EN3160 Assignment 2- Fitting

Instructed by Dr. Ranga Rodrigo Done by: Rajapaksha I.P.D.D.- 210503H

GitHub : https://github.com/DinethraDivanjana2001/EN3166-Image-processing-and-machine-vision/tree/main/A02%20-%20Fitting

1.0. Question 01-Blob Detection in a Sunflower Field Image

First, I converted the sunflower field image to grayscale to simplify blob detection. The algorithm applies Gaussian blur followed by a Laplacian filter to detect circular shapes at multiple scales.

- **Gaussian Blur:** Smooths the image, reducing noise.
- **Laplacian Filter:** Identifies circular edges by detecting intensity changes.

I varied the scale values (σ) between 1.0 and 4.0 in 8 equal steps**,** as larger scales can capture larger structures, while smaller scales highlight finer details.

The range of **σ values** used was from 1.0 to 4.0

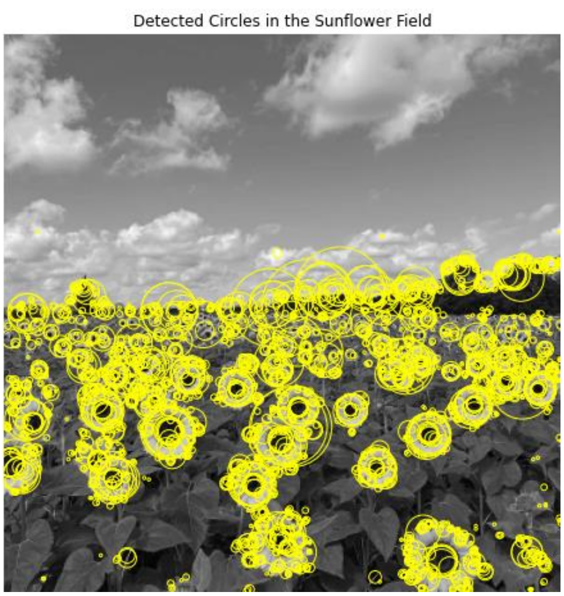

Figure 1-Detected Circles

```
Largest Blob Parameters:
Center: (216, 166)
Radius: 29
Scale: 1.9192898346492004
```

Figure 2 - parameters of the largest circle

```python
def apply_gaussian_laplacian(image, sigma_value):
    """Apply Gaussian Blur followed by Laplacian filtering."""
    blurred = cv.GaussianBlur(image, (0, 0), sigma_value)
    laplacian = cv.Laplacian(blurred, cv.CV_64F)
    return np.abs(laplacian)
```

Figure 2 - Gaussian-Laplacian Transformation

```python
def detect_blobs(image, min_scale, max_scale, num_scales, threshold):
    """Detect blobs across multiple scales and return the detected blobs."""
    scales = np.linspace(min_scale, max_scale, num_scales)
    blobs = []

    for scale in scales:
        adjusted_scale = scale / np.sqrt(2)
        print(f"Processing scale: {adjusted_scale}")

        laplacian_image = apply_gaussian_laplacian(image, adjusted_scale)

        binary_mask = laplacian_image > threshold * laplacian_image.max()

        contours, _ = cv.findContours(binary_mask.astype(np.uint8),
                                      cv.RETR_EXTERNAL,
                                      cv.CHAIN_APPROX_SIMPLE)
        for contour in contours:
            if len(contour) >= 5:
                (x, y), radius = cv.minEnclosingCircle(contour)
                blobs.append(((int(x), int(y)), int(radius), adjusted_scale)

    return sorted(blobs, key=lambda b: -b[1])
```

Figure 4 - detects blobs by applying the Gaussian-Laplacian transformation across scales and extracting circles from contours

2.0. Question - 02

Implemented the RANSAC algorithm to estimate a line and a circle from a noisy set of points.

For line estimation, I constrained the line parameters [a,b][a, b][a,b]. I selected the error as the perpendicular distance from points to the line and defined a threshold for consensus, ensuring only points within this distance were considered in the fitting process. After obtaining the best-fit line, I subtracted the consensus points to analyse the remaining points for circle fitting. Using RANSAC again, I calculated the radial error for circle fitting and set a threshold to determine consensus points for the circle. Finally, I visualized the results by plotting the point set, the estimated line and circle, their inliers, and the sample points used for their estimation, presenting a comprehensive view of the fitting results.

