Electronic and Telecommunication engineering

University of Moratuwa

EN3160 - Image Processing and Machine Vision

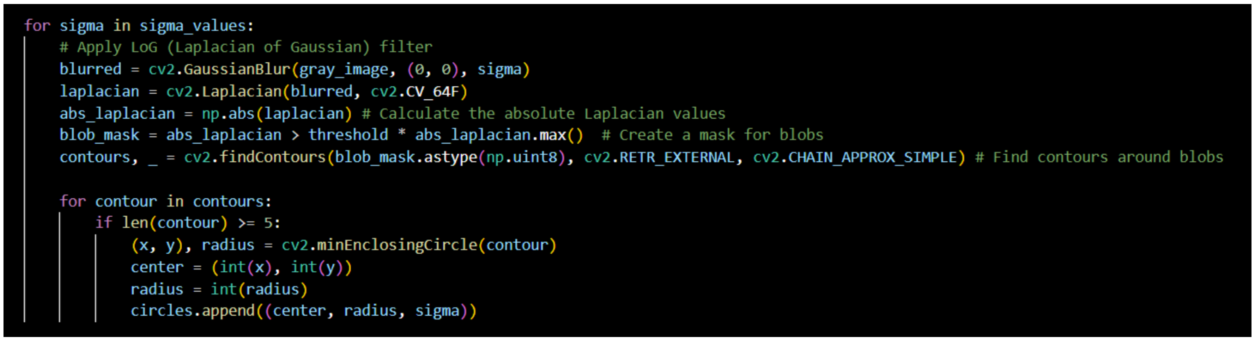
A02 Fitting and Alignment

RANAVIRAJA R.W.H.M.T.A. - Index No. 200520X

GitHub: GitHub

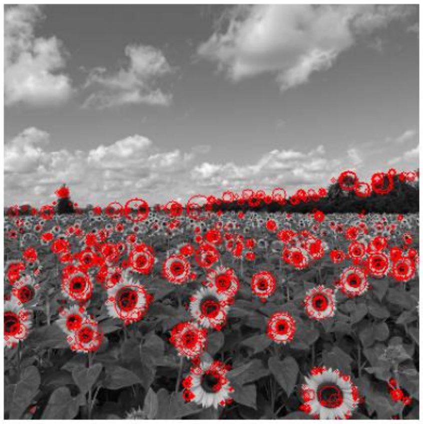
04th October, 2023

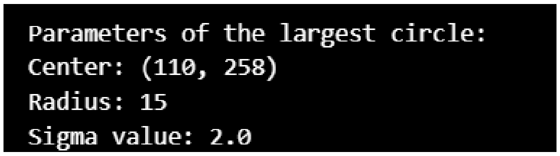
Q1.



```python
for sigma in sigma_values:
    # Apply LoG (Laplacian of Gaussian) filter
    blurred = cv2.GaussianBlur(gray_image, (0, 0), sigma)
    laplacian = cv2.Laplacian(blurred, cv2.CV_64F)
    abs_laplacian = np.abs(laplacian) # Calculate the absolute Laplacian values
    blob_mask = abs_laplacian > threshold * abs_laplacian.max()  # Create a mask for blobs
    contours, _ = cv2.findContours(blob_mask.astype(np.uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) # Find contours around blobs

    for contour in contours:
        if len(contour) >= 5:
            (x, y), radius = cv2.minEnclosingCircle(contour)
            center = (int(x), int(y))
            radius = int(radius)
            circles.append((center, radius, sigma))
```

