EN3160- Image Processing and Machine Vision
A02: Fitting and Alignment

Index No: 200396U Name: Miranda C.M.C.C.

Question 01

```
# Define the LoG filter parameters
sigma = 3
hw = 3*sigma
X, Y = np.meshgrid(np.arange(-hw, hw+1, 1),
np.arange(-hw, hw+1, 1))
LoG_filter = 1/(2*np.pi*sigma**2)*(X**2/
(sigma**2)+Y**2/(sigma**2)-2)*np.exp(-(X**2+
Y**2)/(2*sigma**2))
# Apply the LoG filter
filtered_image = cv.filter2D(gray, -1,
LoG_filter)
# Find local maxima in the filtered image
local_maxima = (filtered_image ==
cv.dilate(filtered_image, np.ones((3, 3))))
maxima_coordinates =np.argwhere(local_maxima)
# Create a list of detected blobs
blobs = []
threshold = 48
for coord in maxima_coordinates:
    if filtered_image[coord[0], coord[1]] >
threshold:
        blobs.append((coord[1], coord[0]))
# Draw the detected blobs on the original image
result = image.copy()
for blob in blobs:
  cv.circle(result, blob, 10, (0, 255, 0), 2)
# Sort the detected blobs by radius
blobs.sort(key=lambda x: filtered_image[x[1],
x[0]], reverse=True)
# Extract the parameters of the largest circle
if blobs:
    largest_blob = blobs[0]
    largest_radius =
filtered_image[largest_blob[1], largest_blob[0]]
    largest_circle_params = (largest_blob[0],
largest_blob[1], largest_radius)
    # Draw the largest circle
    result = image.copy()
    cv.circle(result, (largest_blob[0],
largest_blob[1]), int(largest_radius), (0, 255,
0), 2)
```

Range of δ values used: 2 to 10. (Varying Thresholds)

- Parameters of the largest circle for δ = 2:

  Center: (323, 207)

  Radius: 96

- Parameters of the largest circle for δ = 3:

  Center: (269, 211)

