ECE 661: Computer Vision

Homework 6, Fall 2018

Arindam Bhanja Chowdhury
abhanjac@purdue.edu

1 Overview

This homework deals with image segmentation. Fig 1, 2 and 3 are having an object on a background and the task is to extract the object from the background. Subsequently the contour of the segmented output should also be extracted.

Figure 1: Reference image 1 of homework.

Figure 2: Reference image 2 of homework.

Figure 3: Reference image 3 of homework.

The foreground for the Fig 1 is the red lighthouse, the foreground of the Fig 2 is the head and body of the baby and that of Fig 3 is the jumping man.

2 Otsu's Algorithm

Let the pixels of a given picture be represented in L gray levels [1, 2, ..., L]. The number of pixels at the level i is denoted by n_i and the total number of pixels is N. So the probability distribution for the pixels is

$$p_i = n_i/N \tag{1}$$

Now suppose that we dichotomize the pixels into two classes C_0 and C_1 (background and object respectively) by a threshold at kevel k. C_0 denotes the pixels denotes pixels with levels [1, 2, ..., k] and C_1 denotes the pixels denotes pixels with levels [k+1, 2, ..., L]. Then the probabilites of class occurrence and the class mean levels, respectively, are given by

$$\omega_0 = Pr(C_0) = \sum_{i=1}^{k} p_i = \omega(k) \tag{2}$$

$$\omega_1 = Pr(C_1) = \sum_{i=1}^{k} p_i = 1 - \omega(k) \tag{3}$$

and

$$\mu_0 = Pr(i|C_0) = \sum_{i=1}^{k} ip_i/\omega_0 = \mu(k)/\omega(k) \tag{4}$$

$$\mu_1 = Pr(i|C_1) = \sum_{i=1}^{k} ip_i/\omega_1 = \frac{\mu_T - \mu(k)}{1 - \omega(k)} \tag{5}$$

where

$$\omega(k) = \sum_{i=1}^{k} p_i \tag{6}$$

and

$$\mu(k) = \sum_{i=1}^{k} ip_i \tag{7}$$

and

$$\mu_T = \sum_{i=1}^{L} ip_i \tag{8}$$

which is the total mean level of the original picture. We can easily verify the following relation for any choice of k

$$\omega_0\mu_0 + \omega_1\mu_1 = \mu_T, \quad \omega_0 + \omega_1 = 1 \tag{9}$$

The class variance are given by

$$\sigma_0^2 = \sum_{i=1}^{k} (i - \mu_0)^2 Pr(i|C_0) = \sum_{i=1}^{k} (i - \mu_0)^2 p_i/\omega_0 \tag{10}$$

$$\sigma_1^2 = \sum_{i=1}^{k} (i - \mu_1)^2 Pr(i|C_1) = \sum_{i=1}^{k} (i - \mu_1)^2 p_i/\omega_1 \tag{11}$$

These require second-order cumulative moments (statistics). In order to evaluate the "goodness" of th threshold (at level k) we shall introduce the following discriminant criterion measures (or measures of class separability).

$$\lambda = \sigma_B^2/\sigma_W^2, \quad \kappa = \sigma_T^2/\sigma_W^2, \quad \eta = \sigma_B^2/\sigma_T^2 \tag{12}$$

where

$$\sigma_W^2 = \omega_0\sigma_0^2 + \omega_1\sigma_1^2 \tag{13}$$

$$\sigma_B^2 = \omega_0(\mu_0 - \mu_T)^2 + \omega_1(\mu_1 - \mu_T)^2 = \omega_0\omega_1(\mu_1 - \mu_0)^2 \tag{14}$$

and

$$\sigma_T^2 = \sum_{i=1}^{L}(i - \mu_T)^2 p_i \tag{15}$$

are the within-class variance, the between-class variance and the total variance of levels, respectively. Then our problem is reduced to an optimization problem to search for a threshold k that maximizes one of the object functions (the criterion measures) in (12)

This standpoint is motivated by a conjecture that well-thresholded classes would be separated would be separated in gray levels, and conversely, a threshold giving the best separation of classes in gray levels would be the best threshold.

The discriminant criteria maximizing λ, κ and η, respectively, for k are, however, equivalent to one another; e.g. $\kappa = \lambda + 1$ and $\eta = \lambda/(\lambda + 1)$ in terms of λ, because the following basic relation always holds:

$$\sigma_W^2 + \sigma_B^2 = \sigma_T^2 \tag{16}$$

It is noticed that σ_W^2 and σ_B^2 are functions of threshold level k, but σ_T^2 is independent of k. It is also noted that σ_W^2 is based on the second order statistics while σ_B^2 is based on first order statistics. So σ_B^2 is the simplest measure with respect to k. Thus we adopt η as the criterion measure to evaluate the "goodness" (or separability) of the threshold at level k.

The optimal threshold k* that maximizes η is thus obtained as

$$\eta(k) = \sigma_B^2(k)/\sigma_T^2 \tag{17}$$

$$\sigma_B^2 = \frac{[\mu_T\omega(k) - \mu(k)]^2}{\omega(k)[1 - \omega(k)]} \tag{18}$$

and the optimal threshold k* is

$$\sigma_B^2(k*) = max_{1<=k<L}\ \sigma_B^2(k) \tag{19}$$

So, in the end we have to go through each and every k in the histogram and find the k for which the σ_B^2 between-class variance will be maximum.

