|  |
| --- |
| **Morphology and Segmentation** |

**Dhayanidhi Gunasekaran**

University at Buffalo

*dhayanid@buffalo.edu*

**Abstract**

To perform the Image Morphology, Segmentation and Hough transform for the given set of input images.

**1 Image Morphology**

To remove the noise using two morphology image processing algorithms and to extract the boundary of the noise reduced image.

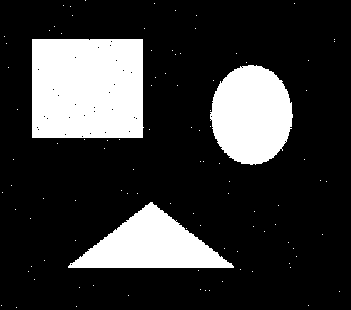


Figure 1: Input Image for Morphology

* 1. **Task Objective**

To reduce the noise using the image morphology algorithm and to extract the boundary.

1. Opening + Closing
2. Closing + Opening

**Opening**



**Closing**



**Dilation**

****

**Erosion**



**Boundary Extraction**



**Kernel**

[[0 1 0]

[1 1 1]

[0 1 0]]

**1.2 Procedure**

The implementation is done using python, opencv and numpy libraries.

1. Images are read using imread function.
2. Dilation and Erosion function are implemented.
3. Noise reduction algorithm mentioned above are implemented
4. Boundaries are extracted using the boundary detection algorithm mentioned.
5. Output image is saved.

**1.3 Code**

import cv2

import numpy as np

# funtion to apply eclipse kernel for an image

def multiply\_kernel(matrix, row, col):

matrix[row,col] = 255

if row-1 >=0:

matrix[row-1, col]=255

if row+1 < matrix.shape[0] :

matrix[row+1,col] = 255

if col-1 >= 0:

matrix[row,col-1] = 255

if col+1 < matrix.shape[1]:

matrix[row,col+1] = 255

return matrix

def dilation(image,kernel):

row = image.shape[0]

col = image.shape[1]

output = np.copy(image)

for i in range(row):

for j in range(col):

if image[i, j] == 255:

output = multiply\_kernel(output,i,j)

return output

img = cv2.imread('original\_imgs/noise.jpg',0)

#kernel = np.ones((3,3),np.uint8)

kernel = [[0,1,0],[1,1,1],[0,1,0]]

kernel =np.array(kernel)

#dilation = dilation(img,kernel)

def invertImage(img):

row = img.shape[0]

col = img.shape[1]

output = np.zeros((row,col))

for i in range(row):

for j in range(col):

if img[i,j] == 255:

output[i,j] = 0

elif img[i,j] == 0:

output[i,j] = 255

return output

def erosion(img,kernel):

invertedimg = invertImage(img)

dilated = dilation(invertedimg,kernel)

output = invertImage(dilated)

return output

eroded\_img = erosion(img,kernel)

cv2.imwrite("erosion\_self.jpg",eroded\_img)

def opening(img,kernel):

return dilation(erosion(img,kernel),kernel)

def closing(img,kernel):

return erosion(dilation(img,kernel),kernel)

def boundary(img,kernel):

return img - erosion(img,kernel)

res\_noise1 = opening(img,kernel)

res\_noise1 = closing(res\_noise1,kernel)

cv2.imwrite("output/res\_noise1.jpg",res\_noise1)

res\_noise2 = closing(img,kernel)

res\_noise2 = opening(res\_noise2,kernel)

cv2.imwrite("output/res\_noise2.jpg",res\_noise2)

bound1 = boundary(res\_noise1,kernel)

cv2.imwrite("output/res\_bound1.jpg",bound1)

bound2 = boundary(res\_noise2,kernel)

cv2.imwrite("output/res\_bound2.jpg",bound2)

**1.4 Output Images**

**1.4.1 Noise reduced Image**

|  |  |
| --- | --- |
|  |  |

Figure 2: Noise reduced images

**1.4.2 Comparing two noise reduced images**

When comparing the two noise reduced images there are 99 pixels which differ from each other.

