EN-2550 Assignment 2

Index - 190521G

Git Hub link :- https://github.com/Lathika-Wathasara/Fundamentals-of-Image-Processing-and-Machine-Vision/tree/master/Assignments

1)

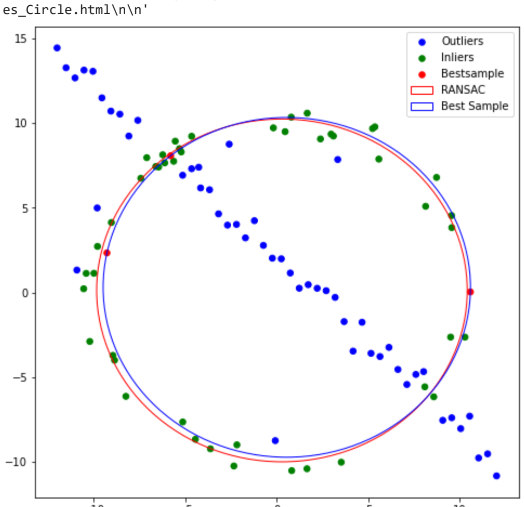
In [ ]:
```
import cv2 as cv
import numpy as np
from scipy import optimize
from scipy . optimize import minimize
from scipy import linalg
import matplotlib . pyplot as plt
import random
```

