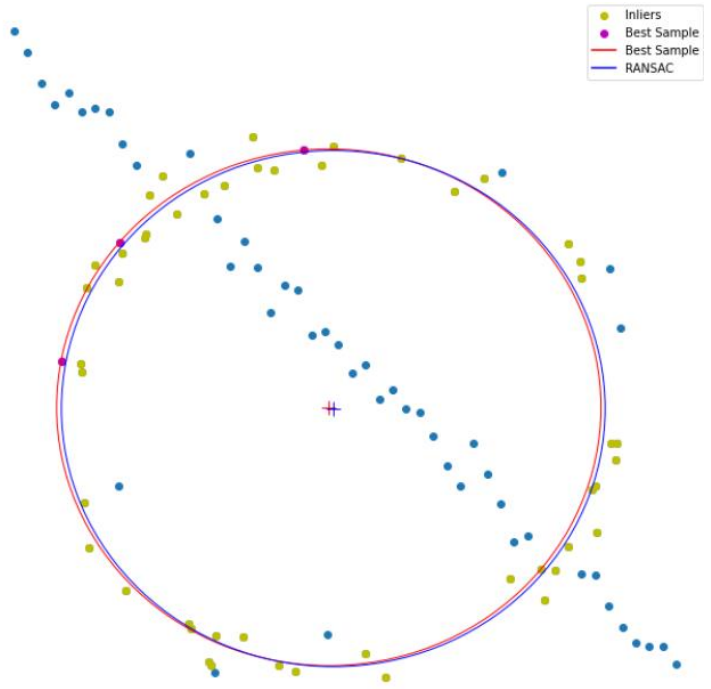
Name: Fernando I.A.M.D.
Index No.: 190172K

GitHub Link: https://github.com/muditha11/ImageProcessingAndMachineVision/tree/main/assignment%202

- Question 1

```
def ransacCircle(x,y,N):
    Td =  1
    TiC = 45 #(N//2)*0.95
    iter = 100
    short_list = []
    for i in range (iter):
        p = np.random.randint(0,len(y))
        q = np.random.randint(0,len(y))
        r = np.random.randint(0,len(y))
        x1,x2,x3,y1,y2,y3 = x[p],x[q],x[r],y[p],y[q],y[r]
        if (x1 != x2 and x2 != x3 and x1 != x3):
                cen,rad = findCircle(x1, y1, x2, y2, x3, y3)
                if (rad<10):
                    inl_cnt = 0
                    inl_x = []
                    inl_y = []
                    for j in range(len(y)):
                        dist = ((x[j] - cen[0])**2 + (y[j] - cen[1])**2)**0.5
                        if (abs(dist-rad)<Td):
                                inl_cnt+=1
                                inl_x.append(x[j])
                                inl_y.append(y[j])
                    if (inl_cnt>TiC):
                        for i in range (iter):
                            p = np.random.randint(0,len(inl_y))
                            q = np.random.randint(0,len(inl_y))
                            r = np.random.randint(0,len(inl_y))
                            x11,x22,x33,y11,y22,y33 = inl_x[p],inl_x[q],inl_x[r],inl_y[p],inl_y[q],inl_y[r]
                            if (x11 != x22 and x22 != x33 and x11 != x33):
                                    cen1,rad1 = findCircle(x11, y11, x22, y22, x33, y33)
                                    inl_x1 = []
                                    inl_y1 = []
                                    inl_cnt1 = 0
                                    error = 0
                                    for j in range(len(inl_y)):
                                        dist = ((inl_x[j] - cen1[0])**2 + (inl_y[j] - cen1[1])**2)**0.5
                                        if (abs(dist-rad1)<Td):
                                                error += abs(dist - rad1)
                                                inl_cnt1+=1
                                                inl_x1.append(inl_x[j])
                                                inl_y1.append(inl_y[j])
                                    if (inl_cnt>TiC):
                                        short_list.append([cen,rad,cen1,rad1,inl_cnt1,error/inl_cnt1,inl_x,inl_y,[x1,x2,x3],[y1,y2,y3]])

    short_list.sort(key=lambda x: (x[4],-x[5]))
    return short_list[-1]
```



