Assignment 02: Fitting

Jaliya Nimantha
210294N

October 22, 2024

Note: All Python codes relevant to this assignment can be found in GitHub repository.

Question 1

The `blob_log()` function is used to detect blobs in the chosen image by applying the Laplacian of Gaussian (LoG) method. It sets the standard deviation (σ) of the Gaussian kernel between 2 and 40, and ignores blobs with intensities lower than the threshold value of 0.1.The results, shown in **Figure 1**

```
blobs_log = blob_log(image_gray, min_sigma=2, max_sigma=40,
    num_sigma=20, threshold=0.1)
max_initial_radius = max(blobs_log[:, 2])
blobs_log[:, 2] = blobs_log[:, 2] * math.sqrt(2)
max_adjusted_radius = max(blobs_log[:, 2])
fig, ax = plt.subplots()
ax.imshow(image_gray, cmap='gray')
ax.axis('off')
for blob in blobs_log:
    y, x, radius = blob
    circle = plt.Circle((x, y), radius, color='blue',
        linewidth=2, fill=False)
    ax.add_patch(circle)
```

Question 2

RANSAC parameters were set as follows:

The threshold for determining inliers was set to 1.0 for both line and circle estimation, representing the maximum allowable deviation for a point to be considered an inlier. This threshold corresponds to the standard deviation of data points divided by 16 and was chosen through trial and error. For the number of samples (s), two points (s = 2) were selected in each iteration for line estimation, as two points are sufficient to define a line. In contrast, three points (s = 3) were chosen per iteration for circle estimation, since three points are required to uniquely define a circle. While not explicitly stated, RANSAC typically aims for a high probability of selecting a good model; in this case, we assume p = 0.99, meaning the algorithm has a 99% chance of selecting inliers. The outlier ratio (e) was approximately 0.6 for line estimation, indicating that 60% of points were considered outliers based on the assumption that 40% of the points are inliers. For circle estimation, the outlier ratio was approximately 0.41, representing 41% of the remaining points after line inliers were treated as outliers.

- **Number of Iterations:** The number of iterations required to ensure with probability $p = 0.99$ that a good model is found can be estimated using the formula:

$$\text{Number of iterations} = \frac{\log(1-p)}{\log(1-(1-e)^s)}$$

Even though the maximum number of iterations was set to 100, these calculated values of 46 and 36 iterations for the line and circle, respectively, were sufficient to find the best-fitting models within the specified probability.

The results, shown in **Figure 2**, demonstrate that the circle obtained via the RANSAC algorithm (in red) provides a better fit compared to the best selected samples (in green).

2.3) Fitting the circle first would result in a model that accurately represents the circular data points, but it would struggle to capture the remaining points needed for the line model. As a result, determining the line parameters based on those remnants would be challenging. Therefore, in RANSAC-based approaches, it is generally more effective to fit simpler models, such as lines, first, and then apply more complex models to handle the remaining points.

