EN3160 – Image Processing and Machine Vision

Assignment 2: Fitting and Alignment

Perera M.S.S - 220470K

GitHub Repository: https://github.com/ShehanPer/EN3160-Assignment-02-220470K.git

1 Question 1: Blob Detection

In this question, I used Laplacian of Gaussian (LoG) filters and scale-space extrema detection to identify blob regions in a sunflower field image.

Listing 1: Blob detection using LoG

```
sigma_values = np.linspace(1.0, 12.0, 12)
scale_space = []
for sigma in sigma_values:
    hw = int(3 * sigma)
    X, Y = np.meshgrid(np.arange(-hw, hw + 1), np.arange(-hw, hw + 1))
    LoG = ((X**2 + Y**2) / (2 * sigma**2) - 1) * np.exp(-(X**2 + Y**2) /
        (2 * sigma**2))
    LoG = (LoG / (np.pi * sigma**4)) * sigma**2 # normalized LoG filter
    response = cv.filter2D(gray, cv.CV_32F, LoG)
    response = np.square(response)
    scale_space.append(response) # store for different sigma
scale_space = np.stack(scale_space, axis=-1)
blobs = []
threshold = 0.1
for s in range(1, len(sigma_values)-1):
    response = scale_space[:, :, s]
    for i in range(1, response.shape[0]-1):
        for j in range(1, response.shape[1]-1):
            local_cube = scale_space[i-1:i+2, j-1:j+2, s-1:s+2]
            if response[i, j] == np.max(local_cube) and response[i, j] >
                threshold:#local maxima detection
                radius = np.sqrt(2) * sigma_values[s]
                blobs.append((j, i, radius))
```

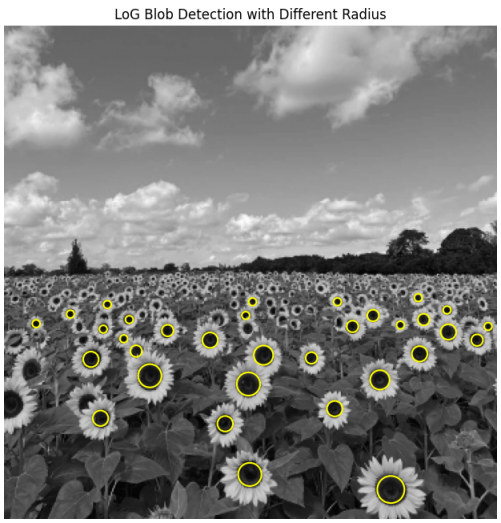
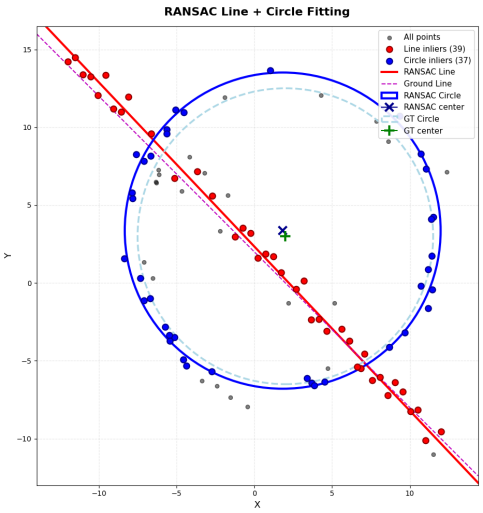


Figure 1 shows the output from the blob detection. The centers of the flowers show a near-perfect match with the blob filter, producing high responses in those areas with threshold > 0.1. It can also be seen how different sizes of Gaussians respond to sunflowers of different radius. By scanning through σ values from 1.0 to 12.0, sunflowers of various sizes were detected. The relationship between radius and σ is given by $\sigma = \text{radius}/\sqrt{2}$. The 3 largest detected blobs had radius of approximately 9.90, 8.49, and 7.07 pixels, corresponding to $\sigma \approx 7.00$, 6.00, and 5.00, respectively.

Figure 1: Detected blobs in the sunflower image with threshold = 0.1.

2 Question 2: RANSAC Line and Circle Fitting

RANSAC was used to robustly fit a line and a circle to a noisy dataset. The estimated line parameters were $a = -0.7258$, $b = -0.6879$, and $d = -1.6124$ with 39/100 inliers, closely matching the ground truth line ($m = -1$, $b = 2$). For the circle, the estimated center was $(1.8281, 3.3601)$ with radius $r = 10.1561$ and 37/61 inliers, which is close to the ground truth circle $(2, 3)$ with $r = 9.5037$. These results demonstrate that RANSAC effectively rejects outliers and accurately recovers the underlying geometric models.



