
ISTANBUL TECHNICAL UNIVERSITY

Faculty of Computer Science and Informatics

# BLG453E

# Computer Vision Homework-4 Report

Author

Cansu Yanık -150170704

1. **Part 1: Hough Circle Detection - Problem Construction**

Algebraic Approach:

Standard form equation of a circle:

$$(x-a)^2+(y-b)^2=r^2$$

An algebraic representation of a circle on a plane:

$$F(\boldsymbol{x}) = a\boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + c = 0$$

$$\text{where: } \boldsymbol{x}^T = \begin{bmatrix} x & y \end{bmatrix}$$

If we have m data points, (m>0), for unknown coefficients $u = [a, b_x, b_y, c]^T$, a set of linear equation $Bu = 0$ can be built.

$$B = \begin{bmatrix} x_1^2 + y_1^2 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_m^2 + y_m^2 & x_m & y_m & 1 \end{bmatrix}$$

An additional constraint is required if number of points m>3 in order to obtain a result:

$||u|| = 1$. Now, we have:

$$||Bu|| = min \text{ s.t. } ||u|| = 1$$

Solution can be found from right singular vector of matrix B associated with the smallest singular value. Center and radius of the circle can be found with this the algebraic equation given below:

$$\left(x + \frac{b_x}{2a}\right)^2 + \left(y + \frac{b_y}{2a}\right)^2 = \frac{||\boldsymbol{b}||^2}{4a^2} - \frac{c}{a}$$

$$(x_c, y_c) = \left(-\frac{b_x}{2a}, -\frac{b_y}{2a}\right)$$

$$r = \sqrt{\frac{||\boldsymbol{b}||^2}{4a^2} - \frac{c}{a}}$$

**Hough Algorithm:**

The parameterization: center (a, b) and the Radius r

$$x = a + R\cos\theta$$

$$y = b + R\sin\theta$$

When the θ varies from 0 to 360, a complete circle of radius R is generated. With the Circle Hough Transform, three parameters (x, y, R) should be found. Thus, the parameter space is 3D. To find the intersection point in the parameter space, an accumulator matrix is needed. Since the parameter space is 3D, also the accumulator matrix would be 3D. For each "edge" point in the original space, we can create a circle in the parameter space and increase the voting number in accumulator. After voting, the local maxima will be circle centers in the original space.

Assumption: Radius is unknown. For each radius, we apply the algorithm. Accumulator array should be A[a,b,r] in the 3D space. Every point in the xy space will be equivalent to a circle in the ab space.

$$a = x1 - R\cos\theta$$

$$b = y1 - R\sin\theta$$

**The algorithm :**

1. Load an image
2. Convert the image to grayscale
3. Apply filtering algorithm on image Gaussian Blurring
4. Apply Canny operator to obtain edges on image
5. For every edge pixel, generate a circle in the ab space
6. For every point on the circle in the ab space, cast votes in the accumulator cells
7. The local maximum voted circles of Accumulator A gives the circle Hough space
8. The maximum voted circle of Accumulator gives the circle

```
1. For each edge pixel(x,y)in the image
2.    For each radius r = 0 to r = 30 // the possible radius
3.      For each theta t = 0 to 360  // the possible  theta 0 to 360
4.         a = x - r * cos(t * PI / 180); //polar coordinate for
center
5.         b = y - r * sin(t * PI / 180);  //polar coordinate for
center
6.         A[a,b,r] +=1; //voting
7.      end
8.    end
9.   end
```

2. **Part 2: Hough Circle Detection - Implementation**

In this part, I was trying to implement the algorithm that was given in part1, but I could not give a good result. But I wanted to add the code I wrote.  For this reason, I used prepared Hough methods from cv2 module.