**1.4.3 Boundary Extraction**

|  |  |
| --- | --- |
|  |  |

Figure 3: Boundary images

**2 Image Segmentation**

Image segmentation is a technique which masks the input image with a kernel and a threshold is determined by plotting the histogram. Based on the threshold, the point or segment is detected.

**2.1 Task Objective**

To mask the input image with a kernel and threshold is determined to separate the point or segment.

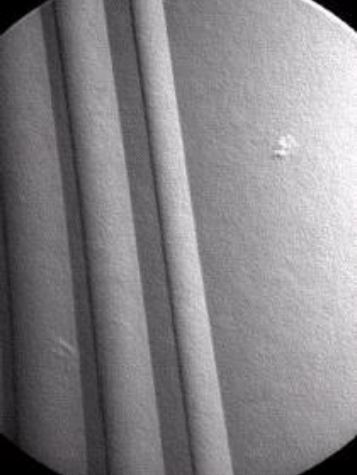


Figure 4: Input image for point detection



Figure 5: Input image for segmentation

**Kernel Used**

**[[-1 -1 -1]**

**[-1 8 -1]**

**[-1 -1 -1]]**

**2.2 Code**

# coding: utf-8

# In[1]:

import cv2

import numpy as np

from matplotlib import pyplot as plt

from collections import Counter

#this function returns the max value for the given matrix

def get\_Max(matrix):

largest\_num = matrix[0][0]

for row\_idx, row in enumerate(matrix):

for col\_idx, num in enumerate(row):

if num > largest\_num:

largest\_num = num

return largest\_num

#This function normalises the image matrix to 0-255 scale

def Normalise\_Matrix(Matrix):

row = len(Matrix)

col = len(Matrix[0])

MAX\_VALUE = get\_Max(Matrix)

for i in range(row):

for j in range(col):

Matrix[i][j] = (Matrix[i][j]/MAX\_VALUE)\*255

return Matrix

#This function generates a matrix with given row,col size filled with zeros

def initialise\_matrix(row, col):

matrix = [[0 for x in range(col)] for y in range(row)]

return matrix

# funtion to get 3\*3 matrix based on its position

def get\_3\_cross3(matrix, row, col):

MAT = initialise\_matrix(3, 3)

if row == 0 or col == 0:

MAT[0][0] = 0

else:

MAT[0][0] = matrix[row-1][col-1]

if row == 0:

MAT[0][1] = 0

else:

MAT[0][1] = matrix[row-1][col]

if row == 0 or col == len(matrix[0])-1:

MAT[0][2] = 0

else:

MAT[0][2] = matrix[row-1][col+1]

if col == 0:

MAT[1][0] = 0

else:

MAT[1][0] = matrix[row][col-1]

MAT[1][1] = matrix[row][col]

if col == len(matrix[0])-1:

MAT[1][2] = 0

else:

MAT[1][2] = matrix[row][col+1]

if row == len(matrix)-1 or col == 0:

MAT[2][0] = 0

else:

MAT[2][0] = matrix[row+1][col-1]

if row == len(matrix)-1:

MAT[2][1] = 0

else:

MAT[2][1] = matrix[row+1][col]

if row == len(matrix)-1 or col == len(matrix[0])-1:

MAT[2][2] = 0

else:

MAT[2][2] = matrix[row+1][col+1]

return MAT

#this function pads 0 for the edge rows and cols

def generatePatchMatrix(matrix, row, col):

PATCH\_MAT = initialise\_matrix(5, 5)

row\_i = row - 2

for i in range(5):

col\_i = col - 2

for j in range(5):

if row\_i < 0 or row\_i > len(matrix)-1 or col\_i < 0 or col\_i > len(matrix[0])-1:

PATCH\_MAT[i][j] = 0

else:

