Purdue University

ECE 661 Computer Vision

Homework 6

Submission: Arjun Kramadhati Gopi

Email: akramadh@purdue.edu

Theory Question

Lecture 14 will present two very famous algorithms for image segmentation: The Otsu Algorithm and the Watershed Algorithm. These algorithms are as different as night and day. Present in your own words the strengths and the weaknesses of each. (Note that the Watershed algorithm uses the morphological operators that we will discuss in Lecture 13.)

The OTSU implementation relies on the bimodality of the histogram. The accuracy is widely dependent on the histogram. Relatively simpler scenes have good bimodal histograms. For example, the picture of the car in this picture. Whereas, the picture of the pigeon is not so simple as it has a lot of similar foreground and background elements. This makes the histogram not so bimodal. Therefore, we can observe the lack of accuracy in our results for the picture with pigeon.
However, the OTSU algorithm proves to be very fast as all that is required is to maximise the numerator which involves a simple calculation. The histogram values from 0 to 256 and hence the maximisation does not take a lot of time as we just need to iterate in this range.
The watershed algorithm employs a completely different approach to segment images. We treat the foreground objects as literal water shed structures. Structures which are essentially blocking the background scene from 'flowing' into the image foreground regions. Using this analogy, we see the image as a topographical map. The pixels with the highest pixel value are considered as 'tall' regions. These are logically, the foreground pixels. So we make use of morphological operators to make holes in the image and see how the 'flooding' occurs. We then build barriers around the place of flooding. In essence we have built the outline of the foreground. One can immediately see that this approach will be much more accurate than the OTSU algorithm as we do not rely on the histogram and hence we do not rely on how the scene in the image is. On the flip side, one can also observe how this approach will prove to be much slower than the OTSU algorithm. In this approach we will have to iterate over each of the pixel unlike in the OTSU implementation where we had to iterate over 256 values.
The bottom line is that OTSU algorithm is a fast and also fairly reliable method of image segmentation.
The watershed approach is a slow but highly accurate method for segmenting images.

Programming tasks

The broad task of this homework is to separate the foreground and the background in the images provided. We do this by performing image segmentation to identify the unique regions (foreground and background). Using these identified regions, we mark the foreground by extracting the contours. The tasks for this homework are as follows:

1. Image segmentation using RGB channels and Otsu's algorithm
2. Image segmentation using texture-based features and Otsu's algorithm
3. Contour extraction of the binary masks obtained from steps 1 and 2

Let us begin with understanding the fundamental methodology at play in the Otsu algorithm.

Otsu Algorithm

The basic idea in the Otsu algorithm is to perform segmentation or clustering using only the image histogram. What is a image histogram? Image histograms are graphical representations of the tonal distribution of the image. Therefore, it is a frequency plot of the number of pixels and their gray levels. Gray levels are intensity levels of the greyness of each pixel. The value varies between 0 and 255. Take a look at the histogram for the image below:

Figure 1: **Image**

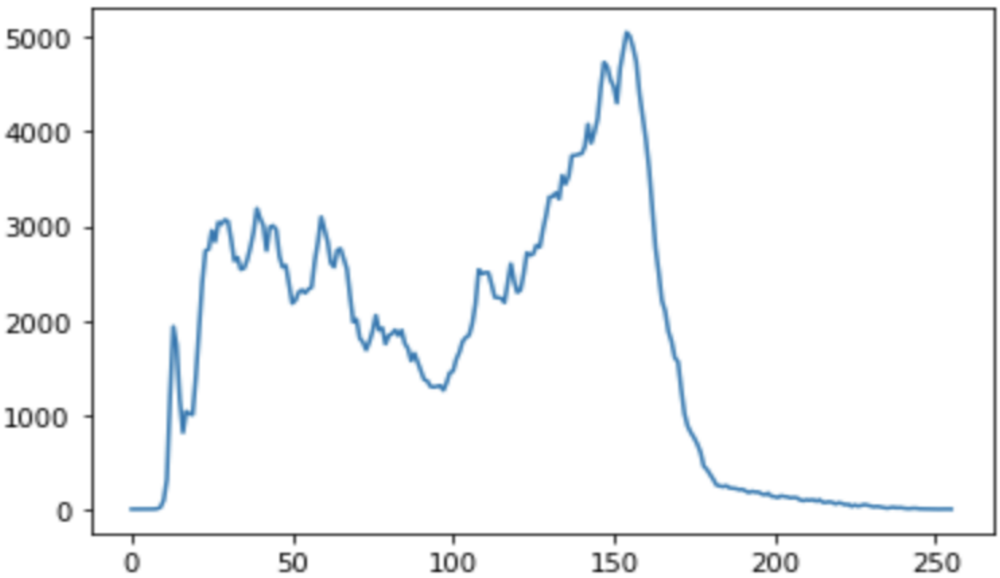

Figure 2: **Image histogram of the image shown above**

The image histogram above shows us the number of pixels per tonal value. We immediately notice that there are two **major bins** in the plot. So, what do these two bins represent? Foreground and the background! We can justify this by logically understanding that the foreground is bound to have different gray levels than the background. It is as simple as that!

Otsu Algorithm gives us an elegant method to **separate** these two bins. By separating the bins what have we essentially done? We have separated the foreground and the background! So how do we actually separate them? We do this by **maximising** the **inter-class variance** of the two classes in the histogram. Logically, this would be the same as **minimizing** the **intra-class** variance. Our job then would be to find the treshold value for the gray level at which the inter-class variance is the highest. The variance is calculated as the difference in the mean values of the classes. Let us name the two classes as C_0 and C_1. For a given threshold value **k** we define the probability of the class as:

$$P(C_0) = \sum_{i=1}^{k} p_i = \omega_0$$

and

$$P(C_1) = \sum_{i=k+1}^{L} p_i = \omega_1$$

The mean value is given by:

$$\mu_0 = \sum_{i=1}^{k} iP(i|C_0)$$

This can in turn be written as:

$$\mu_0 = \sum_{i=1}^{k} i\frac{P(C_0|i)P(i)}{P(c_0)}$$

Since $P(C_0|i)$ is essentially equal to 1 we can write the equation as:

$$\mu_0 = \sum_{i=1}^{k} i\frac{p_i}{\omega_0}$$

The same for the second class:

$$\mu_1 = \sum_{i=k+1}^{L} i\frac{p_i}{\omega_1}$$

The total mean of the entire distribution can be shown to be equal to the sum of the weighted average of the two mean values.