When applying otsu's algorithm to an RGB image, the algorithm is applied separately to the three channels of the image and then the results of the final segmentation is obtained by logically combining them. Otsu's algorithm is also applied in an iterative manner in which it is applied to the foregroun returned by the previous iteration, so as to have better results.

3 Texture Based Segmentation

Another approach to image segmentation consists of characterizing the pixels with texture-based features and then applying otsu's algorithm to these features. Although there not exist different kinds of texture operators, we are only implementing the simplest possible approach for this homework. As we scan an image, we place a window of size NxN at each pixel and compute the variance of the pixels in the window. This is done for at least three values of N, (N=3, 5, 7). The three variances together at each pixel would constitute a texture measure at that pixel. Now, for the segmentation, this is treated as a 3-dimensional characterization of a pixel just as the pixels of the RGB image was treated.

4 Contour Extraction

After the segmentation is done, the contour can be extracted from the image. This contour only separates the foreground from the background. Since the final segmented image is formed by the boolean combination of the different channels, so it has only values of 0 and 1. So this can be used as a mask. This contour algorithm implemented here is with 8-neighbors points. The foreground are the pixels whose values equal to 1, while the background are the pixels which are 0 in the overall mask.

For each pixel in the mask or segmented image, if the pixel is 0, then it is not selected for the contour. If the pixel is 1, and all of its 8 neighbors are 1, then also it is not selected as a contour point. As this indicates that a point is entirely inside the foreground and not on the boundary of the foreground and background. Only if the pixel value is 1, and not all of its 8 neighbors are 1, then only it is selected as a contour point.

5 Procedure for RGB Segmentation

- The channels of the image are separated, they are slightly blurred by a gaussian blur and then their individual histograms are calculated.
- The between class variance of the channels are calculated for each k value as per the equations mentioned in the theory in the earlier sections.
- The k value that creates the highest between class variance is used to threshold the image.
- For the image in Fig 1, the final channnels are combined as follows: $R \; and \; \bar{B} \; and \; \bar{(}G)$. The red segment mask is combined with the inverted blue and green segment masks by logical AND operation. This is because the foreground if primarily red.
- For image in Fig 3, the otsu's algorithm is applied twice iteratively on the blue channel to get the good segmentation. Before every iteration the blue channel is inverted as well. This is because it is observed that the foreground is in the region that has blue value lower than the threshold obtained from otsu. The red segment mask from iteration 1 are combined with these inverted blue segment mask from iteration 2 by logical OR operation to get an intermediate mask. Then green and red segment masks from the iteration 1 are combined with logical AND to get a new mask. This mask only has the region that has the snow. This new mask is then combined with the intermediate mask obtained in the previous step to get a good segment. The operation is like the following $[R \; or \; \bar{B_{iter2}}] \; xor \; [G \; and \; R]$.

- For the image in Fig 2, the red, blue and green filtered masks are all inverted and combined together by logical OR operation like this: $\bar{R}$ *or* $\bar{G}$ *or* $\bar{B}$.
- Contours are then extracted using the 8 neighbor method as mentioned in the theory.

6 Results of RGB Segmentation

Figure 4: RED filtered mask after applying otsu's algorithm on Fig 1 image.

Figure 5: GREEN filtered mask after applying otsu's algorithm on Fig 1 image.

Figure 6: BLUE filtered mask after applying otsu's algorithm on Fig 1 image.

Figure 7: Combined image after RGB segmentation on Fig 1 image.

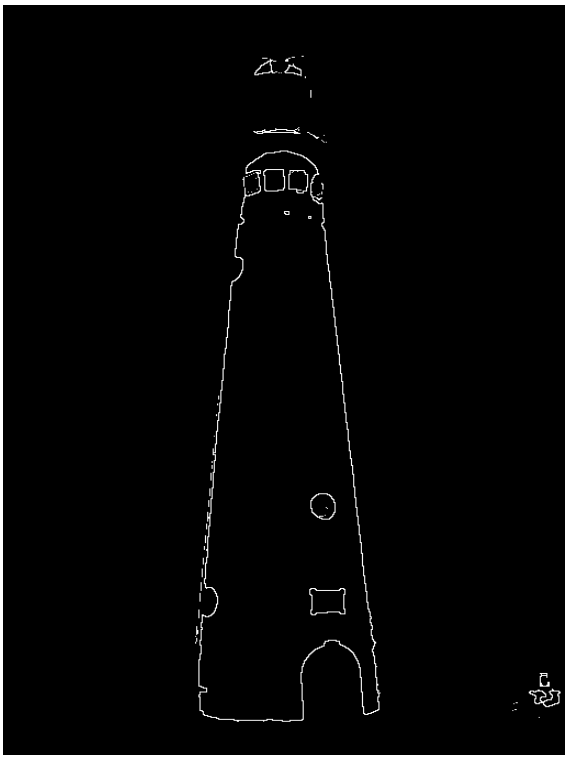

Figure 8: Contour image obtained from RGB segmented Fig 1 image.

Figure 9: RED filtered mask after applying otsu's algorithm on Fig 2 image.

Figure 10: GREEN filtered mask after applying otsu's algorithm on Fig 2 image.

