.append(corners2[index_min[i]])
```

```
    return corners1_corr, corners2_corr


def getSIFT(img, nfeatures = 2000):
    # get the desired number of interest points and descriptive vectors
    sift = cv2.xfeatures2d.SIFT_create(nfeatures = nfeatures)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kps, des = sift.detectAndCompute(img_gray, None)


    return kps, des


def Correspondence_SIFT(img1, kps1, des1, img2, kps2, des2):
    # Keep only the 100 most relavant pairs of interest points
    corr1 = []
    corr2 = []
    dist_min_all = []
    for i in range(len(kps1)):
        pt1 = kps1[i].pt
        pt1 = np.round(pt1)


        # measure the Euclidean distance between two descriptor vector
        dist_min = np.sqrt(np.sum(np.multiply(des1[i] - des2[0], des1[i] - des2[0])))
        for j in range(len(kps2)):
            pt2 = kps2[j].pt
            pt2 = np.round(pt2)


            dist = np.sqrt(np.sum(np.multiply(des1[i] - des2[j], des1[i] - des2[j])))
            if dist <= dist_min:
                dist_min = dist
```

```
            pt1_corr = pt2

        corr1.append(pt1)
        corr2.append(pt1_corr)
        dist_min_all.append(dist_min)

    dist_sorted = sorted(dist_min_all)
    corr1_sorted = []
    corr2_sorted = []

    for i in range(100):
        idx = np.where(dist_min_all == dist_sorted[i])
        idx = idx[0][0]
        corr1_sorted.append(corr1[idx])
        corr2_sorted.append(corr2[idx])
    return corr1_sorted, corr2_sorted




def get_homography(target,source):
    A = np.zeros((8,8))
    b = np.zeros((8,1))
    for i in range(4):
        x1 = source[i][0]
        y1 = source[i][1]
        x2 = target[i][0]
        y2 = target[i][1]
```

```
        A[i*2,:] = [x1,y1,1,0,0,0,-x1*x2,-y1*x2]
        A[i*2+1,:] = [0,0,0,x1,y1,1,-x1*y2,-y1*y2]

        b[i*2] = x2
        b[i*2+1] = y2

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

def get_random_set(pts1, pts2,n):
    # return random set for homography estimation in RANSAC
    order = np.random.permutation(len(pts1))
    output1 = []
    output2 = []
    for i in range(n):
        output1.append(pts1[order[i]])
        output2.append(pts2[order[i]])
    return output1, output2


def get_inliers(pts1, pts2, H, delta):
    # obtain inliers for distance below setting threshold
    temp = np.array([pts1], dtype='float32')
    pts1_trans = cv2.perspectiveTransform(temp, H)
    inliers1 = []
    inliers2 = []
    outliers1 = []
```

```
    outliers2 = []
    num = 0
    for i in range(len(pts1)):
        pt1 = pts1_trans[0][i]
        pt2 = pts2[i]
        dist = np.sqrt((pt1[0] - pt2[0])**2 + (pt1[1] - pt2[1])**2)
        if dist < delta:
            num = num + 1
            inliers1.append(pts1[i])
            inliers2.append(pts2[i])

        if dist >= delta:
            outliers1.append(pts1[i])
            outliers2.append(pts2[i])

    return num, inliers1, inliers2, outliers1, outliers2



def RANSAC(pts1, pts2, epsilon, delta):
    # M is the number of inliers, N is the number of trials
    M = int((1 - epsilon) * len(pts1))
    N = int(np.log(1 - 0.99) / np.log(1 - (1 - epsilon)**4 ))
    inliers1 = []
    inliers2 = []
    for i in range(N):
        # randomly pick subset of four points to construct homography
        pts1_4, pts2_4 = get_random_set(pts1, pts2, 4)
        H = get_homography(pts2_4, pts1_4)
```

```
        num, inliers1, inliers2, outliers1, outliers2 = get_inliers(pts1, pts2, H, delta)
        if num > M:
            #print(num, 'pairs of inliers found')
            break

    return inliers1, inliers2, outliers1, outliers2

def get_homography_LLS(pts1, pts2):
    # whole process for getting homography using multiple correspondences
    # using Linear Least-Square method
    num = len(pts1)
    A = np.zeros((2*num, 8))
    b = np.zeros((2*num, 1))

    for i in range(num):
        x1, y1 = pts1[i]
        x2, y2 = pts2[i]
        A[i*2,:] = [x1,y1,1,0,0,0,-x1*x2,-y1*x2]
        A[i*2+1,:] = [0,0,0,x1,y1,1,-x1*y2,-y1*y2]
        b[i*2] = x2
        b[i*2+1] = y2

