EN3160 Assignment 2 on Fitting and Alignment

Walgampaya H.K.B 220675T
GitHub Link: GitHub

(https://github.com/HimethW/Fitting-and-Alignment/blob/main/Assignment2.ipynb)

1 Question 01

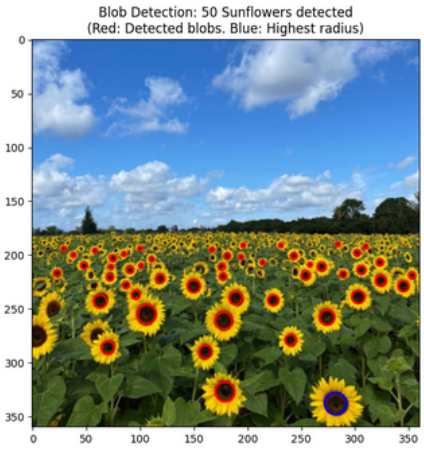
In this question Laplacian of Gaussian and scale-space extrema detection was used to identify blobs and draw circles in the sunflower field image.

Methodology

- First We load the image and convert it to gray scale.
- Then we create a kernal which is a Laplacian of a 2D gaussian distribution and convolve it with the original image.

```python
def create_laplacian_of_gaussian_kernel(sigma):
    half_width = int(3 * sigma)  # Half width of kernal
    size = 2 * half_width + 1    # Make it odd size
    x,y = np.arange(-half_width, half_width + 1),np.arange(-half_width, half_width + 1)
    X, Y = np.meshgrid(x, y)
    # Calculate Laplacian of Gaussian formula
    r_squared = X**2 + Y**2
    log_kernel = ((r_squared)/(2 * sigma**2)-1)*np.exp(-r_squared/(2*sigma**2))/(np.pi*
    sigma**4)
    scaled_log_kernel = (sigma ** 2) * log_kernel # Multiply by sigma^2
    return scaled_log_kernel
log_kernel = create_laplacian_of_gaussian_kernel(sigma)
response = cv.filter2D(gray, cv.CV_32F, log_kernel)
```

- Do this for each sigma and get the scale_space. Then find local maxima for each sigma. These are the centers of the blobs



Radius range	2 to 13
Sigma range	1.41421 to 9.1923
Blobs detected	50

As shown in the image several of the main sunflowers has been identified and the largest one is shown in blue. The largest blobs are as follows.

Center (x, y)	Radius	σ	Conv. Response
(282.0, 338.0)	10.0	7.07	0.3304
(106.0, 255.0)	9.0	6.36	0.3426
(179.0, 327.0)	9.0	6.36	0.3274
(178.0, 261.0)	8.0	5.66	0.3299
(274.0, 258.0)	7.0	4.95	0.3206

The radius range was picked based on the visual size of the sunflower blobs. the radius range is `2 to 13` and a total of 50 sunflowers were detected. This radius range suffices because the largest blob found was at radius 10. The sigma value can be taken by $\frac{r}{\sqrt{2}}$. So the sigma range taken was `1.41421356 to 9.19238816`

As shown below we can see that the response for different sigma values are different from one another. for larger sigma values the kernal size is set to a larger value (3 standard deviations to contain 99.98%). Moreover, for large and small sigma values, the detected blobs from the convolution is different based on the radius of blobs. Finally, to avoid detecting trees and the sky, a threshold value is given to each convolution result (take the top 0.03% percentile)

