Homework 6

Theory Questions

- **Otsu algorithm:** It is simple and fast as it doesn't need to do much computations, just the probabilities and the means which can be calculated from the histogram which are very fast operations. It also allows one to automate the thresholding, as you select the threshold that maximizes the between-class variance. One of the downsides to this is that it is restricted to only binary segmentation, only background and foreground. Another downside is that it is highly sensitive to noise, as it can significantly change the histogram.
- **Watershed algorithm:** This algorithm uses a topological surface using the gradients, so it uses some spatial relationship between pixels to find boundaries. Because it uses the gradients and with how flooding works, it can generate very fine detailed segmentation maps and boundaries. A downside to this algorithm is that the flooding model is resource intensive, several times more than Otsu algorithm. Another issue with the watershed is that it needs preprocessing, and needs it to be precise too: the calculations of the gradients need to be precise for the flooding to work well. Another reason for the preprocessing to be well done is that it is highly sensitive to noise, since image gradients are, so noise needs to be filtered out from the image for watershed to be most effective.

Task 1

RGB Channels Otsu Algorithm

We begin by separating the channels using OpenCV. Then, for each one, we begin by getting the histogram of the image and, considering a threshold t that separates the classes, we calculate the class probabilities according to the Otsu algorithm equations:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i) \tag{1}$$

$$\omega_1(t) = \sum_{i=t}^{N-1} p(i) \tag{2}$$

Where in both cases, we compute these probabilities from the histograms (the number of incidences/the total number). Then we calculate the class means:

$$\mu_0(t) = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_1(t)} \tag{3}$$

$$\mu_1(t) = \frac{\sum_{i=t}^{N-1} ip(i)}{\omega_0(t)} \tag{4}$$

And now, using all these equations, we define the between-class variance, given by:

$$\sigma_b^2(t) = \omega_0(t)\omega_1(t)(\mu_0(t) - \mu_1(t))^2 \tag{5}$$

This variance we want to maximize, so we do so by iterating throught the values of the histogram to find which threshold t maximizes it. Once we find it, we create a binary mask, with 1 being given to the positions of the image where the gray values are higher than t, otherwise they become 0. We then make use of iterative Otsu algorithm, we take the foreground found by the first iteration of Otsu, and we mask the original image with this. We run Otsu on this again, but first we adjust the histogram, by substracting the previous iteration's number of background pixels to the t=0 bin of our histogram, and only consider the previous iteration's number of foreground pixels to get the probabilities. We do all this for all 3 channels, then using logical and (or in our case we use the Hadamard product between all 3, as they are 0s and 1s).

Texture-based Otsu Algorithm

We begin by reading the image as grayscale. For each of the window sizes given we do the following steps (for window size N):

1. Pad the image by N//2.
2. Create a window of size N by N around the pixel.
3. Substract the mean of the window and calculate the variance of this window.
4. Save this variance for this position.
5. Repeat for all the pixels in the image.
6. Finally, normalize the the variance matrix for it to be in the range [0,255] (so we can apply Otsu on it).

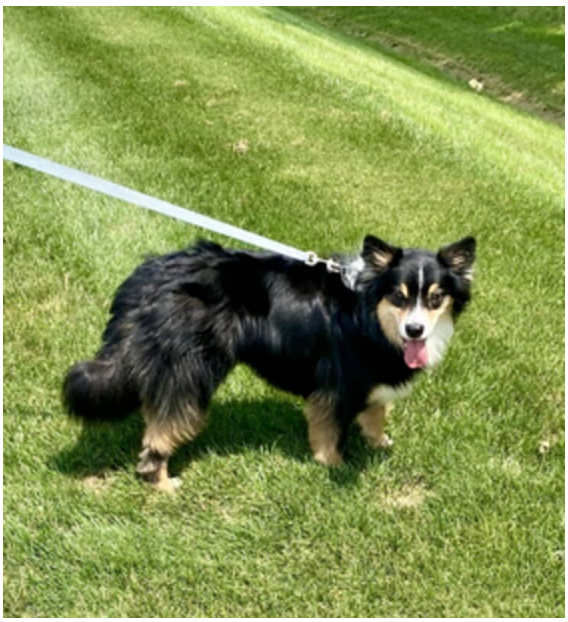
This gives us the texture map generated by the variances of all the windows in the image. We apply Otsu algorithm, that is described in the previous section, on these texture maps. We also make use of iterative Otsu on these. Finally, we combine the masks with a Hadamard product.

Contour Extraction

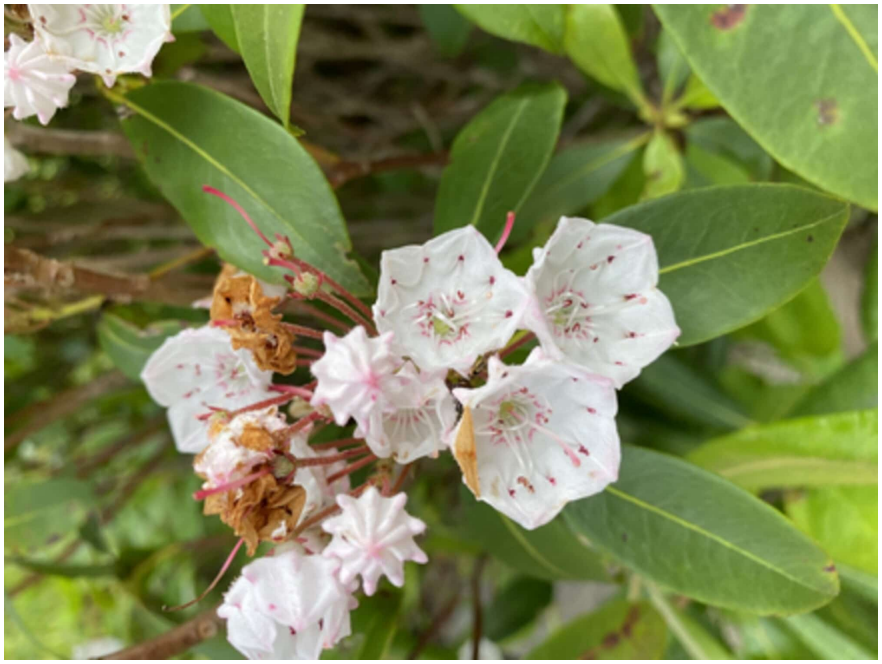
For contours, we only care about the neighborhoods that contain part of the background, so the neighborhoods that have a 0 in them. For this part of the task we use a 3×3 neighborhood. To find the contours on the segmented binary image, we iterate through the entire image, and check if the value we are on is a 1, this is because we also only care about the foreground of the image. If the value we are on is part of the foreground, we check if there is a 0 within the neighborhood, if there is that means that this specific position is part of the boundary; so we mark this position as part of the contour.

Results

Task 1



(a) Dog image



(b) Flower image

Figure 1: Input images for task 1

Task 1: RGB Otsu Algorithm and Contours

The dog in the image is our foreground. The main issue arises when the color of the dog's fur is mostly black, so it will have lower grey values than the other parts of the image. The way Otsu algorithm works is that the background will have lower gray values, so to fix this we swap the background and foreground around, by inverting the 0s and 1s in the binary mask that results from Otsu. For the dog, we invert the binary mask resulting from applying Otsu algorithm to the 3 different channels. We make use of iterative Otsu: 2 iterations for the red channel, and just 1 for both green and blue. The results can be seen in Fig. 11. We can see how some parts of the fur on the body of the dog were not selected correctly as the foreground, this is due to the fact that those parts of the fur are brighter due to the lighting. Similarly, some parts of the face of the dog were also not selected as the foreground, this one may be due to the fact that those parts of the dog's face are not the same color as the fur on its back. These are the parts where it failed, and there is also some noise from the shadows from the grass, as can be seen a little bit further down from the dog. The contours extracted, in Fig. 3 from this example are also, like the segmentation, not very accurate. More specifically, in the parts fur where the segmentation failed, there is a lot of noise, we also notice that the shadow of the dog is also very noticeable. Also, the face has not been contoured correctly, which is to be expected since the segmentation failed there. All in all, it is not too bad considering the simplicity of the algorithm.