PATCH\_MAT[i][j] = matrix[row\_i][col\_i]

col\_i = col\_i + 1

if i > 4:

row\_i = row - 2

else:

row\_i = row\_i + 1

return PATCH\_MAT

# this function does element wise multiplication of the given 2 matrices

def elem\_wise\_operation(kernel, pos,size):

op = initialise\_matrix(size, size)

op = np.multiply(kernel,pos)

return op

# this function returns the sum of all the values in a matrix

def sum\_of\_elems(MAT):

value = 0

row = len(MAT)

col = len(MAT[0])

for i in range(row):

for j in range(col):

value = value + MAT[i][j]

return value

def mask\_Input\_Image(image):

op\_mat = initialise\_matrix(len(image), len(image[0]))

for i in range(len(image)):

for j in range(len(image[0])):

pos\_mat = get\_3\_cross3(image, i, j)

computed\_mat = np.multiply(pos\_mat, kernel)

op\_mat[i][j] = sum\_of\_elems(computed\_mat)

return op\_mat

# In[290]:

#Color\_image = cv2.imread('original\_imgs/point.jpg')

img = cv2.imread('original\_imgs/turbine-blade.jpg', cv2.IMREAD\_GRAYSCALE)

#img = cv2.imread('original\_imgs/point.jpg',0)

kernel = [[-1, -1, -1],[-1, 8, -1],[-1, -1, -1]]

kernel = np.array(kernel)

print(kernel)

print(img.shape)

mask\_img = mask\_Input\_Image(img)

cv2.imwrite("masked\_image.jpg", np.asarray(mask\_img))

print("mask image generated")

histogram\_img = np.copy(Normalise\_Matrix(mask\_img))

arr=[]

for i in range(histogram\_img.shape[0]):

for j in range(histogram\_img.shape[1]):

if histogram\_img[i,j]!=0:

arr.append(histogram\_img[i,j])

C = Counter(arr)

x, y = list(C.keys()), list(C.values())

plt.bar(x, y)

plt.show()

#plt.hist(np.asarray(mask\_img).ravel(),256,[0,256])

#plt.show()

# In[291]:

Color\_image = cv2.imread('original\_imgs/turbine-blade.jpg')

output\_i = np.copy(histogram\_img) \* 0.

for i in range(histogram\_img.shape[0]):

for j in range(histogram\_img.shape[1]):

if np.abs(histogram\_img[i,j]) > 254 :

output\_i[i,j] = 255.

print(str(i)+","+str(j))

cv2.circle(Color\_image, (445,249), 15, (0,255,0), 3)

cv2.imwrite("output/point\_detect.jpg",Color\_image)

# In[287]:

original\_seg = cv2.imread('original\_imgs/segment.jpg')

seg\_img = cv2.imread('original\_imgs/segment.jpg', cv2.IMREAD\_GRAYSCALE)

hist\_img = np.copy(seg\_img)

array=[]

for i in range(hist\_img.shape[0]):

for j in range(hist\_img.shape[1]):

if hist\_img[i,j]!=0:

array.append(hist\_img[i,j])

ounter = Counter(array)

x, y = list(ounter.keys()), list(ounter.values())

plt.bar(x, y)

plt.show()

# In[288]:

output\_img = np.copy(seg\_img) \* 1.

for i in range(seg\_img.shape[0]):

for j in range(seg\_img.shape[1]):

if seg\_img[i,j] < 203: #170.0 and seg\_img[i,j]<190.0:

output\_img[i,j] = 0.