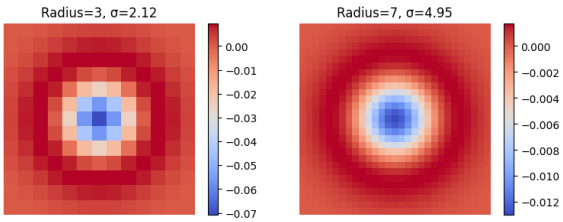


Figure 1: gaussian kernals

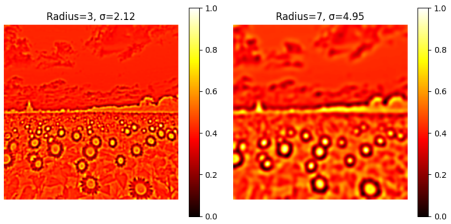


Figure 2: convolution results

2 Question 02

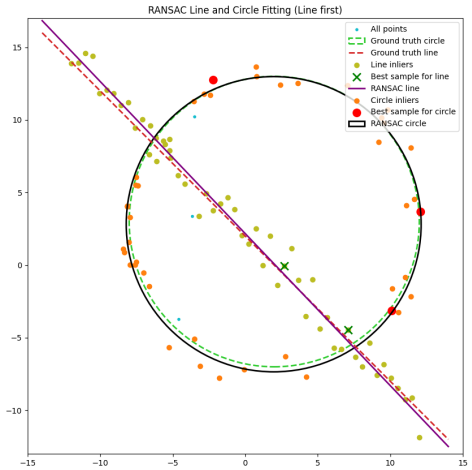
```python
def fit_line_ransac(data, threshold, n_iterations):
    """
    Fits a line to data using a custom RANSAC algorithm.
    The line model is ax + by - d = 0, where [a, b] is a unit normal vector.
    """
    best_inliers = None
    best_model = None       # Best line parameters (a,b,d)
    best_sample = None      # The 2 points used to generate the best model

    for _ in range(n_iterations):
        # Randomly sample 2 points
        sample_indices = np.random.choice(len(data), 2, replace=False)
        sample = data[sample_indices]

        # Fit a line to the sample
        p1, p2 = sample
        # Line equation ax + by - d = 0
        a = p2[1] - p1[1]
        b = p1[0] - p2[0]

        # Ensure it's a unit normal vector
        norm = np.sqrt(a**2 + b**2)
        if norm == 0: continue      #identical points
        a /= norm
        b /= norm

        # Calculate d using one of the points
        d = a * p1[0] + b * p1[1]
        model = (a, b, d)

        # Find inliers by calculating the normal distance
        distances = np.abs(a * data[:, 0] + b * data[:, 1] - d)
        inlier_indices = np.where(distances < threshold)[0]     # Returns a tuple. use [0] get the ar
        inliers = data[inlier_indices]

        # Check if this is the best model so far
        if best_inliers is None or len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_model = model
            best_sample = sample

    # Refit the line using all inliers from the best model (Total Least Squares and SVD)
    if best_inliers is not None and len(best_inliers) > 1:
        centroid = np.mean(best_inliers, axis=0)

        centered_data = best_inliers - centroid
        U, S, Vt = np.linalg.svd(centered_data) # U = col vector, S = row vector, Vt = 2x2 matrix
        # The normal to the line is the last right singular vector
        a, b = Vt[-1]
        d = a * centroid[0] + b * centroid[1]
        best_model = (a, b, d)

    return best_model, best_inliers, best_sample
```

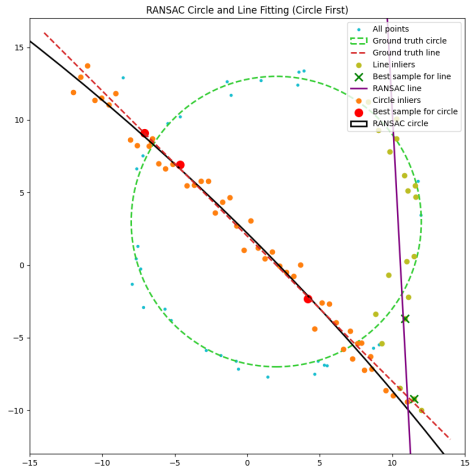
Figure 3: RANSAC for line

```python
def fit_circle_ransac(data, threshold, n_iterations):
    """
    Fits a circle to data using a custom RANSAC algorithm.
    The circle model is (x - xc)^2 + (y - yc)^2 = r^2.
    """
    best_inliers = None
    best_model = None
    best_sample = None

    for _ in range(n_iterations):
        # Randomly sample 3 points
        sample_indices = np.random.choice(len(data), 3, replace=False)
        sample = data[sample_indices]

        # Fit a circle to the sample
        p1, p2, p3 = sample
        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
        ])
        try:
            # Solve for the center (xc, yc)
            xc, yc = np.linalg.solve(A, B)
            # Calculate the radius
            r = np.sqrt((p1[0]-xc)**2 + (p1[1]-yc)**2)
            model = (xc, yc, r)
        except np.linalg.LinAlgError:
            # The 3 points are collinear so cannot form a circle
            continue

        # Find inliers by calculating the radial error
        distances = np.abs(np.sqrt((data[:, 0] - xc)**2 + (data[:, 1] - yc)**2) - r)
        inlier_indices = np.where(distances < threshold)[0]
        inliers = data[inlier_indices]

        # Check if this is the best model so far
        if best_inliers is None or len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_model = model
            best_sample = sample

    # Refit the circle using all inliers and scipy.minimize
    def cost_function(params, inliers):
        xc, yc, r = params
        # Sum of squared radial errors
        return np.sum((np.sqrt((inliers[:, 0] - xc)**2 + (inliers[:, 1] - yc)**2) - r)**2)

    if best_inliers is not None and len(best_inliers) > 2:
        initial_guess = best_model
        result = minimize(cost_function, initial_guess, args=(best_inliers,), method='L-BFGS-B')
        best_model = result.x

    return best_model, best_inliers, best_sample
```

