COMPUTER VISION

ECE 661

HOMEWORK 6

Tanzeel U. Rehman
Email: *trehman@purdue.edu*

0. Watershed Vs. Otsu:

The watershed algorithm is a region based method that uses the morphological operators and measures the catchment basin for each object that appear in the image. It can be used for the cases where the images have multiple and connected/occluded objects together with some form of the distance measure to segment the occluded objects. Watershed algorithm suffers from the signal to noise ratio and thin objects. Otsu threshold on the other hand is the global method that segments the image using a gray level that can sperate the foreground and background. It is usually not well suited for the images with multiple and connected/occluded objects. The watershed algorithm is computationally intensive compared to Otsu. The results of both can suffer from lighting conditions and shades.

1. Introduction:

In this homework Otsu and texture based segmentation approaches were implemented and tested. The brief overview of the implemented approaches is as follows:

1) The Otsu approach was used to identify the gray level from a grayscale image that can best separate the foreground and background. The threshold was obtained for individual RGB channel in an image and were later combined using the logical ‘AND’ operator.

2) The texture based segmentation utilize a sliding window to compute the local variance followed by application of the Otsu thresholding algorithm was used to obtain the best segmenting gray levels by treating each window size as a separate channel of image.

Description of Methods:

A. Otsu thresholding:

Otsu thresholding algorithm uses a clustering mechanism to identify the best grayscale level from the histogram of a grayscale image to segment the background and foreground. The algorithm first constructs the grayscale histogram. Let the image have 256 gray levels then the 256 bins histogram can be given by Eq. 1. Let us assume that a level *k* can best segment the image, then we denote all the pixels with a gray value less than *k* as class 0 (*C0*) and other as class 1 (*C1*). The probability of *C0* and *C1* can be denoted by ω_0 and ω_1 and can be given by Eq. 2. The mean for each class (μ_0, μ_1) and for the overall data (μ_T) can be Eq. 3. Now, we can compute the between class variance as Eq. 4.

$$pi = \frac{ni}{N} \quad \text{(Eq.1)}$$

$$\omega_0 = \sum_{i=0}^{k-1} pi = \sum_{i=0}^{k-1} \frac{ni}{N} \quad \text{(Eq.2)}$$

$$\omega_1 = 1 - \omega_0$$

$$\mu_0 = \sum_{i=0}^{k-1} \frac{i * pi}{\omega_0} \quad \text{(Eq.3)}$$

$$\mu_1 = (\mu_T - \omega_0\mu_0)/\,\omega_1$$

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

$$\sigma_b^2 = \omega_0(\mu_0 - \mu_T)^2 + \omega_1(\mu_1 - \mu_T)^2 = \omega_0\omega_1(\mu_1 - \mu_T)^2 \quad \text{(Eq. 4)}$$

Where, *ni* is the number of pixels at the *ith* gray level and *N* is the total number of pixels. The optimum threshold is selected by iterating the level *k* from 0 to 255 and identifying the level that maximizes the σ_b^2.

B. RGB segmentation:

1) To perform the RGB based segmentation, we first partitioned the RGB image into three different grayscale channels.

2) Then we used the Otsu thresholding algorithm to obtain the segmentation masks for individual channels. Let these masks be mask_1, mask_2 and mask_3.

3) We finally combined these masks to obtain the final composite mask using logical 'AND' operator i.e., mask_1 & mask_2 & mask_3 to best separate the foreground and background.

4) As the final quality of the segmentation mask is image dependent, therefore, we implemented the iterative Otsu thresholding. In every following iteration the mask for specific channel from previous iteration was used. For obtaining the composite mask, different iterations can be provided as a list, where each element of list representing the iteration for specific channel.

C. Texture based segmentation:

1) For the texture based segmentation, we first converted the RGB image into grayscale image using the opencv function.

2) We then used a sliding window having a variable size to compute the local variance around the current anchor position. The resultant image represented the variance of gray scale in original image around the anchoring pixel. For this assignment we used 3,5,7 as window sizes.

3) We applied the Otsu thresholding algorithm to obtain the binary mask for each window size. Finally, we treated each window size as a different channel and combined three masks as per point 3 and 4 in section C of this report.

4) The parameters can be adjusted same as in the last section.

D. Contour Extraction:

To extract the contours, we first cleaned the binary mask obtained using either RGB or Texture segmentation by performing morphological operation such as dilation and erosion. The erosion was performed to remove the noise in the background which is incorrectly segmented as foreground. The dilation was performed to fill the holes present in the foreground due to improper segmentation. The size of kernel used for both dilation and erosion is dependent on the images, therefore was manually tuned for individual image. After applying morphological operators, the contours are extracted as follows:

1) If the pixel under consideration is 0, then it represents the background and will not be used further.

2) If the pixel under consideration is 255 and all 8 neighbors are also 255, then this pixel will also not be used further as it belongs to cluster of foreground and is not on the periphery of the foreground mask.

3) If the pixel under consideration is 255 and all 8 neighbors are not 255, then this the pixel on periphery and will be part of the contour.

E. Assumptions and Notes:

a) For the image of Cat with RGB based segmentation, number of iterations for the blue, green and red channels were 1, 1, and 2, respectively. The kernel size for erosion and dilation was 2 × 2. The masks for BGR were combined using "And" operator.

b) For the image of Cat with texture based segmentation, number of iterations for the blue, green and red channels were 1, 1, and 2, respectively. The kernel size for erosion and dilation was 2 × 2. The masks for BGR were combined using "And" operator.

c) For the image of pigeon with RGB based segmentation, number of iterations for the blue, green and red channels were 1, 1, and 2, respectively. The kernel size for erosion and dilation was 2 × 2. The masks for BGR were combined using "And" operator.

d) For the image of pigeon with texture based segmentation, number of iterations for the blue, green and red channels were 1, 1, and 1, respectively. The kernel size for erosion and dilation was 2 × 2. The masks for BGR were combined using "And" operator.

e) For the image of fox with RGB based segmentation, number of iterations for the blue, green and red channels were 1, 2, and 2, respectively. The kernel size for erosion and dilation was 2 × 2. The masks for BGR were combined using "And" operator.

f) For the image of fox with texture based segmentation, number of iterations for the blue, green and red channels were 1, 1, and 1, respectively. The kernel size for erosion and dilation was 2 × 2. The masks for BGR were combined using "And" operator.

F. Observations:

1) The results of final segmentation are dependent on the quality of the image, the lighting conditions, shading conditions and degree of contrast difference between foreground and background.

2) There are multiple parameters that needs to be adjusted image by image for both methods, therefore, it is difficult to conclude that specific method is better.

3) For the cat image, we observe that no specific difference found between table and cat when using RGB segmentation, therefore, the final mask was not able separate the table from the cat. Also, the impact of shading can also be seen on the front area of cat especially for the blue and green channel. For the Texture based segmentation, we can see that the edge of the table was picked up as foreground, this might be due to limited

local variance. Also, the foreground pixels belonging to cat are mis-segmented as the background by texture based segmentation. For this image, the RGB based segmentation mis-segmented background into foreground, however the texture based segmentation did vice-versa.

4) For pigeon image, since we have multiple background pixels which are white in color, therefore both RGB and texture based methods did not yield very good results. In RGB segmentation, we removed the floor stripes using 2^{nd} iteration on the red channel, however, still the white edge of the ship was still in the segmented mask. For the texture based segmentation the background water was also picked up as foreground with actual foreground having holes in it.

5) For the fox image with RGB segmentation, we performed 2 iterations for both green and red channels as the first iteration does not provide clean outputs. Even with 2 iteration on green and red channels, we were not able to segment the green grass and red fox. I believe this is due to the blurring effect caused by the focus of the camera. The texture based segmentation for this image was able to significantly improve the final segmentation mask. However, we still observe some holes in the actual fox foreground.

6) Generally, we observed that with increase in the window size for the texture based segmentation, the coarser results are obtained.

2. Results:

2.1.Task1 (Otsu image segmentation using RGB values):

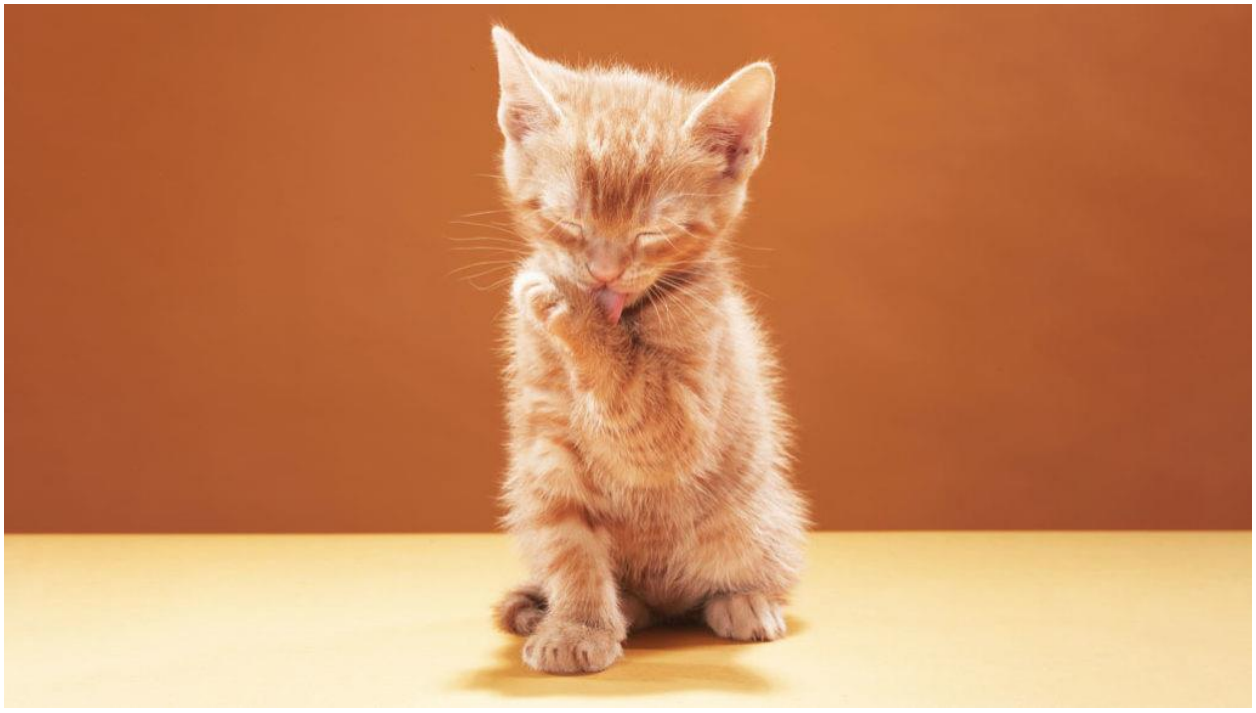

Fig 1: Given image of cat

Fig 2: Foreground generated by the blue channel at 1st iteration with threshold of 112.

