EN2550 ASSIGNMENT 2

FITTINGS AND ALIGNMENTS

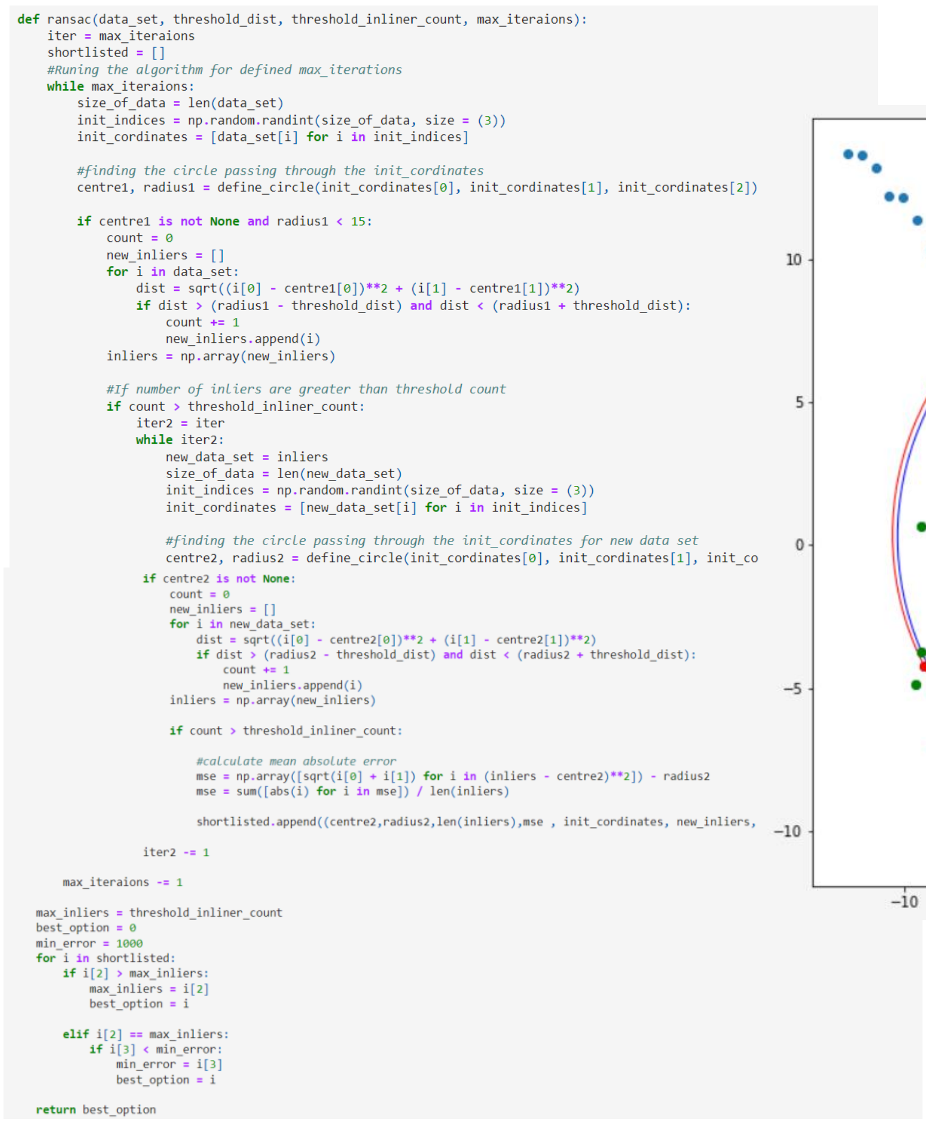
TRESHAN AYESH 190443T

Q1)