Figure 4: RANSAC for circle

d) As shown in the figure below, if we fit the circle first, then the RANSAC performance will be poor or fail completely. The reason is that, the line points will be a large set of outliers for the circle and the RANSAC might select 3 points from those. In that case the estimated circle will be very large and the line points will be considered as inliers to the circle due to the large radius. Then, when using the remaining points for the line it will not match, as the line points are taken up by the circle. This doesn't happen when the line is fitted first. So it is best to first fit the simple model.





3 Question 03

```python
def warp_and_blend_image(image, background_img, destination_points):
    """
    Warp the image to fit the selected background area and blend it
    """
    # Get image dimensions
    h_image, w_image = image.shape[:2]

    # Source points which are the corners of image
    source_points = np.array([
        [0, 0],              # Top-left
        [w_image-1, 0],      # Top-right
        [w_image-1, h_image-1],  # Bottom-right
        [0, h_image-1]       # Bottom-left
    ], dtype=np.float32)

    # Calculate homography matrix
    H, status = cv2.findHomography(source_points, destination_points)
    print(H)

    # Warp the image
    h_background, w_background = background_img.shape[:2]
    warped_image = cv2.warpPerspective(image, H, (w_background, h_background))

    mask = np.ones((h_image, w_image), dtype=np.uint8) * 255
    warped_mask = cv2.warpPerspective(mask, H, (w_background, h_background))

    # Create boolean mask
    mask_bool = warped_mask > 0

    # Blend the image into the background
    result = background_img.copy()
    alpha = 0.7
    result[mask_bool] = (warped_image[mask_bool] * alpha +
                         background_img[mask_bool] * (1-alpha)).astype(np.uint8)

    return result
```

Methodology

- First load the images and then select the destination points using the mouse on the background image where where our original image will be inserted.
- Then, as shown in the image code, get the source image points and calculate the homography matrix H.
- The calculate the warped image using H. This will give us the warped version of our image on a black background. Create a ask that covers the warped image part.
- combine the original background with the warped image using the mask. A parameter `alpha` is introduced to preserve the original texture of the background by keeping a proportion of the background under the warped image.

Figure 5: Result 1

Figure 6: Result 2

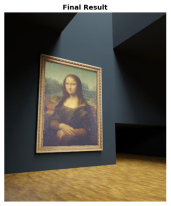


Figure 7: Result 3

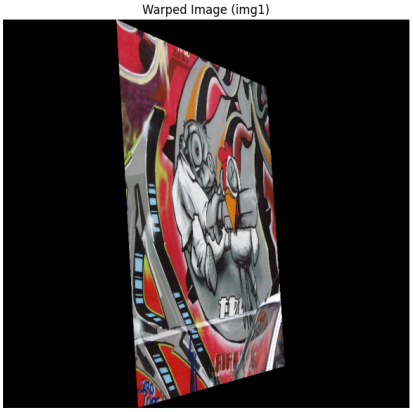
explain the (non-technical) rationale of the choices

1. **Result 1:** Using this method, people who love decorating their cars can try out different patterns before permanently applying them
2. **Result 2:** Marketing teams can try out advertising methods at real locations before printing and installing them which can be costly if done wrong
3. **Result 3:** Home owners can use this to visualize different artworks in their homes and decide whether they like it or not before purchasing them