In [ ]:
```
# np . random . seed ( 0 )
N = 100
half_n = N// 2
r = 10
s = r /16
t = np . random . uniform (0 , 2*np . pi , half_n )
n = s*np . random . randn ( half_n )
x , y = ( r + n)*np . cos ( t ) , ( r + n)*np . sin ( t )
X_circ = np . hstack ( ( x . reshape ( half_n , 1 ) , y . reshape ( half_n , 1 ) ) )
m, b =-1, 2
x = np . linspace (-12, 12 , half_n )
y = m*x + b + s*np . random . randn ( half_n )
X_line = np . hstack ( ( x . reshape ( half_n , 1 ) , y . reshape ( half_n , 1 ) ) )
X = np . vstack ( ( X_circ , X_line ) )
X = np.round(X, 4)
# get circle from 3 points
def get_circle(x1,y1, x2,y2, x3,y3):
    x1y1 =x1**2 +y1**2
    x2y2 =x2**2 +y2**2
    x3y3 =x3**2 +y3**2

    mat = np.array([[x1y1, x1, y1, 1],[x2y2, x2, y2, 1],[x3y3, x3, y3, 1]])
    det_1 = np.round(np.linalg.det(np.hstack((mat[:,1].reshape(3,1),mat[:,2].reshape(3,1),mat[:,3].reshape(3,1)))) ,5)
    det_2 =  np.round(-np.linalg.det(np.hstack((mat[:,0].reshape(3,1),mat[:,2].reshape(3,1),mat[:,3].reshape(3,1)))) ,5)
    det_3 =  np.round(np.linalg.det(np.hstack((mat[:,0].reshape(3,1),mat[:,1].reshape(3,1),mat[:,3].reshape(3,1)))) ,5)
    det_4 =  np.round(-np.linalg.det(np.hstack((mat[:,0].reshape(3,1),mat[:,1].reshape(3,1),mat[:,2].reshape(3,1)))) ,5)

    x_c = (det_2/det_1)/(-2)
    y_c = (det_3/det_1)/(-2)
    r = np.sqrt(x_c**2 + y_c**2 - (det_4/det_1))
    return [x_c, y_c, r]

# get a candidate circle
def candidate_circle(X):
    x,y =X[:,0],X[:,1]
    x_m = np.mean(x)
    y_m = np.mean(y)
    def calc_R(xc, yc):
        #calculate the distance of each 2D points from the center (xc, yc)
        return np.sqrt((x-xc)**2 + (y-yc)**2)
    def f_2(c):
        # calculate the algebraic distance between the data points and the mean circle centered at c=(xc, yc)
        Ri = calc_R(*c)
        return Ri - np.mean(Ri)
    center_estimate = x_m, y_m
    center_2, ier = optimize.leastsq(f_2, center_estimate)
    xc, yc = center_2
    Ri         = calc_R(*center_2)
    R        = np.mean(Ri)

    return ([xc,yc,R])

# ransac circle
def Ransac_Circle(X , threshold, inliners_min_limmit, max_iterations, N, expecting_redius):
    iteretions=0
    inliner_list=[]     # element-->[num of inlinners, error, inliner points list, candidate circle(x0,y0,r), best_sample(point index),best sample  circle(x0,y0,r)]
    while (max_iterations> iteretions):
        a, b, c = [np.random.randint(0,N) for i in range(3)]

        x_c, y_c, r =  get_circle(X[a,0],X[a,1],X[b,0],X[b,1],X[c,0],X[c,1])

        if (np.absolute(r- expecting_redius)> 2*threshold):
            iteretions+=1
            continue
        err_array = np.square(np.sqrt((np.square(X[:,0].reshape(N,1) - x_c))+(np.square(X[:,1].reshape(N,1) - y_c))) - r)
        threshold_sq = threshold**2
        inliners=[]
        for i in range(N):
            if (err_array[i]<= threshold_sq):
                inliners.append(list(X[i]))
        num_inliners= len(inliners)
        if (num_inliners < inliners_min_limmit):
            iteretions+=1
            continue
        #get candidate circle
        x_cc , y_cc, R_c = candidate_circle(np.array(inliners))
        # check new inliner count and calculate error
        inliner_err=0
        inliners =  []
        err_array_with_candidate = np.square(np.sqrt((np.square(X[:,0].reshape(N,1) - x_cc))+(np.square(X[:,1].reshape(N,1) - y_cc))) - R_c)
        for i in range(N):
            if (err_array_with_candidate[i]<= threshold_sq):
                inliners.append(list(X[i]))
                inliner_err += err_array_with_candidate[i]
        num_inliners= len(inliners)
        mean_err = inliner_err/num_inliners
        if (num_inliners < inliners_min_limmit):
            iteretions+=1
            continue
        inliner_list.append([num_inliners, mean_err, inliners, [x_cc,y_cc,R_c], [a,b,c], [x_c, y_c, r ]])
```

```
        iteretions+=1
    return inliner_list

# choose the best match
best_index=0
max_inliners=0
min_err=1000
iterations =100
threshold =1
Min_Inliers_limit = 40
inliner_list = Ransac_Circle(X, threshold, Min_Inliers_limit,iterations,N, r)
for i in range(len(inliner_list)-1):
    if (max_inliners < inliner_list[i][0]):
        max_inliners= inliner_list[i][0]
        min_err = inliner_list[i][1]
        best_index = i
    elif (max_inliners == inliner_list[i][0]) and (min_err > inliner_list[i][1]):
        min_err = inliner_list[i][1]
        best_index = i

# seperate plotting points
inliners = inliner_list[best_index][2]
outliners =[]
for i in range(N):
    if (list(X[i]) not in inliners) :
        outliners.append(list(X[i]))
best_samples =[list(X[inliner_list[best_index][4][0]]), list(X[inliner_list[best_index][4][1]]), list(X[inliner_list[best_index][4][2]])]
for i in range(3):
    inliners.remove(best_samples[i])

#ploting
#ploting the dot diagrams
fig, ax = plt.subplots(figsize=(8,8))

plt.scatter(np.array(outliners)[:,0],np.array(outliners)[:,1],s= np.ones(len(outliners))*30 , color= 'blue', label = 'Outliers')
plt.scatter(np.array(inliners)[:,0],np.array(inliners)[:,1],s= np.ones(len(inliners))*30, color= 'green', label = 'Inliers')
plt.scatter(np.array(best_samples)[:,0],np.array(best_samples)[:,1],s= np.ones(len(best_samples))*30 , color= 'red', label = 'Bestsample')
#ploting circles
x_cc,y_cc,R_c = inliner_list[best_index][3]
RANSAC_circle = plt.Circle((x_cc , y_cc), R_c, fill=False , color = 'red' ,label='RANSAC')
ax.add_artist(RANSAC_circle)

x_c, y_c, r = inliner_list[best_index][5]
Best_Sample_circle = plt.Circle((x_c , y_c), r, fill=False , color = 'blue' ,label='Best Sample')
ax.add_artist(Best_Sample_circle)
plt.legend()
"""Referance
https://sdg002.github.io/ransac-circle/index.html
https://github.com/anubhavparas/ransac-implementation
https://scipy-cookbook.readthedocs.io/items/Least_Squares_Circle.html

"""
```

```
C:\Users\Lathika\AppData\Local\Temp\ipykernel_35432\2956447405.py:34: RuntimeWarning: invalid value encountered in double_scalars
  x_c = (det_2/det_1)/(-2)
C:\Users\Lathika\AppData\Local\Temp\ipykernel_35432\2956447405.py:35: RuntimeWarning: invalid value encountered in double_scalars
  y_c = (det_3/det_1)/(-2)
C:\Users\Lathika\AppData\Local\Temp\ipykernel_35432\2956447405.py:36: RuntimeWarning: invalid value encountered in double_scalars
  r = np.sqrt(x_c**2 + y_c**2 - (det_4/det_1))
```

