Name: Munasinghe M.M.R.H.

Index No : 190399L

Github Repo: https://github.com/RaviduHM99/EN2550-Fundamentals-of-Image-Processing-and-Machine-Vision/tree/main/Assignments/Assignment%202

Question 1

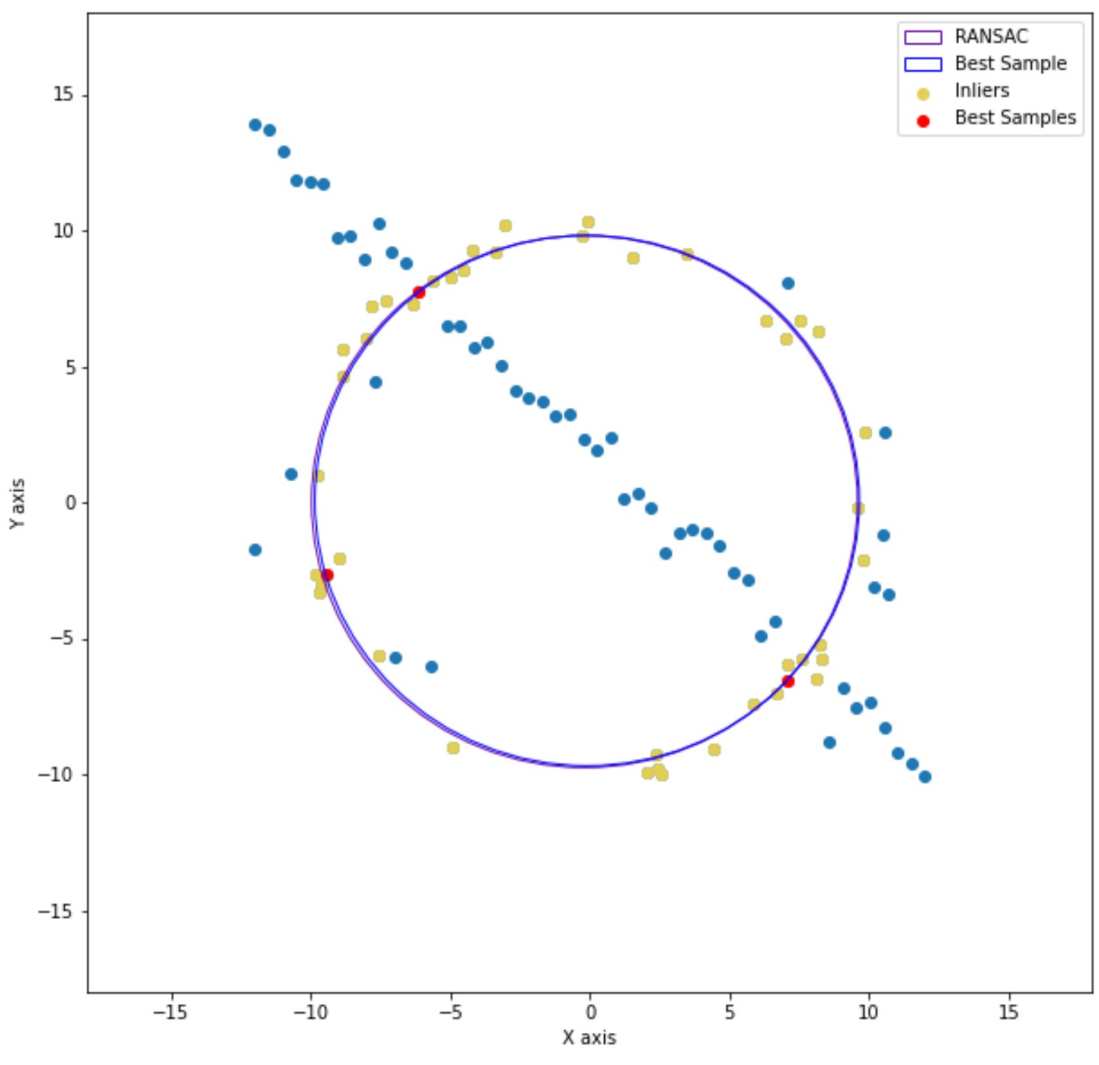
In this question, I implement the RANSAC algorithm. To find most inliers used the RANSAC function one time. Then input those obtained inliers to the RANSAC function to obtain the parameters of the best sample circle.

