Name: A.S.N. Ranasingha

Index No: 200507N

Assignment 2

Question 01

Here minimum and maximum threshold for edge detection is set to 2 and 100 respectively. **The range of sigma values used is from 1 to 10.** Main code parts are given below.

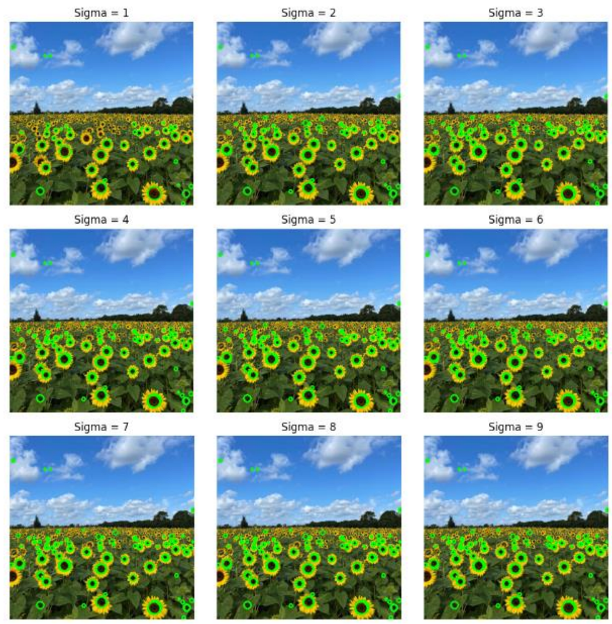
Codes:

```
# Apply GaussianBlur with the current sigma
blurred_image = cv.GaussianBlur(gray_image, (9, 9), sigma)

# Apply Laplacian of Gaussians (LoG) for edge detection
edges = cv.Laplacian(blurred_image, cv.CV_64F)
```

```
# Threshold the LoG edges
_, binary_edges = cv.threshold(edges, min_threshold, max_threshold, cv.THRESH_BINARY)

# Find contours in the binary edges
contours, _ = cv.findContours(binary_edges.astype(np.uint8), cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
```

```
# Filter and draw circles based on the contours
for contour in contours:
    area = cv.contourArea(contour)
    if area < 10 or area > 8000:
        continue

    perimeter = cv.arcLength(contour, True)
    circularity = 4 * np.pi * area / (perimeter ** 2)

    # Filter based on circularity
    if circularity > 0.48:
        (x, y), radius = cv.minEnclosingCircle(contour)
        center = (int(x), int(y))
        radius = int(radius)
        cv.circle(image, center, radius, (0, 255, 0), 2)  # Draw circle perimeter with reduced thickness
```

Output:



Question 02

a) Line estimation using RNASAC algorithm.
Here first I have generated a noisy scattered plot of a circle and a line using a similar code as above. The two plots are generated separately as X_circ and X_line. Then they are combined to create a single plot of scattered points as X_comb. Then the line graph was estimated using the RNASAC algorithm. Several functions were defined in the code as required.

Line Equation

```
def line_equation_from_points(x1, y1, x2, y2):
    # Calculate the direction vector (Δx, Δy)
    delta_x = x2 - x1
    delta_y = y2 - y1

    # Calculate the normalized vector (a, b)
    magnitude = math.sqrt(delta_x**2 + delta_y**2)
    a = delta_y / magnitude
    b = delta_x / magnitude  #b = -delta_x / magnitude

    # Calculate d
    d = (a * x1) + (b * y1)

    # Return the line equation in the form ax + by = d
    return a, b, d
```

Other functions

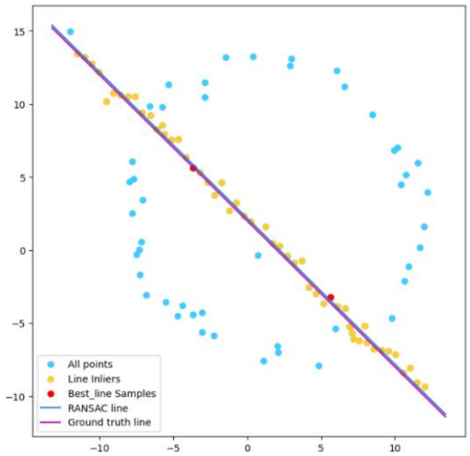
```
# RANSAC to fit a line
def line_tls(x, indices):
    a, b, d = x[0], x[1], x[2]
    return np.sum(np.square(a*X_[indices,0] + b*X_[indices,1] - d))

# Constraint
def g(x):
    return x[0]**2 + x[1]**2 - 1

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

# Computing the consensus (inliers)
def consensus_line(X, x, t):
    a, b, d = x[0], x[1], x[2]
    error = np.absolute(a*X_[:,0] + b*X_[:,1] - d)
    return error < t
```

Here I set threshold to zero, fraction of data points required to assert how well the model fits to 0.5. Then the estimated line and the ground truth line are plotted on the same graph.