$$\mu_T = \omega_0\mu_0 + \omega_1\mu_1$$

The variance for each class can then be defined as:

$$\sigma_0^2 = \sum_{i=1}^{k}(i-\mu_0)^2\frac{p_i}{\omega_0}$$

and

$$\sigma_1^2 = \sum_{i=k+1}^{L}(i-\mu_1)^2\frac{p_i}{\omega_1}$$

The between class variance can be written as:

$$\sigma_B^2 = \omega_0\omega_1(\mu_1-\mu_0)^2$$

The sum of the within class variance can be written as:

$$\sigma_W^2 = \omega_0\sigma_0^2 + \omega_1\sigma_1^2$$

That brings us to the final step. We would finally be required to maximise the ratio:

$$\lambda = \frac{\sigma_B^2}{\sigma_W^2} = \frac{\omega_0\omega_1(\mu_1 - \mu_0)^2}{\omega_0\sigma_0^2 + \omega_1\sigma_1^2}$$

Therefore the bottom line is to find a value k for which $\lambda(k)$ is maximum. This is the fundamental underlying principle we use to segment the image using Otsu Algorithm.

3 channel RGB based Image Segmentation

The basic principle here is as follows:

1. Obtain three images from the original image by separating the 3 channels - R, G and B channels.
2. Convert each of these thee images into grey scale images.
3. Apply the Otsu Algorithm on each the three grey images.
4. Obtain the 3 binary images by performing step 3.
5. Combine the 3 binary images to get one single binary image which can be used by the contour extraction algorithm.

Texture based based Image Segmentation

1. Obtain the grey scale image from the original image.
2. Construct a window of size NxN.
3. Slide the window over every pixel in the grey scale image.
4. For each iteration, calculate the mean intensity value for the window.
5. Subtract this mean value with the intensity at the center of the window. This would be the within-window variance of the intensity levels.
6. Assign this variance value to the pixel at the center of the window for each iteration. This will give us the texture measure at each pixel at the end.
7. Obtain three such images by varying the window sizes : 3x3, 5x5 and 7x7.
8. Apply Otsu Algorithm for each of the three images to get the equivalent binary images.
9. Combine these three images to get one single binary image which has the contours required for extraction.

Contour extraction

The binary image we obtain from the Otsu algorithm comprises of the 1's and 0's. 1 is assigned to the pixel which is a **part** of the foreground object. 0 is assigned to the pixel which is **not a part** of the foreground. So, 0 pixel would be a pixel which belongs to the background scene. Therefore, all out contour extraction algorithm has to do is to extract **border** pixels with value 1. By extracting the border pixel we get the contour of the foreground object. So what exactly is the border pixel? A border pixel is defined as:

1. Pixel with value =1
2. At least one of the 8 neighboring pixels has a value of 0
3. At least one of the 8 neighboring pixels has a value of 1

If **all** of the above three criterion are satisfied then we label the pixel as a border pixel part of the overall contour. In our code, we do the following steps to identity border pixels:

1. Start a raster scan of the final binary image starting from the left top corner.
2. For each pixel encountered in the scan we check it's value.
3. Additionally, we check the values of the 8 neighboring pixels
4. Using these values we run a check based on the criterion for the border pixel. The check is: If the pixel value is 0 then we reject it. If the pixel value is 1 and all of its 8 neighboring pixels are also 1 then again we reject it. If the pixel value is 1 and some of the 8 neighboring pixels have value 1 and the rest have value 0 we say that the pixel is a border pixel.
5. Using the list of the border pixel, we construct the contour which will then show us the foreground object being identified as being separated from the background.

Observations and optimal parameter setting

The overall observation is that the RGB based segmentation algorithm is able to effectively segment the entire foreground. The texture based algorithm is, however, able to effectively segment the foreground object alone. The foreground object is the object of interest and the object in the foreground with the most varying texture features. We can also observe that the RGB based segmentation algorithm is useful if the foreground scene is a uniform scene like, say, a scene of a beach with the water as the background. So, an image with a smooth foreground would call for a RGB based segmenting algorithm. Whereas, an image with a very noisy foreground would benefit from using the texture based algorithm. Optimal parameters for the three input images:

- Image with the cat:
    1. RGB based: 1 iteration of OTSU
    2. Texture based: 1 iteration of OTSU, window size [5,7,9]
- Image with the pigeon:
    1. RGB based: 2 iterations of OTSU
    2. Texture based: 1 iteration of OTSU. window size [9,11,13]

- Image with the fox:
    1. RGB based: 2 iterations of OTSU
    2. Texture based: 1 iteration of OTSU. window size [19,21,23]

