EN3160 Assignment 2
Fitting and Alignment

Name: Kavinda W.M.C.
Index No: 200301D
GitHub link: https://github.com/Chamod-Kavinda

Question 1 – Blob detection using Laplacian of Gaussians.

```
# Define range of sigma values
sigma_values = np.linspace(5, 30, 5)

scale_space = []

for sigma in sigma_values:
    kernel_size = int(4 * sigma) + 1
    kernel_hw = kernel_size // 2
    X, Y = np.meshgrid(np.arange(-kernel_hw, kernel_hw + 1), np.arange(-kernel_hw, kernel_hw + 1))
    log_kernel = (X ** 2 + Y ** 2 - 2 * sigma ** 2) * np.exp(-(X ** 2 + Y ** 2) / (2 * sigma ** 2))

    # Apply LoG filtering to the grayscale image
    log_response = cv.filter2D(gray_im.astype(np.float32), -1, log_kernel)

    # Store the result in the scale space
    scale_space.append(log_response)

# Convert the scale space to a numpy array
scale_space = np.array(scale_space)

local_maxima = maximum_filter(scale_space, size=(3, 3, 3))

maxima_coordinates = np.argwhere((scale_space == local_maxima) & (local_maxima > 0))

detected_circles = []

for coord in maxima_coordinates:
    z, y, x = coord
    radius = int(np.sqrt(2)* sigma_values[z])
    center = (x, y)
    detected_circles.append((center, radius))

largest_circle = max(detected_circles, key=lambda x: x[1])
largest_center, largest_radius = largest_circle
```

```
Parameters of the Largest Circle:
Center: (1341, 627)
Radius: 42
Range of Sigma Values Used: 5.0 to 30.0
```

Detected Circles

The largest detected circle, centered at (1341, 627) with a radius of 42 pixels, prominently stands out in the sunflower field image.

The sigma value range of 5 to 30 enabled effective circle detection, suggesting flexibility in adapting this method for different feature scales.

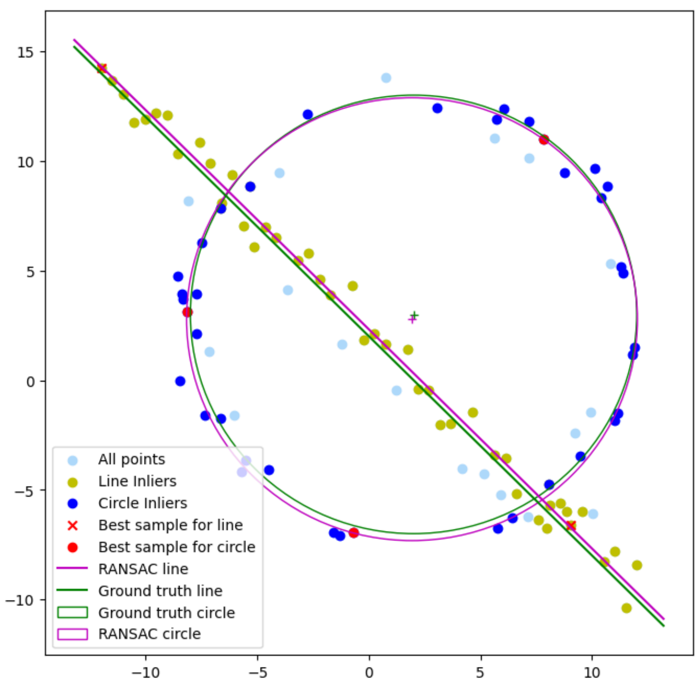
Question 2 – Estimate the line and circle using the RANSAC algorithm

