ECE 661 - Computer Vision
Homework - 6

Due on Tuesday, October 23th, 2016

Dr. Avinash Kak

Naveen Madapana

1. Otsu's Algorithm

Otsu's algorithm aims to identify the threshold value of pixel intensity that successfully distinguishes the foreground from background. The notion of foreground and background greatly depends on how we perceive the image. Hence, it is very common to perceive the foreground as background and vice versa. In the context of Otsu's algorithm, the pixels below the threshold are considered as background and the ones above the threshold are perceived as foreground.

In mathematical terms, we want to the find the pixel intensity value (k) which optimally separates the foreground (class $C1$) from the background (Class $C0$). Let the pixels below k belong to $C0$ and others belong to $C1$. Otsu's approach maximizes the between class variance and minimizes the within class variance among the classes $C0$ and $C1$. The typical histogram of gray scale image that has a well-separated foreground and background.

The remainder of this homework is organized as follows. First, Otsu's algorithm for grayscale image was described. Next, the Otsu's approach was extended to RGB images and texture based methods. Once the foreground is extracted, contour extraction using 8-neighborhood was discussed. Lastly, the results obtained on the given images for each of these methods were depicted.

1.1 Otsu's Algorithm of Grayscale Images

Let us start by defining the notations. Consider a grayscale image I of size $m \times n$. Let i represent the possible values for graylevels with $i = \{1, 2, 3, \ldots, L\}$. The value of L is 255 for grayscale and RGB images. Let n_i be the total number of pixels in an image with the intensity value of i. Finally, let N be the total number of pixels. Hence, the probability that a given pixel will have an intensity value of i is given as:

$$p_i = \frac{n_i}{N}$$

Next, the average intensity level μ_T is given as:

$$\mu_T = \sum_{i=1}^{L} i p_i$$

Let $w_0 = \sum_{i=1}^{k} p_i$ and $w_1 = \sum_{i=k_1}^{L} p_i$ be the probabilities of the foreground and background classes respectively for a given k value. The expected value of both the classes are given as:

$$\mu_0 = \sum_{i=1}^{k} \frac{ip_i}{w_0}\ , \qquad \mu_1 = \sum_{i=k+1}^{L} \frac{ip_i}{w_1}$$

Lastly, let us define the between class variance $\sigma_B^2(k)$. It is expressed in terms of μ_0, μ_1, w_0 and w_1.

$$\sigma_B^2(k) = w_0(\mu_0 - \mu_T)^2 + w_1(\mu_1 - \mu_T)^2 = w_0 w_1(\mu_0 - \mu_1)$$

Now, the objective is to find the optimal value of k that maximizes the between class variance $\sigma_B^2(k)$. Therefore the value of $\sigma_B^2(k)$ is computed for all possible values of k, and the value of k that produces the largest value of $\sigma_B^2(k)$ was considered as the Otsu's threshold. Next, the pixels with $i \leq k$ are considered as background and the ones with $i > k$ are considered as foreground.

1.2 Otsu's Algorithm of RGB Images

Human vision is inherently based on color perception. There are several theories behind how the color is perceived by the human eye. Some of the prominent ones include trichromatic vision and opponent colors. Computationally, there are various formats in which the color images are stored in the computer: RGB, HSV, Lab, etc. In this homework, RGB color format is used to process the images to identify the Otsu's threshold to isolate the foreground from the background.

Grayscale images have only one channel while the color images have three channels. Otsu's algorithm described in the previous section can be easily adapted to the color images. First, Otsu's algorithm is applied to each of the three channels individually. This will result in three threshold values, one for each channel. Now, these threshold values are used to produce three binary masks (1 - foreground and 0 - background). The overall binary mask for the foreground is considered as the logical and of those three binary masks.

1.3 Otsu's Algorithm of Texture of Images

In the cases where the contrast difference between foreground and background is not distinguishable, it is common to create texture representation of the image before applying Otsu's algorithm. This approach is very similar to the RGB images except that we replace the channels R, G and B with the image texture computed at various kernels. The kernel sizes used in this homework include 3 x 3, 5 x 5 and 7 x 7. At every pixel, standard deviation of the pixels present in the kernel window is computed. Since we have three kernel sizes, we will get three channels for each image. The steps are detailed below.

1. Convert the image to grayscale
2. For each kernel window size $N_k = \{3, 5, 7\}$ do:
   (a) For each pixel in the grayscale image, obtain the pixel intensities present in the kernel window of size $N_k \times N_k$ around the pixel
   (b) Calculate the standard deviation (std) of those pixel intensities
   (c) Create a new image of same size as the original image but with this std as the intensities
   (d) Since the these intensities may not range from 0 to 255, we discretized the stds into 256 bins. A new image is created with the discretized stds.
3. The resulting image is constructed by stacking the three images (one for each kernel size). Now, we use this resulting image and follow the approach that we used for the RGB images.

2. Contour Detection

Otsu's algorithm will result in a binary mask (1 - for foreground and 0 - for background). Once we obtain the binary mask, we want to identify the contours. 8 - connectivity is used to construct the set of neighboring pixels. In 8-connectivity, all of the neighboring pixels including the diagonal pixels are considered as neighbors.

In contour detection, we want to identify the pixels that act as borders in the binary mask. At every pixel, we use the elements of the neighboring pixels and mark the current pixel as a border pixel or non-border pixel. For border pixels, the values of 8-neighbors will be a mixture of zeros and ones. However, for non-border pixels, the values will be all zeros or all ones. The algorithm for contour detection is described below:

1. For each pixel (i, j) in the binary mask, do:
   (a) Obtain the values present in the 3×3 window around the pixel
   (b) Sum those nine values in the window
   (c) If the sum is strictly greater than zero and strictly less than 9, mark the pixel as the border. Other pixels which did not satisfy this criteria are marked as non-border pixels.