Figure 11: BLUE filtered mask after applying otsu's algorithm on Fig 2 image.

Figure 12: Combined image after RGB segmentation on Fig 2 image.

Figure 13: Contour image obtained from RGB segmented Fig 2 image.

Figure 14: RED filtered mask after applying otsu's algorithm on Fig 3 image.

Figure 15: GREEN filtered mask after applying otsu's algorithm on Fig 3 image.

Figure 16: BLUE filtered mask after applying otsu's algorithm on Fig 3 image.

Figure 17: BLUE filtered mask after 2nd iteration of otsu's algorithm on Fig 3 image.

Figure 18: BLUE inverted filtered mask after 2nd iteration of otsu's algorithm on Fig 3 image.

Figure 19: Combined image after RGB segmentation on Fig 3 image.

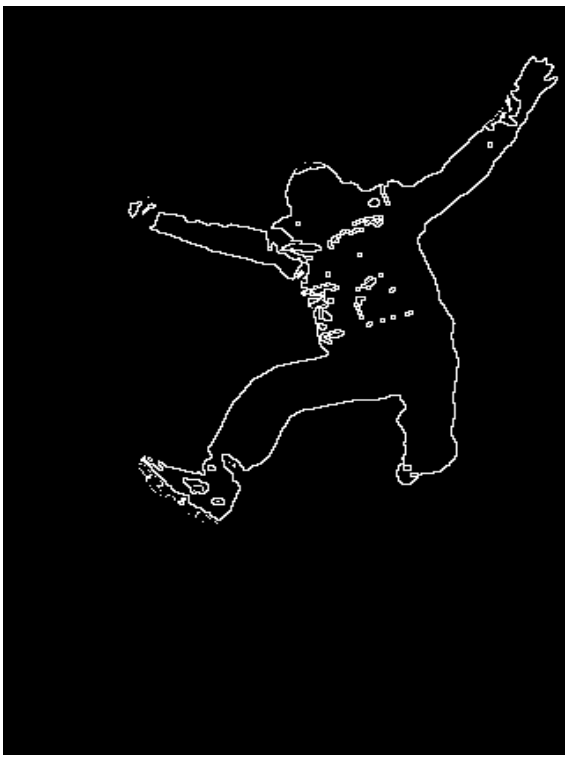

Figure 20: Contour image obtained from RGB segmented Fig 3 image.

7 Procedure for Texture Segmentation

- The images are converted to grayscale.
- A 3x3, 5x5 and 7x7 kernel is constructed and the variance in these neighborhoods around each pixel is calculated.
- Each kind of kernel creates a separate channel. Now, since the variance is a square term, so the pixel values of these channels are not 255, but some very high values of the order of 255*255. So, all these channels are normalized, so that the values of all the pixels are between 0 to 255 range.
- These channels are then combined together and segmented in the same manner as done by otsu in case of the RGB images.
- The image in Fig 1 is only iterated 3 times on all the channels.
- The images in Fig 2 and 3 have to be iterated 4 times on all the channels.
- The filtered masks are then combined by logical OR operations to get the final segmented image.
- Contours are then extracted using the 8 neighbor method as mentioned in the theory.
- The thickness of the contour is also affected by the gaussian blur done before applying otsu's algorithm.

8 Results of Texture Segmentation

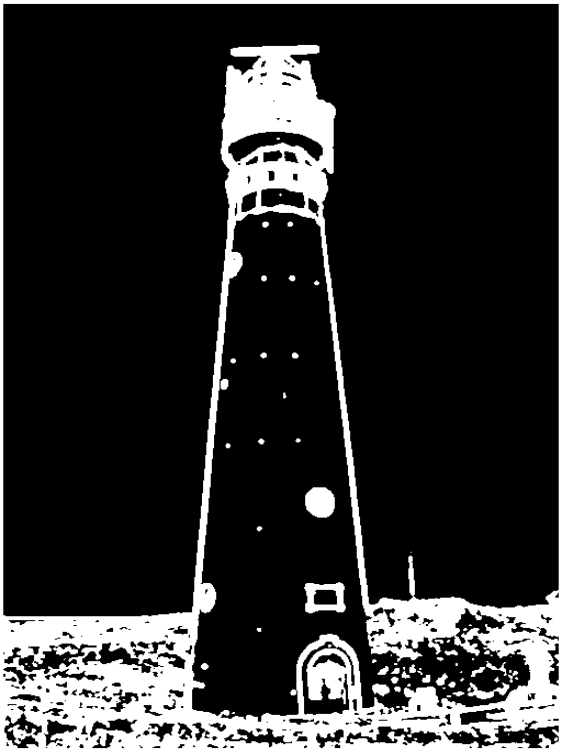

Figure 21: 3x3 filtered mask after 3rd iteration of otsu's algorithm and normalization on Fig 1 image.

Figure 22: 5x5 filtered mask after 3rd iteration of otsu's algorithm and normalization on Fig 1 image.

