HOMEWORK 3
Mythra Varun Balakuntala SM
October 23, 2018

1. Image Segmentation

Image segmentation refers to partitioning images based on certain conditions. Various methods exists for image segmentation like clustering methods, histogram-based methods, compression-based methods. The goal of this assignment is to use one of the most popular methods for histogram thresholding, Otsu's method. Given a mulit-channel image we can apply Otsu thresholding on each channel and logically combine them to get the desired foreground. Here the channels can be BGR, HSV, or Texture channels.

2. Otsu's Method

The procedure of Otsu's thershold selection for gray level histogram is explained in this section. Let the image be represented in L gray levels. Let number of pixels in level i be n_i and the total number of pixels be N. The histogram is normalized and treated as probability distribution using the following transformation,

$$p_i = \frac{n_i}{N} \qquad p_i \geq 0 \tag{1}$$

We see $\sum_{i=1}^{L} p_i = 1$. Now let us say we partition the levels into two classes at level k. Let $C_0 = [1, 2, \ldots, k]$ and $C_1 = [k+1, k+2, \ldots, L]$. The probabilities of class occurrence and class mean levels are given by,

$$\begin{aligned}
\omega_0 &= \sum_{i=1}^{k} p_i = \omega(k) \\
\omega_1 &= \sum_{i=k+1}^{L} p_i = 1 - \omega(k) \\
\mu_0 &= \sum_{i=1}^{k} \frac{ip_i}{\omega_0} = \frac{\mu(k)}{\omega(k)} \\
\mu_1 &= \sum_{i=k+1}^{L} \frac{ip_i}{\omega_0} = \frac{\mu(k)}{\omega(k)} \\
\text{where} \quad \mu(k) &= \sum_{i=1}^{k} ip_i
\end{aligned}$$

The following equations hold based on the definitions,

$$\begin{aligned}
\omega_0 + \omega_1 &= 1 \\
\omega_0\mu_0 + \omega_1\mu_1 &= \mu_T = \sum_{1}^{L} ip_i
\end{aligned}$$

Three variances can be constructed based on variance of each class : σ_B^2 the between class variance, σ_W^2 the intra-class variance and σ_T^2 the total variance. They are related by $\sigma_B^2 + \sigma_W^2 = \sigma_T^2$.
Otsu's threshold is that k which maximizes the between class variance or minimizes the intra-class variance. Essentially the solution to,

$$\max_{1 \leq k \leq L} \sigma_B^2(k) = \max_{1 \leq k \leq L} \frac{(\mu_T\omega(k) - \mu(k))^2}{\omega(k)(1 - \omega(k))}$$

Given a image, the steps to compute the Otsu's threshold are as follows

- Compute the histogram for the image using g bins. Then normalize the histogram based on number of pixels i.e. find $p_i = n_i/N$ where n_i is ith count in the histogram.
- Compute $\mu^T = \sum_{k=1}^{L} ip_i$. Then initialize $\omega(k) = 0$ and $\mu(k) = 0$.
- Iterate over all the k values to find maximum of σ_B^2.
  At each step, the values of $\omega(k)$ and $\mu(k)$ are computed as follows

$$\begin{aligned} \omega(k) &= \omega(k-1) + p_k \\ \mu(k) &= \mu(k-1) + kp_k \end{aligned}$$

  Then we compute

$$\sigma_B^2(k) = \frac{(\mu_T \omega(k) - \mu(k))^2}{\omega(k)(1-\omega(k))}$$

  and if $\sigma_B^2(k) > \sigma_B^2(k-1)$ we update the $k* = k$ (The Otsu's threshold)

3. Multi-channel Otsu

Image segmentation for multiple channels can be performed based on single channel Otsu method described above. The idea is to perform Otsu on each channel and combine them using a logical and operator. This can be done on BGR, HSV or Texture channeled images.
The steps are as follows

1. Extract the channels from given multi-channel image
2. Perform Otsu thresholding on each of the channels
3. Combine the result of each channel using logical and, i.e. Result Image = $\mathbf{omask_1}$ & $\mathbf{omask_2}$ & $\mathbf{omask_3}$ Where $\mathbf{omask_i}$ is the Otsu threshold mask obtained for ith channel. The Otsu mask is a binary image with 1s at all foreground pixels and 0s in the background pixels.
   The foreground and background are chosen based on image/channel i.e. they are image/channel dependent. For example, foreground for image 1 may be all pixels with gray levels $<$ Otsu threshold k_1^* but for image 2 it may be pixels with gray levels $>$ Otsu threshold for k_2^*.

Another method to threshold multi-channel images is to iteratively perform thresholding on each channel,i.e. First threshold the first channel. Then use the foreground obtained for first channel as a mask for second channel. On this resulting image perform Otsu thresholding again. The can be repeated iteratively for all channels.

4. Texture Segmentation

Texture segmentation uses multi-channel Otsu described above. But, there is an additional step of computing the inputs to the multi-otsu algorithm. The way this is done is using local variances in gray-scale version of the image. The steps are described below,

1. Convert image to gray-scale (cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).
2. Compute an image of same size as the gray-scale whose pixel values are equal to the variance in a $N \times N$ neighbourhood centered at the pixel in the gray-scale image.
3. Find multiple images following previous step for different N values (3,5,7,..). Treat each of these images as channels and combine them into a single image with multiple channels.
4. Perform multi-Otsu thresholding on this multi-channel image as shown in previous section.

5. Contour Extraction

Two methods are used for computing the contour of the image.

Method 1

The contour extraction is performed based on 8-connected paths. We assume the foreground is 8-connected. The method used is Moore-Neighbour tracing. The idea is to start at a particular start pixel (found by searching row-wise for first white pixel in Otsu mask).
On reaching a white pixel we look at its 8-neighbours in a clock-wise direction to find the first pixel in the 8-nieghbourhood along the clokwise direction that is white then step to that pixel.
Then, we repeat the search for white pixel in the 8-neighbourhood in a clockwise direction starting next to(in clockwise sense) the previous white pixel we came from.
We keep repeating this step-search until we reach back to the starting pixel. The pixels we stepped on gives the boundary for the image.

As there may be multiple contours in a given image we extend this by choosing a start pixel as any white pixel encountered during raster scan of the image. We consider a white pixel as boundary pixel if there is at-least 1 black pixel in its 8-neighbourhood.
Then we perform Moore-Neighbour tracing starting at this pixel. All pixels found in this contour are stored in a boundary pixel array. The algorithm resumes the raster scan for white pixel after this contour is found. To avoid, reading same contour multiple times, we see if a pixel already exists in the boundary pixel array, if so we ignore it and continue. This is because a white pixel can only be one border.

Method 2

This is done using a very simple method. Find two images, one by expanding the Otsu mask image by 1 pixel and other by shrinking the Otsu mask image by 1 pixel. Find the difference between the two. This gives the border of the image. This is much faster than method 1 if we use inbuilt cv2.dilate and cv2.erode for expansion and shrinking.

6. Results

Observations

1. The segmentation is performed by using both HSV and BGR images. It was seen that using HSV was able to better capture desired foreground (in general across images), but BGR was only good for specific images with good differentiation in colors.

2. The texture channeled images are better for extracting contours across all images, but were not as good for segmentation to capture just the reference object (See results of first image, It captures lighthouse very well but also includes some features on the ground).

3. Closing holes by first dilating and then eroding resulted in better contours and better masks without small holes.

4. Combining masks (Using logical OR) from different methods (HSV,BGR,Texture) resulted in a better segmentation mask.

Original Images

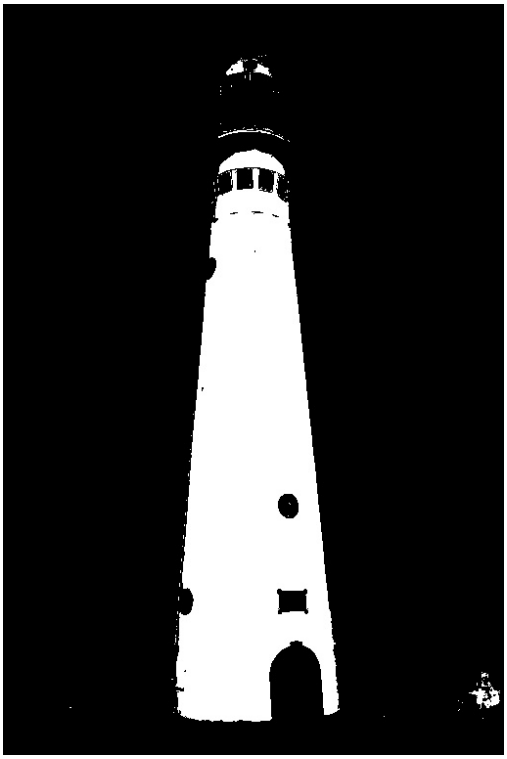
Figure 1: Source image of Baby (Image 2)

(a) Source image of Lighthouse (Image 1)