3. Results

This section presents the results obtained using RGB segmentation and the texture segmentation.

3. Results continued on next page...

3.1 Lighthouse

3.1.1 RGB Segmentation

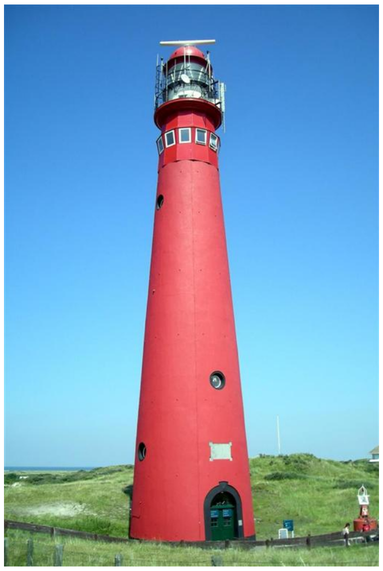
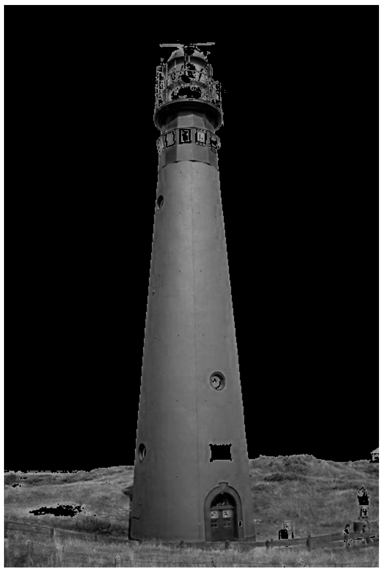
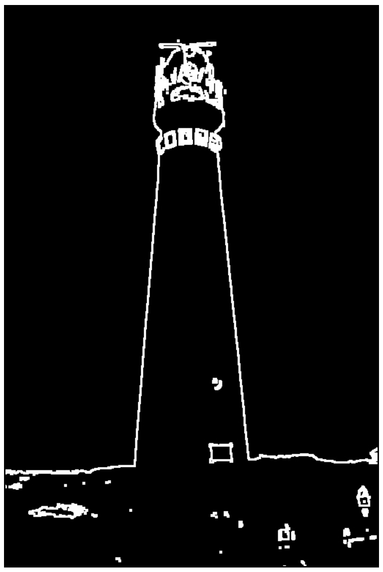


Figure 1: Original image



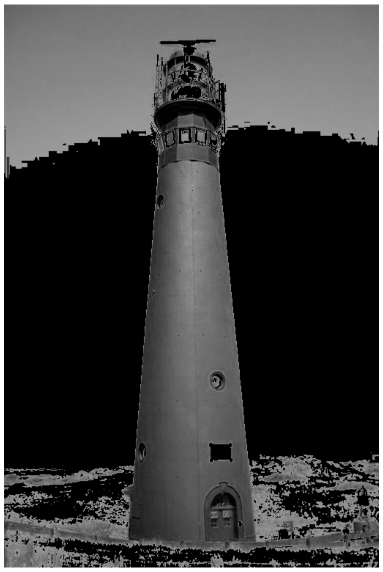
(a) Foreground



(b) Foreground - Contours

Figure 2: Blue Channel

3. Results continued on next page...


(a) Foreground

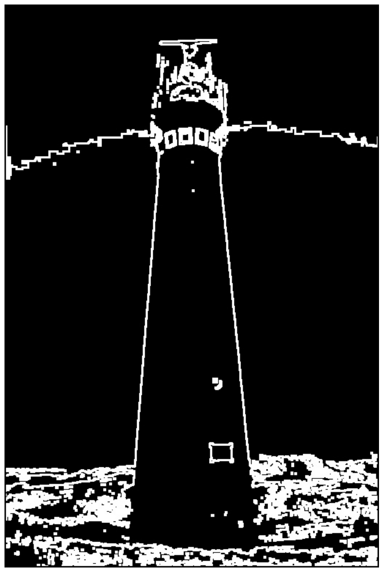
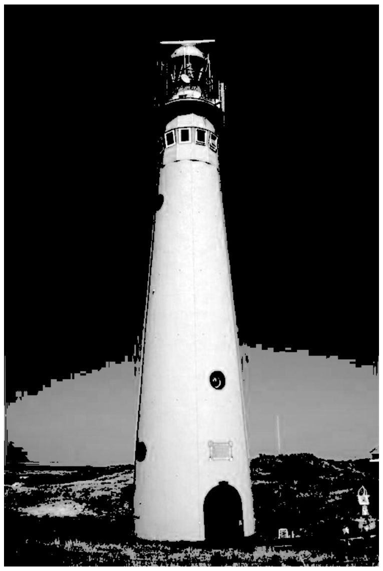