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

    return H

def get_homography_all(img1, img2):
```

```
    kps1, des1 = getSIFT(img1)
    kps2, des2 = getSIFT(img2)
    l1, l2 = Correspondence_SIFT(img1, kps1, des1, img2, kps2, des2)
    inliers1, inliers2, outliers1, outliers2 = RANSAC(l1, l2, epsilon, delta)
    H = get_homography_LLS(inliers1, inliers2)

    return H

def plot_inliers_outliers(img1, img2):
    kps1, des1 = getSIFT(img1)
    kps2, des2 = getSIFT(img2)
    l1, l2 = Correspondence_SIFT(img1, kps1, des1, img2, kps2, des2)
    inliers1, inliers2, outliers1, outliers2 = RANSAC(l1, l2, epsilon, delta)
    h, w, c = img1.shape
    img_synthetic = np.zeros((h, 2*w, c), dtype = np.uint8)
    img_synthetic[:, 0:w, :] = img1
    img_synthetic[:, w:2*w, :] = img2

    plt.figure(dpi=1200)
    plt.imshow(cv2.cvtColor(img_synthetic, cv2.COLOR_BGR2RGB))

    for i in range(len(inliers1)):
        plt.scatter(inliers1[i][0], inliers1[i][1], color = 'yellow', s = 0.3)
        plt.scatter(inliers2[i][0] + w, inliers2[i][1], color = 'yellow', s = 0.3)
        plt.plot([inliers1[i][0], inliers2[i][0] + w], [inliers1[i][1], inliers2[i][1]], color = 'yellow',  linewidth =
0.1)

    for i in range(len(outliers1)):
        plt.scatter(outliers1[i][0], outliers1[i][1], color = 'red', s = 0.3)
```

```
        plt.scatter(outliers2[i][0] + w, outliers2[i][1], color = 'red', s = 0.3)
        plt.plot([outliers1[i][0], outliers2[i][0] + w], [outliers1[i][1], outliers2[i][1]], color = 'red',  linewidth = 0.1)




def mapping(img_target,H):

    P_distort = np.array([0,0,1])
    Q_distort = np.array([0,img_target.shape[0]-1,1])
    R_distort = np.array([img_target.shape[1]-1,img_target.shape[0]-1,1])
    S_distort = np.array([img_target.shape[1]-1,0,1])



    P_world = np.matmul(H,P_distort)
    P_world = P_world / P_world[2]
    Q_world = np.matmul(H,Q_distort)
    Q_world = Q_world / Q_world[2]
    R_world = np.matmul(H,R_distort)
    R_world = R_world / R_world[2]
    S_world = np.matmul(H,S_distort)
    S_world = S_world / S_world[2]

    xmin = np.int32(np.round(np.amin([P_world[0],Q_world[0],R_world[0],S_world[0]])))
    xmax = np.int32(np.ceil(np.amax([P_world[0],Q_world[0],R_world[0],S_world[0]])))
    ymin = np.int32(np.round(np.amin([P_world[1],Q_world[1],R_world[1],S_world[1]])))
    ymax = np.int32(np.ceil(np.amax([P_world[1],Q_world[1],R_world[1],S_world[1]])))
```

```
    xlen = xmax-xmin
    ylen = ymax-ymin

    img_new = np.zeros((ylen,xlen,3), dtype=np.uint8)
    print('The output image size is',xlen,ylen)
    Hinv = np.linalg.inv(H)

    for i in range(xlen):
        for j in range(ylen):
            input = np.array([i+xmin,j+ymin,1])
            output = np.matmul(Hinv,input)
            x = np.int(np.round(output[0]/output[2]))
            y = np.int(np.round(output[1]/output[2]))

            if x>0 and x<img_target.shape[1]-1 and y>0 and y<img_target.shape[0]-1:
                img_new[j,i,:] = img_target[y,x,:]
    return img_new




def get_bounday(img, H):
    # get the four corners' coordinate in transformed domain
    P_distort = np.array([0,0,1])
    Q_distort = np.array([0,img.shape[0]-1,1])
    R_distort = np.array([img.shape[1]-1,img.shape[0]-1,1])
    S_distort = np.array([img.shape[1]-1,0,1])
```