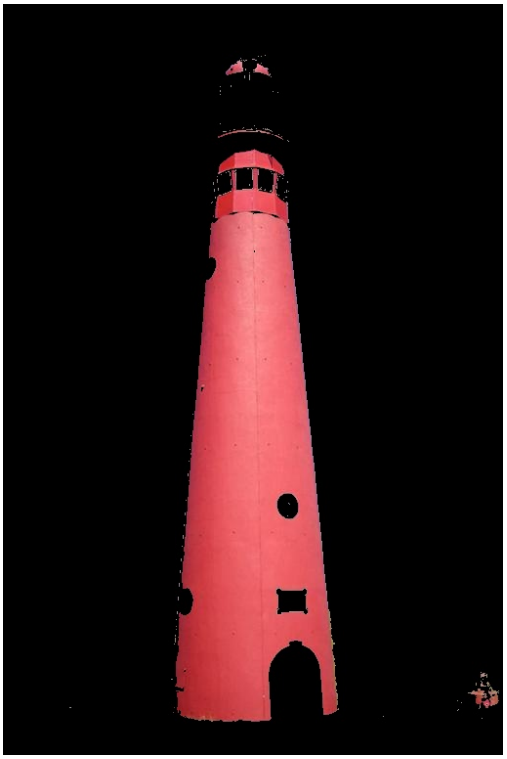
(b) Source image of skier (Image 3)

Results for image 1 (light house)