4 Question 04

Methodology:

- First, both img1 and img2 was loaded and the SIFT detector was run through them to identify features.
- After identifying the keypoints and descriptors, a nearest neighbor algorithm was executed and the Lowe's ratio test was run to identify matching pairs.
- Then four random matching pairs was selected and the homography matrix was computed by estimating the projective transformation.
- After that, each source keypoint was projected to a destination point using the estimated transformation. Then a custom RANSAC algorithm was implemented to compare these predicted values with the true destination values and get the number of inliers. This process was repeated several times. Finally, the best model with the highest number of inliers was taken.
- This was repeated between img1 and img2, then img2 and img3, all the way upto img4 and img5. Finally the resulting matrices were multiplied together sequentially to get the final homography matrix from img1 to img5



The Error between computed and given matrices is as follows:

Transformation	Mean Absolute Error	Frobenius Norm Error
img1 to img2	0.0809	0.5189
img1 to img3	0.1476	0.9363
img1 to img4	0.1308	0.8660
img1 to img5	0.2689	1.7483

```python
def get_sift_features(img1, img2):
    img1_gray = cv2.cvtColor(img1, cv2.COLOR_RGB2GRAY)      #since SIFT works on single channel
    img2_gray = cv2.cvtColor(img2, cv2.COLOR_RGB2GRAY)
    sift = cv2.SIFT_create(nOctaveLayers=3, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)
    keypoints1, descriptors1 = sift.detectAndCompute(img1_gray, None)
    keypoints2, descriptors2 = sift.detectAndCompute(img2_gray, None)
    if descriptors1 is None or descriptors2 is None:
        raise ValueError("No descriptors found in one or both images.")
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)
    #Lowe ratio test
    best_matches = [m for m, n in matches if m.distance < 0.75 * n.distance]
    return best_matches, keypoints1, keypoints2

def get_inliers(src_full, dst_full, tform, thres):
    dst_check = tform(src_full)
    errors = np.sqrt(np.sum((dst_check - dst_full) ** 2, axis=1))
    return np.where(errors < thres)[0]

def find_best_homography(good_matches, keypoints1, keypoints2):
    src_points_list = []  # Will store points from image 1
    dst_points_list = []  # Will store points from image 2
    for i in good_matches:
        query_idx = i.queryIdx  # Get the index of the keypoint in image 1
        point_in_img1 = keypoints1[query_idx].pt  # (x,y)
        src_points_list.append(point_in_img1)
        train_idx = i.trainIdx  # Get the index of the matching keypoint in image 2
        point_in_img2 = keypoints2[train_idx].pt  # (x,y)
        dst_points_list.append(point_in_img2)
    src_full = np.array(src_points_list, dtype=np.float32)
    dst_full = np.array(dst_points_list, dtype=np.float32)

    if len(src_full) < 4:
        raise ValueError(f"Not enough matches ({len(src_full)} < 4) to compute homography.")
    num_points = 4
    thres = 1.0
    iters = 200
    min_inliers = 4
    best_homography = None
    best_inlier_count = 0
    best_inliers = None
    np.random.seed(62)
    for i in range(iters):
        chosen_matches = np.random.choice(good_matches, num_points, replace=False)
        src_points = np.array([keypoints1[m.queryIdx].pt for m in chosen_matches], dtype=np.float32)
        dst_points = np.array([keypoints2[m.trainIdx].pt for m in chosen_matches], dtype=np.float32)
        try:
            tform = transform.estimate_transform('projective', src_points, dst_points)
            inliers = get_inliers(src_full, dst_full, tform, thres)
            if len(inliers) > best_inlier_count:
                best_inlier_count = len(inliers)
                best_homography = tform
                best_inliers = inliers
        except:
            continue
    if best_inlier_count >= min_inliers:
        src_inliers = src_full[best_inliers]
        dst_inliers = dst_full[best_inliers]
        best_homography = transform.estimate_transform('projective', src_inliers, dst_inliers)
        print(f"Custom RANSAC: Best no. of inliers = {best_inlier_count}")
        return best_homography, best_inliers
    else:
        print("Custom RANSAC failed.")
```