Estimate the line

```python
# Computing the consensus (inliers)
def consensus_line(X_, x, t):
    a, b, d = x[0], x[1], x[2]
    error = np.absolute(a*X_[:,0] + b*X_[:,1] - d)
    return error < t

t = 1.   # Threshold value to determine data points that are fit well by model.
d = 0.4*N   # Number of close data points required to assert that a model fits well to data.
s = 2       # Minimum number of data points required to estimate model parameters.

inliers_line = []        # Indinces of the inliers
max_iterations = 200
iteration = 0
best_model_line = []     # Best model normal (a, b) and distance from origin d
best_error = np.inf
best_sample_line = []    # Three-point sample leading to the best model computation
res_only_with_sample = [] # Result (a, b, d) only using the best sample
best_inliers_line = []   # Inliers of the model computed form the best sample

while iteration < max_iterations:
    indices = np.random.randint(0, N, s) # A sample of three (s) points selected at random
    x0 = np.array([1, 1, 0]) # Initial estimate
    res = minimize(fun = line_tls, args = indices, x0 = x0, tol= 1e-6, constraints=cons, options={'disp': False})
    inliers_line = consensus_line(X_, res.x, t) # Computing the inliers
    #print('rex.x: ', res.x)
    #print('Iteration = ', iteration, '. No. inliners = ', inliers_line.sum())
    if inliers_line.sum() > d:
        x0 = res.x
        # Computing the new model using the inliers
        res = minimize(fun = line_tls, args = inliers_line, x0 = x0, tol= 1e-6, constraints=cons, options={'disp': False})
        #print(res.x, res.fun)
        if res.fun < best_error:
            #print('A better model found ... ', res.x, res.fun)
            best_model_line = res.x
            best_eror = res.fun
            best_sample_line = X_[indices,:]
            res_only_with_sample = x0
            best_inliers_line = inliers_line

    iteration += 1
```

Estimate the circle

```python
remaining_points=X
if best_inliers_line is not None:
    remaining_points= remaining_points[best_inliers_line, :]

def circle_consensus(data, model,t):
    center_x, center_y, radius = model
    distances = np.sqrt((data[:, 0] - center_x)**2 + (data[:, 1] - center_y)**2)
    inliers = np.abs(distances - radius) < t
    return inliers

def circle_tls(x, indices, remaining_points):
    x_center, y_center, r = x[0], x[1], x[2]
    # Calculate the squared differences between the distances and the circle's radius
    squared_errors = np.sqrt((remaining_points[indices, 0] - x_center)**2 + (remaining_points[indices, 1] - y_center)**2 )
    # Return the sum of squared errors
    return np.sum(np.abs(squared_errors-r))

distance_treshold=0.8
con_c={'type': 'ineq', 'fun': lambda x: x[2] - distance_treshold}

def ransac_circle(X, t, iter, in_t):
    n = X.shape[0]
    best_error = np.inf
    best_sample_circle = []
    res_only_with_sample = []
    best_inliers_circle = []

    for i in range(iter):
        indices = np.random.choice(n, 3, replace=False)
        x0 = [1, 1, 1]
        res = minimize(fun=circle_tls, args=(indices,X), x0=x0, tol=1e-6, constraints=con_c, options={'disp': False})
        inliers = circle_consensus(X, res.x, t)
        n_inliers = np.sum(inliers)
        if n_inliers > in_t:
            x0 = res.x
            res = minimize(fun=circle_tls, args=(inliers,X), x0=x0, tol=1e-6, constraints=con_c, options={'disp': False})
            if res.fun < best_error:
                best_error = res.fun
                best_inliers_circle = inliers
                best_x_center, best_y_center, best_r = res.x
                best_sample_circle = indices
                res_only_with_sample = x0

    return best_x_center, best_y_center, best_r, best_inliers_circle, best_sample_circle, res_only_with_sample
```



