EN3160 Assignment 2 on Fitting and Alignment

Name: De Silva A.L.U.P

Index: 200105F

GitHub: https://github.com/UlinduP/EN3160_Image_Processing_and_Machine_Vision.git

1.

```python
def LoG(sigma):
    #window size
    n = np.ceil(sigma*6)
    y,x = np.ogrid[-n//2:n//2+1,-n//2:n//2+1]
    y_filter = np.exp(-(y*y/(2.*sigma*sigma)))
    x_filter = np.exp(-(x*x/(2.*sigma*sigma)))
    final_filter = (-(2*sigma**2) + (x*x + y*y) ) * (x_filter*y_filter) * (1/(2*np.pi*sigma**4))
    return final_filter

def LoG_convolve(img):
    log_images = []
    for i in range(1,10):
        y = np.power(k,i)
        sigma_1 = sigma*y
        filter_log = LoG(sigma_1)
        image = cv2.filter2D(img,-1,filter_log)
        image = np.pad(image,((1,1),(1,1)),'constant')
        image = np.square(image)
        log_images.append(image)
    log_image_np = np.array([i for i in log_images])
    return log_image_np
log_image_np = LoG_convolve(img)
print(log_image_np.shape)
```

```python
def detect_blob(log_image_np):
    co_ordinates = []
    (h,w) = img.shape
    for i in range(1,h):
        for j in range(1,w):
            slice_img = log_image_np[:,i-1:i+2,j-1:j+2]
            result = np.amax(slice_img)
            #result_1 = np.amin(slice_img)
            if result >= 0.03:
                z,x,y = np.unravel_index(slice_img.argmax(),slice_img.shape)
                co_ordinates.append((i+x-1,j+y-1,k**z*sigma))
    return co_ordinates
co_ordinates = list(set(detect_blob(log_image_np)))
co_ordinates = np.array(co_ordinates)
co_ordinates = redundancy(co_ordinates,0.5)
```

Generated Results


sigma values were used in the range of **3 to 27** with a step size of 3.

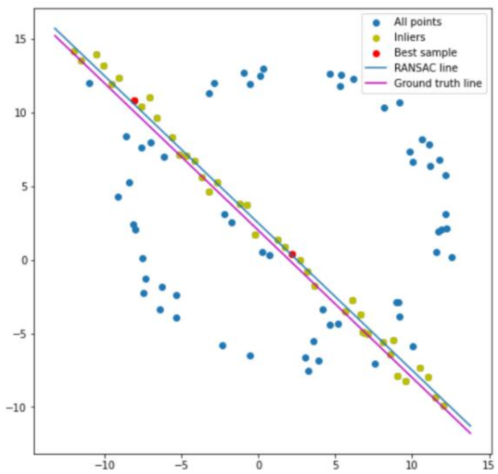
Center of the largest circle: 358, 125

Radius of the largest circle: 23.978

I apply the Laplacian of Gaussian operation to the image in the LoG_convolve function. Then, the blobs are detected after applying a threshold, redundant blobs are removed using detect_blob and redundancy functions, respectively.

2. (a)

```python
inliers_line = []
max_iterations = 50
iteration = 0
best_model_line = []
best_error = np.inf
best_sample_line = []
res_only_with_sample = []
best_inliers_line = []

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': True})
    inliers_line = consensus_line(dataset, res.x, t) # Computing the inliers
    if inliers_line.sum() > d:
        x0 = res.x
        res = minimize(fun = line_tls, args = inliers_line, x0 = x0, tol= 1e-6, constraints=cons, options={'disp': True})
        if res.fun < best_error:
            best_model_line = res.x
            best_eror = res.fun
            best_sample_line = dataset[indices,:]
            res_only_with_sample = x0
            best_inliers_line = inliers_line
    iteration += 1
```



The choice of threshold depends on your data's characteristics and the estimation's desired robustness. A suitable threshold should be set to balance between including true inliers and excluding outliers. The selected normal distance to the estimated line for this dataset is 1.

The number of expected points needs to be selected such that it is neither too restrictive nor lose. I have taken the number of points in the consensus to be 40 percent of all points. Then I am taking the line with the most number of such points.