Fig 3: Foreground generated by the green channel at 1st iteration with threshold of 156.

Fig 4: Foreground generated by the red channel at 1st iteration with threshold of 210.

Fig 5: Foreground generated by the red channel at 2nd iteration with threshold of 235.

Fig 6: Foreground generated by using BGR masks (at respective iterations of 1, 1 and 2) with 'AND' operation between channel wise mask.

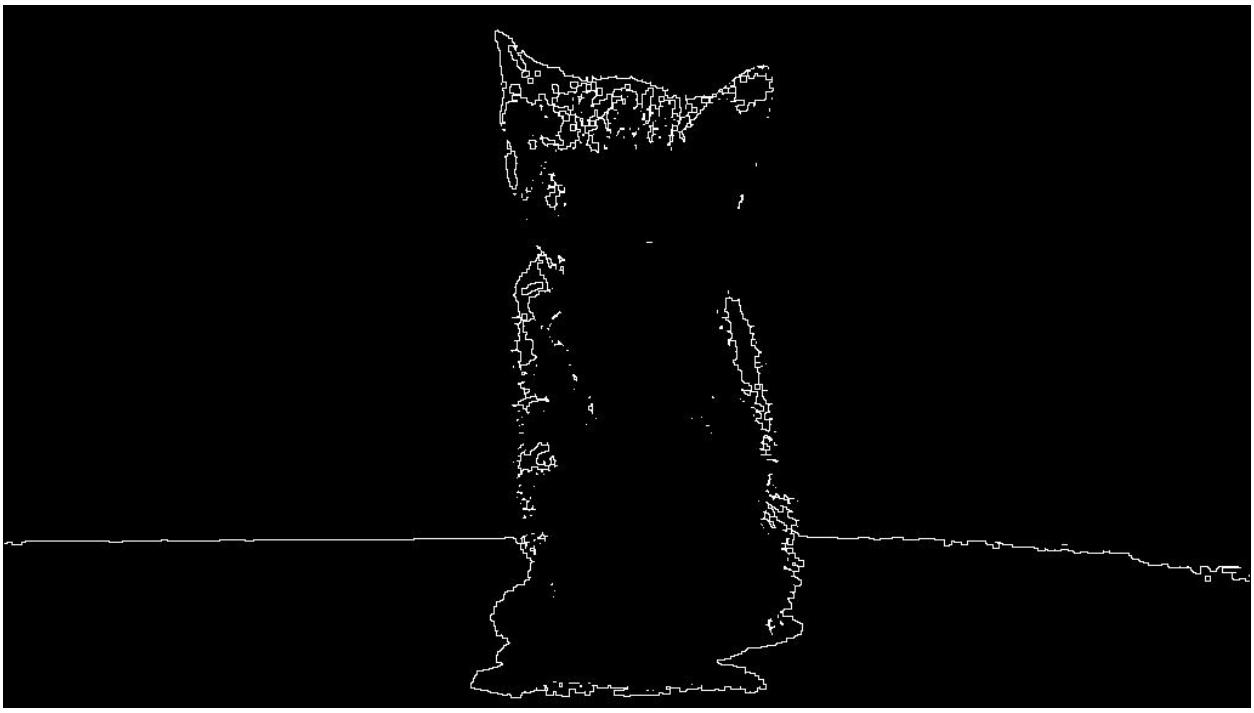

Fig 7: Contours generated by using the mask in Fig 6 with morphological operations for noise removal.

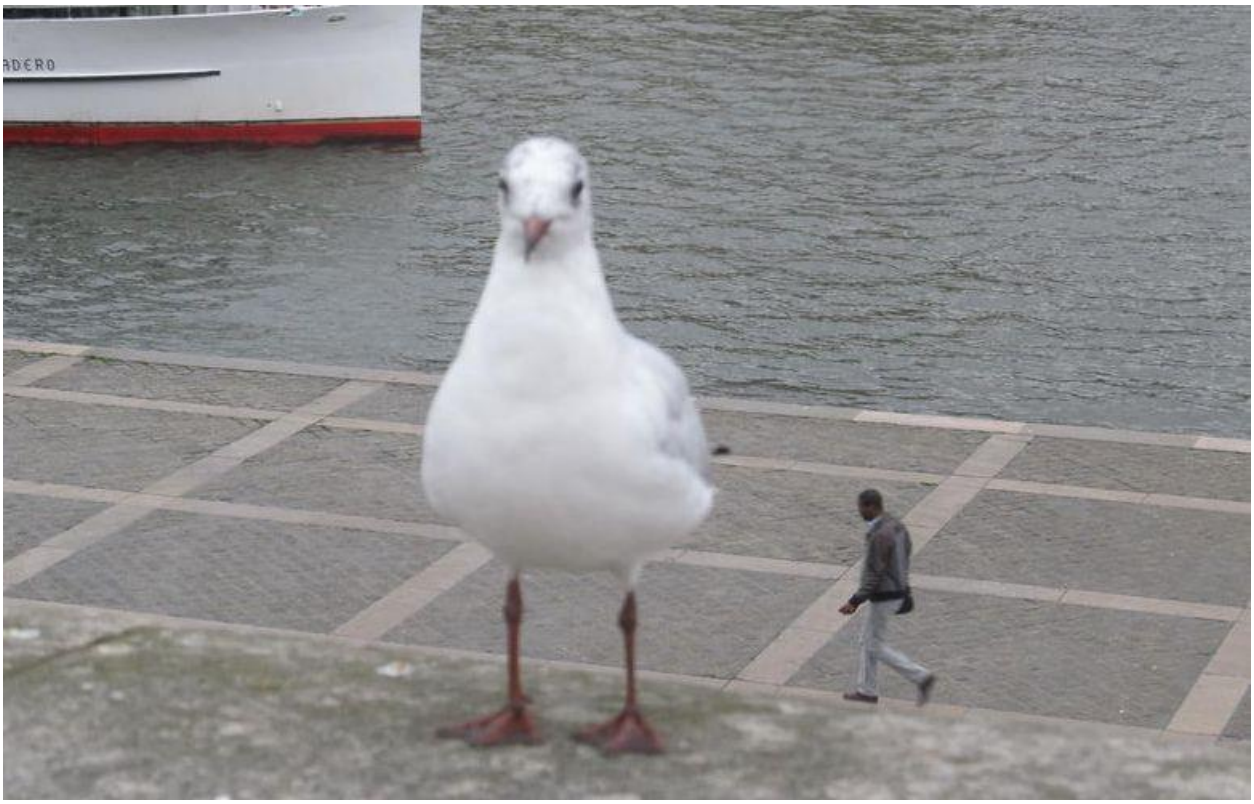

Fig 8: Given image of pigeon

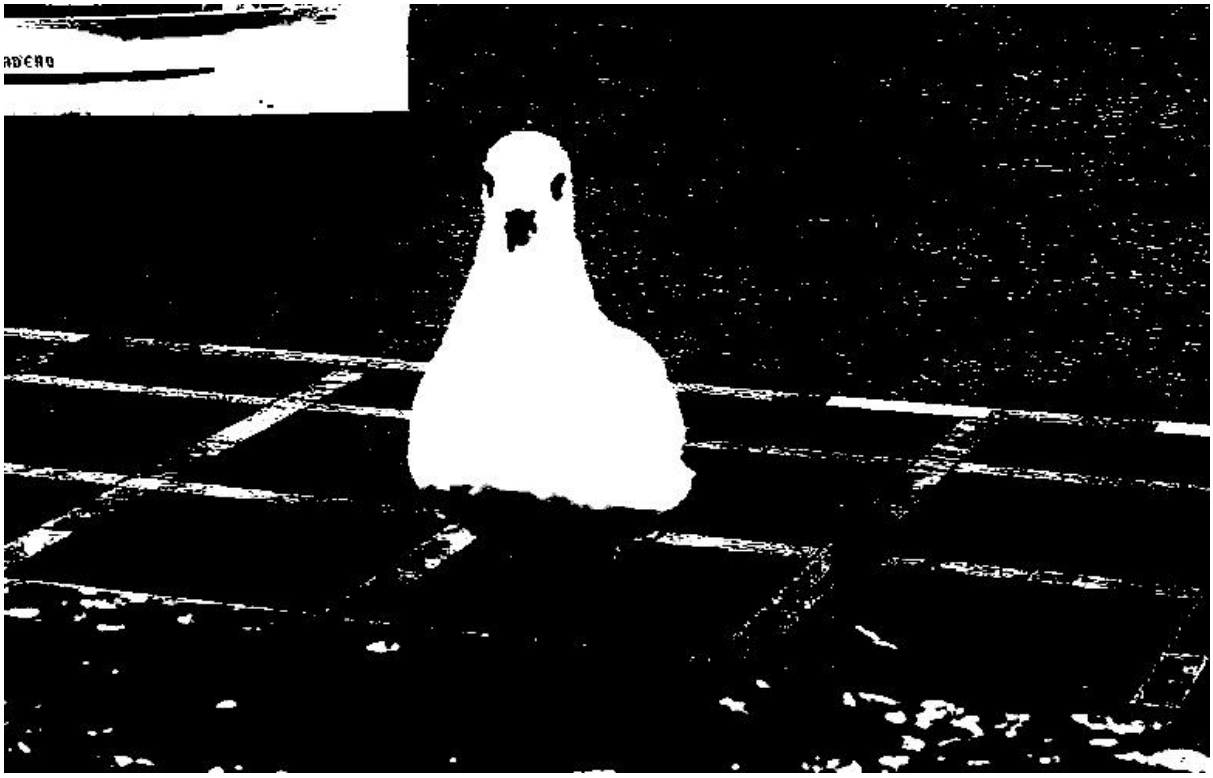

Fig 9: Foreground generated by the blue channel at 1st iteration with threshold of 163.