Figure-1 shows the Hough method. As I said, I used cv2's Hough functions. Figure-2 shoes the output.

```
import cv2
from math import sqrt, pi, cos, sin
import numpy as np

image = cv2.imread("./Material/marker.png")
row, col, ch = image.shape


#Apply Gaussian Blur
blur = cv2.GaussianBlur(image,(5,5),0)
#Convert image to grayscale
blurgray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)


d_length = int(((row**2 + col **2)**(1/2))/4)

#Apply Hough method
circles = cv2.HoughCircles(blurgray, cv2.HOUGH_GRADIENT, 1, 20, param1=50, param2=50,
                           minRadius=1, maxRadius=d_length)

#Displays circles
circles = np.uint16(np.around(circles))
for i in circles[0, :]:
    center = (i[0], i[1])
    # circle center
    cv2.circle(image, center, 2, (0, 0, 255), 3)
    # circle outline
    radius = i[2]
    cv2.circle(image, center, radius, (0, 255, 0), 3)


cv2.imshow("detected circles", image)
cv2.waitKey(0)
```

**Figure-1: Applying Hough Method**

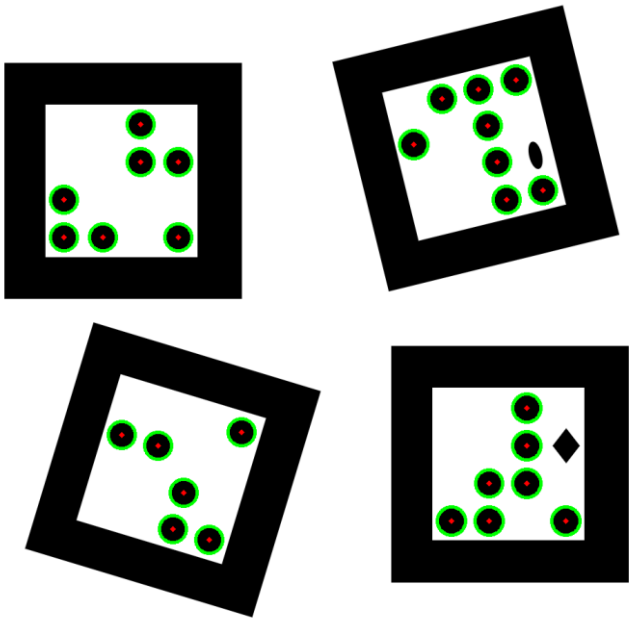

**Figure-2: Output of the Hough method**

Also, I was trying to write algorithm on my own. Firstly, I loaded image, applied Gaussian blur then converted image to grayscale. I applied Canny Operator to grayscale image.

```
import cv2
from math import sqrt, pi, cos, sin
import numpy as np


image = cv2.imread("./Material/marker.png")
row, col, ch = image.shape


#Apply Gaussian Blur
blur = cv2.GaussianBlur(image,(5,5),0)
#Convert image to grayscale
blurgray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
```

```
#Find thresholds
max_treshold,image1 = cv2.threshold(blurgray,0,255,cv2.THRESH_BINARY + cv2.THRESH_OTSU)
low_treshold = max_treshold/3
#Apply canny operator
image2 = cv2.Canny(image1,low_treshold,max_treshold)
```

```
#Create accumulator
A = np.zeros((row+10,col+10,30), dtype = np.float32)

for x in range(row):
    for y in range(col):
        if(image2[x,y] > 0):
            for r in range(20, 30):
                for t in range(360):
                    a = int(x-r * cos(t * (pi/180)))
                    b = int(y-r * sin(t * (pi/180)))
                    A[a][b][r] += 1


image3 = cv2.imread("./Material/marker.png")
for a in range(col):
    for b in range(row):
        for r in range(20, 30):
            if (A[b][a][r] > 150):
                cv2.circle(image3, (a,b), 2, (0, 0, 255), 3)
                cv2.circle(image3, (a,b), r, (0, 255, 0), 3)


cv2.imshow("out", image3)
cv2.imwrite("x.jpg", image3)
cv2.waitKey(0)
```

**Figure-4: Hough Operation**

For every pixel in image, I found white pixels then I calculated polar coordinates and increased the votes of that values.

