

EN3160 Assignment 2 - Fitting and Alignment

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

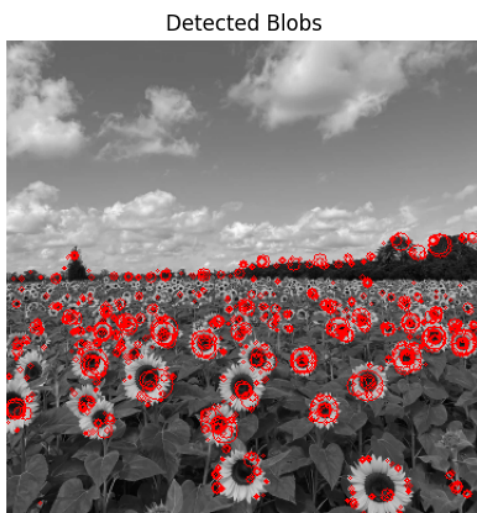
1 Question 1: Blob Detection

```

1 sigma_min = 1.0
2 sigma_max = 2.0
3 sigma_steps = 5
4 blob_threshold = 0.36
5 detected_circles = []
6 for current_sigma in np.linspace(sigma_min, sigma_max, sigma_steps):
7     gaussian_blur = cv.GaussianBlur(gray_img, (0, 0), current_sigma)
8     log_result = cv.Laplacian(gaussian_blur, cv.CV_64F)
9     abs_log = np.abs(log_result)
10    blob_binary = abs_log > blob_threshold * abs_log.max()
11    blob_contours, _ = cv.findContours(blob_binary.astype(np.uint8), cv.RETR_EXTERNAL
    , cv.CHAIN_APPROX_SIMPLE)
12    for blob in blob_contours:
13        if len(blob) >= 5:
14            (cx, cy), r = cv.minEnclosingCircle(blob)
15            circle_center = (int(cx), int(cy))
16            circle_radius = int(r)
17            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

```

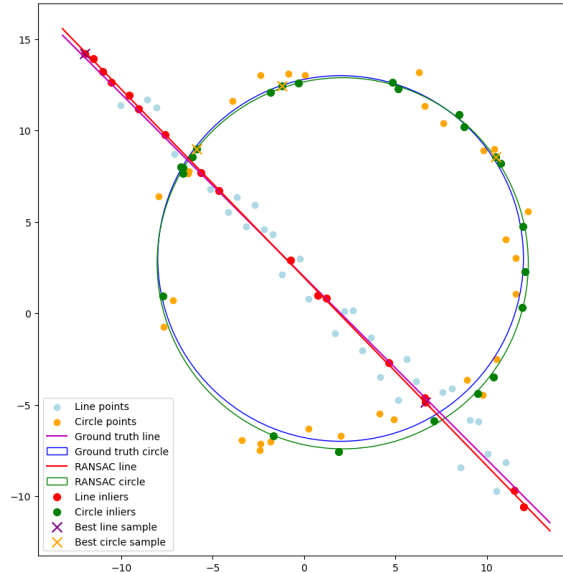
1 # RANSAC for line fitting
2 def ransac_line(X, iterations=10000, threshold=0.15, min_inliers=15):
3     best_model, best_inliers = None, []
4     for _ in range(iterations):
5         sample = X[np.random.choice(len(X), 2, replace=False)]
6         a, b, d = line_eq_from_points(sample[0], sample[1])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

```

```

14     temp = p2[0]**2 + p2[1]**2
15     bc = (p1[0]**2 + p1[1]**2 - temp) / 2
16     cd = (temp - p3[0]**2 - p3[1]**2) / 2
17     det = (p1[0] - p2[0]) * (p2[1] - p3[1]) - (p2[0] - p3[0]) * (p1[1] - p2[1])
18     if abs(det) < 1.0e-6:
19         return None
20     cx = (bc*(p2[1] - p3[1]) - cd*(p1[1] - p2[1])) / det
21     cy = ((p1[0] - p2[0]) * cd - (p2[0] - p3[0]) * bc) / det
22     radius = np.sqrt((cx - p1[0])**2 + (cy - p1[1])**2)
23     return cx, cy, radius
24
25 # RANSAC for circle fitting
26 def ransac_circle(X, iterations=10000, threshold=0.2, min_inliers=15):
27     best_model, best_inliers = None, []
28
29     for _ in range(iterations):
30         sample = X[np.random.choice(len(X), 3, replace=False)]
31         model = circle_eq_from_points(sample[0], sample[1], sample[2])
32         if model is None:
33             continue
34         cx, cy, r = model
35         distances = np.abs(np.sqrt((X[:,0] - cx)**2 + (X[:,1] - cy)**2) - r)
36         inliers = np.where(distances < threshold)[0]
37
38         if len(inliers) >= min_inliers and len(inliers) > len(best_inliers):
39             best_model, best_inliers = model, inliers
40     return best_model, best_inliers
41
42 # Optimize circle fit
43 def optimize_circle(initial_circle, points):
44     def error(params):
45         cx, cy, r = params
46         return np.sum((np.sqrt((points[:,0] - cx)**2 + (points[:,1] - cy)**2) - r)
47         **2)
48     result = minimize(error, initial_circle, method='nelder-mead')
49     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).