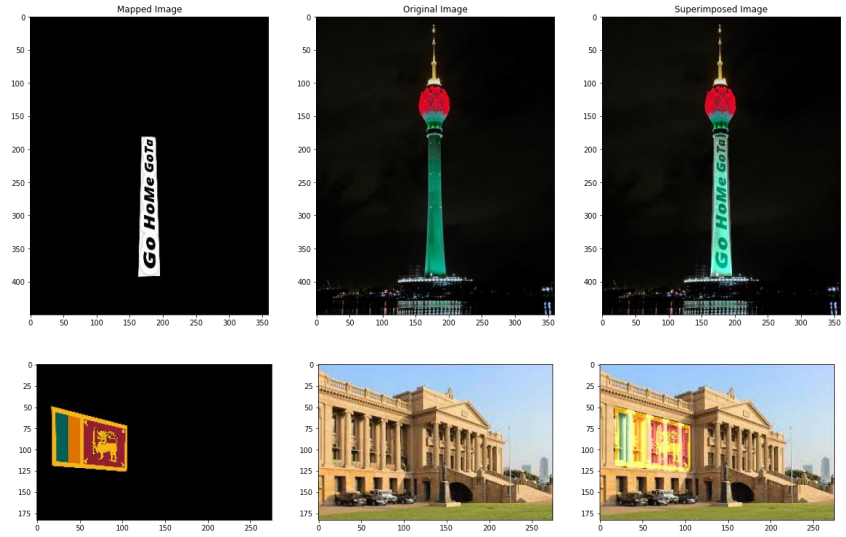
RANSAC method is used to eliminate noise. Our goal here is to find the best fitting circle to the scattered set of points. But when looking at the set of points we can see a line and a circle both. Here Line becomes noise.
Algorithm:

1. Select 3 Random Points and draw a circle. (Candidate Circle)
2. Calculate the number of points lying within a threshold distance from the candidate circle. These points are inliers corresponding to the candidate circle.
3. If the number of inliers is greater than a certain threshold value, go to step 4. If not go back to step 1.
4. Considering the set of inliers find a new candidate circle and find the set of inliers to the new candidate circle.
5. If the number of inliers is greater than the inlier threshold, move to next step. If not move to step 4 for a certain number of iterations
6. For this candidate circle calculate mean absolute error considering new inlier points and the circle. Then shortlist the candidate circle.
7. Go back to step 1. And repeat for a certain number of iterations.
8. When all iterations are complete, examine the shortlist of candidate circles and pick the circle with the maximum number of inliers. If the inlier count is similar, then consider the mean absolute error.

Here there are two stages of finding the best circle. Initially we find the best fit circle and its inliers and then considering the inliers we again find the best fitting circle.

- Question 2

```
N = 4
global n
n = 0
pts_src = np.empty((N, 2))
pts_dst = np.empty((N, 2))

def draw_circle(event, x, y, flags, param):
    global n
    p = param[0]
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(param[1], (x, y), 5, (255, 0, 0), -1)
        p[n] = (x, y)
        n += 1

im_src = cv2.imread('1.jpg')
print(im_src.shape)
pts_src = np.array([[0, 0], [0,84], [492, 0],[492, 84]])

im_dst = cv2.imread('towering-at-an-imposing.jpg')

im_dst_copy = im_dst.copy()
param = [pts_dst, im_dst_copy]
cv2.namedWindow("Select Points", cv2.WINDOW_AUTOSIZE)
cv2.setMouseCallback('Select Points', draw_circle, param)

while (1):
    cv2.imshow("Select Points", im_dst_copy)
    if n == N:
        break
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cv2.destroyAllWindows()
h, status = cv2.findHomography(pts_src, pts_dst)
im_out = cv2.warpPerspective(im_src, h, (im_dst.shape[1],im_dst.shape[0]))

im_out1 = cv2.addWeighted(im_dst,1,im_out,.4,0)

fig, ax = plt.subplots(1,3,figsize= (20,10))

ax[0].imshow(cv2.cvtColor(im_out, cv2.COLOR_BGR2RGB))
ax[1].imshow(cv2.cvtColor(im_dst, cv2.COLOR_BGR2RGB))
ax[2].imshow(cv2.cvtColor(im_out1, cv2.COLOR_BGR2RGB))
plt.show()
```