Out[ ]:
```
'Referance\nhttps://sdg002.github.io/ransac-circle/index.html\nhttps://github.com/anubhavparas/ransac-implementation\nhttps://scipy-cookbook.readthedocs.io/items/Least_Squares_Circle.html\n\n'
```



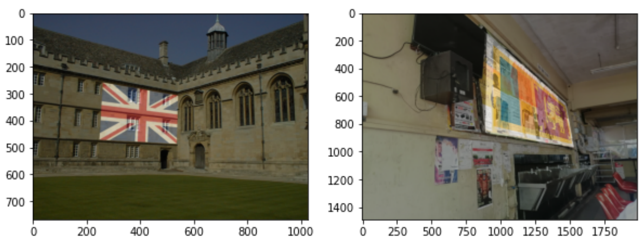
Ransac_Circle function is used to get an list of possible samples which satisfy the minimum inlier requierment. In the retuerning list element, there are 6 components. 1) Num of inliers 2) Average error 3) Inlier points as a list 4) Candidate circle (the circle corresponds to the samples) 5) Best sample (The 3 sample points used to comput the initial circle) 6) Best sample circle (Circle computed by chosen inliers)

After the RANSAC function best matching linliers set is computerd by sorting. Fiest priority is givet to the element with the highest number of inliers. If there are two sets with the same number of inliers, then the second priority is given by the minimum average error. After selecting the best set of inliers, then a avarage circle is computed as "RANSAC_circle" This method can be used to select the best matching points in any kind of data set and do the computations with thoes inliers. It increase the accuracy as well.

2)

In [ ]:
```
#Load images
flag = cv.cvtColor(cv.imread(r'Images\Flag_of_the_United_Kingdom.png'), cv.COLOR_BGR2RGB)
wall = cv.cvtColor(cv.imread(r'Images\002.jpg'), cv.COLOR_BGR2RGB)
flag2 = cv.cvtColor(cv.imread(r'Images\sl_flag.png'), cv.COLOR_BGR2RGB)
wall2 = cv.cvtColor(cv.imread(r'Images\Wall 2.jpeg'), cv.COLOR_BGR2RGB)

# interesting points
```

```python
flag_pts = np.array([[0,0],[383,0],[383,192],[0,192]])
wall_pts = np.array([[261,260],[541,308],[536,487],[248,472]])

flag2_pts = np.array([[0,0],[860,0],[860,470],[0,470]])
wall2_pts = np.array([[902,148],[1472,656],[1530,970],[890,863]])

def get_projection(wall,flag,wall_pts,flag_pts):
    # getting homofraphy
    h, status = cv.findHomography(flag_pts, wall_pts)
    # get transformed image
    transformed_img =  cv.warpPerspective(flag, h,(wall.shape[1],wall.shape[0]))
    #blending images
    blended =cv.addWeighted(wall, 0.6, transformed_img,0.4,0)
    return blended


#reference
"""
https://www.etutorialspoint.com/index.php/319-python-opencv-overlaying-or-blending-two-images
https://learnopencv.com/homography-examples-using-opencv-python-c/
https://theailearner.com/tag/cv2-warpperspective/
https://towardsdatascience.com/image-processing-with-python-applying-homography-for-image-warping-84cd87d2108f

"""
```



