ECE 66100 Homework #6

by

Adrien Dubois (dubois6@purdue.edu)

October 18, 2024

Contents

1 Theory Questions **1**
- 1.1 Question 1: . . . 1
- 1.2 Question 2: . . . 2

2 Otsu's Algorithm: **2**
- 2.1 Implementation: . . . 3
  - 2.1.1 Otsu's Algorithm: . . . 3
  - 2.1.2 Calling coode: . . . 3

3 Texture Based Otsu's Algorithm: **4**
- 3.1 Explanation of the algorithm: . . . 4
- 3.2 Implementation: . . . 4

4 Contouring: **5**
- 4.1 Contouring Implementation: . . . 5
- 4.2 Closing Implementation: . . . 5

5 Results: **6**
- 5.1 Original Images: . . . 6
- 5.2 RGB Otsu's Results: . . . 6
  - 5.2.1 Histograms: . . . 6
  - 5.2.2 Iteration Masks: . . . 9
  - 5.2.3 BGR Final Masks: . . . 11
  - 5.2.4 Segmentation Result: . . . 12
  - 5.2.5 Contouring Results: . . . 13
  - 5.2.6 What worked and what did not work: . . . 15
- 5.3 Texture Based Otsu's Results: . . . 16
  - 5.3.1 Optimal hyperparamters: . . . 16
  - 5.3.2 Histograms: . . . 16
  - 5.3.3 Iteration Masks: . . . 16
  - 5.3.4 BGR Final Masks: . . . 18
  - 5.3.5 Final Contouring Result: . . . 20
  - 5.3.6 What worked and what did not work: . . . 20

6 Complete code printout: **21**

1 Theory Questions

1.1 Question 1:

What are the strengths and weaknesses of Otsu's algorithm? Otsu's algorithm has a few key strengths. First, it is quite a lightweigth algorithm that runs quickly even without any GPU resources as would be

required for Neural Network implementations. This makes it optimal for realtime applications, where the low latency is necessary for efficient runtimes. Additionally, it is an unsupervised method for segementation which means that is does not require any labeled training data.

One of the weakenesses of Otsu's algorithm is that it only works based off of the seperate RGB channels. This reduces the accuracy of the segmentation, and also makes it so that users needed to pre-determine which thresholded class they want to keep depending on whether or not the foreground object is bright (keep class 1 which is closer to white pixels), or the foreground object is dark (keep classs 0 which is closer to black pixels). Additionally, Ostu's algorithm is very sensitive to noise between the foreground and background objects. In this way, since it works on the brightness of the RGB channels, if the whole image is muddied/dark, Otsu's algorithm would likely fail to segment the foreground object. Lastly, finding a global threshold for the whole image could introduce performance losses

1.2 Question 2:

What are the strengths and weaknesses of the Watershed algorithm? Similar to Otsu's algorithm, one of the strengths of the Watershed algorithm is that it is an unsupervised method for segementation and therefore does not require a labeled dataset. Additionally, unlike Otsu's algorithm, the Watershed algorithm performs very well for complex objects, even when boundaries between objects overlap. It also makes no assumption on the pixel value distribution which makes it much more versatile than Otsu. Some of the weaknesses of the Watershed algorithm is that it is still much more sensitve to noise compared to Neural Network based approaches. Additionally, it tends to oversegment images with multiple segmented "fragments" in the image instead of just extracting the foreground. Lastly, the Watershed algorithm is much more computationally expensive compared to other methods while also having much lower performance than simple NN approaches.

2 Otsu's Algorithm:

Otsu's algorithm works by creating a histogram of all the pixel values in the image. It then finds a threshold value, that seperates the histogram into two different classes (foreground and background) by maximizing the between class variance at a certain threshold. The following equations are used to calculate the between class variance:

$$p(i) = \frac{n_i}{N}$$

where:

- n_i is the number of points with pixel value = i
- N is the total number of points in the image

We the calculate the probabilities of picking a point from each class:

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

$$\omega_1(t) = \sum i = t^N p(i)$$

where:

- t = the current test value for the treshold

We can then calculate the class means as follows:

$$\mu_0 = \frac{\sum_{i=0}^{t} i \times p(i)}{\omega_0}$$

$$\mu_1 = \frac{\sum_{i=t}^{N} i \times p(i)}{\omega_1}$$

Lastly, we calculate the between class variance:

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

After calculating the optimal threshold as the value of t that maximizes ω_B, we threshold the image as follows: For all (i,j), we fill in an image canvas using this rule:

- If pixel value as (i,j) $>$ threshold $\rightarrow$ 1
- If pixel value as (i,j) $\leq$ threshold $\rightarrow$ 0

This calculation is performed if we assume that the foreground pixel is a dark object. On the other hand, if the foreground object is brighter than the rest of the image, we need to threshold in the opposite direction by keeping class 1.

This algorithm was run seperately for each of the RGB channels of the image. Lastly, I ran this in an iterative manner, where I would re-run Otsu's algorithm on only the image pixels that remain after applying the segmentation mask on the original image. It is important to note that the re-runs should be run on the remaining pixels, and not on the thresholded image as that would increase the number of 0 valued pixels which would affect the histogram pixel distribution and therefore the between class variance calculation.

2.1 Implementation:

2.1.1 Otsu's Algorithm:

```
# RBG Segmentation
def get_Otsus_segmentation(img_channel, img_name, iter):
    num_bins = int(np.max(img_channel) - np.min(img_channel) + 1)
    hist, bin_edges = np.histogram(img_channel, bins=num_bins)

    # Get histogram probabilities
    p_i = hist / sum(hist)

    # List to save between class variances
    variances = np.zeros((len(bin_edges)))

    # Go through all threshold ts and find the one that maximizes the fisher
    discriminant
    # I only work with the pixels left from the previous iteration of Otsu's algorithm.
    # So, my loop for the t values can only get larger than the previous one
    for t in range(len(bin_edges)):
        # Probability of picking points from each class = sum(p(i))
        w0 = np.sum(p_i[:t])
        w1 = np.sum(p_i[t:])

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

        # Mean levels for each class = sum(i * p(i))
        i = np.arange(len(p_i))
        mu0 = np.sum(i[:t] * p_i[:t])/w0
        mu1 = np.sum(i[t:] * p_i[t:])/w1

        # Between class varianced
        omegaB = w0 * w1 * (mu1 - mu0)**2

        # Append fisher's discriminant to the variances list
        variances[t] = omegaB

    # Save a graph of the histogram with the variances at each t plotted on top
    plt.figure(figsize=(10, 6))
    plt.bar(bin_edges[:-1], hist, width=np.diff(bin_edges), edgecolor="black", align="
    edge", label="Pixel Value Histogram")
    plt.plot(variances, color='r', label="Between class variances at each t")
    plt.legend(loc='upper right')
    plt.savefig(histogram_paths + img_name + "_hist_" + str(iter) + ".jpg", format="jpg"
    )
    plt.close()

    # Return the t that maximized fisher's ratio
    return np.argmax(variances)
```

2.1.2 Calling coode:

```
# Otsu's Algorithm for RGB images:
# Images are in the following order: Climb, Dog, Flower, RVL
iters = [[3,3,4],
            [3,3,3],
            [1,1,1],
            [2,2,2]]
invert = [False, True, False, False]
for i, img_path in enumerate(img_paths):
    img_name = img_path.split("/")[-1][:-4]
    run_Otsus_algorithm_iterative(iters=iters[i], save_path=rgb_path, \
    invert=invert[i], img_path=img_path, img=None, img_name=img_name, \
    histogram_folder="RGB/")
```

3 Texture Based Otsu's Algorithm:

3.1 Explanation of the algorithm:

On the other hand, for the texture based implementation I run a windowing algorithm with a window size of N.

To do so, I first pad the image with 0s using a pad width of N. I can then use numpys' sliding window view method to return an array for each window of shape (2N+1, 2N+1).

Then, I substract the mean window pixel from all the window pixels and calculate the within window variance, normalized from a 0 to 255. I run this algorithm for three values of N, where lower values produced more fine-grain results compared larger values of N. By concatenating there texture maps together into the 3 channels of an image canvas, I can then run my Otsu's algorithm on the textured image data and produce much better contouring results for complex images. The only place where this did not work was for the dog image where the grass textures were taken as contours of foreground objects which produced a lot of noise in my output.