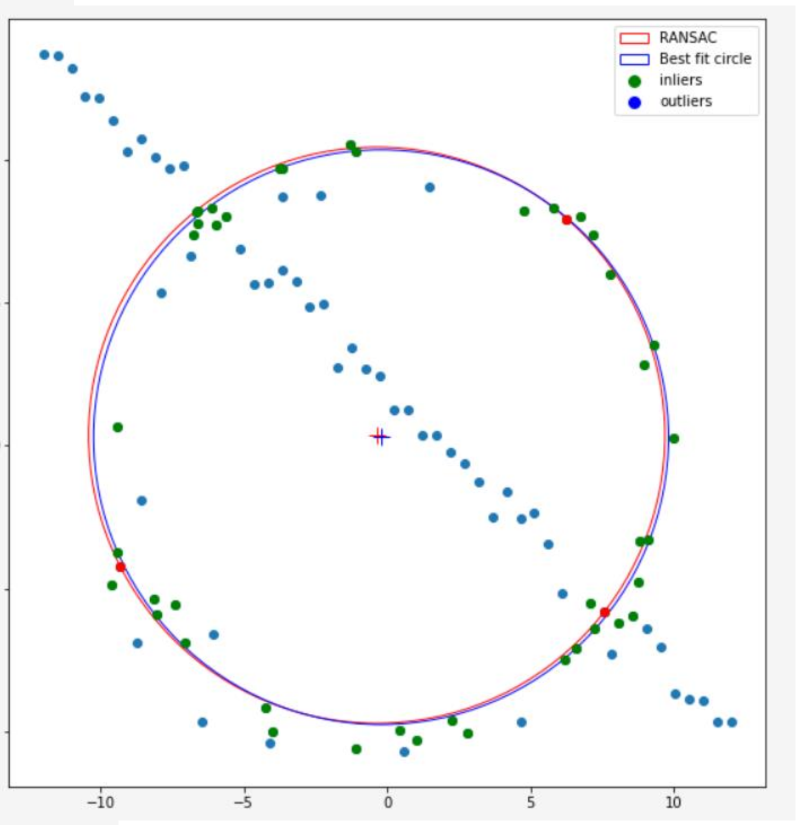
The circle has to be estimated using RANSAC algorithm using the noisy point set provided. The algorithm first selects 3 random points from the dataset and try to fit a circle passing through those 3 points. Then it identifies the points which are between a threshold distance lower and higher than the radius of fitted circle. These points are known as inlier points. If the number of these inlier points are higher than a threshold inlier count, all the inlier points are taken, and a new circle is fitted using 3 random points from the inliers. Again, the number of inliers is calculated and if greater than the threshold inlier count then mean absolute error is calculated using the inlier points.

Finally, the best fit circle is identified which has the maximum number of inlier points. In case two or more circles have the same number of maximum inliers then the circle with least mean square error is taken as the best fit.

The code and the results of the RANSAC algorithm is given below.



