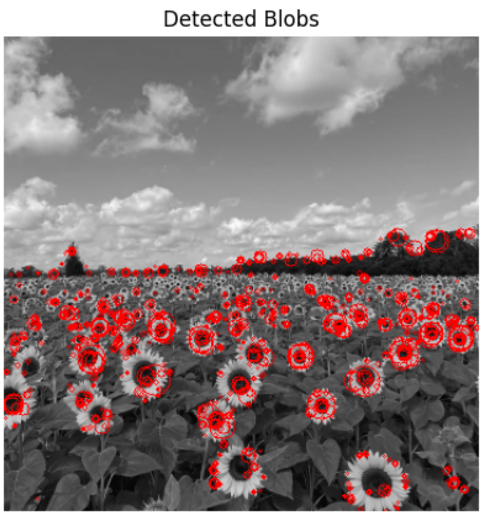
EN3160 Assignment 2 - Fitting and Alignment

Name: VIDMAL H.V.P | **Index No.:** 210668P | **Date:** 23rd October 2024

1 Question 1: Blob Detection

```
sigma_min = 1.0
sigma_max = 2.0
sigma_steps = 5
blob_threshold = 0.36
detected_circles = []
for current_sigma in np.linspace(sigma_min, sigma_max, sigma_steps):
    gaussian_blur = cv.GaussianBlur(gray_img, (0, 0), current_sigma)
    log_result = cv.Laplacian(gaussian_blur, cv.CV_64F)
    abs_log = np.abs(log_result)
    blob_binary = abs_log > blob_threshold * abs_log.max()
    blob_contours, _ = cv.findContours(blob_binary.astype(np.uint8), cv.RETR_EXTERNAL
    , cv.CHAIN_APPROX_SIMPLE)
    for blob in blob_contours:
        if len(blob) >= 5:
            (cx, cy), r = cv.minEnclosingCircle(blob)
            circle_center = (int(cx), int(cy))
            circle_radius = int(r)
            detected_circles.append((circle_center, circle_radius, current_sigma))
```

Parameters of the largest circle:Center: (273, 261), Radius: 12, Sigma value: 2.0



Discussion To address the issue of multiple blob detections in sunflowers and distant trees, we can refine our approach in several ways. Narrowing the range of sigma values could help focus on the expected size of sunflower heads, reducing false detections. Implementing the Difference of Gaussians (DoG) algorithm, known for its lower sensitivity to noise, might improve accuracy in textured areas. Additionally, a post-processing step to filter out unlikely sunflower blobs based on size, color, and shape could significantly enhance the precision of our sunflower detection.

