ECE 661 – Homework 5

Ran Xu

xu943@purdue.edu

10/05/2018

1. Logic and math reductions

I use SIFT algorithm to extract key points from pairs of images and use SSD similarity metric to construct the initial set of correspondences. I set the set size as N = 100 pairs where they have the shortest Euclidian distance.

1.1 RANSAC algorithm

The RANSAC algorithm aims to distinguish the inlier set and the outlier set from the initial set of correspondences. The steps are shown as follows,

Firstly, random select n (n=6) pairs of correspondences from the initial set.

Secondly, use the Linear Least-Squared method as shown in Selection 1.2 to estimate the homography.

Thirdly, use this homography to examine the remaining set, a.k.a. N-n = 94 pairs of correspondence in the initial set. The way to do this is to map all the key points in the first image to the second image and check the x,y-bias from the corresponding key points in the second image. If the x-bias or the y-bias is larger than δ ($\delta = 15$ pixels, which is an empirical value), the correspondences pair is in the temporal outlier set, otherwise the correspondences pair is in the temporal inlier set. The size of the temporal inlier set and the homography are noted down to measure the amount of support for this homography.

Fourthly, after repeating first step to third step for M (M = 100) times, we select the homography with the most amount of support (largest temporal inlier set).

The rational for the number of experiments M = 100 is that suppose that 90% of the initial correspondence are inliers. Then the probability that the randomly selected n = 6 pairs of correspondences are all from inlier sets is $(90\%)^6 = 0.53$. Then with 0.99 confidence that we have at least one selection of the 6 pairs are all from inlier set, we must perform $M_{minimum} = \frac{\lg(1-0.99)}{\lg(1-0.53)} = 6.1$ experiments. The number of experiments M=100 we choose is far larger than the minimum number of experiments, guaranteeing high confidence that we can found at least one set of 6 pairs of correspondences from the inlier set.

Finally, we use this selected homography to distinguish inlier and outlier again, the inlier set now forms the final inlier set for the next fine-tuning step. This homography is also the initialization of the homography in the Non-Linear Least-Square method.

1.2 Linear Least-Square method to estimate homographies

Given a point $P = (x_1, y_1)$ or $(x_1, y_1, 1)$ in homogeneous coordinate (HC) representation in the first image, we use a homography H to map it to a point $P' = (x_1', y_1')$ in the second image. Due to homogenous property, we assume H as follows,

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

Then we have the HC of P'

$$P' = HP = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11}x_1 + h_{12}y_1 + h_{13} \\ h_{21}x_1 + h_{22}y_1 + h_{23} \\ h_{31}x_1 + h_{32}y_1 + 1 \end{bmatrix} = (h_{31}x_1 + h_{32}y_1 + 1) \begin{bmatrix} x_1' \\ y_1' \\ 1 \end{bmatrix}$$

It can be represented in the following way,

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \end{bmatrix}$$

Given 6 pairs of correspondences, a.k.a. 12 equations to be solved. We can re-write the equation as,

$$Ah = b$$

where A is a 12-by-8 matrix, b is the 2D coordinates in the second image, and h is the vector-encoded homography $h = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}]^T$. To solve the vector-encoded homography h with least square error, we have

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

1.3 Levenberg-Marquardt algorithm

Levenberg-Marquardt algorithm aims to get the fine-tuned homography between two images in the pruned inlier correspondence set. The homography we get from RANSAC algorithm serves as the initialization of this step. To formulate the problem, we construct a cost function of the l2 norm between the mapped coordinates and the true coordinate between the correspondence pair. Suppose the coordinate of the i-th correspondence pair are X_i and X_i' and the homography from X to X' is h. Then the optimization problem to find the fine-tuned homography is formulated by,

$$\mathrm{h}^* = \underset{h}{\operatorname{argmin}} \sum_i \left|\left| \mathrm{f}(\mathrm{X_i}, \vec{\mathrm{h}}) - \mathrm{X_i'} \right|\right|^2$$

where $\mathrm{f}(\mathrm{X_i}, \mathrm{h})$ is the mapped 2D coordinates from image 1 to image 2.