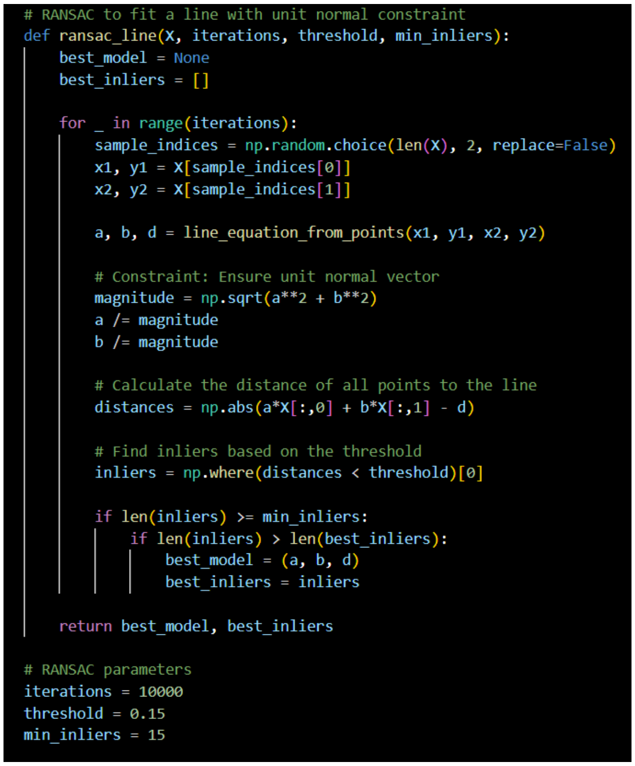
Detected Circles





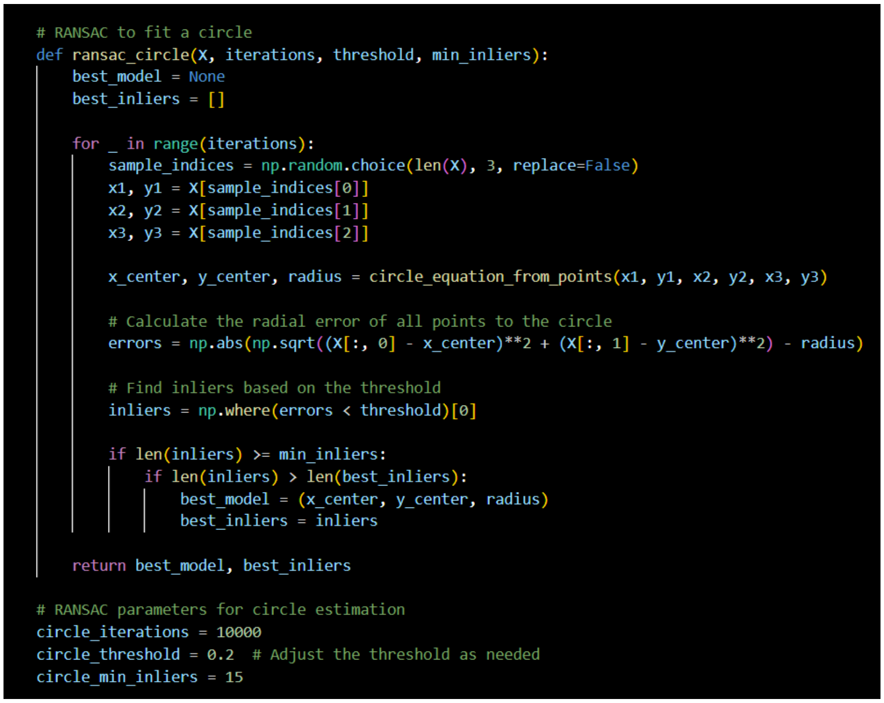
```
Parameters of the largest circle:
Center: (110, 258)
Radius: 15
Sigma value: 2.0
```

Q2. (a)