<u>Finding best fitting curve</u>

```
while iteration < max_iterations:
    indices = np.random.randint(0, N, s) # 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(X_, res.x, t) # Computing the inliers
    if inliers_line.sum() > d:
        x0 = res.x
        # Computing the new model using the inliers
        res = minimize(fun = line_tls, args = inliers_line, x0 = x0, tol= 1e-6, constraints=cons, options={'disp': True})

        if res.fun < best_error:
            #print('A better model found ... ', res.x, res.fun)
            best_model_line = res.x
            best_eror = res.fun
            best_sample_line = X_[indices,:]
            res_only_with_sample = x0
            best_inliers_line = inliers_line

    iteration += 1
```

<u>Output</u>



b) Then the points identified as line inliers are deducted from X_comb and created a new variable as "circ_inliers" with the remaining data points. It was used to estimate the best fit circle using RNASAC algorithm. Here also several functions were defined as required.

```
def fit_circle(data):
    # Fit a circle to the data using least squares
    X, Y = data[:, 0], data[:, 1]
    A = np.column_stack((2*X, 2*Y, np.ones(data.shape[0])))
    b = X**2 + Y**2
    center_and_radius = np.linalg.lstsq(A, b, rcond=None)[0]
    return center_and_radius

def calculate_error(center, radius, data):
    # Calculate radial errors
    distances = np.sqrt((data[:, 0] - center[0])**2 + (data[:, 1] - center[1])**2)
    errors = np.abs(distances - radius)
    return errors
```

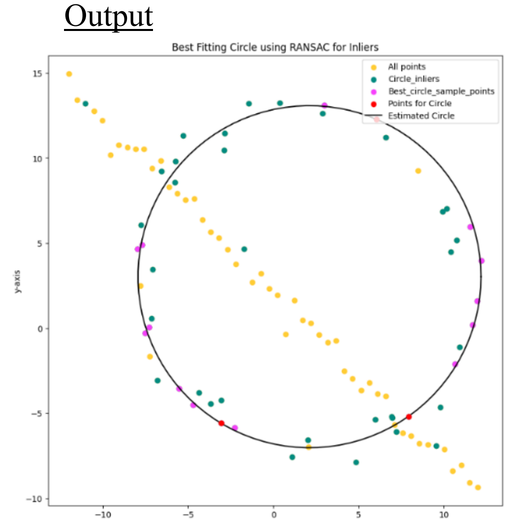
```
def ransac_circle_fit(data, max_iterations, threshold):
    best_circle = None
    best_inliers = None
    max_inliers = 0

    for _ in range(max_iterations):
        # Randomly select 3 inlier points to form a circle
        indices = np.random.choice(data.shape[0], 3, replace=False)
        circle_params = fit_circle(data[indices])
        center = circle_params[:2]
        radius = np.sqrt(circle_params[2] + np.dot(center, center))

        # Calculate errors and find inliers
        errors = calculate_error(center, radius, data)
        inliers = errors < threshold

        # Update the best circle if we have more inliers
        if np.sum(inliers) > max_inliers:
            max_inliers = np.sum(inliers)
            best_circle = (center, radius)
            best_inliers = inliers
            best_circle_indices = indices

    return best_circle, best_inliers, best_circle_indices
```

Then the best fit circle was determined with threshold set to 0.2 and number of iterations set to 1000. The output was then plotted.

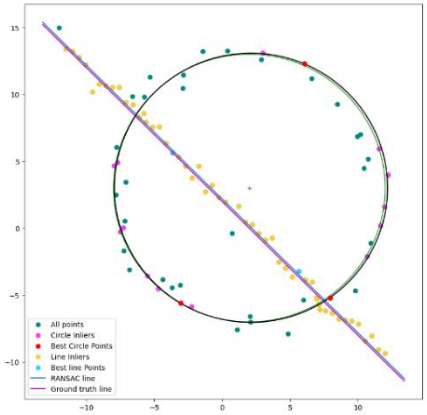
<u>Code</u>

```
# Set RANSAC parameters
max_iterations = 1000
threshold = 0.2

# Run RANSAC to fit the circle
#best_circle, best_inliers = ransac_circle_fit(circle_inlier_points, max_iterations, threshold)
best_circle, best_inliers, best_circle_indices = ransac_circle_fit(circle_inlier_points, max_iterations, threshold)
# Calculate radial errors
radial_errors = calculate_error(best_circle[0], best_circle[1], circle_inlier_points)

# Print the mean error
mean_error = np.mean(radial_errors)
print("Mean radial error:", mean_error)

# Create a circle using the estimated parameters
theta = np.linspace(0, 2 * np.pi, 100)
circle_x = best_circle[0][0] + best_circle[1] * np.cos(theta)
circle_y = best_circle[0][1] + best_circle[1] * np.sin(theta)

#Extract the final three points used to estimate the circle
circle_points = circle_inlier_points[best_circle_indices]
```

<u>Output</u>



c) Then noisy data points plot, ground truth curves and, estimated curves are plotted on the same graph.



d) Fitting the circle first before the line may not give an accurate line estimate. The circle fitting doesn't consider the line's properties, so the line estimate may be incorrect when fitted afterward. Order of fitting matters for accurate results.