(b) Foreground - Contours

Figure 3: green channel


(a) Foreground

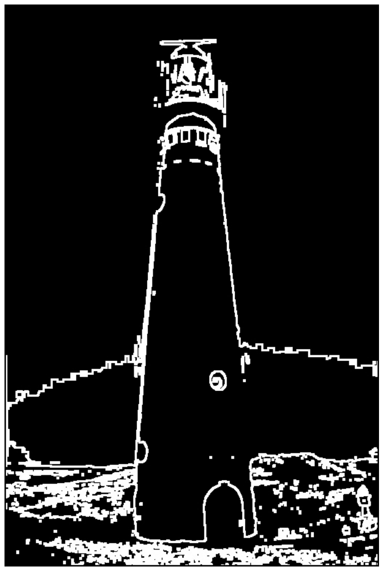

(b) Foreground - Contours

Figure 4: red channel

3.1.2 Texture Segmentation

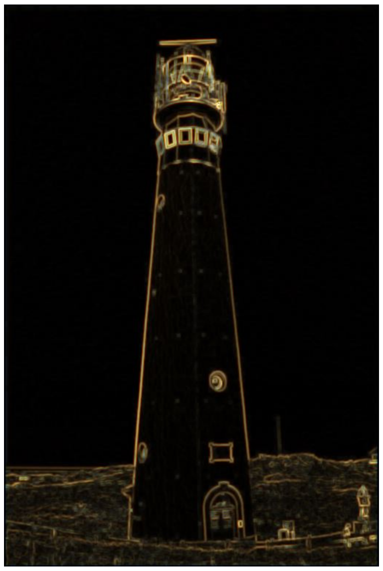
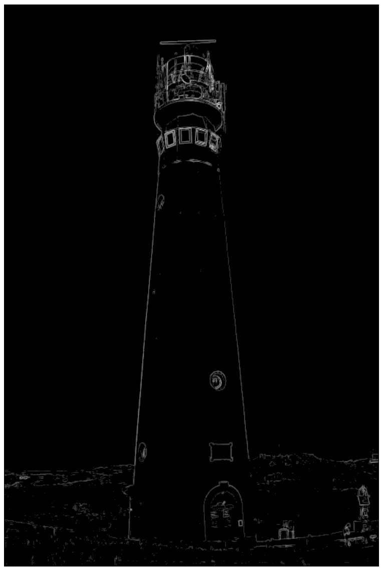
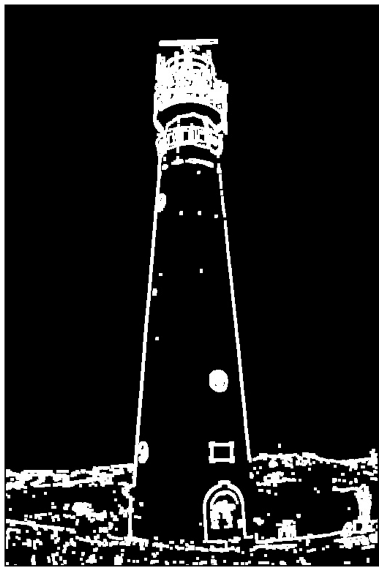


Figure 5: Texture of original image



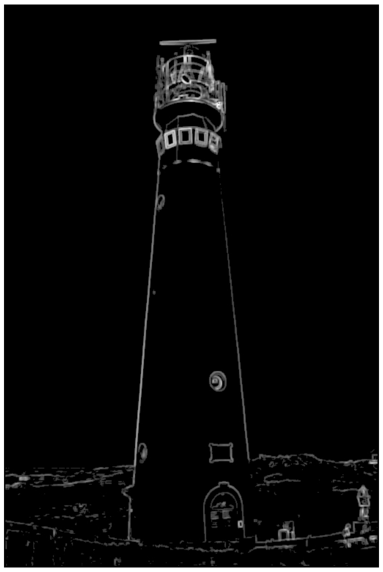
(a) Foreground



(b) Foreground - Contours

Figure 6: 3x3 kernel for texture extraction

3. Results continued on next page. . .


(a) Foreground

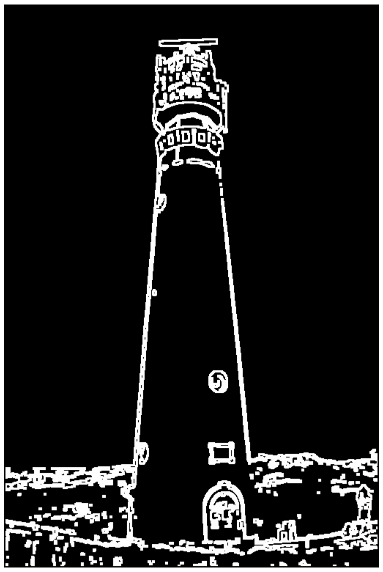
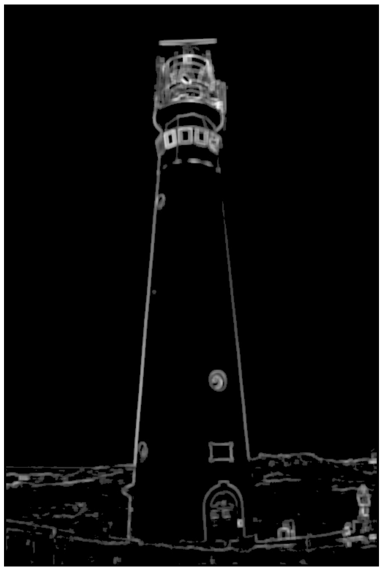