Results

Figure 3: **Greyscale R channel**

Figure 4: **Greyscale G channel**

Figure 5: **Greyscale B channel**

Figure 6: **Post OTSU filtering for R channel using RGB method**

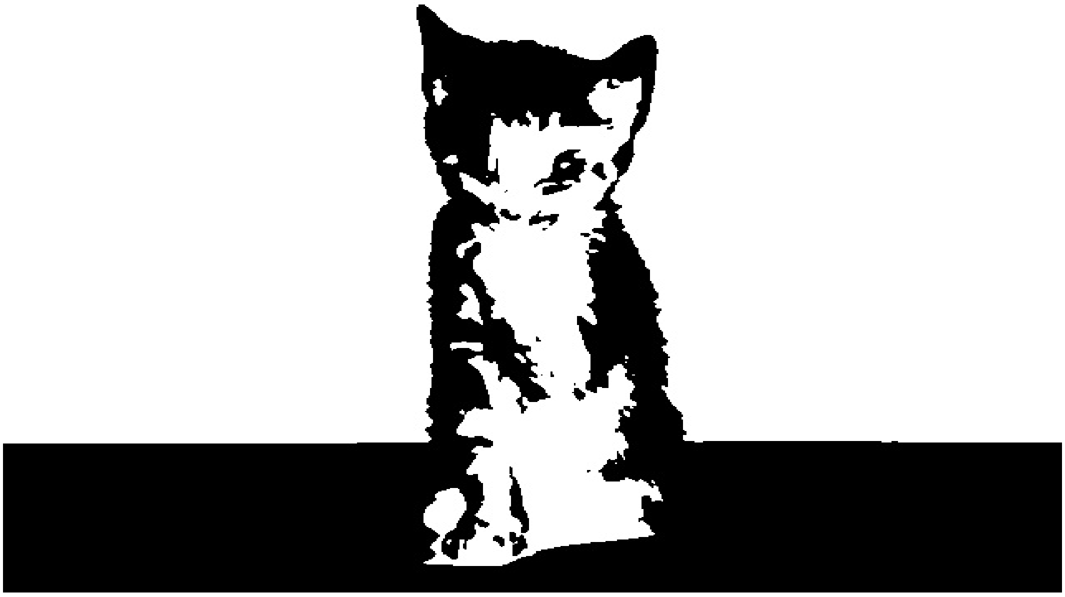

Figure 7: **Post OTSU filtering for G channel using RGB method**

Figure 8: **Post OTSU filtering for B channel using RGB method**

Figure 9: **Post OTSU filtering for all channels using RGB method**

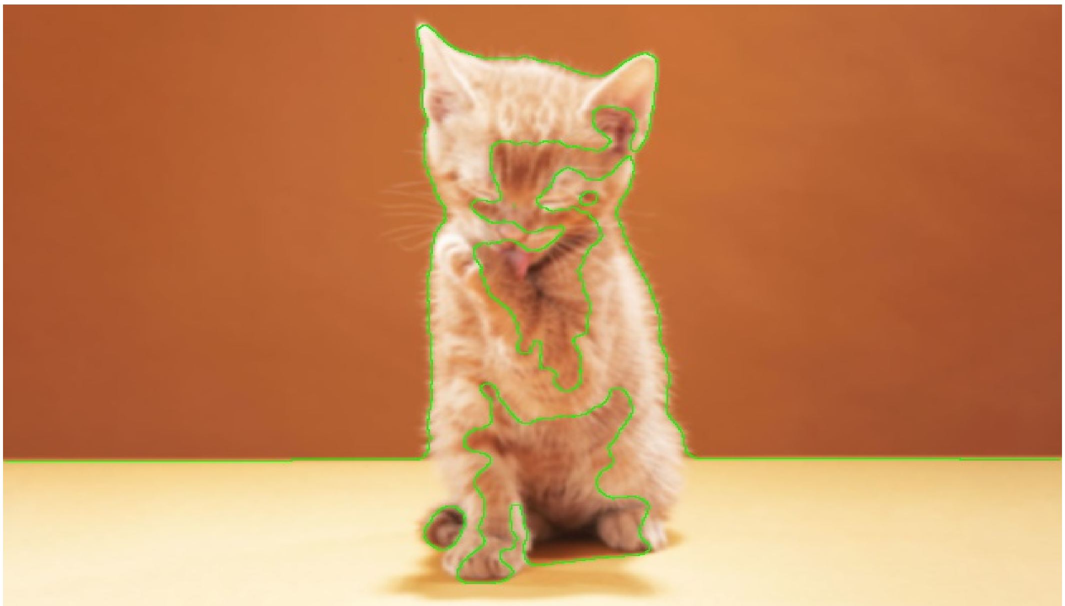

Figure 10: **Contour plot**

Figure 11: **Foreground representation**

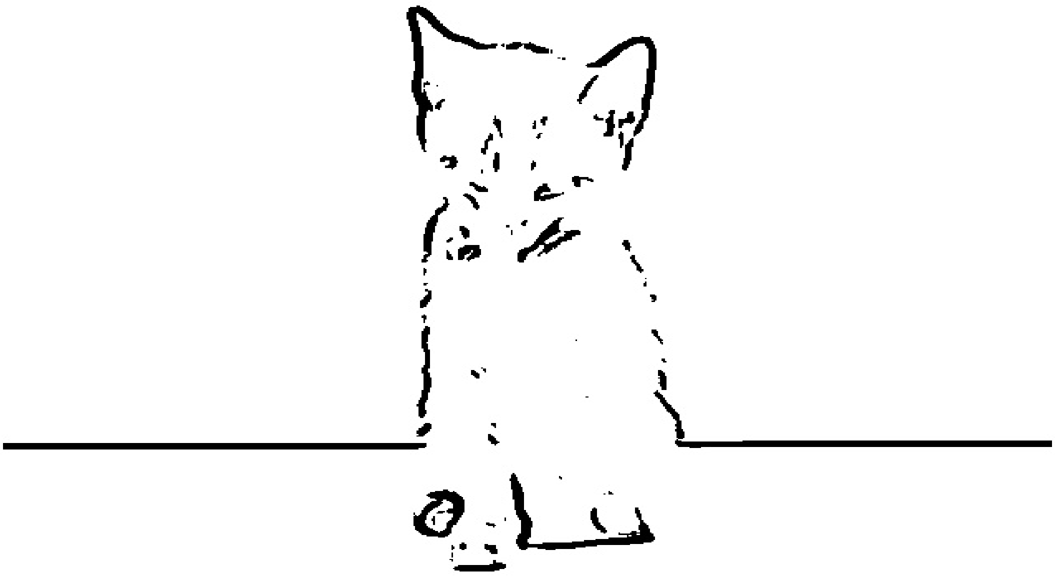

Figure 12: **Post OTSU filtering for R channel using textures**

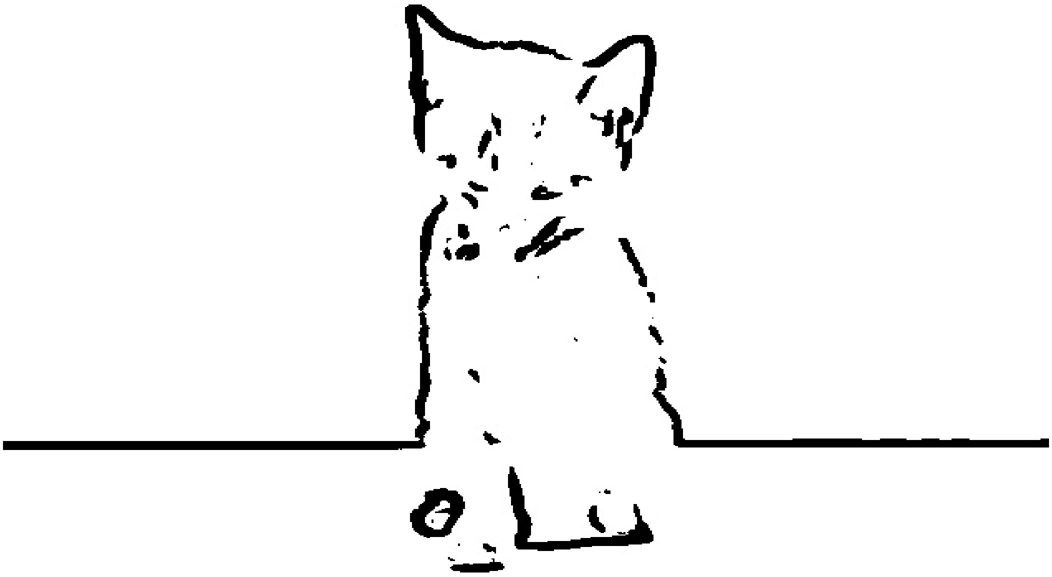

Figure 13: **Post OTSU filtering for G channel using textures**

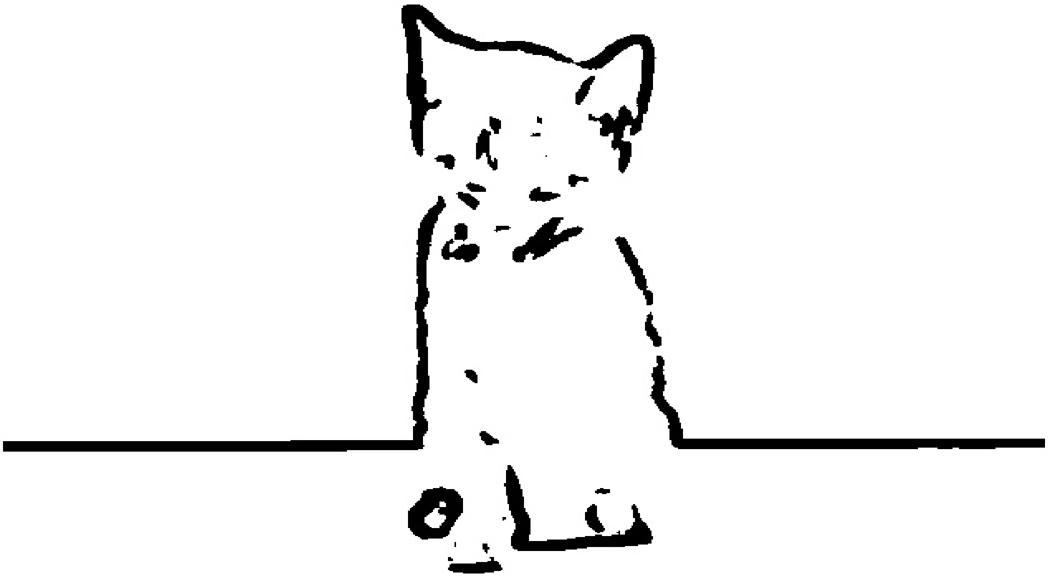

Figure 14: **Post OTSU filtering for B channel using textures**

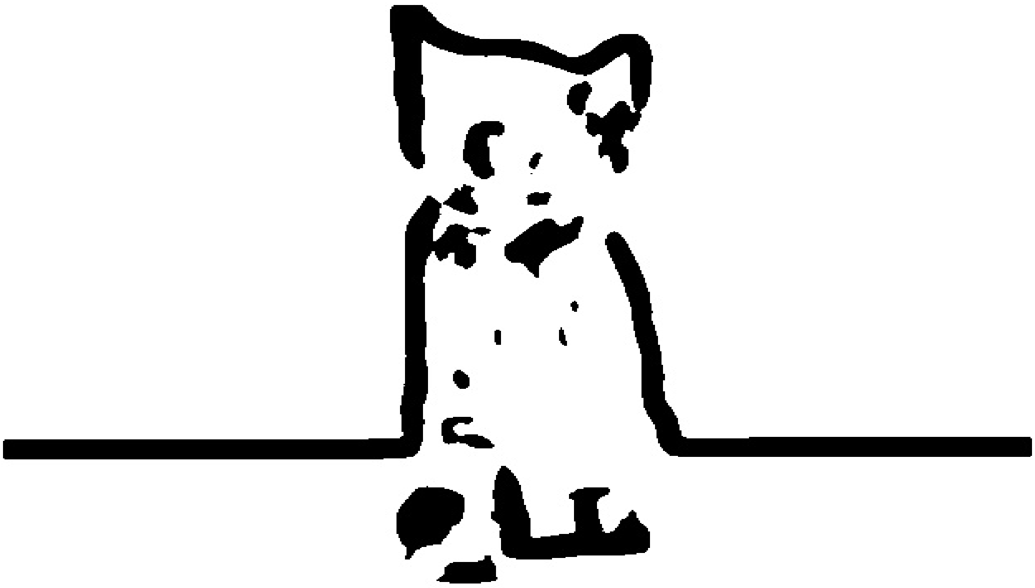

Figure 15: **Post OTSU filtering for all channels using texures**

Figure 16: **Contour plot using RGB method**

Figure 17: **Foreground representation using textures**

Figure 18: **Greyscale R channel**

Figure 19: **Greyscale G channel**

Figure 20: **Greyscale B channel**

Figure 21: **Post OTSU filtering for R channel using RGB method**

Figure 22: **Post OTSU filtering for G channel using RGB method**

Figure 23: **Post OTSU filtering for B channel using RGB method**

Figure 24: **Post OTSU filtering for all channels using RGB method**

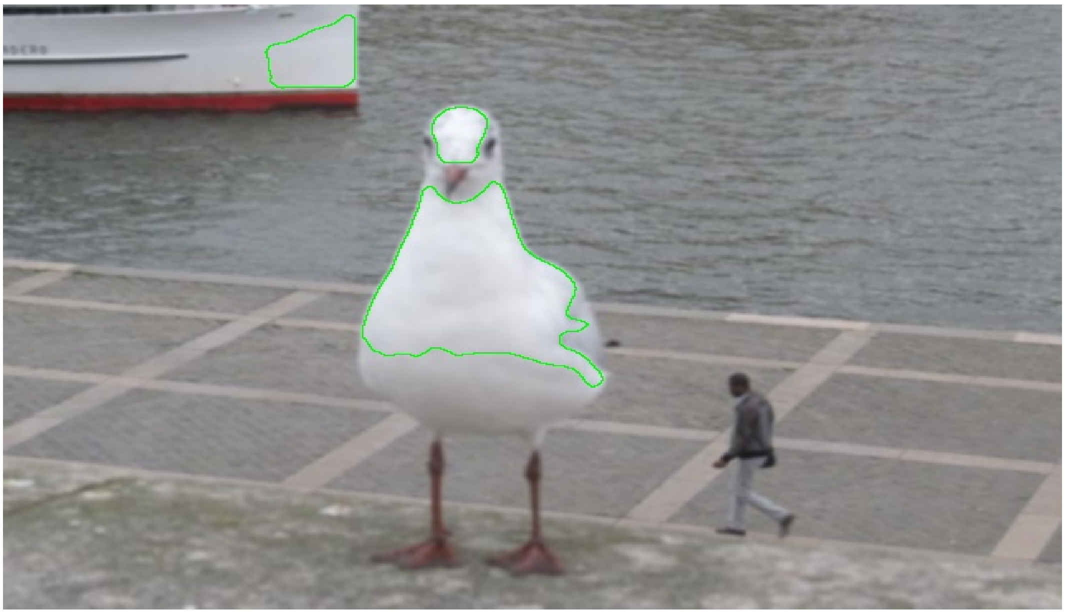

Figure 25: **Contour plot**

Figure 26: **Foreground representation**

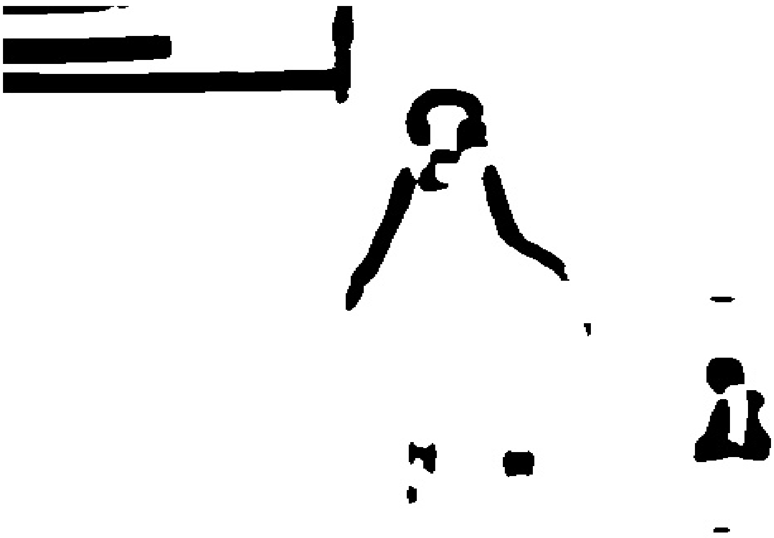

Figure 27: **Post OTSU filtering for R channel using textures**

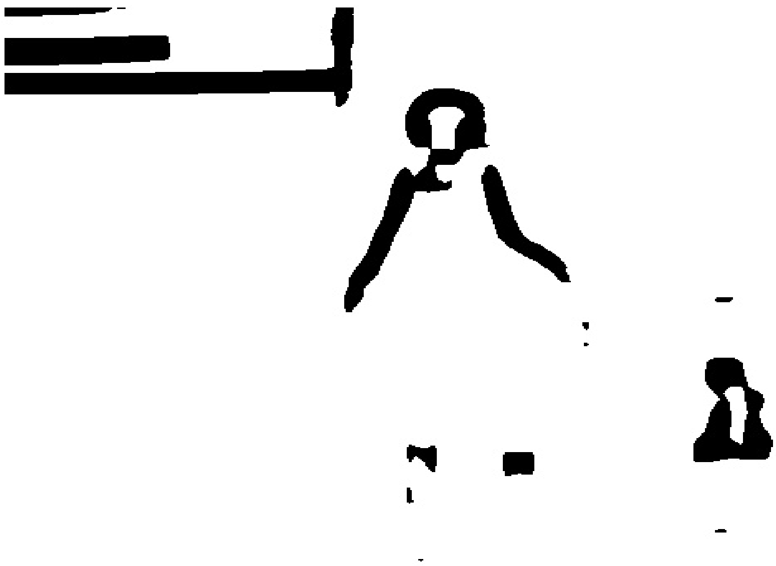

Figure 28: **Post OTSU filtering for G channel using textures**

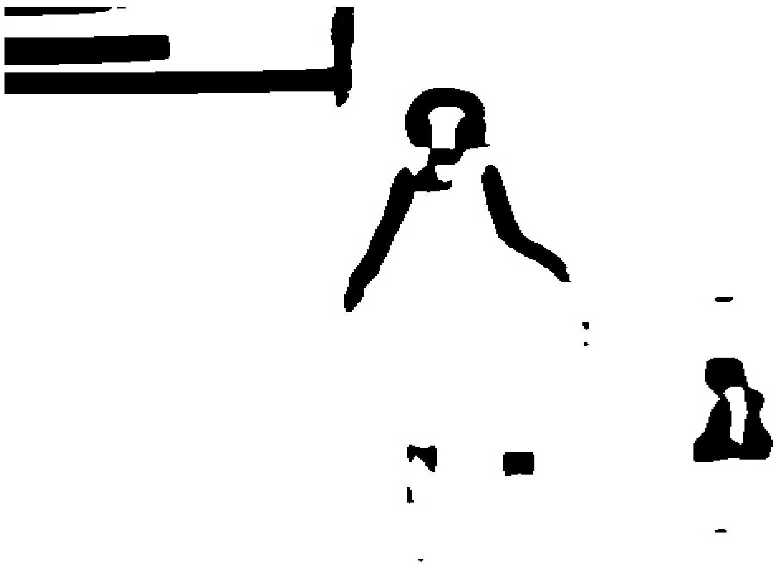

Figure 29: **Post OTSU filtering for B channel using textures**

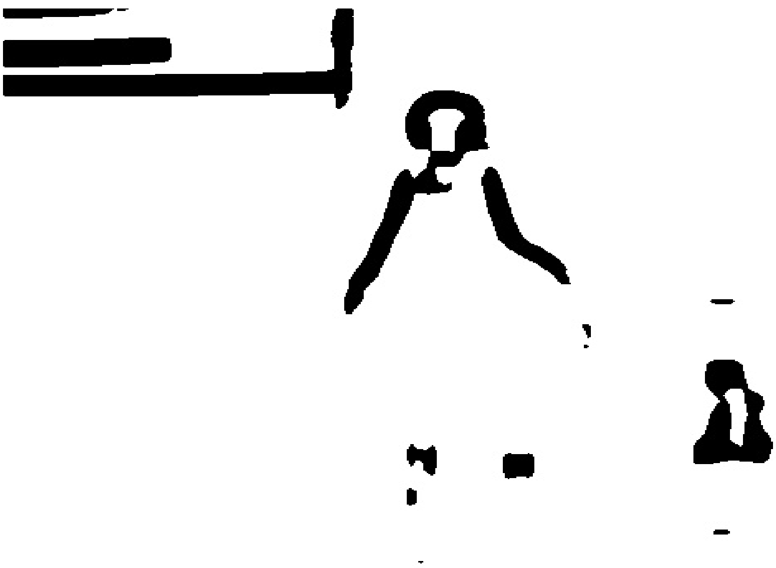

Figure 30: **Post OTSU filtering for all channels using texures**

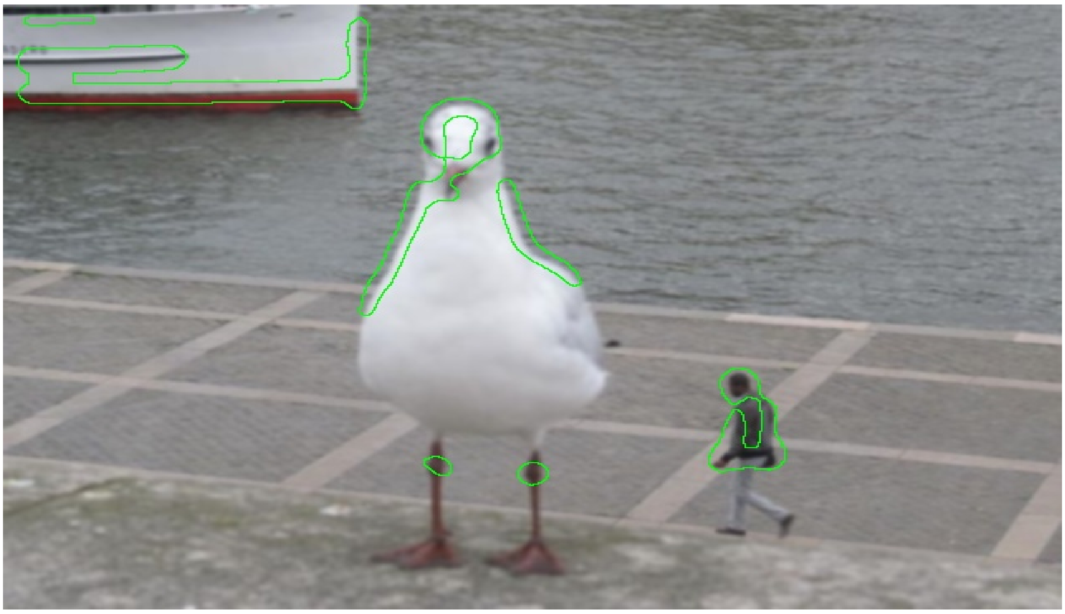

Figure 31: **Contour plot using RGB method**

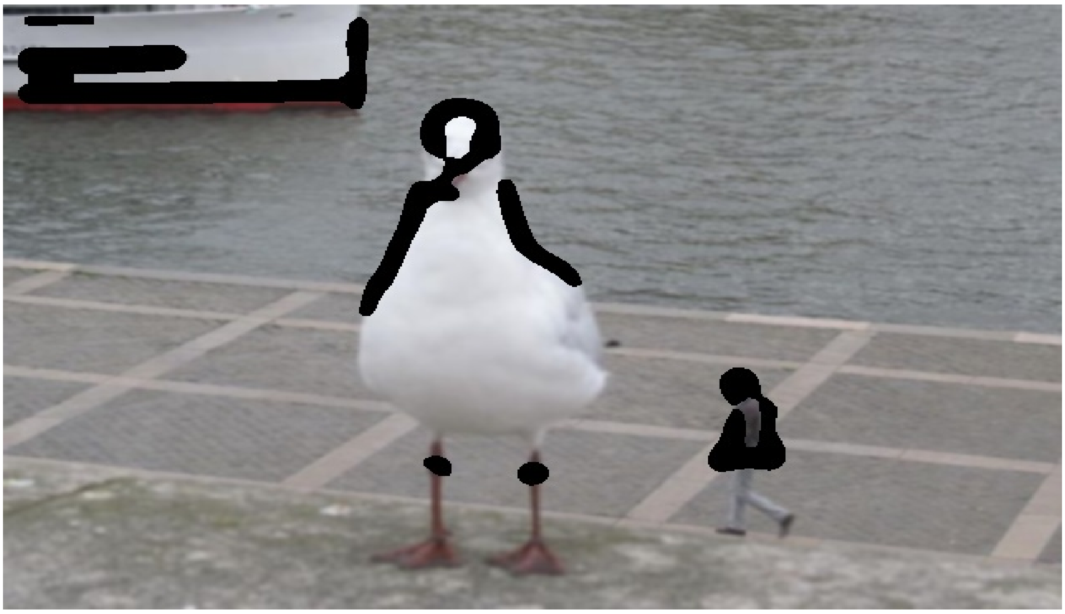

Figure 32: **Foreground representation using textures**

Figure 33: **Greyscale R channel**

Figure 34: **Greyscale G channel**

Figure 35: **Greyscale B channel**

Figure 36: **Post OTSU filtering for R channel using RGB method**

Figure 37: **Post OTSU filtering for G channel using RGB method**

Figure 38: **Post OTSU filtering for B channel using RGB method**

Figure 39: **Post OTSU filtering for all channels using RGB method**

