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

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
```

```python
from skimage.feature import blob_log
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
```

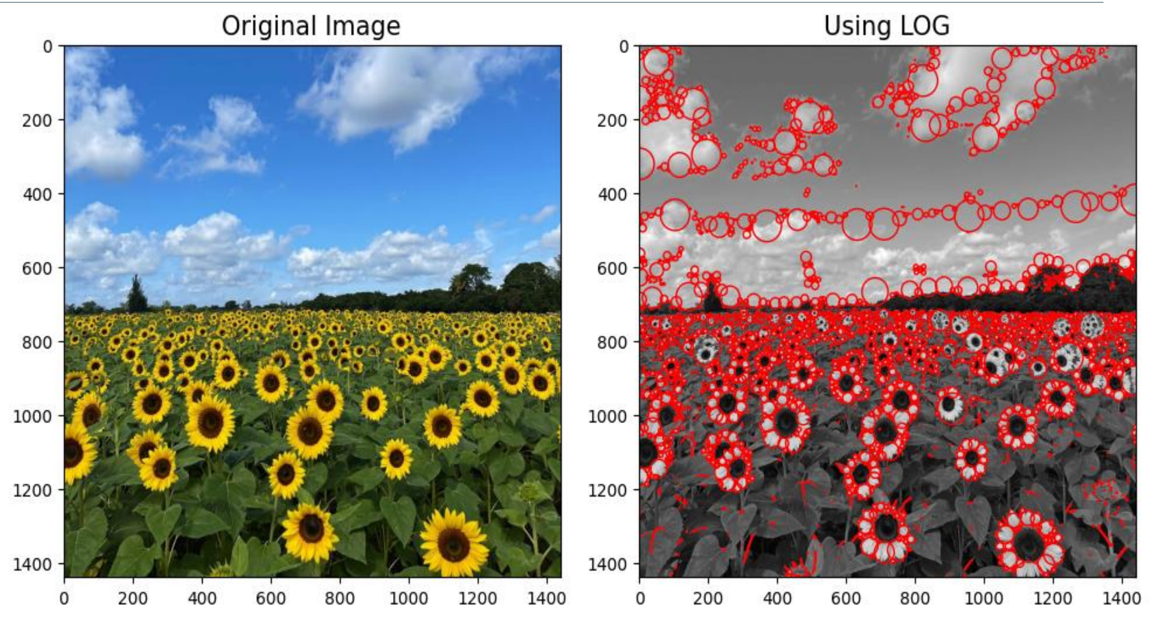

Figure 1: Output image of the blob detection

2. 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.

