# EN3160 Assignment 2 on Fitting and Alignment

Rathnasekara T.S.     200529H

GitHub Link : *Click here to visit*

## Question 01

```python
# define log kernel
def log_kernel(sigma, size):
    if size % 2 == 0:
        size += 1
    sigma2 = sigma ** 2
    idx_range = np.linspace(-(size - 1) / 2., (size - 1) / 2., size)
    x_idx, y_idx = np.meshgrid(idx_range, idx_range)
    tmp_cal = -(np.square(x_idx) + np.square(y_idx)) / (2. * sigma2)
    kernel = np.exp(tmp_cal)
    kernel[kernel < np.finfo(float).eps * np.amax(kernel)] = 0
    k_sum = np.sum(kernel)
    if k_sum != 0:
        kernel /= np.sum(kernel)
    tmp_kernel = np.multiply(kernel, np.square(x_idx) + np.square(y_idx) - 2 * sigma2) / (sigma2 ** 2)
    kernel = tmp_kernel - np.sum(tmp_kernel) / (size ** 2)
    return kernel
```

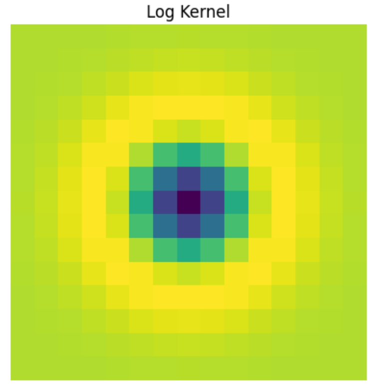



*Figure 1: Laplacian of Gaussians Filter*

Here, Laplacian of gaussian filter has been implemented for blob detection. The shape of the kernel can be shown above.

```python
# blob detection
def detect_blobs(img, sigma_scale, threshold):
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    gray = np.float32(gray)
    cv.normalize(gray, gray, 1, 0, cv.NORM_MINMAX)

    sigma0 = np.sqrt(2)
    k = np.sqrt(2)
    num_scales = sigma_scale
    sigmas = sigma0 * np.power(k, np.arange(num_scales))
    sigmas = [num_scales]*num_scales


    # apply LoG kernel filtering with scaled kernel size and sigma
    img_stack = None
    for i in range(num_scales):
        size = int(2 * np.ceil(4 * sigmas[i]) + 1)
        # with Laplacian response normalization
        kernel = log_kernel(sigmas[i], size) * np.power(sigmas[i], 2)
        filtered = cv.filter2D(gray, cv.CV_32F, kernel)
        filtered = pow(filtered, 2)
        if i == 0:
            img_stack = filtered
        else:
            img_stack = np.dstack((img_stack, filtered))

    # Maximum response extraction
    scale_space = None
    for i in range(num_scales):
        filtered = cv.dilate(img_stack[:, :, i], np.ones((3, 3)), cv.CV_32F, (-1, -1), 1, cv.BORDER_CONSTANT)
        if i == 0:
            scale_space = filtered
        else:
            scale_space = np.dstack((scale_space, filtered))
    max_stack = np.amax(scale_space, axis=2)
    max_stack = np.repeat(max_stack[:, :, np.newaxis], num_scales, axis=2)
    max_stack = np.multiply((max_stack == scale_space), scale_space)

    radius_vec = None
    x_vec = None
    y_vec = None
    for i in range(num_scales):
        radius = np.sqrt(2) * sigmas[i]
        threshold = threshold
        # filter out redundant response
        valid = (max_stack[:, :, i] == img_stack[:, :, i]) * img_stack[:, :, i]
        valid[valid <= threshold] = 0
        (x, y) = np.nonzero(valid)
        if i == 1:
            x_vec = x
            y_vec = y
            radius_vec = np.repeat(radius, np.size(x))
        else:
            x_vec = np.concatenate((x_vec, x), axis=None)
            y_vec = np.concatenate((y_vec, y), axis=None)
            tmp_vec = np.repeat(radius, np.size(x))
            radius_vec = np.concatenate((radius_vec, tmp_vec), axis=None)

    return x_vec, y_vec, radius_vec
```

This is the code for blob detection. First, the image was converted into gray scale and then it was passed through the "LoG" filter to get filtered images. By using that filter we can enhance the areas where the features which are similar to filter, are present.