Question 03

Here, we are superimposing an image on another image. To select the coordinates to points where the second image should be placed, the user is given chance to select the four points manually.

```
def get_user_coordinates(image):
    # Function to get user-selected coordinates
    coordinates = []  # Initialize coordinates list

    def click_event(event, x, y, flags, param):
        if event == cv.EVENT_LBUTTONDOWN:
            coordinates.append((x, y))
            cv.circle(image, (x, y), 5, (0, 0, 255), -1)
            cv.imshow('Image', image)

    cv.imshow('Image', image)
    cv.setMouseCallback('Image', click_event)

    while len(coordinates) < 4:
        cv.waitKey(1)

    return np.array(coordinates, dtype=np.float32)
```

```
def superimpose_images(image_bg, image_fg, points_bg, points_fg, alpha, beta):
    # Compute the homography matrix
    homography_matrix, _ = cv.findHomography(points_fg, points_bg)

    # Warp the flag image to fit the architectural image
    image_fg_warped = cv.warpPerspective(image_fg, homography_matrix, (image_bg.shape[1], image_bg.shape[0]))

    # Blend the images
    blended_image = cv.addWeighted(image_bg, alpha, image_fg_warped, beta, 0)

    return blended_image
```

Here is an example for the usage of the above functions. Then I plotted the results and the original image. Here choosing images with flat surfaces where the flag can fit naturally and match the perspective well for a realistic blend is important.

```
#Example 1
image_bg = cv.imread('Images/Assignment 2 Images/Architectural_set1/001(1).jpg')
image_fg = cv.imread('Images/Assignment 2 Images/Flag_of_the_United_Kingdom_(1-2).svg.png')

# Get user-selected coordinates for both images
points_bg = get_user_coordinates(image_bg.copy())
points_fg = np.array([[0, 0], [image_fg.shape[1], 0], [image_fg.shape[1], image_fg.shape[0]], [0, image_fg.shape[0]]],
                     dtype=np.float32)

# Adjust the values of alpha and beta for blending
alpha = 0.95
beta = 0.4

# Superimpose the images
blended_image = superimpose_images(image_bg, image_fg, points_bg, points_fg, alpha, beta)
```

Question 04

First, images are converted to grayscale and then SIFT features of the two images are calculated and matched. The resultant image is displayed.

Calculate & match SIFT features.

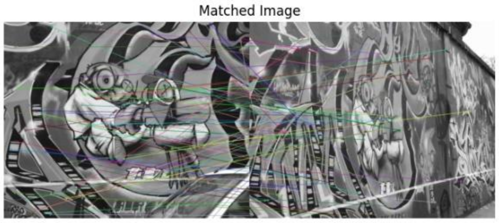
```
# Initialize SIFT detector
sift = cv.SIFT_create()

# Finding the keypoints and descriptors
keys_1, descriptors1 = sift.detectAndCompute(gray1, None)
keys_5, descriptors5 = sift.detectAndCompute(gray5, None)

# BFMatcher with default params
bfMatcher = cv.BFMatcher()

# Match descriptors
matches = bfMatcher.knnMatch(descriptors1, descriptors5, k=2)

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

# Draw the matches
matched_image = cv.drawMatches(gray1, keys_1, gray5, keys_5, good_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

Output

Image 1

Image 5

Matched Image



Ransac homograph function

```
# Implementing RANSAC to compute homography
def ransac_homography(src_points, dst_points, max_iterations, threshold):
    max_inliers = 0
    best_homography = None

    for i in range(max_iterations):
        # Randomly select 4 points
        indices = np.random.randint(0, len(src_points), 4)
        src_sample = src_points[indices]
        dst_sample = dst_points[indices]

        # Compute the homography for the sampled points
        homography = cv.findHomography(src_sample, dst_sample)[0]

        # Warp all source points using the computed homography
        warped_points = cv.perspectiveTransform(src_points, homography)

        # Count inliers (points that are close enough to the warped destination points)
        inliers = np.sum(np.linalg.norm(warped_points - dst_points, axis=2) < threshold)

        if inliers > max_inliers:
            max_inliers = inliers
            best_homography = homography

    return best_homography
```

Final Output

Stitched Image