- The calculate_line_equation function computes normalized line coefficients from two points; total_least_squares_error calculates the sum of squared errors for a line model; constraint_function enforces the normalization constraint a2+b2=1a^2 + b^2 = 1a2+b2=1; and identify_inliers determines inliers by comparing the absolute error to a threshold.(figure 5)
- The compute_circle_from_points function calculates the center and radius of a circle given three points; identify_circle_inliers identifies inliers based on the radial error being below one-third of the circle's radius; get_random_sample selects three random points from a list for circle fitting; calculate_distance_from_center computes the distance of points from the circle’s center; algebraic_distance determines the deviation of point distances from the mean radius; and fit_circle_via_least_squares fits a circle using least-squares optimization to estimate the center and radius.(figure 6)

```
Ratio of Inliers = 7.017543859649122 %
Inliers array has insufficient dimensions for plotting.
Center of RANSAC Circle = (2.0435741271449093, 3.870549248953072)
Radius of RANSAC Circle = 9.432686503699676
```

```python
def calculate_line_equation(point1_x, point1_y, point2_x, point2_y):
    """Calculate line equation coefficients from two points."""
    delta_x = point2_x - point1_x
    delta_y = point2_y - point1_y
    magnitude = math.sqrt(delta_x**2 + delta_y**2)
    coeff_a = delta_y / magnitude
    coeff_b = -delta_x / magnitude
    intercept = (coeff_a * point1_x) + (coeff_b * point1_y)
    return coeff_a, coeff_b, intercept

def total_least_squares_error(coefficients, selected_indices):
    """Calculate total least squares error for a line model."""
    coeff_a, coeff_b, intercept = coefficients
    return np.sum(np.square(coeff_a * data_points[selected_indices, 0] +
                            coeff_b * data_points[selected_indices, 1] - intercept))

def constraint_function(coefficients):
    """Define constraint for coefficients."""
    return coefficients[0]**2 + coefficients[1]**2 - 1

constraints = ({'type': 'eq', 'fun': constraint_function})

def identify_inliers(data, coefficients, threshold):
    """Identify inliers based on the consensus function."""
    coeff_a, coeff_b, intercept = coefficients
    error = np.absolute(coeff_a * data[:, 0] + coeff_b * data[:, 1] - intercept)
    return error < threshold
```

Figure 5

```python
def compute_circle_from_points(points):
    """ Calculate the center and radius of the circle from three points """
    point1, point2, point3 = points[0], points[1], points[2]
    temp = point2[0] ** 2 + point2[1] ** 2
    half_bc = (point1[0] ** 2 + point1[1] ** 2 - temp) / 2
    half_cd = (temp - point3[0] ** 2 - point3[1] ** 2) / 2
    determinant = (point1[0] - point2[0]) * (point2[1] - point3[1]) - (point2[0] - point3[0]) * (point1[1] - point2[1])

    # Center of the circle
    center_x = (half_bc * (point2[1] - point3[1]) - half_cd * (point1[1] - point2[1])) / determinant
    center_y = ((point1[0] - point2[0]) * half_cd - (point2[0] - point3[0]) * half_bc) / determinant

    radius = np.sqrt((center_x - point1[0]) ** 2 + (center_y - point1[1]) ** 2)
    return ((center_x, center_y), radius)

def identify_circle_inliers(points_list, center, radius):
    """ Identify inliers to a circle model based on a threshold """
    inliers = []
    threshold = radius / 3  # Set threshold to one-third of the radius

    for point in points_list:
        distance_error = np.sqrt((point[0] - center[0]) ** 2 + (point[1] - center[1]) ** 2) - radius
        if distance_error < threshold:
            inliers.append(point)

    return np.array(inliers)

def get_random_sample(points_list):
    """ Get a list of three random points from a given list """
    sample_indices = random.sample(range(len(points_list)), 3)
    return np.array([points_list[i] for i in sample_indices])

def calculate_distance_from_center(x_values, y_values, center_x, center_y):
    """ Calculate the distance of each 2D point from the circle's center """
    return np.sqrt((x_values - center_x) ** 2 + (y_values - center_y) ** 2)

def algebraic_distance(center, x_values, y_values):
    """ Calculate the algebraic distance between points and the mean circle centered at the given center """
    distances = calculate_distance_from_center(x_values, y_values, *center)
    return distances - distances.mean()

def fit_circle_via_least_squares(x_mean, y_mean, points):
    """ Fit a circle using least squares optimization """
    x_coordinates = points[:, 0]
    y_coordinates = points[:, 1]
    initial_center = x_mean, y_mean
    estimated_center, _ = optimize.leastsq(algebraic_distance, initial_center, (x_coordinates, y_coordinates))

    center_x, center_y = estimated_center
```