So, after filtering , we extracted coordinates of the points which show the maximum response and respective radius were calculated to detect the blob.

Then, this function returns the coordinates of the local maximum responses and radii were calculated according to the sigma values.

**Sigma values range is from 2 to 9.**

*Figure 2: Code for blob detection*

```
Maximum radius:  12.727922061357857
```

```python
img = cv.imread('the_berry_farms_sunflower_field.jpeg', cv.IMREAD_REDUCED_COLOR_4)
x_all = []
y_all = []
radii_all = []
for sigma_scale in range(2, 10):
    x_coords, y_coords, radii = detect_blobs(img, sigma_scale, 0.03)
    x_all.append(x_coords)
    y_all.append(y_coords)
    radii_all.append(radii)

x_all = np.concatenate(x_all, axis=None)
y_all = np.concatenate(y_all, axis=None)
radii_all = np.concatenate(radii_all, axis=None)

output_img = img.copy()
for i in range(int(len(x_all))):
    cv.circle(output_img, (y_all[i], x_all[i]), int(radii_all[i]), (0, 0, 255), 1)
```

*Figure 4: Blobs detection of  various radii*


*Figure 3: Resultant Images*

According to the code, blobs of various radii were detected using this code and using OpenCV, circles were drawn on bob to identify. The resulting images can be seen above.

# Question 02

## (a) Best fitting Line Detection Using RANSAC

```python
def Line_RANSAC(points,distance_threshold):

    S = 2
    max_iterations = 10000

    best_line = None
    best_inliers_index = []

    for _ in range(max_iterations):

        # Randomly select two points to define a line
        sample_indices = np.random.choice(len(points), size=S, replace=False)
        sample = points[sample_indices]

        # Calculate the unit normal vector to the line
        x1, y1 = sample[0]
        x2, y2 = sample[1]
        if x1 == x2:
            continue  # Avoid division by zero
        direction_vector = np.array([y1 - y2, x2 - x1])
        unit_normal_vector = direction_vector / np.linalg.norm(direction_vector)
        d_init =  np.abs(np.dot(unit_normal_vector, np.array([x1, y1])))

        # Calculate the perpendicular distance from the origin to the line
        perpendicular_distances = [np.abs(np.dot(unit_normal_vector, np.array([x, y]))) for x, y in points]

        # Find inliers based on the distance threshold
        inliers_index = [i for i, distance in enumerate(perpendicular_distances) if  np.abs(distance-d_init) < distance_threshold]

        # Update the best model if this one has more inliers
        if len(inliers_index) > len(best_inliers_index):
            best_line = (unit_normal_vector, d_init)
            best_inliers_index = inliers_index
            best_fitting_points = sample

    return best_line, best_inliers_index, best_fitting_points
```

*Figure 5: RANSAC algorithm for line estimation*

For the detecting inliers in given data points first two random points were detected and then line was estimated using these two points.

Then I defined the distance threshold to select the best inliers and run this loop several times to get best inliers from given data set.

## (b) Circle Estimation Using RANSAC

```python
def circle_RANSAC(points,radial_distance_threshold):

    S = 3
    max_iterations = 10000

    best_circle = None
    best_inliers_index = []

    for _ in range(max_iterations):

        # Randomly select three points to define a circle
        sample_indices = np.random.choice(len(points), size=S, replace=False)
        sample = points[sample_indices]

        # Calculate the center and radius of the estimated circle
        x1, y1 = sample[0]
        x2, y2 = sample[1]
        x3, y3 = sample[2]

        # use optimizer to get values
        result = minimize(lambda params: np.sum(circle_equation(params, sample)**2), [0,0,1])

        #optimized center coordinates and radius
        h_opt, k_opt, r_opt = result.x

        # Calculate the perpendicular distance from the origin to the circle
        radial_distances = [np.abs(np.sqrt((x - h_opt)**2 + (y - k_opt)**2) - r_opt) for x, y in points]

        # Find inliers based on the distance threshold
        inliers_index = [i for i, distance in enumerate(radial_distances) if distance < radial_distance_threshold]

        # Update the best model if this one has more inliers
        if len(inliers_index) > len(best_inliers_index):
            best_circle = (np.array([h_opt,k_opt]), r_opt)
            best_inliers_index = inliers_index
            best_fitting_points = sample

    return best_circle, best_inliers_index, best_fitting_points
```

*Figure 6: Circle Estimation code*

This algorithm is also same as above line detection code. Instead of using two points, here I used three points for estimating the best fitting circles. Apart from that part, all the other parts are same as before.