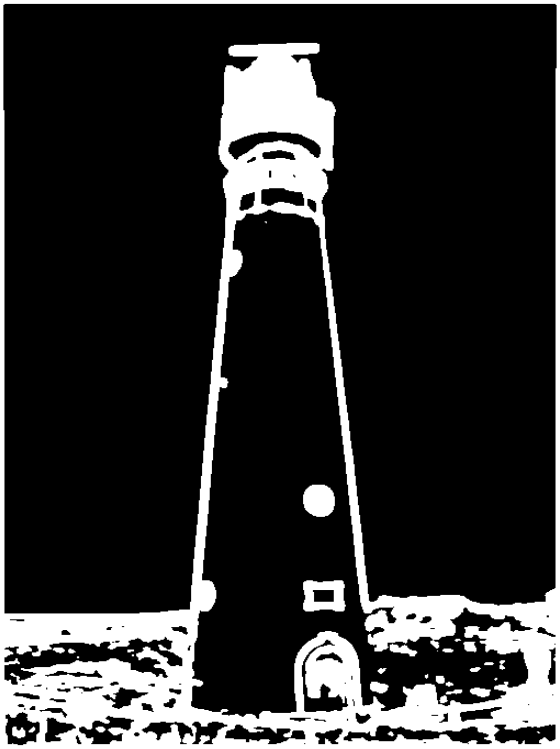

Figure 23: 7x7 filtered mask after 3rd iteration of otsu's algorithm and normalization on Fig 1 image.

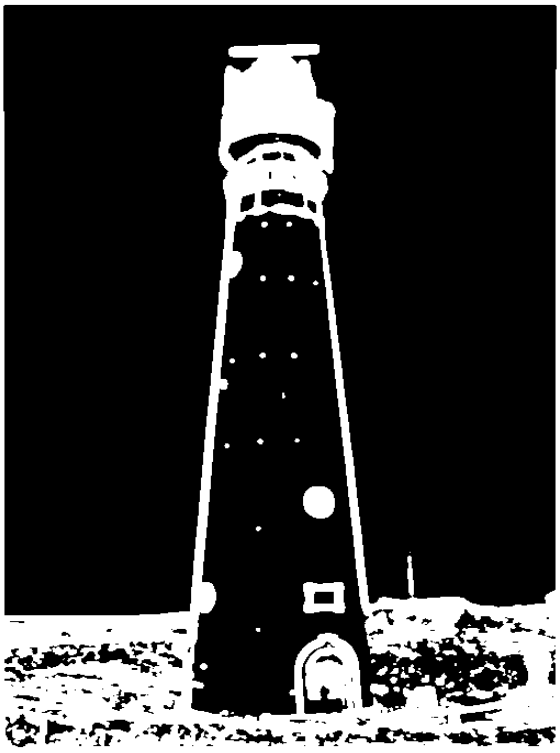

Figure 24: Combined image of Fig 1 after Texture segmentation.

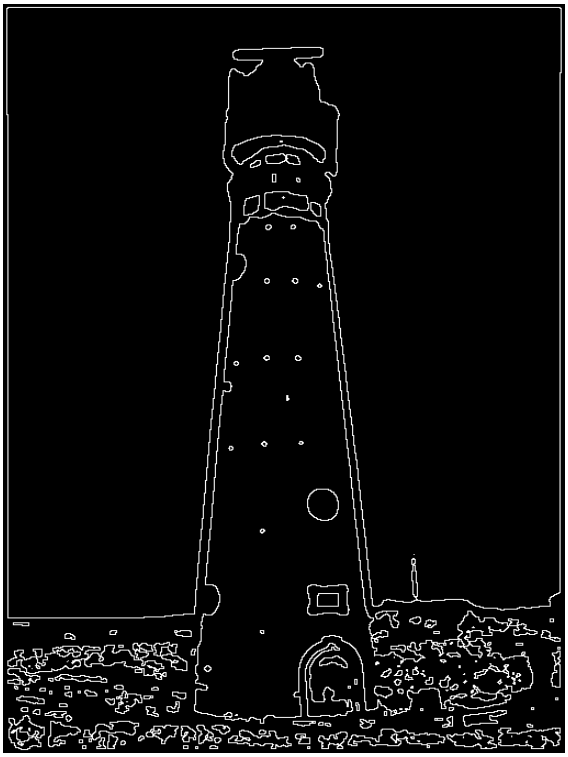

Figure 25: Contour image obtained from Texture segmented Fig 1 image.

Figure 26: 3x3 filtered mask after 4th iteration of otsu's algorithm and normalization on Fig 2 image.

Figure 27: 5x5 filtered mask after 4th iteration of otsu's algorithm and normalization on Fig 2 image.

Figure 28: 7x7 filtered mask after 4th iteration of otsu's algorithm and normalization on Fig 2 image.

Figure 29: Combined image of Fig 2 after Texture segmentation.

Figure 30: Contour image obtained from Texture segmented Fig 2 image.

Figure 31: 3x3 filtered mask after 4th iteration of otsu's algorithm and normalization on Fig 3 image.

Figure 32: 5x5 filtered mask after 4th iteration of otsu's algorithm and normalization on Fig 3 image.

Figure 33: 7x7 filtered mask after 4th iteration of otsu's algorithm and normalization on Fig 3 image.

Figure 34: Combined image of Fig 3 after Texture segmentation.

