Theory Question

1. 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?

    - RANSAC differentiates inliers and outliers in the following way:
        - First, n randomly selected correspondence pairs (x, x') are used to estimate a homography. n is usually between 4 to 10.
        - The computed homography is then applied to the list of all correspondences keypoints in image 1 (domain) to find the its corresponding pixel y in image 2 (range).
        - If the distance between x' (computed from SIFT) and y (computed from the estimated homography) is less than a predefined threshold, then the pair (x,x') are considered an inlier. If the distance is more than the threshold, the pair is considered an outlier.
        - This process is performed N times, and for each iteration, the most number of inliers set is taken as the inlier set.

2. 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.

    - GD is numerically stable but performs slowly as you get closer to the minimum point. This is because the step size reduces with each step towards the minimum point. On the hand, GN is much faster, given that you are already close to the minimum point. If you are far, GN cannot guarantee to reach the minimum point and could overstep. LM combines the two method using a control parameter μ. μ determines which method GD or GN influences the final answer depending on the distance from the minimum point. As it gets closer to minimum point, the μ decreases and this allows LM to act more like GN.

Task 1

For this homework, a panorama was produced consisting of the five images from fig. 1.

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

Figure 1: Input Images

Background

Brief Summary of the process:

1. Correspondences are computed using SIFT
2. RANSAC is used to separate the correspondences into inliers and outliers.
3. The inliers and outliers are drawn to visualize on the pair of images.
4. Using the inliers, the homography is estimated using least squares.
5. To further refine the homography, Levenberg-Marquardt (LM) least square method is used.
6. Using the homographies produced from the Linear Least squares and from the non-linear least squares (LM), the panorama is stitched together.

SIFT Correspondences: SIFT is used to compute the correspondences. The number of features extracted from SIFT was limited to 5000.

RANSAC:
Parameters:

- $\sigma = 2$
- $\epsilon = 0.75$
- $\delta = 3 * \sigma = 6$

- $n = 6$
- $p = 0.99$
- N = $\frac{ln(1-p)}{1-(1-\epsilon)^n}$
- M = $(1-\epsilon) * n_{total}$

Random n sets of correspondences were selected from the list of all the correspondences computed from SIFT. This set was then used to estimate a homography. This homography was applied to all initial keypoints in the domain image (img0) to project their corresponding keypoints on the range image (img1). The projected keypoints are then compared to the keypoint correspondences found from SIFT on the range image. If the distance between the projected keypoint and the SIFT computed keypoint is less than δ then it is considered an inlier. This whole process was repeated N times, and for each iteration, the maximum number of inliers collected are classified as the inliers of the correspondences. The rest of the correspondences become the outliers.

Least Squares Estimation: To calculate the Homography between a point x in the domain space and a point x' in the range space, the following matrix equation was simplified and solved simultaneously:

$$x' = \mathrm{H}x$$

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

$$x'_1 = h_{11}x_1 + h_{12}x_2 + h_{13}x_3$$

$$x'_2 = h_{21}x_1 + h_{22}x_2 + h_{23}x_3$$

$$x'_3 = h_{31}x_1 + h_{32}x_2 + x_3$$

Dividing the first two equations by the third equation

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

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

Simplifying the equations to give following matrix:

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -xx' & -yx' \\ 0 & 0 & 0 & x & y & 1 & -xy' & -yy' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Since all the inliers are being used to estimate the H, the system is over-determined and hence the linear least square estimate is calculated using the pseudo-inverse:

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

.

Non-linear least squares using the Levenberg-Marquardt method: The LM algorithm computes the nonlinear least squares estimation of the homographies. It does this by reducing the error in the cost function, $C(p) = ||X - f_p||^2$. X is the range coordinates, f_p is the nonlinear equation depending on p, the elements of the homographies. With an initial estimation of p, the algorithm converges to the global minimum. It combines the numerical stability of Gradient Descent and the efficiency of the Gauss-Newton Method for computing the least squares estimate.

Observations and Implementation notes:

- The LM method refines the homography but the change is so small that not much change is seen.
- Some of the inliers were incorrectly categorized as outliers and this changed as the RANSAC parameters were changed.
- Increasing ϵ caused the program to take longer to run, but gave better inliers and outliers. To get a good balance between runtime and quality of inliers and outliers, ϵ was set to 0.75.
- Similarly, n was chosen to be 6 rather than 4. This gave a better estimate of H and therefore a better quality of inliers set.
- Since I was using 5000 correspondences, for a better visual, only 250 were drawn on the images. The numbers of inliers and outliers was also limited to 350.
- The middle image was used as the reference image, and each image was then transformed to this reference with multiplying the homographies.