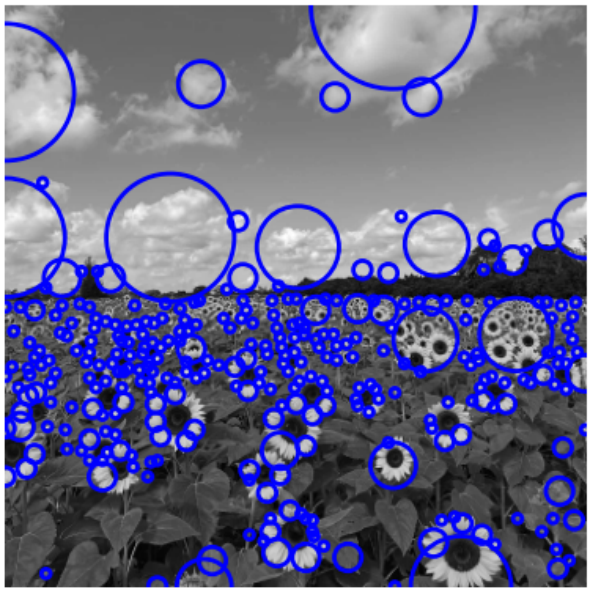

Figure 1

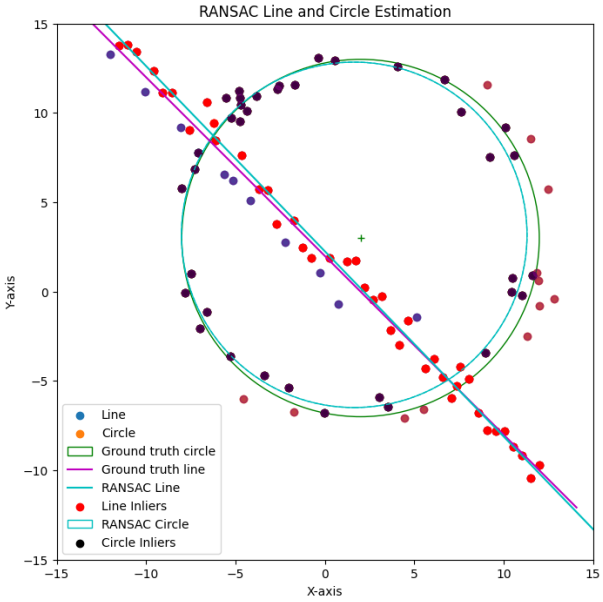

Figure 2

```python
def ransac_circle_estimation(remnants, ax, max_iterations=100,
    c_threshold=1.0, c_num_data=35):
    c_best_error = np.inf
    c_sample = 3
    c_best_model = []
    c_best_inliers = []

    def c_total_LSE(x, indices):
        x0, y0, r = x
        x1, y1 = remnants[indices].T
        error = (np.sqrt((x1 - x0)**2 + (y1 - y0)**2) - r)**2
        return np.sum(error)

    def c_consensus_set(remnants, x, threshold):
        distances = np.abs(np.linalg.norm(remnants - x[:2], axis=1) - x[2])
        return distances < threshold

    i = 0
    while i < max_iterations:
        c_indices = np.random.randint(0, len(remnants), c_sample)
        x0 = np.array([0, 0, 0])
        res = minimize(c_total_LSE, x0, args=c_indices, tol=1e-6)
        c_inliers = c_consensus_set(remnants, res.x, c_threshold)

        if np.sum(c_inliers) > c_num_data:
            x0 = res.x
            res = minimize(c_total_LSE, x0=x0, args=(c_inliers), tol=1e-6)

```

```
            if res.fun < c_best_error:
                c_best_error = res.fun
                c_best_model = res.x
                c_best_inliers = c_inliers
        i += 1

    return c_best_model, c_best_inliers
```

Question 3

In this question, the goal is to overlay a flag onto an architectural image. The process begins by selecting four points on the architectural image, which are used to calculate the homography that aligns the flag onto the structure. Once the homography is computed, the flag is warped accordingly and blended seamlessly with the architectural image.The results, shown in **Figure 3**,

```
while len(selected_points) < 4:
    cv.waitKey(1)  # Keep waiting for user input
cv.destroyAllWindows()

# Set source and destination points for homography transformation
dest_points = np.array(selected_points).astype(np.float32)
height, width = overlay_image.shape[:2]
src_points = np.float32([[0, 0], [width, 0], [0, height], [width, height]])

# Compute the homographic transformation matrix
transformation_matrix = cv.getPerspectiveTransform(src_points, dest_points)

transformed_image = cv.warpPerspective(overlay_image,
    transformation_matrix,
        (original_image.shape[1], original_image.shape[0]))

blend_opacity = 0.5  # Adjust this value for different blending effects
final_image = cv.addWeighted(original_image, 1, transformed_image,
    blend_opacity, 0)
```

Figure 3

Question 4

(a) We use OpenCV's SIFT to identify and match features between Graffiti Image 1 and Image 5. After detecting keypoints and descriptors for both images, we match them, filtering by distance for optimal results. The SIFT feature matching results are shown in Figure 4.

(b) To compute homography using RANSAC, we randomly select four point pairs and calculate the homography. We apply this homography to warp one image, compare it with the other, and filter out inliers. Using the inliers, we recompute the homography. Matching between Image 1 and Image 2 is accurate, while Image 1 and Image 5 show more deviation.

(c) Image 1 was stitched into Image 5 by warping it using the homography and blending. The brightness in the stitched area increased unexpectedly, so it was adjusted. The final result, showing part of a vehicle transferred from Image 1, is displayed in Figure 5.

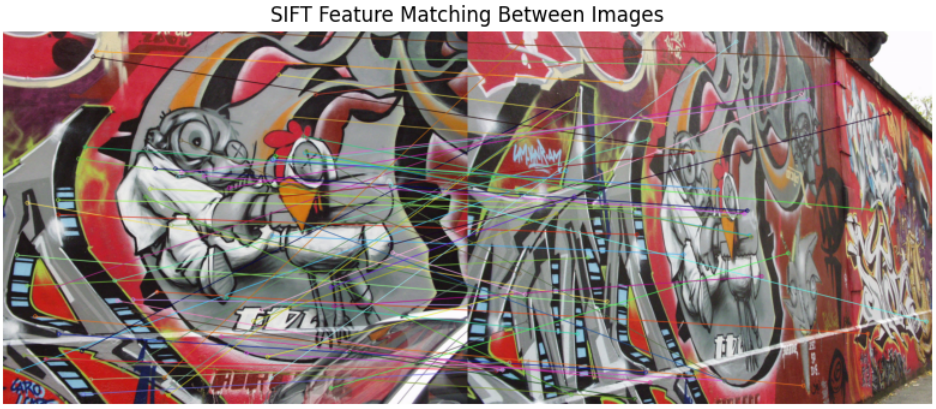

Figure 4

Figure 5

```python
def RANSAC_homography(points1, points2):
    inlier_count, selected_inliers = 0, None
        points = np.hstack((points1, points2))
    num_iterations = int(np.log(1 - 0.95)/np.log(1 - (1 - 0.5)**4))
    for _ in range(num_iterations):
        np.random.shuffle(points)
        pts1, pts1_rem, pts2, pts2_rem = points[:4, :2],
            points[4:, :2], points[:4, 2:], points[4:, 2:]
        H = homography(pts1, pts2)
        inliers = [(pts1_rem[i], pts2_rem[i]) for i in range
            (len(pts1_rem)) if dist(pts1_rem[i], pts2_rem[i], H) < 100]
        if len(inliers) > inlier_count:
            inlier_count = len(inliers)
            selected_inliers = np.array(inliers)
    H = homography(selected_inliers[:, 0], selected_inliers[:, 1])
    return H
```