In [ ]:

```
def RANSAC(X): #RANSAC Algorithm
    Iterations = 100
    Tolerance = 0.8
    Max_inliers_count = 0
    Max_coeff_matrix = []
    Max_inlier_points = []
    Best_3_samples = []
    for iter in range(0, Iterations):
        rand_points = random.sample(range(0, len(X)), 3)
        point1 = rand_points[0]
        point2 = rand_points[1]
        point3 = rand_points[2]

        sample_points = []
        x1, y1 = X[point1][0], X[point1][1]
        x2, y2 = X[point2][0], X[point2][1]
        x3, y3 = X[point3][0], X[point3][1]
        sample_points.append(X[point1])
        sample_points.append(X[point2])
        sample_points.append(X[point3])

        matrix_1 = np.array([[x1, y1, 1],[x2, y2, 1],[x3, y3, 1]])
        matrix_2 = np.array([[-(x1**2 + y1**2)],[-(x2**2 + y2**2)],[-(x3**2 + y3**2)]])

        try:#finding center,radius of the circle using 3 coordinates
            matrix_1_inv = np.linalg.inv(matrix_1)
            coeff_matrix = np.matmul(matrix_1_inv, matrix_2)

            g, f, c = coeff_matrix[0]/2, coeff_matrix[1]/2, coeff_matrix[2] # circle
            radius = (g**2 + f**2 - c)**0.5
            center = [-g, -f]

            if radius < 20: # finding a circle which radius is less than 20.
                inlier_points = []
                inliers_count = 0
                for point in X:
                    distance = ((point[0] - center[0])**2 + (point[1] - center[1])**2)**0.5  # distance to a point from center of the circle
                    diff_dis = abs(distance - radius)
                    if diff_dis < Tolerance:
                        inlier_points.append(point)
                        inliers_count += 1
                if inliers_count >= Max_inliers_count:#Finding maximum inliers, and best estimated circle parameters
                    Max_coeff_matrix = coeff_matrix
                    Max_inliers_count = inliers_count
                    Max_inlier_points = inlier_points[:]
                    Best_3_samples = sample_points[:]
        except: # if matrix_1 is a singular matrix then continue
            continue
    return (Max_coeff_matrix, Max_inliers_count, Max_inlier_points, Best_3_samples)
```

45

Out[ ]: <matplotlib.legend.Legend at 0x23db23e2880>



Question 2

In here first we take 4 point coordinates given by user. These points refer to destination points which use to compute homography later. Afterwards we take vertices of source image as source points and then calculate the homography using 'cv.findHomography()' inbuilt function. The blending of the image is done by using "cv2.addWeighted()" inBuilt fuction.

In [ ]:
```
# Mouse callback function
positions=[]
count=0
def draw_circle(event,x,y,flags,param):
    global positions,count
    # If event is Left Button Click then store the coordinate in the  positions
    if event == cv.EVENT_LBUTTONUP:
        cv.circle(backgnd,(x,y),2,(255,0,0),-1)
        positions.append([x,y])
        count+=1
# Define a window named 'image'
cv.namedWindow('image')
cv.setMouseCallback('image',draw_circle)
while(True):
    cv.imshow('image',backgnd)
    k = cv.waitKey(20) & 0xFF
    if k == 27:
        break
cv.destroyAllWindows()

height, width = backgnd.shape[:2]
h1,w1 = src.shape[:2]
pts1=np.float32([[0,0],[w1,0],[0,h1],[w1,h1]])
pts2=np.float32(positions)
```

```
h, mask = cv.findHomography(pts1, pts2, cv.RANSAC,5.0)#calculate homography
height, width, channels = backgnd.shape
im1Reg = cv.warpPerspective(src, h, (width, height))
cv.addWeighted(im1Reg, 0.5, backgnd, 1, 0, backgnd)#blending images
```

Out[ ]: (-0.5, 799.5, 449.5, -0.5)

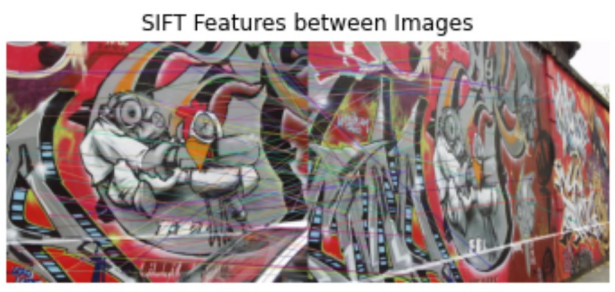
Question 3

Part (A)

SIFT features of two images matched using following code.

In [ ]:
```
SIFT = cv.SIFT_create()
Key_Pots_1, Desp_1 = SIFT.detectAndCompute(Img1,None)
Key_Pots_2, Desp_2 = SIFT.detectAndCompute(Img2,None)
BF = cv.BFMatcher(cv.NORM_L1, crossCheck=True)
Match = BF.match(Desp_1,Desp_2)
Match = sorted(Match, key = lambda x:x.distance)
New_Img = cv.drawMatches(Gray1, Key_Pots_1, Gray2, Key_Pots_2, Match[:100], Gray2, flags=2)
```

Out[ ]: (-0.5, 1599.5, 639.5, -0.5)



Part (B)

I calculate separate homograpies of pictures 1 to 2, 2 to 3, 3 to 4, and 4 to 5 to compute the homography of images 1 to 5. Then, to get the 1 to 5 image homography, we reverse multiply those homography matrices. Directly calculating the homography of images 1 to 5 is quite difficult. Because the two images have a high homography. We may calculate the homography matrix using the RANSAC method by utilizing the Homography function. After computing the homography using the RANSAC method, we compare it to the actual homography matrix to see if the code is accurate. We get the sum of square differences between the two matrices for that. SSD is quite low, as seen in the data.