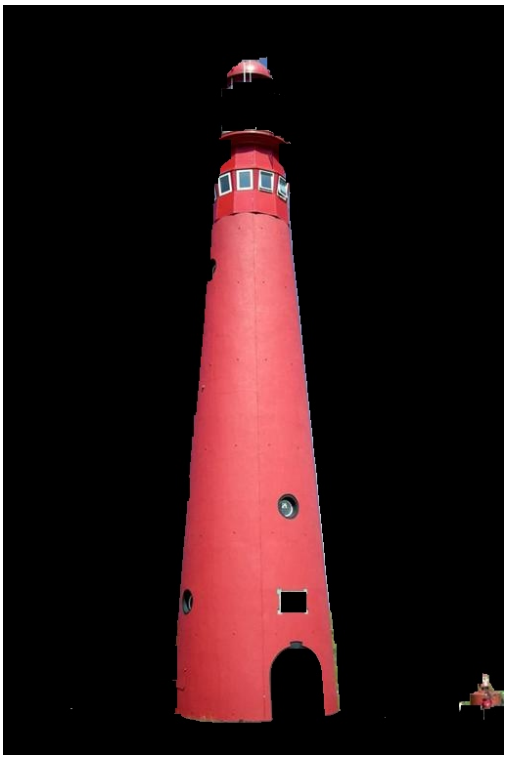
BGR Results



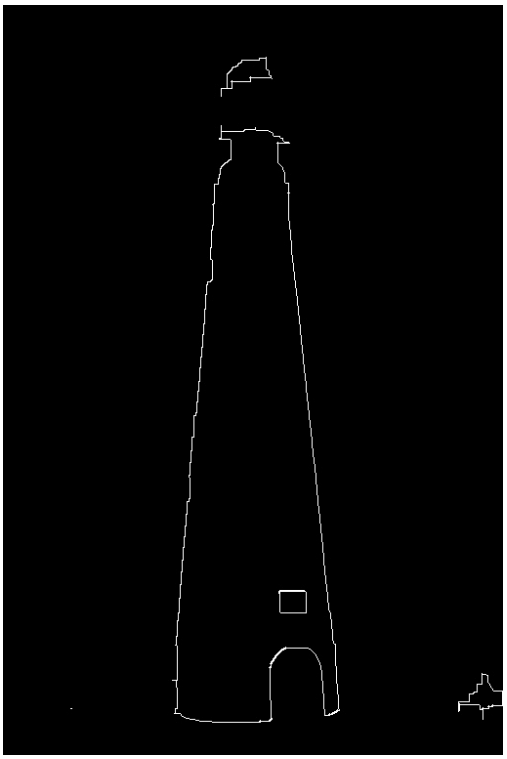
(a) Otsu mask for BGR image 1



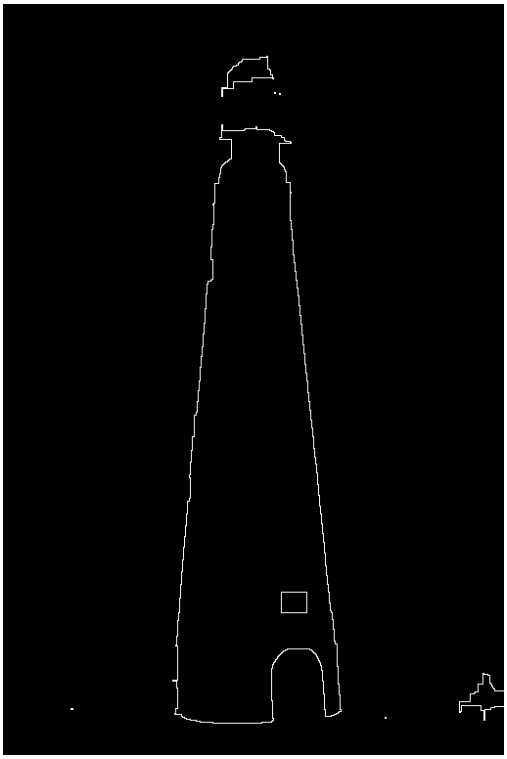
(b) Otsu masked BGR image 1



(a) Otsu masked BGR image 1 with closing

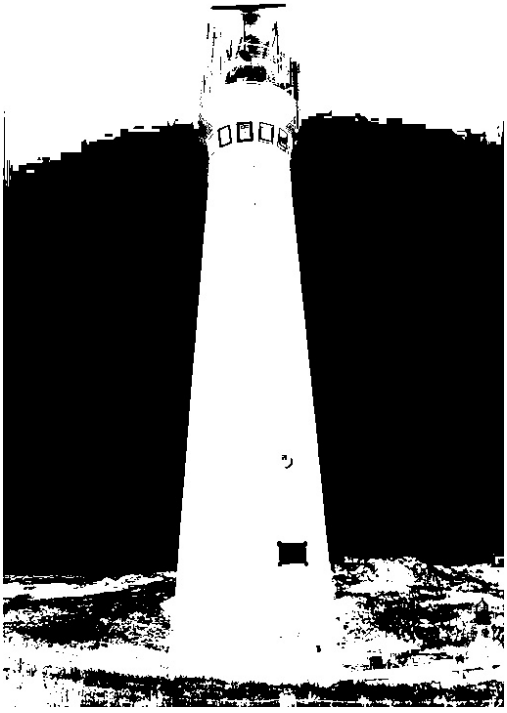


(b) Contour of image 1 Method 1



(a) Contour of image 1 Method 2

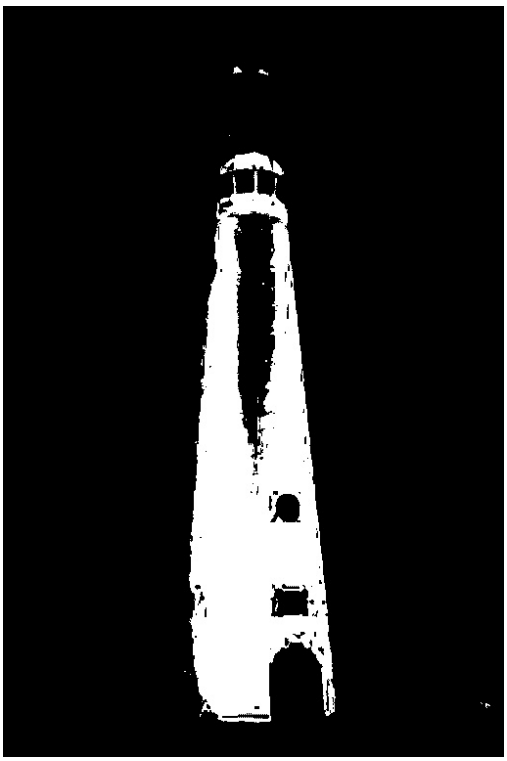
(b) Otsu Mask Channel 1



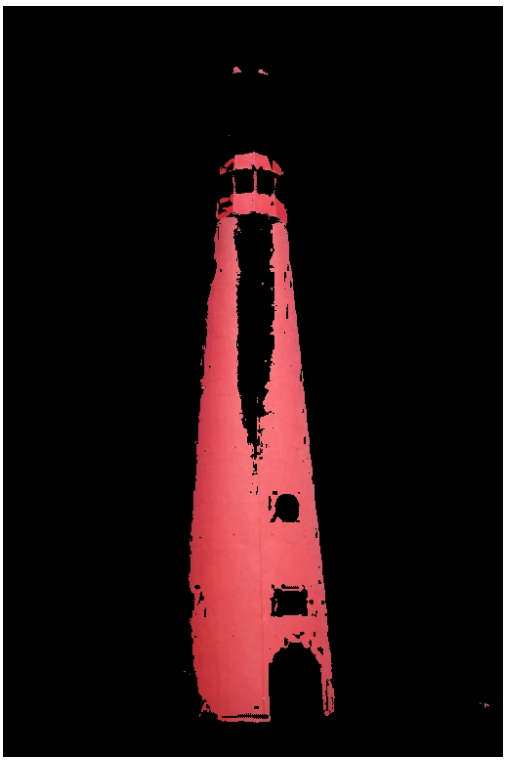
(a) Otsu Mask Channel 2

(b) Otsu Mask Channel 3

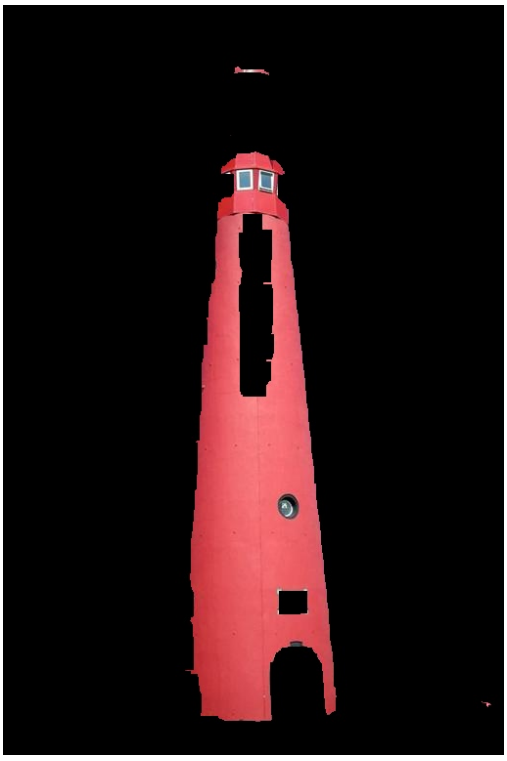
HSV Results



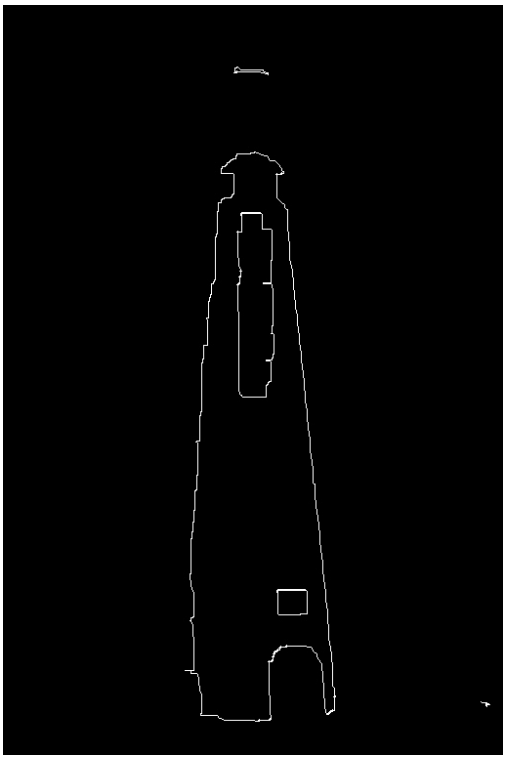
(a) Otsu mask for HSV image 1



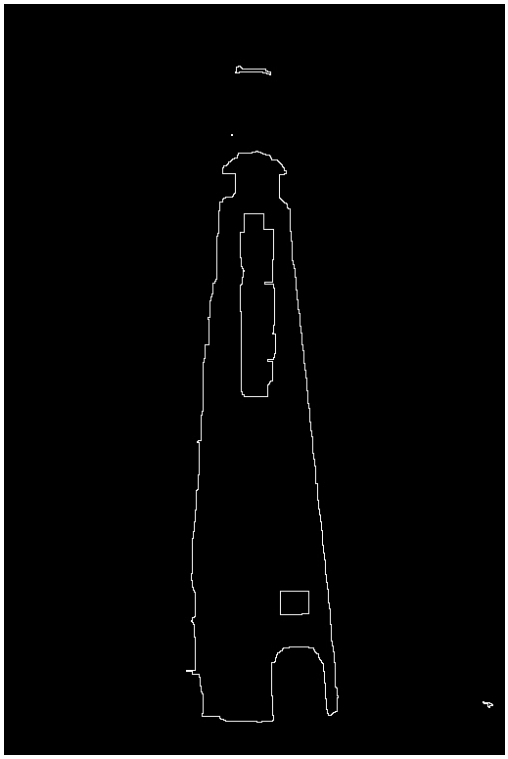
(b) Otsu masked HSV image 1



(a) Otsu masked HSV image 1 with closing

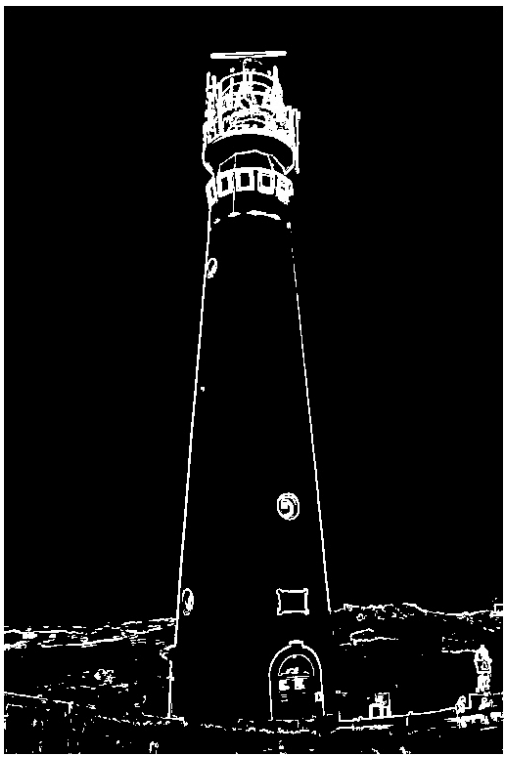


(b) Contour of image 1 Method 1

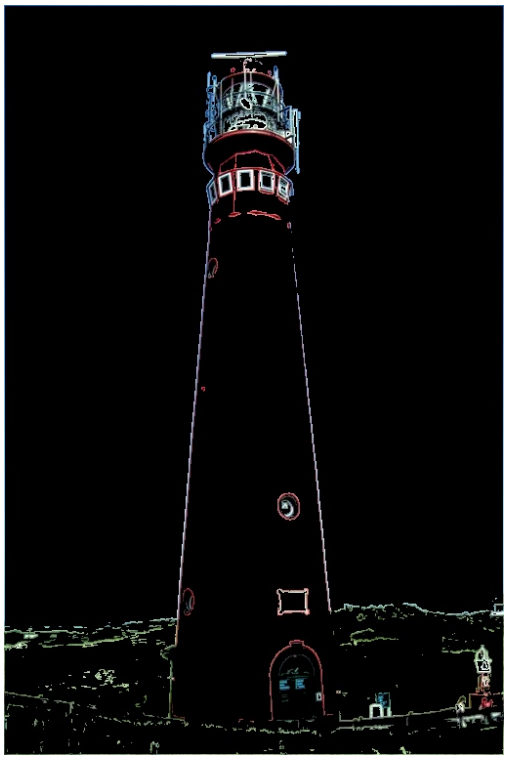


(a) Contour of image 1 Method 2

TEXTURE Results

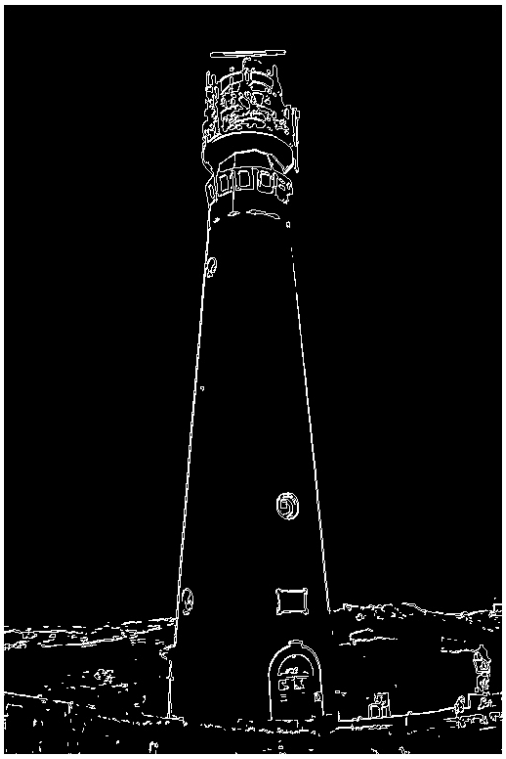


(a) Otsu mask for HSV image 1

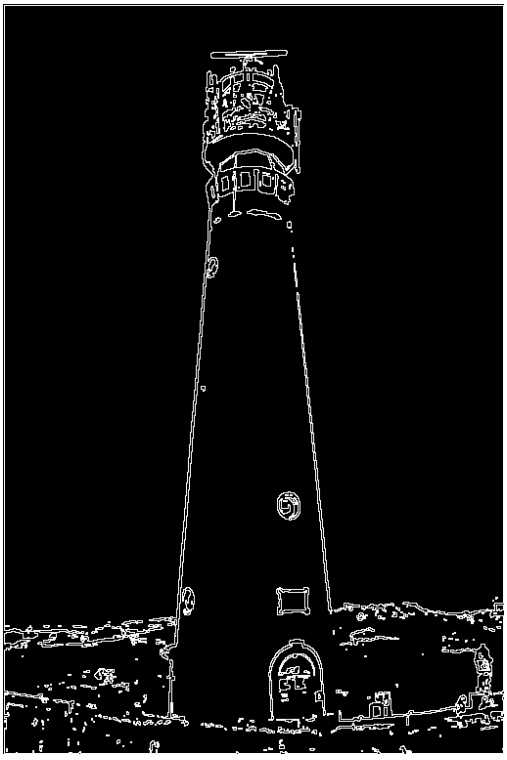


(b) Otsu masked Texture image 1