To solve this optimization problem, Levenberg-Marquardt algorithm combines the Gradient-Descent (GD) method and the Gauss-Newton (GN) method. The high-level intuition is that GD is guaranteed to find the optimal but slow and GN is fast but risky. So LM algorithm combines the two method by setting step progress as follows,

$$\overrightarrow{\delta_h} = \left(J_{\vec{f}}^T J_{\vec{f}} + \mu_k I\right)^{-1} J_{\vec{f}}^T \vec{\epsilon}\left(\overrightarrow{h_k}\right)$$

where $\overrightarrow{h_k}$ is the temporal optimal value of the homography in the k-th step, $\overrightarrow{\delta_h}$ is the step progress in the k-th step, $\vec{\epsilon}\left(\overrightarrow{h_k}\right)$ is the error function which is $\mathrm{X}' - \mathrm{f}(\mathrm{X}, \vec{\mathrm{h}})$, $J_{\vec{f}}$ is the Jacobian derivative of $\vec{f}$ with respect to $\vec{h}$, and finally μ_k is LM's damping coefficient in the k-th step.

Note that in each step k+1, if the cost function $\sum_i \left|\left|\mathrm{f}\left(\mathrm{X_i}, \overrightarrow{\mathrm{h_{k+1}}}\right) - \mathrm{X_i'}\right|\right|^2$ is higher than that in the k-th step. We set $\overrightarrow{h_k} = \overrightarrow{h_{k+1}}$ and $\mu_{k+1} = 2\mu_k$.

LM will stop when either the cost function drops below some threshold, or the step progress $\left|\left|\overrightarrow{\delta_h}\right|\right|$ drops below a threshold, or max iteration reaches.

1.4 image mosaicing method

In Section 1.1 to 1.3, I already get the homography mapping between adjacent images. Suppose the homography from image 1 to 2, 2 to 3, 3 to 4, and 4 to 5 are $h_{12}, h_{23}, h_{34}, h_{45}$. To transform all images to the coordinate in image 3, we compute the homography from image 1, 4 and 5 as follows,

$$h_{13} = h_{23} \cdot h_{12}$$

$$h_{43} = h_{34}^{-1}$$

$$h_{53} = h_{34}^{-1} \cdot h_{45}^{-1}$$

I build the mosaic image with the following steps,

Firstly, I compute the x, y-range of the mapped images, a.k.a. mapped image 1, mapped image 2, mapped image 3, mapped image 4, and mapped image 5, in the coordinate of image 3. Take the minimum of the x, y lower limit and maximum of the x, y higher limit to get the dimension of the final mosaic image.

Secondly, in the final mosaic image, we must compute the values of each pixel. The way to do this is to map each pixel in the final mosaic image to the coordinate of each original image. If the coordinate in the original image is within the x,y range of the original image, this means the original image captures a pixel in the final mosaic image. To conclude, I check the 5 original image if they captures a pixel in the final mosaic image. If none captures, a black pixel will show up. Otherwise, I average the pixels from all original images that capture that pixel.

Note that the mapped coordinates in the original image from the integer coordinates of that in image 3 may not be a integer any more. If that happens, I just round the mapped coordinates to integer.

2. Results

2.1 The extracted correspondences between sets of adjacent images

There are 4 sets of adjacent images. I show the extracted correspondences in Figure 1, 2, 3 and 4. It can be seen that most of the correspondences are correct and should be in the inlier set and however, some correspondences are erroneous.