# In[292]:

cv2.rectangle(original\_seg,(160,125),(210,168),(0,255,0),2)

cv2.rectangle(original\_seg,(250,210),(305,76),(0,255,0),2)

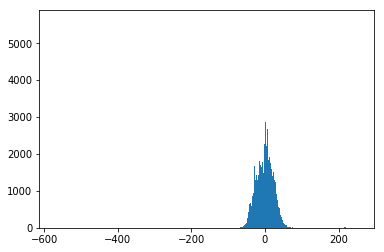
cv2.rectangle(original\_seg,(330,285),(363,20),(0,255,0),2)

cv2.rectangle(original\_seg,(388,255),(425,38),(0,255,0),2)

cv2.imwrite("output/segment\_detect.jpg",original\_seg)

**2.3 Output**

**2.3.1 Histrograms**

****Figure 6: Histogram for Point Detection

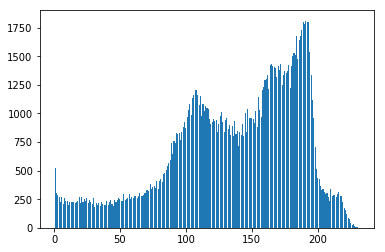
****

Figure 7: Histogram for Segment detection

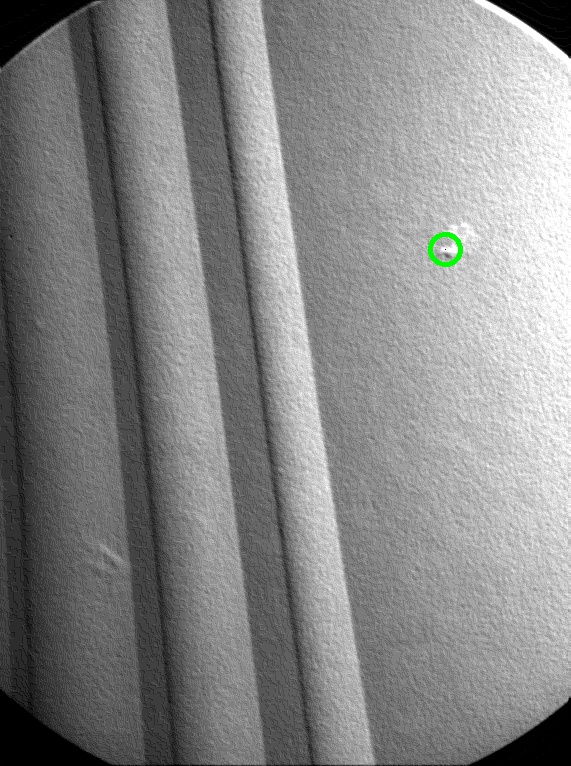
****

Figure 8: Point detection output

Coordinates of the detected point is (445,249)

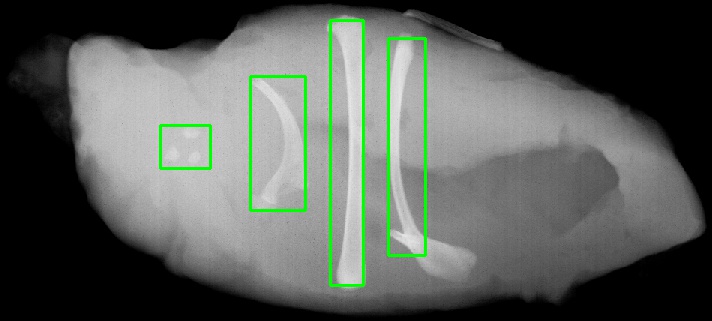


Figure 9: Segment detection output

The coordinates of the bounding boxes from left to right are as follows.

1. (125,160), (125,210), (168,160), (168,210)
2. (76,250), (76,305), (210,250), (210,305)
3. (20,330), (20,363), (285,330), (285,363)
4. (38,388), (38,425), (255,388), (255,425)

**3 Hough Transform**

**3.1 Task Objective**

Use Hough transform to detect the lines in the given image.

To get a red lines, blue lines and coins separately using Hough Transform.

1. Convert the image to HSV format and filter the respective lines using the color range.
2. Use the color filtered image and apply edge detection to get the edges.
3. Hough transform is applied for the edge detected image and the output is determined.
4. The same procedure is applied for Blue line and coins as well.