Figure 40: **Contour plot**

Figure 41: **Foreground representation**

Figure 42: **Post OTSU filtering for R channel using textures**

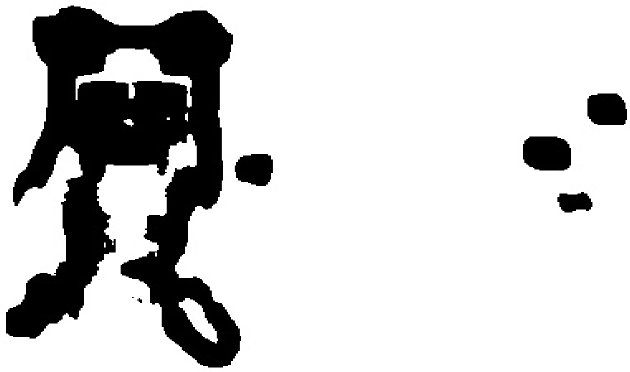

Figure 43: **Post OTSU filtering for G channel using textures**

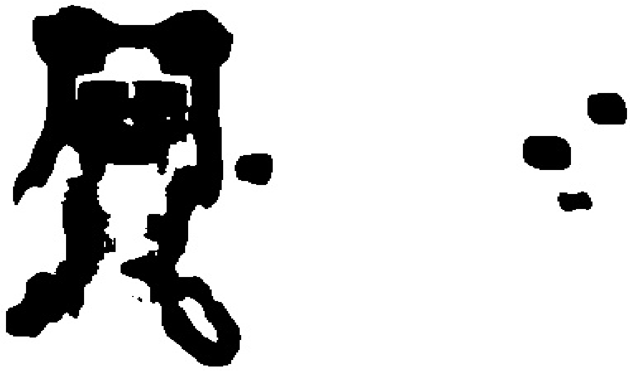

Figure 44: **Post OTSU filtering for B channel using textures**

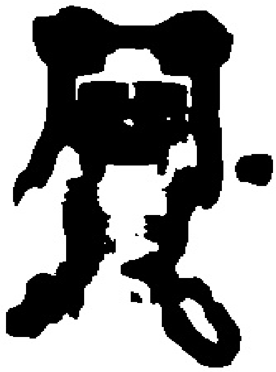

Figure 45: **Post OTSU filtering for all channels using texures**

Figure 46: **Contour plot using RGB method**

Figure 47: **Foreground representation using textures**

Source code

```
"""
Computer Vision - Purdue University - Homework 6

Author : Arjun Kramadhati Gopi, MS-Computer & Information Technology, Purdue University.
Date: Oct 12, 2020


[TO RUN CODE]: python3 segmentimages.py
Output:
    [jpg]: Segmented image which shows the foreground separated from the background.
"""

import cv2 as cv
from matplotlib import pyplot as plt
import numpy as np
import copy

class ImageSegmentation:
    def __init__(self, image_addresses):
        self.image_addresses = image_addresses
        self.originalImages = []
        self.grayscaleImages = []
        self.rgbchannelsdict = {}
        for i in range(len(self.image_addresses)):
            self.originalImages.append(cv.resize(cv.imread(self.image_addresses[i], cv.IMREAD_COLOR), (640, 480)))
            self.grayscaleImages.append(
                cv.resize(cv.imread(self.image_addresses[i], cv.IMREAD_GRAYSCALE), (640, 480)))
```

```python

    def split_channels(self, inputstyle = 'BGR', gaussianblur = True):
        """
        Splits the RGB image into the three channels based on the way the image is read.
        :param inputstyle: BGR is image is read using cv.imread()
        :param gaussianblur: Yes/No image smoothening
        :return: save the individual channels in dictionary
        """
        for queue in range(len(self.originalImages)):
            if inputstyle == 'BGR':
                r,g,b= self.originalImages[queue][:,:,2], self.originalImages[queue][:,:,1], self.originalImages[queue][:,:,0]
            elif inputstyle =='RGB':
                r, g, b = self.originalImages[queue][:, :, 0], self.originalImages[queue][:, :, 1], self.originalImages[queue][:, :, 2]
            if gaussianblur:
                r,g,b = cv.GaussianBlur(r, (5,5), 0), cv.GaussianBlur(g, (5,5), 0), cv.GaussianBlur(b, (5,5), 0)
            cv.imwrite(str(queue)+'Rchanneloriginal.jpg',r)
            cv.imwrite(str(queue) + 'Gchanneloriginal.jpg', g)