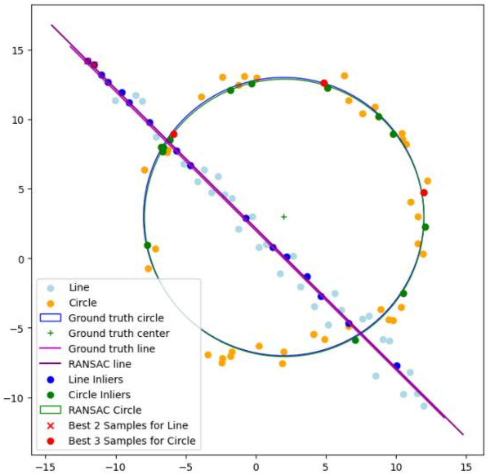
```python
# RANSAC to fit a line with unit normal constraint
def ransac_line(X, iterations, threshold, min_inliers):
    best_model = None
    best_inliers = []

    for _ in range(iterations):
        sample_indices = np.random.choice(len(X), 2, replace=False)
        x1, y1 = X[sample_indices[0]]
        x2, y2 = X[sample_indices[1]]

        a, b, d = line_equation_from_points(x1, y1, x2, y2)

        # Constraint: Ensure unit normal vector
        magnitude = np.sqrt(a**2 + b**2)
        a /= magnitude
        b /= magnitude

        # Calculate the distance of all points to the line
        distances = np.abs(a*X[:,0] + b*X[:,1] - d)

        # Find inliers based on the threshold
        inliers = np.where(distances < threshold)[0]

        if len(inliers) >= min_inliers:
            if len(inliers) > len(best_inliers):
                best_model = (a, b, d)
                best_inliers = inliers

    return best_model, best_inliers

# RANSAC parameters
iterations = 10000
threshold = 0.15
min_inliers = 15
```

(b)



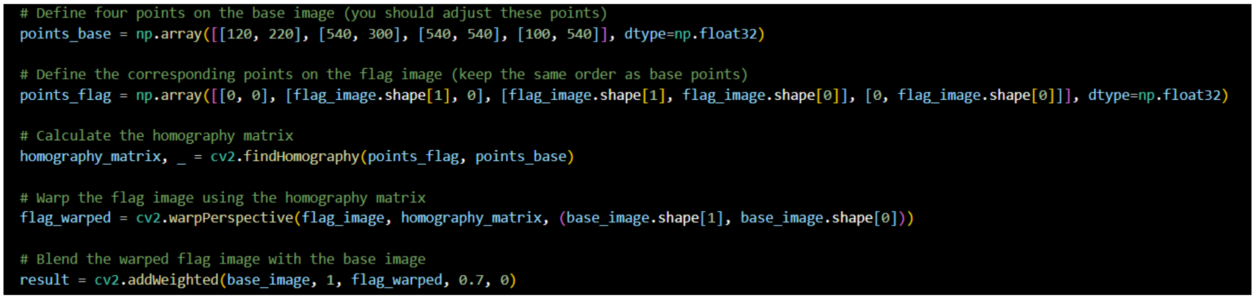
```python
# RANSAC to fit a circle
def ransac_circle(X, iterations, threshold, min_inliers):
    best_model = None
    best_inliers = []

    for _ in range(iterations):
        sample_indices = np.random.choice(len(X), 3, replace=False)
        x1, y1 = X[sample_indices[0]]
        x2, y2 = X[sample_indices[1]]
        x3, y3 = X[sample_indices[2]]

        x_center, y_center, radius = circle_equation_from_points(x1, y1, x2, y2, x3, y3)

        # Calculate the radial error of all points to the circle
        errors = np.abs(np.sqrt((X[:, 0] - x_center)**2 + (X[:, 1] - y_center)**2) - radius)

        # Find inliers based on the threshold
        inliers = np.where(errors < threshold)[0]

        if len(inliers) >= min_inliers:
            if len(inliers) > len(best_inliers):
                best_model = (x_center, y_center, radius)
                best_inliers = inliers

    return best_model, best_inliers

# RANSAC parameters for circle estimation
circle_iterations = 10000
circle_threshold = 0.2  # Adjust the threshold as needed
circle_min_inliers = 15
```

(c)

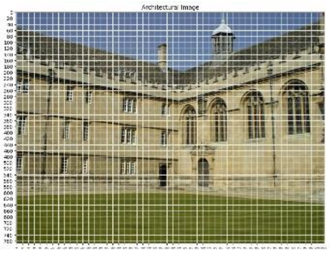


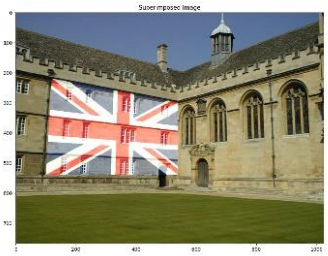
(d) If we fit the circle first, it uses some points which belong to the line. We get a less accurate line after we fit the circle. Therefore, the line should be fit before the circle.