Out[ ]:
```
'\nhttps://www.etutorialspoint.com/index.php/319-python-opencv-overlaying-or-blending-two-images\nhttps://learnopencv.com/homography-examples-using-opencv-python-c/\nhttps://theailearner.com/tag/cv2-warpperspective/\nhttps://towardsdatascience.com/image-processing-with-python-applying-homography-for-image-warping-84cd87d2108f\n\n'
```

First the images are read and converted into the RGB format.

wall_pts are the destination points. flag_pts are the cornors of the flag. The homography function is computed using "cv.findHomography" " cv.warpPerspective" is used to transform the the flag into the wall destination points. Finally thoes two images has to be blended. Blending is done by the function "cv.addWeighted"

3)

In [ ]:
```python
img1 = cv.cvtColor(cv.imread(r'graf\img1.ppm'), cv.COLOR_BGR2RGB)
img2 = cv.cvtColor(cv.imread(r'graf\img5.ppm'), cv.COLOR_BGR2RGB)

#geting key points
sift = cv.SIFT_create()
keypoints_1,  desctriptor_1 = sift.detectAndCompute(img1, None)
keypoints_2,  desctriptor_2 = sift.detectAndCompute(img2, None)

bf = cv.BFMatcher(cv.NORM_L1, crossCheck = True)

matches = bf.match(desctriptor_1,desctriptor_2)
matches = sorted(matches, key = lambda x:x.distance)

fig, ax = plt.subplots(figsize =(8,8))
ax.axis('off')
img_matches = cv.drawMatches(img1, keypoints_1, img2, keypoints_2, matches[:50], img2, flags=2)

"""Reference
https://towardsdatascience.com/understanding-homography-a-k-a-perspective-transformation-cacaed5ca17"""
```

Out[ ]:
```
'Reference\nhttps://towardsdatascience.com/understanding-homography-a-k-a-perspective-transformation-cacaed5ca17'
```