Here the 4 points of the source image to be warped is taken as the four points of the image because the full image is used for warping.
Target points on the destination image to warp the source image is sele mouse clicking.
Then by using the findHomography function the homograph is found.
To warp the homographed images warpPerspective function is used.
Superimposition/Blending is done using addWeighted function.

Here number of source points and target points should be equal.
Also, the order of selecting points should be same in both images.
To find the homograph, findHomography function has few methods. By default, it uses least squares method. We can also use RANSAC method.

- Question 3

a)

```
img1 = cv2.imread('img1.ppm')
img2 = cv2.imread('img5.ppm')

sift = cv2.SIFT_create(nOctaveLayers = 3,contrastThreshold = .1,edgeThreshold = 25,sigma = 1)

keypoints_1, descriptors_1 = sift.detectAndCompute(img1,None)
keypoints_2, descriptors_2 = sift.detectAndCompute(img2,None)

bf_match = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)

matches = bf_match.match(descriptors_1,descriptors_2)
matches = sorted(matches, key = lambda x:x.distance)

match_img = cv2.drawMatches(img1, keypoints_1, img2, keypoints_2, matches[:10], img2, flags=2)
plt.figure(figsize=(15,15))
plt.axis('off')
plt.imshow(cv2.cvtColor(match_img, cv2.COLOR_BGR2RGB))
plt.show()
```

To find sift features in build function of OpenCV was used. But when using SIFT_create if the default parameters are used faulty results are being shown. “contrastThreshold”,” edgeThreshold”,” sigma” parameter values were adjusted as shown in code and when drawing matches only the best ten matches were considered. This was done because the matches are sorted to have the best matches first. And the matches in the latter part of the array is inaccurate.