Figure 1: Extracted correspondences between the 1st image and the 2nd image

Figure 2: Extracted correspondences between the 1st image and the 2nd image

Figure 3: Extracted correspondences between the 1st image and the 2nd image

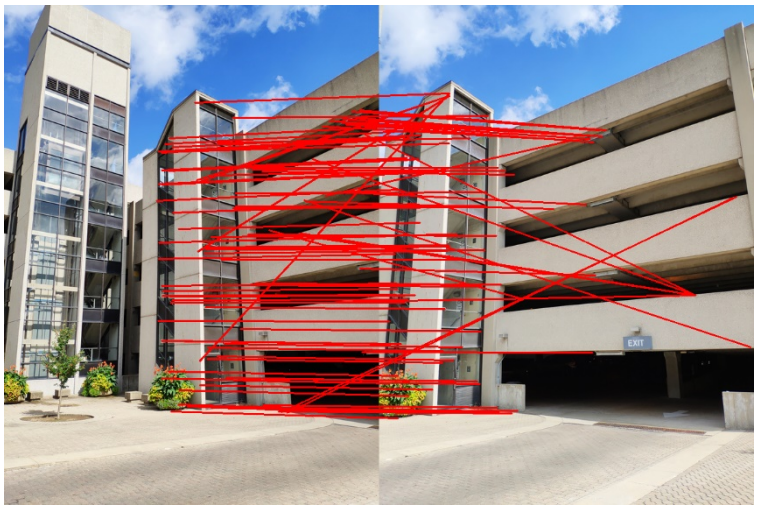

Figure 4: Extracted correspondences between the 1st image and the 2nd image

2.2 The outliers and selected inliers

I show the inlier set (marked in read) and the outlier set (marked in green) in Figure 5 and 6. They are the extracted correspondence between 2nd image and 3rd image and between 3rd image and 4th image. In Figure 5, the initial set is cleaner and 95 of the 100 initial correspondence belongs to the selected inlier set. However, in Figure 6, only 88 of the 100 initial correspondence belongs to the selected inlier set. Finally, in both case, the selected inlier set in RANSAC algorithm is reliable to perform the fine-tuning LM algorithm.

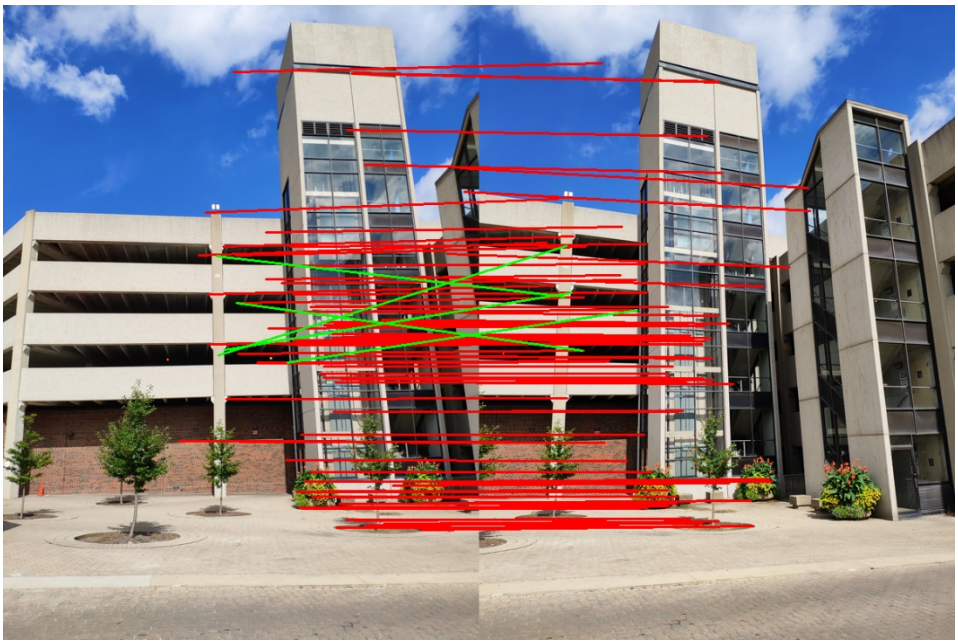

Figure 5: Outlier and selected inlier correspondences between the 2nd image and the 3rd image

Figure 6: Outlier and selected inlier correspondences between the 3rd image and the 4th image

2.3 The final output mosaic

The final output mosaic is shown in Figure 7. It almost perfectly maps all images to one coordinate plane. However, due to the slight luminance change in different images, we can see the boundary of each individual image in the mosaic image.

Figure 7: Final output mosaic

3. The parameters

Matching SIFT key points	Similarity metric	SSD
	Number of initial correspondences	100 (pairs)
RANSAC algorithm	Number of correspondences to find homography in Linear Least-Square Solution	6 (pairs)
	Maximum tolerant error in range plan δ	15 (pixels)
	Number of experiments tried	100