**3.2 Code**

# coding: utf-8

# In[1]:

import cv2

import numpy as np

from matplotlib import pyplot as plt

# In[2]:

def get\_Max(matrix):

largest\_num = matrix[0][0]

for row\_idx, row in enumerate(matrix):

for col\_idx, num in enumerate(row):

if num > largest\_num:

largest\_num = num

return largest\_num

def Normalise\_Matrix(Matrix):

row = len(Matrix)

col = len(Matrix[0])

MAX\_VALUE = get\_Max(Matrix)

for i in range(row):

for j in range(col):

Matrix[i][j] = (Matrix[i][j]/MAX\_VALUE)\*255

return Matrix

def initialise\_matrix(row, col):

matrix = [[0 for x in range(col)] for y in range(row)]

return matrix

def elem\_wise\_multiple(MAT\_A, MAT\_B, row, col):

MAT = initialise\_matrix(3, 3)

for i in range(row):

for j in range(col):

MAT[i][j] = MAT\_A[i][j] \* MAT\_B[i][j]

return MAT

def sum\_of\_elems(MAT):

value = 0

row = len(MAT)

col = len(MAT[0])

for i in range(row):

for j in range(col):

value = value + MAT[i][j]

return value

# funtion to get 3\*3 matrix based on its position

def get\_3\_cross3(matrix, row, col):

MAT = initialise\_matrix(3, 3)

if row == 0 or col == 0:

MAT[0][0] = 0

else:

MAT[0][0] = matrix[row-1][col-1]

if row == 0:

MAT[0][1] = 0

else:

MAT[0][1] = matrix[row-1][col]

if row == 0 or col == len(matrix[0])-1:

MAT[0][2] = 0

else:

MAT[0][2] = matrix[row-1][col+1]

if col == 0:

MAT[1][0] = 0

else:

MAT[1][0] = matrix[row][col-1]

MAT[1][1] = matrix[row][col]

if col == len(matrix[0])-1:

MAT[1][2] = 0

else:

MAT[1][2] = matrix[row][col+1]

if row == len(matrix)-1 or col == 0:

MAT[2][0] = 0

else:

MAT[2][0] = matrix[row+1][col-1]

if row == len(matrix)-1:

MAT[2][1] = 0

else:

MAT[2][1] = matrix[row+1][col]

if row == len(matrix)-1 or col == len(matrix[0])-1:

MAT[2][2] = 0

else:

MAT[2][2] = matrix[row+1][col+1]

return MAT

# In[3]:

def detectEdges(image):

INPUT\_IMAGE = cv2.cvtColor(image,cv2.COLOR\_BGR2GRAY)

# cv.namedWindow("Input Image")

# cv.imshow('Image',inputim)

# cv.waitKey(0)

print(INPUT\_IMAGE.shape)

# Initialise the Gx and Gy matrix

Gx = [[1, 0, -1],

[2, 0, -2],

[1, 0, -1]]

Gy = [[1, 2, 1],

[0, 0, 0],

[-1, -2, -1]]

rows = len(INPUT\_IMAGE)

cols = len(INPUT\_IMAGE[0])

INPUTXEDGE = initialise\_matrix(rows, cols)

NUM\_MAT = initialise\_matrix(rows, cols)

INPUTYEDGE = initialise\_matrix(rows, cols)

# loop through the matrices and calculate Gx \* Input

for i in range(rows):

for j in range(cols):

THREE\_CROSS\_THREE = get\_3\_cross3(INPUT\_IMAGE, i, j)

OUTPUT = elem\_wise\_multiple(Gx, THREE\_CROSS\_THREE, 3, 3)

INPUTXEDGE[i][j] = sum\_of\_elems(OUTPUT)

for i in range(rows):

for j in range(cols):

THREE\_CROSS\_THREE = get\_3\_cross3(INPUT\_IMAGE, i, j)

OUTPUT = elem\_wise\_multiple(Gy, THREE\_CROSS\_THREE, 3, 3)