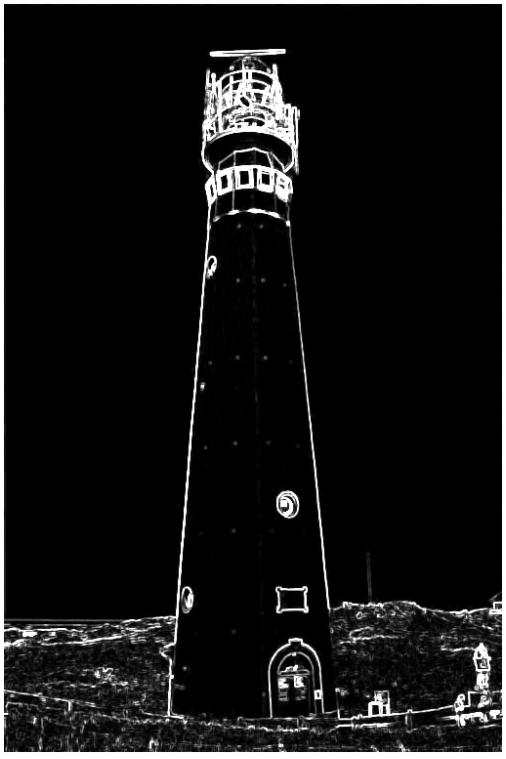
(a) Otsu masked Texture image 1 with closing



(b) Contour of image 1 Method 1



(a) Contour of image 1 Method 2



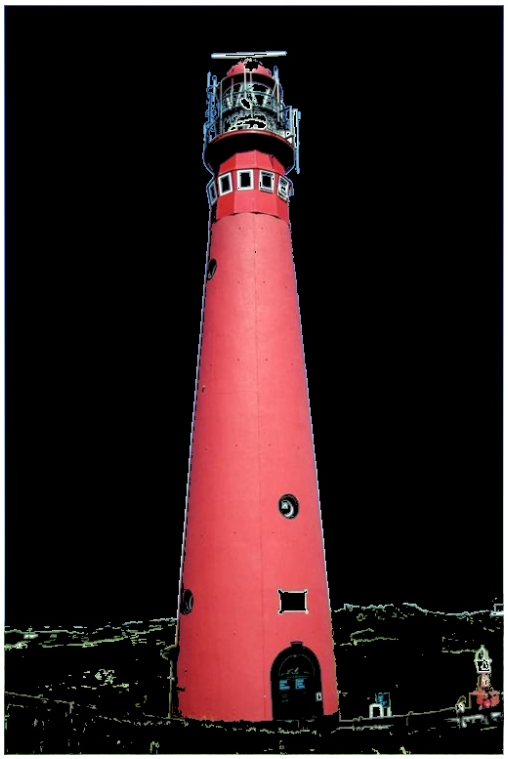
(b) Enhanced Texture at N = 3

(a) Enhanced Texture at N = 5

(b) Enhanced Texture at N = 7

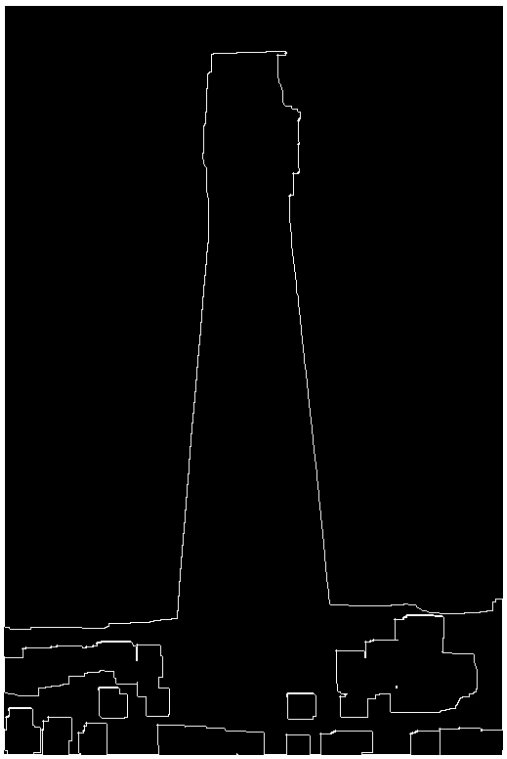
COMBINED MASK Results

(a) Otsu mask for Combined image 1

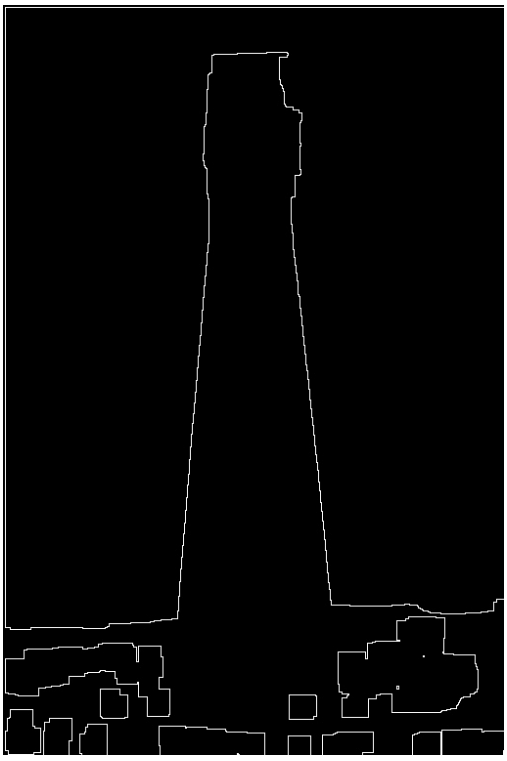


(b) Combined mask image 1

(a) Combined masks image 1 with closing



(b) Contour of image 1 Method 1



(a) Contour of image 1 Method 2

Results for image 2 (Baby)

BGR Results

(a) Otsu mask for BGR image 2

(b) Otsu masked BGR image 2

(a) Otsu masked BGR image 2 with closing

(b) Contour of image 2 Method 1

(a) Contour of image 2 Method 2

(b) Otsu Mask Channel 1

(a) Otsu Mask Channel 2

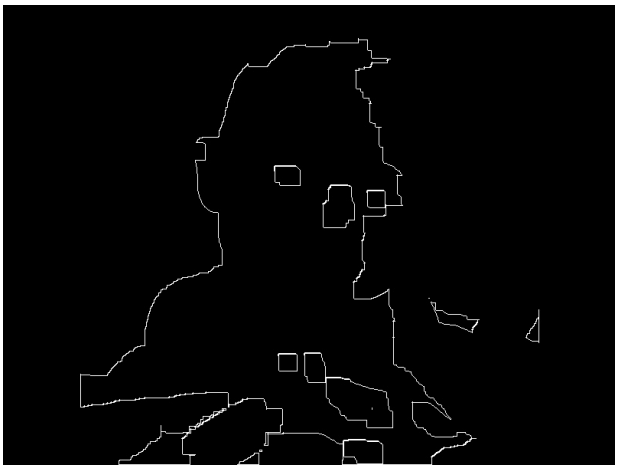
(b) Otsu Mask Channel 3

HSV Results

(a) Otsu mask for HSV image 2

(b) Otsu masked HSV image 2

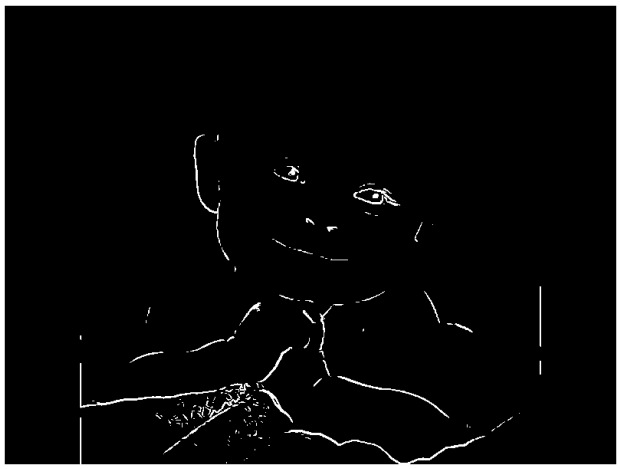
(a) Otsu masked HSV image 2 with closing