```
    P_world = np.matmul(H,P_distort)
    P_world = P_world / P_world[2]
    Q_world = np.matmul(H,Q_distort)
    Q_world = Q_world / Q_world[2]
    R_world = np.matmul(H,R_distort)
    R_world = R_world / R_world[2]
    S_world = np.matmul(H,S_distort)
    S_world = S_world / S_world[2]

    xmin = np.int32(np.round(np.amin([P_world[0],Q_world[0],R_world[0],S_world[0]])))
    xmax = np.int32(np.ceil(np.amax([P_world[0],Q_world[0],R_world[0],S_world[0]])))
    ymin = np.int32(np.round(np.amin([P_world[1],Q_world[1],R_world[1],S_world[1]])))
    ymax = np.int32(np.ceil(np.amax([P_world[1],Q_world[1],R_world[1],S_world[1]])))

    return xmin, ymin, xmax - xmin, ymax - ymin

def get_panorama_boundary(img_center, img_all, H_all):
    # Calculate multiple images mapping to the selected frame
    # then obtain the overall boundary for all mapped images

    h, w, c = img_center.shape
    xmin = 0
    ymin = 0
    xmax = w
    ymax = h

    for i in range(len(img_all)):
```

```
        x, y, w, h = get_bounday(img_all[i], H_all[i])
        xmin = np.min([x, xmin])
        ymin = np.min([y, ymin])
        xmax = np.max([xmax, x + w])
        ymax = np.max([ymax, y + h])

    return xmin, ymin, xmax - xmin, ymax - ymin

def plot_panorama(panorama, img_ori, img_trans, H, x0, y0):
    # plot each transformed images in the synthetic panorama
    x, y, w, h = get_bounday(img_ori, H)

    for i in range(y-y0 , y-y0+h):
        for j in range(x-x0 , x-x0+w):

            if np.sum(panorama[i,j] == np.zeros(3)):
                panorama[i,j] = img_trans[i-(y-y0), j-(x-x0)]

            elif np.sum(img_trans[i-(y-y0), j-(x-x0)] == np.zeros(3)):
                panorama[i,j] = panorama[i,j]

            else:
                panorama[i,j] = (panorama[i,j] + img_trans[i-(y-y0), j-(x-x0)]) / 2




############## Main #################
```

```
epsilon = 0.9
delta = 10

directory = "/home/xu1363/Documents/ECE 661/hw5/input/"
file1 = "1.jpg"
file2 = "2.jpg"
file3 = "3.jpg"
file4 = "4.jpg"
file5 = "5.jpg"

img1 = cv2.imread(directory+file1,cv2.IMREAD_COLOR)
img2 = cv2.imread(directory+file2,cv2.IMREAD_COLOR)
img3 = cv2.imread(directory+file3,cv2.IMREAD_COLOR)
img4 = cv2.imread(directory+file4,cv2.IMREAD_COLOR)
img5 = cv2.imread(directory+file5,cv2.IMREAD_COLOR)

# resize the input image to a smaller size for acceleration
h = int(img1.shape[1]*0.5)
w = int(img1.shape[0]*0.5)
img1 = cv2.resize(img1, (h, w), interpolation = cv2.INTER_AREA)
img2 = cv2.resize(img2, (h, w), interpolation = cv2.INTER_AREA)
img3 = cv2.resize(img3, (h, w), interpolation = cv2.INTER_AREA)
img4 = cv2.resize(img4, (h, w), interpolation = cv2.INTER_AREA)
img5 = cv2.resize(img5, (h, w), interpolation = cv2.INTER_AREA)

H12 = get_homography_all(img1, img2)
H23 = get_homography_all(img2, img3)
H43 = get_homography_all(img4, img3)
H54 = get_homography_all(img5, img4)
```

```
H13 = np.dot(H23,H12)
H53 = np.dot(H54, H43)



plot_inliers_outliers(img1, img2)
plt.axis('off')
plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img12.jpeg")

plot_inliers_outliers(img2, img3)
plt.axis('off')
plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img23.jpeg")

plot_inliers_outliers(img3, img4)
plt.axis('off')
plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img34.jpeg")

plot_inliers_outliers(img4, img5)
plt.axis('off')
plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img45.jpeg")



img13 = mapping(img1, H13)
img23 = mapping(img2, H23)
img43 = mapping(img4, H43)
img53 = mapping(img5, H53)
```

```
img_all = [img1, img2, img4, img5]
H_all = [H13, H23, H43, H53]
x, y, w, h = get_panorama_boundary(img3, img_all, H_all)
panorama = np.zeros((h, w, 3), dtype = np.uint8)
panorama[ 0-y : 0-y+img3.shape[0] , 0-x : 0-x+img3.shape[0], :] = img3

plot_panorama(panorama, img1, img13, H13, x, y)
plot_panorama(panorama, img5, img53, H53, x, y)
plot_panorama(panorama, img2, img23, H23, x, y)
plot_panorama(panorama, img4, img43, H43, x, y)



plt.figure(dpi=1200)
plt.imshow(cv2.cvtColor(panorama, cv2.COLOR_BGR2RGB), cmap='jet')
plt.axis('off')
plt.savefig("/home/xu1363/Documents/ECE 661/hw5/panorama.jpeg")
```