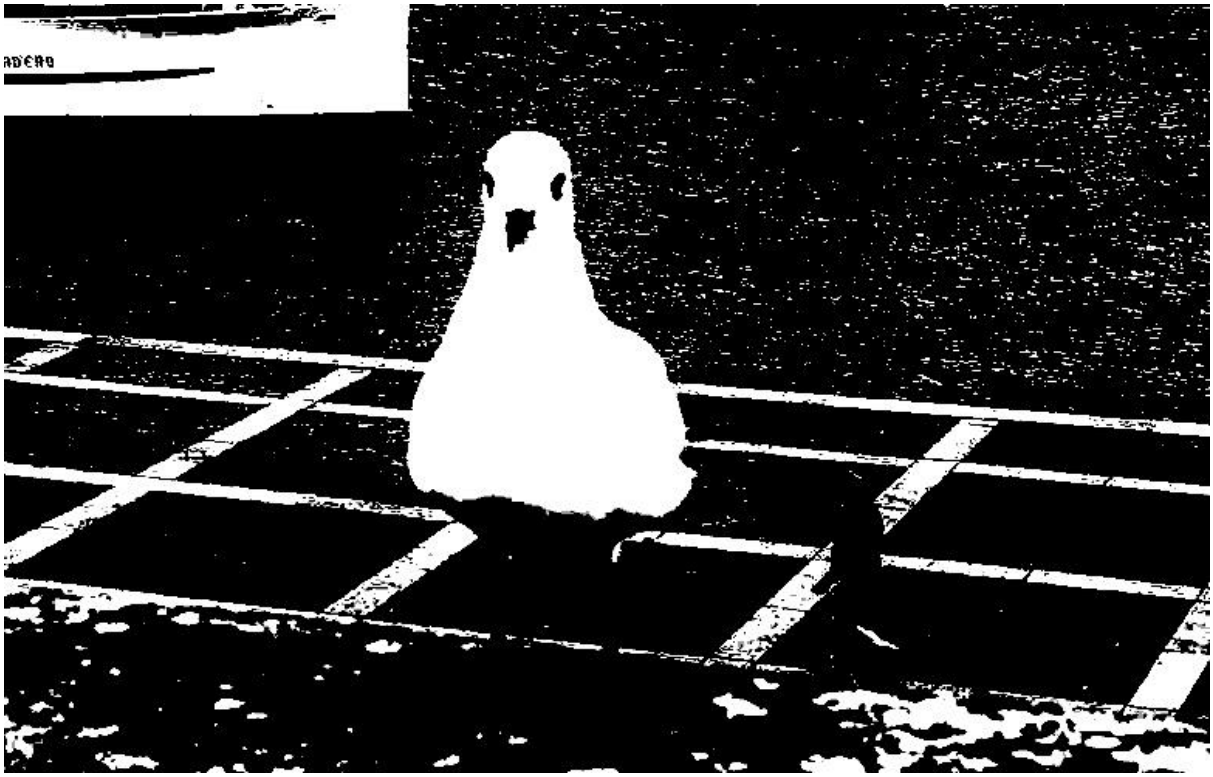

Fig 10: Foreground generated by the green channel at 1st iteration with threshold of 155.

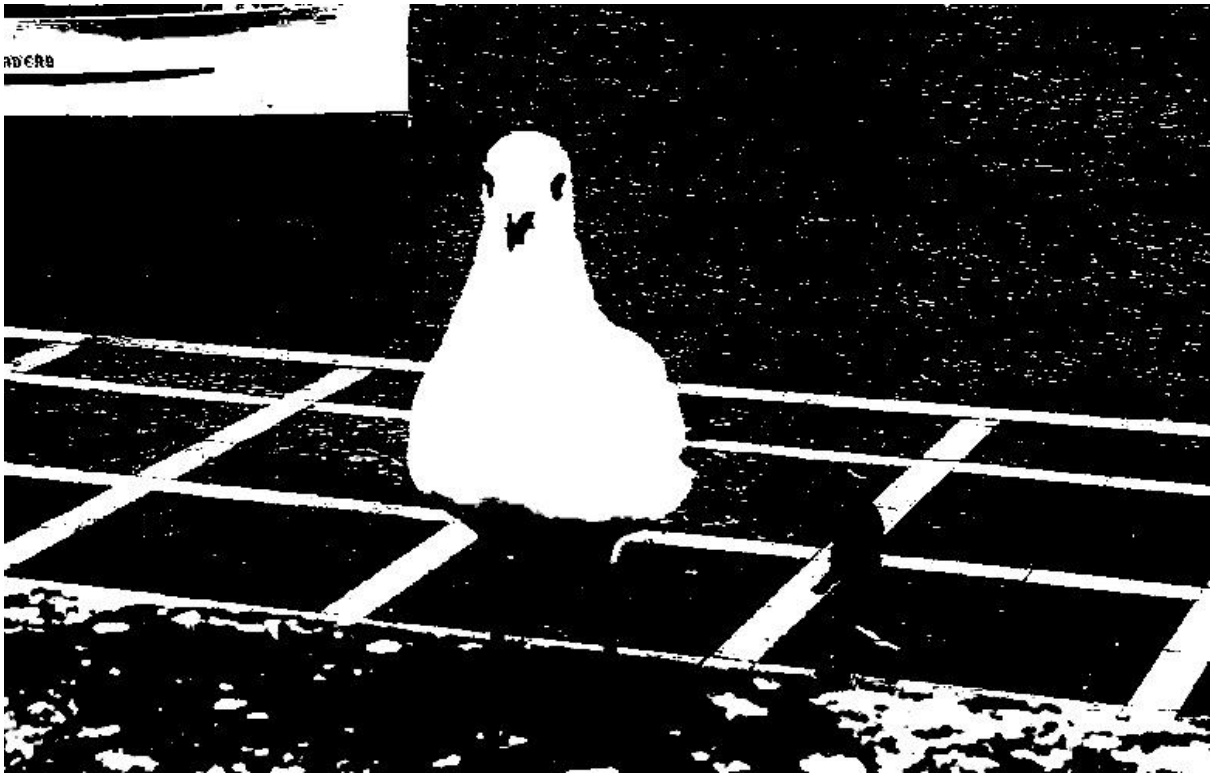

Fig 11: Foreground generated by the red channel at 1st iteration with threshold of 158.

Fig 12: Foreground generated by the red channel at 2nd iteration with threshold of 187.

Fig 13: Foreground generated by using BGR masks (at respective iterations of 1, 1 and 2) with 'AND' operation between channel wise mask.

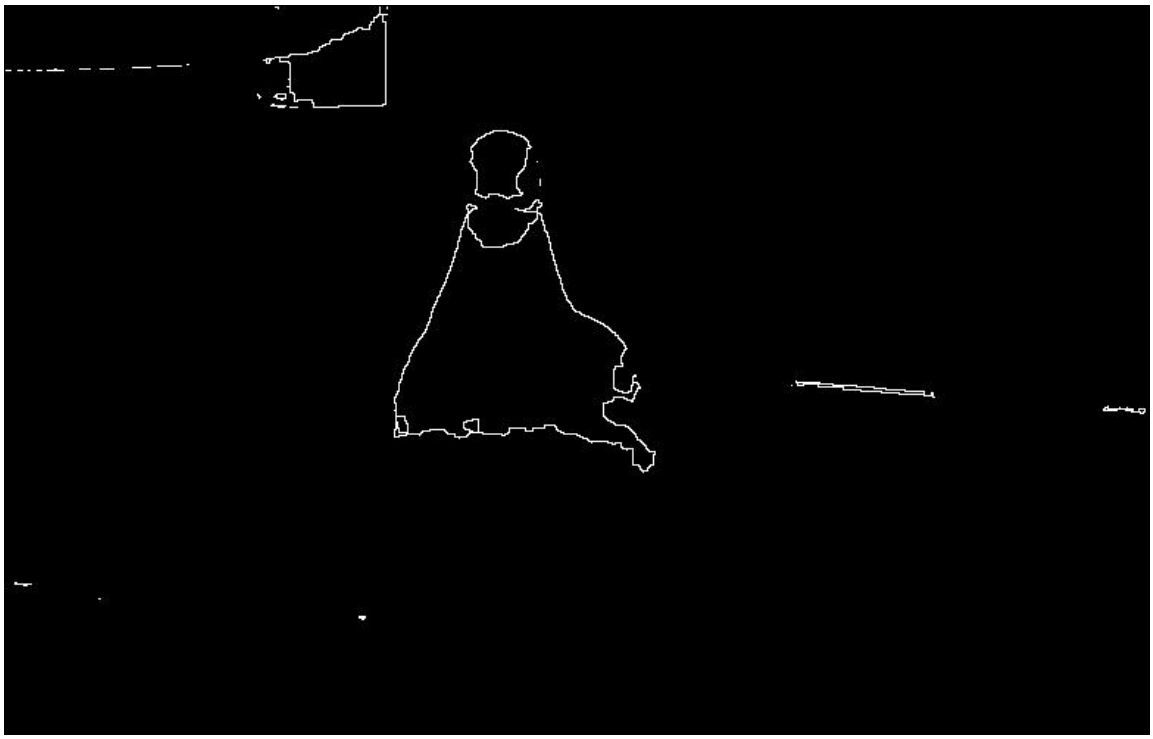

Fig 14: Contours generated by using the mask in Fig 13 with morphological operations for noise removal.

Fig 15: Given image of fox

Fig 16: Foreground generated by the blue channel at 1st iteration with threshold of 44.

Fig 17: Foreground generated by the green channel at 1st iteration with threshold of 109.

Fig 18: Foreground generated by the green channel at 2nd iteration with threshold of 153.

Fig 19: Foreground generated by the red channel at 1st iteration with threshold of 93.

Fig 20: Foreground generated by the red channel at 2nd iteration with threshold of 139.

Fig 21: Foreground generated by using BGR masks (at respective iterations of 1, 2 and 2) with 'AND' operation between channel wise mask.

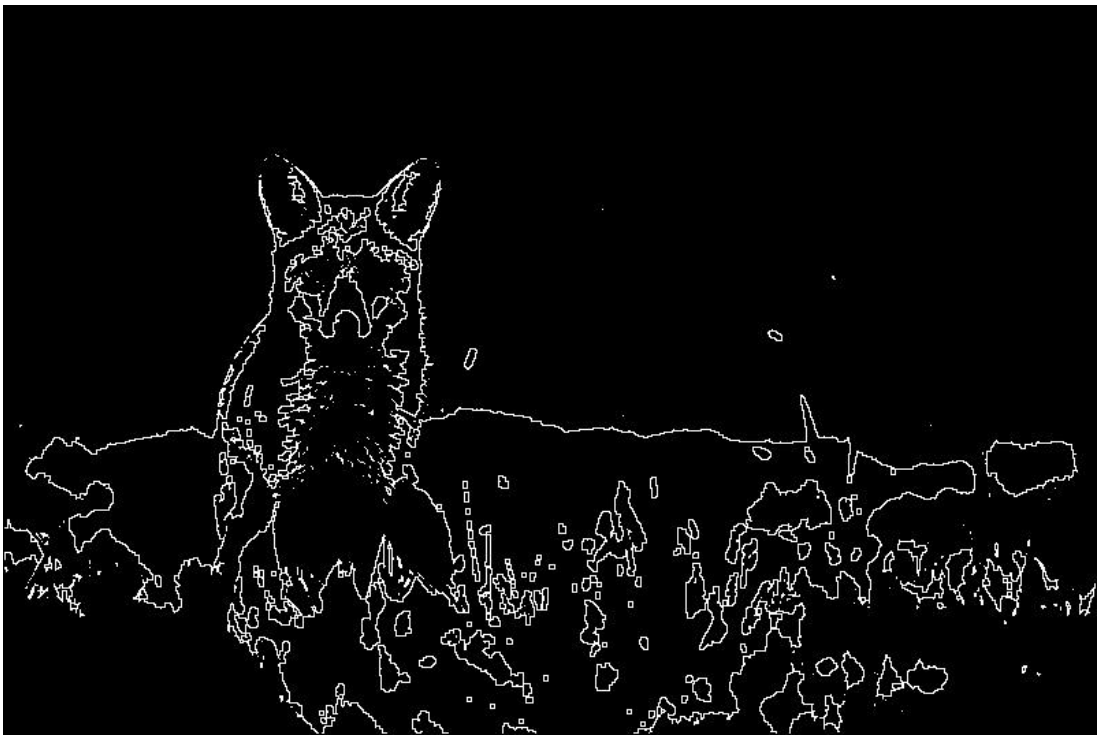

Fig 22: Contours generated by using the mask in Fig 21 with morphological operations for noise removal.