Outputs

(a) Correspondences

Figure 2: Correspondences between Images 0 and 1

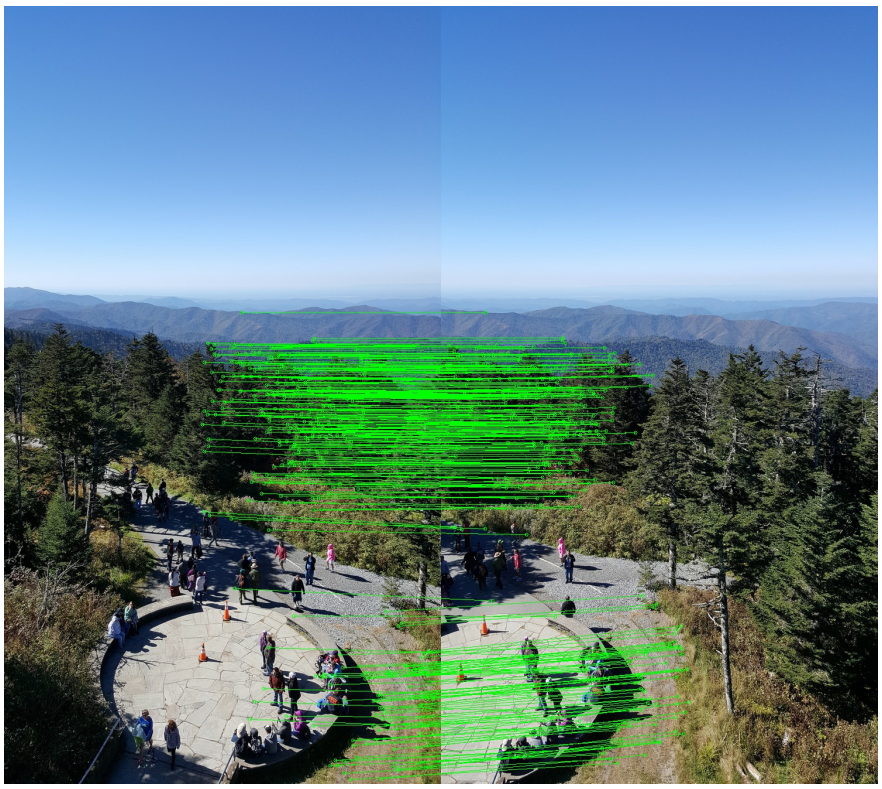
(a) Inliers

(b) Outliers

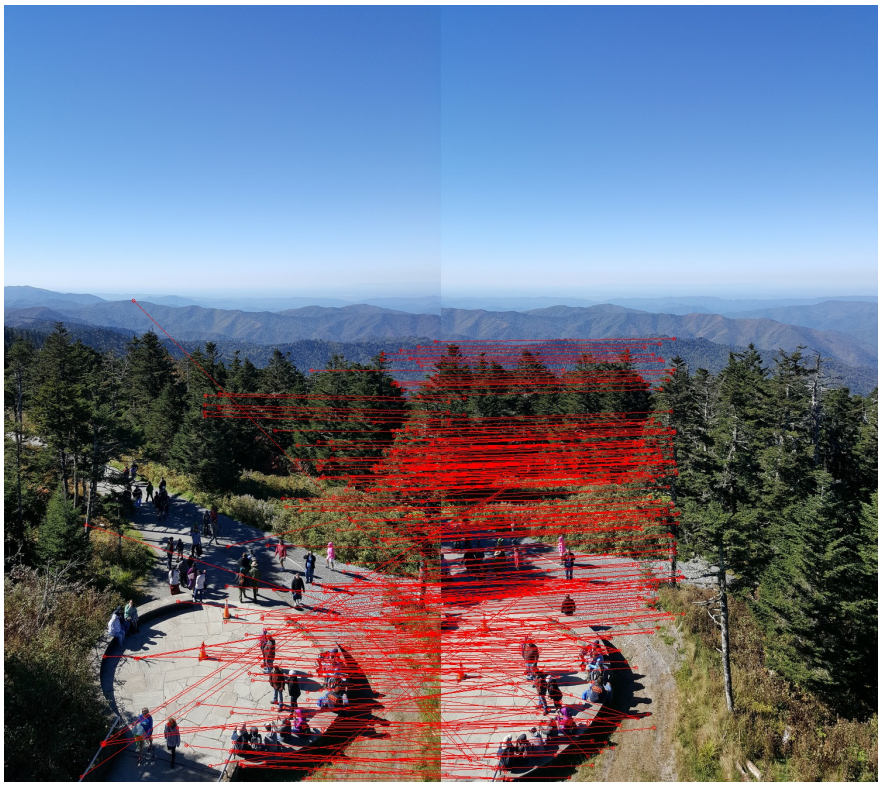
Figure 3: Inliers and Outliers for Image 0 and 1