In [ ]:

```
def siftmatch(img1,img2):
    sift = cv.SIFT_create()
    kp1, descriptors_1 = sift.detectAndCompute(img1, None)
    kp2, descriptors_2 = sift.detectAndCompute(img2, None)
    bf1 = cv.BFMatcher(cv.NORM_L1, crossCheck = True)
    matches1 = bf1.match(descriptors_1, descriptors_2)
    sortmatches1 = sorted(matches1, key = lambda x:x.distance)

    return matches1,[kp1,kp2]
def SSD(corres, h):
    pts1 = np.transpose(np.matrix([corres[0].item(0), corres[0].item(1), 1]))
    estimatep1 = np.dot(h, pts1)
    estimatep2 = (1/estimatep1.item(2))*estimatep1
    pts2 = np.transpose(np.matrix([corres[0].item(2), corres[0].item(3), 1]))
    error = pts2 - estimatep2
    return np.linalg.norm(error)
def Homography(correspondences):
    Lst = []
    for corr in correspondences:
        p1 = np.matrix([corr.item(0), corr.item(1), 1])
        p2 = np.matrix([corr.item(2), corr.item(3), 1])

        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
              p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
        Lst.append(a1)
        Lst.append(a2)
    matrixA = np.matrix(Lst)
    u, s, v = np.linalg.svd(matrixA)
    h = np.reshape(v[8], (3, 3))
    h = (1/h.item(8)) * h
    return h
def ransac(corr, thresh):
    maxInliers = []
    finalH = None
    for i in range(1000):
        corr1 = corr[random.randrange(0, len(corr))]
        corr2 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((corr1, corr2))
        corr3 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr3))
        corr4 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr4))
        h = Homography(randomFour)
        inliers = []
        for i in range(len(corr)):
            d = SSD(corr[i], h)
            if d < 5:
                inliers.append(corr[i])
        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = h
        if len(maxInliers) > (len(corr)*thresh):
            break
    return finalH, maxInliers
def corr_list(matches1,key):
    correspondenceList1 = []
    keypoints1 = [key[0],key[1]]
    for match in matches1:
        (x1, y1) = keypoints1[0][match.queryIdx].pt
        (x2, y2) = keypoints1[1][match.trainIdx].pt
        correspondenceList1.append([x1, y1, x2, y2])
    return correspondenceList1

match1,ky1=siftmatch(img1,img2)
correspondenceList1=corr_list(match1,ky1)
corrs1 = np.matrix(correspondenceList1)
finalH1, inliers1 = ransac(corrs1, 0.6)
match2,ky2=siftmatch(img2,img3)
correspondenceList2=corr_list(match2,ky2)
corrs2 = np.matrix(correspondenceList2)
finalH2, inliers2 = ransac(corrs2, 0.6)
match3,ky3=siftmatch(img3,img4)
correspondenceList3=corr_list(match3,ky3)
corrs3 = np.matrix(correspondenceList3)
finalH3, inliers3 = ransac(corrs3, 0.6)
match4,ky4=siftmatch(img4,img5)
correspondenceList4=corr_list(match4,ky4)
corrs4 = np.matrix(correspondenceList4)
finalH4, inliers4 = ransac(corrs4, 0.6)
#Obtaining the final homography matrix
H = finalH4 @ finalH3 @ finalH2 @ finalH1
print(H)
Original_Homography = [ [6.2544644e-01,5.7759174e-02,2.2201217e+02],
[2.2240536e-01,1.1652147e+00,-2.5605611e+01],
[4.9212545e-04,-3.6542424e-05,1.0000000e+00]]
Calculated_Homography = H
Original_Homography =np.array(Original_Homography)
Calculated_Homography = np.array(Calculated_Homography)
SSD_Calc= np.sum(np.sum((Original_Homography-Calculated_Homography)*(Original_Homography-Calculated_Homography)))
print("SSD Value =",SSD_Calc)
```

```
[[ 6.10391926e-01  7.18952730e-02  2.21962720e+02]
 [ 2.11917736e-01  1.17028153e+00 -2.59496515e+01]
 [ 4.72877676e-04 -1.73172566e-05  9.92228259e-01]]
SSD Value = 0.12143166956061094
```

Part (C)

In [ ]:

```
output = cv.warpPerspective(img1, H, ((img5.shape[1]), img5.shape[0]))
fig, ax = plt.subplots(1,3,figsize = (15,15))
```