ECE661: Homework 6

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

October 22, 2018

In This homework we are required to carry out image segmentation using two different methods,RGB segmentation and Texture based segmentation, using Otsu's algorithm for finding the threshold for segmentation. Then we are required to extract the contour of the segmented output.

For image segmentation using RGB color channels, we first segment the RGB channels individually using Otsu's and then combine the masks obtained from these channels to get the final segmented output.

For image segmentation using textures, we first generate texture features using three different sizes of windows and then segment the three channels separately using Otsu's and then combine the masks obtained to get the final segmented output.

1 Logic and Methodology

The task of this homework can be further divided into the following sub-tasks:

a) Identify threshold using Otsu's Algoritm

b) RGB Segmentation

c) Texture generation and Texture based segmentation

d) Contour extraction

1.1 Otsu's Algorithm

Otus's Algorithm helps us in identifying a threshold which separates the Foreground(C_1) from the background (C_0). Given any gray-scale image (or basically single scalar value at one x,y location), Otsu's Method finds the separation threshold with following steps:

1. Step-1
   Divide the gray-scales into L levels $[1, 2, 3, ..., K, ..L]$ and construct a histogram of the image with L levels in the x axis. Compute the probability distribution function from the histogram such that,

   $$p_i = n_i/N$$

   where p_i is the probability that a pixel value is in the ith level, n_i is the height of the ith column of the in the histogram and N is the sum of height of all columns in the histogram. Also compute the total mean using,$\mu_t = \Sigma_{i=1}^{L} i p_i$

2. Step-2
   Divide the pixels into two classes C_0 and C_1(background and foreground or vice versa) using the threshold k such that C_0 denotes the pixel levels [1,...K] and C_1 denotes the levels [K+1,...., L].

3. Step-3
   Then find the class probabilities $\omega(k) = \Sigma_{i=1}^{K} p_i$ and mean $\mu(k) = \Sigma_{i=1}^{K} i p_i$. Then find between class variance using,
   $$\sigma_B^2(k) = \frac{[\mu_t \omega(k) - \mu(k)]^2}{\omega(k)(1 - \omega(k))}$$

4. Step-4
   Find K^* that maximizes the between class variance. Store K^*

5. Step-5
   For the next iteration, update the probabilities such that the new region is from K* to L. Therefore, take only the values of the histogram corresponding to this region. The formula for finding probabilities remains the same, but now N will become the sum of height of columns of levels K* to L. Repeat Steps-3 and 4. Store new K*.

6. Step-6
   Repeat steps3,4,5 till specified number of iterations

7. Step-7
   Using the final threshold construct a binary mask such that all the pixel values in the image which are greater than this threshold get a value of 1 **else** 0.

It may be noted that, the algorithm presented above will give us a separation threshold but in general we will not know whether our foreground is below or above this threshold. Therefore, this method requires the user to try out manually several iterations to get a satisfactory separation. It is best to judge the number of iterations using the plot of probability distribution, K* and output mask.

As can be seen from figure 1,in order to get the peak in the last portion of the distribution the second

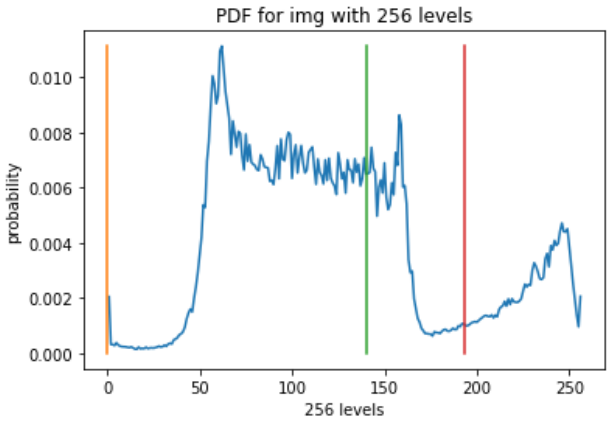


Figure 1: Otsu's method applied onto an image with 2 iterations

iteration(red line) gives us a better separation threshold.

Further, with my implementation we will be only be able to move towards the right in the levels. If our foreground is on the left, then I am inverting the gray-scale image and using the same implementation.