            cv.imwrite(str(queue) + 'Bchanneloriginal.jpg', b)
            self.rgbchannelsdict[queue]={'R': r, 'G': g, 'B': b}

    def filter_masks(self, image):
        """
        Filter the output of the OTSU function using a median blur function.
        :param image: Input image
        :return: returns the filtered image
        """
        filteredimage = cv.medianBlur(image, 13)
        filteredimage = filteredimage > 240
        filteredimage = np.array(filteredimage*255, np.uint8)
        return filteredimage

    def otsu_algorithm(self, image, imagequeue, index, method, iterations):
        """
        Iterative Otsu algorithm implementation.
        :param image: Input image channel
        :param imagequeue: Index value of the image position in the list
        :param index: -
        :param method: RGB/Texture based implementation
        :param iterations: Number of iterations to run OTSU
        :return: returns the OTSU threshold value
        """
        otsucutoff = 0
```

```
        otsucutoffinitial = 0
        templist = []
        mask = None
        for iteration in range(iterations):

            start = 0
            # otsucutoffinitial = otsucutoff
            end = 256
            diff = end - start
            channelhistogram = cv.calcHist([np.uint8(image)], [0], mask, [diff], [start, end])
            levels = np.reshape(np.add(range(diff), 1), (diff, 1))
            maxlambda = -1
            otsucutoff = -1
            plt.hist(image.ravel(), diff, [start, end])
            # plt.savefig('histograms/' + str(imagequeue) + str(index) + method)
            plt.show()
            for i in range(len(channelhistogram)):
                m0k = np.sum(channelhistogram[:i]) / np.sum(channelhistogram)
                m1k = np.sum(np.multiply(channelhistogram[:i], levels[:i])) / np.sum(channelhistogram)
                m11k = np.sum(np.multiply(channelhistogram[i:], levels[i:])) / np.sum(channelhistogram)
                omega0 = m0k
                omega1 = 1 - m0k
                mu0 = m1k / omega0
                mu1 = m11k / omega1
                sqauredifference = np.square(mu1 - mu0)
                lambdavalue = omega0 * omega1 * sqauredifference
                if lambdavalue > maxlambda:
                    maxlambda = lambdavalue
                    otsucutoff = i
            mask = np.zeros(image.shape[:2], np.uint8)
            mask[:,:] = image >= otsucutoff
            mask = mask[:,:]*255
            # templist.append(otsucutoff)
            print(otsucutoff)