(b) Foreground - Contours

Figure 7: 5x5 kernel for texture extraction


(a) Foreground

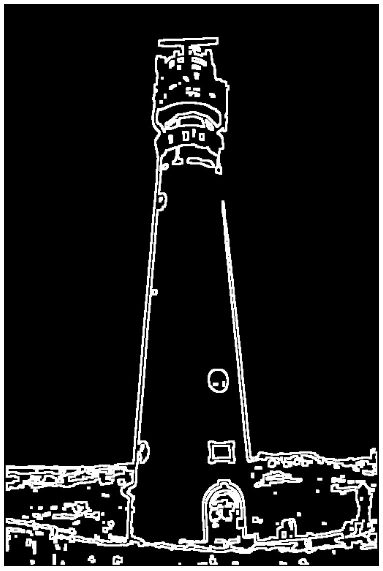

(b) Foreground - Contours

Figure 8: 7x7 kernel for texture extraction

3. Results continued on next page. . .

3.2 Baby

3.2.1 RGB Segmentation

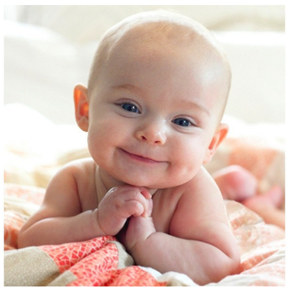


Figure 9: Original image

(a) Foreground (b) Foreground - Contours

Figure 10: Blue Channel

(a) Foreground (b) Foreground - Contours

Figure 11: green channel

3. Results continued on next page. . .

(a) Foreground (b) Foreground - Contours

Figure 12: red channel

3.2.2 Texture Segmentation

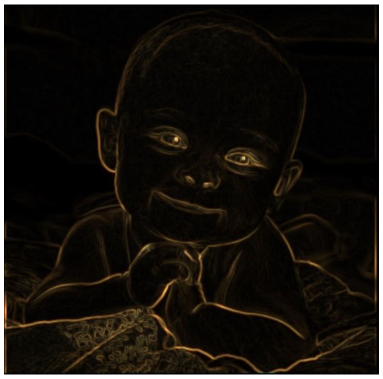
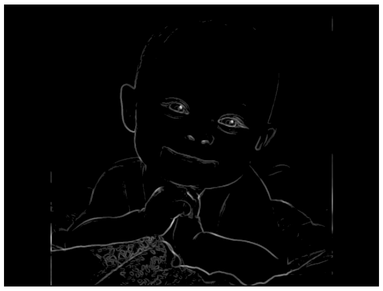


Figure 13: Texture of original image

(a) Foreground (b) Foreground - Contours

Figure 14: 3x3 kernel for texture extraction

3. Results continued on next page...


(a) Foreground

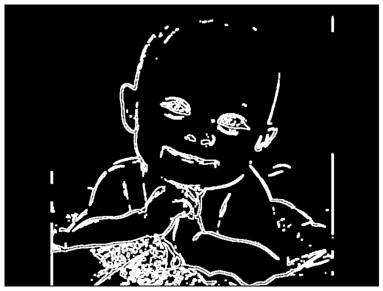
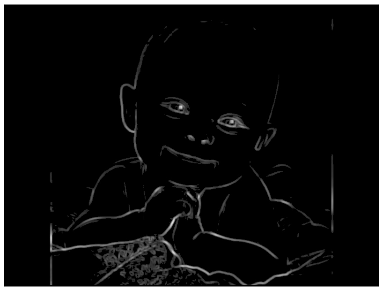

(b) Foreground - Contours

Figure 15: 5x5 kernel for texture extraction


(a) Foreground

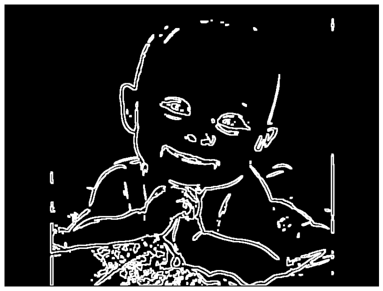

(b) Foreground - Contours

Figure 16: 7x7 kernel for texture extraction

3.3 Ski

3.3.1 RGB Segmentation

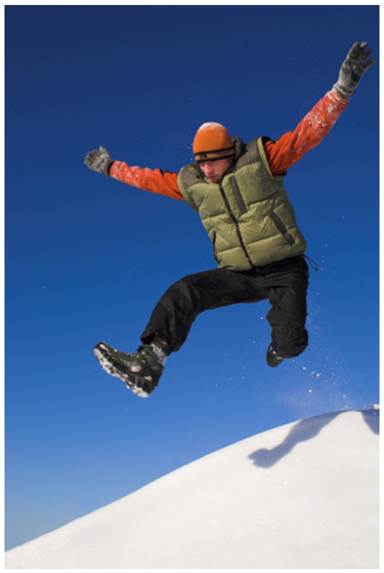
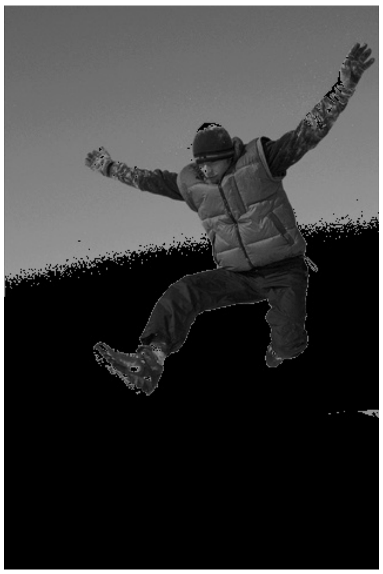


Figure 17: Original image


(a) Foreground

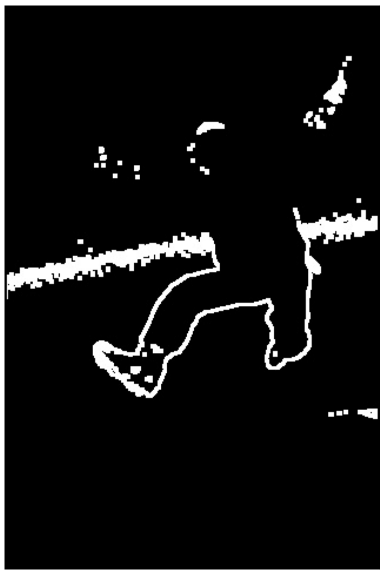
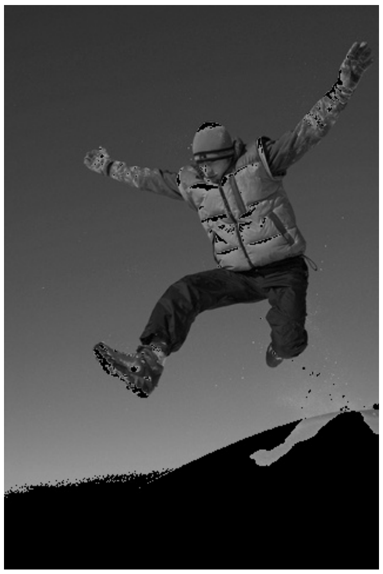

