Name: K.K.D.Kariyawasam

Index: 200289U

Github link: <https://github.com/KavinduKariyawasam/Fitting-and-Alignment>

EN3160 - Image Processing and Machine Vision

Assignment 02 : [Fitting and Alignment](https://online.uom.lk/mod/assign/view.php?id=342349)

1. This code used to detect the blob is as below. In here, scikit-image library is used to detect and visualize blobs in the image. Detected blobs are marked as red circles on the original grayscale image. By iterating through the detected blobs, the largest blob was found, and the parameters are printed. Those parameters are as follows.

Largest Blob Parameters:

Center (x, y): (491.0, 195.0)

Radius: 42.42640687119285

Area: 30.0

from skimage.feature import blob\_log



Figure 1: Output image of the blob detection

import matplotlib.pyplot as plt

import numpy as np

*# Perform blob detection using LOG method*

blobs = blob\_log(sample\_b, max\_sigma=30, threshold=0.01)

largest\_blob = None

largest\_area = -1

*# Iterate through the detected blobs to find the largest blob*

for blob in blobs:

    y, x, area = blob

    if area > largest\_area:

        largest\_area = area

        largest\_blob = blob

*# Getting parameters of the largest blob*

if largest\_blob is not None:

    y, x, largest\_area = largest\_blob

    radius = largest\_area \* np.sqrt(2)  *# Calculating the radius*

1. **a)** **Line estimation with RANSAC algorithm**

The following code estimates a line to the given set of noisy points using the RANSAC algorithm. It was done by randomly selecting two points and estimating a line model. Then iteratively refining the model by considering inliers within a certain threshold. Finally, the best line model and the data points were plotted, and the inliers were highlighted.

import math

from scipy.optimize import minimize

N = X.shape[0]

dataset = X

**def** line\_equation\_from\_points(x1, y1, x2, y2):

*# Calculate the direction vector (Δx, Δy)*

    delta\_x = x2 - x1

    delta\_y = y2 - y1

    magnitude = math.sqrt(delta\_x\*\*2 + delta\_y\*\*2)

    a = delta\_y / magnitude

    b = -delta\_x / magnitude

    d = (a \* x1) + (b \* y1)

    return a, b, d

**def** line\_tls(x, indices):

    a, b, d = x[0], x[1], x[2]

    return np.sum(np.square(a\*dataset[indices,0] + b\*dataset[indices,1] - d))

**def** g(x):

    return x[0]\*\*2 + x[1]\*\*2 - 1

cons = ({'type': 'eq', 'fun': g})

**def** consensus\_line(X, x, t):

    a, b, d = x[0], x[1], x[2]

    error = np.absolute(a\*dataset[:,0] + b\*dataset[:,1] - d)

    return error < t

threshold = 1.

required\_inliers = 0.4\*N

min\_data\_points = 2

inliers\_line = []

max\_iterations = 50

iteration = 0

best\_model\_line = []

best\_error = np.inf

best\_sample\_line = []

res\_only\_with\_sample = []

best\_inliers\_line = []

while iteration < max\_iterations:

    indices = np.random.randint(0, N, min\_data\_points) *# A sample of three (s) points selected at random*

    x0 = np.array([1, 1, 0]) *# Initial estimate*

    res = minimize(fun = line\_tls, args = indices, x0 = x0, tol= 1e-6, constraints=cons, options={'disp': True})

    inliers\_line = consensus\_line(dataset, res.x, threshold) *# Computing the inliers*

    if inliers\_line.sum() > required\_inliers:

        x0 = res.x

*# Using inliers computing the new model*

        res = minimize(fun = line\_tls, args = inliers\_line, x0 = x0, tol= 1e-6, constraints=cons, options={'disp': True})

        if res.fun < best\_error:

            best\_model\_line = res.x

            best\_eror = res.fun

            best\_sample\_line = dataset[indices,:]

            res\_only\_with\_sample = x0

            best\_inliers\_line = inliers\_line

    iteration += 1

**b)** **Circle estimation with RANSAC algorithm**

The following code employs circle estimation using the RANSAC algorithm. It was done by subtracting the consensus inliers of the detected line model and then iteratively fits circles to subsets of the remaining data.