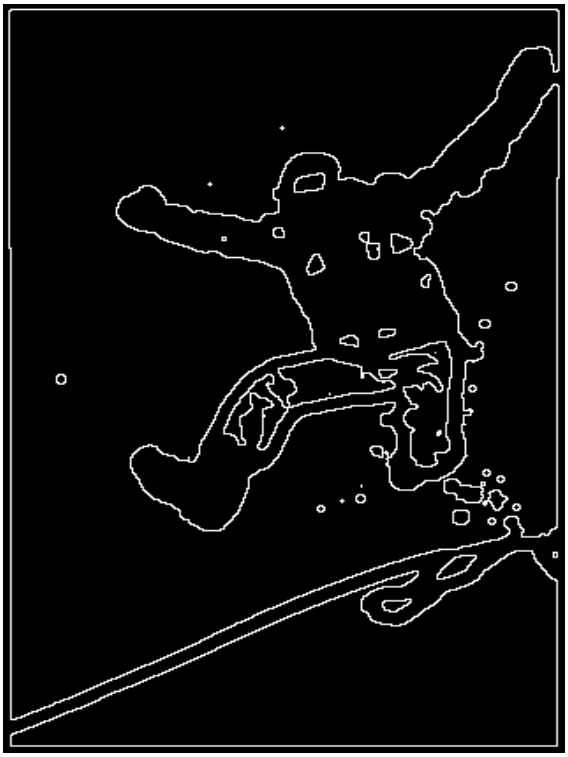

Figure 35: Contour image obtained from Texture segmented Fig 3 image.