        # otsucutoff = np.sum(templist)
        # print(otsucutoff)
        return otsucutoff

    def run_otsu_texture(self, imagequeue, iterations, windows):
        """
        Texture based image segmenting implementation. We create three channels based
        on the three window sizes. Each channel consists of the result from convoluting the
        respective window.
        :param imagequeue: Index value of the image position in the list
```

```
        :param iterations: Number of iterations to run OTSU
        :param windows: window sizes to extract texture
        :return: draw and save the images with the contours and the foreground representations
        """
        templist = []
        greyimage = self.grayscaleImages[imagequeue]

        for index, window in enumerate(windows):
            textureimgfinal = np.zeros((self.originalImages[imagequeue].shape))
            textureimg = np.zeros((self.grayscaleImages[imagequeue].shape))
            windowsize = np.uint8((window-1)/2)
            for row in range(windowsize, greyimage.shape[0]-windowsize):
                 for column in range(windowsize, greyimage.shape[1] - windowsize):
                    slidingwindow = greyimage[row-windowsize:row+windowsize+1, column-windowsize:column+windowsize+1]
                    slidingwindow = slidingwindow - np.mean(slidingwindow)
                    textureimg[row, column] = np.var(slidingwindow)
                    # textureimg[row, column] = np.mean((slidingwindow - np.mean(slidingwindow))**2)
            textureimgfinal[:,:,index] = np.uint8(textureimg/textureimg.max()*255)
            # textureimgfinal[:, :, index] = textureimg*255
            image = textureimgfinal[:, :, index]
            otsucutoff = self.otsu_algorithm(image, imagequeue, index, 'texture', iterations)
            resultimage = np.zeros(self.originalImages[imagequeue].shape)
            print(otsucutoff)
            resultimage[:,:, index] = image <= otsucutoff
            templist.append(resultimage)
            resultimage = resultimage[:,:,index]*255
            cv.imwrite(str(imagequeue)+str(window)+ '.jpg', resultimage)
        combinedimage = np.array(np.logical_and(np.logical_and(templist[0][:, :, 0], templist[1][:, :, 1]),
                                                                templist[2][:, :, 2]) * 255, np.uint8)
        cv.imwrite(str(imagequeue)+'combinedTexturebased.jpg', combinedimage)
        resultimage = self.filter_masks(combinedimage)
        self.draw_foreground_save(resultimage, imagequeue, 'texturemethod')