(a) Red channel after Otsu algorithm

(b) Green channel after Otsu algorithm

(c) Blue channel after Otsu algorithm

(d) Final binary segmentation

Figure 2: Results of RGB Otsu algorithm on the image of the dog, background and foreground have been inverted, iterations for Iterative Otsu (RGB): 2, 1, 1.

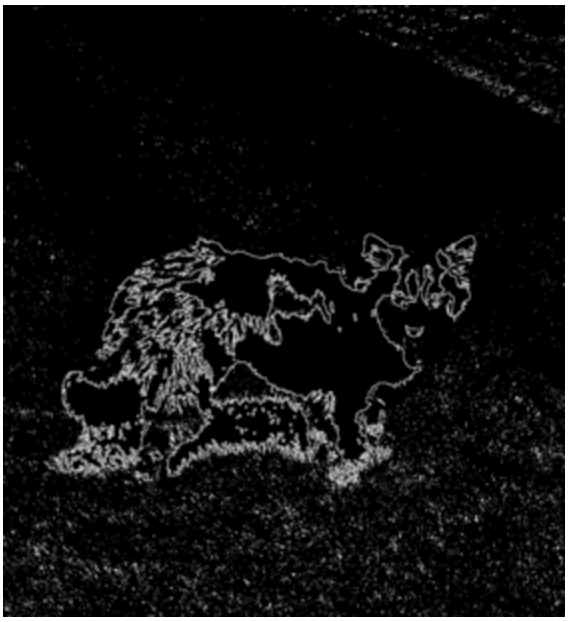
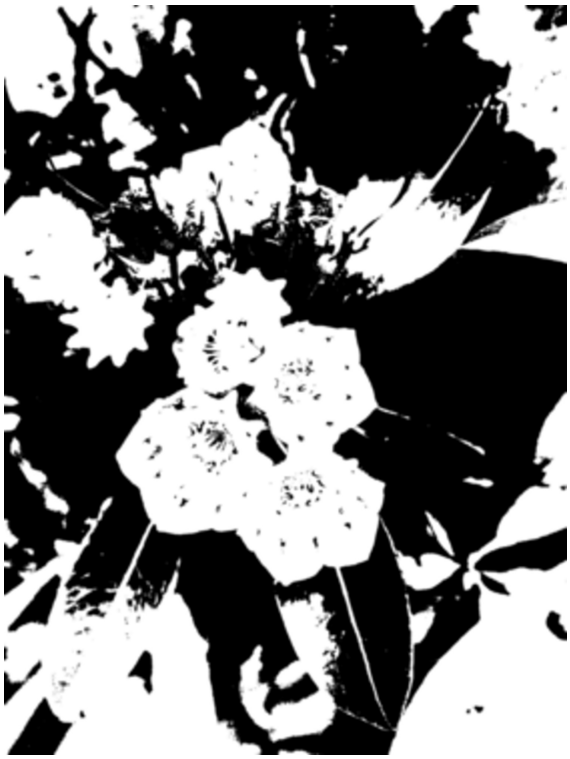


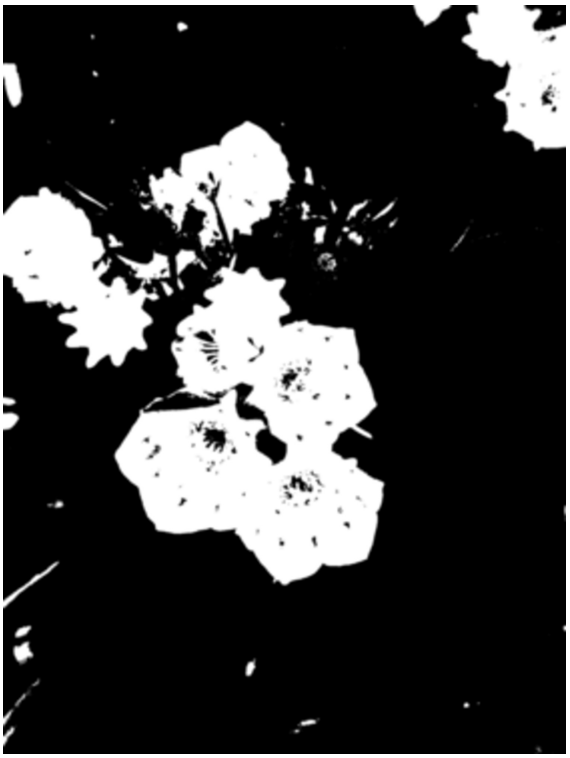
Figure 3: Contours from the final RGB Otsu segmentation

In the case of the flower image, RGB Otsu worked well. The iterations needed for the iterative Otsu was 1 for all RGB channels, meaning that we only needed to use Otsu once for all 3 channels. There was also no need to invert the image since the foreground has higher gray values than the background. This case worked well due to the fact that the flowers were almost all white, whereas the rest of the background was either green only or brown. So in this case it worked well due to that fact, and as expected, the contours also look much better compared to the dog image. In Fig. 5, we can still see a bit of noise, but most of the contours match the actual edges of the flowers of the original image, seen in Fig. 1b.

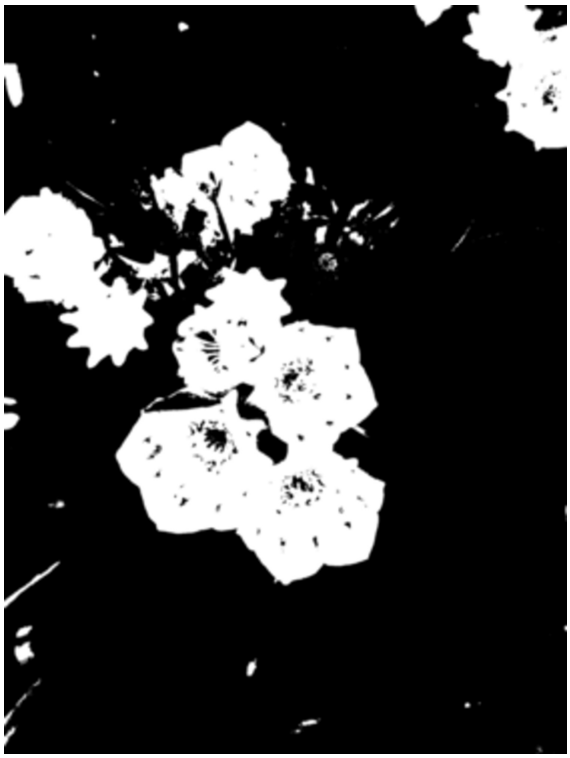
(a) Red channel after Otsu algorithm



(b) Green channel after Otsu algorithm



(c) Blue channel after Otsu algorithm



(d) Final binary segmentation

Figure 4: Results of RGB Otsu algorithm on the image of the flower, iterations for Iterative Otsu (RGB): 1, 1, 1.