Here, algorithms tries to obtain maximum numbers of inliers.

Output of above function were represented as follows.

## (c) Plotting Estimations

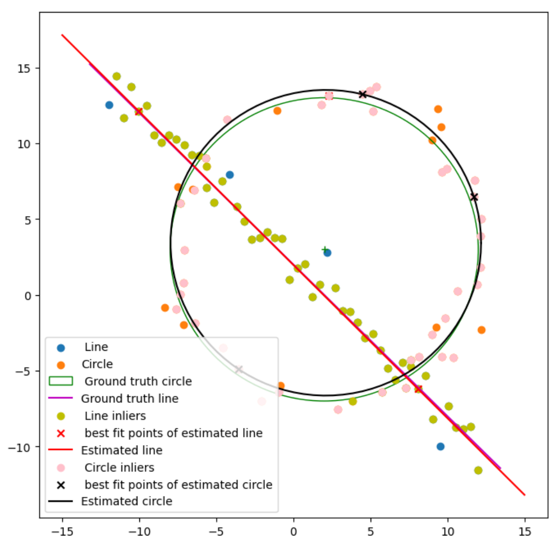



*Figure 7:Estimated line and circle*

## (d) What will happen if we fit the circle first ?

If we first fit the circle, then it uses some points which belong to the line and selected all inliers are not from the circle. So, this gives wrong prediction. As well as , after calculating circle, if we subtract the inliers of circle then line points will be separated into 3 groups and that leads to wrong prediction.

# Question 03

```python
# 4 corners of the image
X = np.array([[0, 0, 1],[image_2.shape[1], 0, 1],[image_2.shape[1], image_2.shape[0], 1],[0, image_2.shape[0], 1]]).T

# selected 4 points
Y = np.array([[coordinates[0][0], coordinates[0][1], 1],[coordinates[1][0], coordinates[1][1], 1],[coordinates[2][0],
              coordinates[2][1], 1],[coordinates[3][0], coordinates[3][1], 1],]).T

# calculate homography matrix
H = cv.findHomography(X[:2].T, Y[:2].T)[0]

# calculate the transformed image
tran_image_2 = cv.warpPerspective(image_2, H, (image_1.shape[1], image_1.shape[0]))

#blending two images
alpha = 0.7
beta = 1 - alpha
blended_img = cv.addWeighted(image_1, alpha, tran_image_2, beta, 0)
blended_img[blended_img > 1] = 1

# convert images to RGB
original_image_rgb = cv.cvtColor(original_image.astype(np.float32), cv.COLOR_BGR2RGB)
flag_img_rgb = cv.cvtColor(image_2.astype(np.float32), cv.COLOR_BGR2RGB)
superimposed_img_rgb = cv.cvtColor(blended_img.astype(np.float32), cv.COLOR_BGR2RGB)
```