4. Source code

```
import numpy as np
import scipy.optimize
import cv2 as cv
import time
def SSD(f1, f2):
    return np.sqrt(np.sum((f1-f2)**2))
def Correspondence(FM1, FM2, kp1, kp2): #(#,128), (##,128),
    N1, N2 = (FM1.shape[0], FM2.shape[0])
    Pairs, Scores = ([], [])
    for idx1 in range(N1):
        Score = np.array([SSD(FM1[idx1,:], FM2[idx2,:]) for idx2 in range(N2)])
        idx2 = np.argmin(Score)
        Scores.append(np.min(Score))
        Pairs.append((idx1,idx2))
    # Select 100 pairs to eliminate outliers
    BestPairs = [x for _,x in sorted(zip(Scores,Pairs))]  # Incresing order
    SelectedPairs = BestPairs[0:int(np.min([100,len(BestPairs)]))]
    Corres = np.zeros((len(SelectedPairs), 4))
    Corres[:,0] = np.array([kp1[idx1].pt[1] for idx1, _ in SelectedPairs])  # h1
    Corres[:,1] = np.array([kp1[idx1].pt[0] for idx1, _ in SelectedPairs])  # w1
    Corres[:,2] = np.array([kp2[idx2].pt[1] for _, idx2 in SelectedPairs])  # h2
    Corres[:,3] = np.array([kp2[idx2].pt[0] for _, idx2 in SelectedPairs])  # w2
    return SelectedPairs, Corres
def DrawPairs(input_image1, input_image2, Coord1, Coord2, Corres):
    left = cv.imread("Pics/%s.png" %input_image1)    # (800, 600, 3) --> (H, W, 3)
    right = cv.imread("Pics/%s.png" %input_image2)   # (800, 600, 3)
    output = np.concatenate((left, right), axis = 1) # (800, 1200, 3)
    for idx in range(Corres.shape[0]):
        w1 = int(Corres[idx,1])   # This is the vertial, height direction
        h1 = int(Corres[idx,0])   # This is the horizontal, height direction
        w2 = int(Corres[idx,3] + left.shape[1])
        h2 = int(Corres[idx,2])
        cv.line(output, (w1,h1),(w2,h2), (0,0,255), 2)  # cv plot (w,h)
    cv.imwrite("Coores_%s_%s.png" %(input_image1,input_image2), output)
def DrawInOutPairs(input_image1, input_image2, InlierCorres, OutlierCorres): # (n,4)
    left = cv.imread("Pics/%s.png" %input_image1)    # (800, 600, 3) --> (H, W, 3)
    right = cv.imread("Pics/%s.png" %input_image2)   # (800, 600, 3)
    output = np.concatenate((left, right), axis = 1) # (800, 1200, 3)
    for idx in range(InlierCorres.shape[0]):
        w1 = int(InlierCorres[idx,1])   # This is the vertial, height direction
        h1 = int(InlierCorres[idx,0])   # This is the horizontal, height direction
        w2 = int(InlierCorres[idx,3] + left.shape[1])
        h2 = int(InlierCorres[idx,2])
        cv.line(output, (w1,h1),(w2,h2), (0,0,255), 2)  # cv plot (w,h)
    for idx in range(OutlierCorres.shape[0]):
        w1 = int(OutlierCorres[idx,1])   # This is the vertial, height direction
        h1 = int(OutlierCorres[idx,0])   # This is the horizontal, height direction
        w2 = int(OutlierCorres[idx,3] + left.shape[1])
        h2 = int(OutlierCorres[idx,2])
        cv.line(output, (w1,h1),(w2,h2), (0,255,0), 2)  # cv plot (w,h)
    cv.imwrite("InOutCoores_%s_%s.png" %(input_image1,input_image2), output)
    print("Inlier: %d pairs. Outlier: %d pairs." %(InlierCorres.shape[0], OutlierCorre
s.shape[0]))
def DrawPoints(imgname, kp, switch = [0]):
    if len(switch) == 1:
        switch = [x for x in range(len(kp))]
    img = cv.imread("Pics/%s.png" %imgname)
    for idx in switch:
        # Draw (h,w) = kp[idx].pt[0], kp[idx].pt[1]
        cv.circle(img,(int(kp[idx].pt[0]), int(kp[idx].pt[1])),
                  2, (0,0,255), -1)  # cv draw (w,h)
    cv.imwrite("KP_%s.png" %(imgname), img)
```

```
def LLS(hw):  # 6 x 4 -- (h1,w1,h2,w2)
    Npairs = hw.shape[0]
    Projection = np.zeros((Npairs*2, 1))    # 12 x 1
    Coefficient = np.zeros((Npairs*2, 8))  # 12 x 8
    for index in range(Npairs):
        Projection[index*2, 0] = hw[index,2]    # x2