```python
 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
```

```python
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
```

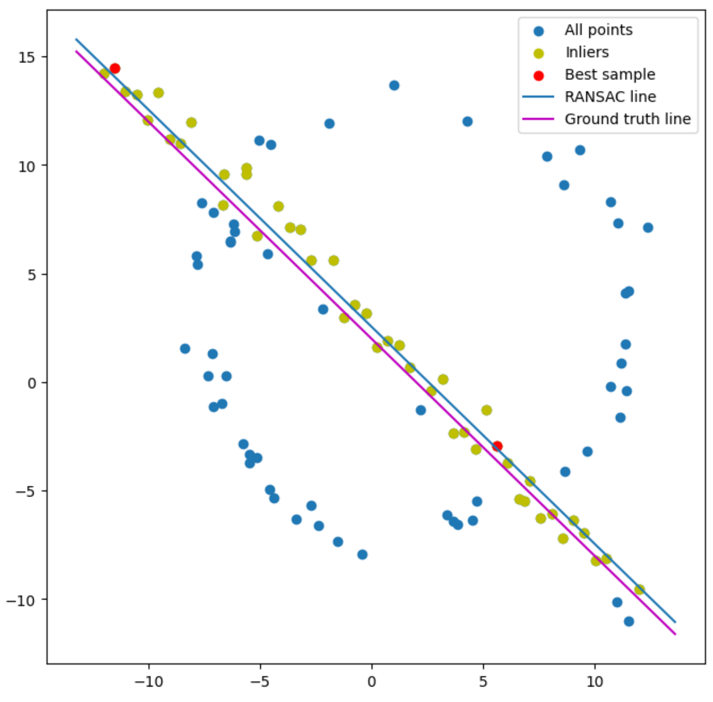

Figure 2: RANSAC line estimation and ground truth line

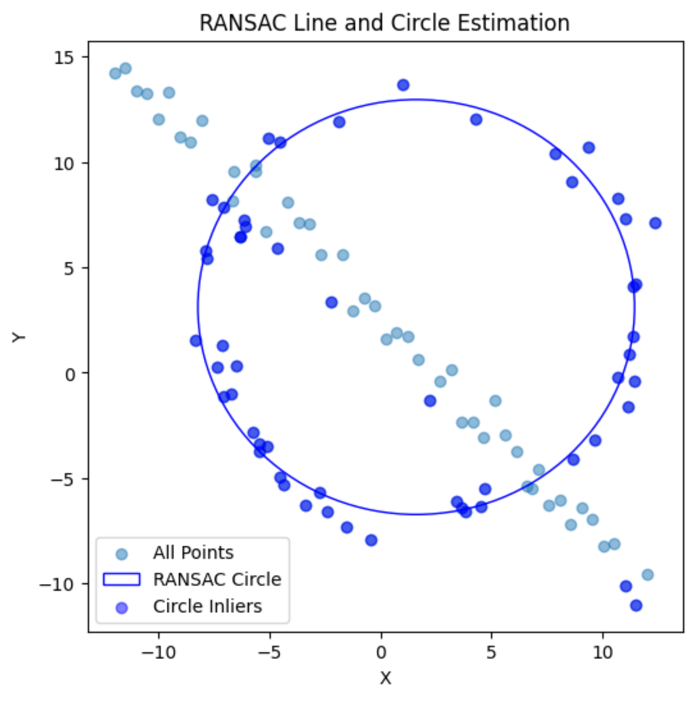

Figure 3: RANSAC circle estimation

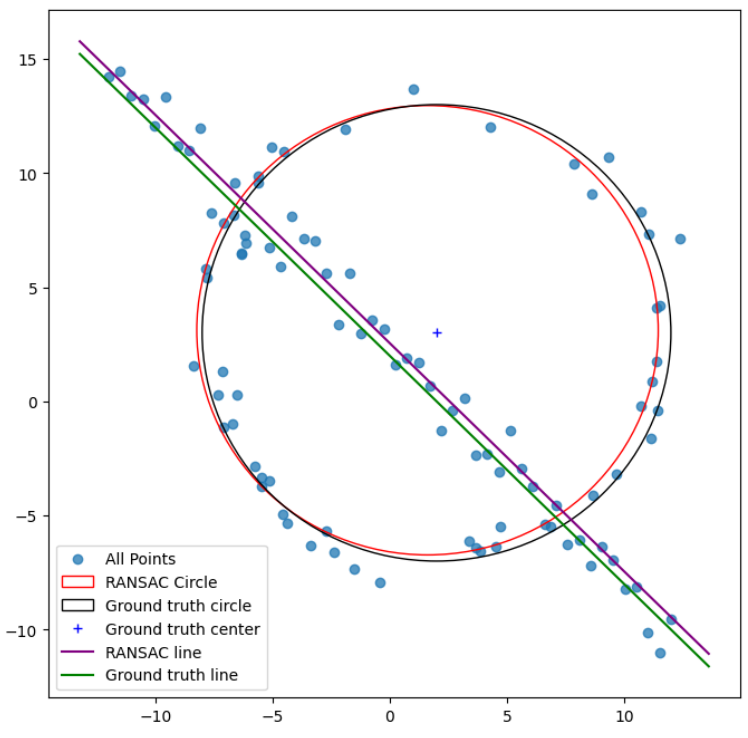

Figure 4:Single plot containing RANSAC estimations and ground truths.

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.

3. 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) & 0xFF
    if key == ord("q"):
        break

cv2.destroyAllWindows()
```

Results were as follows.