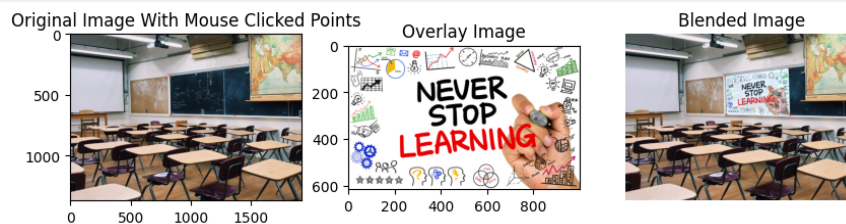
If we fit the circle first, the algorithm might correctly detect the circular part of the data, but it would likely include some points from the line as inliers, resulting in a slightly larger and less accurate circle. Consequently, the remaining points for line fitting would be fewer and more scattered, 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

```

1 # Warp the overlay image based on selected points
2 def warp_overlay_image(overlay_image, target_points):
3     overlay_corners=np.array([[0, 0],[overlay_image.shape[1],0], [overlay_image.shape
4     [1], overlay_image.shape[0]], [0, overlay_image.shape[0]]], dtype=np.float32)
5     homography_matrix, _ =cv.findHomography(overlay_corners, target_points)
6     warped_overlay = cv.warpPerspective(overlay_image, homography_matrix, (
7     reference_image.shape[1], reference_image.shape[0]))
8     return warped_overlay
9
10 # Blend the reference image with the warped overlay image
11 def blend_images(base_image, overlay_image, alpha=0.8):
12     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

```

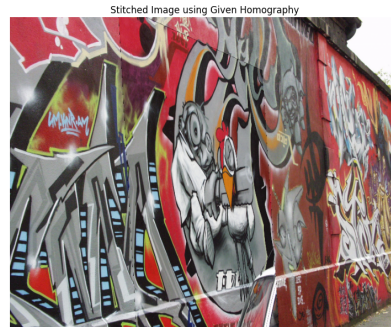
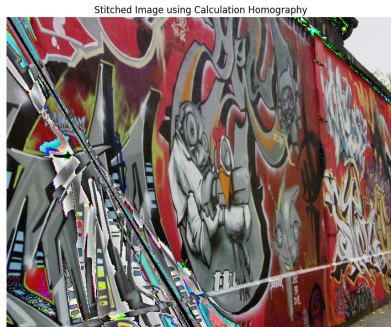
1 def compute_homography(point_pairs):
2     equations = []
3     for pair in point_pairs:
4         p1 = np.matrix([pair[0], pair[1], 1]) # (x1, y1)
5         p2 = np.matrix([pair[2], pair[3], 1]) # (x2, y2)
6         # Create equations for homography computation
7         eq1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2),
8               0, 0, 0, p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0)]
9         eq2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(
10            2), p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1)]
11         equations.extend([eq1, eq2])

```

```

11 # Perform SVD to solve the equations
12 equations_matrix = np.matrix(equations)
13 _, _, v = np.linalg.svd(equations_matrix)
14 # Reshape the solution to a 3x3 matrix and normalize
15 homography = np.reshape(v[8], (3, 3))
16 homography = (1 / homography.item(8)) * homography
17 return homography
18 def compute_loss(match_pair, homography_matrix):
19     point1 = np.transpose(np.matrix([match_pair[0], match_pair[1], 1]))
20     point2 = np.transpose(np.matrix([match_pair[2], match_pair[3], 1]))
21     # Transform point1 using the homography
22     transformed_point = np.dot(homography_matrix, point1)
23     transformed_point /= transformed_point.item(2)
24     # Calculate the error
25     error = np.linalg.norm(point2 - transformed_point)
26     return error
27 def select_random_samples(points_list, sample_size=3):
28     random.seed(0)
29     selected_indices = random.sample(range(len(points_list)), sample_size)
30     return np.array([points_list[i] for i in selected_indices])
31 def ransac_algorithm(matched_points):
32     max_inliers = 0
33     best_homography = None
34     for _ in range(10):
35         sampled_points = select_random_samples(matched_points)
36         # Compute homography from sampled points
37         homography = compute_homography(sampled_points)
38         inlier_count = 0
39         # Count inliers
40         for match in matched_points:
41             if compute_loss(match, homography) < 3:
42                 inlier_count += 1
43         # Update the best homography if more inliers are found
44         if inlier_count > max_inliers:
45             max_inliers = inlier_count
46             best_homography = homography
47     return best_homography
48 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](#).