        Projection[index*2+1, 0] = hw[index,3]  # y2
        Coefficient[index*2, 0] = hw[index,0]  # x1
        Coefficient[index*2, 1] = hw[index,1]  # y1
        Coefficient[index*2, 2] = 1
        Coefficient[index*2, 6] = - hw[index,0] * hw[index,2]  # -x1*x2
        Coefficient[index*2, 7] = - hw[index,1] * hw[index,2]  # -y1*x2
        Coefficient[index*2+1, 3] = hw[index,0]  # x1
        Coefficient[index*2+1, 4] = hw[index,1]  # y1
        Coefficient[index*2+1, 5] = 1
        Coefficient[index*2+1, 6] = - hw[index,0] * hw[index,3]    # -x1*y2
        Coefficient[index*2+1, 7] = - hw[index,1] * hw[index,3]    # -y1*y2
    h = np.matmul(np.linalg.pinv(Coefficient), Projection)
    Homograhy = np.array(([h[0][0], h[1][0], h[2][0]],
                          [h[3][0], h[4][0], h[5][0]],
                          [h[6][0], h[7][0], 1]))
    return Homograhy
def DetInOut(Val_corrs, H, delta):
    # Use H to examine the in-, out- in Val_corrs
    InSize, InList = (0, [])
    for idx in range(Val_corrs.shape[0]):
        x1, y1, x2, y2 = (Val_corrs[idx,0], Val_corrs[idx,1],
                          Val_corrs[idx,2], Val_corrs[idx,3])
        HC1 = np.array([[x1],[y1],[1]])
        Mapped_HC1 = np.matmul(H, HC1)
        (mx1, my1) = (Mapped_HC1[0,0]/Mapped_HC1[2,0],
                      Mapped_HC1[1,0]/Mapped_HC1[2,0])
        if abs(mx1-x2)<=delta and abs(my1-y2)<=delta:
            InSize = InSize + 1
            InList.append(idx)
    return InSize, InList
def Mapping2D(H, Pts): # Pts is 2D numpy array (2,n)
    Pts3D = np.vstack([Pts, np.ones((1, Pts.shape[1]))])
    MappedPts3D = np.matmul(H, Pts3D)
    MappedPts2D = MappedPts3D[0:2, :] / MappedPts3D[2, :]
    return MappedPts2D
def Concatenate(img1, img2, H):
    # Step 1: Find out the shape of concat_img
    Pts = np.array([[0, 0, img1.shape[0]-1, img1.shape[0]-1],
                    [0, img1.shape[1]-1, 0, img1.shape[1]-1]])
    MappedPts = Mapping2D(H, Pts)
    minx1, miny1 = (np.min(MappedPts[0,:]), np.min(MappedPts[1,:]))
    maxx1, maxy1 = (np.max(MappedPts[0,:]), np.max(MappedPts[1,:]))
    minx2, miny2 = (0, 0)
    maxx2, maxy2 = (img2.shape[0]-1, img2.shape[1]-1)
    minx, miny = (int(min(minx1, minx2)), int(min(miny1, miny2)))
    maxx, maxy = (int(max(maxx1, maxx2)), int(max(maxy1, maxy2)))
    NewHeight, NewWidth = (maxx - minx + 1, maxy - miny + 1)

    # Step 2: Construct the mapped_img1 and mapped_img2 in new coordinate
    # The new corrdinate origins at (minx, miny) in img2.corrdinate
    MappedImg1 = np.zeros((NewHeight, NewWidth, 3))
    Weight1 = np.zeros((NewHeight, NewWidth, 3))
    MappedImg2 = np.zeros((NewHeight, NewWidth, 3))
    Weight2 = np.zeros((NewHeight, NewWidth, 3))
    # Map img2 to new coordinate
    MappedImg2[-minx : -minx+img2.shape[0], -miny : -miny+img2.shape[1], :] = img2
    Weight2[-minx : -minx+img2.shape[0], -miny : -miny+img2.shape[1], :] = 1
```

```
    # Map img1 to new coordinate, using inverse mapping strategy
    H_inv = np.linalg.inv(H)
    for MappedBiasedX in range(NewHeight):
        for MappedBiasedY in range(NewWidth):
            MappedPts = np.array([[MappedBiasedX+minx], [MappedBiasedY+miny]])
            Pts = Mapping2D(H_inv, MappedPts)  # Back to img1.coordinate
            X, Y = (int(Pts[0,0]), int(Pts[1,0]))
            if(X>=0 and Y>=0 and X<img1.shape[0] and Y<img1.shape[1]):
                MappedImg2[MappedBiasedX, MappedBiasedY,:] = img1[X,Y,:]
                Weight2[MappedBiasedX, MappedBiasedY, :] = 1

    # Step 3: Finally, concatenate!
    MappedImg = np.zeros((NewHeight, NewWidth, 3))
    for MappedBiasedX in range(NewHeight):
        for MappedBiasedY in range(NewWidth):