(b) Contour of image 2 Method 1

(a) Contour of image 2 Method 2

TEXTURE Results

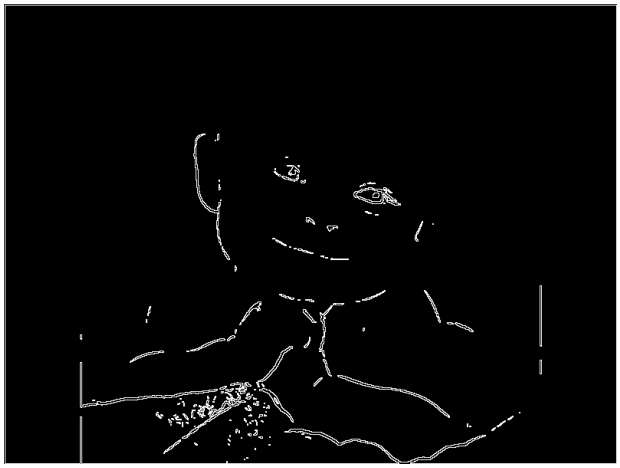


(a) Otsu mask for HSV image 2

(b) Otsu masked Texture image 2

(a) Otsu masked Texture image 2 with closing

(b) Contour of image 2 Method 1



(a) Contour of image 2 Method 2

(b) Enhanced Texture at N = 3

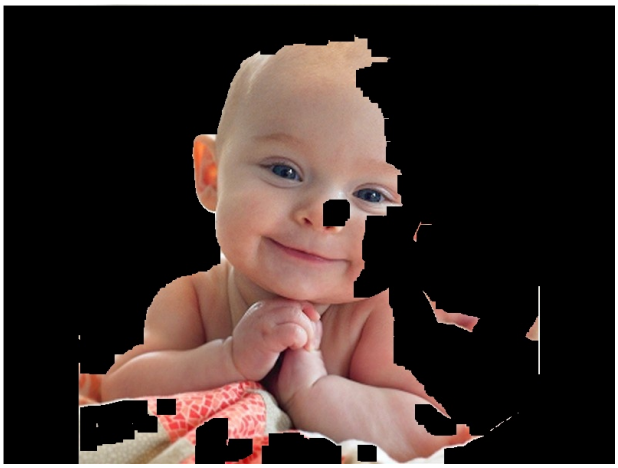
(a) Enhanced Texture at N = 5

(b) Enhanced Texture at N = 7

COMBINED MASK Results

(a) Otsu mask for Combined image 2

(b) Combined mask image 2



(a) Combined mask image 2 with closing

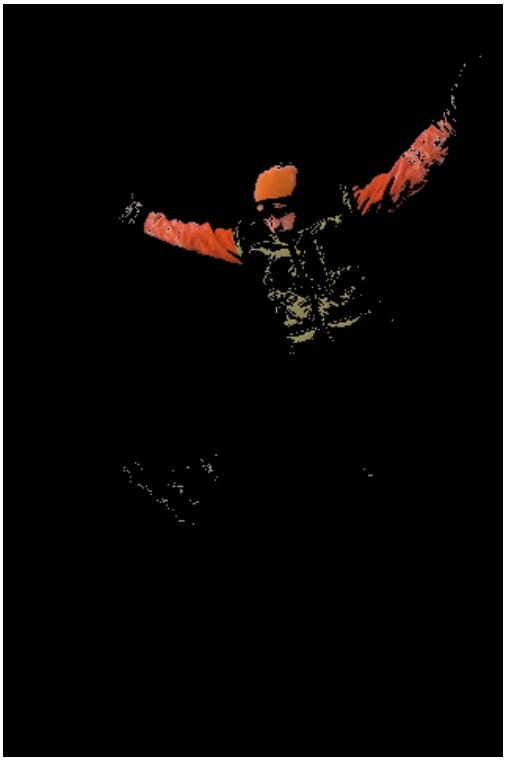
(b) Contour of image 2 Method 1

(a) Contour of image 2 Method 2

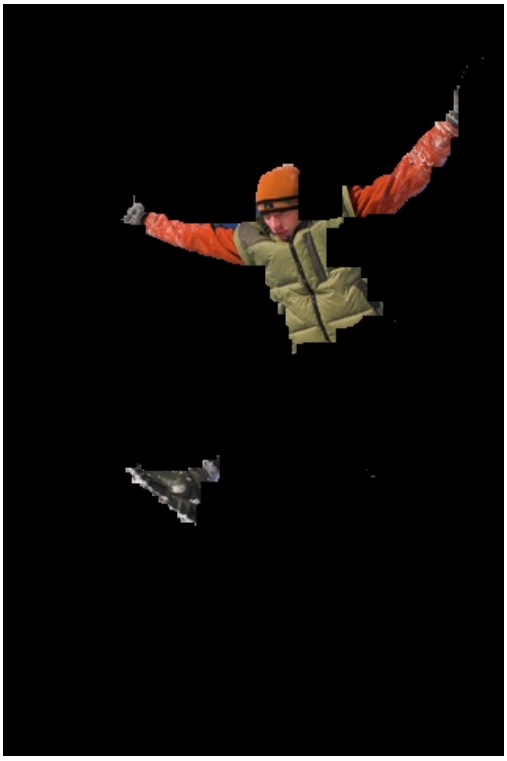
Results for image 3 (Skier)

