Department of Electronic and Telecommunication Engineering
University of Moratuwa

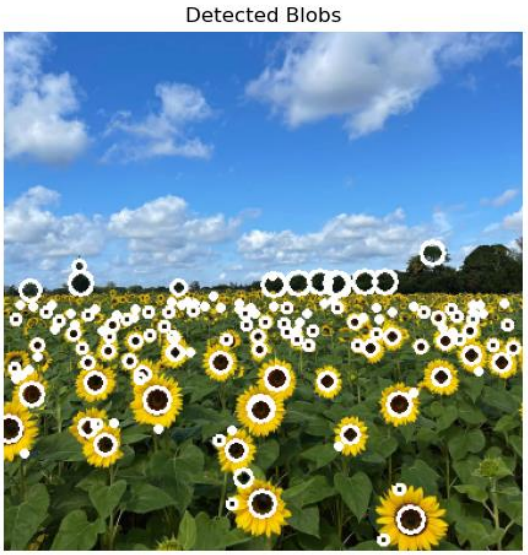
EN3160
Image Processing and Machine Vision

Assignment 2: Fitting and Alignment

210642G – Thennakoon T. M. K. R

GitHub Link for the codes: GitHub

1.



The image was first filtered using a Gaussian kernel, with the standard deviation (σ) values ranging from **1 to 6**. This range was divided into 50 evenly spaced values. A scale space was then created using the selected range of σ values. Non-maximal suppression was initially applied in the scale domain, followed by non-maximal suppression in the spatial domain to identify local maxima. Finally, the maxima were located, and circles were drawn around them using the relation $\sigma = \frac{r}{\sqrt{\{2\}}}$.

Largest blob's cordinates and sigma: (3, 275, 6.0)
Radius of the largest blob: 8.4852

Code Flow:

```
img = cv2.imread("Images/the_berry_farms_sunflower_field.jpeg",
                 cv2.IMREAD_REDUCED_COLOR_4)
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
sigma_values = np.linspace(1,18,50) ; scale_space = detect_blobs(gray_img, sigma_values);
blobs = find_scale_space_extrema(scale_space, sigma_values, threshold=200);
output_img = draw_blobs(img, blobs)
```

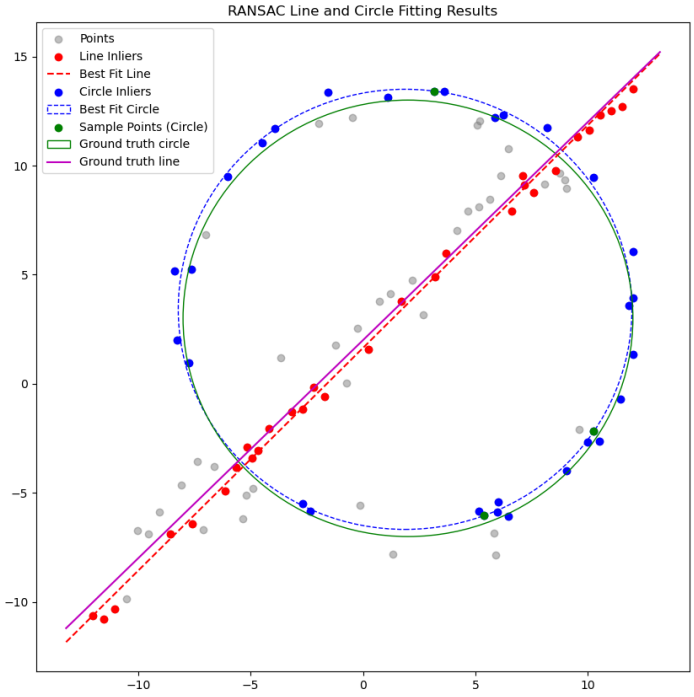
2. a. Below is the highlighted code for line fitting using RANSAC

```
def ransac_line_fitting(points, num_iterations, distance_threshold, min_inliers):
    best_a, best_b, best_d = None, None, None
    best_inliers_count = 0
    n_points = points.shape[0]
    for _ in range(num_iterations):
        random_indices = np.random.choice(n_points, 2, replace=False)
        p1, p2 = points[random_indices]; a, b, d = fit_line_from_points(p1, p2)
        distances = compute_distances(points, a, b, d) ;
        inliers = points[distances < distance_threshold]
        inliers_count = inliers.shape[0]
        if inliers_count > best_inliers_count and inliers_count >= min_inliers:
            best_a, best_b, best_d = a, b, d ; best_inliers_count = inliers_count ;
    return best_a, best_b, best_d, best_inliers_count
```

b. Below is the highlighted code for circle estimation from RANSAC

```
def ransac_circle_fitting(points, num_iterations, distance_threshold, min_inliers):
    best_xc, best_yc, best_r = None, None, None ; best_inliers_count = 0 ;  n_points = points.shape[0] ;
    for _ in range(num_iterations):
        random_indices = np.random.choice(n_points, 3, replace=False)
        p1, p2, p3 = points[random_indices]; xc, yc, r = circle_from_points(p1, p2, p3) ;
        radial_error = radial_distances(points, xc, yc, r) ; inliers = points[radial_error < distance_threshold] ;
        inliers_count = inliers.shape[0]
        if inliers_count > best_inliers_count and inliers_count >= min_inliers:
            best_xc, best_yc, best_r = xc, yc, r ;best_inliers_count =inliers_count ;best_sample_points =[p1, p2, p3];
    return best_xc, best_yc, best_r, best_inliers_count, np.array(best_sample_points)
```