(a) Correspondences

Figure 4: Correspondences between Images 1 and 2



(a) Inliers

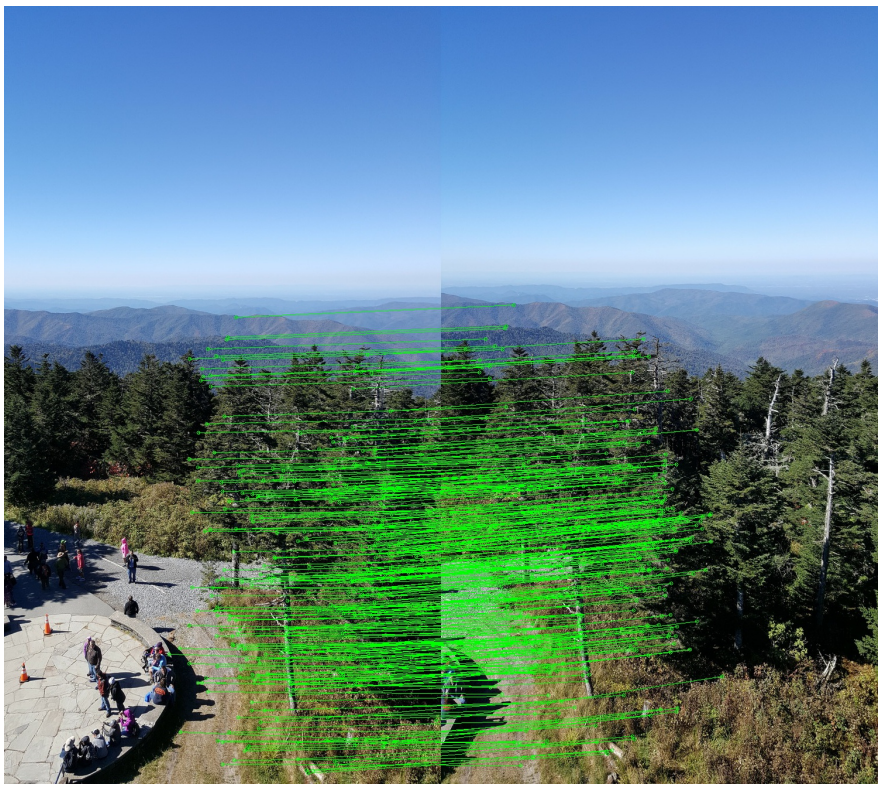


(b) Outliers

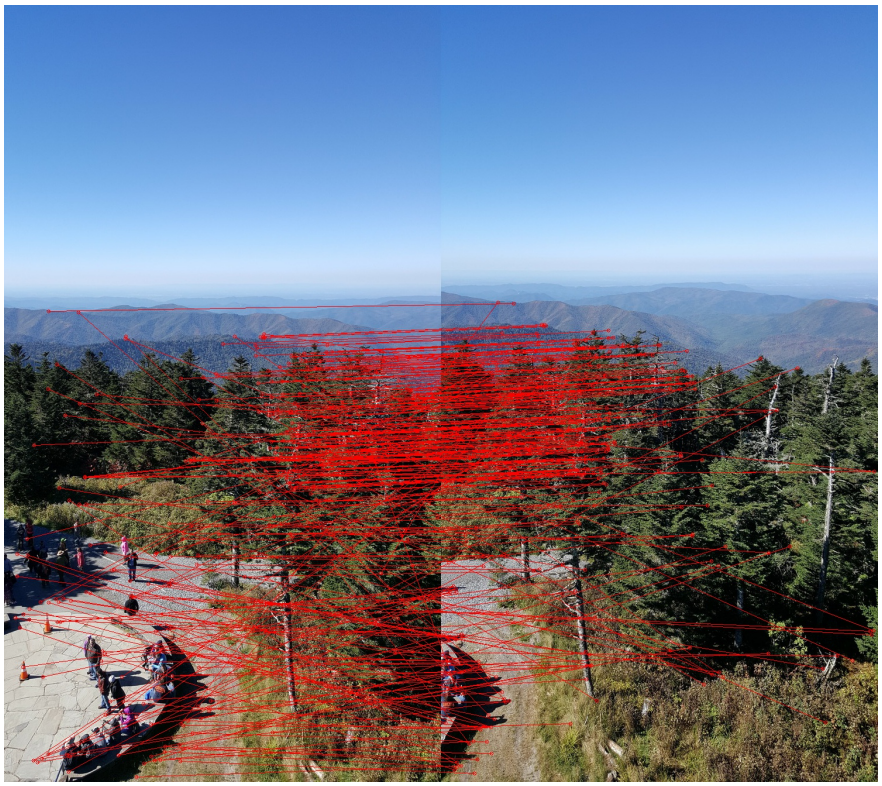
Figure 5: Inliers and Outliers for Image 1 and 2