3.2 Implementation:

```
# Texture Segmentation
def get_texture_ostu(grey_img, N):
    # Create canvas for output image
    output_channel = np.zeros_like(grey_img)

    # Pad the image with extra 0s as necessary
    padded_img = np.pad(grey_img, pad_width=N, mode="constant", constant_values=0)

    # Create an N by N window of the nearby pixels with 0 padding
    # Instead of using a double for loop which is slow, I do this through numpy
    vectorized functions
    # This will return a numpy array of shape: (H, W, 2N + 1, 2N + 1) (list of windows)
    # We do 2N + 1 since we want N on each direction from the input pixel
    window_shape = (2*N + 1, 2*N + 1)
    windows = np.lib.stride_tricks.sliding_window_view(padded_img, window_shape)

    # Substract the window's mean value
    window_means = np.mean(windows, axis=(-2, -1))
    windows = windows - window_means[:,:, None, None]

    # Compute the variance within each window
    window_variances = np.var(windows, axis=(-2, -1))

    # Normalize the variance to 0-1 range and scale to 255 for BW image
    output_channel = (window_variances - window_variances.min()) / (window_variances.max
    () - window_variances.min()) * 255

    # Return a new array with the within pixel variance as the center pixel value
    # These values are normalized to 0-255
    return output_channel.astype(np.uint8)
```

4 Contouring:

While the Texture based Otsu algorithm directly creates contour maps, we need to do some post-processing on the RGB Ostu results to extract the contour information. To do so, I create 3 by 3 windows for all points of the image. I then check for two different masks:

1. Is the center pixel value equal to 1?
2. Are there any 0s in the window?

Performing a logical and computation on these two masks allows for an extremely fast implementation of a contouring algorithm to return border pixels.

4.1 Contouring Implementation:

```
def get_contours(img):
    window_shape = (3,3)
    windows = np.lib.stride_tricks.sliding_window_view(img, window_shape)

    # We only want to look at windows where the center pixel is 1:
    center_pixels = windows[:, :, 1, 1]
    center_pixel_mask = center_pixels == 1

    # Create a mask that will look in each window and check for a 0 pixel
    # If there is a 0, it will return a 1, otherwise return a 0
    zero_mask = np.any(windows == 0, axis=(2, 3)).astype(int)

    final_mask = np.logical_and(center_pixel_mask, zero_mask)

    # Convert to grey scale (0->255)
    return final_mask * 255
```

However, since the point clouds returned by my contouring algorithm were pretty sparse, I also employed a "closing" algorithm to clean up the lines. This involved 3 iterations of dilation which grows the pixel clouds and 1 iteration of erosion.

4.2 Closing Implementation:

```
def run_closing(contour, kernel_size):
    # Closing is dillation followed by erosion
    kernel = np.ones((kernel_size, kernel_size), dtype=np.uint8)

    contour = cv2.dilate(contour.astype(np.float32), kernel, iterations=3)
    contour = cv2.erode(contour.astype(np.float32), kernel, iterations=1)
    return contour
```

5 Results:

5.1 Original Images:

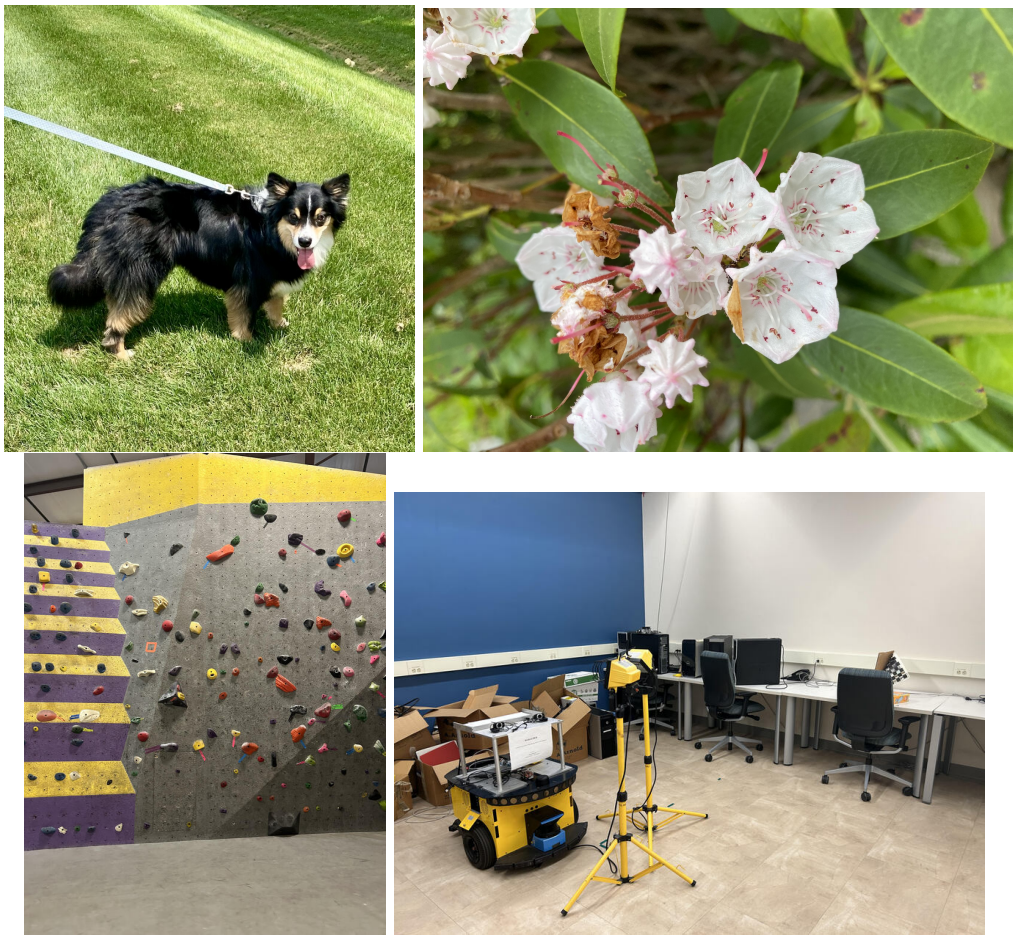

Figure 1: The original imaes I am performing the segmentation on.

For the climbing image, my goal was to have the colored climbing holds be the foreground and the background be the climbing wall. On the other hand, for the office image, my goal was to have the desk spaces and yellow light be the foreground and the background be the floor and walls of the room.

5.2 RGB Otsu's Results:

For the RGB images, I required an iterative approach to get optimal results. These were the hyperparamters that produced the best results:

Image Title	Green Iters	Blue Iters	Red Iters	Which class is kept
Dog	3	3	4	Class 1
Climb	3	3	3	Class 0
Flower	1	1	1	Class 0
Office	2	2	2	Class 0

Table 1: Optimal hyperperameters for the RGB Otsu segmentation.

5.2.1 Histograms:

Included below are the pixel value histograms, along with the variance at every threshold value. I only include examples for the dog image, and my climbing wall image for this section to demonstrate results when thresholding on class 0, and class 1.

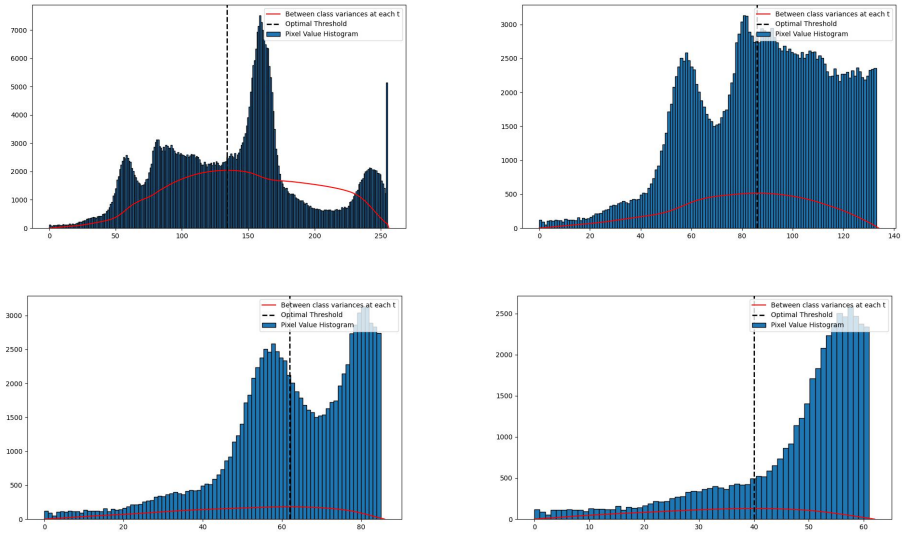

Figure 2: The histograms and variances at each threshold at iteration 0, 1, 2 and 3 for the climbing wall image.