b & c

In [ ]:
```python
import random
def find_sift_match(img1,img2):
    sift = cv.SIFT_create()
    keypoints_1, descriptors_1 = sift.detectAndCompute(img1, None)
    keypoints_2, descriptors_2 = sift.detectAndCompute(img2, None)
    bf= cv.BFMatcher(cv.NORM_L1, crossCheck = True)
    matches = bf.match(descriptors_1, descriptors_2)
    sortmatches = sorted(matches, key = lambda x:x.distance)
    return matches,[keypoints_1,keypoints_2]
def SSD(corres, h):
    pts1 = np.transpose(np.matrix([corres[0].item(0), corres[0].item(1), 1]))
    estimatep1 = np.dot(h, pts1)
    estimatep2 = (1/estimatep1.item(2))*estimatep1
    pts2 = np.transpose(np.matrix([corres[0].item(2), corres[0].item(3), 1]))
    error = pts2 - estimatep2
    return np.linalg.norm(error)
def Homography(correspondences):
    Lst = []
    for corr in correspondences:
        p1 = np.matrix([corr.item(0), corr.item(1), 1])
        p2 = np.matrix([corr.item(2), corr.item(3), 1])
        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)]
        Lst.append(a1)
        Lst.append(a2)
    matrixA = np.matrix(Lst)
    #svd composition
    u, s, v = np.linalg.svd(matrixA)
    h = np.reshape(v[8], (3, 3))
    h = (1/h.item(8)) * h
    return h
def ransac(corr, thresh):
    maxInliers = []
    finalH = None
    for i in range(1000):
        #find 4 random points to calculate a homography
        corr1 = corr[random.randrange(0, len(corr))]
        corr2 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((corr1, corr2))
        corr3 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr3))
        corr4 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr4))
        #call the homography function on those points
        h = Homography(randomFour)
        inliers = []
        for i in range(len(corr)):
            d = SSD(corr[i], h)
            if d < 5:
                inliers.append(corr[i])
        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = h
        if len(maxInliers) > (len(corr)*thresh):
            break
    return finalH, maxInliers
def corr_list(matches1,key):
    correspondenceList1 = []
    keypoints1 = [key[0],key[1]]
    for match in matches1:
        (x1, y1) = keypoints1[0][match.queryIdx].pt
        (x2, y2) = keypoints1[1][match.trainIdx].pt
        correspondenceList1.append([x1, y1, x2, y2])
    return correspondenceList1

img1,img2,img3,img4,img5 = cv.imread(r"graf/img1.ppm"), cv.imread(r"graf/img2.ppm") ,cv.imread(r"graf/img3.ppm"),cv.imread(r"graf/img4.ppm"), cv.imread(r"graf/img5.ppm")
img1,img2,img3,img4,img5 = cv.cvtColor(img1, cv.COLOR_BGR2RGB),cv.cvtColor(img2, cv.COLOR_BGR2RGB),cv.cvtColor(img3, cv.COLOR_BGR2RGB),cv.cvtColor(img4, cv.COLOR_BGR2RGB),cv.

#calculate homographies
match1,ky1=find_sift_match(img1,img2)
corrList1=corr_list(match1,ky1)
corrs1 = np.matrix(corrList1)
finalH1, inliers1 = ransac(corrs1, 0.6)
match2,ky2=find_sift_match(img2,img3)
corrList2=corr_list(match2,ky2)
corrs2 = np.matrix(corrList2)
finalH2, inliers2 = ransac(corrs2, 0.6)
match3,ky3=find_sift_match(img3,img4)
corrList3=corr_list(match3,ky3)
corrs3 = np.matrix(corrList3)
finalH3, inliers3 = ransac(corrs3, 0.6)
match4,ky4=find_sift_match(img4,img5)
corrList4=corr_list(match4,ky4)
corrs4 = np.matrix(corrList4)
finalH4, inliers4 = ransac(corrs4, 0.6)
#Obtaining the homography matrix of 1 to 5
H = finalH4 @ finalH3 @ finalH2 @ finalH1
print(H)
dst1 = cv.warpPerspective(img1, H, ((img5.shape[1]), img5.shape[0]))
```

```
[[ 6.28750210e-01 -1.20760305e-02  2.26104441e+02]
 [ 2.39508867e-01  1.08373688e+00 -2.37227721e+01]
 [ 5.40697214e-04 -1.80260338e-04  9.84497854e-01]]
```

original image destination image output image stitched image

First the keypoints and descriptors have being calculated using the "sift.detectAndCompute" function. then the matching points have being calculated by the function "bf.match".
Then the Ransac function is used with to compute the most suitable points which gives a homography with maximum inliers, out of thoes matched points.
By using thoes best matching points, the final homography has being calculated. Finaly by "cv.warpPerspective" function, the original image is changed as it match with the big picture. By using Ransac, we can reduce the chance of geting the wrong homography matrixes.

The original 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]]

The computed Homography

[[6.28750210e-01 -1.20760305e-02 2.26104441e+02]
[2.39508867e-01 1.08373688e+00 -2.37227721e+01]
[5.40697214e-04 -1.80260338e-04 9.84497854e-01]]