(a) Correspondences

Figure 6: Correspondences between Images 2 and 3

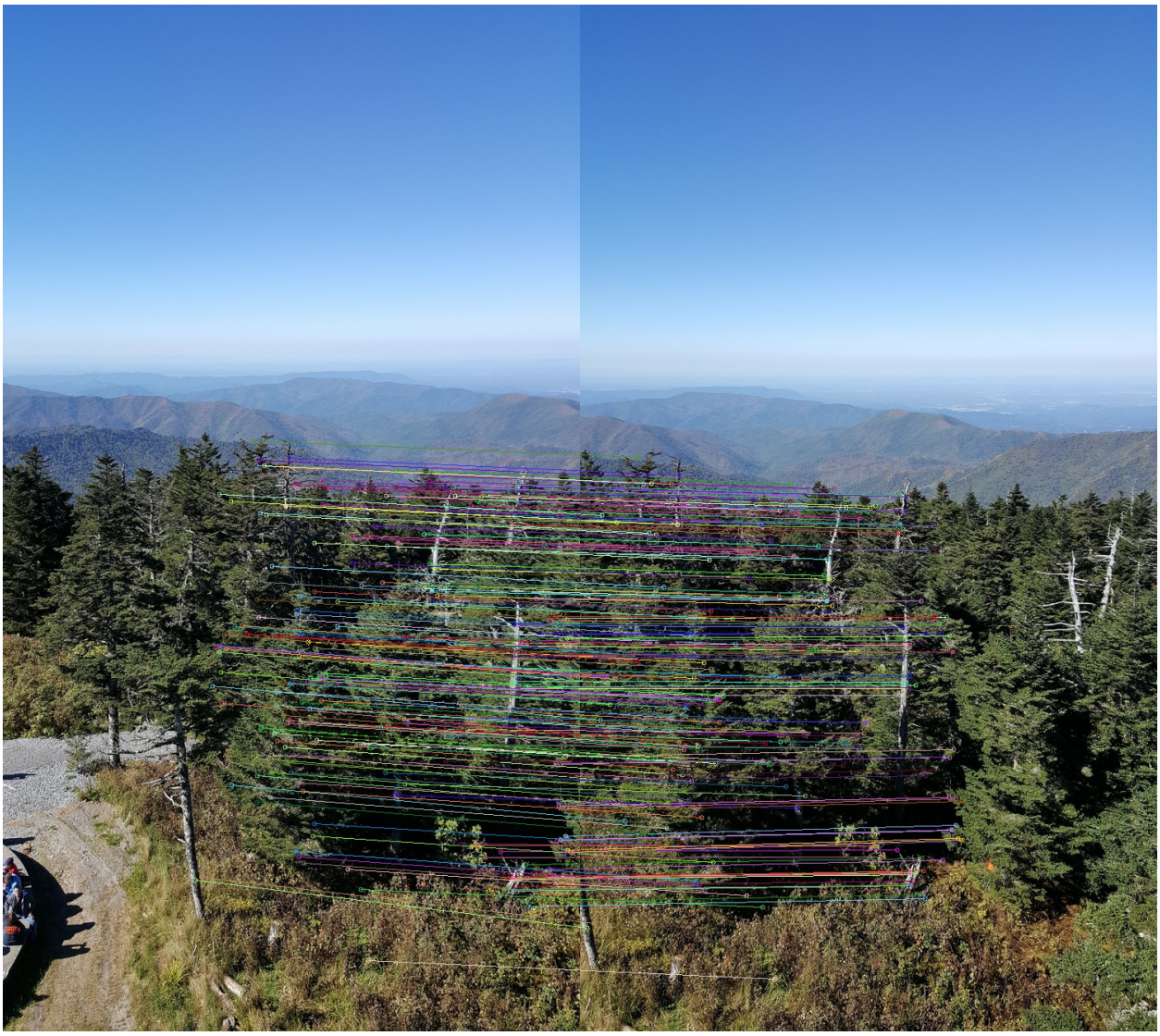


(a) Inliers



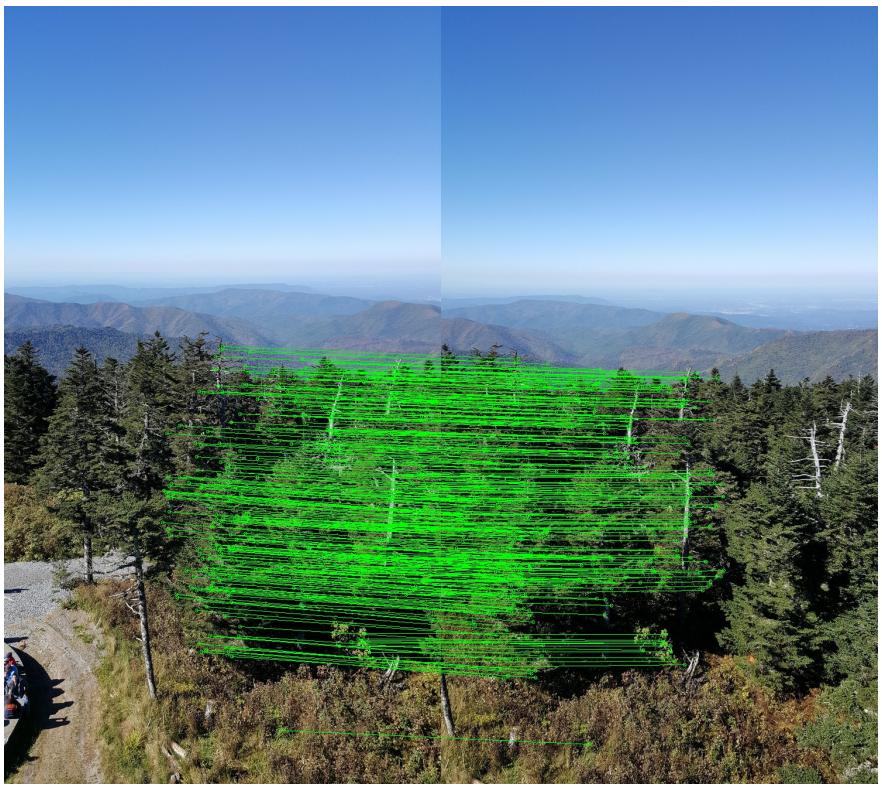
(b) Outliers

Figure 7: Inliers and Outliers for Image 2 and 3

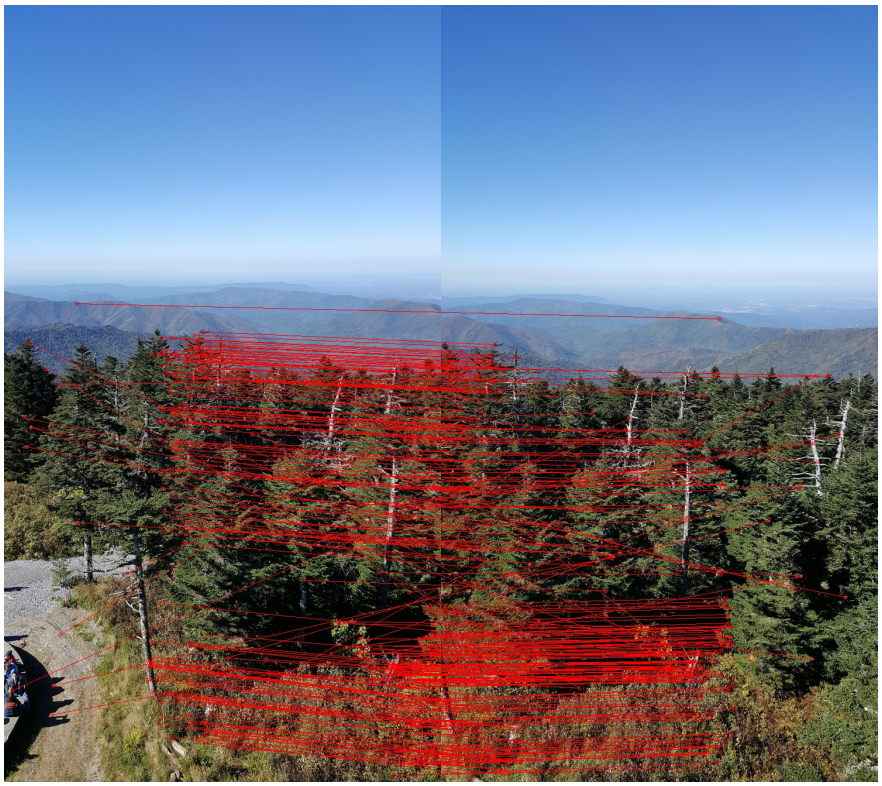


(a) Correspondences

Figure 8: Correspondences between Images 3 and 4



(a) Inliers



(b) Outliers

Figure 9: Inliers and Outliers for Image 3 and 4

(a) Without LM

(b) With LM

Figure 10: Panorama for all the images

Task 2

For this task, a panorama was produced consisting of the five images from fig. 11.