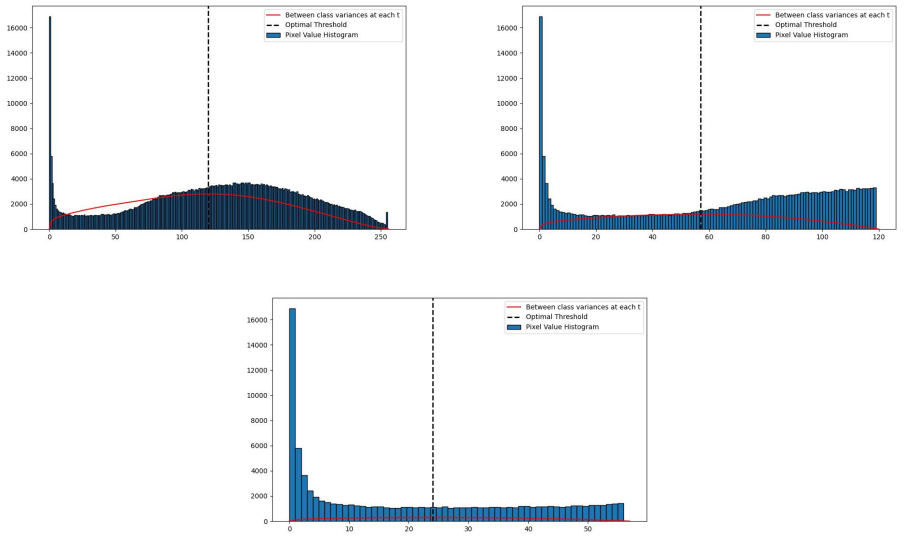

Figure 3: The histograms and variances at each threshold at iteration 0, 1, and 2 for the dog wall image.

5.2.2 Iteration Masks:

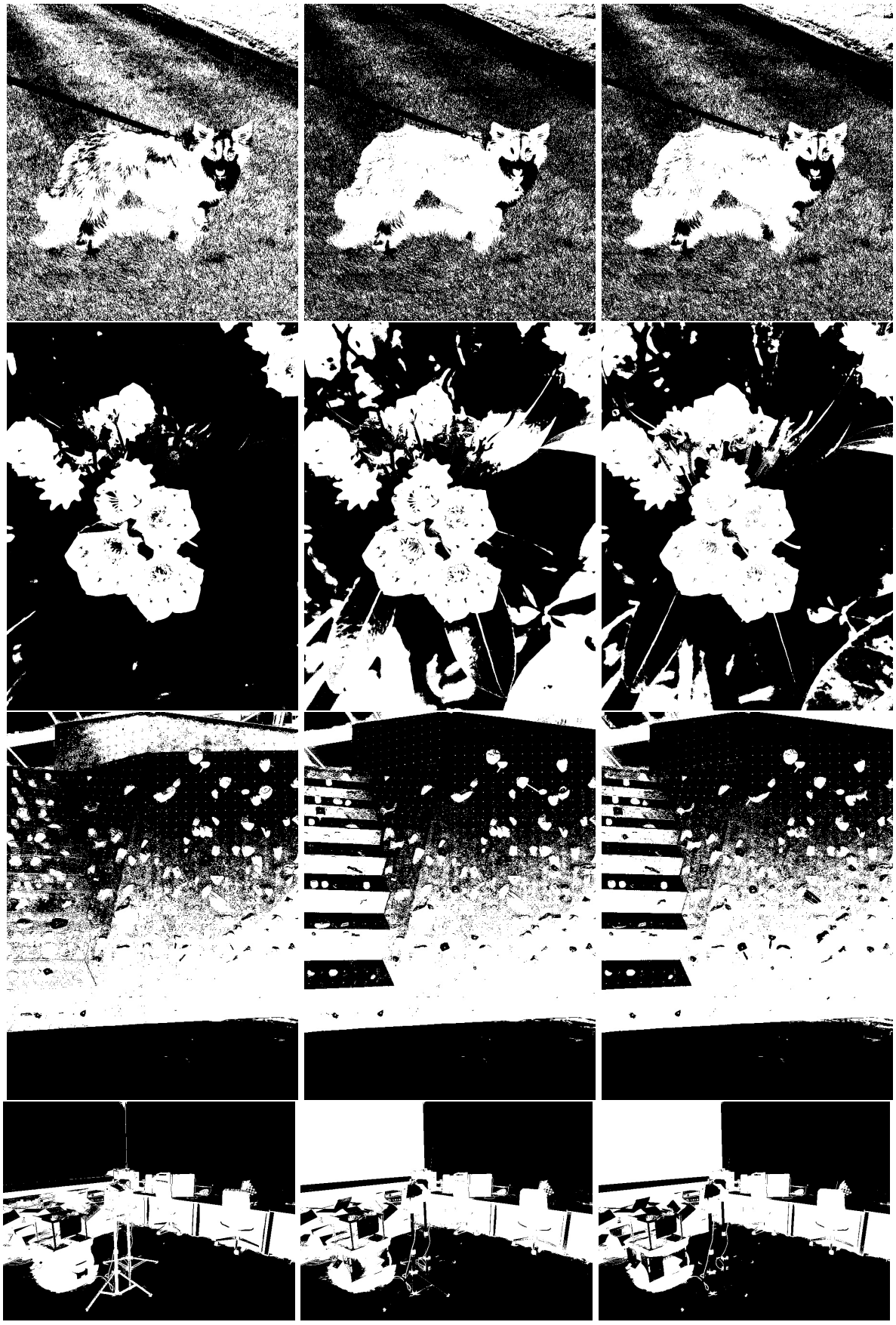

Figure 4: The masks for the first iteration for all images. The images are included in BGR ordering.

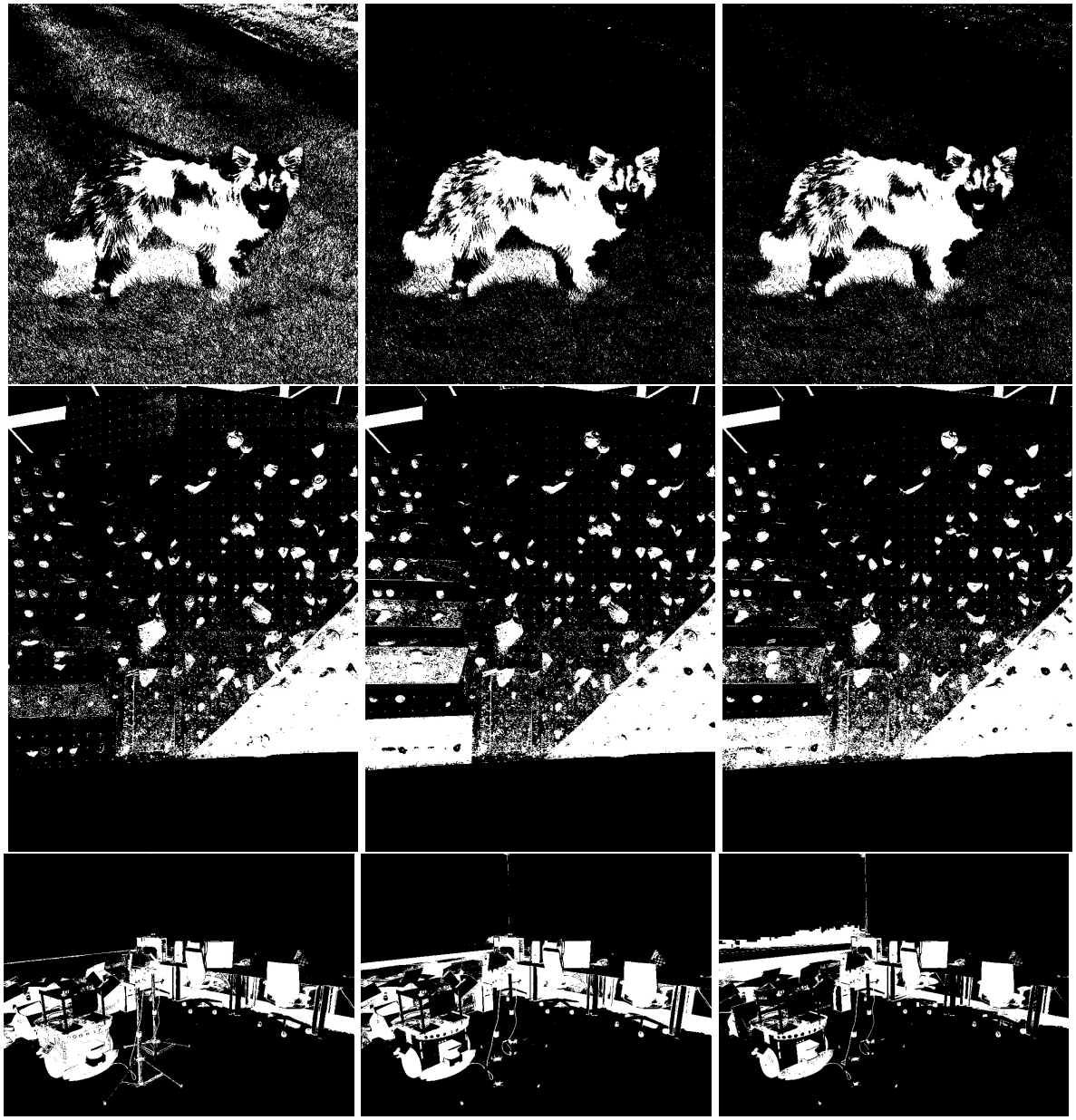

Figure 5: The masks for the second iteration for the images. The images are included in BGR ordering.