*# Subtract the consensus of the best line (remnant)*

line\_inliers = dataset[best\_inliers\_line]

X\_remnant = dataset[~best\_inliers\_line]

*# RANSAC parameters for circle estimation*

max\_iterations\_circle = 100

inlier\_threshold\_circle = 0.5  *# Adjust this threshold as needed*

min\_inliers\_circle = 3

*# Function to estimate circle parameters [x, y, r] from points*

**def** estimate\_circle(points):

*# Define the objective function for circle fitting*

**def** circle\_objective(params, points):

        x, y, r = params

        return np.sum((points[:, 0] - x)\*\*2 + (points[:, 1] - y)\*\*2 - r\*\*2)\*\*2

*# Initialize the optimizer with an initial guess for the circle parameters*

    initial\_guess = [2, 2, 5]  *# Adjust the initial guess as needed*

    result = minimize(circle\_objective, initial\_guess, args=(points,), method='Nelder-Mead')

    x, y, r = result.x

    return x, y, r

*# Function to calculate the radial error (distance from points to the circle)*

**def** circle\_error(params, points):

    x, y, r = params

    distances = np.abs(np.sqrt((points[:, 0] - x)\*\*2 + (points[:, 1] - y)\*\*2) - r)

    return distances

*# RANSAC algorithm for circle estimation on the remnant*

best\_circle = None

best\_inliers\_circle = 0

for \_ in range(max\_iterations\_circle):

*# Randomly select three points*

    random\_indices = np.random.choice(len(X\_remnant), 3, replace=False)

    random\_points = X\_remnant[random\_indices]

*# Estimate the circle parameters [x, y, r]*

    x, y, r = estimate\_circle(random\_points)

*# Calculate the radial error (distance from points to the circle)*

    errors = circle\_error([x, y, r], X\_remnant)

*# Count inliers (points that are within the threshold)*

    inliers = np.sum(errors < inlier\_threshold\_circle)

    if inliers >= min\_inliers\_circle and inliers > best\_inliers\_circle:

        best\_circle = [x, y, r]