(a) 5 (b) 6 (c) 7

(d) 8 (e) 9 (f) 10

Figure 11: Input Images for task 2

Outputs

(a) Correspondences

Figure 12: Correspondences between Images 0 and 1

(a) Inliers

(b) Outliers

Figure 13: Inliers and Outliers for Image 0 and 1

(a) Correspondences

Figure 14: Correspondences between Images 1 and 2

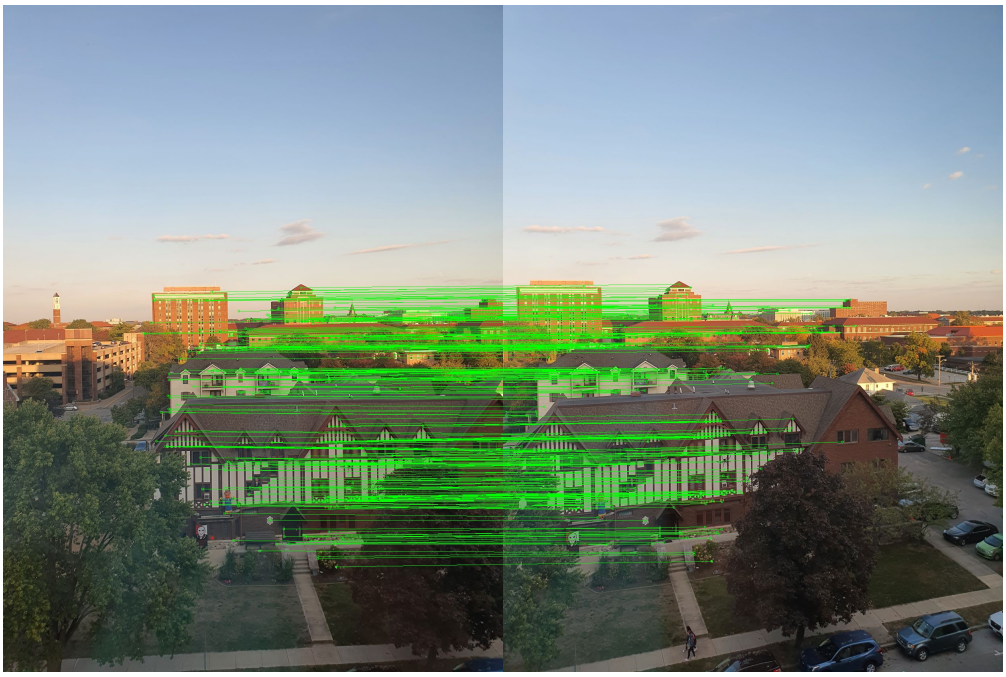
(a) Inliers

(b) Outliers

Figure 15: Inliers and Outliers for Image 1 and 2

(a) Correspondences

Figure 16: Correspondences between Images 2 and 3



(a) Inliers

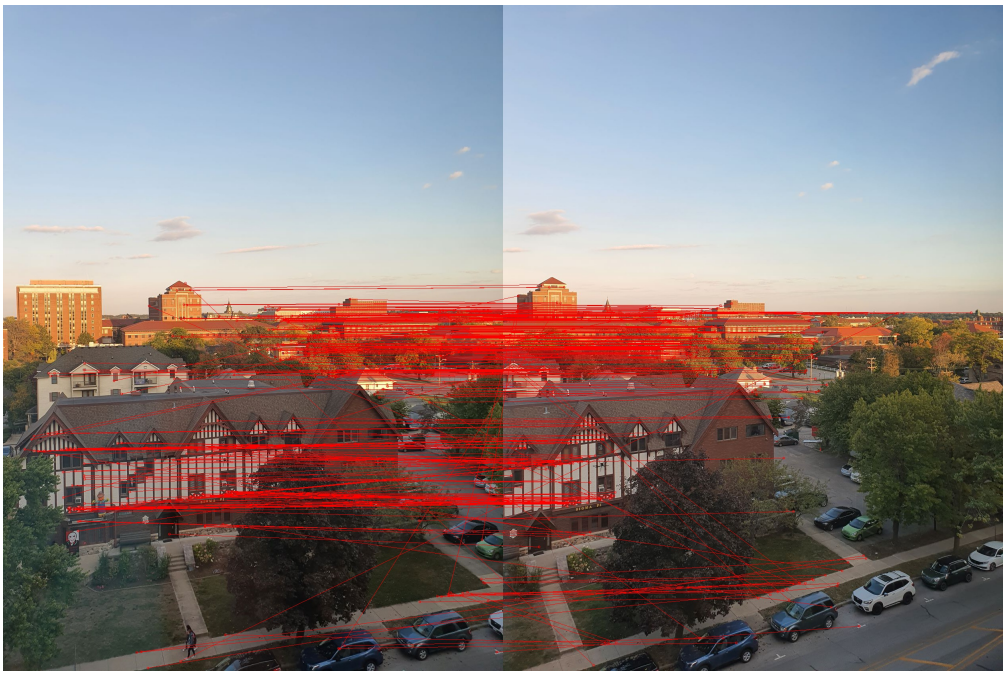
(b) Outliers

Figure 17: Inliers and Outliers for Image 2 and 3

(a) Correspondences

Figure 18: Correspondences between Images 3 and 4

(a) Inliers



(b) Outliers

Figure 19: Inliers and Outliers for Image 3 and 4

(a) Without LM

(b) With LM

Figure 20: Panorama for all the images

Source Code

```
import math
from pickletools import optimize
import numpy as np
import cv2 as cv
import random
from scipy import optimize
#---------------------------Functions-----------------------------#
def drawCorrespondences(corrs, img0, img1, numCorres):
    w = img0.shape[1]
    comb_img = np.concatenate((img0, img1), axis=1)
    for corr in corrs[0:numCorres]:
        x = corr[0]
        xp = [corr[1][0]+w, corr[1][1]]
        color = np.random.randint(0, 255, size=(3, ))
        color = (int(color[0]), int(color[1]), int(color[2]))
        cv.circle(comb_img, tuple(x), 3, tuple(color), 1)
        cv.circle(comb_img, tuple(xp), 3, tuple(color), 1)
        cv.line(comb_img, tuple(x), tuple(xp), tuple(color), 1)
    return comb_img

def getDistance(des1, des2, mode):
    if mode == 'NCC':
        m1 = np.mean(des1)
        m2 = np.mean(des2)
        d = 1 - np.sum((des1-m1)*(des2-m2))/(np.sqrt(np.sum(np.square(des1
                                                -m1))*np.sum(np.square(des2-
                                                m2))))
    elif mode =='SSD':
        d = np.sum(np.square(des1-des2))
    return d

def sift(img1_raw, img2_raw):
    if len(img1_raw.shape) == 3:
        img1 = cv.cvtColor(img1_raw,cv.COLOR_BGR2GRAY)
    else:
        img1 = img1_raw

    if len(img2_raw.shape) == 3:
        img2 = cv.cvtColor(img2_raw,cv.COLOR_BGR2GRAY)
    else:
        img2 = img2_raw

    # Create sift detector
    sift_detector = cv.SIFT_create(5000)

    # Find key points and descriptors with Sift
    kp1, des1 = sift_detector.detectAndCompute(img1,None)
    kp2, des2 = sift_detector.detectAndCompute(img2,None)

    # Find Correspondences between key points
```

```
    #---------Finding correspondence with SSD or NCC-------#
    # corrs = []
    # for i in range(len(kp1)):
    #     distances = []
    #     domain_coord = list(map(int,kp1[i].pt))
    #     for j in range(len(kp2)):
    #         dist = getDistance(des1[i], des2[j], 'SSD')
    #         distances.append(dist)
    #     bestM = list(map(int, kp2[np.argmin(distances)].pt))
    #     corrs.append(np.around([domain_coord, bestM]).astype(int))

    #--------Finding Correspondences with BFMatcher-------#
    corrs = []
    bf = cv.BFMatcher()
    matches=bf.match(des1,des2)
    matches=sorted(matches, key= lambda x:x.distance)
    for mat in matches:
        domain_coord = kp1[mat.queryIdx].pt
        bestM = kp2[mat.trainIdx].pt
        corrs.append(np.around([domain_coord, bestM]).astype(int))
    return corrs

def estimateH(corrs):
    numCorrs = len(corrs)
    if numCorrs<4:
        return 0

     # A and B Matrix
    A = np.zeros((2*numCorrs,8))
    b = np.zeros((2*numCorrs,1))
    for i in range(numCorrs):
        pair = corrs[i]
        xp = pair[1][0]
        yp = pair[1][1]
        x = pair[0][0]
        y = pair[0][1]
        # A matrix
        A[2*i][0:8] = [x, y, 1 , 0 , 0 , 0 ,-x*xp, -y*xp]
        A[2*i+1][0:8] = [0, 0, 0, x, y, 1, -x*yp, -y*yp]
        # B matrix
        b[2*i] = xp
        b[2*i+1] = yp

    # H matrix
    H = np.zeros((3,3))
    H = np.matmul(np.linalg.pinv(A),b)
    H = np.append(H,[1])