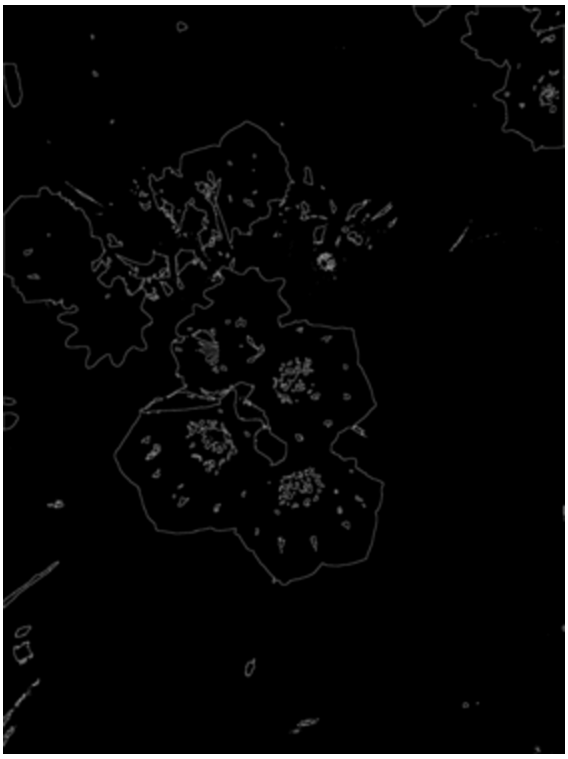
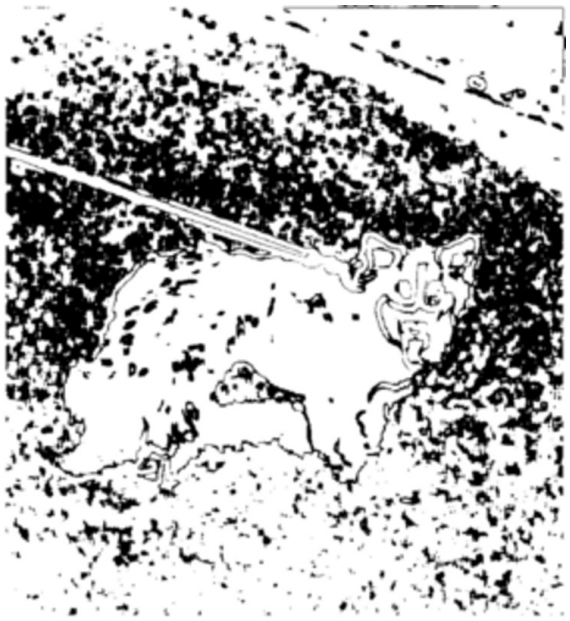


Figure 5: Contours from the final RGB Otsu segmentation for the flower

Task 1: Texture-based Otsu Algorithm

We consider window sizes of 9, 11 and 13 and perform 2, 1, 1 iterations respectively. In Fig. 6 that we have not correctly gotten most of the fur of the dog segmented, as well as the face, but there is far too much noise coming from the grass, and in the upper portion of the image the grass has been identified as being part of the foreground. The dog's shadow on the grass has also been identified as part of the foreground. Compared to the RGB Otsu segmentation, we now see the leash of the dog. When we look at the contour in Fig. 7, however, there is a lot more noise, it is even hard to tell where exactly is the dog at first sight. Overall, the RGB Otsu seems to provide far less noise, and far less incorrect foreground pixels compared to the texture-based method, even at the cost of losing part of the dog's face and a part of the fur.



(a) N=9 texture map after Otsu algorithm

(b) N=11 texture map after Otsu algorithm

(c) N=13 texture map after Otsu algorithm

(d) Final binary segmentation

Figure 6: Results of texture-based Otsu algorithm on the image of the dog, foreground and background have been inverted, iterations for iterative otsu: 2, 1, 1

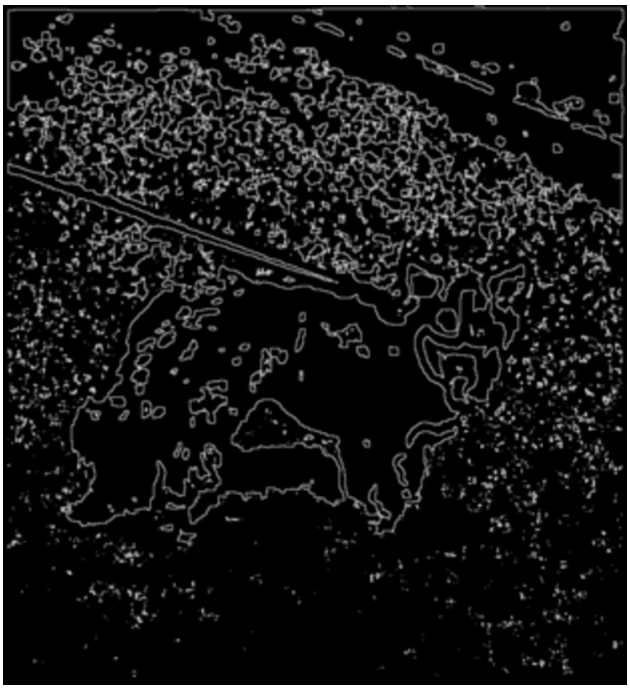
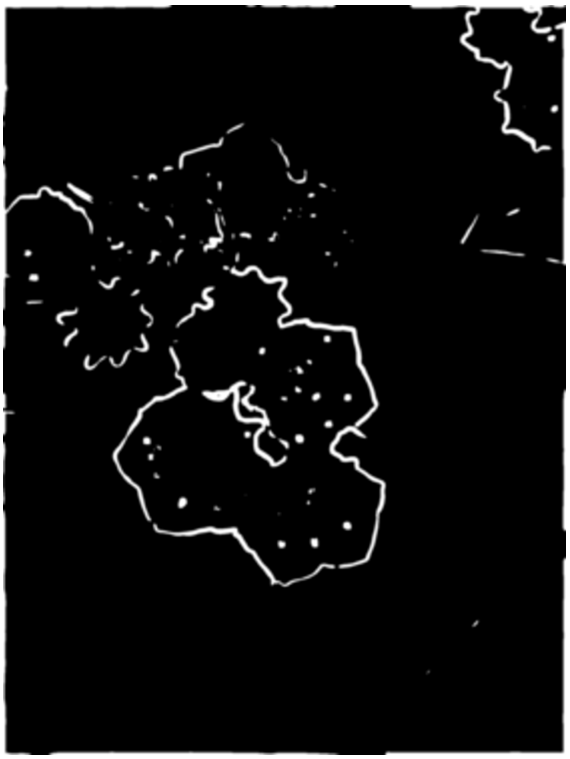
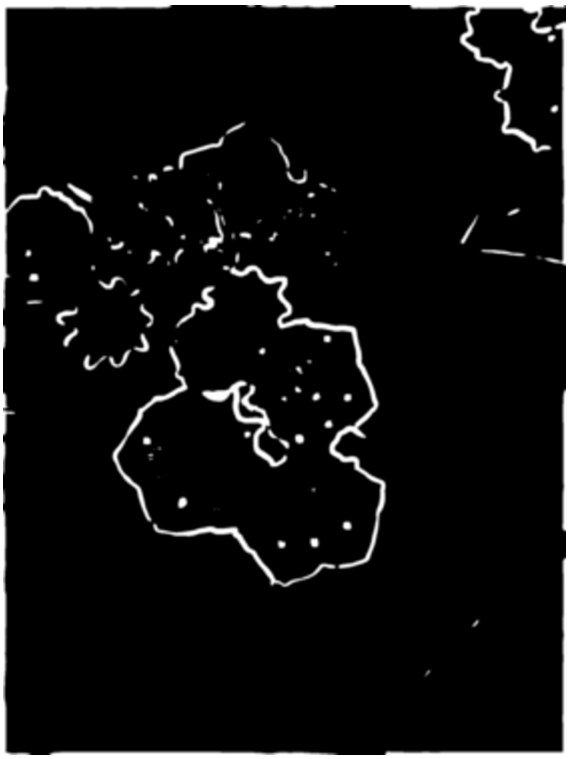


Figure 7: Contours for the texture-based Otsu algorithm on the image of the dog.