Figure 6

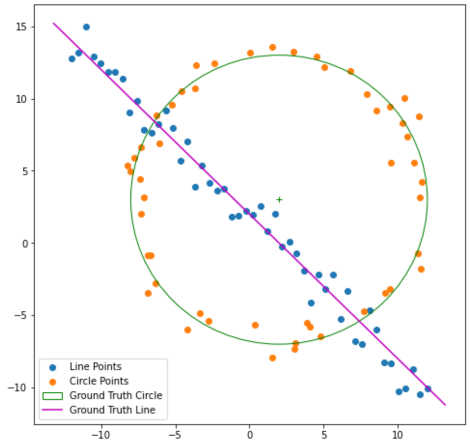

Figure 7 - Original image

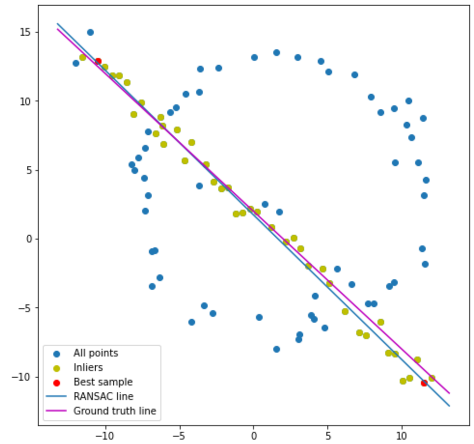

Figure 8- RANSAC Line Fitting

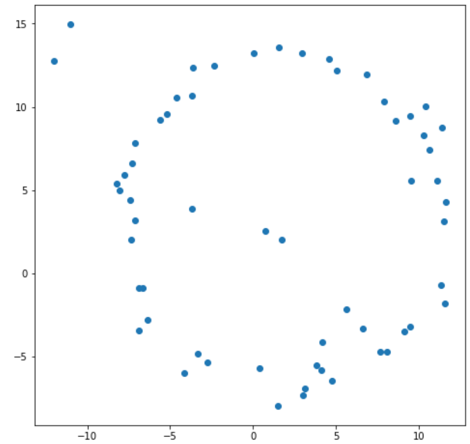

Figure 9 - Detected Outliers and Fitted Circle Visualization

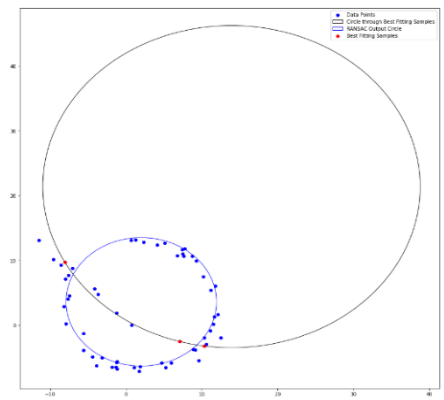

Figure 10- RANSAC Circle Fitting Function

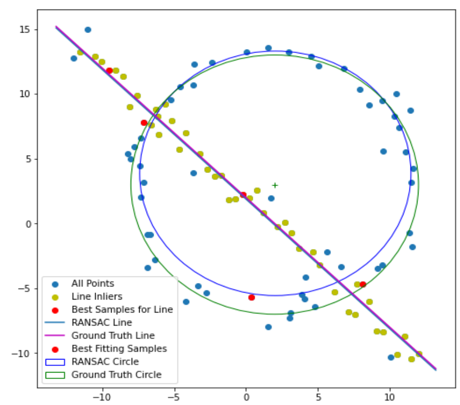

Figure 11 - Revised Plotting Code

- Fitting a circle first on predominantly linear data can lead to poor results because the circle will try to capture all points, even those that follow a line, resulting in a large radius and a poorly positioned center. Many line points might be misclassified as outliers, lowering the inlier ratio and weakening RANSAC performance. Additionally, since a circle requires more parameters than a line, fitting it first may lead to overfitting, reducing RANSAC's effectiveness.

3.0. Question 03

This task overlays a flag onto an architectural image using homography by first loading both images and selecting four points on the surface for placement. These points are used to compute a homography matrix that aligns the flag's corners with them. The flag is then warped to match the architectural perspective, and the images are blended using an `alpha` parameter for transparency, resulting in a seamless integration of the flag into the scene.

```python
# Function to compute the homography and warp the flag image onto the architectural image
def warp_flag_onto_image(architecture_img, flag_img, architecture_points, flag_points):
    # Compute the homography matrix
    homography_matrix, _ = cv2.findHomography(flag_points, architecture_points)

    # Warp the flag image to match the architecture image
    warped_flag = cv2.warpPerspective(flag_img, homography_matrix, (architecture_img.shape[1], architecture_img.shape[0]))

    return warped_flag

# Function to blend the warped flag and architectural image
def blend_images(architecture_img, warped_flag, alpha=0.7):
    # Blend the two images
    blended_image = cv2.addWeighted(architecture_img, 1, warped_flag, alpha, 0)
    return blended_image
```

Figure 11

Figure 12

Figure 13

4.0. Question – 04