2.2.Task 2 (Texture-based Segmentation):

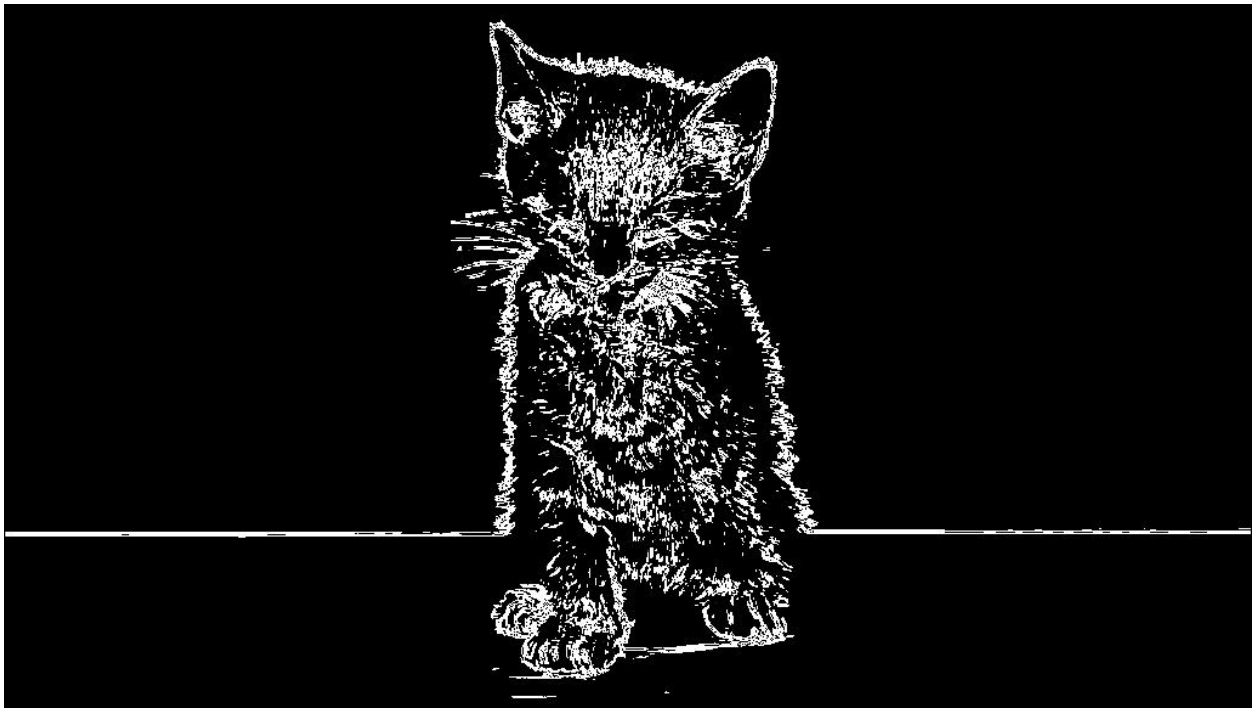

Fig 1: Foreground generated by the window size 3 × 3 at 1st iteration with threshold of 63.

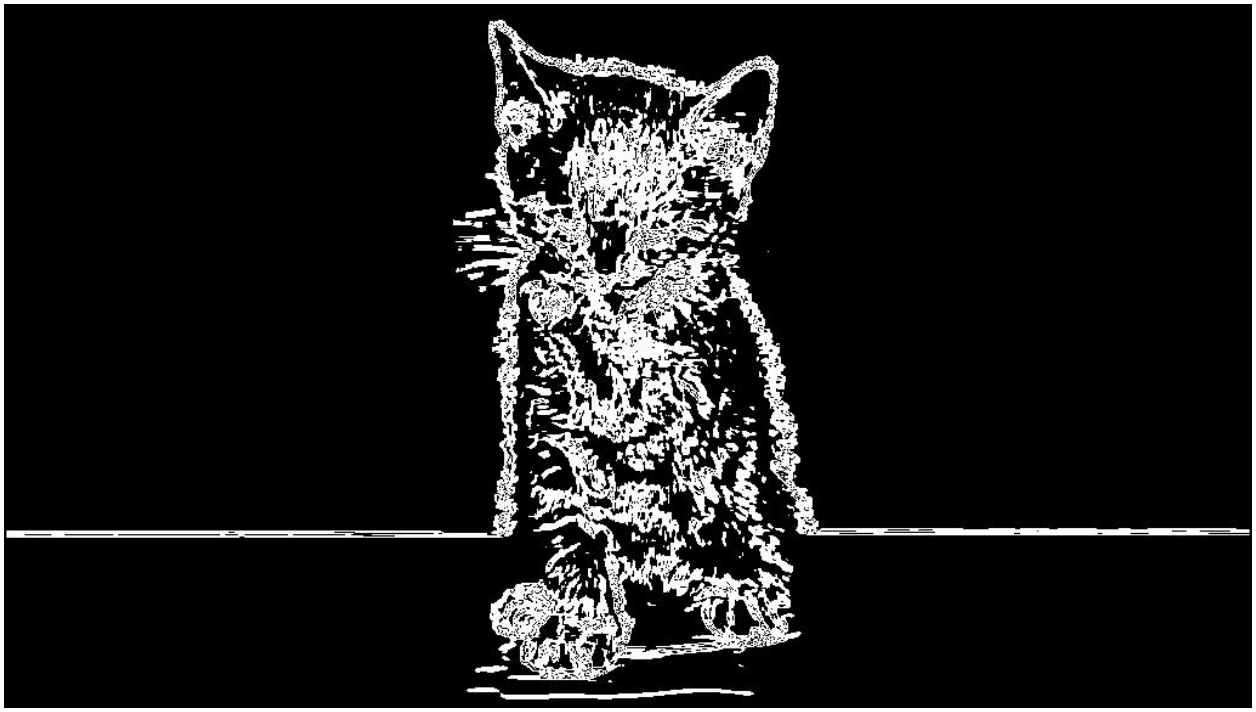

Fig 2: Foreground generated by the window size 5 × 5 at 1st iteration with threshold of 68.

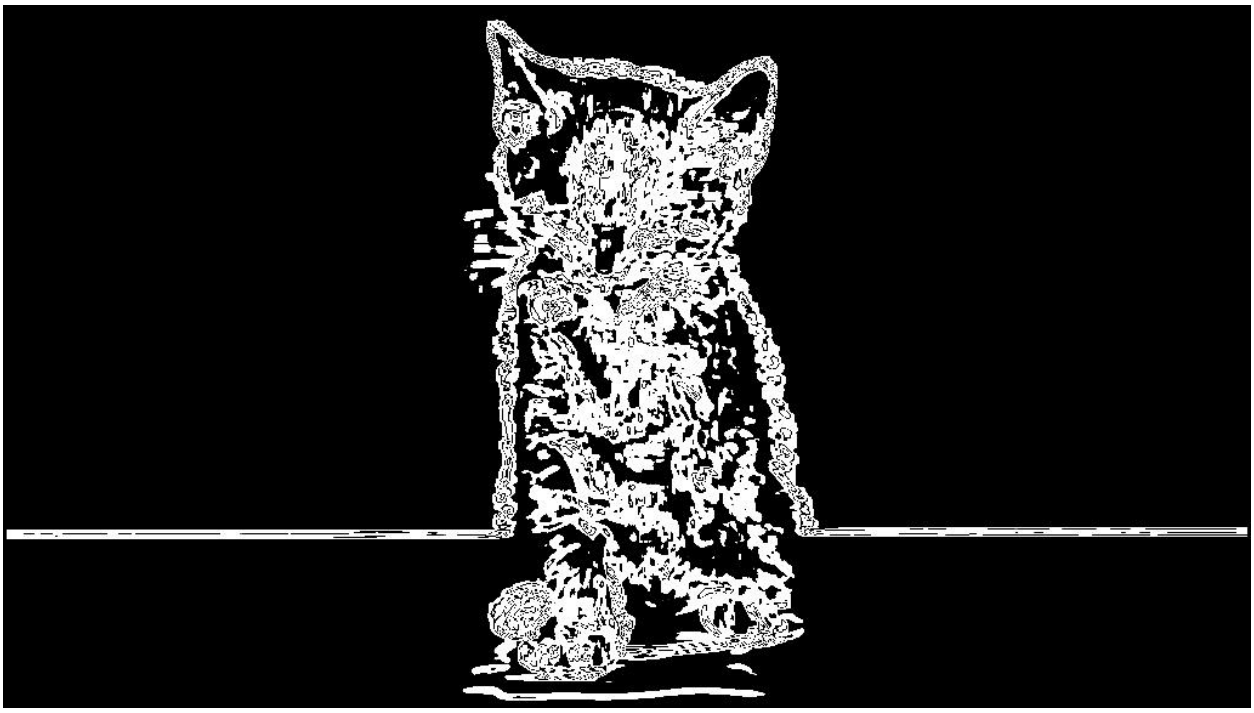

Fig 3: Foreground generated by the window size 7 × 7 at 1st iteration with threshold of 72.