(b) Foreground - Contours

Figure 18: Blue Channel


(a) Foreground

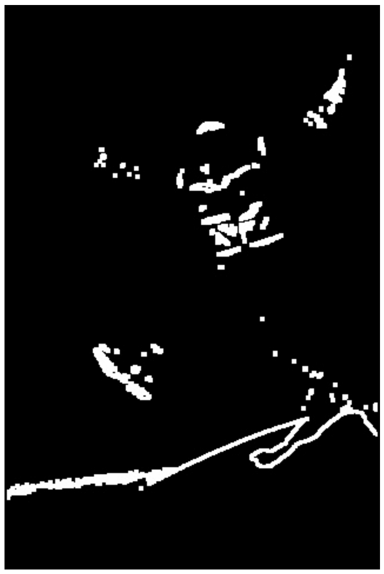
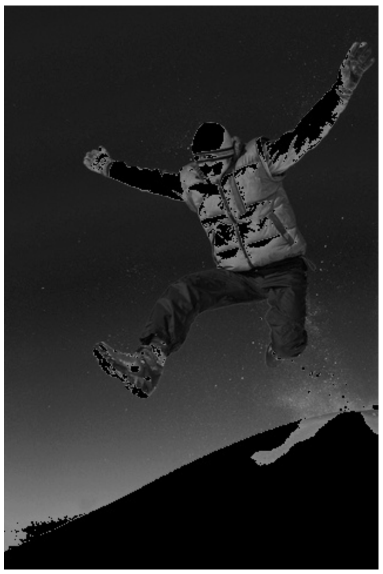

(b) Foreground - Contours

Figure 19: green channel

3. Results continued on next page...


(a) Foreground

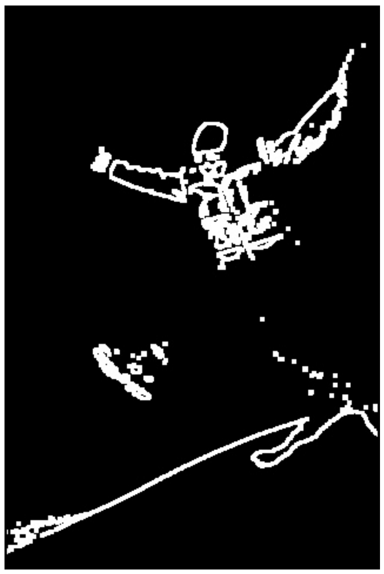

(b) Foreground - Contours

Figure 20: red channel

3.3.2 Texture Segmentation

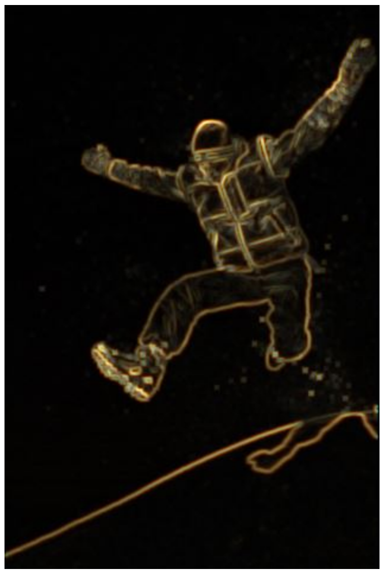
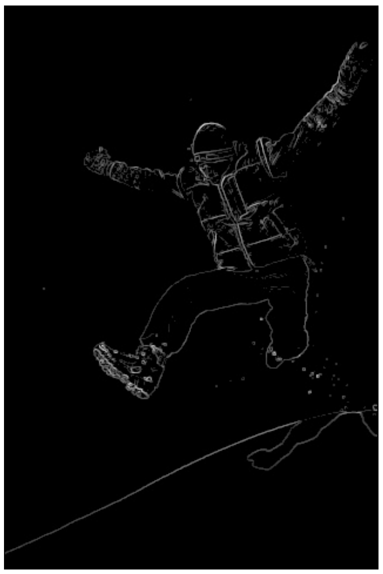


Figure 21: Texture of original image

3. Results continued on next page...


(a) Foreground

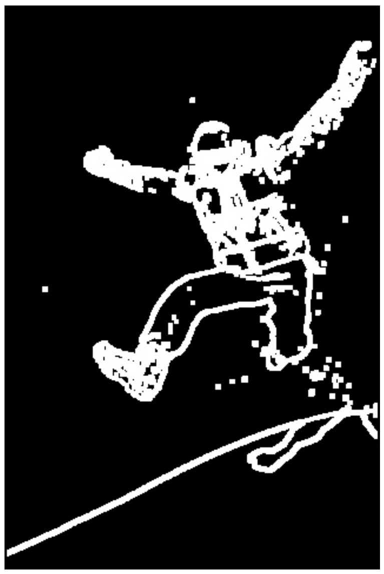
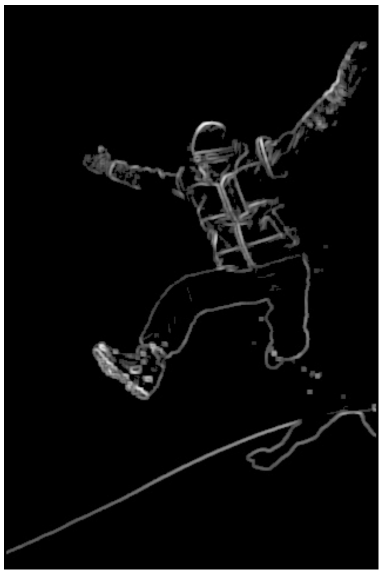