BGR Results

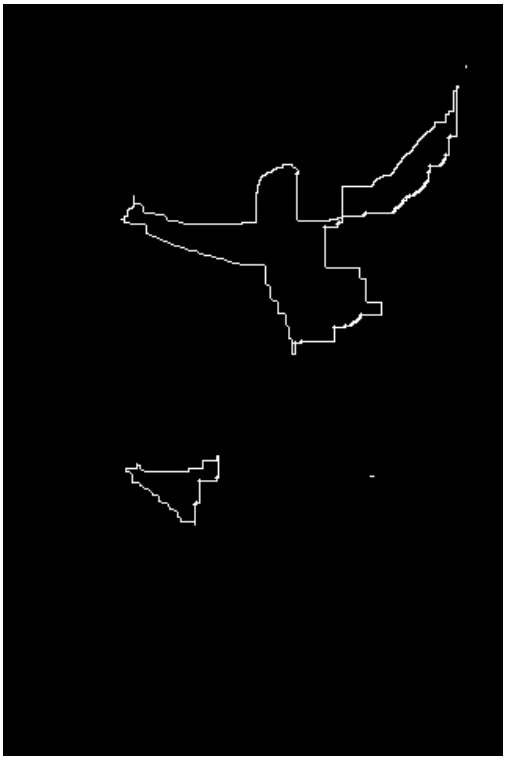
(a) Otsu mask for BGR image 3



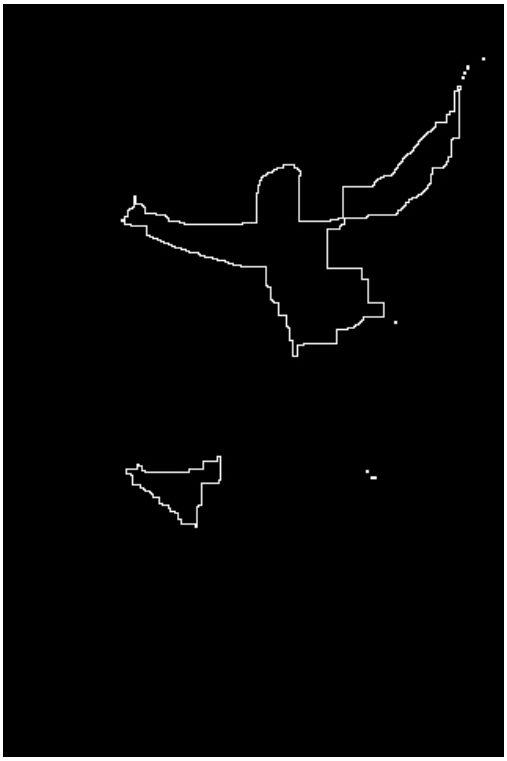
(b) Otsu masked BGR image 3



(a) Otsu masked BGR image 3 with closing



(b) Contour of image 3 Method 1



(a) Contour of image 3 Method 2

(b) Otsu Mask Channel 1

(a) Otsu Mask Channel 2

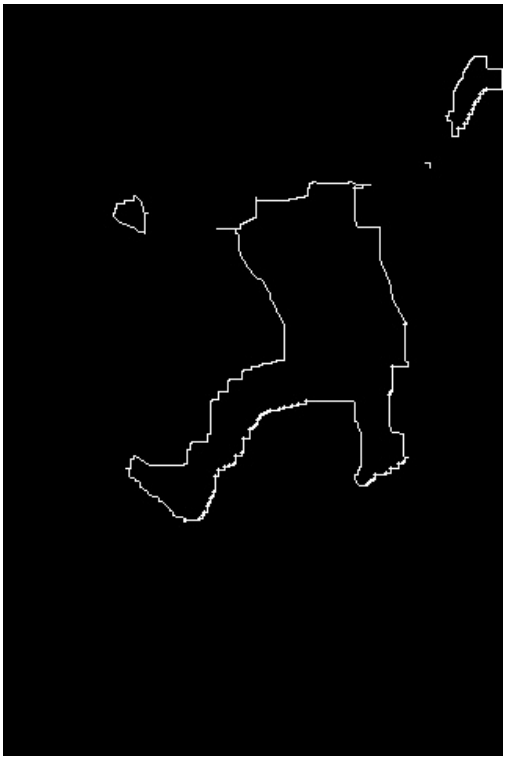
(b) Otsu Mask Channel 3

HSV Results

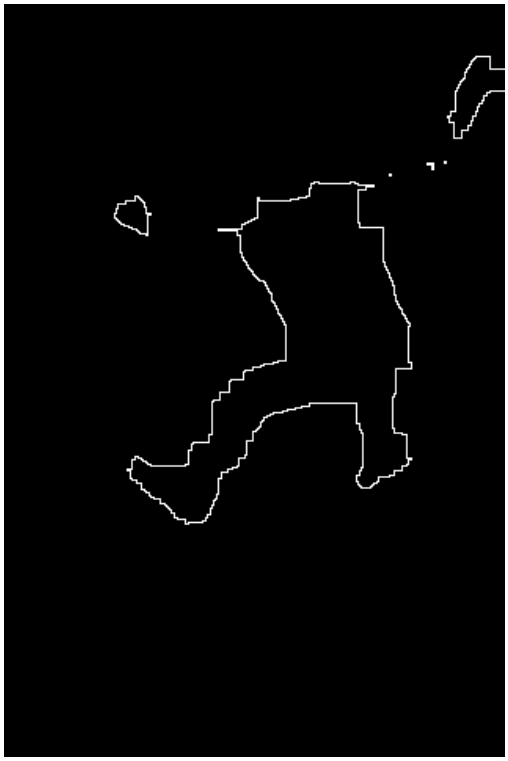
(a) Otsu mask for HSV image 3

(b) Otsu masked HSV image 3

(a) Otsu masked HSV image 3 with closing



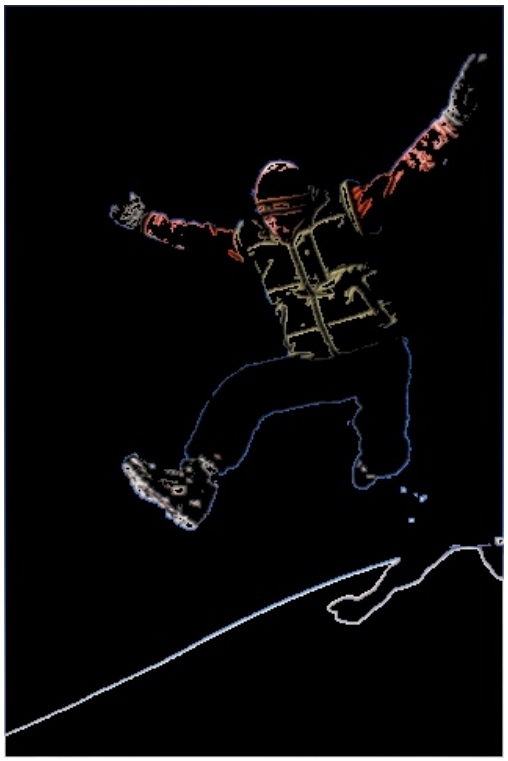
(b) Contour of image 3 Method 1



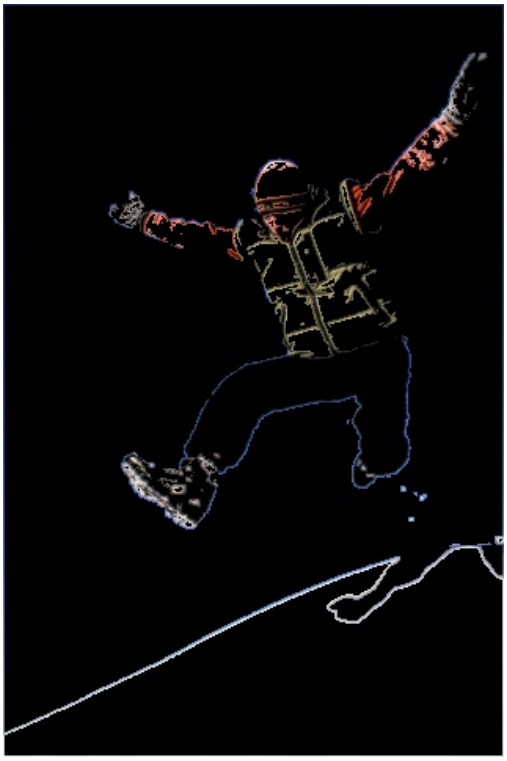
(a) Contour of image 3 Method 2

TEXTURE Results

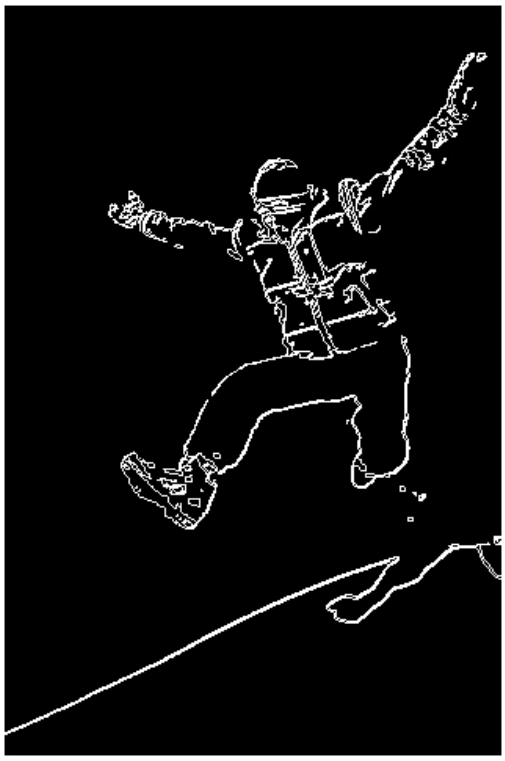
(a) Otsu mask for Texture image 3



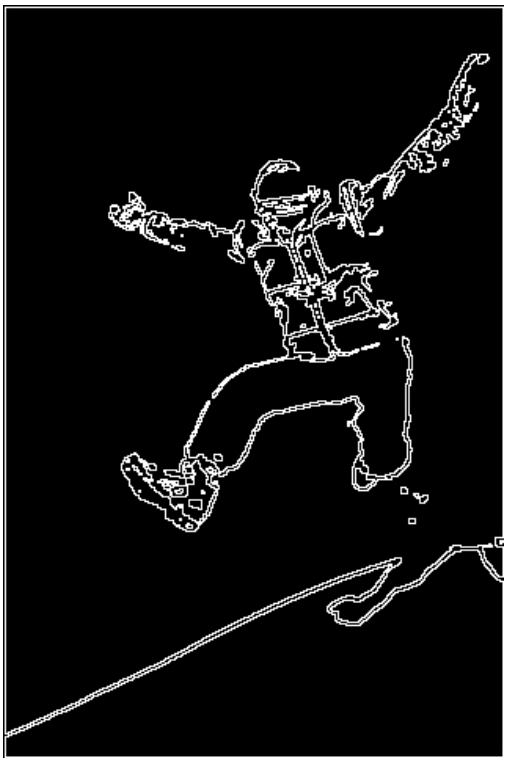
(b) Otsu masked Texture image 3



(a) Otsu masked Texture image 3 with closing

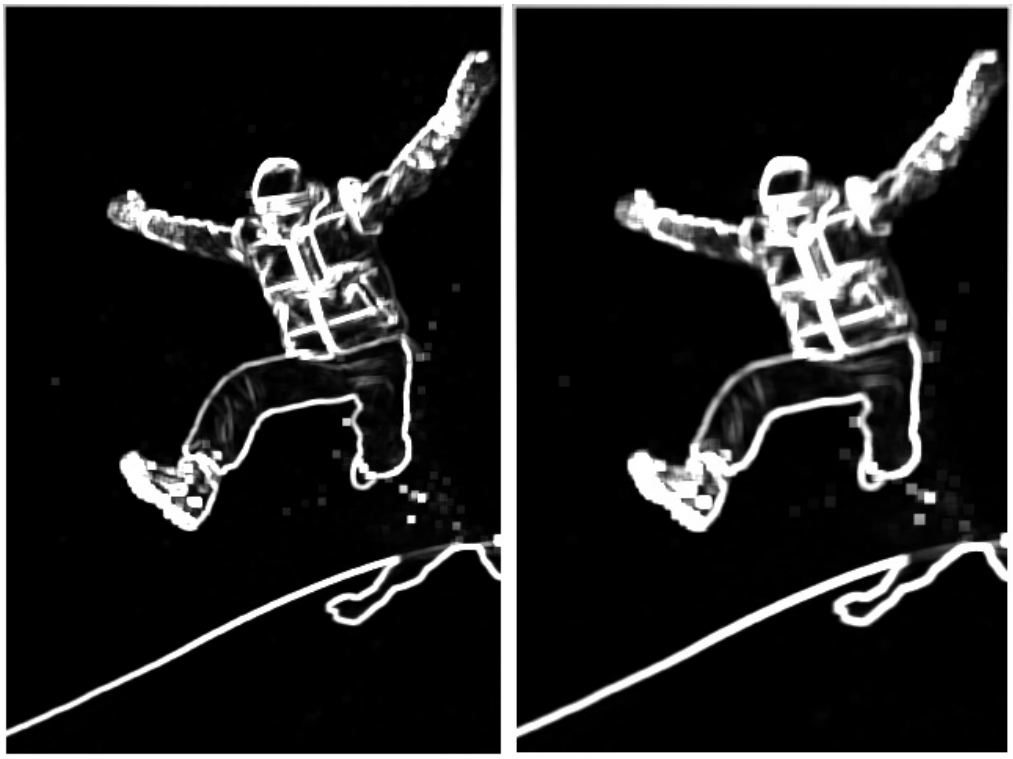


(b) Contour of image 3 Method 1



(a) Contour of image 3 Method 2

(b) Enhanced Texture at N = 3

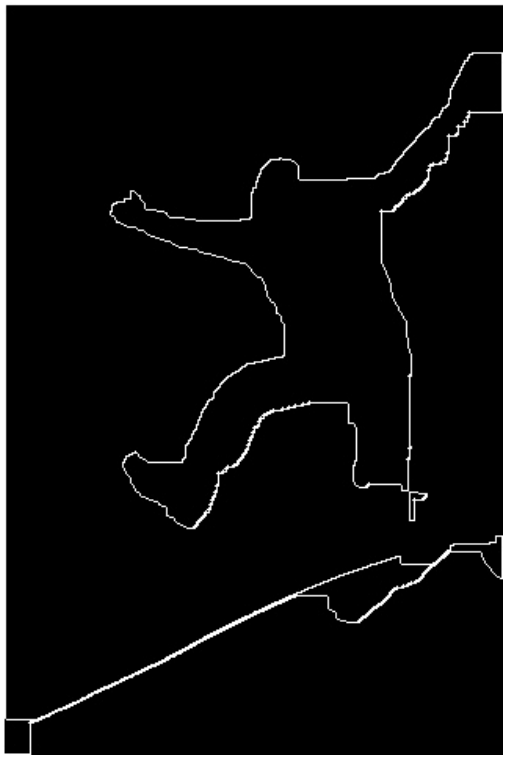


(a) Enhanced Texture at N = 5 (b) Enhanced Texture at N = 7

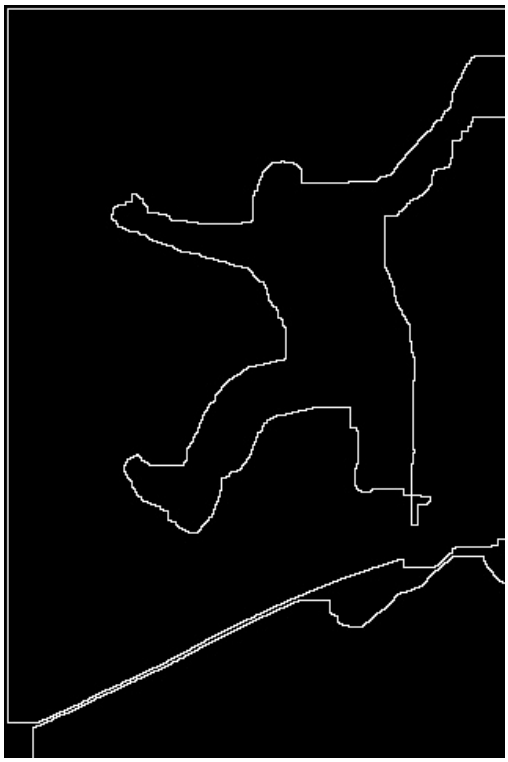
COMBINED MASK Results

(a) Otsu mask for Combined image 3 (b) Combined mask image 3

(a) Combined mask image 3 with closing



(b) Contour of image 3 Method 1



(a) Contour of image 3 Method 2

7. Codes

```
#!/usr/bin/env python3
################################################################################
# Script for homework 6 ECE661
# Image Segmentation file
#  - Contains functions to perform thresholding using Otsu's method.
#  - Functions to perform Otsu on multiple channels
#  - Functions for texutre based segmentation
#  - Function for contour extraction
# Author: Mythra Balakuntala
################################################################################

import cv2, numpy as np, math, time
import matplotlib.pyplot as plt
import time


class ImageSegment():
    def __init__(self):
        self.img_pth = {    1:'HW6Pics/lighthouse.jpg',
                            2:'HW6Pics/baby.jpg',
                            3:'HW6Pics/ski.jpg',
                            }
        self.ot_choice = {  1:np.array([[1,1,1],[0,0,1],[1,1,1]]),
                            2:np.array([[0,1,0],[0,0,0],[1,1,1]]),
                            3:np.array([[0,0,0],[0,0,1],[1,1,1]]),
                            }
# Histogram generation function
    def hist(self,img, gbins=256, scale = 255):
        #-------------------------------------------------------------
        # Intesity histogram generation function
        # Inputs: img - The image (1-channel), gbins - number of bins
        #   scale - scale of input data (0-255 for gray level image)
        # Outputs: hist_counts - Histogram counts
        #-------------------------------------------------------------
        hist_counts = np.zeros(gbins)
        for i in range(img.shape[0]):
            for j in range(img.shape[1]):
                hist_counts[int(img[i,j]/np.ceil(scale/gbins))] += 1
        return(hist_counts)

# Otsu threshold function
    def otsu(self, img, gbins=256, foreg = 0):
        #-------------------------------------------------------------
        # Implementation of otsu's method to maximize the inter-class
        # variance.
        # Inputs: img - The image (1-channel), gbins - number of bins,
        #   foreg - foreground indicator.. 0:lower than thres,
        #    1: greater than thres
        # Outputs: Otsu thresholded image
        #-------------------------------------------------------------
        # Generate histogram bins
        res_img = np.ones(img.shape)
        N = img.shape[0]*img.shape[1]  #total number of pixels
        k= 0
        print('Bins, pixels = ',gbins, N)
```

```
        hist_counts = self.hist(img, gbins)/N
        wk = 0
        mk = 0
        mt = sum((np.arange(gbins)+1)*hist_counts)
        max_sb = 0.0;
        for i in range(1,gbins+1):
            wk += hist_counts[i-1]
            mk += i*hist_counts[i-1]
            if(wk == 0 or (1-wk)==0):
                continue
            sb = ((mt*wk - mk)**2)/(wk*(1-wk));
            if ( sb >= max_sb ):
                k = i
                max_sb = sb
        print('Otsu threshold is ',k)
        if foreg:
            res_img[np.where(img<=k)] = 0
        else:
            res_img[np.where(img>=k)] = 0
        return(res_img)

# Multi channel Otsu threshold function
    def multi_otsu(self, img, type=1, fchoice=np.array([])):
        #-------------------------------------------------------------
        # Otsu for multi channel inputs, the channels are combined
        #    based on type.
        # Inputs: img - The image (with any number of channels)
        #   type - 1: If no. of channels > 1 then logical and is used.
        #          2: If no. of channels > 1 iteratively threshold.
        # Outputs: Otsu thresholded image
        # Uses the otsu function to compute thresholded image
        #-------------------------------------------------------------
        if fchoice.shape[0] == 0:
            fchoice = np.ones(img.shape[2])
        rec_img = img
        if(len(img.shape)==2):      # If only one channel
            return(self.otsu(img))
        elif type==1:               # Logical and each channel otsu
            res_img = np.ones(img.shape[:2])    # Resultant image