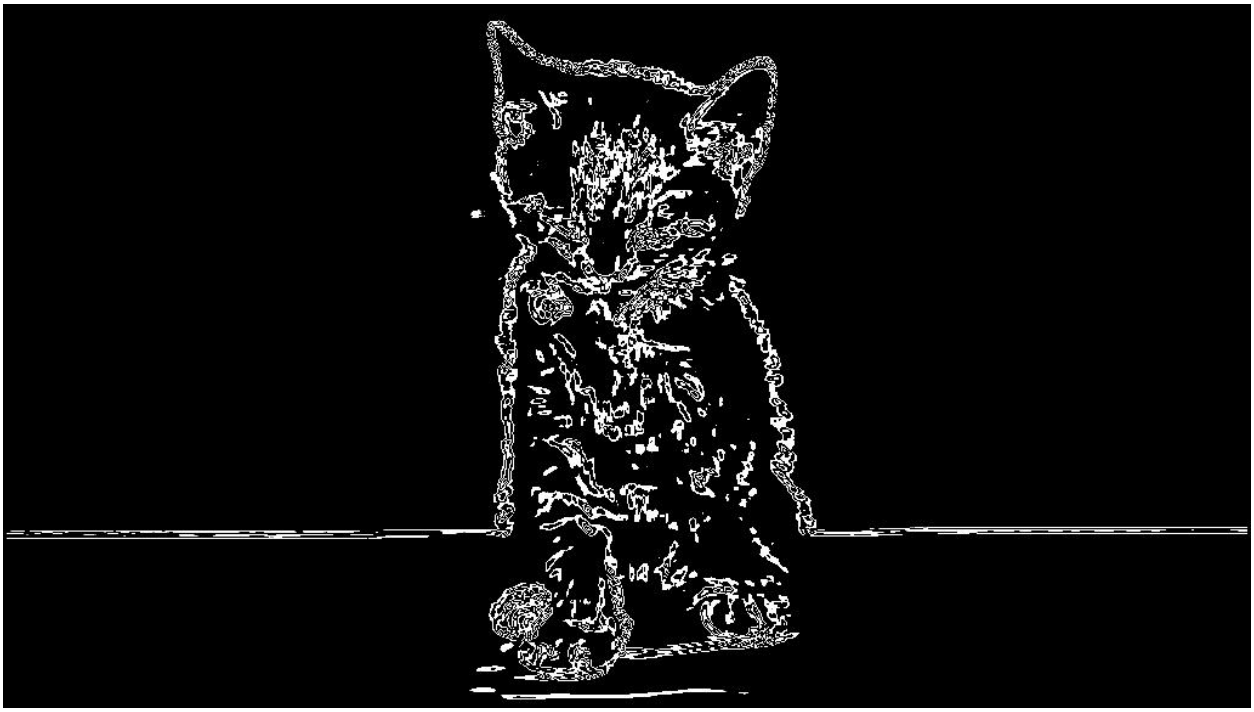

Fig 4: Foreground generated by the window size 7 × 7 at 2nd iteration with threshold of 150.

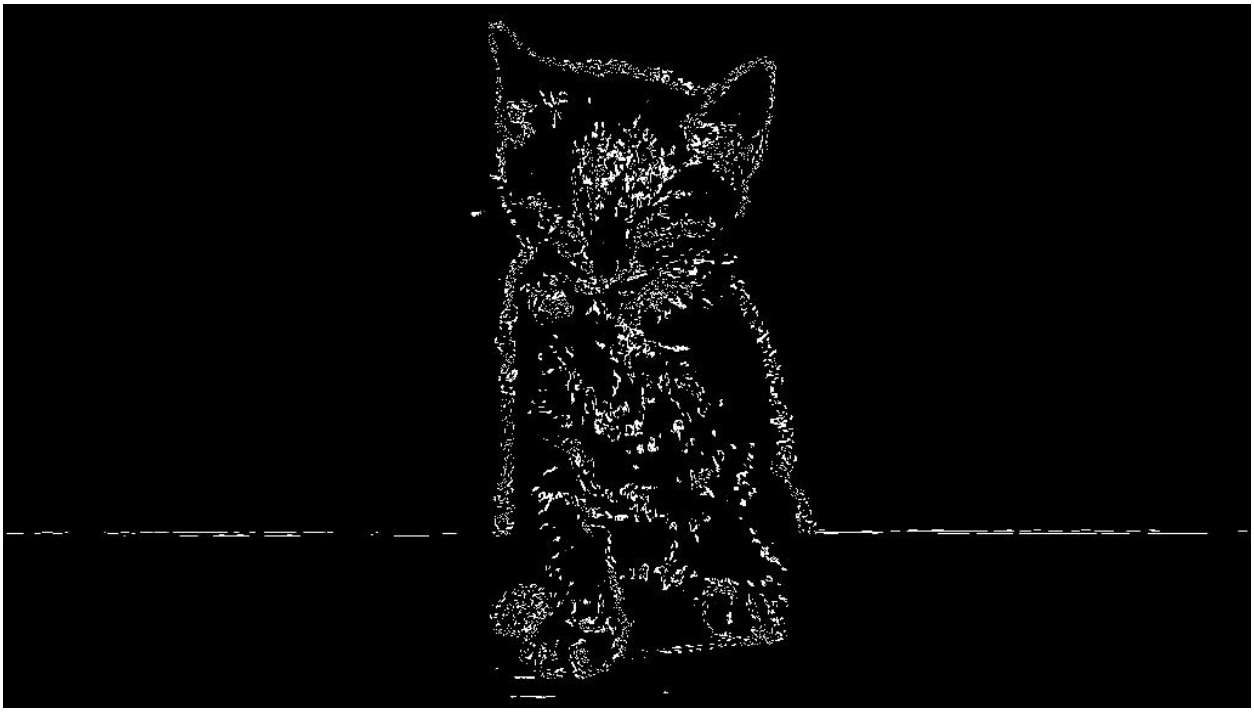

Fig 5: Foreground generated by using masks with different window sizes (at respective iterations of 1, 1 and 2) with 'AND' operation between channel wise mask.

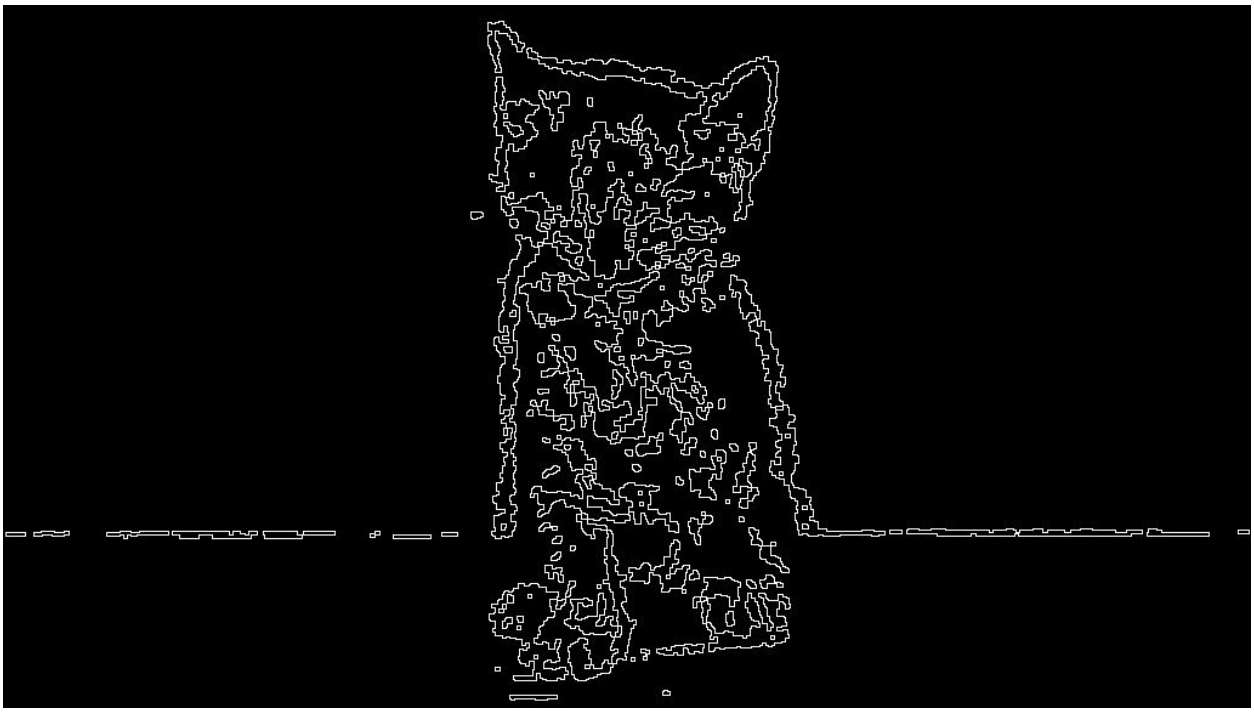

Fig 6: Contours generated by using the mask in Fig 5 with morphological operations for noise removal.

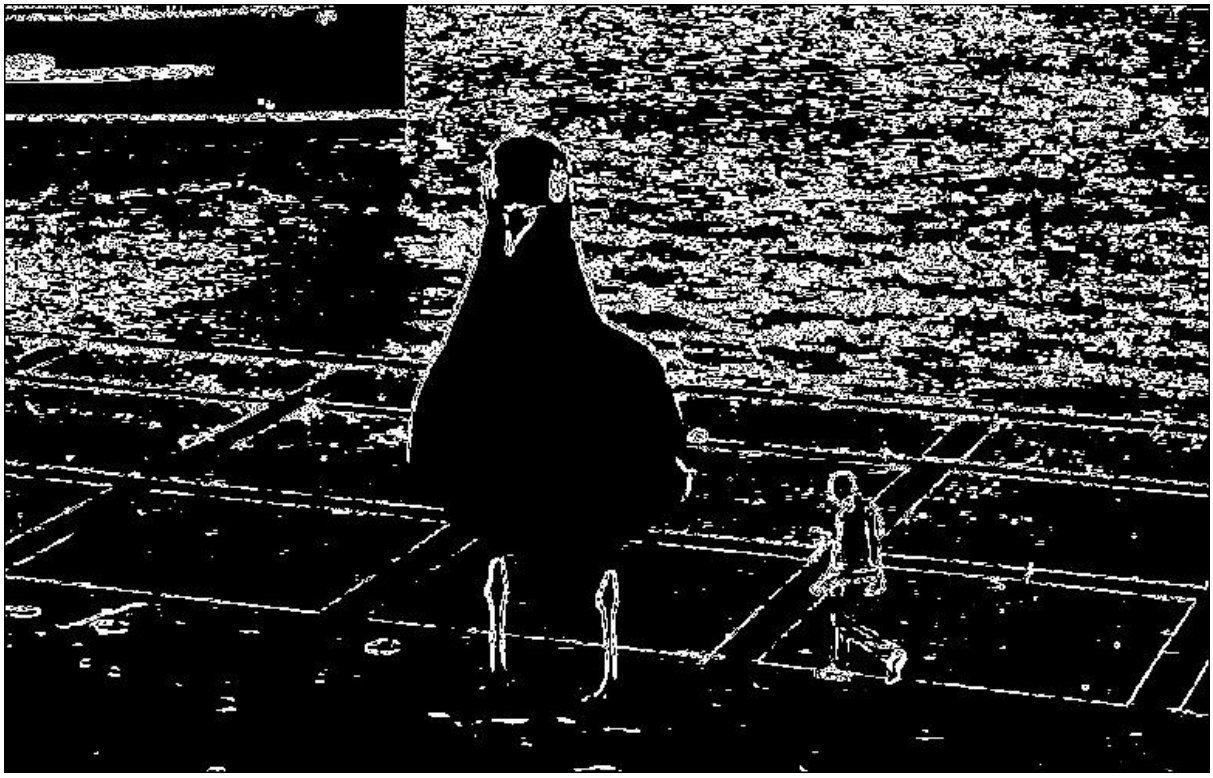

Fig 7: Foreground generated by the window size 3 × 3 at 1st iteration with threshold of 83.