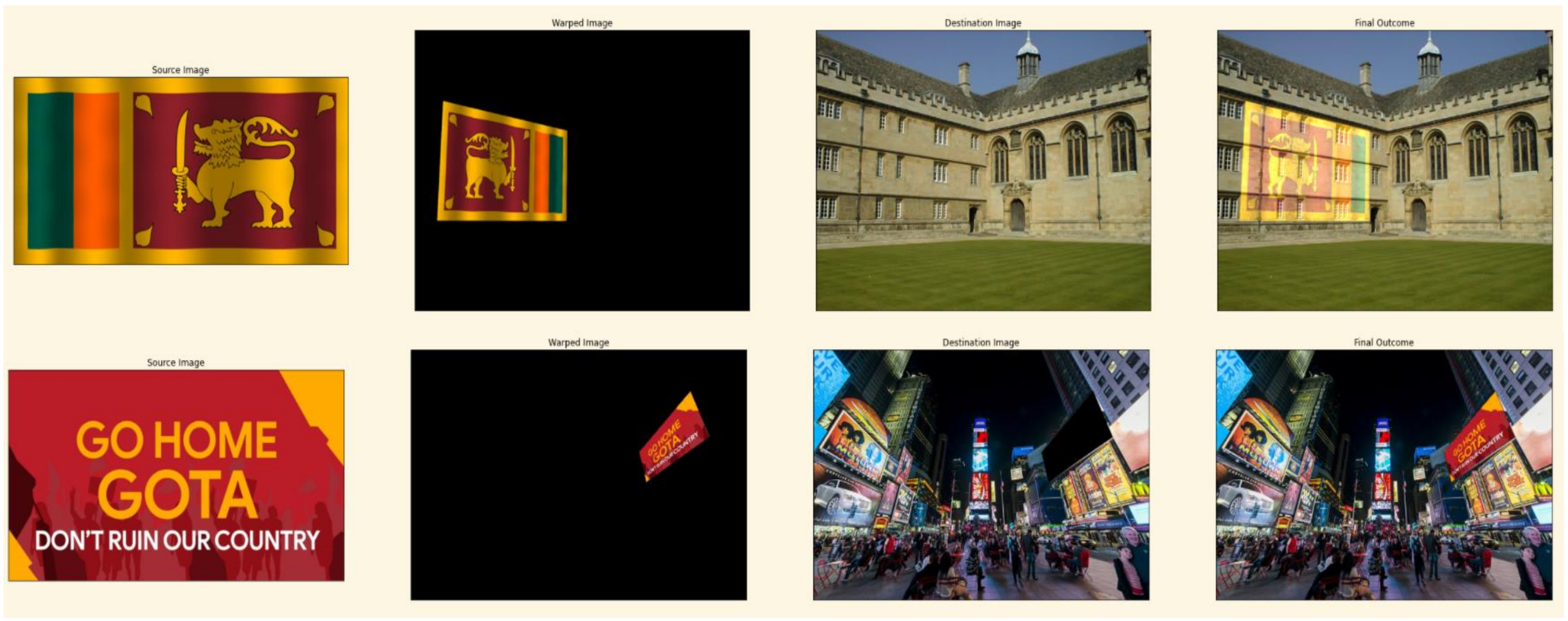
```
def ransac(data_set, threshold_dist, threshold_inliner_count, max_iteraions):
    iter = max_iteraions
    shortlisted = []
    #Runing the algorithm for defined max_iterations
    while max_iteraions:
        size_of_data = len(data_set)
        init_indices = np.random.randint(size_of_data, size = (3))
        init_cordinates = [data_set[i] for i in init_indices]

        #finding the circle passing through the init_cordinates
        centre1, radius1 = define_circle(init_cordinates[0], init_cordinates[1], init_cordinates[2])

        if centre1 is not None and radius1 < 15:
            count = 0
            new_inliers = []
            for i in data_set:
                dist = sqrt((i[0] - centre1[0])**2 + (i[1] - centre1[1])**2)
                if dist > (radius1 - threshold_dist) and dist < (radius1 + threshold_dist):
                    count += 1
                    new_inliers.append(i)
            inliers = np.array(new_inliers)

            #If number of inliers are greater than threshold count
            if count > threshold_inliner_count:
                iter2 = iter
                while iter2:
                    new_data_set = inliers
                    size_of_data = len(new_data_set)
                    init_indices = np.random.randint(size_of_data, size = (3))
                    init_cordinates = [new_data_set[i] for i in init_indices]

                    #finding the circle passing through the init_cordinates for new data set
                    centre2, radius2 = define_circle(init_cordinates[0], init_cordinates[1], init_co
                    if centre2 is not None:
                        count = 0
                        new_inliers = []
                        for i in new_data_set:
                            dist = sqrt((i[0] - centre2[0])**2 + (i[1] - centre2[1])**2)
                            if dist > (radius2 - threshold_dist) and dist < (radius2 + threshold_dist):
                                count += 1
                                new_inliers.append(i)
                        inliers = np.array(new_inliers)

                        if count > threshold_inliner_count:

                            #calculate mean absolute error
                            mse = np.array([sqrt(i[0] + i[1]) for i in (inliers - centre2)**2]) - radius2
                            mse = sum([abs(i) for i in mse]) / len(inliers)

                            shortlisted.append((centre2,radius2,len(inliers),mse , init_cordinates, new_inliers,

                    iter2 -= 1

        max_iteraions -= 1

    max_inliers = threshold_inliner_count
    best_option = 0
    min_error = 1000
    for i in shortlisted:
        if i[2] > max_inliers:
            max_inliers = i[2]
            best_option = i

        elif i[2] == max_inliers:
            if i[3] < min_error:
                min_error = i[3]
                best_option = i

    return best_option
```