  Radius: 96

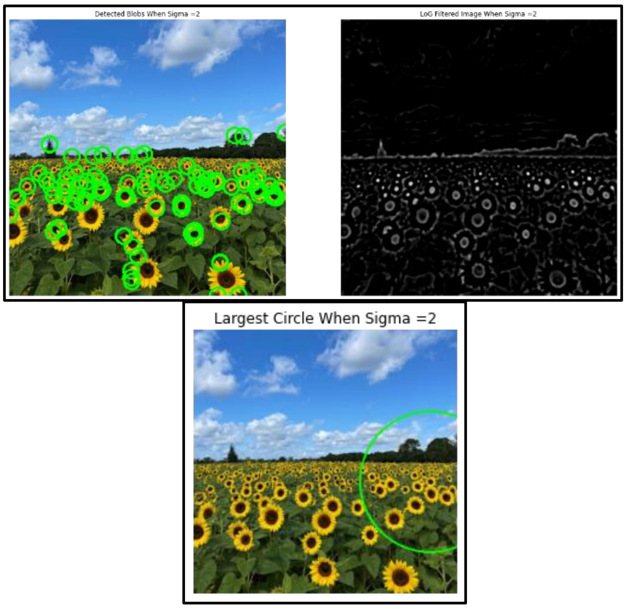

Figure 1: Output for δ = 2

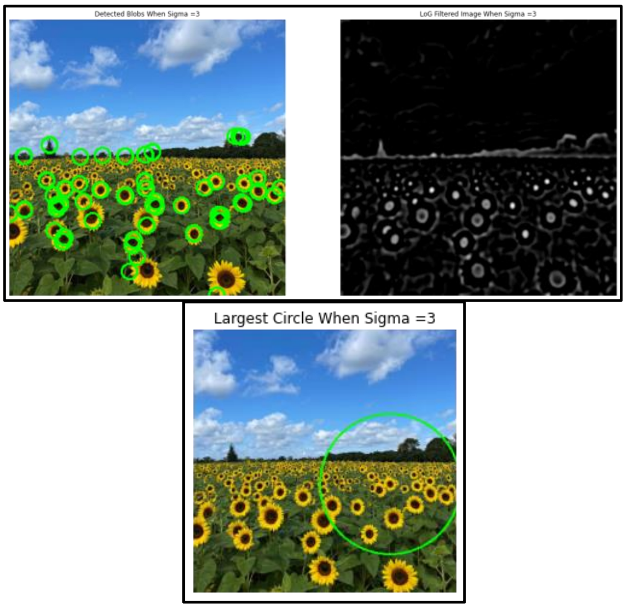

Figure 2: Output for δ = 3

Question 02

(a)

```
def line_equation_from_points(x1,y1, x2, y2):
    # Line equation in the form ax + by = d
    # Calculate the direction vector (Δx, Δy)
    delta_x = x2 - x1
    delta_y = y2 - y1
    # Calculate the normalized vector (a, b)
    magnitude = math.sqrt(delta_x**2 +
delta_y**2)
    a = delta_y / magnitude
```

```
    b = -delta_x / magnitude
    # Calculate d
    d = (a * x1) + (b * y1)
    return a, b, d
def line_tls(x, indices):
    # Return the total least squares error
    a, b, d = x[0], x[1], x[2]
    return np.sum(np.square(a*dataset[indices,0]
+ b*dataset[indices,1] - d))
def g(x):
    # Constraint
    return x[0]**2 + x[1]**2 - 1
cons = ({'type': 'eq', 'fun': g})
def consensus_line(X, x, t):
    # Computing the inliners
    a, b, d = x[0], x[1], x[2]
    error = np.absolute(a*dataset[:,0] +
b*dataset[:,1] - d)
    return error < t
```

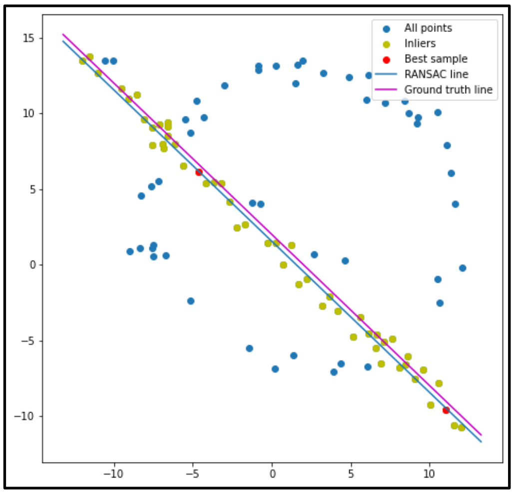

Figure 3: Best line model using the RNASAC algorithm.

(b)

```
def circle_equation(points):
    # Return the center and radius
    p1, p2, p3 = points[0], points[1], points[2]
    temp = p2[0] * p2[0] + p2[1] * p2[1]
    bc = (p1[0]*p1[0]+p1[1] * p1[1] - temp) / 2
    cd = (temp-p3[0]* p3[0] - p3[1] * p3[1]) / 2
    det = (p1[0]-p2[0]) * (p2[1] - p3[1]) -
(p2[0] - p3[0]) * (p1[1] - p2[1])
    # Center of circle
    cx=(bc*(p2[1]-p3[1])-cd*(p1[1]-p2[1]))/det
    cy=((p1[0]-p2[0])*cd-(p2[0]-p3[0])*bc)/det
    radius=np.sqrt((cx-p1[0])**2+(cy-p1[1])**2)
    return ((cx, cy), radius)
def get_inliers(data_list, center, r):
    # The threshold value is taken as 1/5th of the
radius
    inliers = []
    thresh = r//3
    for i in range(len(data_list)):
        error = np.sqrt((data_list[i][0]-
center[0])**2 + (data_list[i][1]-
center[1])**2) - r
        if error < thresh:
            inliers.append(data_list[i])
    return np.array(inliers)
def random_sample(data_list):
    sample_list = []
    random.seed(0)
    rand_nums = random.sample(range(1,
len(data_list)), 3)
    for i in rand_nums:
        sample_list.append(data_list[i])
    return np.array(sample_list)
def calc_R(x_, y_, xc, yc):
    return np.sqrt((x_-xc)**2 + (y_-yc)**2)
def f_2(c, x_, y_):
    Ri = calc_R(x_, y_, *c)
    return Ri - Ri.mean()
def estimateCircle(x_m, y_m, points):
    x_ = points[:,0]
    y_ = points[:,1]
    center_estimate = x_m, y_m
    center_2, ier = optimize.leastsq(f_2,
center_estimate, (x_, y_))
    xc_2, yc_2 = center_2
    Ri_2 = calc_R(x_, y_, *center_2)
    R_2 = Ri_2.mean()
    return (xc_2, yc_2), R_2
```

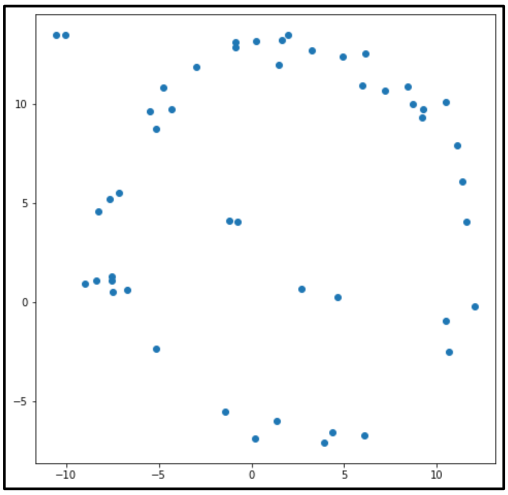

Figure 4: Circle that fits the remnant using RANSAC.