1.2 RGB segmentation

For RGB segmentation the idea is to find individual masks for different color channels separately and then combine the masks using an y logical operator. The steps involved are as follows:

a) Step-1: Find individual color channel and store separately

b) Step-2: Perform Otsu's method and obtain a binary mask for all color channels

c) Step-3: Perform a logical operation on the different color masks to get the final segmented output. This step greatly depends on the colors present in our foreground and background. We might need to perform different combinations of and/or/not to get the final segmented image. For example, for the lighthouse image, our lighthouse is in red and background has two prominent colors blue and green. To get the final image we can do

$$final = R \cap G' \cap B'$$

where G' and B' are the compliments of masks for green the blue.

1.3 Texture generation and Texture based segmentation

We generate texture features for using variance in an NxN (N is odd) window. The steps to generate the texture feature are:

a) Step-1: Read the image as a gray-scale image

b) Step-2: pad the image with (N-1)/2 **repetitive** borders. (repetitive borders will give better results compared to zero padding because when we will do scaling of variances then zero padding can come amongst the highest of variances and thus affect the final output)

c) Step-3: Given N for every point (i,j) in the gray-scale image, We construct a NxN window in the padded image and find the variance of gray scale values in the window and assign this value to the texture feature at (i,j) location.

d) Step-4: Scale the texture feature matrix to bring it to the range of 0-255 using a simple transformation.

e) Step-5: Repeat Steps3 and 4 for three different values of N.

f) Step-6: Stack the texture feature matrices page by page.

After generation of the texture features we then treat one texture page as an individual channel and then repeat the same process carried out in RGB segmentation.
For my implementation, I have used the values 3,5 and 7 for three different sizes of window to generate the texture features.

1.4 Contour Extraction

After obtaining the final combined mask from any of the methods. The mask thus obtained will have a value of 1 for the foreground and 0 for the background. I am using the information of 8-neighbours of any pixel to construct the contour using the following steps:

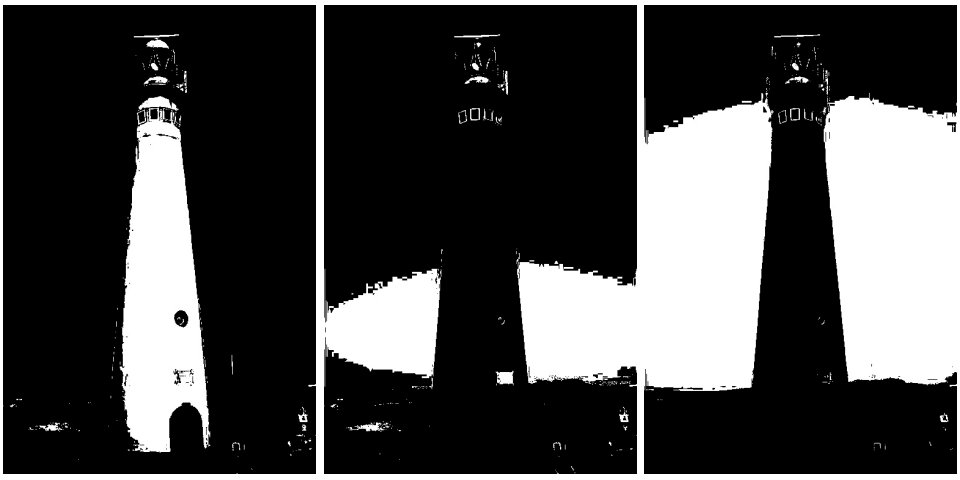
- Start with an empty(all zeros) contour matrix of the same size as the mask
- Loop over image pixels (i,j) in the mask
- **if** pixel value in the mask is **0** (ie in the background), then move onto the next (i,j)
- **if** pixel value in the mask is **1**, check **if** all 8-neighbours (ie 3x3 window) are **all 1s** . **If** yes (ie interior of foreground and not on the border), then move on to next (i,j) **else** record 1 at (i,j) in the contour matrix

Further, Sometimes it is useful to smooth the segmented image before contour extraction in order to get a clear contour.