(a) Final RANSAC results.

```
num_consensus =20
threshold =0.8
iteration = 100
best_param = None
best_inliers = []

for i in range(iteration):
    idx = np.random.choice(len(X),2, replace = False)
    p1, p2 = X[idx]

    # vector of p2-p1
    vec = p2-p1
    normal_vec = np.array([-vec[1],vec[0]])
    normal_vec = normal_vec/np.linalg.norm(normal_vec) # ||[a, b]||=1

    d = np.dot(normal_vec,p1)

    distances = np.abs(X @  normal_vec-d)

    inliers = np.where(distances<threshold)[0]

    if len(inliers)>len(best_inliers) and len(inliers) >= num_consensus:
        best_inliers = inliers
        best_param = np.array([normal_vec[0], normal_vec[1], d])
```

(b) Line fitting approach.

```
# RANSAC for circle fitting
X_remnant = np.delete(X, best_inliers, axis=0)

iteration_circle = 100
best_param_circle = None
best_circle_inliers = []
num_consensus_circle = 30
threshold_circle =0.6

for i in range(iteration_circle):
    idx_circle = np.random.choice(len(X_remnant),3, replace=False)
    p1, p2, p3 = X_remnant[idx_circle]


    mat = np.vstack([p2 - p1, p3 - p1])
    if np.linalg.matrix_rank(mat) < 2:
        continue

    A = np.array([
        [2*(p2[0]-p1[0]), 2*(p2[1]-p1[1])],
        [2*(p3[0]-p1[0]), 2*(p3[1]-p1[1])]
    ])

    b = np.array([
        p2[0]**2 + p2[1]**2 - p1[0]**2 - p1[1]**2,
        p3[0]**2 + p3[1]**2 - p1[0]**2 - p1[1]**2
    ])
    center = np.linalg.solve(A, b)
    xc, yc = center
    r = np.sqrt((p1[0]-xc)**2 + (p1[1]-yc)**2)


    dists = np.sqrt((X_remnant[:, 0] - xc)**2 + (X_remnant[:, 1] - yc)**2)
    radial_error = np.abs(dists - r)

    inliers = np.where(radial_error < threshold_circle)[0]

    if len(inliers) > len(best_circle_inliers) and len(inliers) >= num_consensus_circle:
        best_circle_inliers = inliers
        best_param_circle = np.array([xc, yc, r])
```

(c) Circle fitting approach.

Figure 2: RANSAC-based model fitting results and corresponding implementation steps.

What Happens If the Circle is Fitted First?

If I try to fit the circle first, most of the line points become outliers, and the model struggles to find a proper circle. It can give wrong parameters or fail to detect the circle at all. Fitting the line first removes those points, so the circle fitting works much better afterward.

3 Question 3: Homography Transformation

For the homography transformation, two images were used, and points were selected via mouse clicks. Figures 3 (a) and (b) show the input and resulting overlay.

Listing 2: Computing and applying homography

```
main_img_points = np.array(img_points, dtype=np.float32)  # convert clicked points to array
flag_points = np.array([[0, 0], [flag_img.shape[1], 0], [flag_img.shape[1], flag_img.shape[0]], [0, flag_img.shape[0]]], dtype=np.float32)
H, _ = cv.findHomography(flag_points, main_img_points)  # compute homography matrix
warped_flag = cv.warpPerspective(flag_img, H, (main_img.shape[1], main_img.shape[0]))  # warp flag image to fit target region
mask = np.zeros_like(main_img, dtype=np.uint8)
cv.fillConvexPoly(mask, main_img_points.astype(int), (255, 255, 255))
flag_region = cv.bitwise_and(warped_flag, mask)
background = cv.bitwise_and(main_img, cv.bitwise_not(mask))
blend_img = cv.add(background, flag_region)  # combine warped flag with background
```

(a) Overlay result using homography.

(b) Second overlay result.

Figure 3: Comparison of overlay results using homography on two different images.

In this task, I selected pairs of images where one was superimposed onto a planar region of another. While choosing the images, I carefully considered their aspect ratios to ensure that the overlaid images did not appear stretched or distorted. Attention was also given to maintaining proper proportions and perspective alignment, resulting in a natural and visually consistent final composition.