INPUTYEDGE[i][j] = sum\_of\_elems(OUTPUT)

OP = initialise\_matrix(rows, cols)

for i in range(rows):

for j in range(cols):

OP[i][j] = (((INPUTXEDGE[i][j]\*\*2)+(INPUTYEDGE[i][j]\*\*2))\*\*0.5)

INPUTXEDGE = Normalise\_Matrix(INPUTXEDGE)

INPUTYEDGE = Normalise\_Matrix(INPUTYEDGE)

OP = Normalise\_Matrix(OP)

INPUTXEDGE = np.asarray(INPUTXEDGE)

INPUTYEDGE = np.asarray(INPUTYEDGE)

OP = np.asarray(OP)

print("edge detected")

return OP

# In[4]:

def hough\_line(img):

# Rho and Theta ranges

thetas = np.deg2rad(np.arange(-90.0, 90.0))

width, height = img.shape

diag\_len = np.ceil(np.sqrt(img.shape[0] \* img.shape[0] + img.shape[1] \* img.shape[1])) # max\_dist

# Hough accumulator array of theta vs rho

accumulator = np.zeros((2 \* int(diag\_len), len(thetas)))

rhos=[]

for i in range(img.shape[0]):

for j in range(img.shape[1]):

if img[i,j]!= 0.:

for x in range(len(thetas)):

# Calculate rho. diag\_len is added for a positive index

rho = round(i \* np.cos(thetas[x]) + j \* np.sin(thetas[x]))+diag\_len

rhos.append(rho)

accumulator[int(rho), int(x)] += 1

return accumulator, thetas, rhos

# In[75]:

def minimise\_no\_rhos(rho,theta):

rho\_op =[]

theta\_op = []

count= 0

array = []

for x in rho:

count+=1

if len(rho\_op) == 0:

rho\_op.append(x)

else:

if (x - rho[count-1]) < 10:

array.append(x)

else:

rho\_op.append(np.median(array))

theta\_op.append(theta[count-1])

array = []

return rho\_op,theta\_op

# In[6]:

img = cv2.imread('original\_imgs/hough.jpg')

hsv\_img = cv2.cvtColor(img,cv2.COLOR\_BGR2HSV)

lower\_red = np.array([170,110,110])

upper\_red = np.array([180,150,150])

mask1 = cv2.inRange(hsv\_img, lower\_red, upper\_red)

output\_hsv = hsv\_img.copy()

output\_hsv[np.where(mask1==0)] = 0

cv2.imwrite("onlyred.jpg",output\_hsv)

# In[7]:

edges = detectEdges(output\_hsv)

cv2.imwrite("red\_edge.jpg",edges)

# In[76]:

accumulators,thetas,rhos = hough\_line(edges)

print(accumulators.shape)

# code to get the indices which are above the range

idx = np.argmax(accumulators)

print(idx)

print(accumulators[int(idx/accumulators.shape[1]),int(idx%accumulators.shape[1])])

mask = [accumulators > 150.] [0] \* 1.

accum = accumulators \* mask

rho = []

theta = []

for i in range(mask.shape[0]):

for j in range(mask.shape[1]):

if mask[i,j]==1:

rho.append(i)

theta.append(thetas[j])

# In[77]:

print(rho)

print(theta)

# In[79]:

r,t= minimise\_no\_rhos(rho,theta)

# In[33]:

rho = [257,258,352,353,445,446,541,542,636,637,734,733]

theta = [-1.53588974175501 ,-1.53588974175501,-1.53588974175501, -1.53588974175501,-1.53588974175501,-1.53588974175501]

#gray\_img = cv2.imread('original\_imgs/hough.jpg', cv2.IMREAD\_GRAYSCALE)

dup = np.copy(edges)