            if(Weight1[MappedBiasedX,MappedBiasedY,0] == 0 and
               Weight2[MappedBiasedX,MappedBiasedY,0] == 0):
                MappedImg[MappedBiasedX,MappedBiasedY,:] = 0
            else:
                MappedImg[MappedBiasedX,MappedBiasedY,:] = \
                    (MappedImg1[MappedBiasedX, MappedBiasedY,:] * \
                     Weight1[MappedBiasedX,MappedBiasedY,:] + \
                     MappedImg2[MappedBiasedX, MappedBiasedY,:] * \
                     Weight2[MappedBiasedX,MappedBiasedY,:]) / \
                    (Weight1[MappedBiasedX,MappedBiasedY,:] + \
                     Weight2[MappedBiasedX,MappedBiasedY,:])
    return MappedImg
def ConcatenateAll(img_list, H_list): # 5 images and 5 Hs, all mapping to img_middle
    # Step 1: Find out the shape of concat_img in img_list[2].coordinate
    minx, miny = (0, 0)
    maxx, maxy = (img_list[2].shape[0]-1, img_list[2].shape[1]-1)
    minx_img, miny_img = (np.zeros((5,)).astype("int"), np.zeros((5,)).astype("int"))
    maxx_img, maxy_img = (np.zeros((5,)).astype("int"), np.zeros((5,)).astype("int"))
    for idx, (img, H) in enumerate(zip(img_list, H_list)):
        Pts = np.array([[0, 0, img.shape[0]-1, img.shape[0]-1],
                        [0, img.shape[1]-1, 0, img.shape[1]-1]])
        MappedPts = Mapping2D(H, Pts)
        minx_img[idx], miny_img[idx] = (int(np.min(MappedPts[0,:])), int(np.min(Mapped
Pts[1,:])))
        maxx_img[idx], maxy_img[idx] = (int(np.max(MappedPts[0,:])), int(np.max(Mapped
Pts[1,:])))
        minx, miny = (int(min(minx_img[idx], minx)), int(min(miny_img[idx], miny)))
        maxx, maxy = (int(max(maxx_img[idx], maxx)), int(max(maxy_img[idx], maxy)))
    NewHeight, NewWidth = (maxx - minx + 1, maxy - miny + 1)

    # Step 2: Construct the MappedImg_list in img_list[2].coordinate
    # The new corrdinate origins at (minx, miny) in img2.corrdinate
    MappedImg_list, Weight_list = ([], [])
    for idx, (img, H) in enumerate(zip(img_list, H_list)):
        MappedImg = np.zeros((NewHeight, NewWidth, 3))
        Weight = np.zeros((NewHeight, NewWidth, 3))
        # Map img to new coordinate, using inverse mapping strategy
        H_inv = np.linalg.inv(H)
        for MappedBiasedX in range(minx_img[idx] - minx, maxx_img[idx] - minx):
            if MappedBiasedX % 300 == 0:
                print MappedBiasedX,
            for MappedBiasedY in range(miny_img[idx] - miny, maxy_img[idx] - miny):
                MappedPts = np.array([[MappedBiasedX+minx], [MappedBiasedY+miny]])
                Pts = Mapping2D(H_inv, MappedPts)  # Back to img1.coordinate
                X, Y = (int(Pts[0,0]), int(Pts[1,0]))
                if(X>=0 and Y>=0 and X<img.shape[0] and Y<img.shape[1]):
                    MappedImg[MappedBiasedX, MappedBiasedY,:] = img[X,Y,:]
                    Weight[MappedBiasedX, MappedBiasedY, :] = 1
```

```
        MappedImg_list.append(MappedImg)
        Weight_list.append(Weight)

    # Step 3: Finally, concatenate!
    MappedImg = np.zeros((NewHeight, NewWidth, 3))
    for MappedBiasedX in range(NewHeight):
        if MappedBiasedX % 300 == 0:
            print MappedBiasedX,
        for MappedBiasedY in range(NewWidth):
            SummedWeight = (Weight_list[0][MappedBiasedX,MappedBiasedY,:] + \
                    Weight_list[1][MappedBiasedX,MappedBiasedY,:] + \
                    Weight_list[2][MappedBiasedX,MappedBiasedY,:] + \
                    Weight_list[3][MappedBiasedX,MappedBiasedY,:] + \
                    Weight_list[4][MappedBiasedX,MappedBiasedY,:])
            if(SummedWeight[-1]==0):
                MappedImg[MappedBiasedX,MappedBiasedY,:] = 0