(b) Foreground - Contours

Figure 22: 3x3 kernel for texture extraction


(a) Foreground

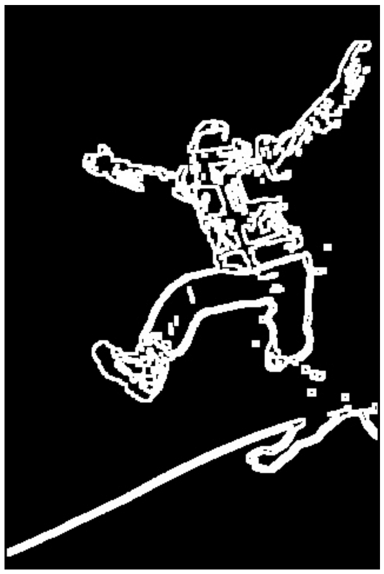
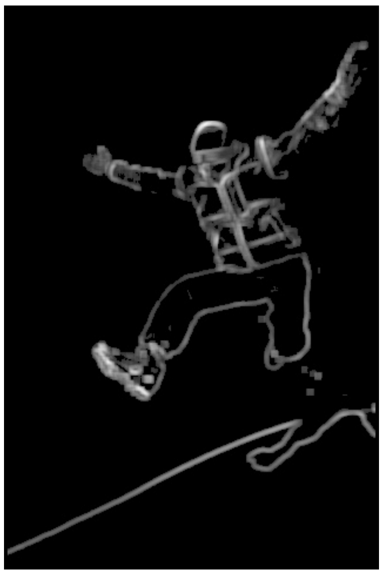

(b) Foreground - Contours

Figure 23: 5x5 kernel for texture extraction


(a) Foreground

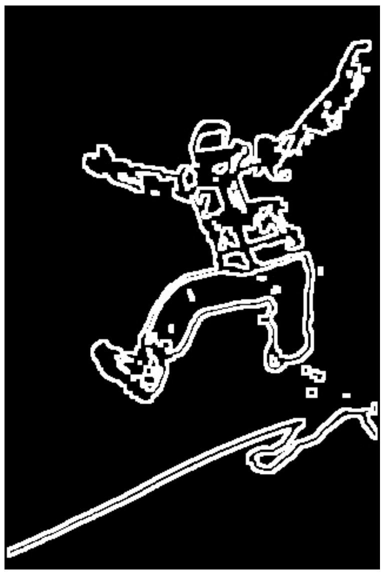

(b) Foreground - Contours

Figure 24: 7x7 kernel for texture extraction

4. Results Discussion

- For RGB segmentation and Texture based segmentation, the final binary mask need not necessarily be the AND operation of the binary masks of all of the channels. For instance, the images with sky (blue color) as background, the binary mask corresponding to the blue channel is more relevant. Hence, we can ignore the green and red channels for such images.
- For contour detection, it is observed that, as the kernel size increases the thickness of the contour increases. In this homework, kernel size of 5 x 5 is used.
- When creating texture based representation of the images, standard deviation is used instead of variance as the variance can potentially be in the order of 1000s while standard deviation would be a value less than 255.
- For the image of the Baby, during RGB segmentation, the accuracy of segmentation obtained by using only one of the channels is almost same as using the three channels combined. This is due to the fact that the image of the baby is mostly white.
- In most cases, texture based segmentation provides good accuracy in comparison to the RGB segmentation.
- After contour extraction, it is good to apply erosion and dilation to remove the noise. This would result in superior results.