(c)

```
def RANSAC_Circle(data_list, itr):
    best_sample = []
    best_center_sample = (0,0)
    best_radius_sample = 0
    best_inliers = []
    max_inliers = len(data_list)*0.9
    for i in range(itr):
       samples = random_sample(data_list)
       center,radius =
circle_equation(samples)
```

```
        inliers = get_inliers(data_list,c enter,
radius)
        num_inliers = len(inliers)
        if num_inliers > max_inliers:
            best_sample = samples
            max_inliers = num_inliers
            best_center_sample = center
            best_radius_sample = radius
            best_inliers = inliers
     return best_center_sample,
best_radius_sample, best_sample,
best_inliers
# Function to compute homography
def compute_homography():
    global points_image1
    global blended_image
    points_image2 = np.array([[0, 0],
[image2.shape[1], 0], [image2.shape[1],
image2.shape[0]], [0, image2.shape[0]]],
dtype=np.float32)
    # Compute the homography matrix
    homography_matrix, _ =
cv.findHomography(points_image2,
np.array(points_image1, dtype=np.float32))
    warped_image2 = cv.warpPerspective(image2,
homography_matrix, (image1.shape[1],
image1.shape[0]))
    alpha = 0.4
    blended_image = cv.addWeighted(image1, 1,
warped_image2, alpha, 0)
```
(d) In this scenario, we initially apply

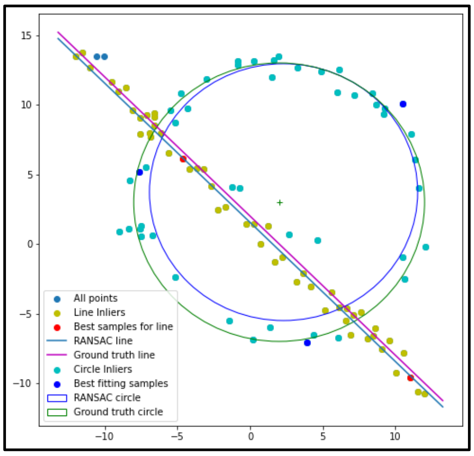

Figure 5: Circle with RANSAC

(d)

In this scenario, we initially apply RANSAC to fit a line and subtract it from the noisy data points. Subsequently, we use RANSAC to fit a circle, benefiting from the improved data quality achieved by the initial line fitting process. By doing this, RANSAC identifies circle inliers more accurately by randomly selecting points, ensuring a better circle fit.

However, if we reverse the order and apply RANSAC to fit the circle first, it may select random points from the line inliers as well. This can lead to inaccurate fits due to the inclusion of erroneous data points in the circle fitting process.

Question 03

```
# Function to compute homography
def compute_homography():
    global points_image1
    global blended_image
points_image2 = np.array([[0, 0],
[image2.shape[1], 0], [image2.shape[1],
image2.shape[0]], [0, image2.shape[0]]],
dtype=np.float32)
    # Compute the homography matrix
    homography_matrix, _ =
cv.findHomography(points_image2,
np.array(points_image1, dtype=np.float32))
    # Warp image2
    warped_image2 =
cv.warpPerspective(image2,
homography_matrix, (image1.shape[1],
image1.shape[0]))
    # Blend the images
    alpha = 0.6
    blended_image = cv.addWeighted(image1,
1, warped_image2, alpha, 0)
```