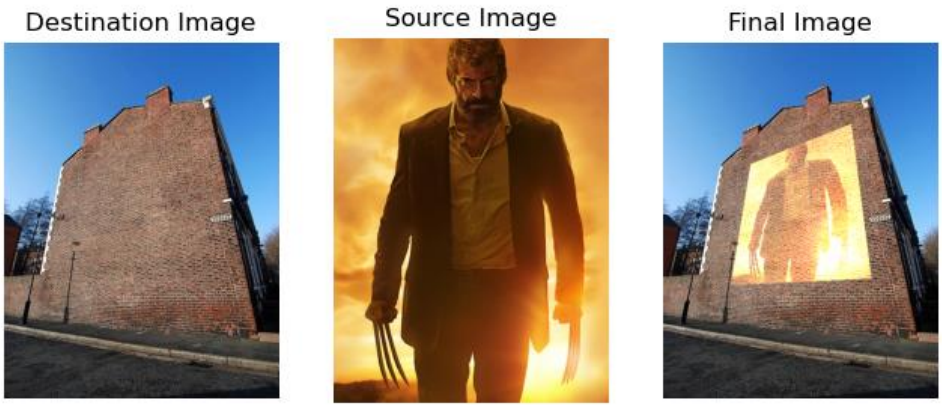
c.



d. fitting the circle first can lead to inaccuracies and misinterpretations due to the influence of outliers (data points of the line) and the model will be getting difficult to estimate a circle.

3. In this task I created my own code for homography calculation. Using this we can calculate homography transform function. For this kind of calculation, we need at least 4 points pairs from source image and destination image. I used corner points from the source image and 4 artribary points from the destination image which get the shape which I need to be transformed.

```
def compute_homography(src_pts, dst_pts):
    assert src_pts.shape == dst_pts.shape, assert src_pts.shape[0] >= 4, num_points = src_pts.shape[0];
    A = []
    for i in range(num_points):
        x_src, y_src = src_pts[i][0], src_pts[i][1] ;  x_dst, y_dst = dst_pts[i][0], dst_pts[i][1]
        # Two rows per correspondence
        A.append([-x_src, -y_src, -1, 0, 0, 0, x_dst * x_src, x_dst * y_src, x_dst])
        A.append([0, 0, 0, -x_src, -y_src, -1, y_dst * x_src, y_dst * y_src, y_dst])
    A = np.array(A);      AtA = np.dot(A.T, A);
    eigenvalues, eigenvectors = np.linalg.eig(AtA);  smallest_eigenvalue_index = np.argmin(eigenvalues)
    h = eigenvectors[:, smallest_eigenvalue_index];  H = h.reshape(3, 3);     return H;
```



4. a. After matching computed features are as follows

b. Img1 and img5 exhibit a low number of detectable features and high error rates due to high differences of perspective. Calculating a homography matrix under these conditions has a high probability of yielding inaccurate results. When considering the entire sequence from img1 to img5, the differences between consecutive are relatively low. Therefore, computing homography matrices between these adjacent images and combining them produces a more accurate overall homography.

Code of computing homography using RANSAC

```
def compute_ransac_homography(src_pts, dst_pts, num_iterations=1000, threshold=5.0):
    max_inliers = 0; best_H = None;
    for k in range(num_iterations):
        indices = random.sample(range(len(src_pts)), 4);  sampled_src_pts = src_pts[indices]
        sampled_dst_pts = dst_pts[indices]
        H = compute_homography(sampled_src_pts, sampled_dst_pts); inliers = 0
        for i in range(len(src_pts)):
            transformed_point = apply_homography(H, src_pts[i])
            error = np.linalg.norm(transformed_point - dst_pts[i])
            if error < threshold:
                inliers += 1
        if inliers > max_inliers:
            max_inliers = inliers;  best_H = H
    inlier_src_pts = []; inlier_dst_pts = []
    for i in range(len(src_pts)):
        transformed_point = apply_homography(best_H, src_pts[i]);
        error = np.linalg.norm(transformed_point - dst_pts[i])
        if error < threshold:
            inlier_src_pts.append(src_pts[i]); inlier_dst_pts.append(dst_pts[i])
    if len(inlier_src_pts) >= 4:
        best_H = compute_homography(np.array(inlier_src_pts), np.array(inlier_dst_pts))
    return best_H, max_inliers
```

c. Final stitched image and comparison as follows