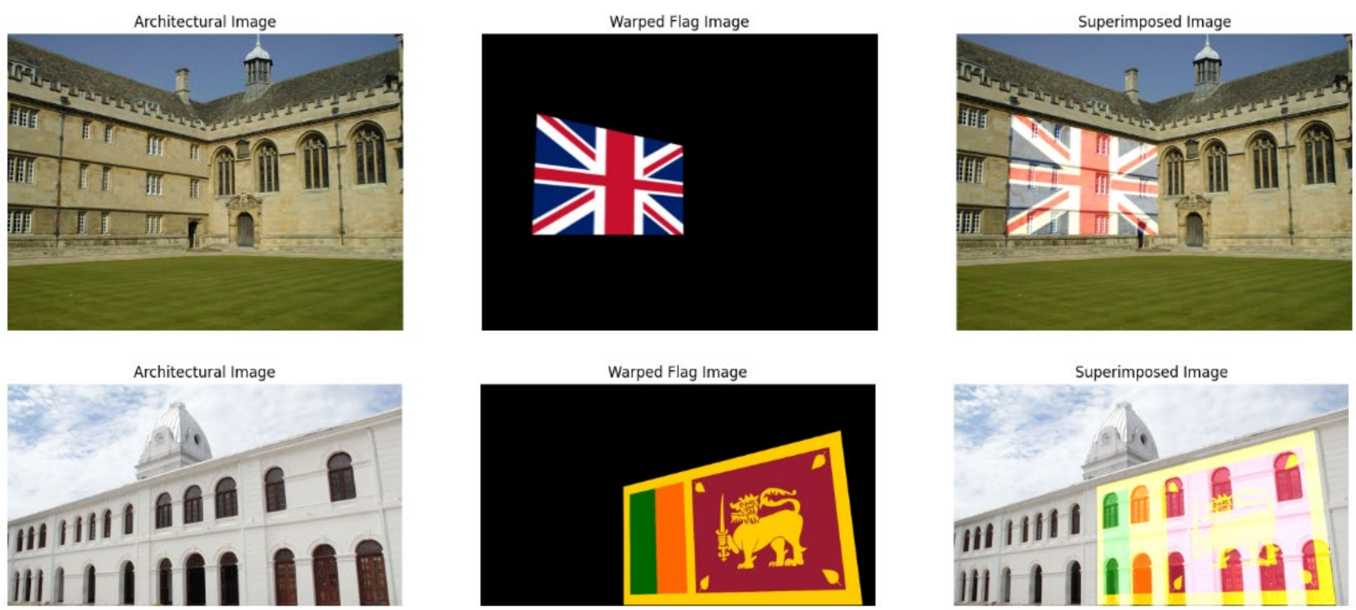
The generated output shows the RANSAC algorithm's effectiveness in accurately estimating robust line and circle models from noisy data, prominently displaying inliers and facilitating a clear comparison with ground truth models, thereby validating the algorithm's reliability for model fitting in the presence of outliers.

What will happen if we fit the circle first?

If fit the circle first, the RANSAC algorithm might struggle to accurately estimate the circle due to the presence of outlier points belonging to the line, leading to a potentially poor circle fit and subsequently impacting the line fitting results. It's generally recommended to fit the dominant model (the line) first to identify and remove its inliers, leaving a cleaner dataset for secondary model fitting (the circle).

Question 3 – Computing Homography

```python
corner_points = []

def click_event(event, x, y, flags, param):
    global corner_points
    if event == cv.EVENT_LBUTTONDOWN:
        corner_points.append((x, y))
        cv.circle(architectural_image1, (x, y), 5, (0, 0, 255), -1)
        cv.imshow('Architectural Image', architectural_image1)

        if len(corner_points) == 4:
            cv.destroyAllWindows()

architectural_image1 = cv.imread('Images/architectural image.jpg', cv.IMREAD_COLOR)
flag_image1 = cv.imread('Images/Flag_of_the_United_Kingdom.png', cv.IMREAD_COLOR)

architectural_imaga_rgb = cv.cvtColor(architectural_image1, cv.COLOR_BGR2RGB)

# Display the architectural image and set a mouse callback function
cv.imshow('Architectural Image', architectural_image1)
cv.setMouseCallback('Architectural Image', click_event)
cv.waitKey(0)

pts_architecture = np.array(corner_points, dtype=np.float32)
pts_flag = np.array([[0, 0], [flag_image1.shape[1], 0], [flag_image1.shape[1], flag_image1.shape[0]], [0, flag_image1.shape[0]]], dtype=np.float32)

homography_matrix, _ = cv.findHomography(pts_flag, pts_architecture)

# Warp the flag image
flag_warped = cv.warpPerspective(flag_image1, homography_matrix, (architectural_image1.shape[1], architectural_image1.shape[0]))
flag_warped_rgb = cv.cvtColor(flag_warped, cv.COLOR_BGR2RGB)

alpha1 = 0.6

superimposed_image = cv.addWeighted(architectural_imaga_rgb, 1, flag_warped_rgb, alpha1, 0)
```