```

```
        self.draw_contour_save(self.extract_contour(resultimage), imagequeue, 'texturemethod')

    def run_otsu_rgb(self, imagequeue, iterations):
        """
        RGB based image segmenting implementation. Using the three split channels, we run
        the OTSU alorithm on each of the channels. We then filter the images based on the
        OTSU cutoff. We combine the three images which will then be used by the contour
        extractor.
        :param imagequeue: Index value of the image position in the list
        :param iterations: Number of iterations to run OTSU
        :return: draw and save the images with the contours and the foreground representations

        """
        templist = []
        for index, channel in enumerate(['R','G','B']):
            image = self.rgbchannelsdict[imagequeue][channel]
            otsucutoff = self.otsu_algorithm(image, imagequeue, index, 'rgb', iterations)
            resultimage = np.zeros(self.originalImages[imagequeue].shape)
            resultimage[:,:, index] = image <= otsucutoff
            templist.append(resultimage)
            resultimage = resultimage[:,:,index]*255
            cv.imwrite(str(imagequeue)+channel + '.jpg', resultimage)
        combinedimage = np.array(np.logical_and(np.logical_and(templist[0][:, :, 0], templist[1][:, :, 1]),
                                                templist[2][:, :, 2]) * 255, np.uint8)
        cv.imwrite(str(imagequeue)+'combinedRGBbased.jpg', combinedimage)
        resultimage = self.filter_masks(combinedimage)
        self.draw_foreground_save(resultimage, imagequeue, 'rgbmethod')
        self.draw_contour_save(self.extract_contour(resultimage), imagequeue, 'rgbmethod')

    def draw_foreground_save(self, image, imagequeue, method):
        """
        Draw the foreground as black using the contour extracted.
        :param image: Input image
        :param imagequeue: Index value of the image position in the list
        :param method: RGB/Texture based implementation
        :return: Save the image
```

```
        """
        r, g, b = self.rgbchannelsdict[imagequeue]['R'], self.rgbchannelsdict[imagequeue]['G'], self.rgbchannelsdict[imagequeue]['B']
        r,g,b = copy.deepcopy(r),copy.deepcopy(g),copy.deepcopy(b)
        truthplot = np.logical_and(np.logical_not(image),1)
        b[truthplot] = 0
        g[truthplot] = 0
        r[truthplot] = 0
        resultimage = cv.merge([b, g, r])
        cv.imwrite(str(imagequeue) +method+ 'foreground.jpg', resultimage)