2 Results

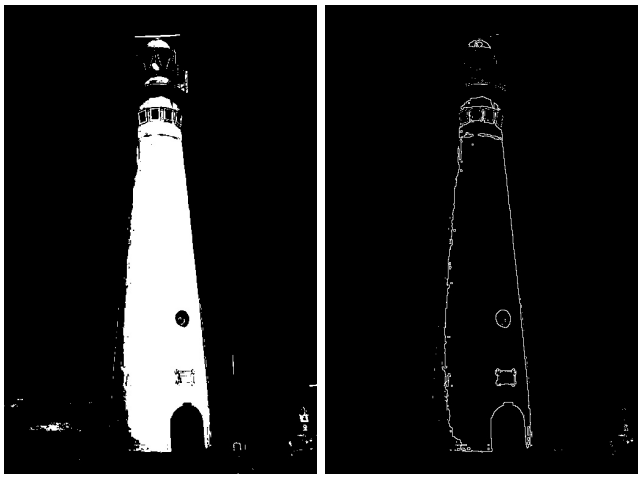
2.1 Lighthouse

Figure 2: original image



(a) Red Mask (b) Green Mask (c) Blue Mask

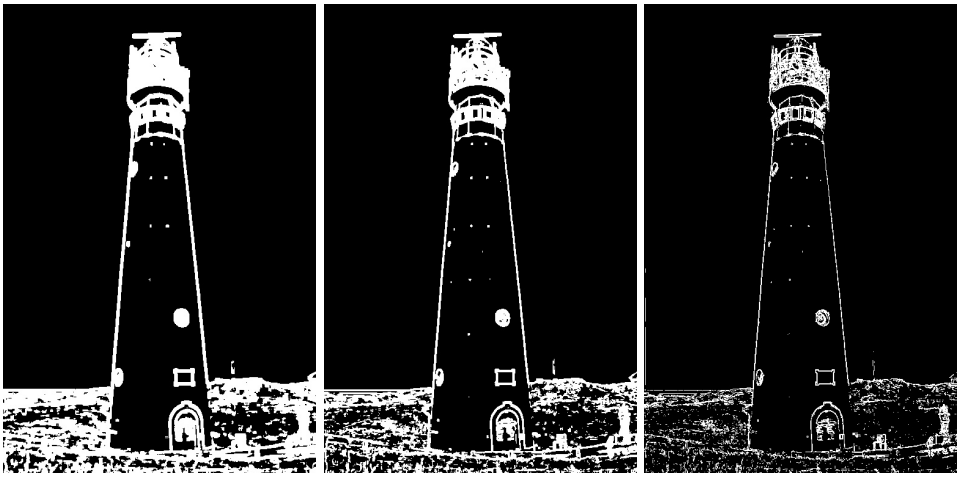
Figure 3: RGB Segmentation



(a) Combined Color Segmented output

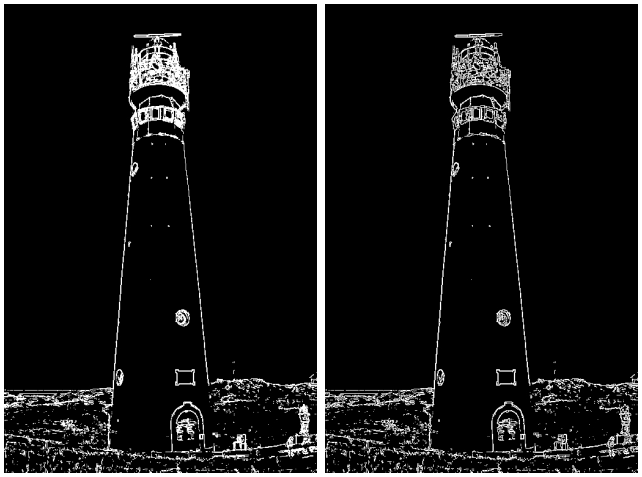
(b) Contour

Figure 4: Texture features image



(a) N=3 Mask (b) N=5 Mask (c) N=7 Mask

Figure 5: Texture Segmentation



(a) Combined Texture Segmented output (b) Contour

2.2 Baby

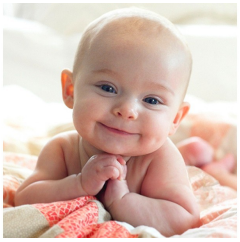
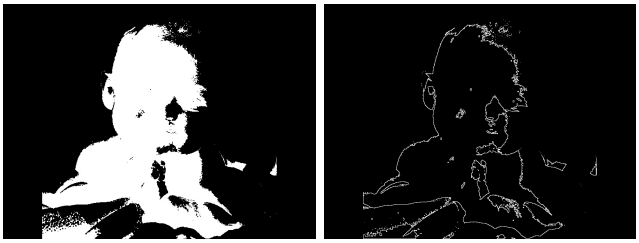