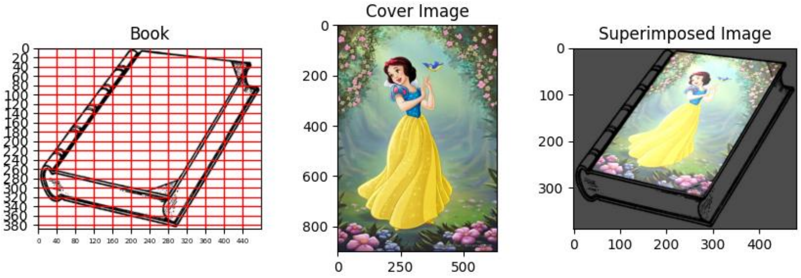
Q3.

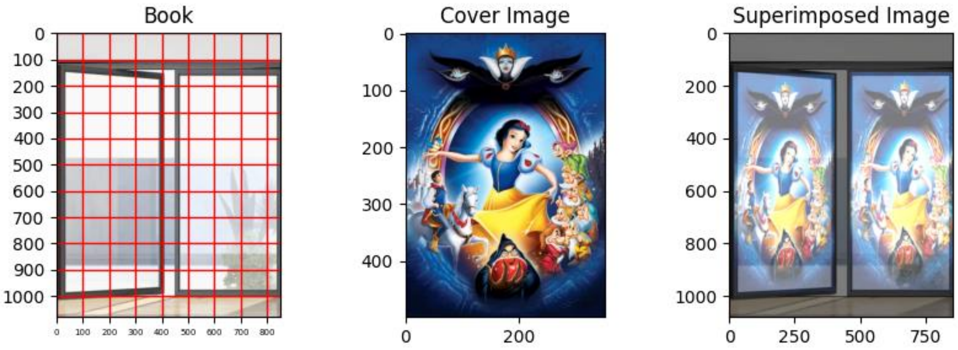


```
# Define four points on the base image (you should adjust these points)
points_base = np.array([[120, 220], [540, 300], [540, 540], [100, 540]], dtype=np.float32)

# Define the corresponding points on the flag image (keep the same order as base points)
points_flag = 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)

# Calculate the homography matrix
homography_matrix, _ = cv2.findHomography(points_flag, points_base)

# Warp the flag image using the homography matrix
flag_warped = cv2.warpPerspective(flag_image, homography_matrix, (base_image.shape[1], base_image.shape[0]))

# Blend the warped flag image with the base image
result = cv2.addWeighted(base_image, 1, flag_warped, 0.7, 0)
```