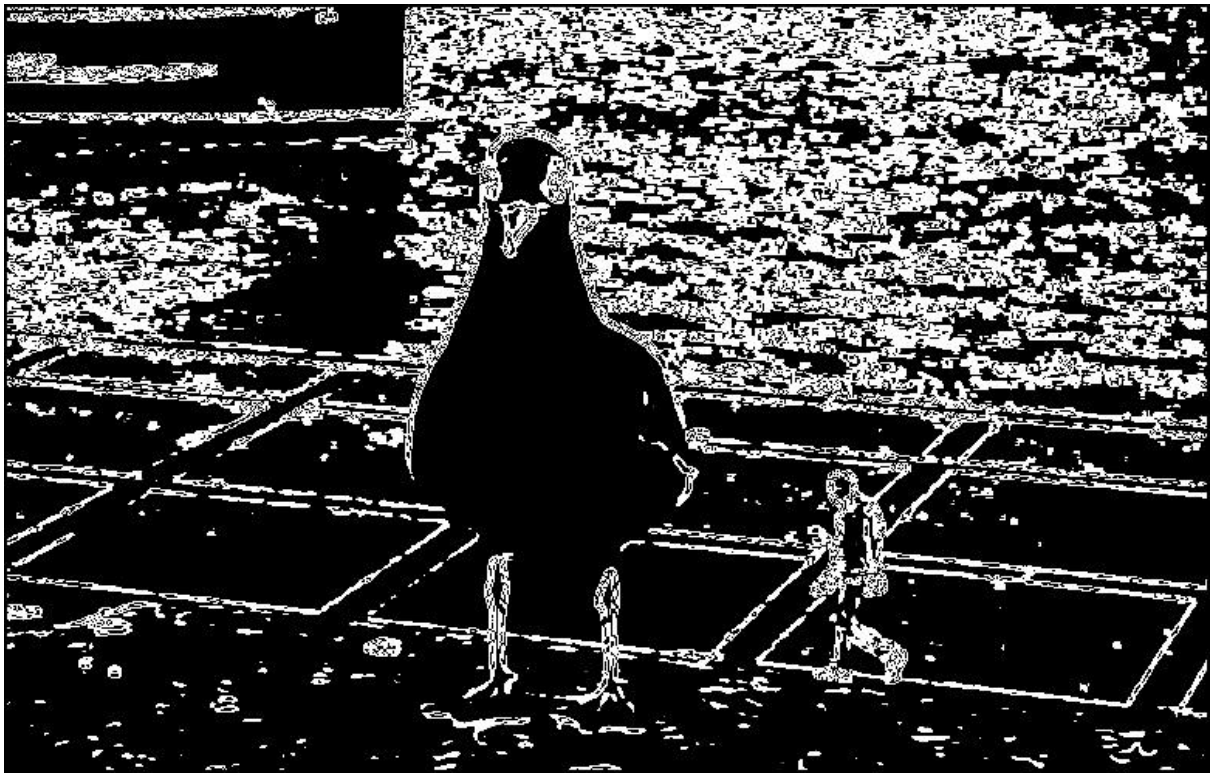

Fig 8: Foreground generated by the window size 5 × 5 at 1st iteration with threshold of 93.

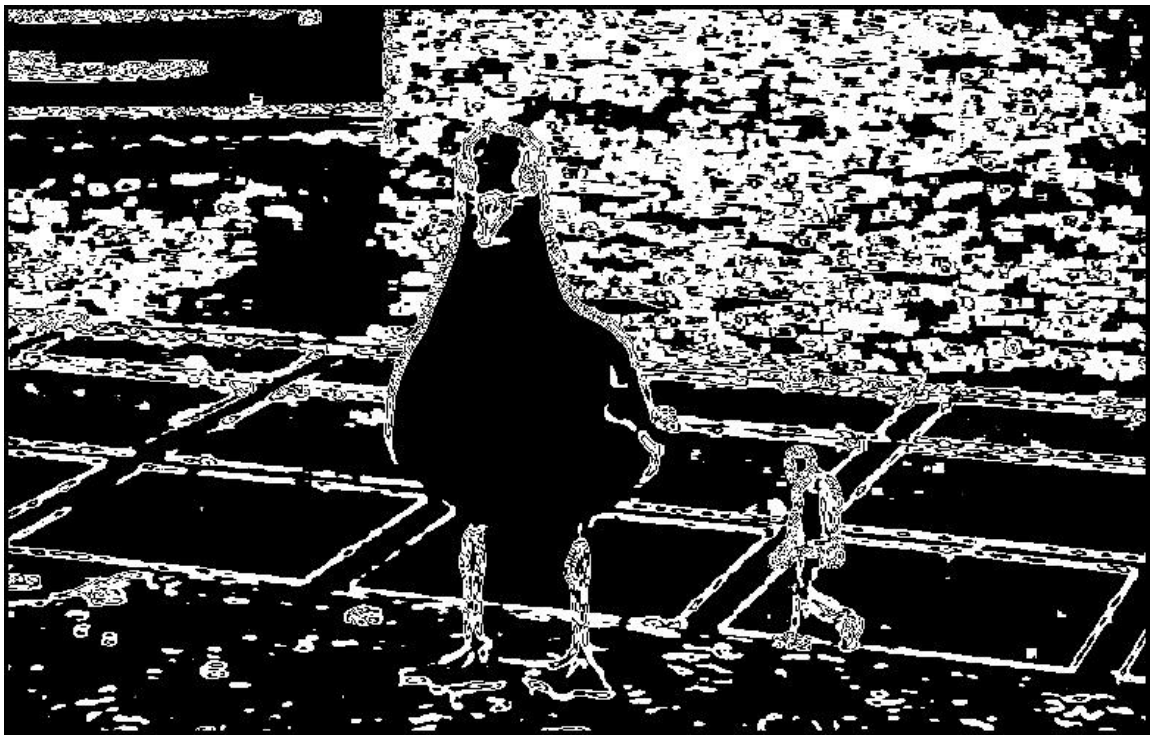

Fig 9: Foreground generated by the window size 7 × 7 at 1st iteration with threshold of 98.

Fig 10: Foreground generated by using masks with different window sizes (at respective iterations of 1, 1 and 1) with 'AND' operation between channel wise mask.

Fig 11: Contours generated by using the mask in Fig 10 with morphological operations for noise removal.

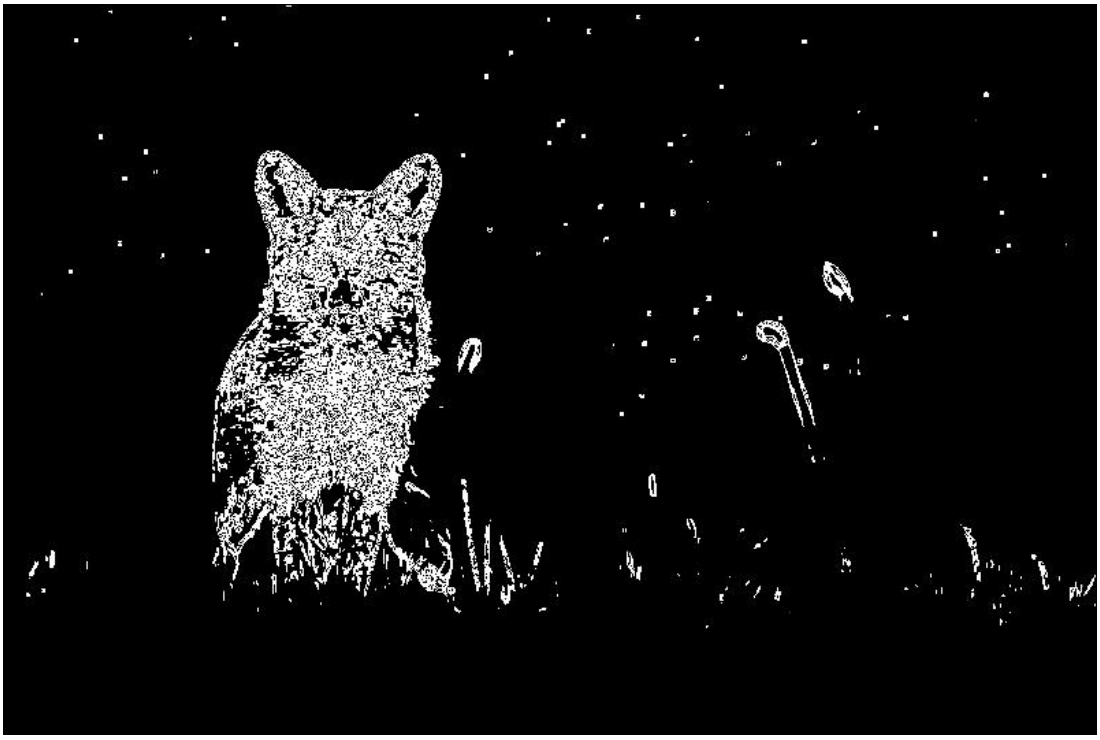

Fig 12: Foreground generated by the window size 3 × 3 at 1st iteration with threshold of 77.