Figure 6: original image

(a) Red Mask (b) Green Mask (c) Blue Mask

Figure 7: RGB Segmentation



(a) Combined Color Segmented output (b) Contour

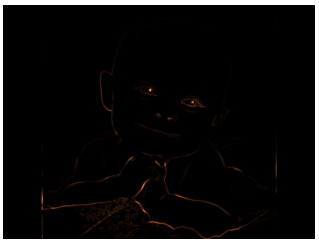


Figure 8: Texture features image

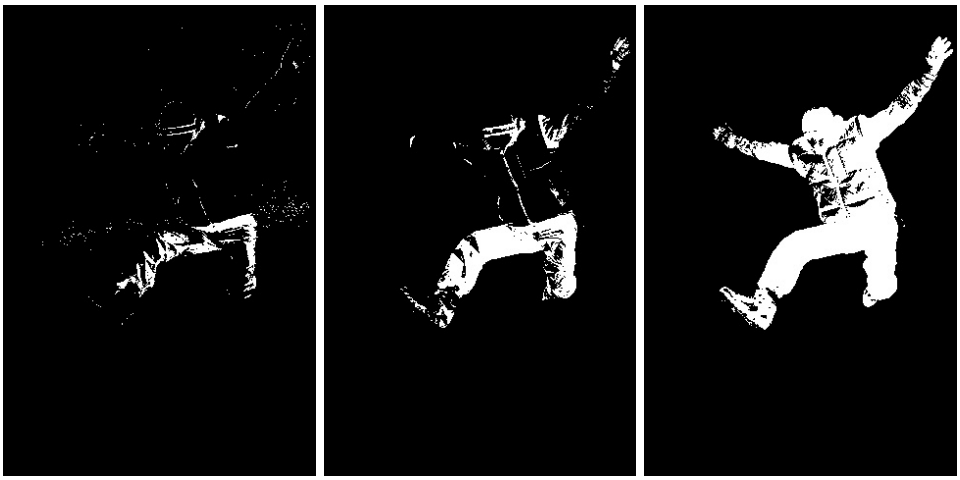
(a) N=3 Mask (b) N=5 Mask (c) N=7 Mask

Figure 9: Texture Segmentation

(a) Combined Texture Segmented output (b) Contour

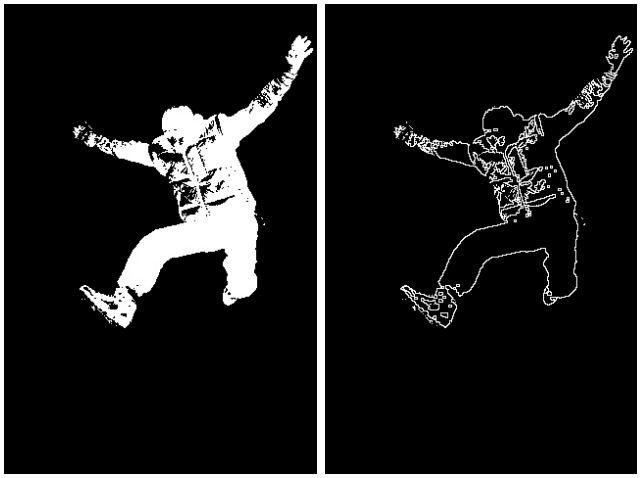
2.3 Ski

Figure 10: original image



(a) Red Mask (b) Green Mask (c) Blue Mask

Figure 11: RGB Segmentation



(a) Combined Color Segmented output

(b) Contour

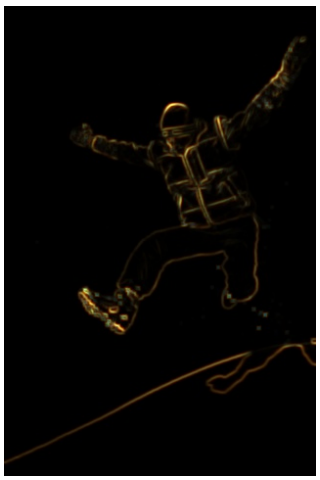
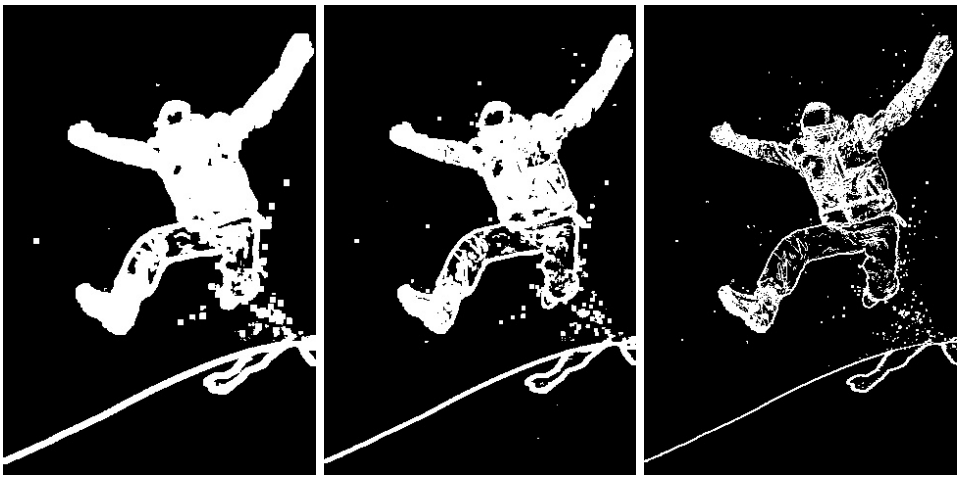
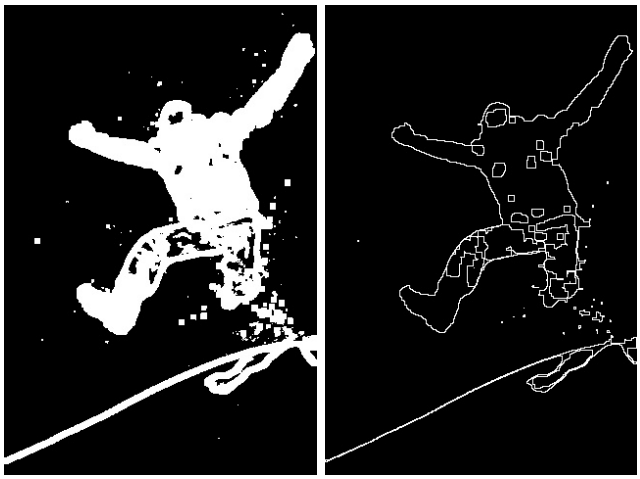


Figure 12: Texture features image



(a) N=3 Mask (b) N=5 Mask (c) N=7 Mask

Figure 13: Texture Segmentation



(a) Combined Texture Segmented output (b) Contour

3 Observations

a) The performance of segmentation methods, i.e. RGB or texture, depends mainly on the characteristics of our original image and the foreground and background in this image. As it can be observed, that RGB segmentation was superior in for the 'ski' image in comparison to the 'baby' image, whereas texture based segmentation was superior for the 'baby' image than the 'ski' image.

b) RGB based segmentation was found to work better when we had our foreground/background as a single gray level and no sharing of gray levels between the foreground and background. For instance, in the 'ski' image the background was sky blue and white snow (white has pure blue) therefore when we generated the mask for the blue channel we got our complete foreground easily.

c) Texture based segmentation gave all the edges/ borders of different components in the image. It can be used to get the contour directly but it can also pick edges/patterns from the background. For instance, in the 'baby' image we saw poor performance of RGB segmentation (mainly because all the colors were resembled very closely) but with Texture based segmentation, we were able to pickup the borders of the baby's body but also picked up the texture in the blanket(i.e. background).

d) Gaussian smoothing of the segmented output before extracting the contour can help in reducing noisy data from the background. I had used Gaussian smoothing for texture segmented mask before finding its contour to remove noisy data in the background.

e) Obtaining the threshold from Otsu's by changing number of iterations and identifying where our foreground is in the distribution was a manual process and therefore this scheme may not the suitable for automatic segmentation.