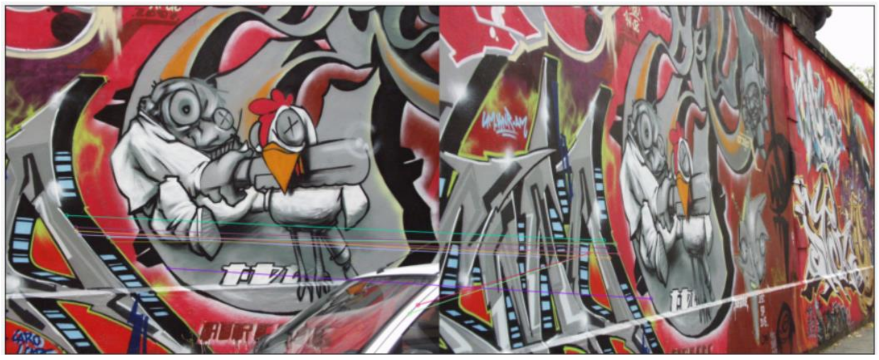
Q2) First a user interface is written in OpenCV to collect 4 destination points which the source image needs to be superimposed. Then homography must be calculated between the given source image and four points selected as above. Then the source image is warped using the cv2.warpPerspective() and finally, it is blended to the destination image using cv2.addWeighted() method. Here if the passed 'blend' Boolean value is false, the polygon made by the 4 points selected in the destination image, is filled by black pixels and then the source image is directly added.

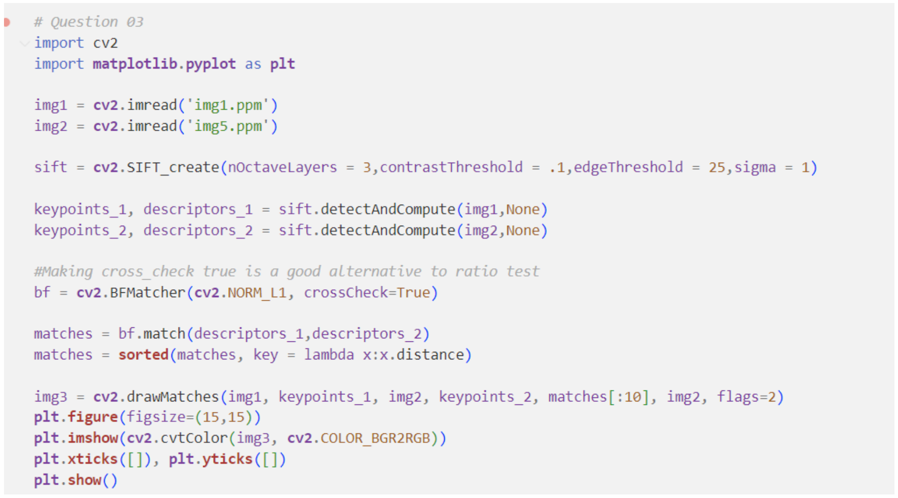
```python
def stich_image(src_img, dst_image , blend):
    N = 4
    global n
    n = 0
    pts_src = np.empty((N, 2))
    pts_dst = np.empty((N, 2))

    im_src = cv2.imread(src_img)
    size = im_src.shape

    # Create a vector of source points.
    pts_src = np.array(
                        [
                            [0,0],
                            [size[1] - 1, 0],
                            [size[1] - 1, size[0] -1],
                            [0, size[0] - 1 ]
                            ],dtype=float
                        )


    im_dst = cv2.imread(dst_image)

    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)
```

```python
while (1):
    cv2.imshow("Select Points", im_dst_copy)
    if n == N:
        cv2.destroyAllWindows()
        break

    if cv2.waitKey(1) & 0xFF == ord('q'):
        cv2.destroyAllWindows()
        break

h, status = cv2.findHomography(pts_src, pts_dst)
im_out = cv2.warpPerspective(im_src, h, (im_dst.shape[1],im_dst.shape[0]))

if blend == True:
    im_out1 = cv2.addWeighted(im_dst,1,im_out,.6,0)
else:
    cv2.fillConvexPoly(im_dst, pts_dst.astype(int), 0, 16)
    im_out1 = im_dst + im_out
```



Q3) a) Feature matching between the two images are done using OpenCV function cv. SIFT_create(). Then the function detectAndCompute is used to get the key points of each image. This function will return two values – the keypoints and descriptors.Next the *BFmatcher* module is used to match the keypoints of two images.





```python
# Question 03
import cv2
import matplotlib.pyplot as plt

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)

#Making cross_check true is a good alternative to ratio test
bf = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)

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

img3 = cv2.drawMatches(img1, keypoints_1, img2, keypoints_2, matches[:10], img2, flags=2)
plt.figure(figsize=(15,15))
plt.imshow(cv2.cvtColor(img3, cv2.COLOR_BGR2RGB))
plt.xticks([]), plt.yticks([])
plt.show()
```

b) Here it is expected to compute the homography between 2 images using the method RANSAC. First the matching feature points are computed using SIFT. Next random 4 points are selected from those matching set and Homography is calculated using those points. For that, using the 4 matching points following matrix is created.