            # Array to store otsu for each channel
            thres_img = np.zeros(img.shape)
            for i in range(img.shape[2]):
                thres_img[:,:,i] = self.otsu(img[:,:,i],foreg = fchoice[i])    # Comput otsu
                # And with result until prev channel
                res_img = thres_img[:,:,i]*res_img
                # self.viewimg(thres_img[:,:,i])      # View each channel result ***
                # cv2.imwrite(('img' + str(3) +'_ot'+str(i+1)+'_rgb.jpg'),(255\
                # *thres_img[:,:,i]).astype(np.uint8))
            return(res_img.astype(np.uint8))
        elif type==2:               # Iteratively apply otsu prev channel result
            # Array to store otsu for each channel
            thres_img = np.zeros(img.shape)
            thres_img[:,:,0] = self.otsu(img[:,:,0],foreg = fchoice[0])  # For first channel
            # self.viewimg(thres_img[:,:,0])
            rec_img = (thres_img[:,:,0])[:,:,np.newaxis]*img
            # self.viewimg(rec_img.astype(np.uint8))
```

```
            for i in range(1,img.shape[2]):
                # Compute otsu on next channel based on foreground of prev.
                self.viewimg(thres_img[:,:,i-1]*rec_img[:,:,i])
                thres_img[:,:,i] = self.otsu(thres_img[:,:,i-1]*rec_img[:,:,i],foreg = fchoice[i])
                rec_img = ((thres_img[:,:,i])[:,:,np.newaxis]*rec_img).astype(np.uint8)
                # rec_img[:,:,i] = ((thres_img[:,:,i])*rec_img[:,:,i]).astype(np.uint8)
                self.viewimg(rec_img)
                self.viewimg(thres_img[:,:,i])
            return(thres_img[:,:,i])

    # Funtion to extract contours
        def img_contour(self, img):
            #------------------------------------------------------------
            # Contour extraction function
            # Inputs: img - The binary image
            # Outputs: Binary image with contour
            # Searches clockwise from any white pixel
            #------------------------------------------------------------
            bpix = np.array([[-1,-1]])  # Array to store boundary pixels
            pimg = np.zeros((img.shape[0]+2,img.shape[1]+2))
            pimg[1:-1,1:-1] = img
            clk_ind = np.array([[-1,-1],[-1,0],[-1,1],[0,1],[1,1],[1,0],[1,-1],[0,-1]])
            for i in range(img.shape[0]):
                for j in range(img.shape[1]):
                    bpixq = False   # Indicator for boundary pixel
                    # Check if it is a boundary pixel
                    if img[i,j] == 1:
                        for m in range(8):
                            # If one of the neighbours is black then boundary pixel
                            if pimg[i+1+clk_ind[m,0],j+1+clk_ind[m,0]] == 0:
                                bpixq = True
                                break
                    # 3 Checks, 1: if white pixel, 2: if boundary pixel,
                    #   3: if already in one of the boundary pixels
                    if(bpixq and sum(np.all((bpix-[i,j])==0, 1))==0):
                        i1, j1 = i, j
                        pk = 0  # position of previous bpix
                        while True:
                            for m in range(8):
                                k = (m+pk+1)%8
                                # Check if pixel is white, if so step to it
                                if pimg[i1+1+clk_ind[k,0],j1+1+clk_ind[k,1]] == 1:
                                    # add it to boundary pixels
                                    bpix = np.vstack((bpix, [i1+clk_ind[k,0],j1+clk_ind[k,1]]))
                                    # Update position of current bpix
                                    i1 = i1+clk_ind[k,0]
                                    j1 = j1+clk_ind[k,1]
                                    pk = (k+4)%8    # Start index for next pixel
                                    break
                            # On returning to start pixel break
                            if i1 == i and j1 == j:
                                break
            bpix = bpix[1:,:]   # Remove first row of [-1,-1]
            # Compose result image
            res_img1 = np.zeros(img.shape)
            res_img1[bpix[:,0],bpix[:,1]]=1 # set boundary pixels to 1
```

```
        # Method 2: Dilates the image and subtracts the eroded image
        ker = np.ones((2,2),np.uint8)
        res_img2 = cv2.dilate(img,ker,iterations=1) - cv2.erode(img,ker,iterations=1)
        return(res_img1, res_img2)

# Kernel Variance function
    def ker_var(self,img, N):
        #-------------------------------------------------------------
        # Computes variance for texture segmentation
        # Inputs: img, Kernel size
        # Outputs: array with variance at each pixel
        #-------------------------------------------------------------
        pimg = np.zeros((img.shape[0] + N-1,img.shape[1] + N-1))    # Padded image
        pimg[int(N/2):-int(N/2),int(N/2):-int(N/2)] = img
        var_img = np.zeros(img.shape)
        for i in range(img.shape[0]):
            for j in range(img.shape[1]):
                var_img[i,j] = pimg[i:i+N,j:j+N].var() #var = mean(abs(x - x.mean())**2)
        #Scale to 255
        var_img = (5000*var_img/var_img.max())   #var is 0-max
        var_img[np.where(var_img>255)] =255     # Scale appropriately
        return(var_img)