#gs = gray\_img.shape

diag\_len = np.ceil(np.sqrt(dup.shape[0] \*dup.shape[0]+ dup.shape[1] \* dup.shape[1]))

indices = []

for i in range(dup.shape[0]):

for j in range(dup.shape[1]):

for t in theta:

rho\_dup = round(i \* np.cos(t) + j \* np.sin(t))+diag\_len

#print(rho\_dup, rho)

for r in rho:

if r == rho\_dup:

indices.append([i, j])

# In[34]:

gray\_img = cv2.imread('original\_imgs/hough.jpg')

output = np.zeros((gray\_img.shape[0],gray\_img.shape[1]))

for i in indices:

gray\_img[i[0],i[1]] = (0,255,0)

cv2.imwrite("output/red\_line.jpg",gray\_img)

# In[49]:

img\_b = cv2.imread('original\_imgs/hough.jpg')

hsv\_img\_b = cv2.cvtColor(img\_b,cv2.COLOR\_BGR2HSV)

lower\_blue = np.array([95,60,110],np.uint8)

upper\_blue = np.array([110,140,150],np.uint8)

mask1 = cv2.inRange(hsv\_img\_b, lower\_blue, upper\_blue)

output\_hsv\_b = hsv\_img\_b.copy()

output\_hsv\_b[np.where(mask1==0)] = 0

cv2.imwrite("onlyblue.jpg",output\_hsv\_b)

# In[50]:

edges\_b = detectEdges(output\_hsv\_b)

cv2.imwrite("blue\_edge.jpg",edges\_b)

# In[62]:

accumulators\_b,thetas\_b,rhos\_b = hough\_line(edges\_b)

print(accumulators\_b.shape)

# code to get the indices which are above the range

idx\_b = np.argmax(accumulators\_b)

print(idx\_b)

print(accumulators\_b[int(idx\_b/accumulators\_b.shape[1]),int(idx\_b%accumulators\_b.shape[1])])

mask\_b = [accumulators\_b > 125.] [0] \* 1.

accum\_b = accumulators\_b \* mask\_b

rho\_b = []

theta\_b = []

for i in range(mask\_b.shape[0]):

for j in range(mask\_b.shape[1]):

if mask\_b[i,j]==1:

rho\_b.append(i)

theta\_b.append(thetas\_b[j])

# In[80]:

r\_b,t\_b= minimise\_no\_rhos(rho\_b,theta\_b)

# In[65]:

rho\_b = [489,490,564,565,637,638,708,709,779,780,853,854,930,931]

theta\_b =[-0.9424777960769379]

#gray\_img = cv2.imread('original\_imgs/hough.jpg', cv2.IMREAD\_GRAYSCALE)

dup\_b = np.copy(edges\_b)

#gs = gray\_img.shape

diag\_len\_b = np.ceil(np.sqrt(dup\_b.shape[0] \*dup\_b.shape[0]+ dup\_b.shape[1] \* dup\_b.shape[1]))

indices\_b = []

for i in range(dup\_b.shape[0]):

for j in range(dup\_b.shape[1]):

for t in theta\_b:

rho\_dup = round(i \* np.cos(t) + j \* np.sin(t))+diag\_len\_b

#print(rho\_dup, rho\_b)

for r in rho\_b:

if r == rho\_dup:

indices\_b.append([i, j])

# In[66]:

gray\_img\_b = cv2.imread('original\_imgs/hough.jpg')

#output = np.zeros((gray\_img\_b.shape[0],gray\_img\_b.shape[1]))

for i in indices\_b:

gray\_img\_b[i[0],i[1]] = (0,255,0)

cv2.imwrite("output/blue\_line.jpg",gray\_img\_b)

**3.3 Output**

****

Figure 10: Red Line



Figure 11: Blue Line

**4 Conclusion**

All the tasks such as Morphology, Segmentation and Hough transform are implemented for the given input images and the respective code and output images are provided in the report.

**References**

[1] Richard Szeliski (2010), Computer Vision: Algorithms and Applications

[2] <https://docs.opencv.org/2.4/doc/tutorials/>

[3] <https://docs.python.org/3/>