            else:
                MappedImg[MappedBiasedX,MappedBiasedY,:] = \
                    (MappedImg_list[0][MappedBiasedX, MappedBiasedY,:] * \
                     Weight_list[0][MappedBiasedX,MappedBiasedY,:] + \
                     MappedImg_list[1][MappedBiasedX, MappedBiasedY,:] * \
                     Weight_list[1][MappedBiasedX,MappedBiasedY,:] + \
                     MappedImg_list[2][MappedBiasedX, MappedBiasedY,:] * \
                     Weight_list[2][MappedBiasedX,MappedBiasedY,:] + \
                     MappedImg_list[3][MappedBiasedX, MappedBiasedY,:] * \
                     Weight_list[3][MappedBiasedX,MappedBiasedY,:] + \
                     MappedImg_list[4][MappedBiasedX, MappedBiasedY,:] * \
                     Weight_list[4][MappedBiasedX,MappedBiasedY,:]) / SummedWeight
    return MappedImg
def Loss_Func(h, Corres):  # Corres in (n, 4)
    Homo = np.array([[h[0], h[1], h[2]], [h[3], h[4], h[5]], [h[6], h[7], 1]])
    X = Corres[:,0:2]
    X_GT = Corres[:,2:4]
    X_Transpose = np.transpose(X)
    X3D = np.ones((3, Corres.shape[0]))
    X3D[0:2, :] = X_Transpose
    MappedX3D = np.matmul(Homo, X3D)
    MappedX2D_T = np.transpose(MappedX3D[0:2, :]/MappedX3D[2,:])
    return (MappedX2D_T - X_GT).flatten()
```

```
time1 = time.time()
NLLS = 1
sift = cv.xfeatures2d.SIFT_create()
input_image1, input_image2 = ("2", "3")
# Input
img = cv.imread('Pics/%s.png' %input_image1)
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
# SIFT Feature
kp1, des1 = sift.detectAndCompute(gray,None)  # kp are all in (w,h) order
DrawPoints(input_image1, kp1)

# Input
img = cv.imread('Pics/%s.png' %input_image2)
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
# SIFT Feature
kp2, des2 = sift.detectAndCompute(gray,None)
DrawPoints(input_image2, kp2)

# Draw Correspondence
Pairs, Corres = Correspondence(des1, des2, kp1, kp2)
DrawPairs(input_image1, input_image2, kp1, kp2, Corres)
```

```
# 100 pairs between (Corres[:,0], Corres[:,1]) and (Corres[:,2], Corres[:,3])
delta = 15  # Maximum tolerant x and y bias in the range plane to disguish inlier and
outlier
Nexp = 100  # Number of experiment we take
Ncorrs = 6  # Number of correspondence pairs to compute homography
# np.random.seed(0)
SaveList = []
for idx_exp in range(Nexp):
    # Randomly select $Ncorrs pairs
    idx = np.random.choice(100, Ncorrs, replace=False)  # Select Ncoors from 100 pairs
    Selected_corrs = Corres[idx,:]
    Val_corrs = np.delete(Corres, idx, axis=0)

    # Get homography using LLS method
    H = LLS(Selected_corrs)

    # Determine in-out and note down size of inlier set
    InSize, _ = DetInOut(Val_corrs, H, delta)
    SaveList.append((InSize, H))

# Choose the highest-accepted LLS homography to determine in-, out- list
SortedList = sorted(SaveList, key=lambda pair: pair[0], reverse = True)
TmpHomography = SortedList[0][1]
InSize, InList = DetInOut(Corres, TmpHomography, delta)
InlierCorres = Corres[np.array(InList), :]
OutlierCorres = np.delete(Corres, InList, axis=0)

# Initialize it using LLS
H_LLS = LLS(InlierCorres)
print("H_LLS = ", H_LLS)

# Fine-tune it using LM(Non-LLS)
if NLLS:
    h_init = [H_LLS[0][0], H_LLS[0][1], H_LLS[0][2], H_LLS[1][0],
              H_LLS[1][1], H_LLS[1][2], H_LLS[2][0], H_LLS[2][1]]
    sol = scipy.optimize.least_squares(Loss_Func, h_init, method = 'lm', args = [Inlie
rCorres])
    H_LLS = np.array([[sol.x[0], sol.x[1], sol.x[2]],
                      [sol.x[3], sol.x[4], sol.x[5]],
                      [sol.x[6], sol.x[7], 1]])
    print("H_NLLS = ", H_LLS)

DrawInOutPairs(input_image1, input_image2, InlierCorres, OutlierCorres)