5. Code

Listing 1: HW6 code

```python
import cv2
import numpy as np
import os, time, sys
import matplotlib.pyplot as plt
from os.path import basename, dirname, splitext, join

def create_contours(img_mask, kernel_size = 3):
    '''
    Description:
        Given the binary image mask, this function can be used to identify the
        contours in the given mask. 8-connectivity is used to identify the
        neighboring pixels of any given pixel.
    Input arguments:
        * img_mask: 2D np.ndarray consiting of zeroes and ones.
        * kernel_size: Any odd number greater than three.
    Return:
        * img_contour: np.ndarray of same size as the img_mask. This is an
        image consisting of 255 (border pixel) or 0 (non-border pixel)
    '''
    assert img_mask.ndim == 2, 'img_mask should be a binary image'
    assert kernel_size%2 ==1, 'Kernel size should be an odd number'
    assert kernel_size >= 3, 'Kernel size should be greater than 3'

    half_sz = kernel_size/2

    img_contour = np.zeros_like(img_mask).astype(np.uint8)

    for ridx in range(half_sz, img_mask.shape[0]-half_sz):
        for cidx in range(half_sz, img_mask.shape[1]-half_sz):
            temp = img_mask[ridx-half_sz:ridx+half_sz, cidx-half_sz:cidx+half_sz]
            value = np.mean(temp.flatten())
            if(value != 0 and value != 1):
                img_contour[ridx, cidx] = 255
    return img_contour

def create_img_texture(img, kernel_sizes = [3, 5, 7]):
    '''
    Description:
        Create texture representation of the image (trep). trep will have
        three channels each corresponding to one of the kernel sizes.
    Input arguments:
        * img: np.ndarray. Gray scale image (M x N) or RGB image (M x N x 3)
        * kernel_sizes: list of odd numbers greater than 3. length of the list
        should be 3.
    Return:
        * img_texture: texture represetnation of the image.
    '''
    assert len(kernel_sizes) == 3, 'No. of kernel sizes should be 3.'
    assert img.ndim == 2 or img.ndim == 3, 'Image should be either rgb or grayscale'

    img_texture = np.zeros((img.shape[0], img.shape[1], len(kernel_sizes)))

    for kidx, ksize in enumerate(kernel_sizes):
        img_texture[:, :, kidx] = compute_img_variance(img, ksize)

    return img_texture

def compute_img_variance(img, kernel_size):
    '''
    Description:
        Compute the image variance at each pixel within a kerne window of size
        kernel_size
    Input arguments:
        * img: 2D np.ndarray. It can be a rgb or gray scale image
        * kernel_size: An odd positive integer greater than 3.
    Return:
        * img_texture: 2D np.ndarray of similar dimension as img
    '''
    assert kernel_size%2 == 1, 'Kernel size should be odd number'
    assert img.ndim == 2 or img.ndim == 3, 'Image should be either rgb or grayscale'

    img_texture = np.zeros((img.shape[0], img.shape[1]))
    half_sz = kernel_size / 2

    for ridx in range(half_sz, img.shape[0]-half_sz):
        for cidx in range(half_sz, img.shape[1]-half_sz):
            if(img.ndim == 2):
                temp = img[ridx-half_sz:ridx+half_sz,\
                 cidx-half_sz:cidx+half_sz]
            else:
                temp = img[ridx-half_sz:ridx+half_sz, \
                cidx-half_sz:cidx+half_sz, :]
            img_texture[ridx, cidx] = np.std(temp.flatten())

    return img_texture

def otsu_channels(rgb_img, init_thresh = 0, display = False, out_dir = '.', img_name = None, ch_
    '''
    Description:
        Compute the otsu threshold, foreground and background for an image
        with three channels. The three channels can be either RGB or texture
        representations.
    Input arguments:
        * rgb_img: np.ndarray of size M x N x 3 and of type np.uint8. Or it
        can be an absolute path to the RGB image.
        * init_thresh: Otsu's algorithm will find a threshold that
        distinguishes foreground from background. It looks for a threshold
        starting from init_thresh. Default value is set to 0.
        * display: If True, the images/graphs will be displayed.
        * out_dir: Absolute path to the folder where we want the images to be
        written.
        * img_name: string. The name of the input image.
        * ch_names: list of strings. Channel names. Its size should be 3.
        * write_flag: If True, the output images will be written to the disk.
    Return:
        * thresh_list: list of three np.uint8 integers. Each value corresponds
        to the Otsu's threshold of one of the channels.
        * mask: np.ndarray of same size as rgb_img but in 2D (no third
        dimension). This is an array of True/False. True indicates the
        foreground and False indicating the background.
        * foreground: np.ndarray of same size as rgb_img. The pixels
        corresponding to background are suppressed to zero.
        * background: np.ndarray of same size as rgb_img. The pixels
        corresponding to foreground are suppressed to zero.
    '''
    ## Assertions
    if(isinstance(rgb_img, str)):
        out_dir = dirname(rgb_img)
        img_name = splitext(basename(rgb_img))[0]
        rgb_img = cv2.imread(rgb_img)
    elif(isinstance(rgb_img, np.ndarray)):
        assert rgb_img.ndim == 3, 'Input image should be a gray scale image.'
        assert img_name is not None, 'img_name can not be None'

    if(not isinstance(init_thresh, list)):
        init_thresh = [init_thresh, init_thresh, init_thresh]

    thresh_list = [0]*3
    mask_list = [None] * 3

    if ch_names is None: ch_names = ['blue', 'green', 'red']

    for ch_idx in range(rgb_img.ndim):
        print 'Processing: ', ch_names[ch_idx]
        img = rgb_img[:, :, ch_idx]
        thresh_list[ch_idx], mask_list[ch_idx], foreground, background = \
            otsu(np.copy(img), init_thresh = init_thresh[ch_idx], display = display)
        if(write_flag):
            fore_img_cont = create_contours(mask_list[ch_idx])
            back_img_cont = create_contours(np.logical_not(mask_list[ch_idx]))
            out_path = join(out_dir, img_name + '_' + ch_names[ch_idx])
            cv2.imwrite(out_path+'_fore.jpg', foreground)
            cv2.imwrite(out_path+'_back.jpg', background)
            cv2.imwrite(out_path+'_fore_cont.jpg', fore_img_cont)
            cv2.imwrite(out_path+'_back_cont.jpg', back_img_cont)

    print 'Putting everything together. '

    ## For lighthouse
    # mask = np.logical_or(np.logical_not(mask_list[0]), mask_list[2])
    # mask = np.logical_and(mask, np.logical_not(mask_list[1]))

    ## For baby
    # mask = np.logical_and(mask_list[0], mask_list[1])
    # mask = np.logical_and(mask, mask_list[2])
    ## For ski
    # mask = np.logical_not(mask_list[0])
    mask = np.logical_and(np.logical_not(mask_list[0]), mask_list[1])
    mask = np.logical_and(mask, mask_list[2])

    mask_3d = np.zeros((mask.shape[0], mask.shape[1], 3))
    mask_3d[:,:,0] = mask
    mask_3d[:,:,1] = mask
    mask_3d[:,:,2] = mask

    foreground = np.copy(rgb_img)
    foreground[~mask] = 0
    background = np.copy(rgb_img)
    background[mask] = 0

    if(display):