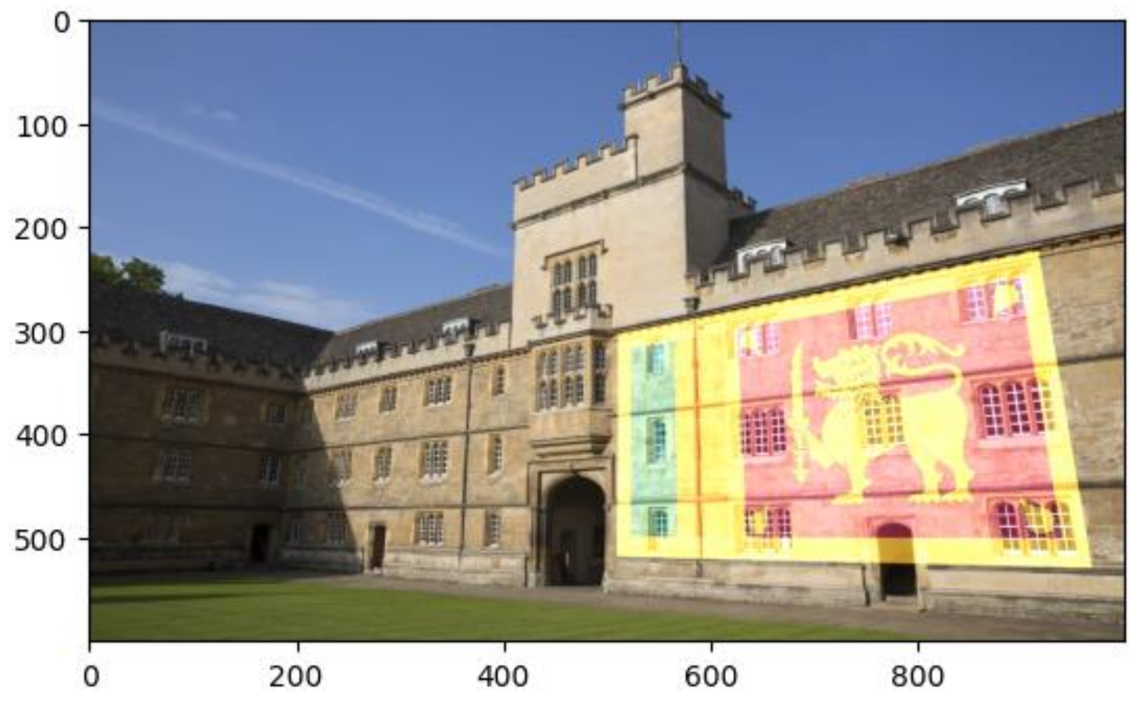

Figure 5: Superimposed image

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.