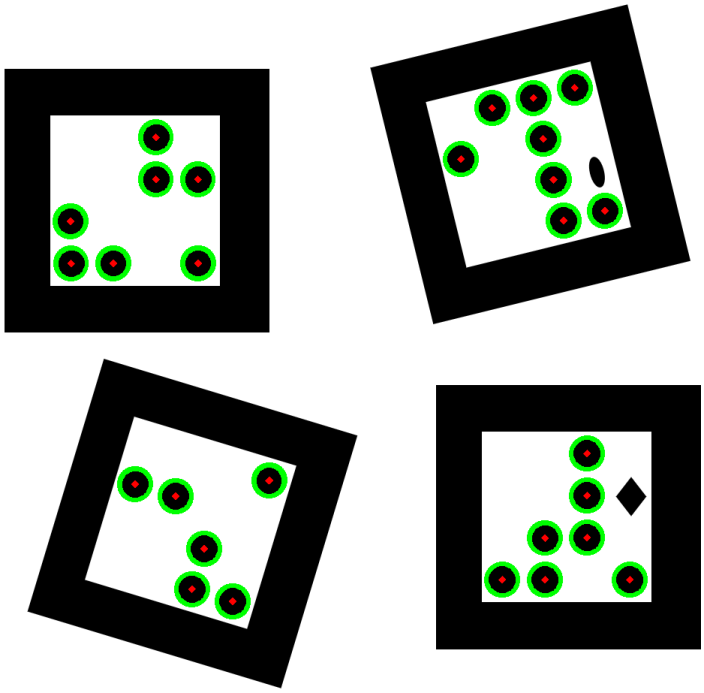

**Figure-4: Output of my Hough operation**

## 3. Part 3: 3D Image Segmentation

For a and b parts, I wrote a function called *regionGrowing(img, seed, neighborhood)*. It takes three parameters which are image, seed point and the neighborhood number. If I send 4 as neighborhood parameter, function will use 4-neighborhood method.

```
import nibabel as nib
import numpy as np


threshold = 10

def regionGrowing(img, seed, neighborhood):

    row, col = img.shape
    size = row * col

    #If 4 neighborhood, uses corresponding points
    if (neighborhood == 4):
        neighborPts = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        steps = 4
    elif (neighborhood == 8):
        neighborPts = [(1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1)]
        steps = 8

    #result image
    segmentedImg = np.zeros((row, col, 1), np.uint8)
    regionSize = 1

    #neighbors points
    neighbors_x = []
    neighbors_y = []
    neighborIntensities = []

    #seed intensity
    mean = img[seed]
    #intensity difference
    difference=0
```

**Figure-5: regionGrowing Function**

```
while (regionSize < size):

    if(difference > threshold):
        break

    for i in range(steps):
        #Finds neighbor points
        x = neighborPts[i][0] + seed[0]
        y = neighborPts[i][1] + seed[1]

        #If point is in the image (checks image borders)
        if ((x >= 0) and (x < row) and (y >= 0) and (y < col)):
                #Prevent looking previous point
                if segmentedImg[x, y] == 0:
                    neighbors_x.append(x)
                    neighbors_y.append(y)
                    neighborIntensities.append(img[x, y])
                    segmentedImg[x, y] = 255

    #Find min distance between intensities
    minDistance = abs(neighborIntensities[0]-mean)
    i=1
    index=0
    for i in range(len(neighborIntensities)):
        distance = abs(neighborIntensities[i] - mean)
        #Finds that point index
        if distance < minDistance:
            minDistance = distance
            index = i

    difference = minDistance

    #Finds new mean
    total = mean*regionSize + neighborIntensities[index]
    mean = total/regionSize

    #Finds next seed point
    seed = [neighbors_x[index], neighbors_y[index]]
    #Remove that point from array
    neighborIntensities[index] = neighborIntensities[-1]
    neighbors_x[index] = neighbors_x[-1]
    neighbors_y[index] = neighbors_y[-1]

    regionSize = regionSize+1

return segmentedImg
```

**Figure-6: regionGrowing Function**

The code in figure-6 shows the process. For each slice of image, I found a white pixel. Therefore, If I cannot find a white pixel, I won't call the region growing function and that process will not be performed for black images. Then I calculated a dice score. I only calculated it for one image with z_index=46. But I'm not sure if I have correctly calculated the dice score.