Homography matrix =

h11	h12	h13
h21	h22	h23
h31	h32	h33

$$\begin{bmatrix} x_s^{(1)} & y_s^{(1)} & 1 & 0 & 0 & 0 & -x_d^{(1)}x_s^{(1)} & -x_d^{(1)}y_s^{(1)} & -x_d^{(1)} \\ 0 & 0 & 0 & x_s^{(1)} & y_s^{(1)} & 1 & -y_d^{(1)}x_s^{(1)} & -y_d^{(1)}y_s^{(1)} & -y_d^{(1)} \\ & & & & \vdots & & & & \\ x_s^{(i)} & y_s^{(i)} & 1 & 0 & 0 & 0 & -x_d^{(i)}x_s^{(i)} & -x_d^{(i)}y_s^{(i)} & -x_d^{(i)} \\ 0 & 0 & 0 & x_s^{(i)} & y_s^{(i)} & 1 & -y_d^{(i)}x_s^{(i)} & -y_d^{(i)}y_s^{(i)} & -y_d^{(i)} \\ & & & & \vdots & & & & \\ x_s^{(n)} & y_s^{(n)} & 1 & 0 & 0 & 0 & -x_d^{(n)}x_s^{(n)} & -x_d^{(n)}y_s^{(n)} & -x_d^{(n)} \\ 0 & 0 & 0 & x_s^{(n)} & y_s^{(n)} & 1 & -y_d^{(n)}x_s^{(n)} & -y_d^{(n)}y_s^{(n)} & -y_d^{(n)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Single Value decomposition method in linear algebra module of NumPy is used to compute the homography matrix as above. Next for the calculated homography matrix the relevant correspondence points are calculated and the matching destination points and correspondence points are compared to find the inliers. If the number of inliers is greater than a predefined inlier threshold, next all the inliers are taken and another homography is calculated using 4 random points from those inliers. This is iterated for max iterations and the homography with maximum number of inliers are taken as the best estimate.

As the features mapped from img1 to img5 given in the dataset were not enough for an accurate computation of homography matrix, features are mapped from img1 to img2 to img3 to img4 to img5. Then each homography matrix is multiplied to obtain the homography from img1 to img5.

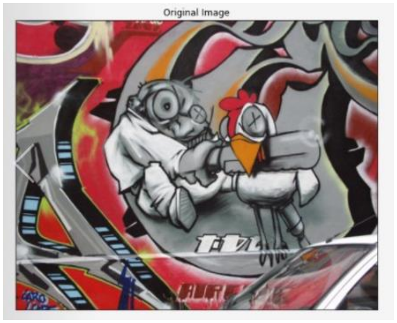
Homography matrix from the code -

```
[[ 6.03475416e-01  8.20940368e-02  2.24521474e+02]
 [ 1.57218880e-01  1.16349906e+00 -3.63335246e+00]
 [ 3.91111946e-04  6.94964014e-05  1.01730698e+00]]
```

Homography matrix given in the dataset-

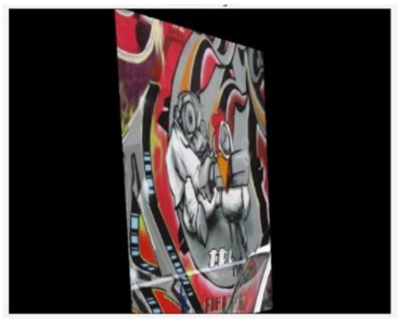
```
6.2544644e-01    5.7759174e-02    2.2201217e+02
2.2240536e-01    1.1652147e+00   -2.5605611e+01
4.9212545e-04   -3.6542424e-05    1.0000000e+00
```

