Department of Electronics & Telecommunication Engineering
University of Moratuwa

EN3160

Assignment 2 on Fitting and Alignment

Individual Assignment

Gunawardana W.N.M - 210199D

Date - 2024.10.23

1 Blob Detection

In this part, implemented blob detection using the Laplacian of Gaussian (LoG) method to identify blobs in the image. We utilize the relationship $\sigma = \frac{r}{\sqrt{2}}$ to select appropriate σ values for blob detection based on a given radius r. The following code shows the LoG function used for blob detection:

```python
def laplacian_of_gaussian(sigma):
    kernel_radius = int(3 * sigma)  # Kernel radius based on sigma
    x_coords, y_coords = np.meshgrid(np.arange(-kernel_radius, kernel_radius + 1), np.arange(-kernel_radius, kernel_radius + 1))
    squared_distances = (x_coords**2 + y_coords**2) / (2 * sigma**2)
    gaussian_term = np.exp(-squared_distances)
    laplacian_term = (squared_distances - 1)
    log_kernel = laplacian_term * gaussian_term / (np.pi * sigma**4)
    return log_kernel
```

Listing 1: Laplacian of Gaussian function

After computing the LoG, local maxima were detected using the following method:

```python
def find_local_maxima(log_image, sigma):
    maxima_coords = []
    height, width = log_image.shape
    offset = 1
    for row in range(offset, height - offset):
        for col in range(offset, width - offset):
            local_patch = log_image[row - offset:row + offset + 1, col - offset:col + offset + 1] max_value = np.max(local_patch)  # Find the local maximum

            if max_value >= 0.09:  # Threshold check
                local_max_row, local_max_col = np.unravel_index(np.argmax(local_patch), local_patch.shape)
                maxima_coords.append((row + local_max_row - offset, col + local_max_col - offset))

    return set(maxima_coords)
```

Listing 2: Detecting Local Maxima for Blob Detection

The blobs were detected for values of r ranging from 1 to 10. For each σ calculated from the radius, the corresponding largest circles were successfully detected.

Figure 1: Blob detection results using LoG method.

2 Line and Circle Fitting using RANSAC

We applied the RANSAC algorithm to fit a line and a circle to noisy data points. For the line fitting, we used two points to define the model, while for the circle fitting, three points were selected. The following parameters were used for RANSAC:

- **s = 2 for line, s = 3 for circle**: Minimum points to define the shape.
- **Error threshold (t)**: 1 for line, 1.2 for circle.

- **Consensus size (d)**: 40 points for line and circle.

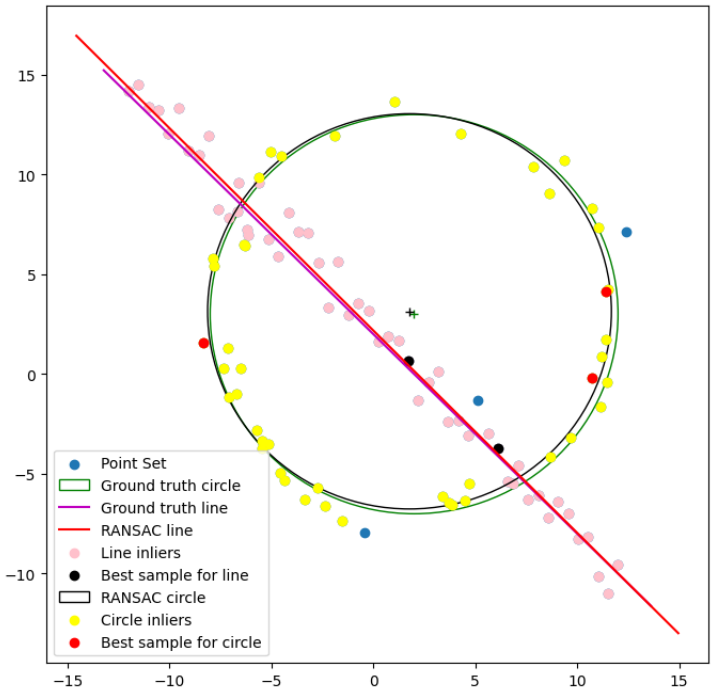

Figure 2: Sample plot of the line.

The RANSAC algorithm was implemented as follows:

```
for i in range(iters):
    indices = np.random.choice(np.arange(0, N), size=min_points, replace=False)
    params = line_eq(X[indices[0]], X[indices[1]])
    inliers = consensus_line(params, thres, X)[0]
    print(f'Iteration {i}: No. of inliers = {len(inliers)}')

    if len(inliers) >= d:   # compute again
        res = least_squares_line_fit(inliers, params, X)
        if res.fun < best_error:
            best_error = res.fun
            best_model_line = params
            best_fitted_line = res.x
            best_line_inliers = inliers
            best_sample_points = indices

line_inliers = consensus_line(best_fitted_line, 1.2, X)[0]
```

Listing 3: RANSAC for Line Fitting

The least-squares fitting was performed using the following function:

```
def least_squares_line_fit(indices, initial, X):  # line fitting with scipy minimize
    res = minimize(fun=tls_error_line, x0=initial, args=(indices, X), constraints=
    constraint_dict, tol=1e-6)
    return res
```

Listing 4: Least-squares line fitting

The fitting for both line and circle was successful, and the inliers were identified effectively.