    def draw_contour_save(self, contours, imagequeue, method):
        """
        Draw the foreground as black using the contour extracted.
        :param contours: Contours we extracted using the extraction algorithm.
        :param imagequeue: Index value of the image position in the list
        :param method: RGB/Texture based implementation
        :return: Save the image
        """
        r,g,b = self.rgbchannelsdict[imagequeue]['R'],self.rgbchannelsdict[imagequeue]['G'],self.rgbchannelsdict[imagequeue]['B']
        r, g, b = copy.deepcopy(r), copy.deepcopy(g), copy.deepcopy(b)
        truthplot = np.logical_and(contours,1)
        b[truthplot] = 0
        g[truthplot] = 255
        r[truthplot] = 0
        resultimage = cv.merge([b,g,r])
        cv.imwrite(str(imagequeue)+method+'contourplot.jpg', resultimage)

    def extract_contour(self, image, style = 1):
        """
        Function to extract the contours from the post-OTSU comined image. For all
        purposes we use the 8-neighbors window size to find the border pixels.
        :param image: Input image of the combined channels
        :param style: 1 for 8 neighbors, 2 for 4 neighbors
        :return: Returns the array of the contours. 255 for border pixels. 0 for others.
        """
        contourplot = np.zeros((image.shape[0],image.shape[1]))
        for row in range(1, image.shape[0]-1):
            for column in range(1, image.shape[1]-1):
                # print(str(column) + " out of " + str(image.shape[0]))
```

```python
                if image[row,column] == 0:
                    if style == 1:
                        window = image[row-1:row+2, column-1:column+2]
                        if 255 in window:
                            contourplot[row, column] = 255
                    elif style ==2:
                        if(image[row+1, column] == 255 or image[row - 1,column] == 255 or image[row, column+1] == 0 or image[row, column-1] == 0):
                            contourplot[row, column] = 255
        return contourplot


if __name__ == '__main__':

    """
    Code starts here.
    """
    tester = ImageSegmentation(['hw6_images/cat.jpg','hw6_images/pigeon.jpg','hw6_images/Red-Fox_.jpg'])
    tester.split_channels()
    iterations_rgb = [1,2,2]
    iterations_texture = [1,1,1]
    windows=[[5,7,9],[9,11,13],[19,21,23]]
    for i in range(len(iterations_rgb)):
        tester.run_otsu_rgb(i, iterations_rgb[i])
    for i in range(len(iterations_texture)):
        tester.run_otsu_texture(i, iterations_texture[i], windows)
```