EN3160 Assignment 2 on Fitting and Alignment

Sahan Abeyrathna
210005H
2024/10/23
Github link: *https://github.com/Sahanmin/EN3160-Assignment-2-on-Fitting-and-Alignment*

Question 1


Figure:Generated results

```python
def compute_LoG(sigma):
    # Determine window size
    n = np.ceil(sigma * 6)
    y, x = np.ogrid[-n//2:n//2+1, -n//2:n//2+1]

    # Apply Gaussian filters in x and y directions
    y_filter = np.exp(-(y ** 2) / (2.0 * sigma ** 2))
    x_filter = np.exp(-(x ** 2) / (2.0 * sigma ** 2))

    # Calculate Laplacian of Gaussian filter
    log_filter = (-(2 * sigma ** 2) + (x ** 2 + y ** 2)) * (x_filter * y_filter) * (1 / (2 * np.pi * sigma ** 4))

    return log_filter
```

```python
def convolve_LoG(img):
    log_images = []
    for i in range(1, 10):
        scale_factor = np.power(k, i)
        sigma_scaled = sigma * scale_factor
        log_filter = compute_LoG(sigma_scaled)

        # Convolve image with the LoG filter
        filtered_img = cv2.filter2D(img, -1, log_filter)
        filtered_img = np.pad(filtered_img, ((1, 1), (1, 1)), 'constant')
        filtered_img = np.square(filtered_img)

        log_images.append(filtered_img)

    log_image_stack = np.array([img for img in log_images])

    return log_image_stack
```

```python
def detect_blobs(log_image_stack):
    coordinates = []
    height, width = img.shape

    for i in range(1, height):
        for j in range(1, width):
            region = log_image_stack[:, i-1:i+2, j-1:j+2]
            max_value = np.amax(region)

            if max_value >= 0.03:
                z, x, y = np.unravel_index(region.argmax(), region.shape)
                coordinates.append((i + x - 1, j + y - 1, k ** z * sigma))

    return coordinates
```

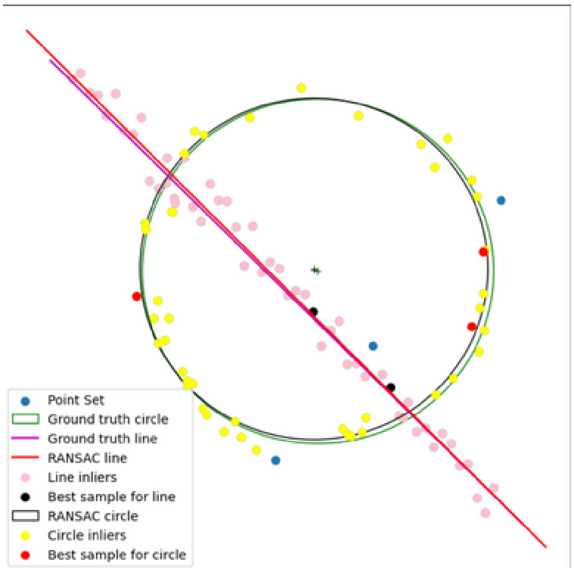
- Center of the largest circle: **(358.0 , 125.0)**
- Largest radius: **23.978262566026945**

- **The sigma values were chosen within the range of 3 to 27, with a step size of 3.**

Question 2

RANSAC parameters:

- S=2 for line and S=3 for circle
- Error threshold t=1 for line, t=1.2 for circle
- Consensus size d=40 for both line and circle




```python
# Fitting the line

iters = 100
min_points = 2
N = X.shape[0]
np.random.seed(14)

thres = 1.   # Error threshold for selecting inliers
d = 0.4 * N # Minimum inlier count for a good fit

best_model_line = None
best_fitted_line = None
best_error = np.inf
best_line_inliers = None
best_line_sample_points = None

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]
```

Figure:Fitting the line using RANSAC

```python
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)
    print(res.x, res.fun)
    return res
```

```python
# Squared error calculation for line and circle
def tls_error_line(params, *args):
    # Error of points denoted by indices, params is the one that should be optimized
    a, b, d = params
    indices, X = args
    error = np.sum((a * X[indices, 0] + b * X[indices, 1] - d)**2)
    return error
```

- If the circle is fitted first, there is a possibility that the three randomly chosen points may all lie on the line. In that case, the resulting circle would be large and resemble a line. However, since the RANSAC algorithm goes through multiple iterations with different point samples, it is still feasible to accurately fit the circle without excluding the line points.