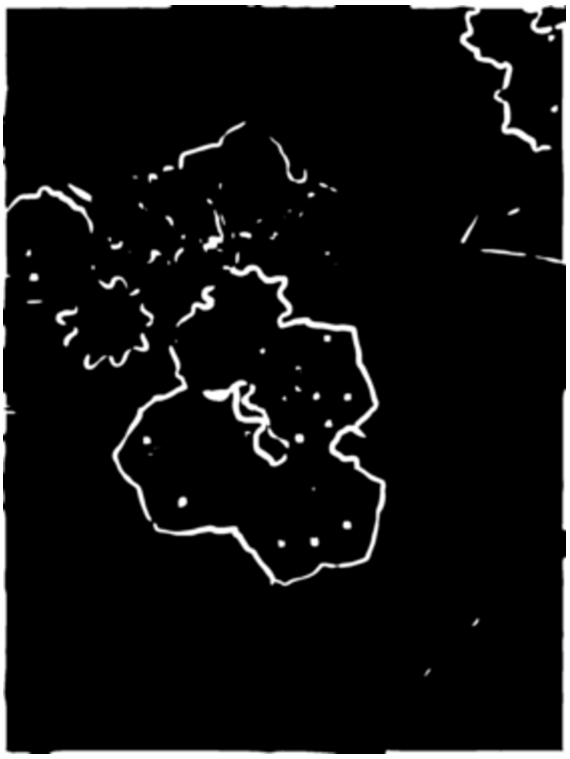
For the flower we considered window sizes of 17, 19 and 21, and iterations of 1, 1, 1 for iterative Otsu. Going for lower window sizes only provided worse results as it made the lines seen in Fig. 4 smaller, thus worse. This one did not generate good results, maybe if we were only considering the edges. One reason might be due to the fact that the variance in the points of interest was low over this whole region, so it was not picked up by Otsu. The contours look a bit better as we can observe in Fig. 9.



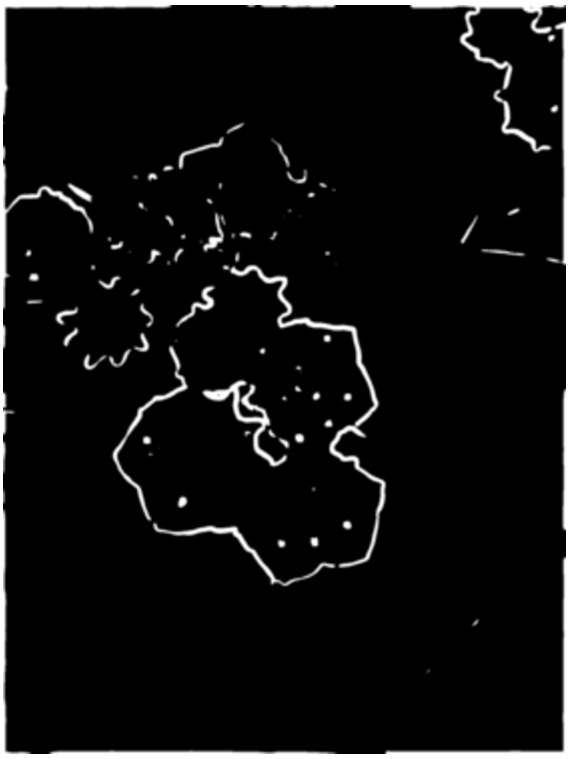
(a) N=17 texture map after Otsu algorithm



(b) N=19 texture map after Otsu algorithm



(c) N=21 texture map after Otsu algorithm



(d) Final binary segmentation

Figure 8: Results of texture-based Otsu algorithm on the image of the flower, iterations for iterative Otsu: 1, 1, 1

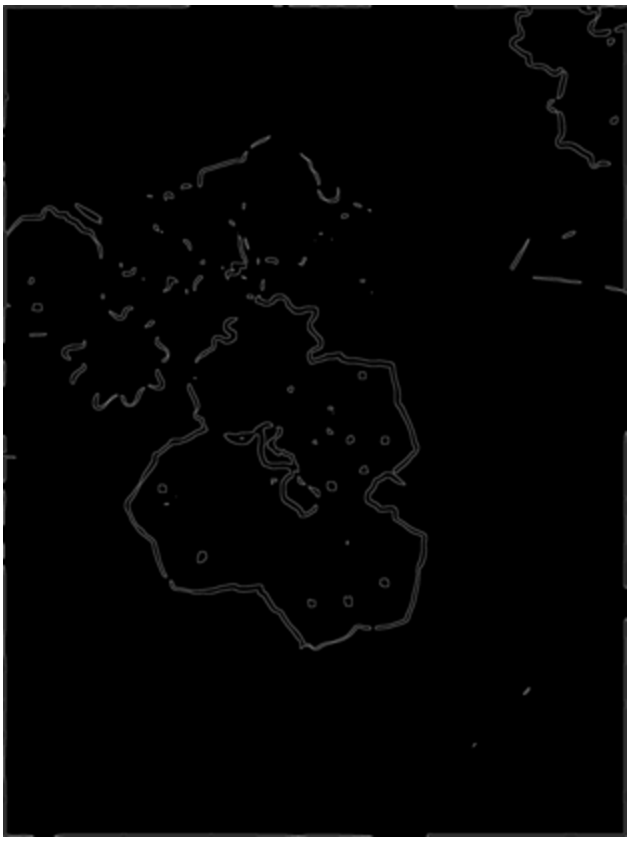
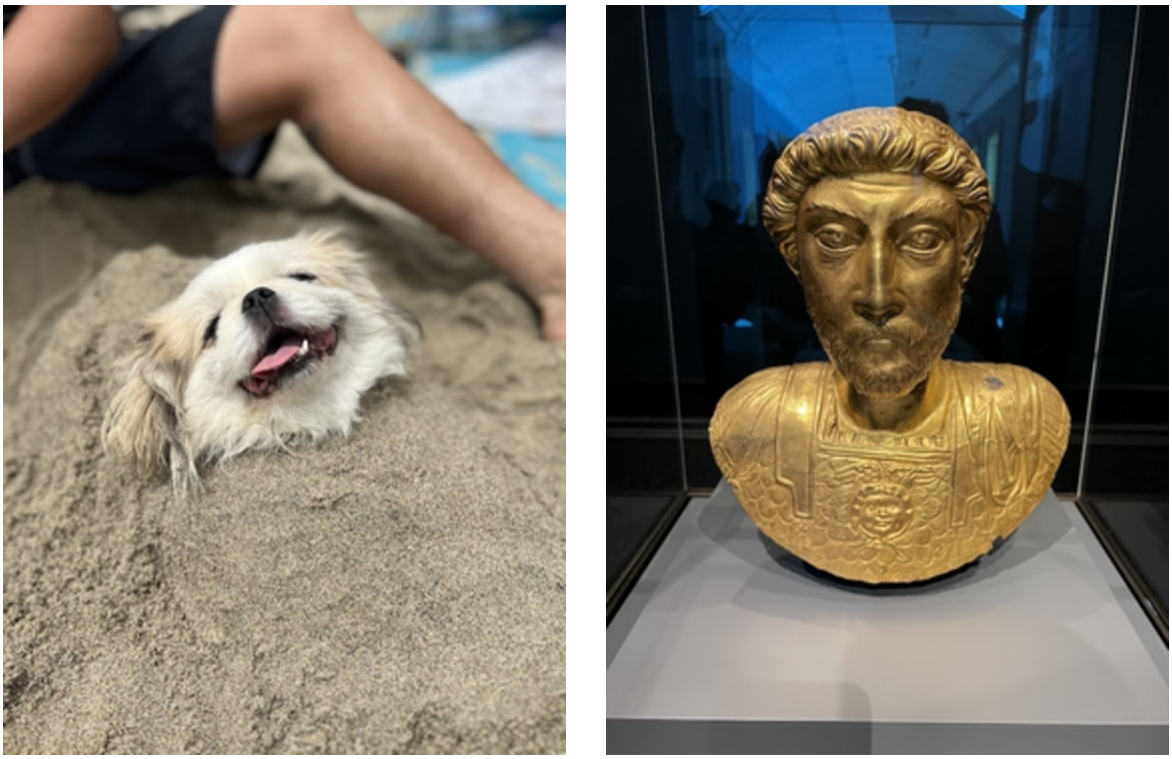