b)

```
def RansacHomo(sourcePoints,destinationPoints):
    iter = 500
    Td= 1
    maxInliers = []
    finalH = None
    for i in range(iter):
        p = np.random.randint(0,len(sourcePoints))
        q = np.random.randint(0,len(sourcePoints))
        r = np.random.randint(0,len(sourcePoints))
        s = np.random.randint(0,len(sourcePoints))

        P = [sourcePoints[p][0][0],sourcePoints[p][0][1],destinationPoints[p][0][0],destinationPoints[p][0][1]]
        Q = [sourcePoints[q][0][0],sourcePoints[q][0][1],destinationPoints[q][0][0],destinationPoints[q][0][1]]
        x = np.vstack((P,Q))
        R = [sourcePoints[r][0][0],sourcePoints[r][0][1],destinationPoints[r][0][0],destinationPoints[r][0][1]]
        x = np.vstack((x,R))
        S = [sourcePoints[s][0][0],sourcePoints[s][0][1],destinationPoints[s][0][0],destinationPoints[s][0][1]]
        x = np.vstack((x,S))

        aList = []
        for corr in x:
            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)]
            aList.append(a1)
            aList.append(a2)

        matrixA = np.matrix(aList)

        u, s, v = np.linalg.svd(matrixA)

        H = np.reshape(v[8], (3, 3))
        H = (1/H.item(8)) * H
        inliers = []
        for k in range(len(sourcePoints)):

            d = geometricDistance([sourcePoints[k][0][0],sourcePoints[k][0][1],destinationPoints[k][0][0],destinationPoints[k][0][1]], H)
            if d < Td:
                inliers.append([sourcePoints[k][0][0],sourcePoints[k][0][1],destinationPoints[k][0][0],destinationPoints[k][0][1]])
        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = H


    maxInliers1 = []
    finalH1 = None
    for i in range(iter):
            p = np.random.randint(0,len(maxInliers))
            q = np.random.randint(0,len(maxInliers))
            r = np.random.randint(0,len(maxInliers))
            s = np.random.randint(0,len(maxInliers))

            P = [maxInliers[p][0],maxInliers[p][1],maxInliers[p][2],maxInliers[p][3]]
            Q = [maxInliers[q][0],maxInliers[q][1],maxinliers[q][2],maxInliers[q][3]]
            x = np.vstack((P,Q))
            R = [maxInliers[r][0],maxInliers[r][1],maxInliers[r][2],maxInliers[r][3]]
            x = np.vstack((x,R))
            S = [maxInliers[s][0],maxInliers[s][1],maxInliers[s][2],maxInliers[s][3]]
            x = np.vstack((x,S))

            aList = []
            for corr in x:
                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)]
                aList.append(a1)
                aList.append(a2)

            matrixA = np.matrix(aList)

            u, s, v = np.linalg.svd(matrixA)

            H = np.reshape(v[8], (3, 3))
            H = (1/H.item(8)) * H
            inliers = []
            for k in range(len(maxInliers)):

                d = geometricDistance([maxInliers[k][0],maxInliers[k][1],maxInliers[k][2],maxInliers[k][3]], H)
                if d < Td:
                    inliers.append([maxInliers[k][0],maxInliers[k][1],maxInliers[k][2],maxInliers[k][3]])

            if len(inliers) > len(maxInliers1):
                maxInliers1 = inliers
                finalH1 = H

    return maxInliers,finalH1
```

```
def homography(img1,img2):
    sift = cv2.SIFT_create(nOctaveLayers = 3,contrastThreshold = .1,edgeThreshold = 25,sigma =1)

    keyPoints1, descriptors1 = sift.detectAndCompute(img1, None)
    keyPoints2, descriptors2 = sift.detectAndCompute(img2, None)


    bf = cv2.BFMatcher()
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)
    goodMatches = []
    for m, n in matches:
        if m.distance < 0.95 * n.distance:
            goodMatches.append(m)
    MIN_MATCH_COUNT = 10
    if len(goodMatches) > MIN_MATCH_COUNT:
        sourcePoints = np.float32([keyPoints1[m.queryIdx].pt for m in goodMatches]).reshape(-1, 1, 2)
        destinationPoints = np.float32([keyPoints2[m.trainIdx].pt for m in goodMatches]).reshape(-1, 1, 2)
        maxInliers1,finalH1 = RansacHomo(sourcePoints,destinationPoints)

    return finalH1
```

```
def geometricDistance(correspondence, h):

    p1 = np.transpose(np.matrix([correspondence[0], correspondence[1], 1]))
    estimatep2 = np.dot(h, p1)
    estimatep2 = (1/estimatep2.item(2))*estimatep2

    p2 = np.transpose(np.matrix([correspondence[2], correspondence[3], 1]))
    error = p2 - estimatep2
    return np.linalg.norm(error)
```

Homograph is calculated using RANSAC algorithm. To calculate Homograph four points are required. Initially using SIFT matches are created (source and destination). Then by choosing four random matches the Homography matrix is calculated. Then inliers are calculated by considering distance between transformed source points (Homograph matrix * source points) and destination points and comparing it with a distance threshold. If the number of inlier points is greater than a certain threshold value, by considering the set of inliers again a homography is calculated. Then again inliers to that are found. If number of inliers are greater than the threshold and if it has the maximum number of inliers, then it is considered as the final Homograph.

Here also homograph is calculated in two stages. Initially the best fitting homograph is found and then considering the inliers of it again the best fitting homograph is found.