4 Source Code:

4.1 main file

```
"""
ECE661: hw6 main file
@author: Rahul Deshmukh
email: deshmuk5@purdue.edu
PUID: 0030004932
"""
#import libraries
import cv2
import sys
import numpy as np
sys.path.append('../../')
import MyCVModule as MyCV
#define path
readpath='../images/' # path of images to be read
savepath='../results/' #path for saving results of images

# define case 0,1,2
case=0

# define readnames
if case==0:
    readname='lighthouse'
elif case==1:
    readname='baby'
elif case==2:
    readname='ski'
savepath=savepath+readname+'/'
# read color image
imgC=cv2.imread(readpath+readname+'.jpg')
img_gray=cv2.imread(readpath+readname+'.jpg',0)
m=imgC.shape[0]
n=imgC.shape[1]

#%%
# define number of otsu iterations for the RGB channels
if case==0:
    N=[2,3,2]#R,G,B
    imgR=imgC[:,:,2]
    imgG=imgC[:,:,1]
    imgB=imgC[:,:,0]
elif case==1:
    N=[1,1,1]
    imgR=MyCV.invert_img(imgC[:,:,2])
    imgG=MyCV.invert_img(imgC[:,:,1])
    imgB=MyCV.invert_img(imgC[:,:,0])
elif case==2:
    N=[4,4,2]
    imgR=MyCV.invert_img(imgC[:,:,2])
    imgG=MyCV.invert_img(imgC[:,:,1])
    imgB=MyCV.invert_img(imgC[:,:,0])
L=256 # number of levels
# find k_star from otsu using color channels
print('Red')
k_R=MyCV.Otsu(imgR,L,N[0])
print('Green')
k_G=MyCV.Otsu(imgG,L,N[1])
print('Blue')
k_B=MyCV.Otsu(imgB,L,N[2])
# find masks for the color channels
maskR=MyCV.img_mask(imgR,L,k_R)
```

```
maskG=MyCV.img_mask(imgG,L,k_G)
maskB=MyCV.img_mask(imgB,L,k_B)
# save binary images
cv2.imwrite(savepath+readname+'_maskR.jpg',255*maskR)
cv2.imwrite(savepath+readname+'_maskG.jpg',255*maskG)
cv2.imwrite(savepath+readname+'_maskB.jpg',255*maskB)
# make combined image
if case==0:
    color_seg_img = maskR*MyCV.not_img(maskG)*MyCV.not_img(maskB)
    option=0;sigma=2.5;N=5
elif case==1:
#    color_seg_img =MyCV.not_img(MyCV.or_img(MyCV.or_img(maskR,maskG),maskB))
    color_seg_img =maskR*maskG*maskB
    option=0;sigma=2.5;N=5
elif case==2:
    color_seg_img = MyCV.or_img(maskB,maskG)#*MyCV.not_img(maskR)
    option=0;sigma=2.5;N=10
#save color based segmented image
cv2.imwrite(savepath+readname+'_color_seg.jpg',255*color_seg_img)
# contour extraction of color segmented image
cont_img=MyCV.contour_img(color_seg_img,option,sigma,N)
#save contour
cv2.imwrite(savepath+readname+'_color_contour.jpg',255*cont_img)
#%%
#-------------texture based segmentation----------------------#
N=[3,5,7]
text_img=MyCV.textured_img(img_gray,N)
# save textured image
cv2.imwrite(savepath+readname+'_text_img.jpg',text_img)
# otsu's segmentation on textured image
# define number of otsu iterations for the RGB channels
if case==0:
    N=[3,3,3]#R,G,B
    imgR=MyCV.invert_img(text_img[:,:,2])
    imgG=MyCV.invert_img(text_img[:,:,1])
    imgB=MyCV.invert_img(text_img[:,:,0])
elif case==1:
    N=[3,3,3]
    imgR=MyCV.invert_img(text_img[:,:,2])
    imgG=MyCV.invert_img(text_img[:,:,1])
    imgB=MyCV.invert_img(text_img[:,:,0])
elif case==2:
    N=[4,4,4]
    imgR=MyCV.invert_img(text_img[:,:,2])
    imgG=MyCV.invert_img(text_img[:,:,1])
    imgB=MyCV.invert_img(text_img[:,:,0])
L=256 # number of levels
# find k_star from otsu using color channels
k_R=MyCV.Otsu(imgR,L,N[0])
k_G=MyCV.Otsu(imgG,L,N[1])
k_B=MyCV.Otsu(imgB,L,N[2])
# find masks for the color channels
maskR=MyCV.not_img(MyCV.img_mask(imgR,L,k_R))
maskG=MyCV.not_img(MyCV.img_mask(imgG,L,k_G))
maskB=MyCV.not_img(MyCV.img_mask(imgB,L,k_B))
# save binary images
cv2.imwrite(savepath+readname+'_mask1.jpg',255*maskR)
cv2.imwrite(savepath+readname+'_mask2.jpg',255*maskG)
cv2.imwrite(savepath+readname+'_mask3.jpg',255*maskB)
# make combined image
if case==0:
    text_seg_img=maskR*maskG*maskB
    option=0;sigma=2.5;N=5
elif case==1:
    text_seg_img=maskR*maskG*maskB
```

```
    option=0;sigma=1.5;N=3
elif case==2:
    text_seg_img=MyCV.or_img(MyCV.or_img(maskB,maskG),maskR)
    option=1;sigma=4;N=10
#save color based segmented image
cv2.imwrite(savepath+readname+'_text_seg.jpg',255*text_seg_img)
# contour extraction of texture segmented image
cont_img=MyCV.contour_img(text_seg_img,option,sigma,N)
#save contour
cv2.imwrite(savepath+readname+'_texture_contour.jpg',255*cont_img)
```