9 Comments

Otsus method exhibits the relatively good performance if the histogram can be assumed to have bimodal distribution and assumed to possess a deep and sharp valley between two peaks. It really works great when there are only two kind of objects in the image, foreground and background. And when their colors are well separated by two peaks in the histogram.

```
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy, matplotlib.pyplot as plt
from scipy import signal, optimize


#================================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 6
#================================================================================

#================================================================================
# FUNCTIONS CREATED IN HW6.
#================================================================================

def applyOtsu( img=None, kRange=[0, 256] ):
    '''
    Takes in a single channel image and finds out the threshold value k using
    otsu algorithm. Also applies the k to filter the image and returns a binary
    filtered image.
    The kRange specifies the range of bins of the histogram that will be used to
    find the threshold.
    '''
    hist = cv2.calcHist( [img], channels=[0], mask=None, histSize=[256], ranges=[0, 256] )
    # The [0] means calculate the hist on the 0th channel.

    # Only take the rows of hist that falls in the range of kRange.
    # Basically this can be used to truncate the histogram.
    hist = hist[ kRange[0] : kRange[1], : ]

    # Finding the probability distribution.
    N = np.sum( hist )     # Total number of pixels in the histogram.
    P = hist / N
    PT = np.sum(P)     # Sum of all probability. This theoritically should be 1,
    # but because of the floating point division may be like 9.999999, or even 1.00001.
    # So if it is 1.00001, then subtracting the sum of all probability (when k will
    # be in the last bin) from 1, will make the count -ve. So PT will be used instead
    # of 1.

#--------------------------------------------------------------------------------

    # Bins can have different ranges. Not necessarily from 0 to 256.
    # If bins are like 100 to 150. So bin values will be like: [100, 101, 102, ... 149, 150].
    bins = np.mgrid[ kRange[0] : kRange[1] ]    # Bin values.
    #print(bins.shape)

    muT = np.matmul( bins, P )
    #print(N, P.shape, muT.shape, muT, np.sum(img)/N)

    varBlist = []   # List of between class variances.
    for idx, k in enumerate( bins ):
        wk = np.sum( P[ : idx+1 ] )
        muk = np.matmul( bins[ : idx+1 ], P[ : idx+1 ] )

        # Calculating the variances.
        varB = ( muT * wk - muk ) * ( muT * wk - muk ) / ( wk * ( PT - wk ) + 0.000001 )
        varBlist.append( varB )
        # The 0.000001 is there to avoid division by 0 (which will happen in case
        # when k pointing to the end of the last bin so wk will be equal to PT then
        # which means all the counts in the entire histogram is considered, and so
        # the dinominator will be 0).
        # In some cases, if the first bin also does not have any pixels, then wk
        # may be 0. The 0.00001 saves that case as well.
```

```
        #print(wk, varB)

#-------------------------------------------------------------------------------

    varBarray = np.array( varBlist )    # Array of between class variances.

    # Calculate the threshold value (k).
    maxVarB = np.amax( varBarray, axis=0 )      # Max varB (between class variance).
    maxVarBidx = np.argmax( varBarray, axis=0 )
    bestK = bins[ maxVarBidx ]      # Index of the max varB in the varBarray is best k.

    #print(bestK, maxVarB)

    return bestK[0]     # bestK is an array of 1 element. So returning 0th element of that.

#===============================================================================

def segmentByOtsu( img, kRanges=[[0, 256], [0, 256], [0, 256]] ):
    '''
    This function takes in an image and applies otsu segmentation to the image.
    The kRanges specifies the range of bins of the histogram that will be used to
    find the threshold in the different channels.
    '''
    # Splitting the images and making a list of the channels.
    if len(img.shape) == 2:     # Gray image.
        imgChannels = [img]
        h, w = img.shape
        c = 1
    elif len(img.shape) == 3:   # Colored image.
        b, g, r = cv2.split(img)
        imgChannels = [b, g, r]
        h, w, c = img.shape
    else:
        print( '\nERROR: Image is not gray or colored. Aborting.\n' )
        return

#-------------------------------------------------------------------------------

    # Calculate the histogram of the channels.
    fig1 = plt.figure(1)
    fig1.gca().cla()        # Clear the axes for new plots.
    color = ['b', 'g', 'r']
    kList, filteredImgList = [], []
    combinedImg = np.ones( (h, w), dtype=np.uint8 ) * 255

    for idx, ch in enumerate( imgChannels ):
        # Blurring the image a little bit for segmentation to work better.
        ch = cv2.GaussianBlur( ch, (5,5), 0 )

        kRange = kRanges[ idx ]

        hist = cv2.calcHist( [ch], channels=[0], mask=None, histSize=[256], ranges=[0,256] )
        # The [0] means calculate the hist on the 0th channel.

        # Only take the rows of hist that falls in the range of kRange.
        # Basically this can be used to truncate the histogram.
        hist = hist[ kRange[0] : kRange[1], : ]

        plt.plot( hist, color=color[idx] )  # Plot histogram.

        # Apply otsu algorithm to the current channel.
        k = applyOtsu( ch, kRange=kRange )
        kList.append( k )

        plt.plot( [k,k], [0,100], color[idx], linewidth=2.0 )     # Show k on histogram.
```

```
        # Create the binary filtered image by using the k value.
        _, filteredImg = cv2.threshold( ch, k, 255, cv2.THRESH_BINARY )

        filteredImgList.append( filteredImg )

        plt.xlabel('bins'); plt.ylabel('no. of pixels'); plt.title('Histogram'); plt.grid()

    #plt.show()
    #fig1.savefig( f'./hist_2.png' )

    # Return the list of k's and filteredImgs.
    return kList, filteredImgList

#================================================================================

def calcVarImg( img=None, kernel=None ):
    '''
    This function takes in an image and a kernel size and calculates a variance
    image which is just an array containing the variance values calculated from a
    neighborhood of the size of the given kernel around every pixel of given image.