The implementation stitches img1.ppm onto img5.ppm by first detecting and matching keypoints using the SIFT (Scale-Invariant Feature Transform) algorithm, applying Lowe's ratio test to filter good matches. Next, it employs a custom RANSAC (Random Sample Consensus) method to compute the best homography matrix based on the matched points, ensuring robust estimation. Finally, the homography is used to warp img1.ppm and seamlessly stitch it onto img5.ppm. This workflow effectively combines feature matching, homography computation, and image warping to create a cohesive stitched image.

```
[[ 4.76241806e-03  2.64303109e-02  2.54729367e-02]
 [-1.46911664e-03  6.40402890e-02  7.62394910e-02]
 [-1.19908222e-03 -7.70266362e-04  1.00000000e+00]]
```

Figure 14- Homography Matrix from RANSAC

```
[[ 6.70443673e+00 -7.22470754e+00 -1.39447473e+00]
 [ 5.29689825e+00 -5.17070305e+00 -2.37703088e+02]
 [ 1.02250666e-02 -1.32862858e-02  1.00000000e+00]]
```

Figure 15 Homography Matrix from RANSAC:

Figure 16- Output

```python
def ransac(matched_points):
    maxInliers = 0
    best_H = None
    for i in range(10):
        random_points = random_sample(matched_points)
        homography = calculateHomography(random_points)
        num_inliers = 0
        for i in range(len(matched_points)):
            d = loss(matched_points[i], homography)
            if d < 3:
                num_inliers += 1
        if num_inliers > maxInliers:
            maxInliers = num_inliers
            best_H = homography

    return best_H
```

```python
def calculateHomography(correspondences):
    temp_list = []
    for points in correspondences:
        p1 = np.matrix([points.item(0), points.item(1), 1]) # (x1,y1)
        p2 = np.matrix([points.item(2), points.item(3), 1]) # (x2,y2)

        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
              p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
        temp_list.append(a1)
        temp_list.append(a2)

    assemble_matrix = np.matrix(temp_list)

    #svd composition
    u, s, v = np.linalg.svd(assemble_matrix)

    #reshape the min singular value into a 3 by 3 matrix
    h = np.reshape(v[8], (3, 3))

    #normalize
    h = (1/h.item(8)) * h
    return h
```