# Concatenate
img1 = cv.imread('Pics/%s.png' %input_image1)
img2 = cv.imread('Pics/%s.png' %input_image2)
img_concat = Concatenate(img1, img2, H_LLS)  # Concat img1 to img2 using H
cv.imwrite("Concat_%s_%s.png" %(input_image1, input_image2), img_concat)
time2 = time.time()
print("Execution time = %.1fs" %(time2-time1))
```

```
time1 = time.time()
NLLS = 1
sift = cv.xfeatures2d.SIFT_create()
image_pairs = [("1", "2"), ("2", "3"), ("3", "4"), ("4", "5")]
H_LLS_initlist = []
for input_image1, input_image2 in image_pairs:
    # Input
    img = cv.imread('Pics/%s.png' %input_image1)
    gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    # SIFT Feature
    kp1, des1 = sift.detectAndCompute(gray,None)  # kp are all in (w,h) order
```

```
    DrawPoints(input_image1, kp1)
    # Input
    img = cv.imread('Pics/%s.png' %input_image2)
    gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    # SIFT Feature
    kp2, des2 = sift.detectAndCompute(gray,None)
    DrawPoints(input_image2, kp2)
    # Draw Correspondence
    Pairs, Corres = Correspondence(des1, des2, kp1, kp2)
    DrawPairs(input_image1, input_image2, kp1, kp2, Corres)

    # 100 pairs between (Corres[:,0], Corres[:,1]) and (Corres[:,2], Corres[:,3])
    delta = 15  # Maximum tolerant x and y bias in the range plane to disguish inlier
and outlier
    Nexp = 100  # Number of experiment we take
    Ncorrs = 6  # Number of correspondence pairs to compute homography
    # np.random.seed(0)
    SaveList = []
    for idx_exp in range(Nexp):
        # Randomly select $Ncorrs pairs
        idx = np.random.choice(100, Ncorrs, replace=False)  # Select Ncoors from 100 p
airs
        Selected_corrs = Corres[idx,:]
        Val_corrs = np.delete(Corres, idx, axis=0)

        # Get homography using LLS method
        H = LLS(Selected_corrs)

        # Determine in-out and note down size of inlier set
        InSize, _ = DetInOut(Val_corrs, H, delta)
        SaveList.append((InSize, H))

    # Choose the highest-accepted LLS homography to determine in-, out- list
    SortedList = sorted(SaveList, key=lambda pair: pair[0], reverse = True)
    TmpHomography = SortedList[0][1]
    InSize, InList = DetInOut(Corres, TmpHomography, delta)
    InlierCorres = Corres[np.array(InList), :]

    # Initialize it using LLS
    H_LLS = LLS(InlierCorres)

    # Fine-tune it using LM(Non-LLS)
    if NLLS:
        h_init = [H_LLS[0][0], H_LLS[0][1], H_LLS[0][2], H_LLS[1][0],
                  H_LLS[1][1], H_LLS[1][2], H_LLS[2][0], H_LLS[2][1]]
        sol = scipy.optimize.least_squares(Loss_Func, h_init, method = 'lm', args = [I
nlierCorres])
        H_LLS = np.array([[sol.x[0], sol.x[1], sol.x[2]],
                          [sol.x[3], sol.x[4], sol.x[5]],
                          [sol.x[6], sol.x[7], 1]])
        H_LLS_initlist.append(H_LLS)
        print("H_NLLS = ", H_LLS)
    else:
        H_LLS_initlist.append(H_LLS)
        print("H_LLS = ", H_LLS)

# Concatenate
img_list, H_list = ([], [])
input_image_list = ["1", "2", "3", "4", "5"]
for input_image in input_image_list:
    img = cv.imread('Pics/%s.png' %input_image)
    img_list.append(img)
H_list.append(np.matmul(H_LLS_initlist[1], H_LLS_initlist[0]))
```

```
H_list.append(H_LLS_initlist[1])
H_list.append(np.identity(3))
H_list.append(np.linalg.inv(H_LLS_initlist[2]))
H_list.append(np.matmul(np.linalg.inv(H_LLS_initlist[2]), np.linalg.inv(H_LLS_initlist
[3])))
img_concat = ConcatenateAll(img_list, H_list) # 5 images and 5 Hs, all mapping to img_
middle
if NLLS:
    cv.imwrite("ConcatAll_NLLS.png", img_concat)
else:
    cv.imwrite("ConcatAll_LLS.png", img_concat)
time2 = time.time()
print("\nExecution time = %.1fs" %(time2-time1))
```