Result (8- neighborhood): "Dice similarity score is 1.1557320131359372"

Result (4- neighborhood): "Dice similarity score is 0.5497747290347854"

Figure-7 shows an example segmentation results.

Same process is applied for 4-neighborhood.

```
import cv2
import nibabel as nib
from regionGrowing import regionGrowing
import numpy as np


img = nib.load('./Material/V_seg_05.nii')


row, col, z_index = img.shape


found = False

#For each slice find a white pixel
for z in range(z_index):
    for i in range(row):
        for j in range(col):
            if img.get_fdata()[i,j,z] > 0:
                if found==False:
                    found = True
                    seed = (i,j)

    #If white pixel is found, calls regiongrowing function, and saves ima
    if(found == True):
        output = regionGrowing(img.get_fdata()[:,:,z], seed, 8)
        found = False
        print(str(z) + " Processing...")
        #cv2.imwrite("./N-8_Segmentations/8N_"+str(z)+".jpg", output)

    if (z == 46):
        seg = output


gt = img.get_fdata()[:,:,46]
dice = np.sum(seg[gt==1])*2.0 / (np.sum(seg) + np.sum(gt))
print ('Dice similarity score is {}'.format(dice))
```

```
##### 4-neighborhood #####
found = False

for z in range(z_index):
    for i in range(row):
        for j in range(col):
            if img.get_fdata()[i,j,z] > 0:
                if found==False:
                    found = True
                    seed = (i,j)


    if(found == True):
        output = regionGrowing(img.get_fdata()[:,:,z], seed, 4)
        found = False
        print(str(z) + " Processing...")
        #cv2.imwrite("./N-4_Segmentations/4N_"+str(z)+".jpg", output)

    if (z == 46):
        seg2 = output


gt2 = img.get_fdata()[:,:,46]
dice2 = np.sum(seg2[gt2==1])*2.0 / (np.sum(seg2) + np.sum(gt2))
print ('Dice similarity score is {}'.format(dice2))
```

**Figure-7: For every slice, calls region growing function**

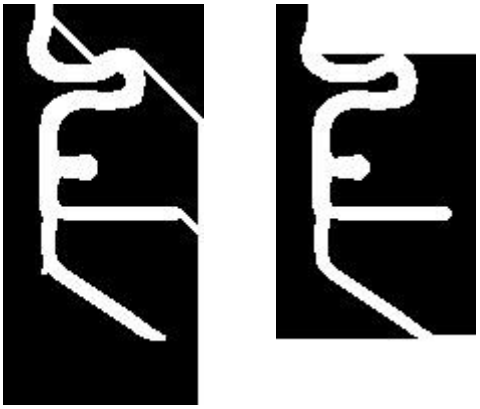

**Figure-8: example segmentation result (z = 46) (8- neighborhood and 4- neighborhood)**

**Figure-9: example segmentation result (z = 49) (8- neighborhood)**

**c)** In my code, I traverse all points in the image and search for a white pixel. When I find a white pixel, I select it as seed point. But actually, to be able to gain better segmentation result, a more proper seed point can be selected. Because seed intensity and the position are important to get a reliable result. For this reason, these should be considered. Also, more than one seed points can be selected. But instead of selecting seed points manually, they should be selected automatically. And also borders of objects or the object intensities can be made more apparent. thus, to separate objects from the background will be easy and intensity difference between them will be high. In my opinion, these preprocessing operations will increase dice score.