        best\_inliers\_circle = inliers

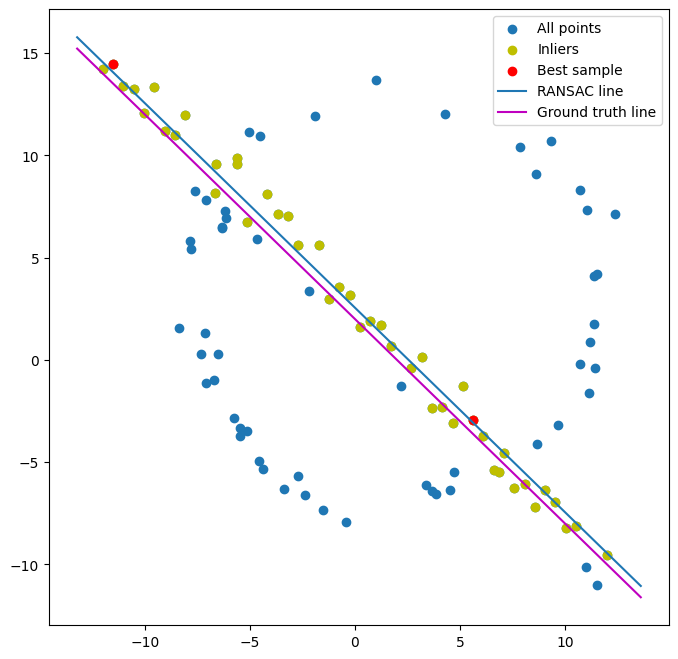
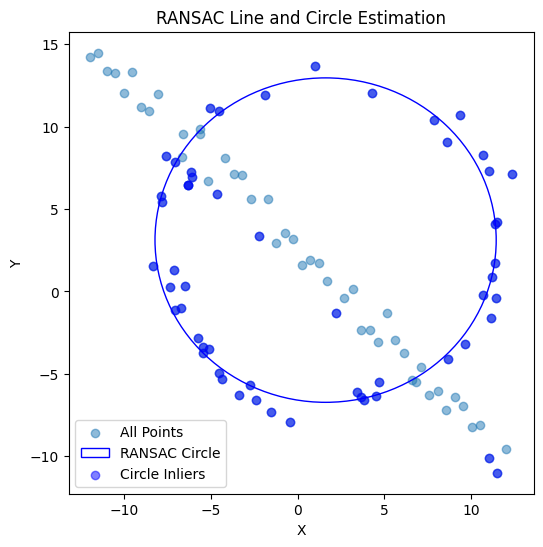
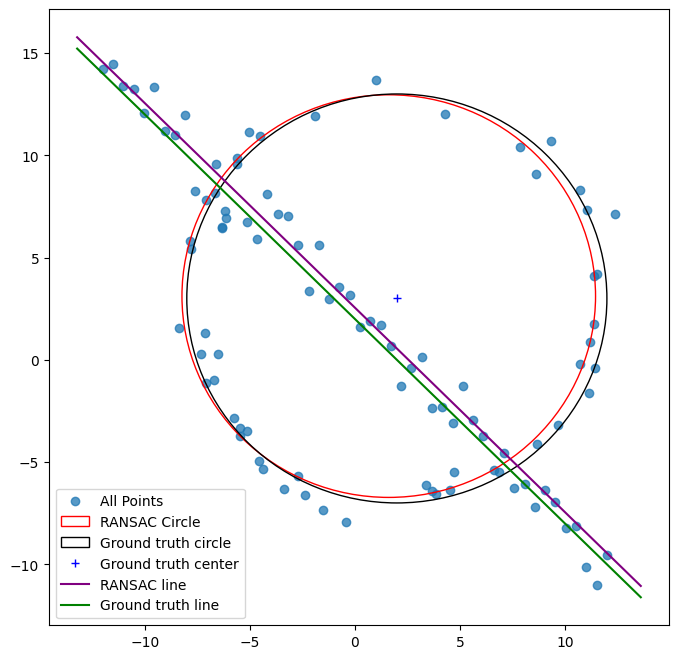


Figure 4:Single plot containing RANSAC estimations and ground truths.

Figure 3: RANSAC circle estimation

Figure 2: RANSAC line estimation and ground truth line

**Discussion:**

According to the results obtained we can see that the line and the circle estimated using the RANSAC algorithm is almost same as the ground truth line and circle. The RANSAC algorithm performs a robust estimation for the line and the circle.

**c)** **What will happen if we fit the circle first?**

If we find the circle first, algorithm might think take some points of the line as a part of circle with a big radius. This is because part of a circle with a large radius can be approximated as a line. This can affect the finding of real line and the circle. So, it is better to start with the line first.

1. When implementing this task, used *setMouseCallbacks* function in OpenCV library to select four points from the background image by clicking on the image. Then using the *findHomography* function it computes a homography matrix to map the other image (here Sri Lankan flag) onto selected points. The resulted superimposed image was then plotted.

import cv2

import numpy as np

*# Function to handle mouse click events*

**def** mouse\_callback(event, x, y, flags, param):

**global** points

    if event == cv2.EVENT\_LBUTTONDOWN:

        if len(points) < 4:

            points.append((x, y))

            cv2.circle(image\_copy, (x, y), 5, (0, 0, 255), -1)

            cv2.imshow("Select Points", image\_copy)

            if len(points) == 4:

                compute\_homography()

*# Function to compute homography and place the flag*

**def** compute\_homography():

**global** points

    if len(points) == 4:

        architectural\_points = np.array(points, dtype=np.float32)

        flag\_points = np.array([[0, 0], [flag\_image.shape[1], 0], [flag\_image.shape[1], flag\_image.shape[0]], [0, flag\_image.shape[0]]], dtype=np.float32)

        homography\_matrix, \_ = cv2.findHomography(flag\_points, architectural\_points)

        flag\_warped = cv2.warpPerspective(flag\_image, homography\_matrix, (image.shape[1], image.shape[0]))

        result = cv2.addWeighted(image, 1, flag\_warped, 0.7, 0)

        cv2.imshow("Result", result)

        plt.imshow(cv2.cvtColor(result, cv2.COLOR\_BGR2RGB) )

image = cv2.imread('background.jpeg')

flag\_image = cv2.imread('flag.jpg')

image = cv2.resize(image, (1000, 600))