4.2 Functions

```
#=====================HW6=================================#
#%%
# function for contour extraction from binary mask
def contour_img(img,option,sigma,N):
    """
    Using 8-neighbours for defining foreground
    Input: img= binary image mxn
           option= 0 or 1 for gaussian smoothing of binary image
           prior to extraction
           sigma= std dev for gaussian filter
           N: size of gaussain filter
    Output: cont_img= contour image (binary image)
    """
    m=img.shape[0];n=img.shape[1]
    # do gaussian smoothing based on option value
    if option==1:
        img = cv2.GaussianBlur(img,(5,5),sigma)
    # create empty image
    cont_img=np.zeros((m,n))
    N=3# size of neighbours window
    pad_img=np.zeros((m+2*N//2,n+2*N//2))
    pad_img[N//2:-N//2,N//2:-N//2]=img
    # loop over img xy coordinates
    for i in range(m):
        for j in range(n):
            # make NxN window
            window=pad_img[i:i+2*N//2,j:j+2*N//2]
            # window center should be 1 for contour point
            if window[1,1]==1:
                # if all 1s in window then not a contour point
                if np.sum(window)!=N**2:
                    cont_img[i,j]=1
    return(cont_img)
#%%
# function for texture based segmentation
def textured_img(img,N):
    """
    Input: img: grayscale image matrix mxn
           N: list of 3x1 specifying sizes of window
           all N_i are assumed to be odd numbers
    Output: text_img: mxnx3 image matrix with window variances as pixel
                      values.
    """
    m=img.shape[0];n=img.shape[1]
    #define empty texture image
    text_img=np.zeros((m,n,3))
    #loop over pixel pages in text_img
    for i in range(3):
        #pad the original image using Ni
        pad = N[i]//2
        pad_img=cv2.copyMakeBorder(img,pad,pad,pad,pad,cv2.BORDER_REPLICATE)
#         pad_img=np.zeros((m+2*(N[i]//2),n+2*(N[i]//2)))
```

```
#         pad_img[(N[i]//2):-(N[i]//2),(N[i]//2):-(N[i]//2)]=img
        #loop over xy coordinates of text_img
        for j in range(m):
            for k in range(n):
                window=pad_img[j:j+2*(N[i]//2),k:k+2*(N[i]//2)]
                text_img[j,k,i]=np.var(window)
        #convert text_img pixels to the range of 0-255
        min_x=np.min(text_img[:,:,i])
        max_x=np.max(text_img[:,:,i])
        text_img[:,:,i]=(255/(max_x-min_x))*(text_img[:,:,i]-min_x*np.ones((m,n)))
    #convert text_img pixel values to integers
    text_img=text_img.astype(int)
    return(text_img)
#%%
# function for inverting a biinary image
def or_img(img1,img2):
    """
    Input: img1,2: binary images
    output: logical OR of image
    """
    op=(img1+img2-img1*img2)
    return(op)
#%%
# function for inverting a biinary image
def invert_img(img):
    """
    Input: img: single color channel image