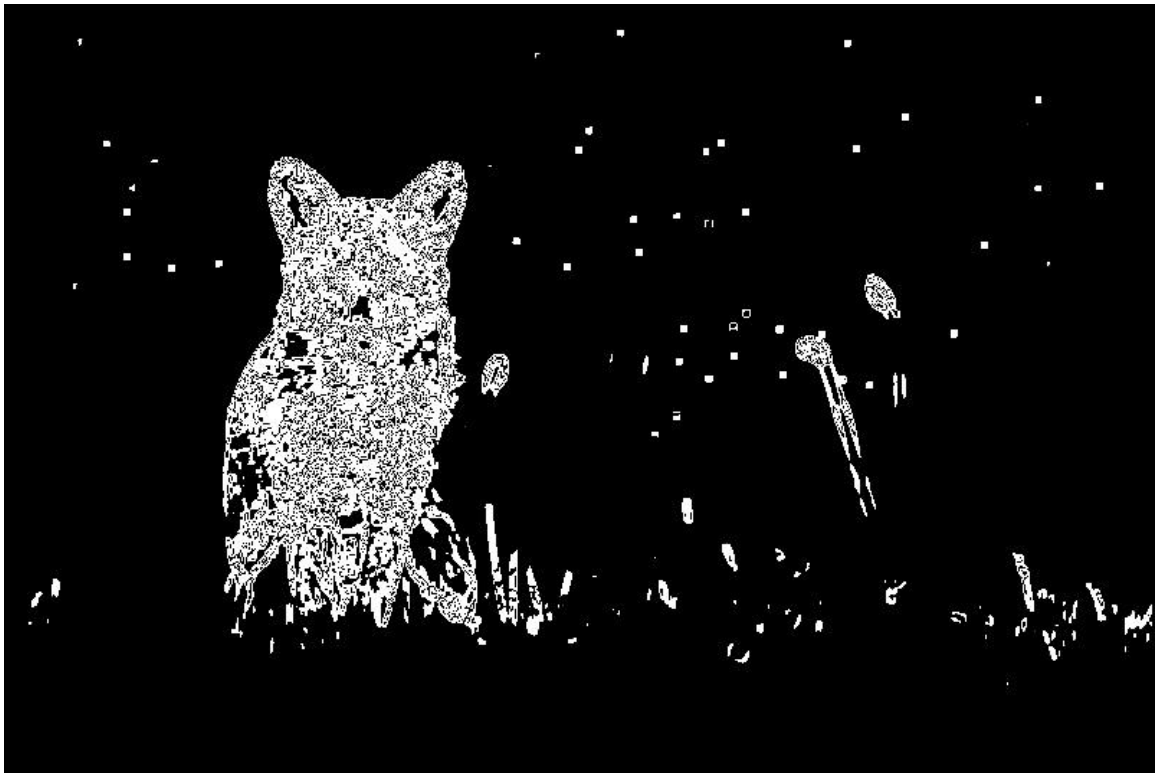

Fig 13: Foreground generated by the window size 5 × 5 at 1st iteration with threshold of 81.

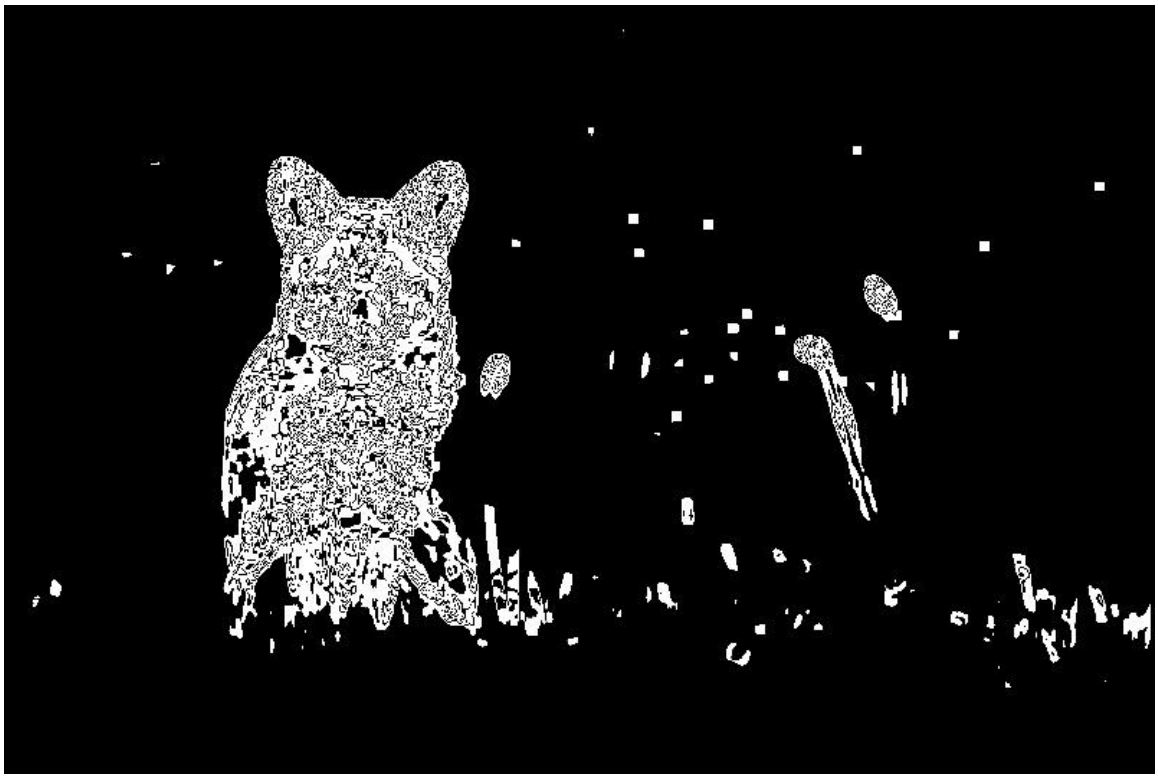

Fig 14: Foreground generated by the window size 7 × 7 at 1st iteration with threshold of 84.

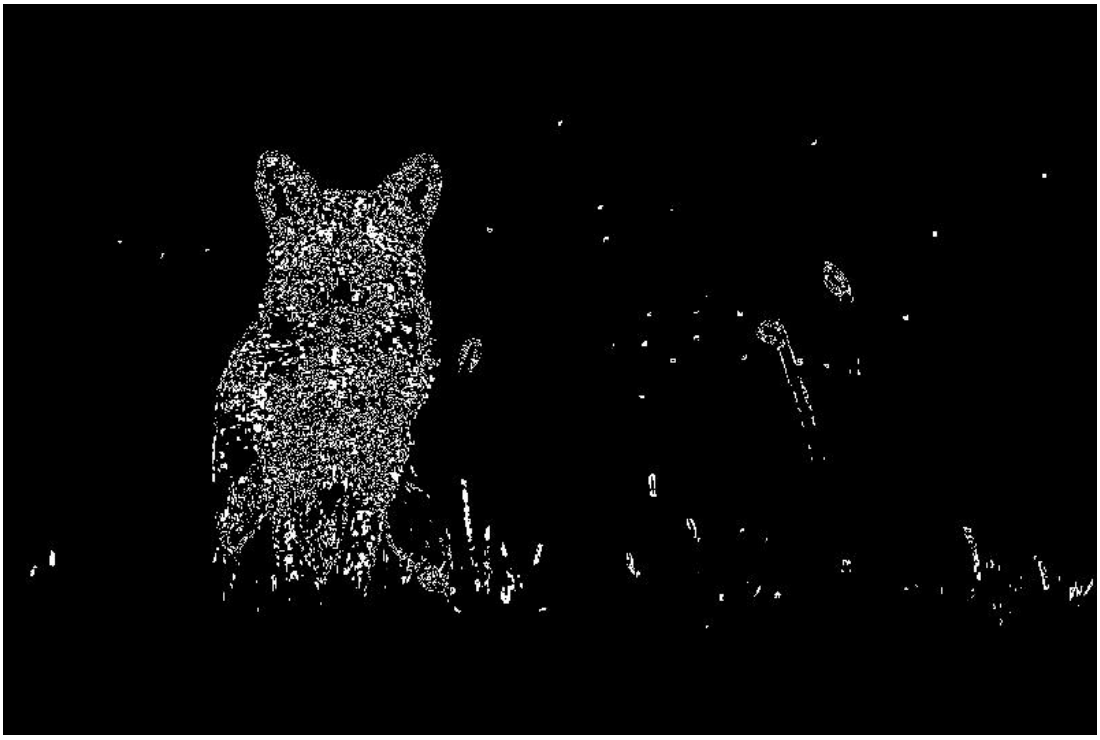

Fig 15: Foreground generated by using masks with different window sizes (at respective iterations of 1, 1 and 1) with 'AND' operation between channel wise mask.

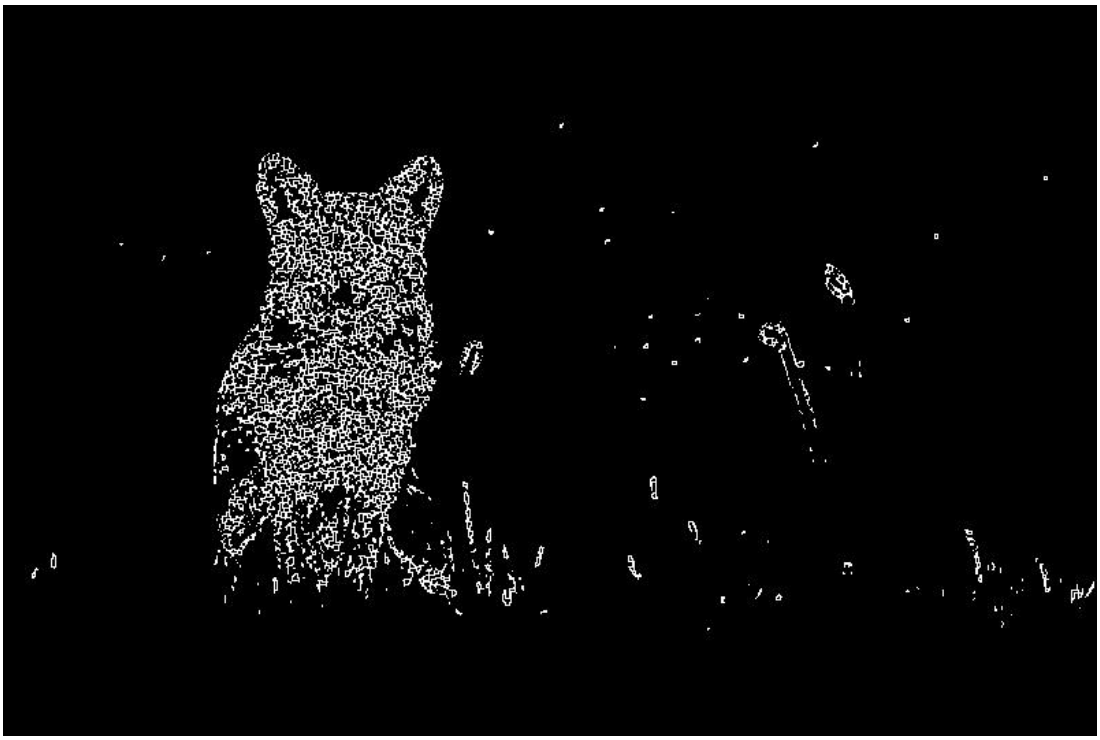

Fig 16: Contours generated by using the mask in Fig 15 with morphological operations for noise removal.