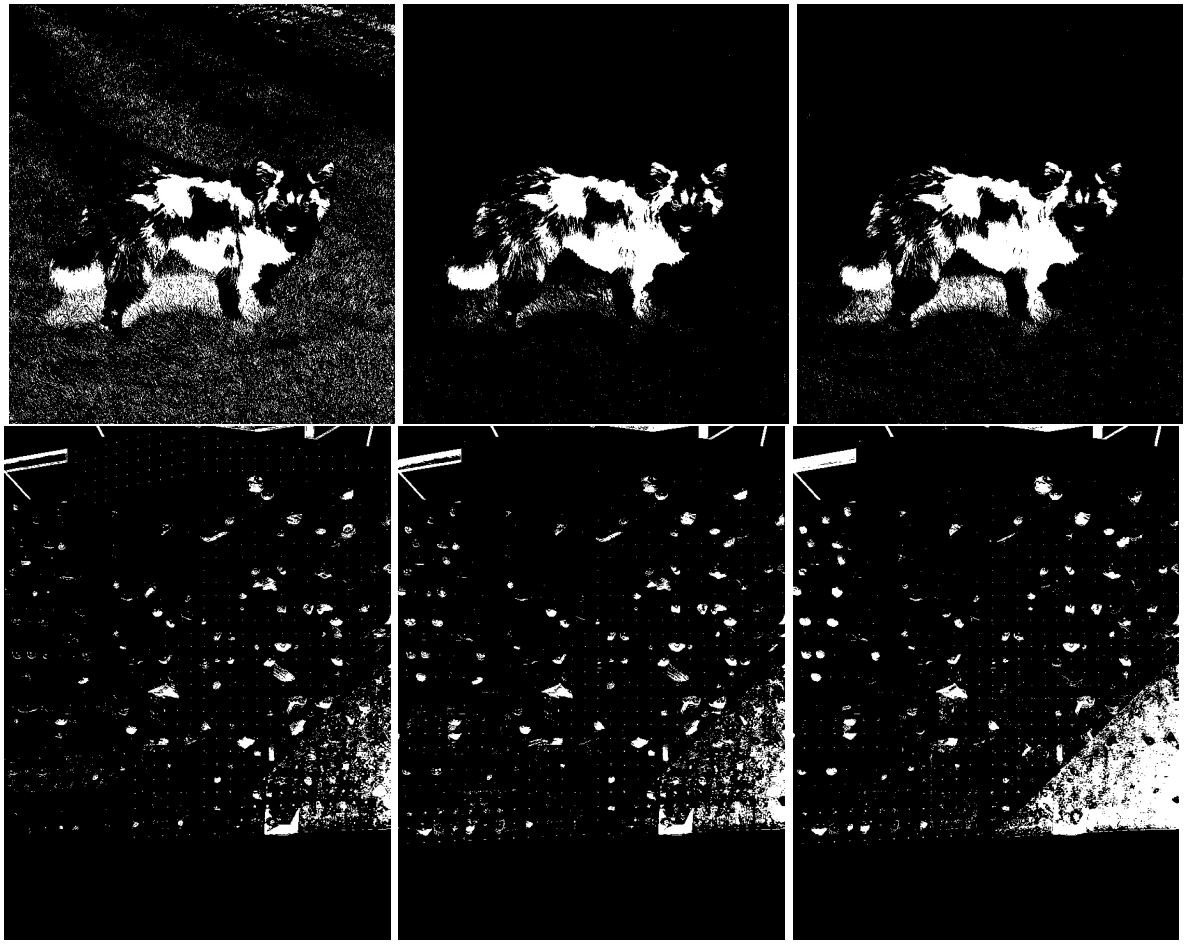

Figure 6: The masks for the third iteration for the images. The images are included in BGR ordering.

Figure 7: The masks for the third iteration for the climbing image's red channel.

5.2.3 BGR Final Masks:

For the following image, I plot the final graph but instead of just created a logical and of the black and white masks, I take the final mask for each channel and re-plot it in BGR format. Therefore, a white pixel in channel 1's output would result in a blue pixel in this image, while a white pixel in channel 2's output mask would result in a green pixel etc. This lets us understand which components of the image

are segmented from which channel through the thresholding.

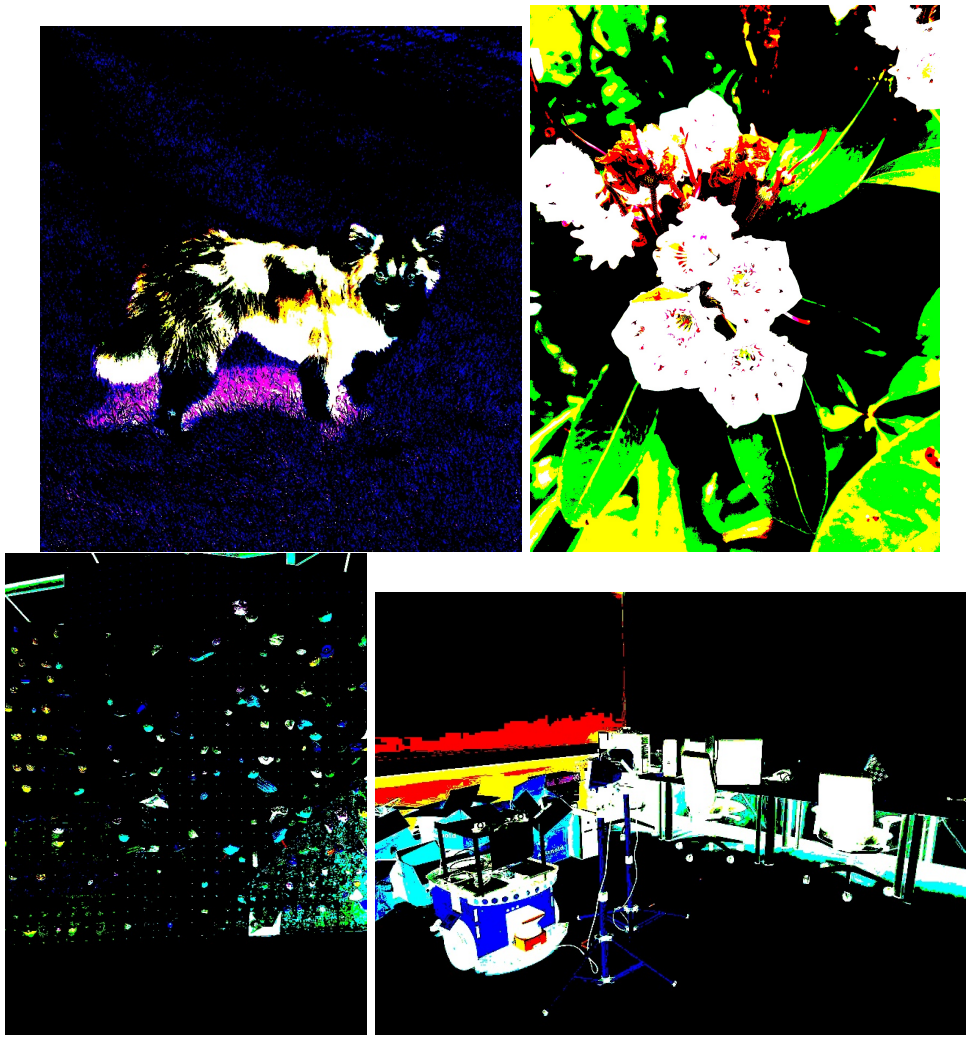

Figure 8: The final BGR channel masks for each image.

5.2.4 Segmentation Result:

Lastly, I include below the final Segmentation results for the RGB based Otsu segmentation.