Figure 9: Contours for the texture-based Otsu algorithm on the image of the flowers

Task 2

The input images for Task 2 are shown in Fig. 10. The foreground are: the dog and the golden bust.



(a) Dog in the beach image (b) Golden bust image

Figure 10: Input images for task 2

Task 1: RGB Channels Otsu Algorithm

We used the iterations: 3, 3, 3 for the red, blue and green channel respectively. Even in a difficult image (the dog and the sand colors are not extremely different), with iterative Otsu we have good results. It has correctly segmented the face of the dog with a very low amount of noise present from the sand. We can see that the contour results are positive, aside from the noise.

(a) Red channel after Otsu algorithm

(b) Green channel after Otsu algorithm

(c) Blue channel after Otsu algorithm

(d) Final binary segmentation

Figure 11: Results of RGB Otsu algorithm on the image of the dog on the sand, iterations for Iterative Otsu (RGB): 3, 3, 3.

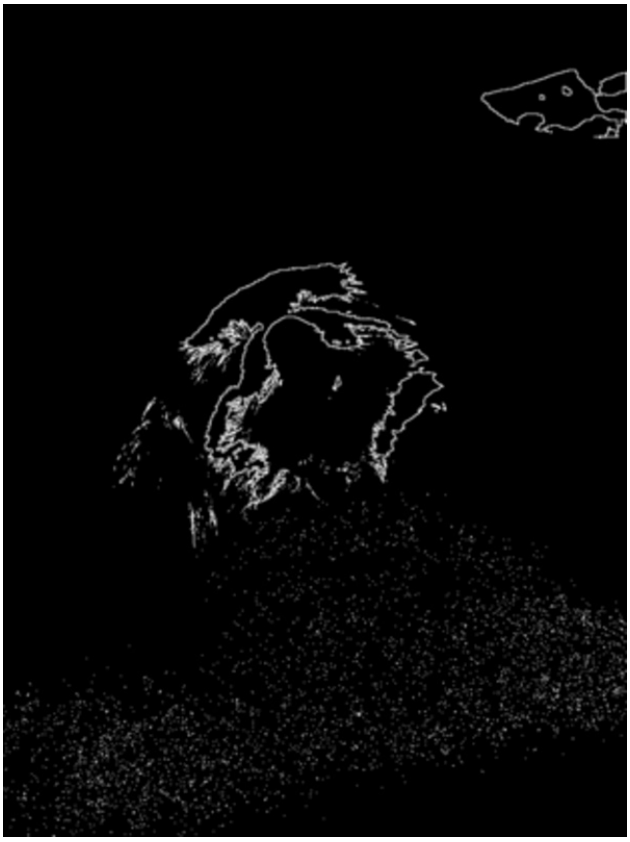


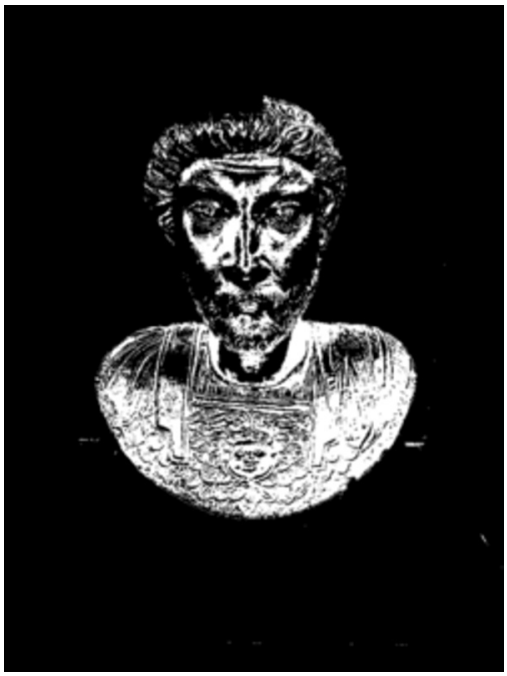
Figure 12: Contours for the RGB channels Otsu algorithm on the image of the dog on the sand

As for the golden bust image, we have similarly positive results, except that for this case we had to invert the Otsu results from the blue channel. The resulting binary mask is also not bad, as it only includes the bust, without the stand it is on, although it did fail in certain darker spots of the bust, which can be seen in the cheeks in Fig. 13d. The contour also looks quite good, capturing part of the details of even the armor.

(a) Red channel after Otsu algorithm

(b) Green channel after Otsu algorithm

(c) Blue channel after Otsu algorithm



(d) Final binary segmentation

Figure 13: Results of RGB Otsu algorithm on the image of the golden bust, background and foreground have been inverted but only for the blue channel, iterations for Iterative Otsu (RGB): 1, 1, 1.

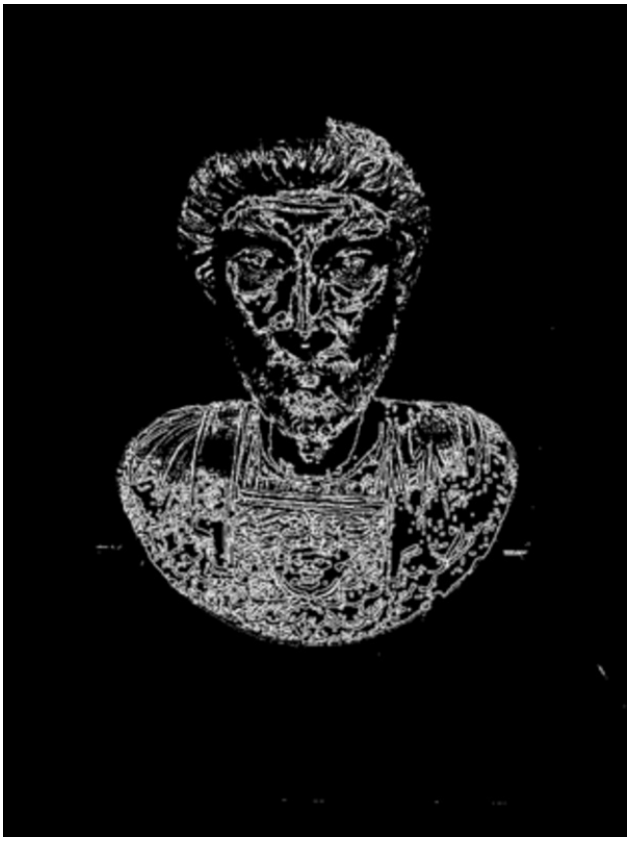
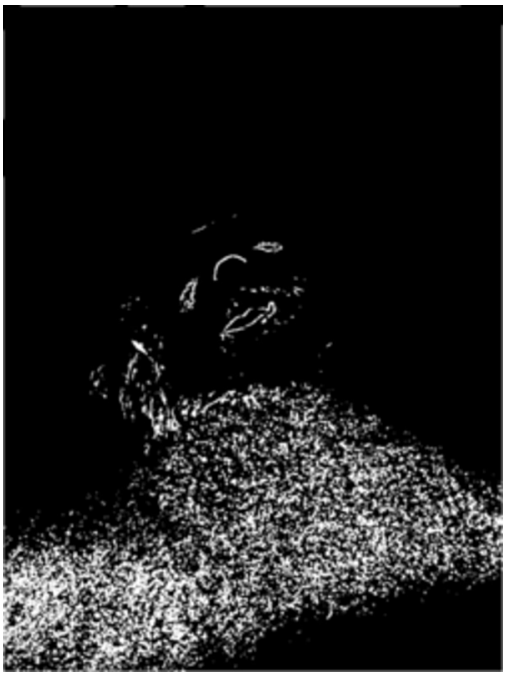