3. Observation of different methods:

1. Large scale results in reduced num of corners.

Source Code:

3.1.Function calls for Otsu and Texture segmentation:

```
'''----------------Main Code for Task 1.1-------------'''
im1=cv2.imread('cat.jpg')
iters_channel_1 = [1,1,2]
Segmented_1 = Otsu_RGB(im1,'cat',iters_channel_1)
cv2.imwrite('cat' +  '_segmented.jpg',Segmented_1)
#Perform morphological operations and get contours
Segmented_1 = erode_mask(Segmented_1,2)
Segmented_1 = dilate_mask(Segmented_1,2)
Contour_Seg_1 = Get_Contours(Segmented_1)
cv2.imwrite('cat' +  '_Contours.jpg',Contour_Seg_1)
print('-------------------------------')

im2=cv2.imread('pigeon.jpg')
iters_channel_2 = [1,1,2]
Segmented_2 = Otsu_RGB(im2,'pigeon',iters_channel_2)
cv2.imwrite('pigeon' +  '_segmented.jpg',Segmented_2)
#Perform morphological operations and get contours
Segmented_2 = erode_mask(Segmented_2,2)
Segmented_2 = dilate_mask(Segmented_2,2)
Contour_Seg_2 = Get_Contours(Segmented_2)
cv2.imwrite('pigeon' +  '_Contours.jpg',Contour_Seg_2)
print('-------------------------------')

im3=cv2.imread('Red_Fox_.jpg')
iters_channel_3 = [1,1,2]
Segmented_3 = Otsu_RGB(im3,'Red_Fox_',iters_channel_3)
cv2.imwrite('Red_Fox_' +  '_segmented.jpg',Segmented_3)
#Perform morphological operations and get contours
Segmented_3 = erode_mask(Segmented_3,4)
Segmented_3 = dilate_mask(Segmented_3,2)
```

```
Contour_Seg_3 = Get_Contours(Segmented_3)
cv2.imwrite('pigeon' +  '_Contours.jpg',Contour_Seg_3)
print('------------------------------')

'''------------Main Code for Task 1.2---------------'''
image_gray1 = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
window_size = [3,5,7]
text_img_1 = Texture_Image(image_gray1,window_size)
Segmented_Text_1 = Otsu_RGB(text_img_1,'cat_Texture',iters_channel_1)
cv2.imwrite('cat_Texture' +  '_segmented.jpg',Segmented_Text_1)
#Perform morphological operations and get contours
Segmented_Text_1 = dilate_mask(Segmented_Text_1,5)
Segmented_Text_1 = erode_mask(Segmented_Text_1,2)
Contour_Tex_1 = Get_Contours(Segmented_Text_1)
cv2.imwrite('cat_Texture' +  '_Contours.jpg',Contour_Tex_1)
print('------------------------------')

image_gray_2 = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)
text_img_2 = Texture_Image(image_gray_2,window_size)
Segmented_Text_2 = Otsu_RGB(text_img_2,'pigeon_Texture',[1,1,1])
cv2.imwrite('pigeon_Texture' +  '_segmented.jpg',Segmented_Text_2)
#Perform morphological operations and get contours
Segmented_Text_2 = dilate_mask(Segmented_Text_2,2)
Segmented_Text_2 = erode_mask(Segmented_Text_2,2)
Contour_Tex_2 = Get_Contours(Segmented_Text_2)
cv2.imwrite('pigeon_Texture' +  '_Contours.jpg',Contour_Tex_2)
print('------------------------------')

image_gray_3 = cv2.cvtColor(im3, cv2.COLOR_BGR2GRAY)
text_img_3 = Texture_Image(image_gray_3,window_size)
Segmented_Text_3 = Otsu_RGB(text_img_3,'Red_Fox_Texture',[1,1,1])
cv2.imwrite('Red_Fox' +  '_segmented.jpg',Segmented_Text_3)
#Perform morphological operations and get contours
Segmented_Text_3 = dilate_mask(Segmented_Text_3,2)
Segmented_Text_3 = erode_mask(Segmented_Text_3,2)
Contour_Tex_3 = Get_Contours(Segmented_Text_3)
cv2.imwrite('Red_Fox' +  '_Contours.jpg',Contour_Tex_3)
print('------------------------------')
```

A. Code for Otsu thresholding:

```
def Otsu_GrayScale(Gray_image, channel_mask=None):
    '''
    Function for obtaining a segmentation mask using grayscale image
    Inputs:
        Gray_image: A single channel grayscale image
        channel_mask: An existing or no mask for itetratively refining the segmentation results
```

```
Output: A signle channel Segmentation mask
'''
#Initialize the parameters
distribution = np.zeros((256,1))
max_sigma_b_squared =-1;
Thresholded_img = np.copy(Gray_image)
#probabilities and mean of levels lower than gray value
omega0 = 0
mu = 0

#For the 1st iteration mask is none, so use original image parameters
if channel_mask is None:
   histogram, _ = np.histogram(Gray_image, bins=256, range=(0, 255))
   mu_total = np.mean(Gray_image)
else:
   '''
   This is 2nd to onwards iterations. Perform elementwise multiplication
   to get rid of background using mask from previous iteration and compute
   mean and hsitogram of only non-zero entries of image
   '''
   masked_img=np.multiply(Gray_image,(channel_mask/255))
   #Remove the zeros from histogram as they are coming from existing background
   histogram, _ = np.histogram(masked_img, bins=256, range=(0.5, 255.5))
   #Set zro entries to nan
   masked_img[masked_img==0]=np.nan
   mu_total=np.nanmean(masked_img)

#Distribution from histogram
distribution = histogram / np.sum(histogram)

for k in range(256):
   omega0 = omega0 + distribution[k]
   omega1 = 1 - omega0
   mu = mu + k*distribution[k]
   #Avoid dividing by zero warning
   if omega0 == 0 or omega1==0:
      continue
   #Between class variance
   sigma_b_squared = ((mu_total * omega0 - mu)**2)/(omega0*omega1)
   #Find the new threshold level based on the between class variance
   if sigma_b_squared > max_sigma_b_squared:
      max_sigma_b_squared = sigma_b_squared
      otsu_threshold = k
print("Best Otsu Threshold found:",otsu_threshold)
#Check if threshold is greater than zero else generate an empty image
if otsu_threshold > 0 :
```

```
        Thresholded_img[Thresholded_img>otsu_threshold]=255
        Thresholded_img[Thresholded_img<otsu_threshold]=0
    else:
        Thresholded_img=np.zeros_like(Gray_image)

    return Thresholded_img

def Otsu_RGB(Color_img, savename, itertions = [1,1,1]):
    '''
    Function for obtaining a composite segmentation mask using RGB image
    Inputs:
        Color_img: A three channel RGB image
        savename: A string for saving the channel wise segmentation mask
        iterations: A list indicating the number of iterations for every channel
    Output: A composite Segmentation mask
    '''
    # Get the shape of image
    h,w,channels=Color_img.shape
    # Array for saving the masks for all channels
    masks=np.zeros_like(Color_img)
    #Composite RGB mask
    RGB_mask = np.zeros((h,w),np.uint8)

    # Pass through all image channels and extract the binary mask
    for channel in range(channels):
        '''
        Flag for the 1st iteration, in the next iterations the mask from previous
        iteration will be used
        '''
        channel_wise_mask = None
        # Run the Otsu algorithm iteratively
        for n in range(itertions[channel]):
            img_gray = Color_img[:,:,channel]
            channel_wise_mask = Otsu_GrayScale(img_gray,channel_wise_mask)
            #Save the channelwise mask
            savefilename=savename+ '_Ch_' + str(channel) + '_iter_' +str(n+1) + '.jpg'
            cv2.imwrite(savefilename,channel_wise_mask)
        #Arrange the masks from different channels in an array
        masks[:,:,channel] = channel_wise_mask
        #Perform And to obtain the final RGB mask
        RGB_mask = masks[:,:,0] & masks[:,:,1] & masks[:,:,2]
        #out_img=np.bitwise_and(RGB_mask, mask_ch)
    return RGB_mask
```

B. Function for Texture based segmentation:

```
def Texture_Image(Gray_image, kernel_sizes):
```

```
    # Get the shape of image
    h,w=Gray_image.shape

    # Generate an empty variance matrix with masks from different windows being
concatenated as 3rd channel
    variances = np.zeros((h,w,len(kernel_sizes)),np.uint8)
    #Loop through all the windows given by a list
    for k, ksize in enumerate(kernel_sizes):
        #Get the slidding window and compute its texture statistics
        kernel_half = np.int(ksize/2)
        #TODO: Change the loops to vectorized version. Time consuming
        for y in range(kernel_half, h - kernel_half):
            for x in range(kernel_half, w- kernel_half):
                #Window = Gray_image[y-kernel_half : y + kernel_half+1, x-kernel_half : x +
kernel_half+1]
                #vari = np.var(Window)
                variances[y,x,k] = Get_window(Gray_image,kernel_half,x,y)
    return variances
def Get_window(image,kernel_size,x,y):
    '''
    Function for finding thevariance inside a kernel.
    Requires the image, kernelsize and current image coordinates
    to define the current region occupied by kernel
    '''
    # Current kernel centered at x and y
    Window = image[y-kernel_size : y + kernel_size+1, x-kernel_size : x + kernel_size+1]
    vari = np.var(Window)
    return np.int(vari)
```

C. Function for Contours extraction:

```
def Get_Contours(masked_image):
    # Get the shape of image
    h,w=masked_image.shape
    #Generate an empty image to hold the contours
    contour_img = np.zeros(masked_image.shape,dtype=np.uint8)
    for i in range(1,h-1):
        for j in range(1,w-1):
            if masked_image[i,j]==255:  #Check if we are in foreground region
                #Check neighboorhood
                window=masked_image[i-1:i+2,j-1:j+2]
                #Check if all are foreground pixels or not
                if np.sum(window) < 255*9:
                    contour_img[i,j]=255

    return contour_img
```

```
def Get_window(image,kernel_size,x,y):
    '''
    Function for finding thevariance inside a kernel.
    Requires the image, kernelsize and current image coordinates
    to define the current region occupied by kernel
    '''
    # Current kernel centered at x and y
    Window = image[y-kernel_size : y + kernel_size+1, x-kernel_size : x + kernel_size+1]
    vari = np.var(Window)
    return np.int(vari)

def erode_mask(mask,errode_size):
    struct = np.ones((errode_size,errode_size),np.uint8)
    mask = cv2.erode(mask,struct)
    return mask

def dilate_mask(mask,dilate_size):
    struct = np.ones((dilate_size,dilate_size),np.uint8)
    mask = cv2.dilate(mask,struct)
    return mask
```