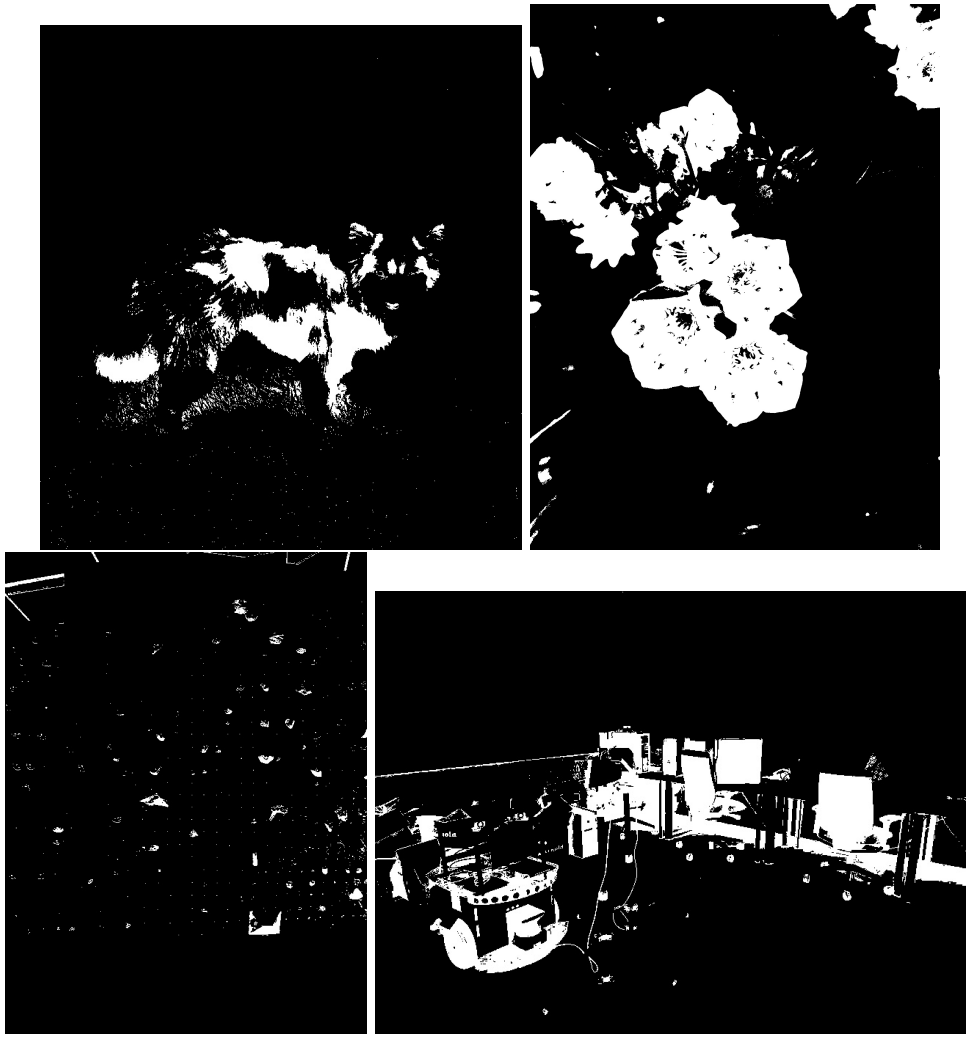

Figure 9: The final segmentation masks for each image with RGB Otsu.

5.2.5 Contouring Results:

The following are the results of the contouring pipeline on the RGB Otsu's algorithm. These take the segmentation masks and return a point cloud that represents the border pixels of the image. The first set of results do not include any post-processing.

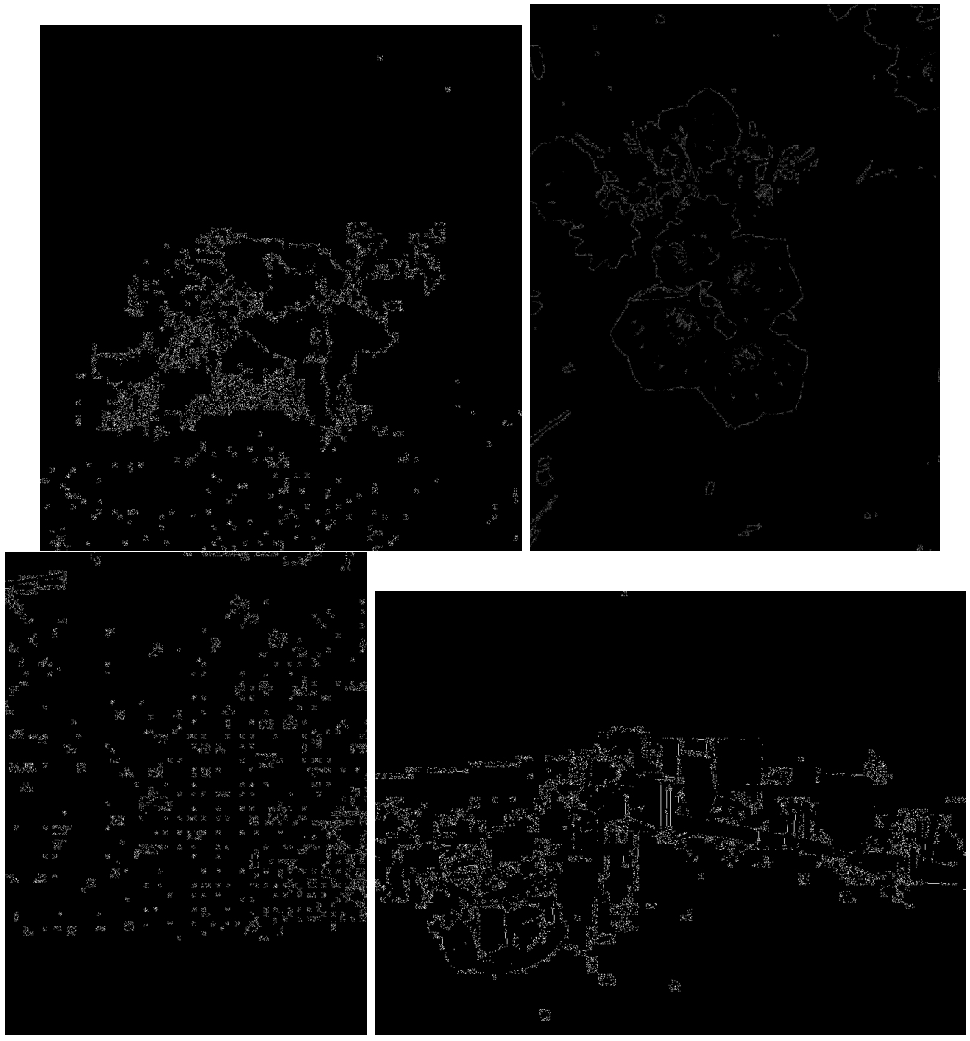

Figure 10: The image contours from the RGB Otsu's segmenation mask without any post-processing. The images are included in the following order: Dog, Flower, Climbing wall and Office space.

However, since these points clouds are pretty sparse, I decided to run erosion and dilation on the images to improve the results into continuous line segments. In this way, I ran a "closing" algorithm on the generated contours that involved 3 iterations of dilation, followed by 1 iteration of erosion which produced the best results. However, while this worked well for the flower image, you can see that for fine-grain details as seen in the climbing wall the dilation procedure overlapped with one another and created within-contour artifacts by filling in the space.