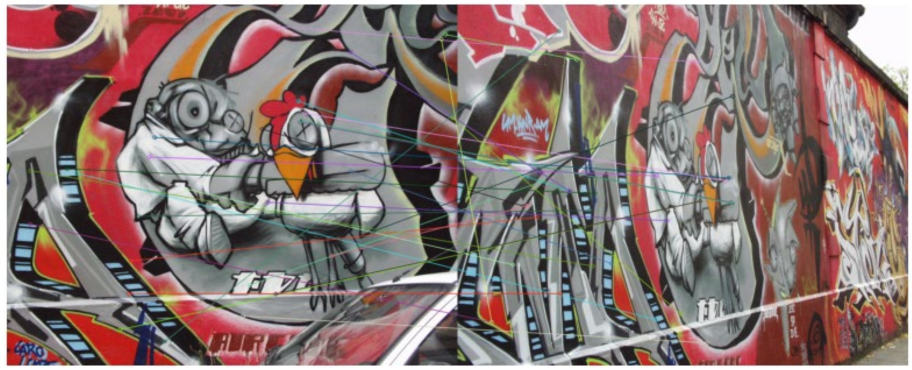


This effectively superimposes a flag onto an architectural image by utilizing homography transformations and image blending. The output result showcases a seamless integration of the flag with the architectural background, demonstrating the code's ability to create visually appealing and customizable compositions, making it a useful tool for creative image manipulation.

Question 4 – Stitch the two Graffiti images

Compute and match SIFT features

```python
# Create a SIFT detector
sift = cv.SIFT_create()

# Find key points and descriptors in both images
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints5, descriptors5 = sift.detectAndCompute(img5, None)

# Create a Brute Force Matcher
bf = cv.BFMatcher()

matches = bf.knnMatch(descriptors1, descriptors5, k=2)

good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)

matched_img = cv.drawMatchesKnn(img1, keypoints1, img5, keypoints5, [good_matches], None, flags=2)
matched_img=cv.cvtColor(matched_img, cv.COLOR_BGR2RGB)
```