2 Question 2: Noisy Point Set Generation

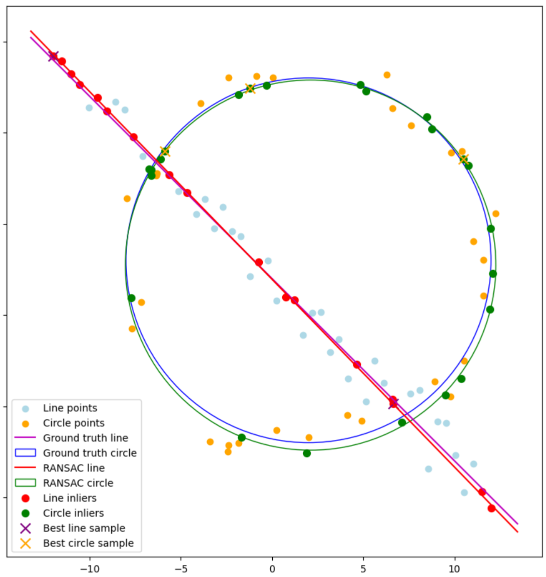
```
# RANSAC for line fitting
def ransac_line(X, iterations=10000, threshold=0.15, min_inliers=15):
    best_model, best_inliers = None, []
    for _ in range(iterations):
        sample = X[np.random.choice(len(X), 2, replace=False)]
        a, b, d = line_eq_from_points(sample[0], sample[1])
        distances = np.abs(a*X[:,0] + b*X[:,1] - d)
        inliers = np.where(distances < threshold)[0]
        if len(inliers) >= min_inliers and len(inliers) > len(best_inliers):
            best_model, best_inliers = (a, b, d), inliers
    return best_model, best_inliers
# Circle equation from three points
def circle_eq_from_points(p1, p2, p3):
```

```
    temp = p2[0]**2 + p2[1]**2
    bc = (p1[0]**2 + p1[1]**2 - temp) / 2
    cd = (temp - p3[0]**2 - p3[1]**2) / 2
    det = (p1[0] - p2[0]) * (p2[1] - p3[1]) - (p2[0] - p3[0]) * (p1[1] - p2[1])
    if abs(det) < 1.0e-6:
        return None
    cx = (bc*(p2[1] - p3[1]) - cd*(p1[1] - p2[1])) / det
    cy = ((p1[0] - p2[0]) * cd - (p2[0] - p3[0]) * bc) / det
    radius = np.sqrt((cx - p1[0])**2 + (cy - p1[1])**2)
    return cx, cy, radius

# RANSAC for circle fitting
def ransac_circle(X, iterations=10000, threshold=0.2, min_inliers=15):
    best_model, best_inliers = None, []

    for _ in range(iterations):
        sample = X[np.random.choice(len(X), 3, replace=False)]
        model = circle_eq_from_points(sample[0], sample[1], sample[2])
        if model is None:
            continue
        cx, cy, r = model
        distances = np.abs(np.sqrt((X[:,0] - cx)**2 + (X[:,1] - cy)**2) - r)
        inliers = np.where(distances < threshold)[0]

        if len(inliers) >= min_inliers and len(inliers) > len(best_inliers):
            best_model, best_inliers = model, inliers
    return best_model, best_inliers
# Optimize circle fit
def optimize_circle(initial_circle, points):
    def error(params):
        cx, cy, r = params
        return np.sum((np.sqrt((points[:,0] - cx)**2 + (points[:,1] - cy)**2) - r)**2)
    result = minimize(error,initial_circle,method='nelder-mead')
    return result.x
```

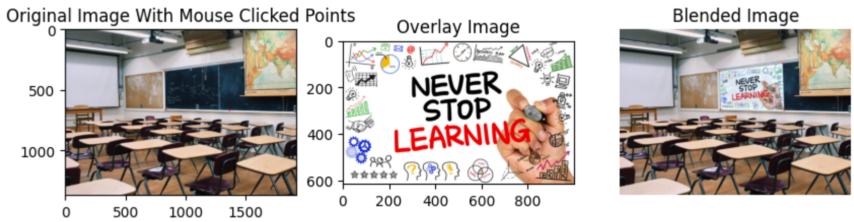


Discussion This code implements RANSAC for fitting both a line and a circle to noisy data containing outliers. It generates synthetic data with 100 points split between a line and a circle, then applies RANSAC to fit these shapes. The algorithm first fits a line using ransac_line(), which iteratively selects two random points to define a line and counts inliers within a threshold distance. After identifying line inliers, it removes them and applies ransac_circle() to the remaining points, using sets of three points to define potential circles. The best fits are determined by maximizing inlier count. The code then optimizes the circle fit using optimize_circle(). Finally, it visualizes the results, plotting original points (blue for line, orange for circle), estimated shapes (red line, green circle), inliers (red for line, green for circle), and best samples (purple 'x' for line, orange 'x' for circle).

If we fit the circle first, the algorithm might correctly detect the circular part of the data, but it would likely include some points from the line as inliers, resulting in a slightly larger and less accurate circle. Consequently, the remaining points for line fitting would be fewer and more scattered, potentially leading to a less accurate line fit. This approach would likely yield less accurate results overall compared to fitting the line first, especially given that there are more points conforming to the line in this dataset.