*# Create a copy of the image for point selection*

image\_copy = image.copy()

*# Create a window for point selection*

cv2.namedWindow("Select Points")

cv2.setMouseCallback("Select Points", mouse\_callback)

*# List to store selected points*

points = []

*# Main loop*

while True:

    cv2.imshow("Select Points", image\_copy)

    key = cv2.waitKey(1) & **0x**FF

    if key == ord("q"):

        break

cv2.destroyAllWindows()

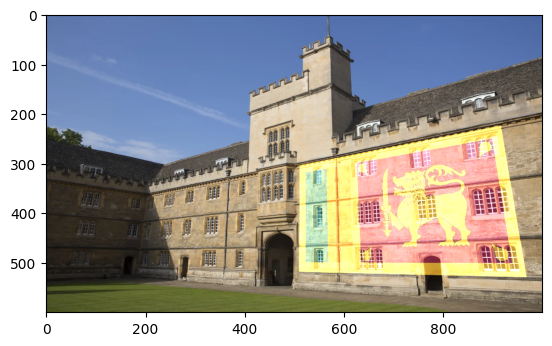
Results were as follows.

Figure 5: Superimposed image

**Discussion:**

When I was choosing the images, I took the background image to be a building with generally a flat surface. This makes it easy to superimpose the flag on the building.

1. **a)** SIFT was used to compute and match the features of the two images and after performing the feature matching lines were drawn between the matching key points of the two images.

import cv2 as cv

import matplotlib.pyplot as plt

im1, im5 = cv.imread("graffiti images\img1.ppm"), cv.imread("graffiti images\img5.ppm")

sift = cv.SIFT\_create()

key\_points\_1, descriptors\_1 = sift.detectAndCompute(im1,None) *#sifting*

key\_points\_2, descriptors\_2 = sift.detectAndCompute(im5,None)

bf\_match = cv.BFMatcher(cv.NORM\_L1, crossCheck=True)  *#feature matching*

matches = sorted(bf\_match.match(descriptors\_1, descriptors\_2), key = **lambda** x:x.distance)

im = cv.drawMatches(im1, key\_points\_1, im5, key\_points\_2, matches[:250], im5, flags=cv.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS)  *#draw lines between the matching features of two images*

fig, ax = plt.subplots(figsize=(10,10))

im = cv.cvtColor(im, cv.COLOR\_BGR2RGB)

ax.set\_title("SIFT Features"), ax.imshow(im)

plt.show()

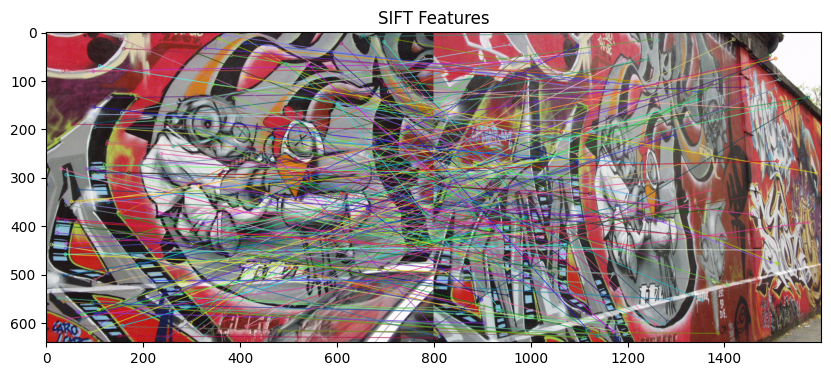


Figure 6: SIFT features.

**b), c)** In this task two graffiti images were stitched using RANSAC. To tackle this task first random numbers were generated to select random feature correspondences. Then the Homography matrices were calculated. Iterate through the five images and for each pair of images, SIFT features are matched to find the best Homography matrix. Finally, these matrices are combined, and the final transformation was created and image 1 and image 5 are stitched.