(b) Subtracting the consensus of the best line

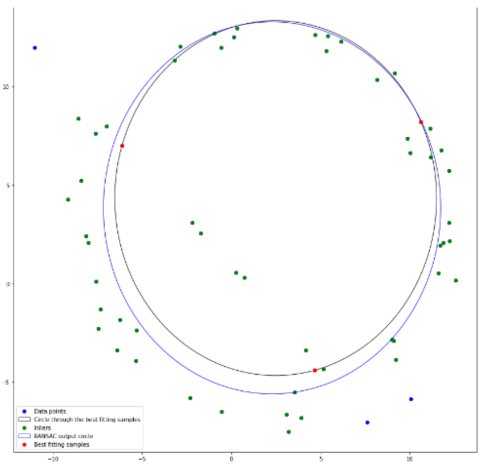
```
outliers_indices = np.where(np.logical_not(best_inliers_line))[0]
outliers_data = dataset[outliers_indices, :]
```

I have set the threshold of error (Radial) to be 1/3rd of the radius and the number of points that must be in the consensus to be 40 percent of all points left after removing line inliers. Then, I am taking the circle with the most number of such inliers and estimating the best circle fitting the selected inliers.

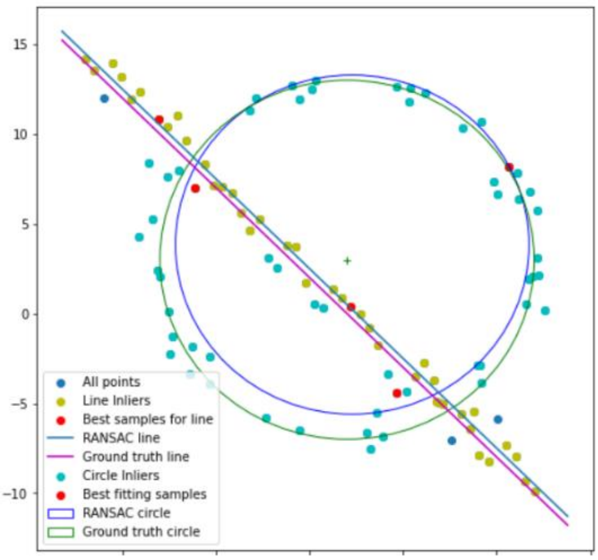
Code for the RANSAC circle

```
def RANSAC_Circle(data_list, itr):
    """ Return the center, radius and the best sample and its inliers used to f
    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)  # Generating a random sample of 3 p
        center, radius = circle_equation(samples) # Calculting the center and t
        inliers = get_inliers(data_list, center, radius) # Get the list of inli
        num_inliers = len(inliers)

        # If a better approximation has been reached
        if num_inliers > max_inliers:
            best_sample = samples
            max_inliers = num_inliers
            best_center_sample = center
            best_radius_sample = radius
            best_inliers = inliers

    print("Center of Sample=", best_center_sample)
    print("Radius of Sample=", best_radius_sample)

    return best_center_sample, best_radius_sample, best_sample, best_inliers
```

RANSAC Circle Fitting



(c)



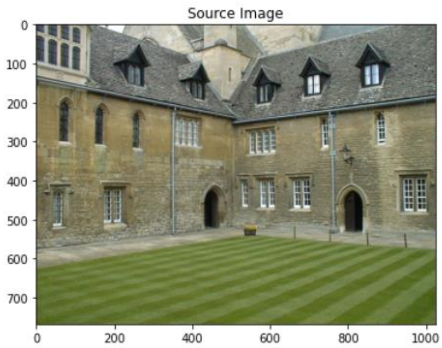

```
def estimateCircle(x_m, y_m, points):
    x_ = points[:,0]
    y_ = points[:,1]
    center_estimate = x_m, y_m
    center_2, ier = optimize.leastsq(f_2, center_estimate, (x_, y_))

    xc_2, yc_2 = center_2

    Ri_2 = calc_R(x_, y_, *center_2)
    R_2 = Ri_2.mean()
    # residu_2    = sum((Ri_2 - R_2)**2)
    return (xc_2, yc_2), R_2
def circle_equation(points):
    """ Return the center and radius of the circle from three points """
    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):
    """ Returns the list of inliers to a model of a circle from a set of points. The threshol
    inliers = []
    thresh = r//5

    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)
```