But here to have an accurate homograph there should be accurate matches created by SIFT. But when considering images 1 and 5 due to the large difference in rotation and orientation the matches created by SIFT is not very accurate. Therefore, the Homograph calculated by these functions is not very accurate.

```
[[-9.99895408e-01 -5.45712234e-02  2.74966736e+02]
 [-1.65508524e+00 -9.03294743e-02  4.55140991e+02]
 [-3.63642317e-03 -1.98464819e-04  1.00000000e+00]]
```

Calculated Homography matrix

```
[[ 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]]
```

Correct Homography matrix

Therefore, first we calculate Homography matrices between images 1to2, 2to3, 3to4 and 4to5 and then by taking the matrix multiplication of these matrices in reverse order we can get the Homography matrix from 1to5 as shown in the code below.

By this it was able to achieve a homography matrix very closer to the correct homography matrix.

```
img1 = cv2.imread('img1.ppm')
img2 = cv2.imread('img2.ppm')
img3 = cv2.imread('img3.ppm')
img4 = cv2.imread('img4.ppm')
img5 = cv2.imread('img5.ppm')


H12 = homography(img1,img2)
H23 = homography(img2,img3)
H34 = homography(img3,img4)
H45 = homography(img4,img5)

H15 = H45 @ H34 @ H23 @ H12
print(H15)
```

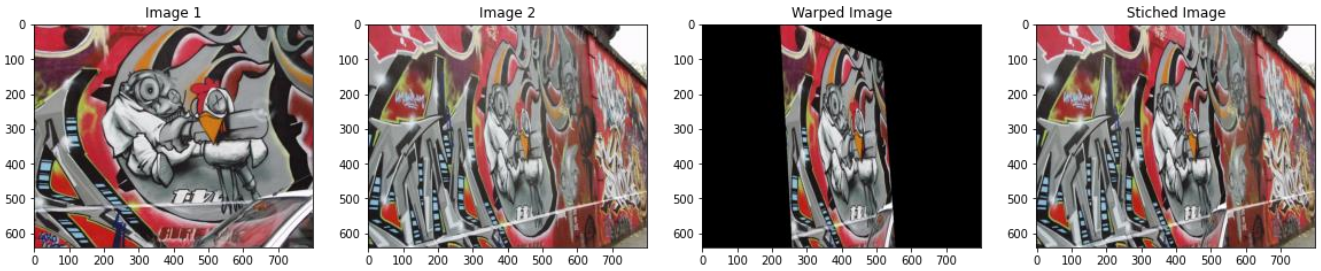
```
[[ 6.25220453e-01  4.82833957e-02  2.20986126e+02]
 [ 2.23754319e-01  1.14698519e+00 -2.53684278e+01]
 [ 4.99804967e-04 -6.45730003e-05  9.94652753e-01]]
```

Homography matrix by multiplication of 1-2,2-3,3-4,4-5 homography matrices

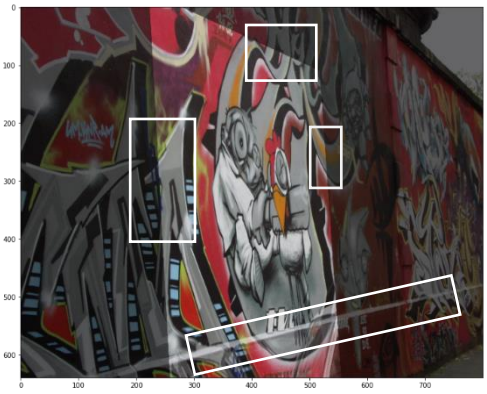
c)

```
im1_warped = cv2.warpPerspective(img1, H15, (img5.shape[1],img5.shape[0]))
ret, threshold = cv2.threshold(im1_warped, 10, 1, cv2.THRESH_BINARY_INV)
img2_thresholded = np.multiply(threshold, img5)
img_blended = cv2.addWeighted(img2_thresholded, 1, im1_warped, 1, 0)
```

Here the warped image is thresholded and blended.



This shows the result of the warping done using finally calculated homography matrix. To better see the stitching in the image below the second image intensity is reduced.



As shown the image 1 coincides with image 5.