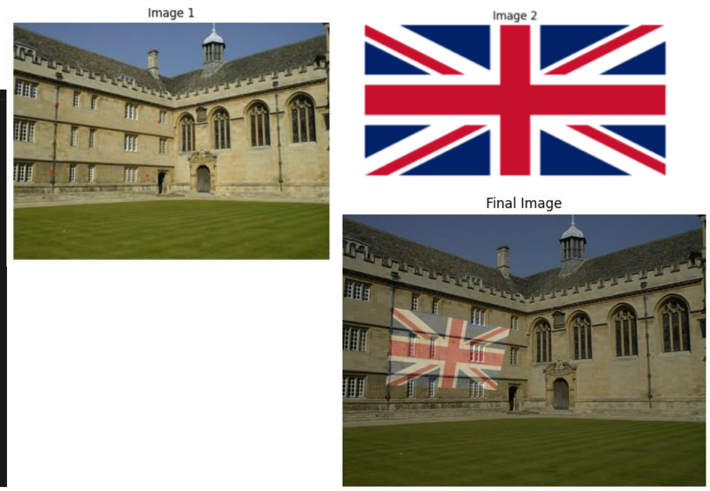


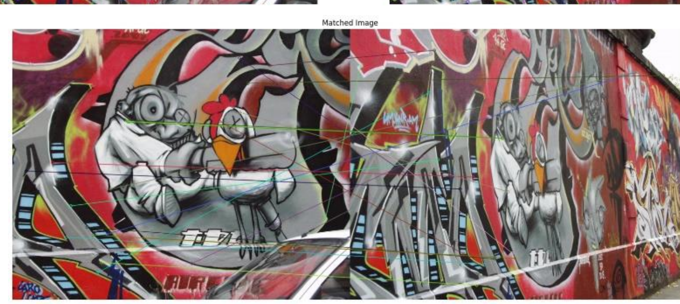

*Figure 8: Superimposed image and code*

Here I used selected four points and four points of image 2 to calculate homography matrix and then do warping using in built function of OpenCV . After these images were blended to obtain superimposed images.

# Question 04

(a)

```python
#create a sift object
sift = cv.SIFT_create(nOctaveLayers=3, contrastThreshold=0.1, edgeThreshold=10, sigma=1)

#detect keypoints and compute descriptors
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

#draw keypoints
img1_keypoints = cv.drawKeypoints(img1, kp1, None)
img2_keypoints = cv.drawKeypoints(img2, kp2, None)

#match keypoints
bf = cv.BFMatcher()
matches = bf.knnMatch(des1, des2, k=2)

#apply ratio test
valid_points = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        valid_points.append([m])

#draw matches
img_matches = cv.drawMatchesKnn(img1, kp1, img2, kp2,valid_points, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```


(b) , (C)

```python
# Define a function to compute the Euclidean distance between two points after applying the homography
def dist(P1, P2, H):
    p1 = np.array([P1[0], P1[1], 1])
    p2 = np.array([P2[0], P2[1], 1])

    p2_estimate = np.dot(H, p1.T)
    p2_estimate = (1 / p2_estimate[2]) * p2_estimate

    return np.linalg.norm(p2.T - p2_estimate)

# Define a function for RANSAC-based homography estimation
def RANSAC_homography(points1, points2, threshold=40, num_iterations=100, min_inlier_count=10):
    inlier_count, selected_inliers, best_H = 0, None, None
    points = np.hstack((points1, points2))

    for _ in range(num_iterations):
        np.random.shuffle(points)
        POINTS1, POINTS1_rem, POINTS2, POINTS2_rem = points[:4, :2], points[4:, :2], points[:4, 2:], points[4:, 2:]
        H = cv.findHomography(POINTS1, POINTS2)[0]
        inliers = [(POINTS1_rem[i], POINTS2_rem[i]) for i in range(len(POINTS1_rem)) if dist(POINTS1_rem[i], POINTS2_rem[i], H) < threshold]

        if len(inliers) > inlier_count and len(inliers) >= min_inlier_count:
            inlier_count = len(inliers)
            selected_inliers = np.array(inliers)
            best_H = H

    return best_H
```

```
[[ 5.90970205e-01  2.57665638e-02  2.24386771e+02]
 [ 2.07077242e-01  1.09334071e+00 -1.68916590e+01]
 [ 4.29459424e-04 -1.35640806e-04  1.00000000e+00]]
```

Homography matrix

Here, I used RANSAC algorithm to find inliers to generate homography matrix and then do warping to get warped image for blending with image 2.

```python
key_points_1, Descriptor_1 = sift.detectAndCompute(IMG1, None)
key_points_2, Descriptor_2 = sift.detectAndCompute(IMG2, None)

# Create a Brute-Force Matcher and perform keypoint matching
bf = cv.BFMatcher()
matches = bf.knnMatch(Descriptor_1, Descriptor_2, k=2)
good, POINTS1, POINTS2 = [], [], []

# Apply the ratio test to filter good matches and extract corresponding keypoints
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good.append([m])
        POINTS1.append(key_points_1[m.queryIdx].pt)
        POINTS2.append(key_points_2[m.trainIdx].pt)

# Convert lists to NumPy arrays for further processing
good, POINTS1, POINTS2 = np.array(good), np.array(POINTS1), np.array(POINTS2)

#calculate homography using RANSAC using build in function
H = RANSAC_homography(POINTS2, POINTS1)
```