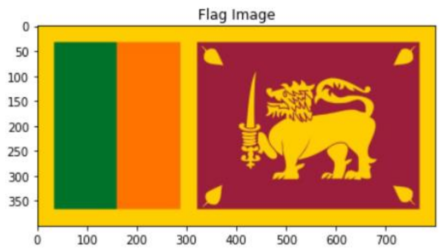
(d) If you fit the circle first, you may miss the line inliers while fitting the circle, which can lead to an inaccurate estimation of the line. The order of fitting can impact the results, so it is generally a good practice to prioritize fitting the dominant model (line in this case) first and then removing its inliers before estimating the secondary model (circle).

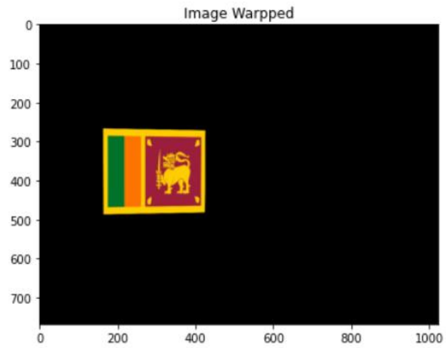
3.

```python
h, status = cv.findHomography(p, p_flag) # Calculating homography between image and flag

# Warping image of flag
warped_img = cv.warpPerspective(image_superimposed, np.linalg.inv(h), (image_background.shape[1],image_background.shape[0]))

blended = cv.addWeighted(image_background, 0.4, warped_img, 0.9, 0.0)
fig, ax = plt.subplots(1,1,figsize= (8,8))
ax.imshow(cv.cvtColor(blended,cv.COLOR_BGR2RGB))
```

The Sri Lankan flag is blended to look like it is hanging on the wall. The 2014 cricket World Cup winning moment is placed to make it look like the match is telecasted on the giant screen. I am using the findHomography function to calculate the homography and warpPerspective to warp the image of the flag.







4. (a)

SIFT Feature Matching

```python
def sift_match(im1, im2):
    GOOD_MATCH_PERCENT = 0.85
    # 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
```

RANSAC

```python
def ransac(matched_points):
    maxInliers = 0
    best_H = None
    for i in range(10):
        random_points = random_sample(matched_points)

        # Generate the homography
        homography = calculateHomography(random_points)
        num_inliers = 0

        # Find the inliers
        for i in range(len(matched_points)):
            d = loss(matched_points[i], homography)
            if d < 3:
                num_inliers += 1

        if num_inliers > maxInliers:
            maxInliers = num_inliers
            best_H = homography
```

SIFT Features Between Images 1 and 5

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.

(b)

Homography Calculation

```
def calculateHomography(correspondences):
    temp_list = []
    for points in correspondences:
        p1 = np.matrix([points.item(0), points.item(1), 1]) # (x1,y1)
        p2 = np.matrix([points.item(2), points.item(3), 1]) # (x2,y2)

        a2 = [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)]
        a1 = [-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)]
        temp_list.append(a1)
        temp_list.append(a2)

    assemble_matrix = np.matrix(temp_list)

    #svd composition
    u, s, v = np.linalg.svd(assemble_matrix)

    #reshape the min singular value into a 3 by 3 matrix
    h = np.reshape(v[8], (3, 3))

    #normalize
    h = (1/h.item(8)) * h
```

Calculated Homography

```
[[ 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]]
```

Provided Homography

```
6.2544644e-01   5.7759174e-02   2.2201217e+02
2.2240536e-01   1.1652147e+00  -2.5605611e+01
4.9212545e-04  -3.6542424e-05   1.0000000e+00
```

Calculated Homography before multiplication

```
[[ 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]]
```

The homography between images was calculated using calculateHomography function. After calculation, it was realized that the calculated homography and the provided homography proved to be highly different compared to the calculated homography before multiplication and the actual homography. Therefore, I calculated homographies for five images separately and obtained the final homography between images one and five by multiplying them sequentially. By comparison of the above final calculated and provided homographies, we can say that the homography is similar.

(c)

Image one and image five were stitched using the generated homography. Image 5 was positioned over the warped image 1. As mentioned above, the best matches between images 1 and 5 were scarce.