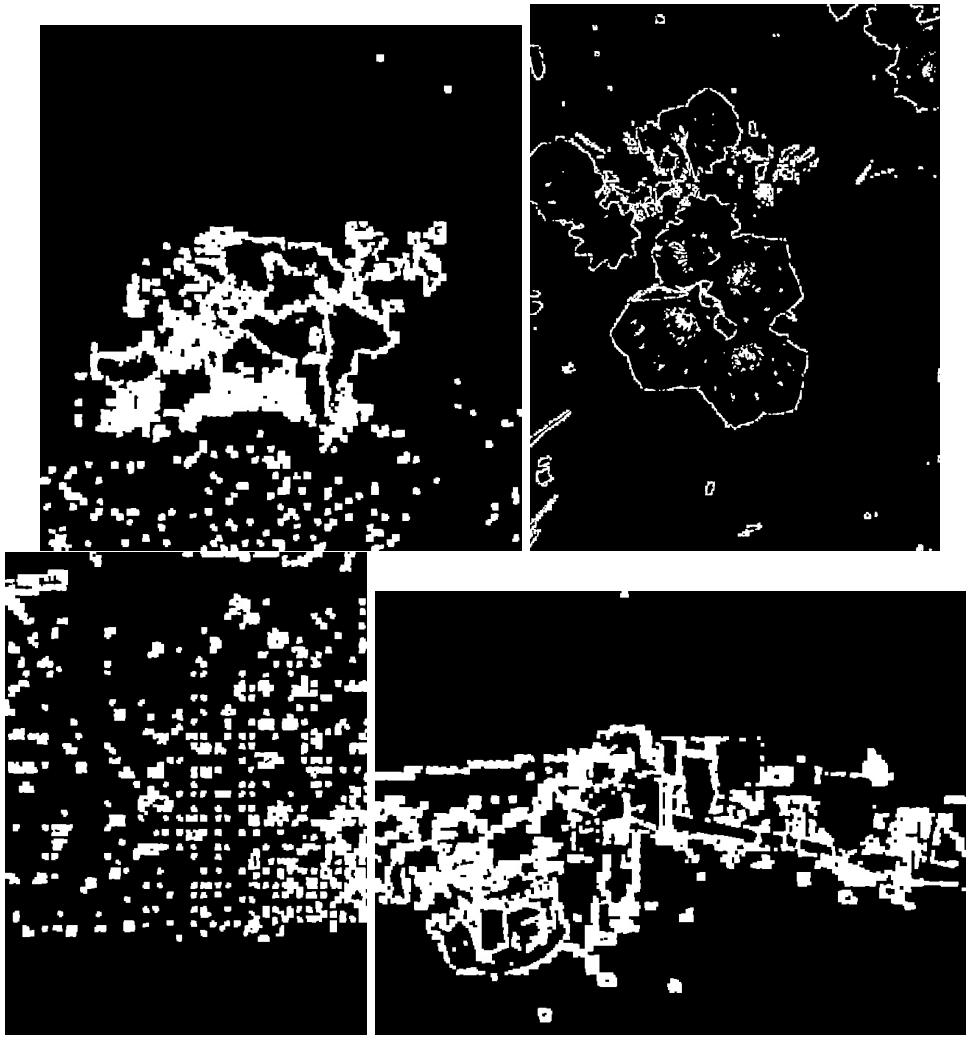

Figure 11: The image contours from the RGB Otsu's segmenation mask without any post-processing. The images are included in the following order: Dog, Flower, Climbing wall and Office space.

5.2.6 What worked and what did not work:

The best result was on the flower image where there was a very large color difference between the white flower and the green background leafs. On the other hand, the dog was especially hard to segment due to the large color differences within its coat: I could get the black portions well segemented, but that meant that I would miss the lighter brown areas. This is due to the fact that we are creating a global filter based on the intensity of the RGB values. I think it would be interesting to work on HSV or other color spaces instead of RGB and see if they algorithm performs there too. Working in that way could help to segment more holds on the climbing wall that were not pure red/green/blue. Working based on color saturation would have especially improved this image's result due to the bright color of the holds against the grey background wall. On the other hand, the office space segmentation worked very well but it oversegmented the task. In this way, it also captured the area beloow the desks instead of just the main objects.

As was previously stated, the contouring results are very promising, especially for the flower shape. The climbing wall contour suffered from lots of artifact point clouds from the bolt holes which greatly decreased the performance of the "closing" operation. This issue was also noticed to a lesser degree by the grass textures in the dog image.

5.3 Texture Based Otsu's Results:

5.3.1 Optimal hyperparamters:

Image Title	N values	Channel 1 (C_1) Iters	C_2 Iters	C_3 Iters	Class kept
Dog	[5, 4, 3]	1	1	1	Class 1
Climb	[3, 5, 7]	1	1	1	Class 1
Flower	[3, 5, 7]	2	2	2	Class 1
Office	[1, 2, 3]	3	2	2	Class 1

Table 2: Optimal hyperperameters for the RGB Otsu segmentation.

5.3.2 Histograms:

Included below are the histograms for all images along with the chosen threshold.

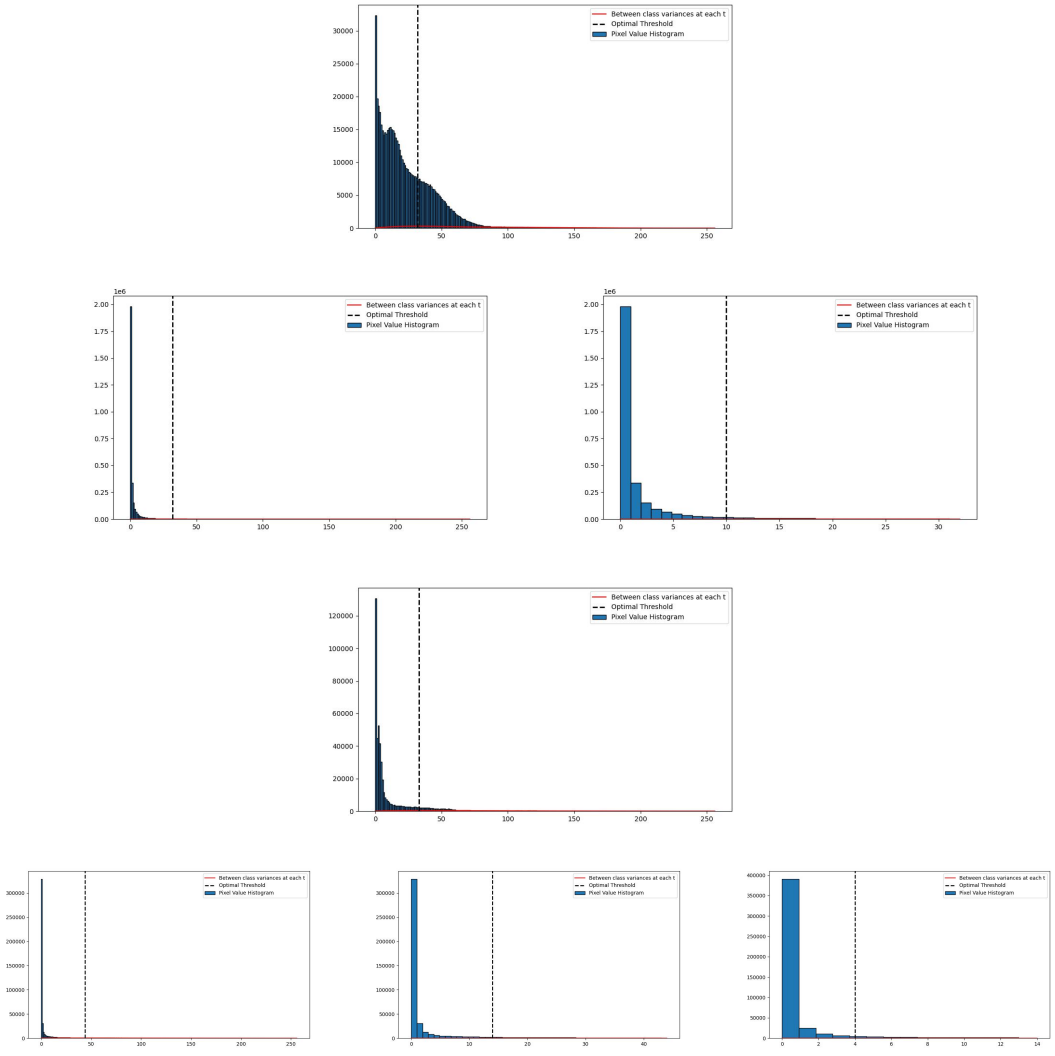

Figure 12: The histograms for all images, at all iterations. Histograms in the same row belong to the same image but at different iterations of Otsu's Algorithm. The columns are in the following order: Dog, Flower, Climbing wall and Office space.

5.3.3 Iteration Masks:

Included below are the masks for each channel, at each iteration step of Otsu's Algorithm after running the texture windowing.