3 Question 3

```python
# Warp the overlay image based on selected points
def warp_overlay_image(overlay_image, target_points):
    overlay_corners=np.array([[0, 0],[overlay_image.shape[1],0], [overlay_image.shape[1], overlay_image.shape[0]], [0, overlay_image.shape[0]]], dtype=np.float32)
    homography_matrix, _ =cv.findHomography(overlay_corners, target_points)
    warped_overlay = cv.warpPerspective(overlay_image, homography_matrix, (reference_image.shape[1], reference_image.shape[0]))
    return warped_overlay
# Blend the reference image with the warped overlay image
def blend_images(base_image, overlay_image, alpha=0.8):
    return cv.addWeighted(base_image, 1, overlay_image, alpha, 0)
```

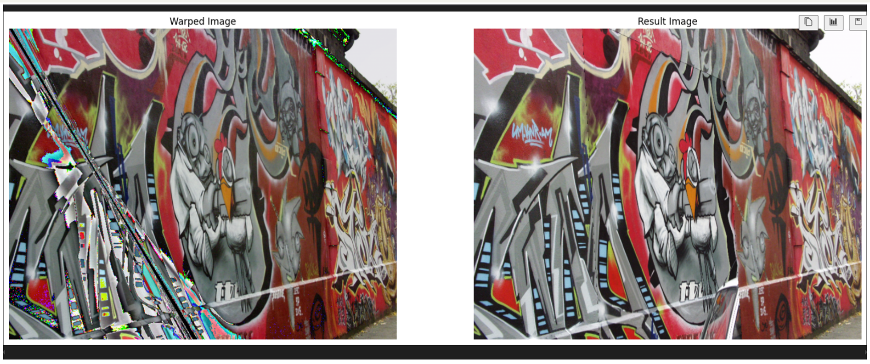


Discussion When the mouse points are selected in a non-sequential manner (i.e., not in a consistent clockwise or counter-clockwise order), the flag may incorrectly extend beyond the intended rectangular region defined by the clicked points. Interactive point selection on the building image offers an intuitive way to outline a planar surface, ensuring accurate placement of the overlay. The calculated homography matrix compensates for perspective distortions, allowing the UK flag to align seamlessly with the building's surface.

4 Question 4

```python
def compute_homography(point_pairs):
    equations = []
    for pair in point_pairs:
        p1 = np.matrix([pair[0], pair[1], 1])  # (x1, y1)
        p2 = np.matrix([pair[2], pair[3], 1])  # (x2, y2)
        # Create equations for homography computation
        eq1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2),
               0, 0, 0, p2.item(0) * p1.item(0), p2.item(0) * p1.item(1),p2.item(0)]
        eq2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2), p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1)]
        equations.extend([eq1, eq2])
```

```python
    # Perform SVD to solve the equations
    equations_matrix = np.matrix(equations)
    _, _, v = np.linalg.svd(equations_matrix)
    # Reshape the solution to a 3x3 matrix and normalize
    homography = np.reshape(v[8], (3, 3))
    homography = (1 / homography.item(8)) * homography
    return homography
def compute_loss(match_pair, homography_matrix):
    point1 = np.transpose(np.matrix([match_pair[0], match_pair[1], 1]))
    point2 = np.transpose(np.matrix([match_pair[2], match_pair[3], 1]))
    # Transform point1 using the homography
    transformed_point = np.dot(homography_matrix, point1)
    transformed_point /= transformed_point.item(2)
    # Calculate the error
    error = np.linalg.norm(point2 - transformed_point)
    return error
def select_random_samples(points_list, sample_size=3):
    random.seed(0)
    selected_indices = random.sample(range(len(points_list)), sample_size)
    return np.array([points_list[i] for i in selected_indices])
def ransac_algorithm(matched_points):
    max_inliers = 0
    best_homography = None
    for _ in range(10):
        sampled_points = select_random_samples(matched_points)
        # Compute homography from sampled points
        homography = compute_homography(sampled_points)
        inlier_count = 0
        # Count inliers
        for match in matched_points:
            if compute_loss(match, homography) < 3:
                inlier_count += 1
        # Update the best homography if more inliers are found
        if inlier_count > max_inliers:
            max_inliers = inlier_count
            best_homography = homography
    return best_homography
print(homography_1_5)
```



$$\text{homography\_1\_5} = \begin{bmatrix} 0.78176252 & -0.75556942 & 0.09696644 \\ 1.30358382 & -1.24252927 & -0.43687914 \\ 0.00253628 & -0.00546429 & 1.0 \end{bmatrix}$$

Discussion The discrepancy in the computed homography matrix is likely due to variations in keypoint selection, keypoint quality, and the behavior of the RANSAC algorithm, which can cause deviations from the ideal matrix. As a result, the stitched image appears distorted because the homography matrix fails to accurately capture the transformation required for proper image alignment.

You can find my notebook from github.