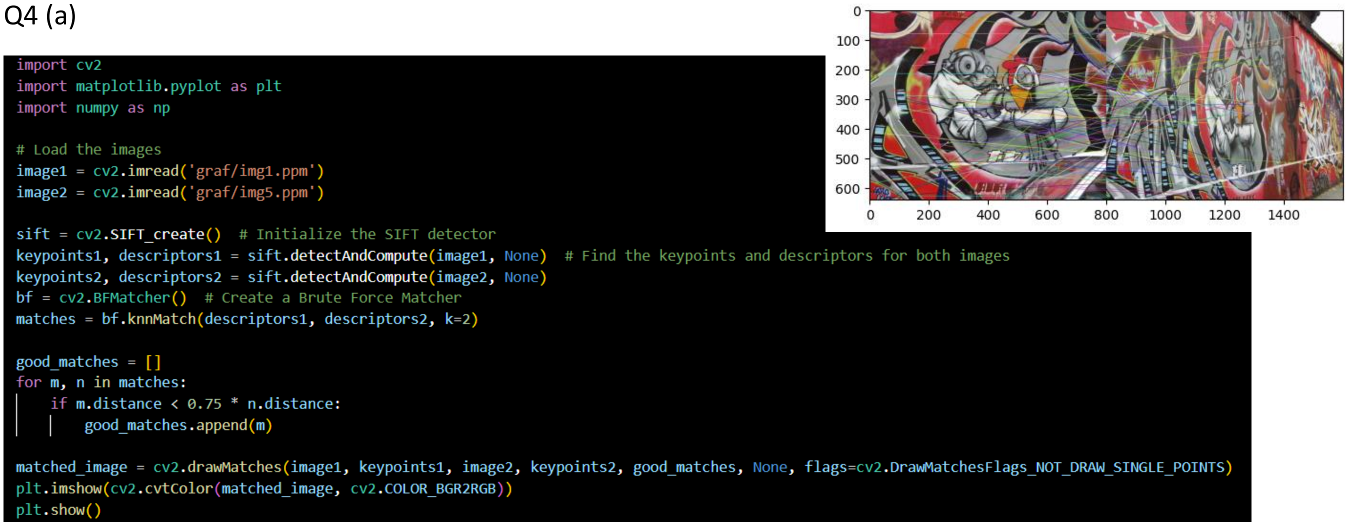






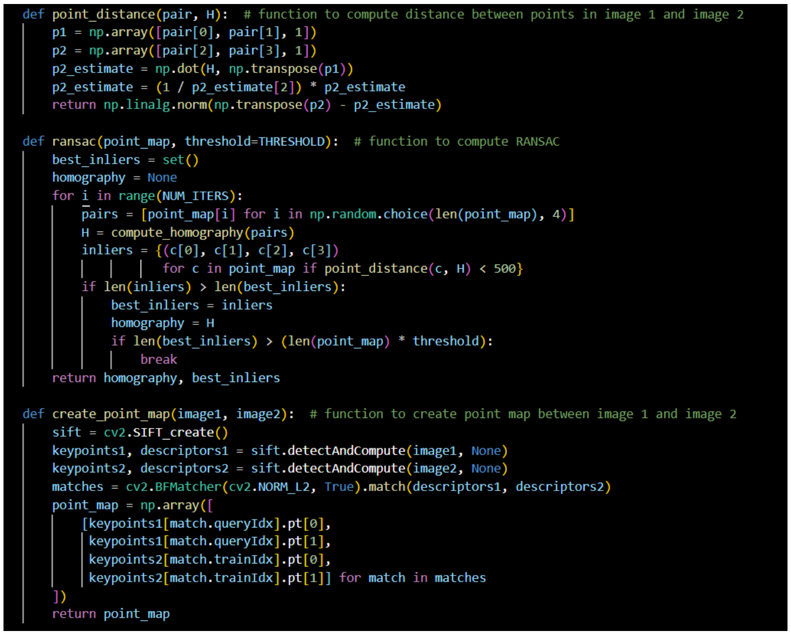


Q4 (a)

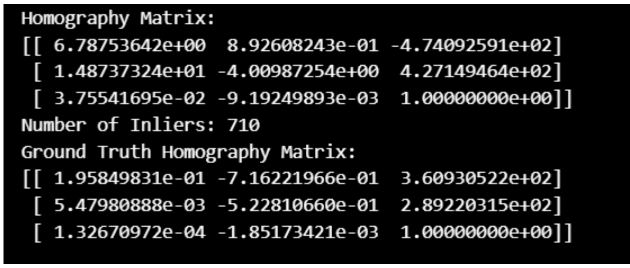


```
import cv2
import matplotlib.pyplot as plt
import numpy as np

# Load the images
image1 = cv2.imread('graf/img1.ppm')
image2 = cv2.imread('graf/img5.ppm')

sift = cv2.SIFT_create()  # Initialize the SIFT detector
keypoints1, descriptors1 = sift.detectAndCompute(image1, None)  # Find the keypoints and descriptors for both images
keypoints2, descriptors2 = sift.detectAndCompute(image2, None)
bf = cv2.BFMatcher()  # Create a Brute Force Matcher
matches = bf.knnMatch(descriptors1, descriptors2, k=2)

good_matches = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good_matches.append(m)

matched_image = cv2.drawMatches(image1, keypoints1, image2, keypoints2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
plt.show()
```