    output: inverted color matrix
    """
    op=255*np.ones((img.shape[0],img.shape[1]))-img
    return(op)
#%%
#function for finding not a binary image
def not_img(mat):
    """
    Input: mat:2d matrix with only ones and zeros
    Output: logical not of mat
    """
    op=np.ones((mat.shape[0],mat.shape[1]))
    op=op-mat
    return(op)
#%%
# function for generating binary mask
def img_mask(img,L,k):
    """
    Input:img: image matrix mxn
            L: number of levels in histogram
            k: separation threshold using Otsu
    Output:mask: binary image size mxn
    """
    m=np.shape(img)[0];n=np.shape(img)[1]
    mask=np.zeros((m,n))
    nbins=256/L
    for i in range(m):
        for j in range(n):
            if img[i,j]//nbins>=k:
                mask[i,j]=1
    return(mask)
#%%
def Otsu(img,L,N):
    """
    Input: img: image matrix mxn
            L: number of levels in histogram
            N: number of iterations to be performed
    Outut: k_star: threshold that separates foreground and background
```

```
    """
    # get initial PDF of image pixels
    h=img_histogram(img,L)
    p=h/np.sum(h)
    p_ini=p
    # find u_t total mean
    ip=np.arange(1,L+1)
    u_t=ip@p
    # loop over number of iterations
    k_star=[0]
    ipnew=ip
    for iN in range(N):
        # initialize omega,u sigma as zeros
        omega=np.zeros((L-k_star[iN],1))
        u=np.zeros((L-k_star[iN],1))
        sigma2_b=np.zeros((L-k_star[iN],1))
        for k in range(L-k_star[iN]):
            omega[k]=np.sum(p[:k])
            u[k]=ipnew[:k]@p[:k]
            if omega[k]>0 and omega[k]<1:
                sigma2_b[k]= ((u_t*omega[k]-u[k])**2)/(omega[k]*(1-omega[k]))
        # find k_star at max sigma2_b value
        k_star.append(k_star[-1]+np.argmax(sigma2_b))
        # find image with pixel values greater than k_star only
        # find new probability in the region k_star-L
        hnew=h[k_star[-1]:L]
        ipnew=np.arange(k_star[-1]+1,L+1)
        p=hnew/np.sum(hnew)
        u_t=ipnew@p
    # make plots
    plt.plot(ip,p_ini)
    plt.plot([k_star,k_star],[0,np.max(p_ini)])
    plt.title('PDF for img with '+str(L)+' levels')
    plt.xlabel(str(L)+' levels')
    plt.ylabel('probability')
    plt.show()
    return(k_star[-1])
#%%
# function for generating histogram of image pixels
def img_histogram(img,L):
    """
    Input: img: image matrix mxn
           L: number of levels in histogram
    Outut: h:Lx1 vector with histogram values ni/N
    """
    nbins=256/L
    h=np.zeros((L,1))
    # raster scan of image to read pixel values
    for i in range(np.shape(img)[0]):
        for j in range(np.shape(img)[1]):
            #update ith bin in p by 1, where i depends on img[i,j]
            h[int(img[i,j]//nbins)]+=1
    return(h)
#=======================HW6=======================#
```