Compute the homography using RANSAC

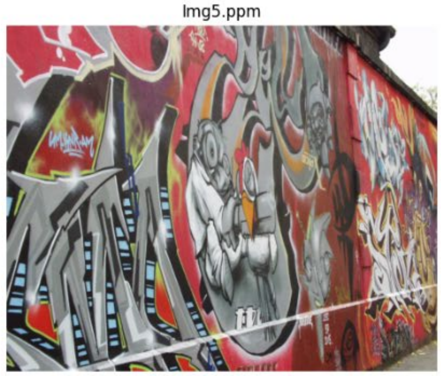
```python
def random_indices(n, t):
    random_num = np.random.randint(n, size=t)
    counts = np.zeros(np.shape(random_num))

    for i in range(len(random_num)):
        counts[i] = np.sum(random_num==random_num[i])
    if np.sum(counts) == len(counts):
        return random_num
    else:
        return random_indices(n,t)

def Compute_Homography(src_points, dest_points):
    x1, y1, x2, y2, x3, y3, x4, y4 = dest_points[0], dest_points[1], dest_points[2], dest_points[3], dest_points[4], d
    x1T, x2T, x3T, x4T = src_points[0], src_points[1], src_points[2], src_points[3]
    zero_matrix = np.array([[0], [0], [0]])

    matrix_A = np.concatenate((np.concatenate((zero_matrix.T,x1T, -y1*x1T), axis = 1), np.concatenate((x1T, zero_matri
                            np.concatenate((zero_matrix.T,x2T, -y2*x2T), axis = 1), np.concatenate((x2T, zero_matrix.T
                            np.concatenate((zero_matrix.T,x3T, -y3*x3T), axis = 1), np.concatenate((x3T, zero_matrix.T
                            np.concatenate((zero_matrix.T,x4T, -y4*x4T), axis = 1), np.concatenate((x4T, zero_matrix.T
    W, v = np.linalg.eig(((matrix_A.T) @ matrix_A))
    temph= v[:,np.argmin(W)]
    H = temph.reshape((3,3))
    return H
```

```
Homography
[[ 6.13931730e-01  3.62478058e-02  2.24307445e+02]
 [ 2.18446715e-01  1.12279099e+00 -2.08646458e+01]
 [ 4.84866494e-04 -1.04989860e-04  1.00000000e+00]]
```

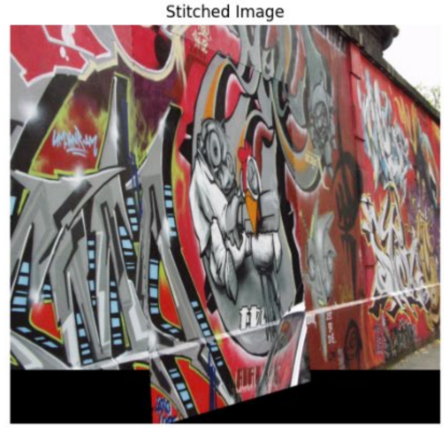
```python
probability = 0.999
sample_size = 4
epsilon = 0.5

N = int(np.ceil(np.log(1-probability) / np.log(1-((1-epsilon)**sample_size))))
H_list = []

for i in range(4):
    sift = cv.SIFT_create()
    keypoints1, descriptors1 = sift.detectAndCompute(img_list[i],None)
    keypoints2, descriptors2 = sift.detectAndCompute(img_list[i+1],None)
    bf_match = cv.BFMatcher(cv.NORM_L1, crossCheck=True)
    matches = sorted(bf_match.match(descriptors1, descriptors2), key = lambda x:x.distance)
    Src_Points = [keypoints1[k.queryIdx].pt for k in matches]
    Dest_Points = [keypoints2[k.trainIdx].pt for k in matches]
    threshold, best_inliers, best_H = 2, 0, 0

    for i in range(N):
        random_points = random_indices(len(Src_Points)-1, 4)

        src_points = []
        for j in range(4):
            src_points.append(np.array([[Src_Points[random_points[j]][0], Src_Points[random_points[j]][1], 1]]))

        dest_points = []
        for j in range(4):
            dest_points.append(Dest_Points[random_points[j]][0])
            dest_points.append(Dest_Points[random_points[j]][1])

        H = Compute_Homography(src_points, dest_points)
        inliers = 0

        for k in range(len(Src_Points)):
            X = [Src_Points[k][0], Src_Points[k][1], 1]
            HX = H @ X
            HX /= HX[-1]
            err = np.sqrt(np.power(HX[0]-Dest_Points[k][0], 2) + np.power(HX[1]-Dest_Points[k][1], 2))

            if err < threshold:
                inliers +=1
```

Img1.ppm

Img5.ppm



Stitched Image



The code effectively stitches img1.ppm onto img5.ppm, producing a visually coherent composite image where the content of both images is seamlessly integrated. SIFT feature matching and RANSAC-based homography estimation contribute to the successful stitching outcome.