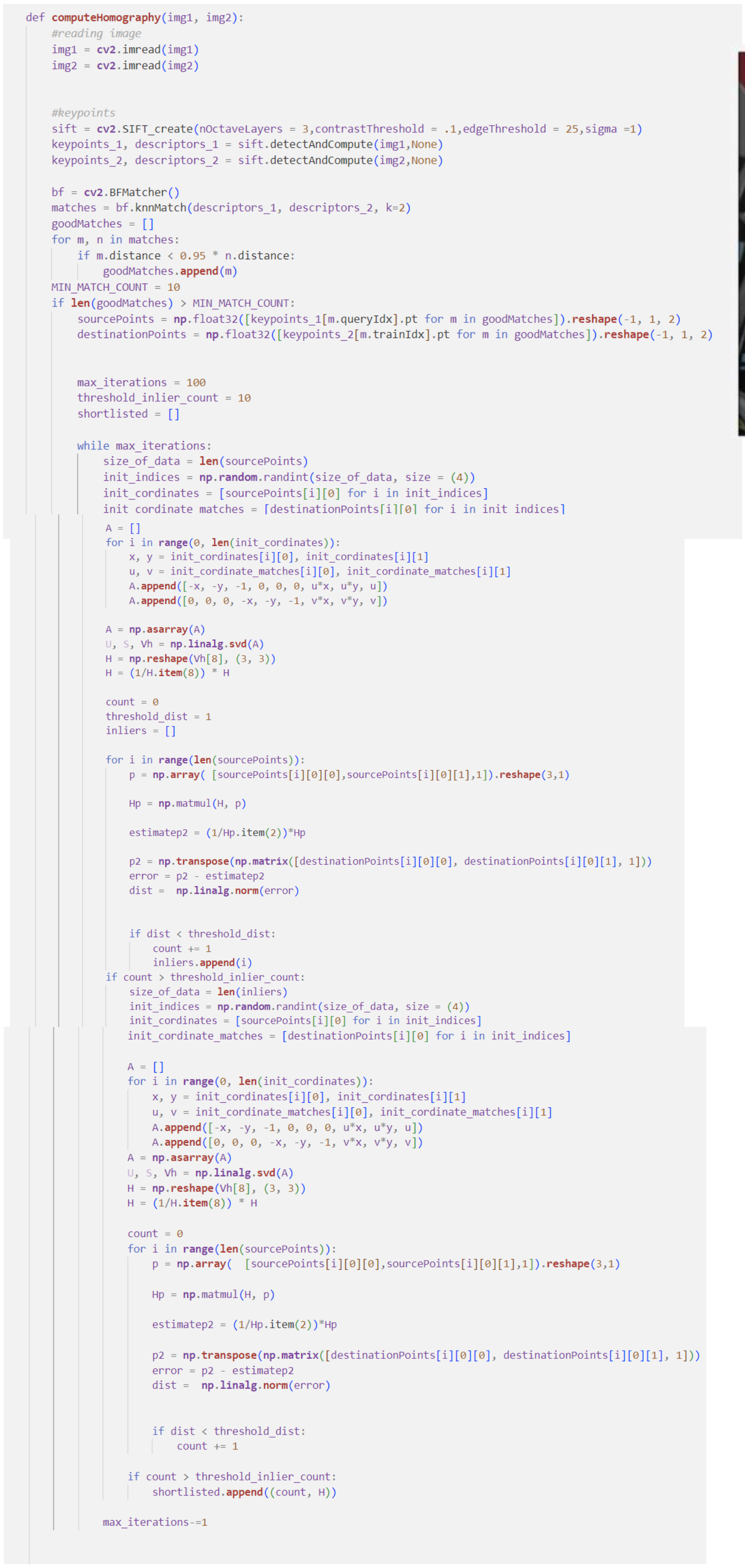
Img1



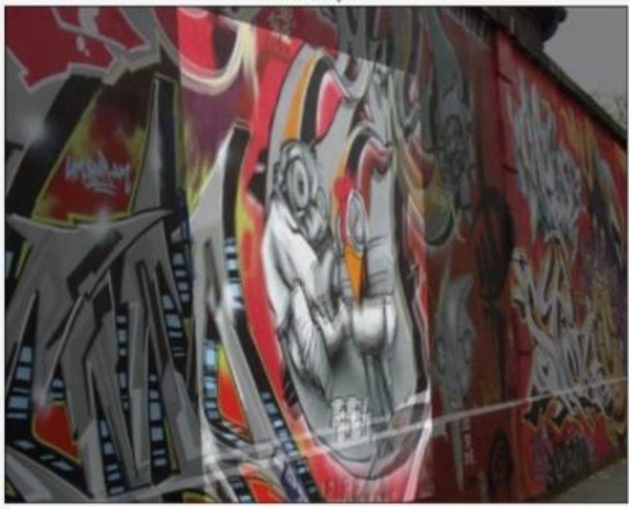
Img5

Warped Image





```python
def computeHomography(img1, img2):
    #reading image
    img1 = cv2.imread(img1)
    img2 = cv2.imread(img2)


    #keypoints
    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 = cv2.BFMatcher()
    matches = bf.knnMatch(descriptors_1, descriptors_2, 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([keypoints_1[m.queryIdx].pt for m in goodMatches]).reshape(-1, 1, 2)
        destinationPoints = np.float32([keypoints_2[m.trainIdx].pt for m in goodMatches]).reshape(-1, 1, 2)


        max_iterations = 100
        threshold_inlier_count = 10
        shortlisted = []

        while max_iterations:
            size_of_data = len(sourcePoints)
            init_indices = np.random.randint(size_of_data, size = (4))
            init_cordinates = [sourcePoints[i][0] for i in init_indices]
            init_cordinate_matches = [destinationPoints[i][0] for i in init_indices]
            A = []
            for i in range(0, len(init_cordinates)):
                x, y = init_cordinates[i][0], init_cordinates[i][1]
                u, v = init_cordinate_matches[i][0], init_cordinate_matches[i][1]
                A.append([-x, -y, -1, 0, 0, 0, u*x, u*y, u])
                A.append([0, 0, 0, -x, -y, -1, v*x, v*y, v])

            A = np.asarray(A)
            U, S, Vh = np.linalg.svd(A)
            H = np.reshape(Vh[8], (3, 3))
            H = (1/H.item(8)) * H

            count = 0
            threshold_dist = 1
            inliers = []

            for i in range(len(sourcePoints)):
                p = np.array( [sourcePoints[i][0][0],sourcePoints[i][0][1],1]).reshape(3,1)

                Hp = np.matmul(H, p)

                estimatep2 = (1/Hp.item(2))*Hp

                p2 = np.transpose(np.matrix([destinationPoints[i][0][0], destinationPoints[i][0][1], 1]))
                error = p2 - estimatep2
                dist =  np.linalg.norm(error)


                if dist < threshold_dist:
