EN3160 Assignment 2 on Fitting and Alignment

Name: Himeka S.H.D

Index No:200222K

GitHub Link: [Image_Processing/Assignment_2 at main · DHimeka/Image_Processing (github.com)]

1) Blob detection using Laplacian of Gaussian and Scale Space.

- First the relevant libraries were imported and then loaded the image give in low resolution.
- converted the image to grayscale as Blob detection often takes place for grayscale images.
- Selected the sigma value range to be from 5 to 50.9 with a step of 0.9
- Scale space was stored in a list and then converted to an numpy array

This function iterates through sigma values and filter image with LoG filter and the filtered result is stored in the scale space.

```
for sigma in sigma_values:
    # Calculate the LoG kernel for the current sigma
    kernel_hw = (int(4 * sigma) + 1) // 2
    X, Y = np.meshgrid(np.arange(-kernel_hw, kernel_hw + 1), np.arange(-kernel_hw, kernel_hw + 1))
    LoG = (X ** 2 + Y ** 2 - 2 * sigma ** 2) * np.exp(-(X ** 2 + Y ** 2) / (2 * sigma ** 2))
    # LoG filtering to the grayscale image
    response = cv.filter2D(gray_im.astype(np.float32), -1, LoG)
    # Store the result in the scale space
    scale_space.append(response)
```

Then the local maxima using the scale space numpy array was found using maximum_filter function. A new numpy array to store maximum coordinates were then created..Then iterated through that array and store the detected circle parameters in a new array. Here radius was selected so that it maximizes the strength.

Detected Circles

```
for coordinates in maxima_coordinates:
    z, y, x = coordinates
    # Adjust the scale factor to maximize strength
    radius = int(np.sqrt(2) * sigma_values[z])
    center = (x, y)
    circles_detected.append((center, radius))
```

Next the largest circle of detected circles found using max function and parameters of largest circle were printed. Lastly drew red circles in all circles of detected circles and displayed the final image.

Outputs:

```
Parameters of the Largest Circle:
Center: (359, 172)
Radius: 35
Range of Sigma Values Used: 5.0 to 50.900000000000002
```

2) RANSAC implementation for a line and a circle.
The given code generated noisy points for a circle and line. First it imports relevant libraries. 100 points with 50 for each initialized for both classes. The line (m, b) and circle (r, center) parameters were set. The points are stored in separate data lists and then concatenated.

a) RANSAC line implementation.

First the variables and lists are initialized for the line. The first step of the algorithm, random sample generated using 2 points.

```
def sample_points(X,n):
    indices = np.random.randint(0, N, n) #randomly selecting n indices from 0 to N
    return X_[indices,:] #returning n points from X_ with the selected indices
```

Next we assume a model function to fit the 2 random points. It uses the total least square method, which finds the error minimizing the sum of squared distances of all points from the line.

b) RANSAC Circle estimation

Subtracted the inliers of the best line(remnants) and estimated the circle that fits the other points

```
line_outliers = np.where(l_best_inliers_line==False)[0]
remnants = np.array([X[line_outliers[i]] for i in range(len(line_outliers))])
print('remnants',len(remnants))
```

First the variables and lists are initialized for the circle. The first step of the algorithm, random sample generated using 3 points.

```
def sample_points_circle(N1,n):
    c_indices = np.random.randint(0, N1, n)
    return c_indices
```

Next we assume a model function to fit the 3 random points. It uses the total least squared difference between the actual distance from each data point to the circle's boundary, finds the error minimizing the sum of those distances of all points to the circle.

```
def line_model(x, indices):
    a,b,d = x[0],x[1],x[2] #model parameters
    return np.sum(np.square(a*X_[indices,0] +
b*X_[indices,1] - d)
```

```
def circle_model(x, indices):
    x0, y0, r = x
    x1, y1 = remnants[indices].T
    return np.sum((np.sqrt((x1 - x0)**2 + (y1 - y0)**2)
- r)**2)
```

A function to apply the constraint ‖[a, b]ᵀ‖ = 1.

```
def line_constraint(x):
    # Extract the model parameters
    a, b = x[0], x[1]
    # Apply the constraint ‖[a, b]T‖ = 1
    return np.linalg.norm([a, b]) - 1
cons = ({'type': 'eq', 'fun': line_constraint})
```

A constrain function is not necessarily implemented.