4 Question 4: SIFT Feature Matching and Stitching

I computed SIFT features, matched them, estimated the homography within RANSAC, and stitched images. The following code was used to get the SIFT feature matching

Listing 3: SIFT feature matching

```
def get_sift_features(img1, img2, ratio_test=0.8):
    img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create(nOctaveLayers=3, contrastThreshold=0.09,
        edgeThreshold=25, sigma=1) # Create SIFT detector
    kp1, des1 = sift.detectAndCompute(img1_gray, None)
    kp2, des2 = sift.detectAndCompute(img2_gray, None)
    bf = cv2.BFMatcher() # Match descriptors using BFMatcher
    matches_knn = bf.knnMatch(des1, des2, k=2)
    best_matches = [m for m, n in matches_knn if m.distance < ratio_test
        * n.distance]
    return best_matches, kp1, kp2
```

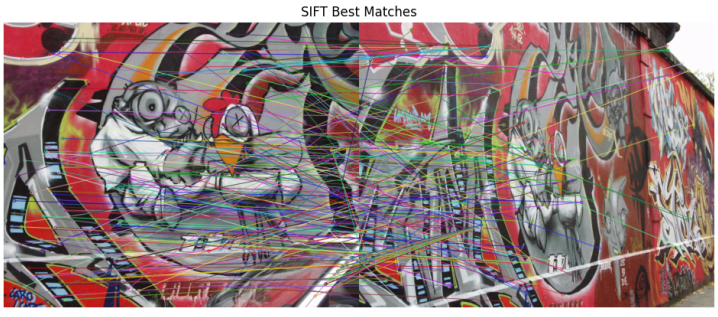
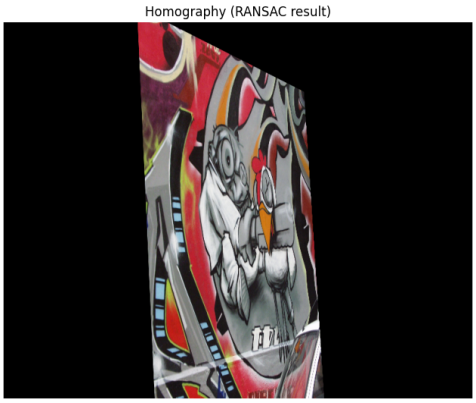


Figure 4: SIFT feature matching

I used the RANSAC method to compute the best homography matrix, which robustly handles outliers in the matched points between images.

Listing 4: RANSAC-based homography

```python
def find_Htransform(best_matches, keypoints1, keypoints5):
    num_points, threshold, iterations = 4, 8, 500
    best_homography, best_inlier_count, best_inliers = None, 0, None
    all_test_points = []
    all_dest_points = []
    for match in best_matches:
        all_test_points.append(np.array(keypoints1[match.queryIdx].pt))
        all_dest_points.append(np.array(keypoints5[match.trainIdx].pt))
    all_test_points = np.array(all_test_points)
    all_dest_points = np.array(all_dest_points)
    for i in range(iterations):
        selected_idx = np.random.choice(len(best_matches), num_points,
           replace=False)
        selected_points = [best_matches[i] for i in selected_idx]
        test_points = []
        dest_points = []
        for point in selected_points:
            test_points.append(np.array(keypoints1[point.queryIdx].pt))
            dest_points.append(np.array(keypoints5[point.trainIdx].pt))
        test_points = np.array(test_points)
        dest_points = np.array(dest_points)
        h_transform = transform.estimate_transform("projective",
           test_points, dest_points)
        calc_points = h_transform(all_test_points)
        errors = np.sqrt(np.sum(np.square(calc_points-all_dest_points),
           axis=1))
        inliers = np.where(errors<threshold)[0]
        if len(inliers)> best_inlier_count:
            best_inlier_count = len(inliers)
            best_inliers = inliers
            best_homography = h_transform
    return best_homography, best_inliers
```



(a) Homography transformation.

(b) Stitched image.

Figure 5: Homography Transformation and stitched image

The homography matrix computed using RANSAC is similar to the provided dataset matrix (`H1to5p`) :

$$H_{\text{computed}} = \begin{bmatrix} 0.6461 & 0.0322 & 227.36 \\ 0.2468 & 1.1410 & -24.32 \\ 0.00057 & -0.000086 & 1 \end{bmatrix}, \quad H_{\text{given}} = \begin{bmatrix} 0.6254 & 0.0578 & 222.01 \\ 0.2224 & 1.1652 & -25.61 \\ 0.00049 & -0.0000365 & 1 \end{bmatrix}.$$