                    count += 1
                    inliers.append(i)
            if count > threshold_inlier_count:
                size_of_data = len(inliers)
                init_indices = np.random.randint(size_of_data, size = (4))
                init_cordinates = [sourcePoints[i][0] for i in init_indices]
                init_cordinate_matches = [destinationPoints[i][0] for i in init_indices]

                A = []
                for i in range(0, len(init_cordinates)):
                    x, y = init_cordinates[i][0], init_cordinates[i][1]
                    u, v = init_cordinate_matches[i][0], init_cordinate_matches[i][1]
                    A.append([-x, -y, -1, 0, 0, 0, u*x, u*y, u])
                    A.append([0, 0, 0, -x, -y, -1, v*x, v*y, v])
                A = np.asarray(A)
                U, S, Vh = np.linalg.svd(A)
                H = np.reshape(Vh[8], (3, 3))
                H = (1/H.item(8)) * H

                count = 0
                for i in range(len(sourcePoints)):
                    p = np.array(  [sourcePoints[i][0][0],sourcePoints[i][0][1],1]).reshape(3,1)

                    Hp = np.matmul(H, p)

                    estimatep2 = (1/Hp.item(2))*Hp

                    p2 = np.transpose(np.matrix([destinationPoints[i][0][0], destinationPoints[i][0][1], 1]))
                    error = p2 - estimatep2
                    dist =  np.linalg.norm(error)


                    if dist < threshold_dist:
                        count += 1

                if count > threshold_inlier_count:
                    shortlisted.append((count, H))

            max_iterations-=1
```


Final Output

The Homography calculated using RANSAC isn’t perfect. As the homography matrix computed from the code is different from the homography matrix given in the dataset, the stitched image doesn’t result in a perfect outcome. To emphasize the stitched image1 onto image5 contrast of image5 is slightly lowered.

GitHub link -

https://github.com/TreshanAyesh/EN2550-Fundamentals-of-Image-Processing/tree/main/Assignments/Assignment2