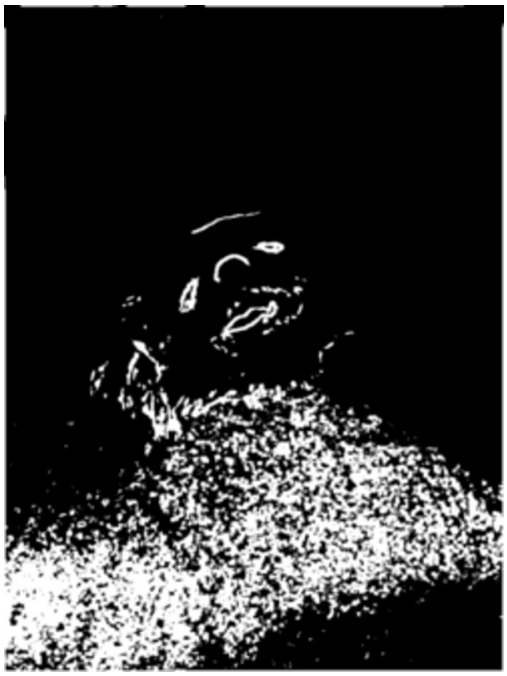
Figure 14: Contours for the RGB channels Otsu algorithm on the image of the golden bust.

Task 1: Texture-based Otsu Algorithm

The texture-based Otsu did not perform well on these two images. As we can see for the dog in Fig. 15, it only got the eyes, a part of the mouth of the dog and some edges, but mostly the segmentation consists of the sand. The contours, shown in Fig. 16 are not much better, they missed the entire face of the dog and only have the eyes and mouth.

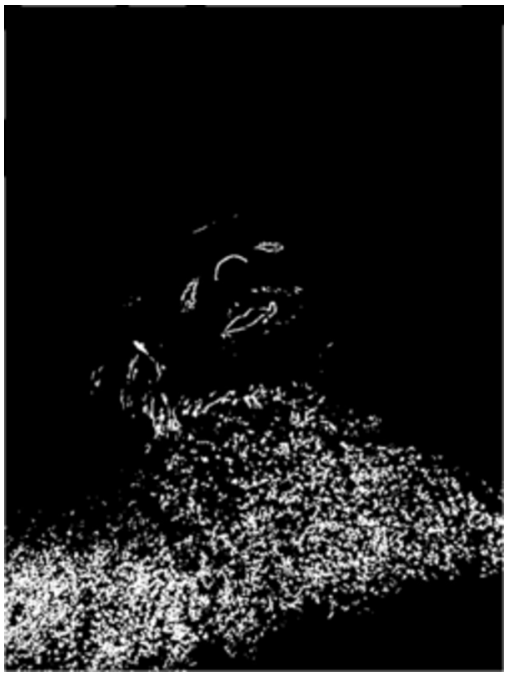


(a) N=3 texture map after Otsu algorithm



(b) N=5 texture map after Otsu algorithm

(c) N=7 texture map after Otsu algorithm



(d) Final binary segmentation

Figure 15: Results of texture-based Otsu algorithm on the image of the dog in the sand, iterations for iterative otsu: 1, 1, 1

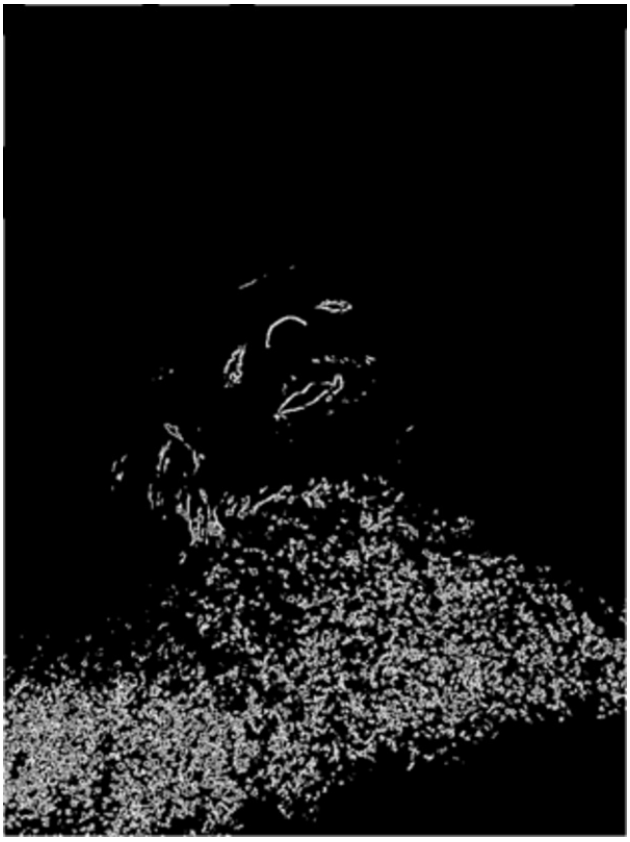
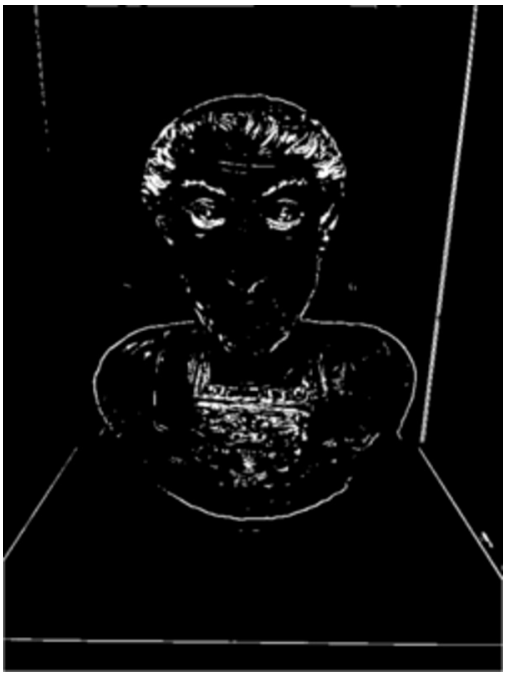


Figure 16: Contours for the texture-based Otsu algorithm on the image of the dog in the sand.