    The input kernel has the format (h, w).
    '''
    kernelH, kernelW = kernel[0], kernel[1]
    if len(img.shape) == 1:     # Image is grayscale.
        imgH, imgW = img.shape
    else:       # Image is colored, then convert to grayscale.
        img = cv2.cvtColor( img, cv2.COLOR_BGR2GRAY )
        imgH, imgW = img.shape

    kernelForMU = np.ones( kernel ) / ( kernelH * kernelW )

    meanImg = signal.convolve2d( img, kernelForMU, mode='same' )

    # Convert the img from uint8 to float image so that when doing the square of
    # the image the values do not round off to 256.
    imgSquare = np.asarray( img, dtype=np.float32 ) * np.asarray( img, dtype=np.float32 )

    #print( np.amin(imgSquare), np.amax(imgSquare) )

    meanImgSquareImg = signal.convolve2d( imgSquare, kernelForMU, mode='same' )

    varianceImg = meanImgSquareImg - meanImg * meanImg

    #print( np.amin( varianceImg ), np.amax( varianceImg ) )

    return varianceImg

#================================================================================

def normalize( img, scale=255 ):
    '''
    Normalize the image and scale it to the scale value.
    '''
    img = ( img - np.amin( img ) ) / ( np.amax( img ) - np.amin( img ) )
    img = img * 255
    img = np.asarray( img, dtype=np.uint8 )

    return img

#================================================================================

def findContour( mask ):
    '''
    This function takes in a binary image and finds the contour that separate the
```

```
    foreground and the background in the image.
    '''
    maskH, maskW = mask.shape
    contour = np.zeros( mask.shape, dtype=np.uint8 )

    N = 3       # 3x3 window, with 8-neighbors.
    halfSize = int( N / 2 )

    # Only if the center pixel value is 255, and not all of its surrounding pixels
    # equals to 255, is considered as a valid contour point between the foreground
    # and background.

    for x in range( halfSize, maskW-halfSize ):
        for y in range( halfSize, maskH-halfSize ):
            window = mask[ y - halfSize : y + halfSize + 1,
                           x - halfSize : x + halfSize + 1 ]
            #print( window.shape )
            if window[1, 1] == 255 and np.sum( window ) < int( N * N * 255 ):
                contour[y, x] = 255

    return contour

#================================================================================

if __name__ == '__main__':

    # TASK 1.1

    # Loading the images.

    filepath = './HW6Pics'

#--------------------------------------------------------------------------------

    # Color based segmentation for lighthouse image.
    filename = '1.jpg'
    img = cv2.imread( os.path.join( filepath, filename ) )
    kList, filteredImgList = segmentByOtsu( img, kRanges=[[0, 256], [0, 256], [0, 256]] )

    # Red is the dominant color in the lighthouse image. Hence the red mask is
    # anded with the compliment of blue and green masks to get a segmented lighthouse.
    # This observation is done after looking at the original image and the filtered
    # image returned by segmentByOtsu function.

    bChan, gChan, rChan = filteredImgList[0], filteredImgList[1], filteredImgList[2]

    cv2.imwrite( f'./bchannel_{filename[:-4]}.png', bChan )
    cv2.imwrite( f'./gchannel_{filename[:-4]}.png', gChan )
    cv2.imwrite( f'./rchannel_{filename[:-4]}.png', rChan )

#--------------------------------------------------------------------------------

    combinedImg = rChan   # Starting with red.

    bChanInv = cv2.bitwise_not( bChan )  # Complimenting blue.

    # Anding with the compliment of blue.
    combinedImg = cv2.bitwise_and( combinedImg, bChanInv )

    gChanInv = cv2.bitwise_not( gChan )  # Complimenting green.

    # Anding with the compliment of green.
    combinedImg = cv2.bitwise_and( combinedImg, gChanInv )

    channels = np.hstack( ( bChan, gChan, rChan ) )
```

```
    cv2.imshow( 'Channels', channels )
    cv2.imshow( 'Combined Image', combinedImg )
    cv2.waitKey(0)
    cv2.imwrite( f'./combined_image_{filename[:-4]}.png', combinedImg )

#================================================================================

    # Color based segmentation for ski image.
    filename = '3.jpg'
    img = cv2.imread( os.path.join( filepath, filename ) )
    kList, filteredImgList = segmentByOtsu( img, kRanges=[[0, 256], [0, 256], [0, 256]] )

    bChan, gChan, rChan = filteredImgList[0], filteredImgList[1], filteredImgList[2]

    cv2.imwrite( f'./bchannel_{filename[:-4]}.png', bChan )
    cv2.imwrite( f'./gchannel_{filename[:-4]}.png', gChan )
    cv2.imwrite( f'./rchannel_{filename[:-4]}.png', rChan )

#--------------------------------------------------------------------------------

    # NEED ITERATION 2.

    # The foreground is filtered out in the blue image. So this means the foreground
    # actually lies below the k threshold. So inverting the bChan to get the
    # region that contains the foreground.
    # This observation is done after looking at the original image and the filtered
    # image returned by segmentByOtsu function.
    bChanInv = cv2.bitwise_not( bChan )  # Complimenting blue.

    img2 = img[:,:,0]

    kList2, filteredImgList2 = segmentByOtsu( img2, kRanges=[[0, kList[0]]] )
    bChan2 = filteredImgList2[0]

    cv2.imwrite( f'./bchannel_iter_2_{filename[:-4]}.png', bChan2 )

#--------------------------------------------------------------------------------

    # Inverting the bChan again to get the foreground.
    bChan2Inv = cv2.bitwise_not( bChan2 )  # Complimenting blue.

    # There are some missing parts in this image, which were visible in the red
    # channel. So this is or-ed with the red to get a better contour. This is
    # a temporary image.
    # But still the part of the snow. So the green and red channel was and-ed
    # and then this xor-ed with the temporary image.
    # This observation is done after looking at the original image and the filtered
    # image returned by segmentByOtsu function.

    # Oring with the red channel of iteration 1.
    combinedImg = cv2.bitwise_or( bChan2Inv, rChan )

    # Oring green and red channel of iteration 1.
    GandR = cv2.bitwise_and( gChan, rChan )

    # Xoring with the green channel of iteration 1.
    combinedImg = cv2.bitwise_xor( combinedImg, GandR )

    cv2.imshow( 'Combined Image', combinedImg )
    cv2.waitKey(0)
    cv2.imwrite( f'./combined_image_{filename[:-4]}.png', combinedImg )