        cv2.imshow('Foreground', foreground)
        cv2.imshow('Background', background)

    if(write_flag):
        fore_img_cont = create_contours(mask)
        back_img_cont = create_contours(np.logical_not(mask))
        out_path = join(out_dir, img_name)
        cv2.imwrite(out_path+'_fore.jpg', foreground)
        cv2.imwrite(out_path+'_back.jpg', background)
        cv2.imwrite(out_path+'_fore_cont.jpg', fore_img_cont)
        cv2.imwrite(out_path+'_back_cont.jpg', back_img_cont)
        if(display): cv2.imshow('Foreground contours', fore_img_cont)

    if(display): cv2.waitKey(0)

    return thresh_list, mask, foreground, background

def otsu_iter(img, num_iter = 1, display = False, out_dir = '.', img_name = None, ch_names = Non
    '''
    Description:
        Compute the otsu threshold, foreground and background for an image
        with one or three channels. If there are three channels, they can be
        either RGB or texture representations.
        In this function the Otsu's thresholds are determined in more than one
        ITERATIONS. The no. of iterations is determined by num_iter.
    Input arguments:
        * img: np.ndarray of size M x N x 3 or M x N. It is of type np.uint8.
        * display: If True, the images/graphs will be displayed.
        * out_dir: Absolute path to the folder where we want the images to be
         written.
        * img_name: string. The name of the input image.
        * ch_names: list of strings. Channel names. Its size should be 3.
        * write_flag: If True, the output images will be written to the disk.
    Return:
        * init_thresh: list of three np.uint8 integers. Each value corresponds
         to the Otsu's threshold of one of the channels.
        * mask: np.ndarray of same size as img but in 2D (no third dimension).
         This is an array of True/False. True indicates the foreground and
         False indicating the background.
        * foreground: np.ndarray of same size as img. The pixels corresponding
         to background are suppressed to zero.
        * background: np.ndarray of same size as img. The pixels corresponding
        to foreground are suppressed to zero.
    '''
    init_thresh = 0
    for iter_idx in range(num_iter):
        if(img.ndim == 2):
            init_thresh, mask, foreground, background = otsu(np.copy(img), init_thresh = init_th
        else:
            init_thresh, mask, foreground, background = \
            otsu_channels(np.copy(img), init_thresh = init_thresh, \
                          display = display, out_dir = out_dir, img_name = \
                          img_name, ch_names = ch_names, write_flag = \
                          write_flag)
    return init_thresh, mask, foreground, background

def otsu(gray_img, init_thresh = 0, display = False):
    '''
    Description: Apply Otsu's algorithm for foreground extraction.
    Input arguments:
        * gray_img: gray scale image
        * display: If True, histogram, variance and images of both the
        foreground and the background are displayed.
    Return:
        * thresh_level: An np.uint8 integer. Threshold gray level that
        separates the foreground and the background
        * mask: A foreground mask. 2D np.ndarray of same size as the image
        consisting of True/Ones and False/Zeros.
        * foreground: Foreground of the image. 2D np.ndarray of same size as
        the image where background is suppressed to zero intensity value.
        * background: Background of the image. 2D np.ndarray of same size as
        the image where the foreground is suppressed to zero intensity value.
    '''
    ## Assertions
    if(isinstance(gray_img, str)):
        gray_img = cv2.imread(gray_img, 0)
    elif(isinstance(gray_img, np.ndarray)):
        assert gray_img.ndim == 2, 'Input image should be a gray scale image.'

    ## Display the original image
    if(display):
        cv2.imshow('Original image', gray_img)
        cv2.waitKey(0)

    ## Construct histogram of the given image
    hist = np.zeros(256,)
    total_num_pixels = np.sum((gray_img >= init_thresh).flatten())
    for idx in range(init_thresh, hist.size):
        hist[idx] = np.sum((gray_img == idx).flatten()) / \
        float(total_num_pixels)
    # if(display):
    #     plt.plot(hist)
    #     plt.show()

    ## Find the between class variance at each intensity value [0, 255].
    betw_cls_var = np.zeros(256,)
    for idx in range(init_thresh + 1, hist.size):
        # prob. of background
        w0 = np.sum(hist[init_thresh:idx])
        # prob. of foreground
        w1 = np.sum(hist[idx:])

        if(w0 == 0 or w1 == 0): continue

        # Mean of the background
        mu0 = np.sum(np.multiply(range(init_thresh, idx), \
                                 hist[init_thresh:idx])) / w0
        # Mean of the foreground
        mu1 = np.sum(np.multiply(range(idx, hist.size), hist[idx:])) / w1
        # Update the between class variance at current threshold level ('idx')
        # print w0, w1, mu0, mu1
        betw_cls_var[idx] = w0 * w1 * (mu0 - mu1)**2

    # if(display):
    #     plt.plot(betw_cls_var)
    #     plt.show()

    ## Compute the threshold level. It is argmax of b/w class variances.
    thresh_level = np.argmax(betw_cls_var)
    ## Compute the foreground mask
    mask = gray_img > thresh_level
    ## Compute the foreground and the background
    foreground = np.copy(gray_img)
    foreground[~mask] = 0
    background = np.copy(gray_img)
    background[mask] = 0

    if(display):
        cv2.imshow('Foreground', foreground)
        cv2.imshow('Background', background)
        cv2.waitKey(0)

    return thresh_level, mask, foreground, background

#############################
########### MAIN ############
#############################

texture = True

# img_path = r'Images\img1.jpg'
# img_path = r'Images\img2.jpg'
```

```
img_path = r'Images\img3.jpg'

out_dir = dirname(img_path)
img_name = splitext(basename(img_path))[0]
if(texture): out_dir = join(out_dir, img_name) + '_texture'
write_flag = True
display = False

######################
## RGB Segmentation ##
######################
if(not texture):
    img = cv2.imread(img_path)
    ch_names = ['blue', 'green', 'red']
    init_thresh, mask, foreground, background = otsu_iter(img, \
        display = display, out_dir = out_dir, img_name = img_name,\
         ch_names = ch_names, write_flag = write_flag)

##########################
## Texture Segmentation ##
##########################

if(texture):
    img = cv2.imread(img_path, 0)
    ch_names = ['3x3', '5x5', '7x7']
    print 'Creating the texture: '
    img_texture = create_img_texture(img)
    cv2.imwrite(join(out_dir, img_name+'_texture.jpg'), img_texture)
    bins = np.linspace(0, np.max(img_texture.flatten()), num = 257)
    img_texture = (np.digitize(img_texture, bins) - 1).astype(np.uint8)
    cv2.imshow('img_texture', img_texture)
    cv2.waitKey(0)
    init_thresh, mask, foreground, background = otsu_iter(img_texture, display\
     = display, out_dir = out_dir, img_name = img_name, ch_names = ch_names,\
      write_flag = write_flag)
```