4. **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()
```

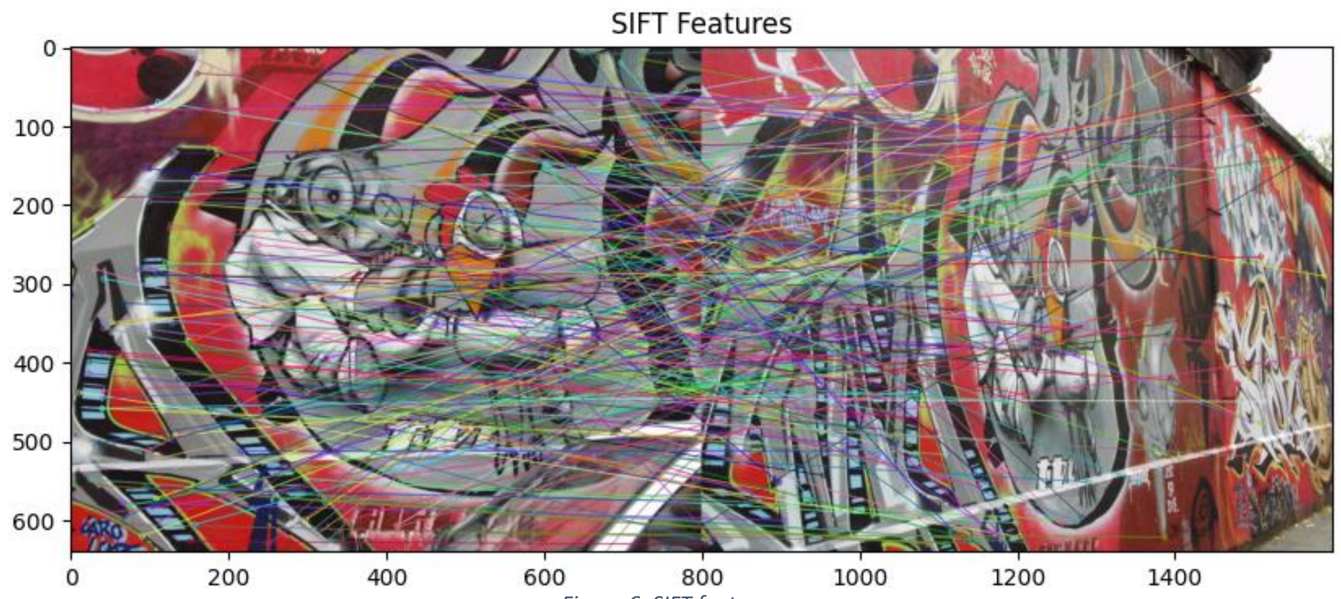
Figure 6: SIFT features.

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_i
ndices[k]][0], source_points[random_indices[k]][1], 1]]))
            sampled_destination_points.append(destination_points[random_in
dices[k]][0])
            sampled_destination_points.append(destination_points[random_in
dices[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]
```

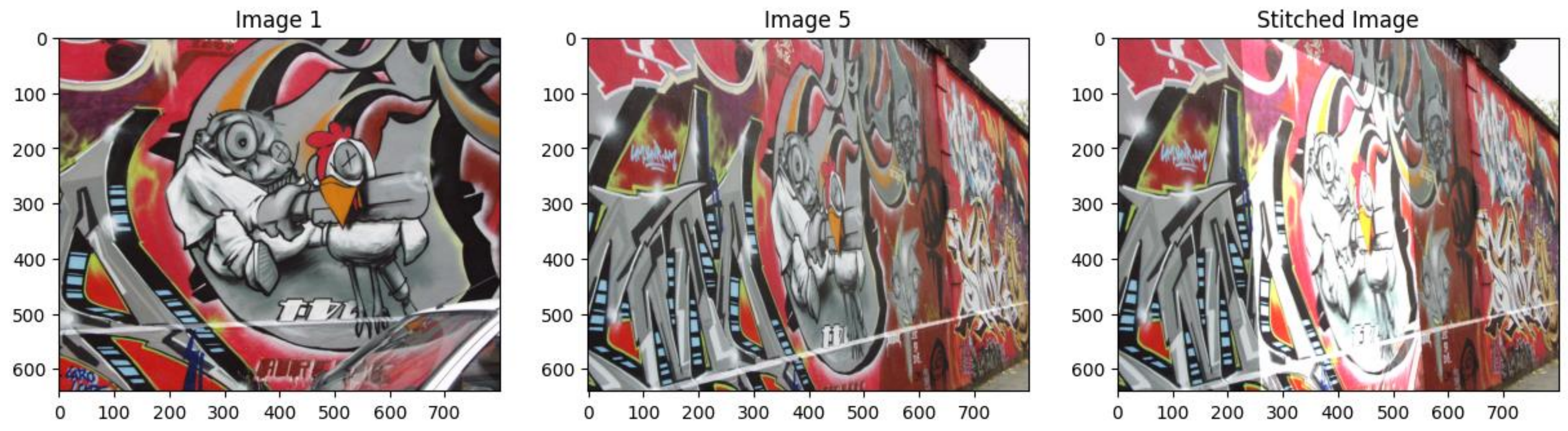

Figure 7: Final output

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.