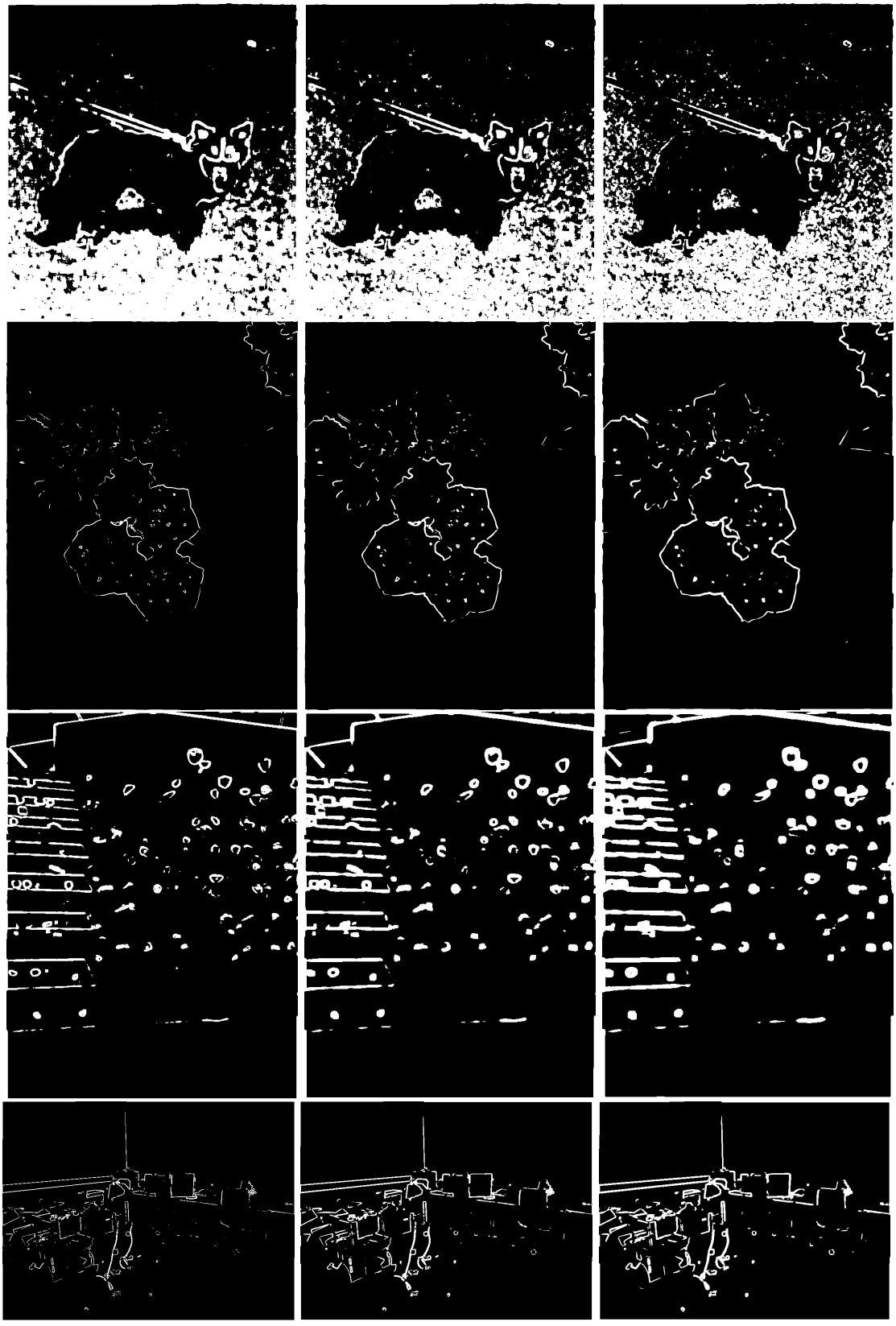

Figure 13: The results from the first iteration of Otsu's algorithm on the outputs of the Texture Windowing method. The images are in the following order: each column denotes Channel 1, Channel 2, Channel 3 as defined by the N parameter list, and the rows are Dog, Flower, Climbing wall and lastly the office space.

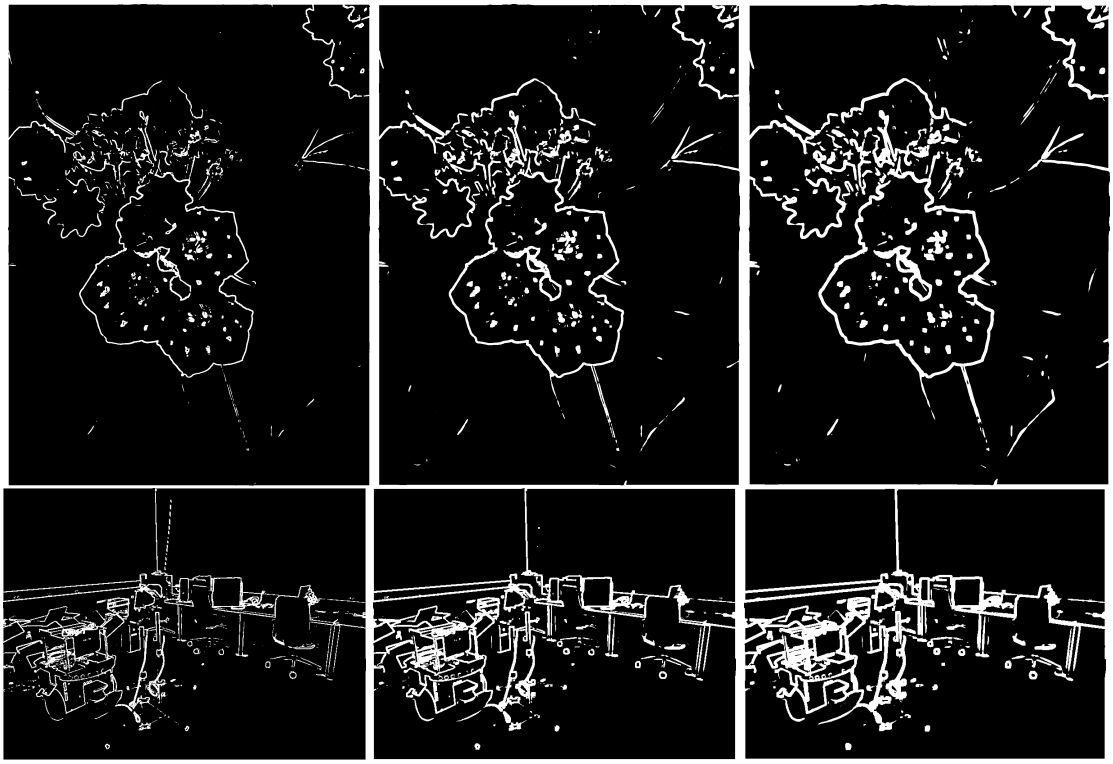

Figure 14: The results from the second iteration of Otsu's algorithm on the outputs of the Texture Windowing method. The images are in the following order: each column denotes Channel 1, Channel 2, Channel 3 as defined by the N parameter list, and the rows are flower and the office space.

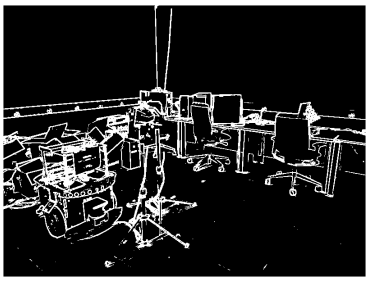

Figure 15: The results from the third iteration of Otsu's algorithm on the outputs of the Texture Windowing method. This image is from the first channel of the office space image.

5.3.4 BGR Final Masks:

For the following graphs, I take each channel's mask output and graph them as RGB values. Therefore, a white pixel from the mask in channel 1 would show up as pure blue, a white pixel from channel 2 would be pure green etc. This allows for some intuition on which channel is responsible for extracting which portion of the information.