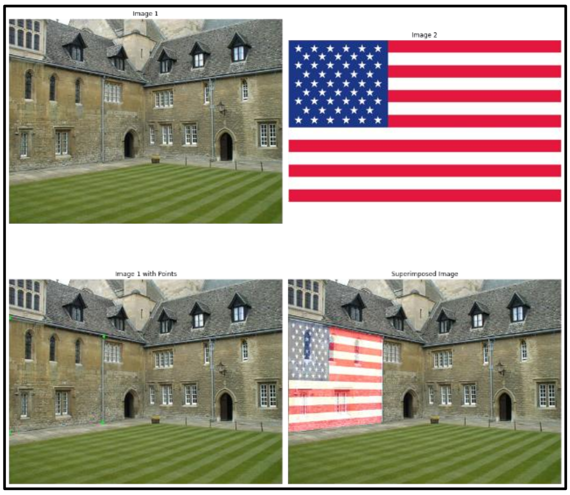

Figure 6: Superimposed Image

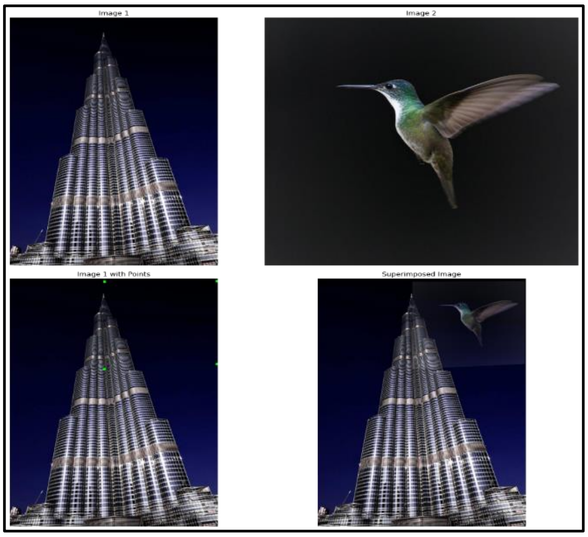

Figure 7: Superimposed Image

Question 04

(a)

```
# Initiate SIFT detector
sift = cv.SIFT_create()
# Find the keypoints and descriptors with SIFT
keypoints1, descriptors1 =
sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 =
sift.detectAndCompute(gray2, None)
```

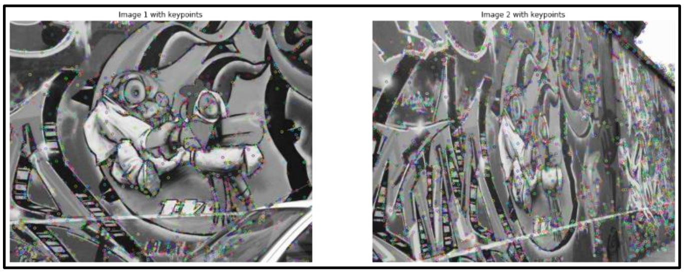

Figure 8: SIFT features.

(b)

```
def compute_homography_RANSAC(matches,
keypoints1, keypoints2, num_iterations=1000,
initial_threshold=4.0):
    best_homography = None
    best_inliers_count = 0
    for _ in range(num_iterations):
        # Randomly select 4
        sample_indices =
np.random.choice(len(matches), 4, replace=False)
        src_points =
np.float32([keypoints1[matches[i].queryIdx].pt
for i in sample_indices])
        dst_points =
np.float32([keypoints2[matches[i].trainIdx].pt
for i in sample_indices])
        # Compute the homography
        homography, _ =
cv.findHomography(src_points, dst_points,
cv.RANSAC, initial_threshold)
        # Count inliers
        inliers_count = 0
        for i, match in enumerate(matches):
            src_pt =
keypoints1[match.queryIdx].pt
            dst_pt =
keypoints2[match.trainIdx].pt
            src_pt_hom = np.array([src_pt[0],
src_pt[1], 1.0])
            projected_pt = np.dot(homography,
src_pt_hom)
            projected_pt /= projected_pt[2]
            error =
np.linalg.norm(np.array([dst_pt[0], dst_pt[1],
1.0]) - projected_pt)
            if error < initial_threshold:
                inliers_count += 1
# Keep the homography with the most inliers
        if inliers_count >
best_inliers_count:
            best_homography = homography
            best_inliers_count =
inliers_count
    return best_homography,
best_inliers_count
```

Figure 9: Matched Image

(c)

```
# Warp image1 to align it with image5
stitched_image = cv.warpPerspective(image1,
known_homography, (image2.shape[1],
image2.shape[0]))
# Blend the two images
alpha = 0.3
beta = 1 - alpha
blended_image =
cv.addWeighted(stitched_image, alpha,
image2, beta, 0.0)
```

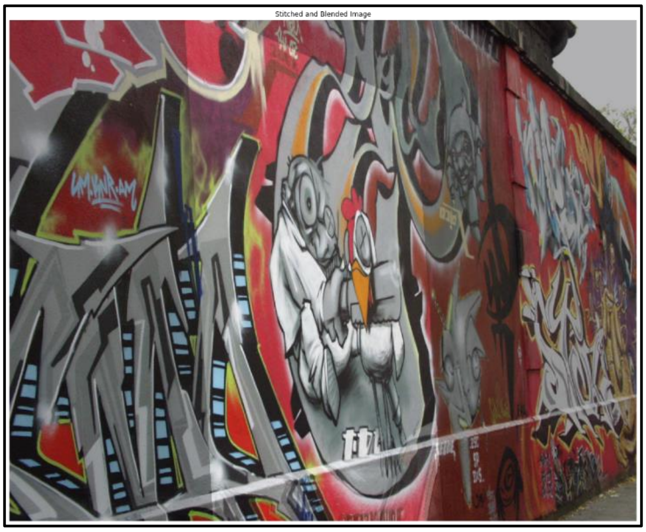

Figure 10: Stitched Image

- **GitHub Link: [A02 Assignment]**