3 Image Superimposition

point selection of the image is done as follows:

```
# Number of points to select
N = 4
```

```python
selected_points = np.empty((N, 2), dtype=np.float32)
current_point = 0
# Callback function to handle mouse events
def select_points(event, x, y, flags, param):
    global current_point
    if event == cv.EVENT_LBUTTONDOWN and current_point < N:
        cv.circle(param, (x, y), 5, (0, 255, 0), -1)
        selected_points[current_point] = (x, y)
        current_point += 1
# Make a copy of the image to draw on
im1_copy = im1.copy()
# Set up the window and callback
cv.namedWindow('Image 1', cv.WINDOW_AUTOSIZE)
cv.setMouseCallback('Image 1', select_points, im1_copy)
# Event loop to display the image and capture points
while current_point < N:
    cv.imshow('Image 1', im1_copy)
    if cv.waitKey(20) & 0xFF == 27:  # Exit on 'Esc' key
        break
```

Listing 5: Point selection from image

Implemented image superimposition using a projective transformation. The following function was used to compute the homography and superimpose one image on top of another:

```python
def superimpose(image, logo, dst_points, beta=0.3, alpha=1):
    logo_height, logo_width, _ = logo.shape
    dst_height, dst_width, _ = image.shape
    src_points = np.array([[0, logo_height], [logo_width, logo_height], [logo_width, 0], [0, 0]])
    if logo_height < dst_height:
        logo = np.pad(logo, ((0, dst_height - logo_height), (0, 0), (0, 0)), mode='constant')
    if logo_width < dst_width:
        logo = np.pad(logo, ((0, 0), (0, dst_width - logo_width), (0, 0)), mode='constant')
    tform = transform.estimate_transform('projective', src_points, dst_points)
    warped_logo = transform.warp(logo, tform.inverse, output_shape=(dst_height, dst_width))
    warped_logo = (warped_logo * 255).astype(np.uint8)  # Scale to [0, 255] and convert to uint8
    if warped_logo.shape[0] > dst_height:
        warped_logo = warped_logo[:dst_height, :]
    if warped_logo.shape[1] > dst_width:
        warped_logo = warped_logo[:, :dst_width]
    blended_image = cv.addWeighted(image, alpha, warped_logo, beta, 0)
    return blended_image
```

Listing 6: Image Superimposition with Projective Transform

The results of superimposing a logo onto an image are shown below:

Figure 3: Superimposed image using projective transform 1.

Figure 4: Superimposed image using projective transform 2.

Figure 5: Superimposed image using projective transform 3.

4 Image Stitching

We applied image stitching techniques by computing homographies between adjacent images. The homographies were accumulated, and the first image was transformed through the series of homographies to stitch it with the final image.

```
def stitch_images(images):
    global final_transform  # Declare final_transform as global
    final_transformed_image = images[0]  # Initialize with the first image
    plt.figure(figsize=(15, 10))
    for i in range(1, len(images)):
        img1, img2 = images[i-1], images[i]
        good_matches, keypoints1, keypoints2 = sift_features(img1, img2)
        tform, _ = find_best_homography(good_matches, keypoints1, keypoints2)
        final_transformed_image = transform.warp(final_transformed_image, tform.inverse)
        final_transform = final_transform + tform
```

Listing 7: Image Stitching using Homographies

The stitched image result is shown below:

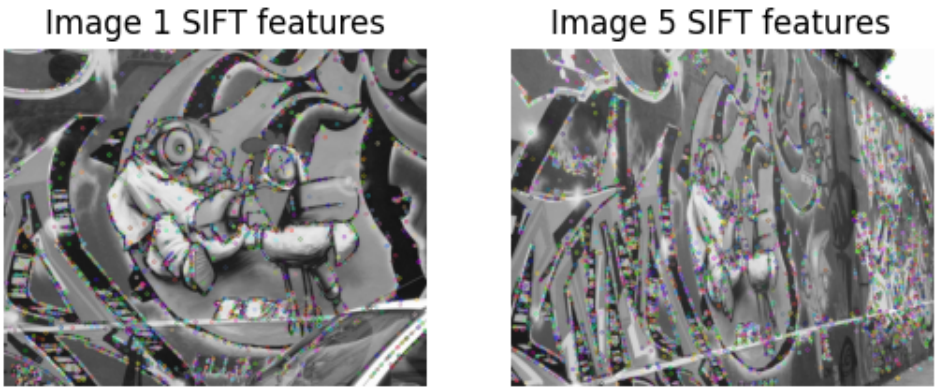

Figure 6: SIFT feature matching.

Figure 7: Image stitching results using computed homographies.

The computed homographies were found to be accurate, and the final stitched image aligns well with the original images.

```
Computed Homography:
[[ 6.11404459e-01  5.03189502e-02  2.21391678e+02]
 [ 2.11980223e-01  1.14096503e+00 -2.14952739e+01]
 [ 4.74861344e-04 -5.18621014e-05  9.90604809e-01]]

Actual Homography:
[[ 6.2544644e-01  5.7759174e-02  2.2201217e+02]
 [ 2.2240536e-01  1.1652147e+00 -2.5605611e+01]
 [ 4.9212545e-04 -3.6542424e-05  1.0000000e+00]]

Sum of squared errors: 233525.3846370272
```

Figure 8: Homography comparison

Link for the GitHub Repository: Link