*# RANSAC parameters*

p\_value = 0.99

s = 4

epsilon = 0.5

N = int(np.ceil(np.log(1 - p\_value) / np.log(1 - ((1 - epsilon) \*\* s))))

Hs = []

*# Loop through the images and compute Homography matrices*

for i in range(4):

    sift = cv.SIFT\_create()

    key\_points\_1, descriptors\_1 = sift.detectAndCompute(gray\_imgs[i], None)

    key\_points\_2, descriptors\_2 = sift.detectAndCompute(gray\_imgs[i + 1], None)

    bf\_match = cv.BFMatcher(cv.NORM\_L1, crossCheck=True)

    matches = sorted(bf\_match.match(descriptors\_1, descriptors\_2), key=**lambda** x: x.distance)

    source\_points = [key\_points\_1[k.queryIdx].pt for k in matches]

    destination\_points = [key\_points\_2[k.trainIdx].pt for k in matches]

    threshold, best\_inliers, best\_H = 2, 0, 0

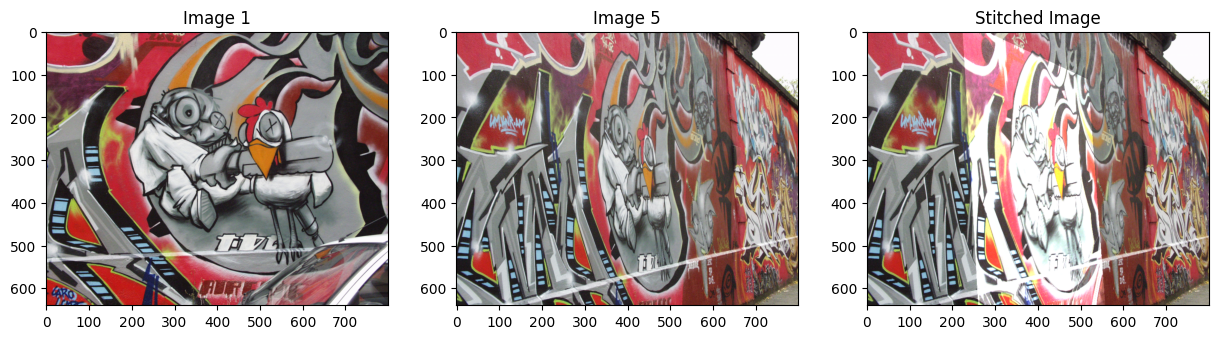
    for j in range(N):

        random\_indices = random\_number(len(source\_points) - 1, 4)

        sampled\_source\_points = []

        sampled\_destination\_points = []

        for k in range(4):

            sampled\_source\_points.append(np.array([[source\_points[random\_indices[k]][0], source\_points[random\_indices[k]][1], 1]]))

            sampled\_destination\_points.append(destination\_points[random\_indices[k]][0])

            sampled\_destination\_points.append(destination\_points[random\_indices[k]][1])

        H = compute\_homography(sampled\_source\_points, sampled\_destination\_points)

        inliers = 0

        for k in range(len(source\_points)):

            X = [source\_points[k][0], source\_points[k][1], 1]

            HX = H @ X

            HX /= HX[-1]

            err = np.sqrt(np.power(HX[0] - destination\_points[k][0], 2) + np.power(HX[1] - destination\_points[k][1], 2))

            if err < threshold:

                inliers += 1

        if inliers > best\_inliers:

            best\_inliers = inliers

            best\_H = H

    Hs.append(best\_H)

*# Compute the final Homography matrix*

H1\_to\_5 = Hs[3] @ Hs[2] @ Hs[1] @ Hs[0]

H1\_to\_5 /= H1\_to\_5[-1][-1]

Figure 7: Final output

**Output :**

Computed Homography = [[ 6.32263200e-01 4.84235039e-02 2.22075179e+02]

[ 2.29731111e-01 1.14414839e+00 -2.46178043e+01]

[ 5.09689102e-04 -7.58547765e-05 1.00000000e+00]]

Provided 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

**Discussion:**

The computed Homography and the provided Homography matrices are not much different. It is a satisfactory result, and the final stitched image is aligned and blended reasonably well.