Now the set of inliers (points consistent with the mode) are differentiated from line points by calculating the error and getting points below a threshold

```
def consensus_line(X, x, l_threshold):
    a, b, d = x[0], x[1], x[2] #line model parameters
    error = np.absolute(a*X_[:,0] + b*X_[:,1] - d)
#absolute distance of all points from the line
    return error < l_threshold #returning indices of
points with distance less than threshold
```

Now the set of inliers (points consistent with the mode) are differentiated from circle points by calculating the error distances and getting points below a threshold

```
def consensus_circle(remnants, x, c_threshold):
    distances = np.abs(np.linalg.norm(remnants - x[:2],
axis=1) - x[2])
    return distances < c_threshold
```

The next step is to repeat this process for 100 iterations. Optimized using scipy.optimize package

```
l_iteration = 0
while l_iteration < l_max_iterations:

    indices = np.random.randint(0, N,
l_estimate_data_points)
    x0 = np.array([1, 1, 0])  # Initial estimate
    res = minimize(fun=line_model, args=indices,
x0=x0, tol=1e-6, constraints=cons, options={'disp':
True})

    l_inliers_line = consensus_line(X_, res.x,
l_threshold)

    if l_inliers_line.sum() > l_data_points:
        x0 = res.x
        res = minimize(fun=line_model,
args=l_inliers_line, x0=x0, tol=1e-6,
constraints=cons, options={'disp': True})

        if res.fun < l_best_error:
            print(f'A better model found: (a, b, d) =
{res.x}, Error = {res.fun}')
            l_best_model_line = res.x
            best_error = res.fun
            l_best_sample_line = sample_points(X_,
l_estimate_data_points)
            l_res_only_with_sample = x0
            l_best_inliers_line = l_inliers_line

    l_iteration += 1
```

Lastly the same process repeated for 100 iterations. Optimized using scipy.optimize package

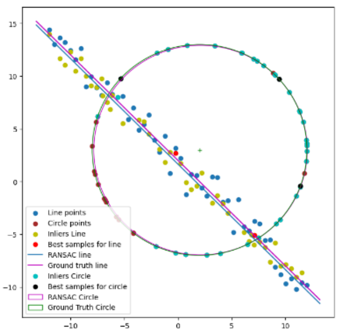
```
c_iteration = 0
while c_iteration < c_max_iterations:
    c_indices = np.random.randint(0,len(remnants),
c_estimate_data_points)
    x0 = np.array([0,0,0]) #initial guess
    res = minimize(circle_model, x0, args=c_indices,
tol=1e-6) #minimize the error
    c_inliers = consensus_circle(remnants, res.x,
c_threshold)  #find the consensus set

    if np.sum(c_inliers) > c_data_points:
            x0 = res.x
            res = minimize(circle_model, x0=x0,
args=(c_inliers),tol= 1e-6)
            if res.fun < c_best_error:
                print(f'A better model found: (x0, y0,
r) = {res.x}, Error = {res.fun}')
                c_best_error = res.fun
                c_best_sample=
sample_points_circle(len(remnants),
c_estimate_data_points)
                c_best_model = res.x
                c_best_inliers = c_inliers

    c_iteration += 1
```

The relevant items are then plotted using matplotlib.

c) Outputs:



d) If we fit the circle first, the 3 random sample points may or may not lie along the line. Ultimately a large circle will be drawn taking drawn along those points. The line points will then be inliers to the circle. So even if we fit the line next, it will still be a line.

3) Warping images using mouse clicks.

1.First the relevant libraries are imported. Then the maximum mouse clicks for the 2 images specified. Next 2 arrays to specify mouse points in each images are initialized.

```
base_points = np.empty((NUM_POINTS, 2))
flag_points = np.empty((NUM_POINTS, 2))
```

2.When the left mouse button is clicked, the **record_mouse_points** function marks the clicked point with a blue circle on an image and records its coordinates in an array, allowing the user to select specific points of interest for homography calculation.

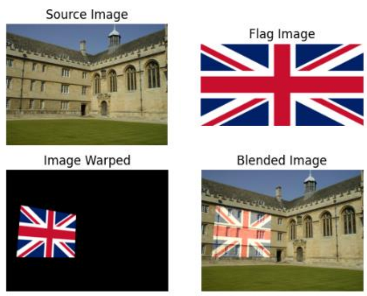
```
def record_mouse_points(event, x, y, flags,
param):
    global point_index
    points, image = param
    if event == cv.EVENT_LBUTTONDOWN and
point_index < NUM_POINTS:
        cv.circle(image, (x, y), 5, (255, 0,
0), -1)
        points[point_index] = (x, y)
        point_index += 1
```

3. After Loading the images, we select to be blended areas from both the images, by calling the record_mouse_points function on a copy of clicking image, until 4 mouse clicks satisfied.