(b)

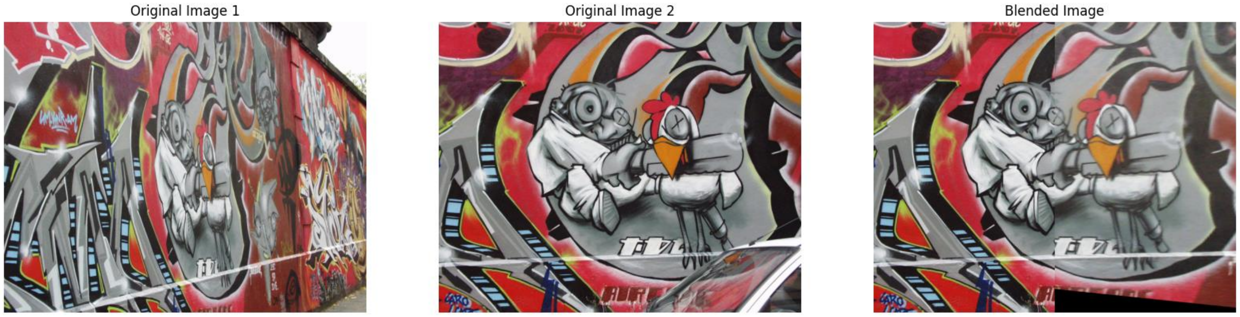


```
def point_distance(pair, H):  # function to compute distance between points in image 1 and image 2
    p1 = np.array([pair[0], pair[1], 1])
    p2 = np.array([pair[2], pair[3], 1])
    p2_estimate = np.dot(H, np.transpose(p1))
    p2_estimate = (1 / p2_estimate[2]) * p2_estimate
    return np.linalg.norm(np.transpose(p2) - p2_estimate)

def ransac(point_map, threshold=THRESHOLD):  # function to compute RANSAC
    best_inliers = set()
    homography = None
    for i in range(NUM_ITERS):
        pairs = [point_map[i] for i in np.random.choice(len(point_map), 4)]
        H = compute_homography(pairs)
        inliers = {(c[0], c[1], c[2], c[3])
                   for c in point_map if point_distance(c, H) < 500}
        if len(inliers) > len(best_inliers):
            best_inliers = inliers
            homography = H
            if len(best_inliers) > (len(point_map) * threshold):
                break
    return homography, best_inliers

def create_point_map(image1, image2):  # function to create point map between image 1 and image 2
    sift = cv2.SIFT_create()
    keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(image2, None)
    matches = cv2.BFMatcher(cv2.NORM_L2, True).match(descriptors1, descriptors2)
    point_map = np.array([
        [keypoints1[match.queryIdx].pt[0],
         keypoints1[match.queryIdx].pt[1],
         keypoints2[match.trainIdx].pt[0],
         keypoints2[match.trainIdx].pt[1]] for match in matches
    ])
    return point_map
```



```
Homography Matrix:
[[ 6.78753642e+00  8.92608243e-01 -4.74092591e+02]
 [ 1.48737324e+01 -4.00987254e+00  4.27149464e+02]
 [ 3.75541695e-02 -9.19249893e-03  1.00000000e+00]]
Number of Inliers: 710
Ground Truth Homography Matrix:
[[ 1.95849831e-01 -7.16221966e-01  3.60930522e+02]
 [ 5.47980888e-03 -5.22810660e-01  2.89220315e+02]
 [ 1.32670972e-04 -1.85173421e-03  1.00000000e+00]]
```

(c)


Original Image 1

Original Image 2

Blended Image