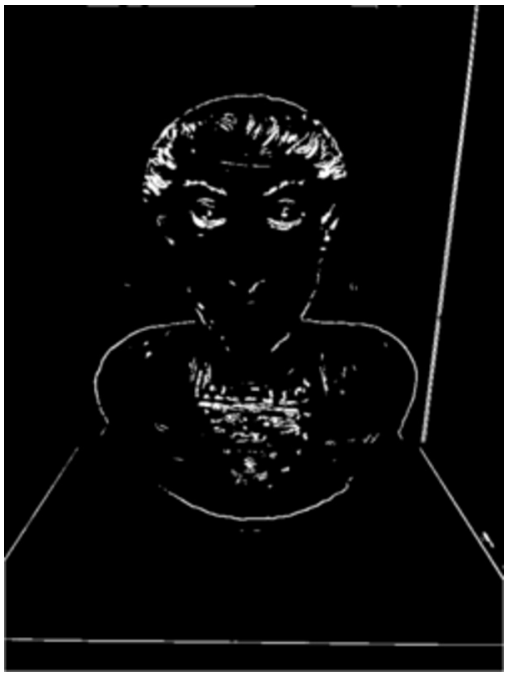
We have a similar story for the golden bust, the binary masks have lost much of the bust as well as the details compared to the RGB channel Otsu. It has not even gotten the full edge of the bust, as can be seen in Fig. 18.



(a) N=3 texture map after Otsu algorithm

(b) N=5 texture map after Otsu algorithm

(c) N=7 texture map after Otsu algorithm



(d) Final binary segmentation

Figure 17: Results of texture-based Otsu algorithm on the image of the golden bust, iterations for iterative otsu: 1, 1, 1

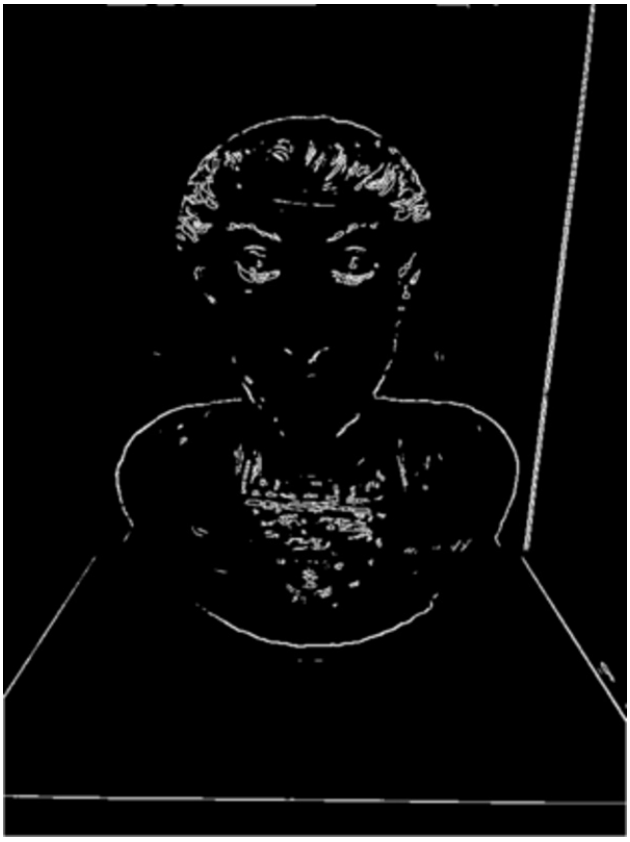


Figure 18: Contours for the texture-based Otsu algorithm on the image of the golden bust.

For both the dog and bust images, we used the same N, (3, 5, 7) since going with higher values did not improve our result, but instead made us lose more of the finer details. Also since most of the segmentation is not there, we only use 1 iteration for all masks. Inverting background and foreground here provided little value as all it did was just designate almost the entire image as the foreground, which is of no value for us.
All in all, the RGB channel Otsu algorithm has consistently given significantly better results in all 4 of the images tested.