Ex:

```
cv.setMouseCallback('Base Image', record_mouse_points,
(base_points, base_image_copy))

# Collect mouse points for the base image
while True:
    cv.imshow('Base Image', base_image_copy)
    if point_index == NUM_POINTS:
        break
    key = cv.waitKey(20)
    if key & 0xFF == 27:
        break
```



4.

Next we proceed with homography calculations, computing the homography matrix that represents the transformation between two sets of corresponding points in 2 image points.

```
homography, status =
cv.findHomography(base_points, flag_points)
```

5. Then warp the images using warpPerspective function in OpenCV. The transformation according to the homography matrices calculated before, takes place here

```
warped_flag = cv.warpPerspective(flag_image,
np.linalg.inv(homography), (base_image.shape[1],
base_image.shape[0]))
```

Next the images blended to match the brightness and then displayed using matplotlib.

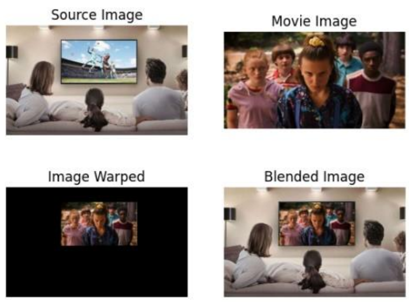
```
alpha = 1
beta = 0.5
blended_image = cv.addWeighted(base_image, alpha, warped_flag,
beta, 0.0)
```

6. If we want to reduce the brightness of area covered by 4 mouse clicks in the base image, we implement a mask

```
# Create a mask for the selected region
mask = np.zeros_like(base_image)
cv.fillConvexPoly(mask,
base_points.astype(int), (1, 1, 1))  # Fill
the selected region with white (1, 1, 1)

# Define a lower intensity for the selected
region
lower_intensity = 1  # Adjust this value as
needed

# Adjust the intensity of the selected region
in the blended image
blended_image = base_image * (1 - mask) +
(warped_movie * mask *
lower_intensity).astype(np.uint8)
```



4) Stitching Images

a) Computing and matching SIFT features
Necessary Libraries and images loaded. First we initiate SIFT detector.

```
sift = cv.SIFT_create()
```

Then we detect SIFT features and compute the SIFT descriptors for both images. Next we use a Brute Force Matcher a function which is used to compare features between 2 images.

```
bf_matcher = cv.BFMatcher()
```

Matching the descriptors take place and k nearest neighbor approach finds out the top 2 matching features of descriptor1

```
matches = bf_matcher.knnMatch(descriptors1,
descriptors5, k=2)
```

Lowe's ratio to filter out ambiguous(false positive) matches. For each matches it checks if it satisfies the threshold level and store the satisfied matches in an array.

```
good_matches = []
for match1, match2 in matches:
    if match1.distance < 0.75 *
match2.distance:
        good_matches.append(match1)
```

Lastly good matches locations are extracted and plotted the matching using drawMatches and matplotlib.

b) Computing Homography Matrix: RANSAC Estimation is used.

First it reads and loads five images and uses the SIFT algorithm to detect key points and compute their descriptors for the first four images. Then making descriptors for each images and matching for among images take place.

Similar pairs are then paired up. Then the homography transformation matrix is calculated
Euclidian distances calculated to check how matching the points are.
Then uses RANSAC algorithm to estimate the transformation. It detect inliers while handling outlier, particularly running the process for 100 iterations. RANSAC do this by estimating a homography matrix. Take the transformation with most inliers

The results are printed and compared.
Computed

```
[[-5.50041603e-01 -1.26774480e-01  2.48483430e+02]
 [-1.24521339e+00 -3.22731467e-01  5.20420491e+02]
 [-2.96157762e-03 -5.12272518e-04  1.00000000e+00]]
```

Original

```
[[.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]]
```

Observed results are not the same. That may be due to real-world challenges such as noise, perspective changes, or imperfect feature matching.

c) Stitching the 2 images.

The homography matrix was converted to a numpy matrix and then warped the 2 images (for both computed homography matrix and given original homography matrix)

```
image_perspective = cv.warpPerspective(img1, original_H, (img5.shape[1], img5.shape[0]))
```

The perspective images were blended and then displayed

```
threshold, mask = cv.threshold(cv.cvtColor(image_perspective, cv.COLOR_BGR2GRAY), 1, 255, cv.THRESH_BINARY)
mask_inv = cv.bitwise_not(mask)
img5_bg = cv.bitwise_and(img5, img5, mask=mask_inv)
dst = cv.addWeighted(img5_bg, 1, image_perspective, 1, 0)
output_image = cv.cvtColor(dst, cv.COLOR_BGR2RGB)
```

The blended output images using both the matrices were almost identical.