    H = np.reshape(H, (3,3))

    return H

def getInliers(H, corrs, delta):
    dom_cords = np.zeros((len(corrs), 3), np.uint16)
    range_cords = np.zeros((len(corrs), 3), np.uint16)
```

```
    for i in range(len(corrs)):
        pair = corrs[i]
        dom = pair[0]
        rang = pair[1]
        dom_cords[i] = [dom[0], dom[1], 1]
        range_cords[i] = [rang[0], rang[1], 1]
    Y = np.matmul(H, np.transpose(dom_cords))
    Y = Y / Y[2,:]

    err = np.square(range_cords.T - Y)
    dist = np.sum(err,axis=0)
    inliers = np.array(corrs)[dist<=delta]
    outliers = np.array(corrs)[dist>delta]
    return inliers, outliers

def RANSAC(corrs, epsilon, sigma, n, p):
    N = math.ceil(math.log(1-p)/math.log(1-(1-epsilon)**n))
    delta = 3*sigma
    n_total = len(corrs)
    M = math.ceil((1-epsilon)*n_total)
    outlier_cords = []
    inlier_cords = []
    numInliers = 0

    for iter in range(N):
        corrSample = random.sample(corrs, n)
        H_estimate = estimateH(corrSample)
        inliers, outliers = getInliers(H_estimate, corrs, delta)
        if len(inliers)>numInliers:
            numInliers = len(inliers)
            inlier_cords = inliers
            outlier_cords = outliers
    return inlier_cords, outlier_cords

def drawInliersOutliers(img0, img1, inliers, outliers, maxPts):
    w = img0.shape[1]
    comb_img1 = np.concatenate((img0, img1), axis=1)
    comb_img2 = comb_img1.copy()
    for i in range(len(inliers) if len(inliers)<maxPts else maxPts):
        pts = inliers[i]
        x = pts[0]
        xp = pts[1] + [w,0]
        cv.circle(comb_img1, tuple(x), 3, (0,255,0), 1)
        cv.circle(comb_img1, tuple(xp), 3, (0,255,0), 1)
        cv.line(comb_img1, tuple(x), tuple(xp), (0,255,0), 1)
    for i in range(len(outliers) if len(outliers)<maxPts else maxPts):
        pts = outliers[i]
        x = pts[0]
        xp = pts[1] + [w,0]
        cv.circle(comb_img2, tuple(x), 3, (0,0,255), 1)
        cv.circle(comb_img2, tuple(xp), 3, (0,0,255), 1)
        cv.line(comb_img2, tuple(x), tuple(xp), (0,0,255), 1)

    return comb_img1, comb_img2
```

```
def costFunc(h, inliers):
    X = []
    F = []
    for i in range(len(inliers)):
        pair = inliers[i]
        d_x = pair[0][0]
        d_y = pair[0][1]
        r_x = pair[1][0]
        r_y = pair[1][1]
        X.append(r_x)
        X.append(r_y)
        F.append((h[0]*d_x + h[1]*d_y + h[2])/(h[6]*d_x + h[7]*d_y + h[8])
                                                )
        F.append((h[3]*d_x + h[4]*d_y + h[5])/(h[6]*d_x + h[7]*d_y + h[8])
                                                )
    err = np.array(X) - np.array(F)

    return err

def getPixelValue(img, coord):
    pt = np.array([coord[0], coord[1]])
    x = int(np.floor(coord[0]))
    y = int(np.floor(coord[1]))
    d1 = np.linalg.norm(pt - np.array([x,y]))
    d2 = np.linalg.norm(pt - np.array([x, y+1]))
    d3 = np.linalg.norm(pt - np.array([x+1, y]))
    d4 = np.linalg.norm(pt - np.array([x+1, y+1]))
    pixel = (img[y][x][:]*d1 + img[y+1][x][:]*d2 + img[y][x+1][:]*d3 + img
                                            [y+1][x+1][:]*d4)/(d1+d2+d3+d4)
    return pixel

def mapPixels(canvas, img, h):
    h = np.linalg.pinv(h)

    # Create a matrix of coordinates for the canvas
    worldCoords = np.array([[i, j, 1] for i in range(canvas.shape[1]) for
                                            j in range(canvas.shape[0])])
    worldCoords = np.transpose(worldCoords)
    # Transform those coordinates using H
    imgCoords = np.matmul(h, worldCoords)
    imgCoords = imgCoords/imgCoords[2,:]

    # isolate the pixel coordinates that will change based on the current
                                            image
    temp = imgCoords[0, :] >= 0
    worldCoords = worldCoords[:, temp]
    imgCoords = imgCoords[:, temp]

    temp = imgCoords[0, :] <= img.shape[1]-1
    worldCoords = worldCoords[:, temp]
    imgCoords = imgCoords[:, temp]

    temp = imgCoords[1, :] >= 0
```

```
    worldCoords = worldCoords[:, temp]