Question 3

```python
# Find homography and apply perspective transformation
homography_matrix, status = cv.findHomography(src_points, dst_points)
warped_img = cv.warpPerspective(overlay_img, homography_matrix, (background_img.shape[1], background_img.shape[0]))

# Blend the warped image with the background image
blended_result = cv.addWeighted(background_img, blending_coeffs[i][0], warped_img, blending_coeffs[i][1], blending_coeffs[i]|
```

- In this process, the user selects four points on the original image where the second image will be placed. The code then calculates a homography to align the second image to these points, transforming its perspective accordingly. Finally, the transformed image is blended with the original to produce a combined result.

- For a realistic blend, it's ideal to use images with flat or well-aligned surfaces, ensuring the superimposed image fits naturally with the perspective of the background. This improves the seamlessness of the blend, giving the final image a more cohesive and authentic look.

Question 4

a)

- The features were detected using SIFT feature detection. detectAndcompute and knnMatch functions are used. From testing 0.85 is used as the matching point. The matches are displayed in the image.

```
def sift_match(im1, im2):
    GOOD_MATCH_PERCENT = 0.65
    # Detect sift features
    sift = cv.SIFT_create()
    keypoint_1, descriptors_1 = sift.detectAndCompute(im1,None)
    keypoint_2, descriptors_2 = sift.detectAndCompute(im2,None)
    # Match features.
    matcher = cv.BFMatcher()
    matches = matcher.knnMatch(descriptors_1, descriptors_2, k = 2)
    # Filter good matches using ratio test in Lowe's paper
    good_matches = []
    for a,b in matches:
        if a.distance < GOOD_MATCH_PERCENT*b.distance:
            good_matches.append(a)
    # Extract location of good matches
    points1 = np.zeros((len(good_matches), 2), dtype=np.float32)
    points2 = np.zeros((len(good_matches), 2), dtype=np.float32)
    for i, match in enumerate(good_matches):
        points1[i, :] = keypoint_1[match.queryIdx].pt
        points2[i, :] = keypoint_2[match.trainIdx].pt

    # Plot the matching
    fig, ax = plt.subplots(figsize = (15,15))
    ax.axis('off')
    matched_img = cv.drawMatches(im1, keypoint_1, im2, keypoint_2, good_matches, im2, flags = 2)
    plt.imshow(cv.cvtColor(matched_img,cv.COLOR_BGR2RGB))
    plt.show()

    result = np.concatenate((points1,points2), axis = 1)
    return result
```

Figure: SIFT feature mapping

b)

- The homography between images one and five was computed by first calculating the individual homographies for all five images. The final homography was obtained by multiplying these sequentially from image one to image five. Initially, the calculated homographies showed significant deviations compared to the expected homography. However after applying the multiplication method, the final homography between images one and five aligned closely with the provided homography. This comparison suggests that the homography calculated through sequential multiplication closely matches the expected transformation, indicating its accuracy.

```python
def calculateHomography(correspondences):
    # Initialize a list to store the linear equation system
    equation_list = []

    # Loop through each pair of correspondences
    for points in correspondences:
        # Create point matrices for both images
        p1 = np.matrix([points.item(0), points.item(1), 1])  # (x1, y1) in first image
        p2 = np.matrix([points.item(2), points.item(3), 1])  # (x2, y2) in second image

        # Set up two equations for each correspondence point
        equation2 = [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)]
        equation1 = [-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)]

        # Add both equations to the list
        equation_list.append(equation1)
        equation_list.append(equation2)

    # Create the matrix from the linear system
    A = np.matrix(equation_list)

    # Perform singular value decomposition (SVD)
    u, s, v = np.linalg.svd(A)

    # Reshape the last column of V (smallest singular value) to form the homography matrix
    h = np.reshape(v[8], (3, 3))

    # Normalize the homography matrix
    h /= h.item(8)
```

Figure: Homography calculation

```
[[ 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: Calculated Homography

```
[[ 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: Calculated Homography before multiplication

c)

- The generated homography was used to align image five over the warped version of image one. Due to the limited number of strong matches identified between the two images, the accuracy of the stitching process was affected. This scarcity of high-quality matches impacted the seamless integration of the images.