#================================================================================

    # Color based segmentation for baby image.
```

```
    filename = '2.jpg'
    img = cv2.imread( os.path.join( filepath, filename ) )
    kList, filteredImgList = segmentByOtsu( img, kRanges=[[0, 256], [0, 256], [0, 256]] )

    bChan, gChan, rChan = filteredImgList[0], filteredImgList[1], filteredImgList[2]

    cv2.imwrite( f'./bchannel_{filename[:-4]}.png', bChan )
    cv2.imwrite( f'./gchannel_{filename[:-4]}.png', gChan )
    cv2.imwrite( f'./rchannel_{filename[:-4]}.png', rChan )

#-------------------------------------------------------------------------------

    # The baby is mostly white. So the red, blue and green are present in equal
    # amounts. The images obtained are inverted and or-ed together to fill up
    # some of the black patches.
    bChanInv = cv2.bitwise_not( bChan )  # Complimenting blue.
    gChanInv = cv2.bitwise_not( gChan )  # Complimenting blue.
    rChanInv = cv2.bitwise_not( rChan )  # Complimenting blue.

    combinedImg = rChanInv   # Starting with red.

    # Oring with the compliment of green.
    combinedImg = cv2.bitwise_or( combinedImg, gChanInv )
    combinedImg = cv2.bitwise_or( combinedImg, bChanInv )

    channels = np.hstack( ( bChan, gChan, rChan ) )
    cv2.imshow( 'Channels', channels )
    cv2.imshow( 'Combined Image', combinedImg )
    cv2.waitKey(0)
    cv2.imwrite( f'./combined_image_{filename[:-4]}.png', combinedImg )

#===============================================================================

    # Texture based segmentation for lighthouse image.

    filename = '1.jpg'
    img = cv2.imread( os.path.join( filepath, filename ) )

    kernelH, kernelW = 3, 3
    varianceImg3x3 = calcVarImg( img, (kernelH, kernelH) )

    cv2.imshow( 'Variance Image', varianceImg3x3 )
    cv2.imwrite( f'./texture_image_{filename[:-4]}_{kernelH}x{kernelW}.png', varianceImg3x3 )

    # Since varianceImg3x3 has values which are very large than 255 because of the
    # square terms involved. So they have to be normalized and converted to a range
    # of 0 to 255.

    normalizedImg3x3 = normalize( varianceImg3x3 )
    cv2.imshow( f'Normalized Image {kernelH}x{kernelW}', normalizedImg3x3 )
    cv2.imwrite( f'./normalized_texture_image_{filename[:-4]}_{kernelH}x{kernelW}.png', normalizedImg3x3 )
    cv2.waitKey(0)

#-------------------------------------------------------------------------------

    kernelH, kernelW = 5, 5
    varianceImg5x5 = calcVarImg( img, (kernelH, kernelH) )

    cv2.imshow( 'Variance Image', varianceImg5x5 )
    cv2.imwrite( f'./texture_image_{filename[:-4]}_{kernelH}x{kernelW}.png', varianceImg5x5 )

    # Since varianceImg5x5 has values which are very large than 255 because of the
    # square terms involved. So they have to be normalized and converted to a range
    # of 0 to 255.
```

```
    normalizedImg5x5 = normalize( varianceImg5x5 )
    cv2.imshow( f'Normalized Image {kernelH}x{kernelW}', normalizedImg5x5 )
    cv2.imwrite( f'./normalized_texture_image_{filename[:-4]}_{kernelH}x{kernelW}.png', normalizedImg5x5 )
    cv2.waitKey(0)

#-------------------------------------------------------------------------------

    kernelH, kernelW = 7, 7
    varianceImg7x7 = calcVarImg( img, (kernelH, kernelH) )

    cv2.imshow( 'Variance Image', varianceImg7x7 )
    cv2.imwrite( f'./texture_image_{filename[:-4]}_{kernelH}x{kernelW}.png', varianceImg7x7 )

    # Since varianceImg7x7 has values which are very large than 255 because of the
    # square terms involved. So they have to be normalized and converted to a range
    # of 0 to 255.

    normalizedImg7x7 = normalize( varianceImg7x7 )
    cv2.imshow( f'Normalized Image {kernelH}x{kernelW}', normalizedImg7x7 )
    cv2.imwrite( f'./normalized_texture_image_{filename[:-4]}_{kernelH}x{kernelW}.png', normalizedImg7x7 )
    cv2.waitKey(0)

    normalizedImg = cv2.merge( ( normalizedImg3x3, normalizedImg5x5, normalizedImg7x7 ) )

    cv2.imshow( 'Normalized Image Combined', normalizedImg )
    cv2.imwrite( f'./combined_normalized_texture_image_{filename[:-4]}.png', normalizedImg )
    cv2.waitKey(0)

#-------------------------------------------------------------------------------

    # Applying otsu to find the segments from the normalized combined texture images.

    originalNormImg = copy.deepcopy( normalizedImg )

    nIter = 3       # Number of iterations for otsu.
    bchan = normalizedImg3x3
    gchan = normalizedImg5x5
    rchan = normalizedImg7x7
    kList=[256, 256, 256]

    for i in range( nIter ):
        normalizedImg = cv2.merge( ( bchan, gchan, rchan ) )    # Merging.
        cv2.imshow( 'normalizedImg', normalizedImg )
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        kRanges = [ [ 0, kList[0] ], [ 0, kList[1] ], [ 0, kList[2] ] ]
        #kRanges = [ [ kList[0], 256 ], [ kList[1], 256 ], [ kList[2], 256 ] ]

        kList, filteredImgList = segmentByOtsu( normalizedImg, kRanges=kRanges )

        bChan, gChan, rChan = filteredImgList[0], filteredImgList[1], filteredImgList[2]

        print( kList )

        cv2.imwrite( f'./bchannel_{filename[:-4]}.png', bChan )
        cv2.imwrite( f'./gchannel_{filename[:-4]}.png', gChan )
        cv2.imwrite( f'./rchannel_{filename[:-4]}.png', rChan )
        #channels = np.hstack( ( bChan, gChan, rChan ) )
        #cv2.imshow( 'Channels', channels )

        andedChannel = cv2.bitwise_or( bChan, gChan )
        andedChannel = cv2.bitwise_or( andedChannel, rChan )
        cv2.imshow( 'Combined Anded Channel', andedChannel )
        cv2.imwrite( f'./final_ORED_texture_image_{filename[:-4]}.png', andedChannel )
```

```
        cv2.waitKey(0)

#================================================================================

    # Finding the contours.

    # The mask image created earlier is read in.
    maskFileName = 'combined_image_3.png'
    #maskFileName = 'final_ORED_texture_image_3.png'
    mask = cv2.imread( maskFileName )

    # The mask is 3 channel but it is created by boolean operations, hence it has
    # values as only 0,0,0 and 255,255,255.

    # Converting it into single channel, as the findContour only takes in single
    # channel images.
    mask = cv2.cvtColor( mask, cv2.COLOR_BGR2GRAY )

    cv2.imshow( 'Mask', mask )
    cv2.waitKey(0)
    contour = findContour( mask )
    cv2.imshow( 'Contour', contour )
    cv2.waitKey(0)
    cv2.imwrite( f'./{maskFileName[:-4]}_contour.png', contour )
```