    imgCoords = imgCoords[:, temp]

    temp = imgCoords[1, :] <= img.shape[0]-1
    worldCoords = worldCoords[:, temp]
    imgCoords = imgCoords[:, temp]

    pts = worldCoords.shape[1]
    for i in range(pts):
        point = imgCoords[:,i]
        y = worldCoords[1][i]
        x = worldCoords[0][i]
        canvas[y][x][:] = getPixelValue(img, point)

    return canvas

def panorama(imgs, H):

    # Taking center image as reference image
    refImg = int((len(imgs)+1)/2)-1
    print("Ref Img: ",refImg)

    h_temp = np.eye(3)
    # Finding H to tranform each image to refImg frame
    for i in range(refImg-1,-1,-1):
        h_temp = np.matmul(h_temp, H[i])
        H[i] = h_temp

    h_temp = np.eye(3)
    for i in range(refImg, len(H)):
        h_temp = np.matmul(h_temp, np.linalg.pinv(H[i]))
        H[i] = h_temp

    H.insert(refImg, np.eye(3, dtype=np.float64))

    # Add translation for refImg to be in the middle of panorama
    tx = 0
    for i in range(refImg):
        tx = tx + imgs[i].shape[1]
    H_t = np.array([1, 0, tx, 0, 1, 0, 0, 0, 1], dtype=float)
    H_t = H_t.reshape(3,3)
    for i in range(len(H)):
        H[i] = np.matmul(H_t, H[i])

    # Create the canvas
    canvas_w = 0
    canvas_h = 0
    for img in imgs:
        img_w = img.shape[1]
        img_h = img.shape[0]
        canvas_h = max(img_h, canvas_h)
        canvas_w = canvas_w + img_w
    canvas = np.zeros((canvas_h, canvas_w, 3), np.uint8)
```

```
    # Map pixels
    for i, img in enumerate(imgs):
        canvas = mapPixels(canvas, img, H[i])

    return canvas
#-----------------------------Task 1-------------------------------#

# HW images
img0 = cv.imread('HW5/0.jpg')
img1 = cv.imread('HW5/1.jpg')
img2 = cv.imread('HW5/2.jpg')
img3 = cv.imread('HW5/3.jpg')
img4 = cv.imread('HW5/4.jpg')

# My own Images
img5 = cv.imread('HW5/5.jpg')
img6 = cv.imread('HW5/6.jpg')
img7 = cv.imread('HW5/7.jpg')
img8 = cv.imread('HW5/8.jpg')
img9 = cv.imread('HW5/9.jpg')
img10 = cv.imread('HW5/10.jpg')

# imgs = [img0, img1, img2, img3, img4]

imgs = [img5, img6, img7, img8, img9, img10]

hMatrix_LS = []
hMatrix_LM = []

for i in range(len(imgs)-1):
    # Get images
    img0 = imgs[i]
    img1 = imgs[i+1]

    # Use SIFT to find key correpondences
    print("Computing SIFT Correspondences")
    corres = sift(img0, img1)
    corres_img = drawCorrespondences(corres, img0, img1, 250)
    cv.imwrite('HW5/img_'+str(i)+str(i+1)+'_corres_img.jpeg', corres_img)
    print(str(len(corres))+" SIFT Correspondences found between Image "+
                                        str(i)+" and Image "+str(i+1))

    # Use RANSAC to get inliers and outliers
    print("Applying RANSAC")
    inliers, outliers = RANSAC(corres, epsilon=0.75, sigma=2, n=6, p=0.999
                                        )
    inliers_img, outliers_img = drawInliersOutliers(img0,img1,inliers,
                                        outliers, 350)
    cv.imwrite('HW5/img_'+str(i)+str(i+1)+'_inliers_img.jpeg', inliers_img
                                        )
    cv.imwrite('HW5/img_'+str(i)+str(i+1)+'_outliers_img.jpeg',
                                        outliers_img)
    print("RANSAC completed. Inliers: "+str(len(inliers))+" Outliers: "+
                                        str(len(outliers)))
```

```
    # Get a better estimate of H from all the inliers
    H_estimate_LS = estimateH(inliers)
    hMatrix_LS.append(H_estimate_LS)

    # Using LM to get a more refined H estimate
    H_estimate_LM = H_estimate_LS.reshape((1,9))[0]
    H_estimate_LM = optimize.least_squares(costFunc, H_estimate_LM, args=[
                                        inliers], method='lm').x
    H_estimate_LM = H_estimate_LM.reshape((3,3))
    hMatrix_LM.append(H_estimate_LM)

# Build the Panaroma
print("Building Panorama")
img_panorama_LS = panorama(imgs, hMatrix_LS)
img_panorama_LM = panorama(imgs, hMatrix_LM)
cv.imwrite('HW5/img2_panorama_LS.jpeg', img_panorama_LS)
cv.imwrite('HW5/img2_panorama_LM.jpeg', img_panorama_LM)
```