# Texutre based segementation function
    def tex_seg(self,img):
        #-------------------------------------------------------------
        # Texture segmentation uses 3 kernel sizes 3,5,7 and uses variance
        #   of gray level histograms to generate 3 channels
        # Then Otsu is applied on this to obtain the resulting image
        # Inputs: img - Gray image
        # Outputs: Three channel variance image
        #-------------------------------------------------------------
        t_img = np.zeros((img.shape[0],img.shape[1],3))
        for i in range(3):
            t_img[:,:,i] = self.ker_var(img,2*i+3)
        return(t_img)

# Function to view image
    def viewimg(self, img):
        cv2.namedWindow('resultImage',cv2.WINDOW_NORMAL)
        cv2.imshow('resultImage',img)
        cv2.resizeWindow('resultImage', 600,600)
        key = 1
        print('Press q to quit')
        while key !=ord('q'):       # Quit on pressing q
            key = cv2.waitKey(0)
        cv2.destroyAllWindows()

# Main function
    def main(self, img_id):
        #-------------------------------------------------------------
        # Calls otsu, texture segmentation and contour extraction
        # Inputs: img_id - Id for the first image
        # Outputs: None ,
        # Writes output images from otsu, texture segmentation and
```

```
# contour extraction to the folder
#-------------------------------------------------------
# Initialize images
img_rgb = cv2.imread(self.img_pth[img_id])  # RGB
img_gry = cv2.cvtColor(img_rgb,cv2.COLOR_BGR2GRAY)  # GRAY
img_hsv = cv2.cvtColor(img_rgb,cv2.COLOR_BGR2HSV)   # HSV
img_txt = self.tex_seg(img_gry) # Variance texture image
cv2.imwrite(('img' + str(img_id) +'_txt1.jpg'),(img_txt[:,:,0]).astype(np.uint8))
cv2.imwrite(('img' + str(img_id) +'_txt2.jpg'),(img_txt[:,:,1]).astype(np.uint8))
cv2.imwrite(('img' + str(img_id) +'_txt3.jpg'),(img_txt[:,:,2]).astype(np.uint8))
# Color threshodling for segmentation-------------------------------
ker = np.ones((20,20),np.uint8) # kernel for filling holes
# Perform multi otsu for HSV image
res_img = self.multi_otsu(img_hsv,1,self.ot_choice[img_id][0,:]).astype(np.float)
res_hsv = res_img #store
cv2.imwrite(('img' + str(img_id) +'_omask_hsv.jpg'),(255*res_img).astype(np.uint8))
cv2.imwrite(('img' + str(img_id) +'_hsv.jpg'),(res_img[:,:,np.newaxis]\
*img_rgb).astype(np.uint8))
# Dilate and erode image to remove holes
res_img = cv2.morphologyEx(res_img,cv2.MORPH_CLOSE, ker)
cv2.imwrite(('img' + str(img_id) +'_ed_hsv.jpg'),(res_img[:,:,np.newaxis]\
*img_rgb).astype(np.uint8))
c_img1,c_img2 = self.img_contour(res_img)   # Get contours
cv2.imwrite(('img' + str(img_id) +'c1_hsv.jpg'),255*c_img1)
cv2.imwrite(('img' + str(img_id) +'c2_hsv.jpg'),255*c_img2)
#-----------------------------------------------------------------------
# On BGR images
# Perform multi otsu for rgb image
res_img = self.multi_otsu(img_rgb,1,self.ot_choice[img_id][1,:]).astype(np.float)
res_rgb = res_img #store
cv2.imwrite(('img' + str(img_id) +'_omask_rgb.jpg'),(255*res_img).astype(np.uint8))
cv2.imwrite(('img' + str(img_id) +'_rgb.jpg'),(res_img[:,:,np.newaxis]\
*img_rgb).astype(np.uint8))
# Dilate and erode image to remove holes
res_img = cv2.morphologyEx(res_img,cv2.MORPH_CLOSE, ker)
cv2.imwrite(('img' + str(img_id) +'_ed_rgb.jpg'),(res_img[:,:,np.newaxis]\
*img_rgb).astype(np.uint8))
c_img1,c_img2 = self.img_contour(res_img)   # Get contours
cv2.imwrite(('img' + str(img_id) +'c1_rgb.jpg'),255*c_img1)
cv2.imwrite(('img' + str(img_id) +'c2_rgb.jpg'),255*c_img2)
#-----------------------------------------------------------------------
# On Texture images
# Perform multi otsu for txt image
res_img = self.multi_otsu(img_txt,1,self.ot_choice[img_id][2,:]).astype(np.float)
res_txt = res_img
cv2.imwrite(('img' + str(img_id) +'_omask_txt.jpg'),(255*res_img).astype(np.uint8))
cv2.imwrite(('img' + str(img_id) +'_txt.jpg'),(res_img[:,:,np.newaxis]\
*img_rgb).astype(np.uint8))
# Dilate and erode image to remove holes
# res_img = cv2.morphologyEx(res_img,cv2.MORPH_CLOSE, ker)
cv2.imwrite(('img' + str(img_id) +'_ed_txt.jpg'),(res_img[:,:,np.newaxis]\
*img_rgb).astype(np.uint8))
c_img1,c_img2 = self.img_contour(res_img)   # Get contours
cv2.imwrite(('img' + str(img_id) +'c1_txt.jpg'),255*c_img1)
cv2.imwrite(('img' + str(img_id) +'c2_txt.jpg'),255*c_img2)
#-----------------------------------------------------------------------
```

```
        # On Combined masks
        # Or the results
        res_img = np.logical_or(np.logical_or(res_hsv,res_rgb).astype(np.uint8)\
        ,res_txt).astype(np.uint8)
        print(res_img)
        cv2.imwrite(('img' + str(img_id) +'_omask_cmb.jpg'),(255*res_img)\
        .astype(np.uint8))
        cv2.imwrite(('img' + str(img_id) +'_cmb.jpg'),(res_img[:,:,np.newaxis]\
        *img_rgb).astype(np.uint8))
        # Dilate and erode image to remove holes
        res_img = cv2.morphologyEx(res_img,cv2.MORPH_CLOSE, ker)
        cv2.imwrite(('img' + str(img_id) +'_ed_cmb.jpg'),(res_img[:,:,np.newaxis]\
        *img_rgb).astype(np.uint8))
        c_img1,c_img2 = self.img_contour(res_img)   # Get contours
        cv2.imwrite(('img' + str(img_id) +'c1_cmb.jpg'),255*c_img1)
        cv2.imwrite(('img' + str(img_id) +'c2_cmb.jpg'),255*c_img2)


if __name__ == '__main__':
    imgseg = ImageSegment()
    imgseg.main(1)  # imgseg.main(img id)
    imgseg.main(2)
    imgseg.main(3)
```