Source code

```
import numpy as np
import cv2

def get_hist(image):
    # we just get the histogram with 256 bins (255 and 0 so 256 in total)
    hist = np.bincount(image.ravel(),minlength=256)
    hist = hist.astype(float)
    return hist

def get_im_color(filename):
```

```
    # this function loads the image and splits it into red green and blue cha
    # it is split in this specific way since opencv loads it as bgr as defaul
    img = cv2.imread(filename)
    blue, green, red = cv2.split(img)
    return red,green,blue

def otsu_algorithm(image, foreground=0, background=0, invert=False):
    # performs otsu algorithm, we take in the foreground and background for t
    # and to handle that we need to know the number of pixels that belong to
    # also see the iter_otsu function
    # we first create an empty image which will contain the binary mask
    im = np.zeros_like(image).astype(np.uint8)
    pix = np.arange(0,256,1)

    im_hist = get_hist(image)
    # since we only care about the foreground of the previous step, we substr
    # and our total will now be the pixels that belonged to the foreground in
    im_hist[0] -= background
    im_hist /= foreground
    # we define our maximums as 0 to find the new maximums easily, since they
    sigma_max = 0.0
    level = 0
    # we iterate through our the positions in our histogram
    for idx in range(1,len(pix)):
        # follow the otsu algorithm equations
        w_0 = im_hist[:idx].sum()
        w_1 = im_hist[idx:].sum()
        if w_0 == 0 or w_1 ==0:
            continue
        mu_0 = (im_hist[:idx] * pix[:idx]).sum()/w_0
        mu_1 = (im_hist[idx:] * pix[idx:]).sum()/w_1

        sigma = w_0*w_1*((mu_0 - mu_1)**2)
        if sigma > sigma_max:
            sigma_max = sigma
            level = idx
    im[image >= level] = 1
    im[image < level] = 0
    im = im.astype(bool)
    # this invert here is for the case when the foreground we want is black,
    if invert:
        return np.logical_not(im)
    return im

def iter_otsu(image, iters=0, invert=False):
```

```
    # this is the implementation of the iterative otsu algorithm
    img = image.copy()
    # for the first iteration, the "foreground" is the entire image and the n
    n_fg = (image.shape[0] * image.shape[1])
    n_bg = 0
    # we run it once and get the foreground
    fg = otsu_algorithm(image, n_fg, n_bg, invert=invert)
    # now if iters > 0 we go into this loop
    for iter in range(iters):
        # we need to know the number of pixels that are in the foreground now
        n_fg = np.count_nonzero(fg)
        # we get the background pixels easily since its just the total number
        n_bg = (image.shape[0] * image.shape[1]) - n_fg
        # we mask the image to only consider the foreground
        n_img = cv2.bitwise_and(img,img, mask=(fg*255).astype(np.uint8))
        # we use the masked image, the # of foreground and background pixels
        new_fg = otsu_algorithm(n_img, n_fg, n_bg, invert=invert)
        fg = new_fg
    return fg

def texture_maps(image, N=[3,5,7]):
    maps = []
    for i in range(len(N)):
        # we pad the image so the borders are not ignored, we pad it with the
        # especially since we then normalize all the variance values in the a
        # we pad it with n//2 for each value of n, we only use 3 values of n
        n = N[i]
        pad = n//2
        n_image = cv2.copyMakeBorder(image, pad,pad,pad,pad, cv2.BORDER_CONST
        sigma_m = np.zeros_like(image,dtype=float)
        # iterate through the image and calculate the variance and save it in
        for j in range(image.shape[0]):
            for i in range(image.shape[1]):
                window = (n_image[j:(j+n),i:(i+n)]).astype(float)
                window -= window.mean()
                var = window.var()
                sigma_m[j,i] = var
        # we normalize and append it to the map list, we set the type as np.u
        sigma_m = 255*(sigma_m - sigma_m.min())/(sigma_m.max() - sigma_m.min(
        maps.append(sigma_m.astype(np.uint8))
    return maps

def find_contour(binary_mask, window_size=3):
    # we use a 3x3 window size, this is never changed
    # again, we use padding of 3//2, so 1
```

```
    # we find the contour by checking which values of the array are part of t
    # then check if there is a 0 anywhere in the window, if there is, we know
    # of the border of the foreground, so we set this position to be 255 so w
    pad = int(window_size//2)
    contours = np.zeros_like(binary_mask)
    n_bmask = cv2.copyMakeBorder(binary_mask, pad,pad,pad,pad, cv2.BORDER_CON
    for j in range(binary_mask.shape[0]):
        for i in range(binary_mask.shape[1]):
            if binary_mask[j,i] == 255:
                window = n_bmask[j:(j+window_size), i:(i+window_size)]
                if 0 in window:
                    contours[j,i] = 255
    return contours.astype(np.uint8)
# for the next two we have invert as a list since we treat each channel diffe
# invert[0] is for the red channel/first texture map
# invert[1] is for the green channel/second texture map
# invert[2] is for the blue channel/last texture map
def otsu_rgb(filename, iters=[0,0,0], invert=[False,False,False]):
    # this function does all the necessary calls for the rgb otsu
    img_r, img_g, img_b = get_im_color(filename)
    seg_img_r = iter_otsu(img_r, iters=iters[0], invert=invert[0])
    seg_img_g = iter_otsu(img_g, iters=iters[1], invert=invert[1])
    seg_img_b = iter_otsu(img_b, iters=iters[2], invert=invert[2])
    seg_img = ((seg_img_r*seg_img_g*seg_img_b)*255).astype(np.uint8)
    contour = find_contour(seg_img)
    return seg_img, contour, (seg_img_r*255).astype(np.uint8), (seg_img_g*255

def otsu_texture(filename, texture_n=[1,2,3], iters=[0,0,0], invert=[False,Fa
    # this is just for the texture based otsu algorithm
    n_list = texture_n
    img_gray = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
    map1, map2, map3 = texture_maps(img_gray, N=n_list)
    seg_map1 = iter_otsu(map1, iters[0],invert=invert[0])
    seg_map2 = iter_otsu(map2, iters[1],invert=invert[1])
    seg_map3 = iter_otsu(map3, iters[2],invert=invert[2])
    seg_map = ((seg_map1*seg_map2*seg_map3)*255).astype(np.uint8)
    contour = find_contour(seg_map)
    return seg_map, contour, (seg_map1*255).astype(np.uint8), (seg_map2*255).

### TASK 1
# dog
## rgb otsu
dog = "pics/dog_small.jpg"
iterations = [1,0,0]
seg_dog, dog_cont, seg_r, seg_g, seg_b = otsu_rgb(dog, iters=iterations, inve
```

```
name = "pics/task_1_dog"
cv2.imwrite(name+".jpg",seg_dog)
cv2.imwrite(name+"_contours.jpg",dog_cont)
cv2.imwrite(name+"_red.jpg",seg_r)
cv2.imwrite(name+"_green.jpg",seg_g)
cv2.imwrite(name+"_blue.jpg",seg_b)
### texture otsu
iterations = [1,0,0]
ns = [9,11,13]
seg_dog, dog_cont, seg_n1, seg_n2, seg_n3 = otsu_texture(dog, texture_n=ns, i
name = "pics/task_1_texture_dog"
cv2.imwrite(name+".jpg",seg_dog)
cv2.imwrite(name+"_contours.jpg",dog_cont)
cv2.imwrite(name+"_n"+str(ns[0])+".jpg",seg_n1)
cv2.imwrite(name+"_n"+str(ns[1])+".jpg",seg_n2)
cv2.imwrite(name+"_n"+str(ns[2])+".jpg",seg_n3)

# flower
## rgb otsu
flower = "pics/flower_small.jpg"
iterations = [0,0,0]
seg_flower, flower_cont, seg_r, seg_g, seg_b = otsu_rgb(flower, iters=iterati
name = "pics/task_1_flower"
cv2.imwrite(name+".jpg",seg_flower)
cv2.imwrite(name+"_contours.jpg",flower_cont)
cv2.imwrite(name+"_red.jpg",seg_r)
cv2.imwrite(name+"_green.jpg",seg_g)
cv2.imwrite(name+"_blue.jpg",seg_b)

## texture otsu
iterations = [0,0,0]
ns = [17,19,21]
seg_flower, flower_cont, seg_n1, seg_n2, seg_n3 = otsu_texture(flower, textur
name = "pics/task_1_texture_flower"
cv2.imwrite(name+".jpg",seg_flower)
cv2.imwrite(name+"_contours.jpg",flower_cont)
cv2.imwrite(name+"_n"+str(ns[0])+".jpg",seg_n1)
cv2.imwrite(name+"_n"+str(ns[1])+".jpg",seg_n2)
cv2.imwrite(name+"_n"+str(ns[2])+".jpg",seg_n3)

### TASK TWO

# golden bust
## rgb otsu
bust = "pics/golden_bust.jpg"
```

```
iterations = [0,0,0]
seg_bust, bust_cont, seg_r, seg_g, seg_b = otsu_rgb(bust, iters=iterations, i
name = "pics/task_2_bust"
cv2.imwrite(name+".jpg",seg_bust)
cv2.imwrite(name+"_contours.jpg",bust_cont)
cv2.imwrite(name+"_red.jpg",seg_r)
cv2.imwrite(name+"_green.jpg",seg_g)
cv2.imwrite(name+"_blue.jpg",seg_b)
## texture otsu
iterations = [0,0,0]
ns = [3,5,7]
seg_bust, bust_cont, seg_n1, seg_n2, seg_n3 = otsu_texture(bust, texture_n=ns
name = "pics/task_2_texture_bust"
cv2.imwrite(name+".jpg",seg_bust)
cv2.imwrite(name+"_contours.jpg",bust_cont)
cv2.imwrite(name+"_n"+str(ns[0])+".jpg",seg_n1)
cv2.imwrite(name+"_n"+str(ns[1])+".jpg",seg_n2)
cv2.imwrite(name+"_n"+str(ns[2])+".jpg",seg_n3)

# dog in the sand
## rgb otsu
beach = "pics/dog_beach.jpg"
iterations = [2,2,2]
seg_beach, beach_cont, seg_r, seg_g, seg_b = otsu_rgb(beach, iters=iterations
name = "pics/task_2_dogbeach"
cv2.imwrite(name+".jpg",seg_beach)
cv2.imwrite(name+"_contours.jpg",beach_cont)
cv2.imwrite(name+"_red.jpg",seg_r)
cv2.imwrite(name+"_green.jpg",seg_g)
cv2.imwrite(name+"_blue.jpg",seg_b)
## texture otsu
iterations = [0,0,0]
ns = [3,5,7]
seg_beach, beach_cont, seg_n1, seg_n2, seg_n3 = otsu_texture(beach, texture_n
name = "pics/task_2_texture_dogbeach"
cv2.imwrite(name+".jpg",seg_beach)
cv2.imwrite(name+"_contours.jpg",beach_cont)
cv2.imwrite(name+"_n"+str(ns[0])+".jpg",seg_n1)
cv2.imwrite(name+"_n"+str(ns[1])+".jpg",seg_n2)
cv2.imwrite(name+"_n"+str(ns[2])+".jpg",seg_n3)
```