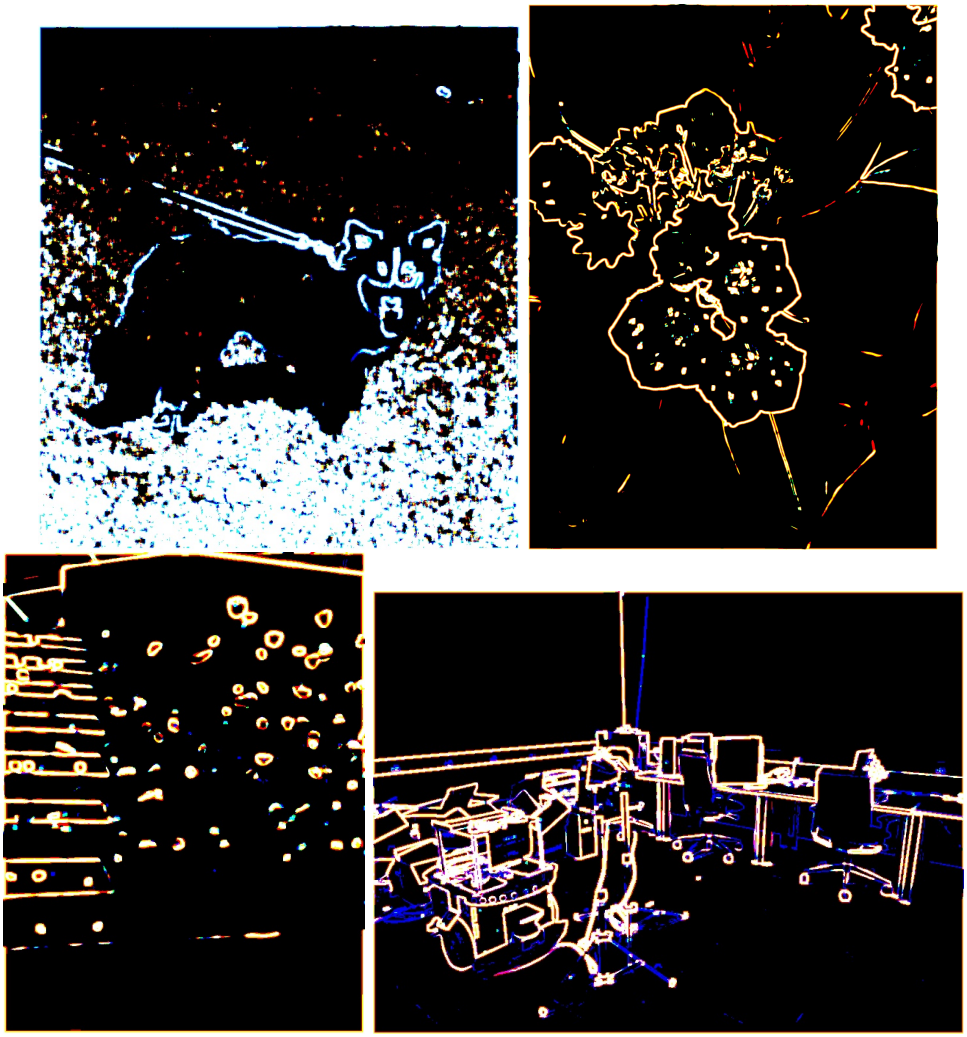

Figure 16: The BGR space outputs for the Texture based contouring. The results are in the following order: Dog, Flower, Climbing wall and Office space.

5.3.5 Final Contouring Result:

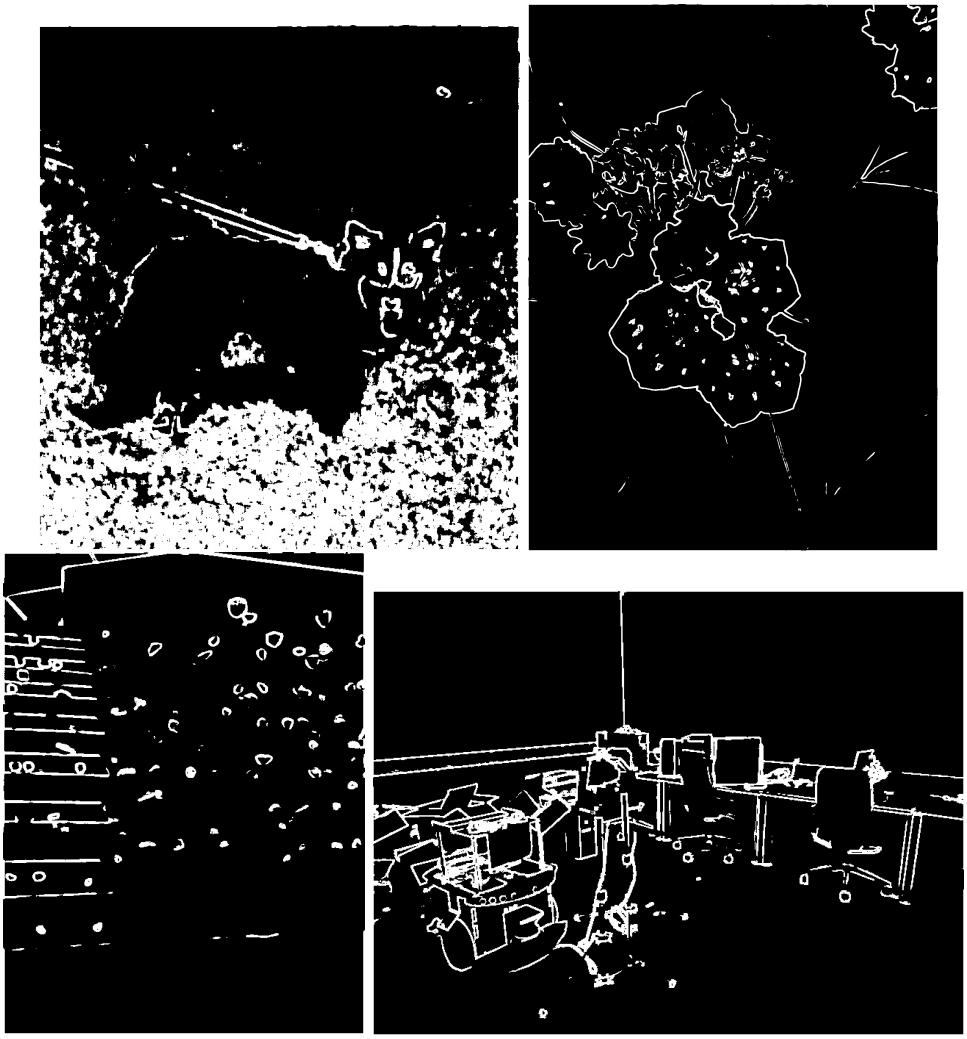

Figure 17: The final outputs for the Texture based contouring. The results are in the following order: Dog, Flower, Climbing wall and Office space.

5.3.6 What worked and what did not work:

To summarize the results for the texture based algorithm, I noticed a significant improvement in the contour output from this implementation compared to the previous implementation based on the RGB values following by contour extraction. This improvement is especially noticeable for the contour around the office space image which is very clear with this algorithm, but did not work on the RGB based method. I believe that the windowing effect based on variance was much more effective to determine boundary points compared to my previous contour implementation that naively looks for points on boundaries through the inclusion of both 1 and 0 pixels. The climbing wall also had much better performance, thoough there are larger artifacts in the horizontal edges on the left side of the wall that were not present in the RGB image. These lines are between very large shifts in color; however, they are not part of the actual climbing holds so an optimal algorithm would not segment them out. Lastly, the dog image was much more difficult to get accurate contour maps using this algorithm due to the fine detail in the grass textures. In this way, the algorithm would extract all of the grass textures before finding the contour of the dog due to the fine-grain differences in pixel values from one blade of grass to another. It may be necessary to somehow mask out these known green background first before running this algorithm and see how the performance improves.

6 Complete code printout:

```
# Imports
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os

contour_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/Output_Images/
    Contour_Extraction/"
rgb_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/Output_Images/RGB_Otsus/"
texture_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/Output_Images/
    Texture_Segmentation/"
histogram_paths = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/Histograms/"

# Split images into RBG:
def get_bgr_split_images(img_path):
    img = cv2.imread(img_path)
    return cv2.split(img)

# Display image Object:
def display_img_object(img, isOpenCV=True):
    if isOpenCV:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(img)
    plt.show()

# RBG Segmentation
def get_Otsus_segmentation(img_channel, img_name, iter, subfolder):
    num_bins = int(np.max(img_channel) - np.min(img_channel) + 1)
    hist, bin_edges = np.histogram(img_channel, bins=num_bins)

    # Get histogram probabilities
    p_i = hist / sum(hist)

    # List to save between class variances
    variances = np.zeros((len(bin_edges)))

    # Go through all threshold ts and find the one that maximizes the fisher
    discriminant
    # I only work with the pixels left from the previous iteration of Otsu's algorithm.
    # So, my loop for the t values can only get larger than the previous one
    for t in range(len(bin_edges)):
        # Probability of picking points from each class = sum(p(i))
        w0 = np.sum(p_i[:t])
        w1 = np.sum(p_i[t:])

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

        # Mean levels for each class = sum(i * p(i))
        i = np.arange(len(p_i))
        mu0 = np.sum(i[:t] * p_i[:t])/w0
        mu1 = np.sum(i[t:] * p_i[t:])/w1

        # Between class varianced
        omegaB = w0 * w1 * (mu1 - mu0)**2

        # Append fisher's discriminant to the variances list
        variances[t] = omegaB

    optimal_threshold = np.argmax(variances)

    # Save a graph of the histogram with the variances at each t plotted on top
    plt.figure(figsize=(10, 6))
    plt.bar(bin_edges[:-1], hist, width=np.diff(bin_edges), edgecolor="black", align="
    edge", label="Pixel Value Histogram")
    plt.plot(variances, color='r', label="Between class variances at each t")
    plt.axvline(x=optimal_threshold, color="k", linestyle="--", linewidth=2, label="
    Optimal Threshold")
    plt.legend(loc='upper right')
    plt.savefig(histogram_paths + subfolder + img_name + "_hist_" + str(iter) + ".jpg",
    format="jpg")
```

```
    plt.close()

    # Return the t that maximized fisher's ratio
    return optimal_threshold


# Texture Segmentation
def get_texture_ostu(grey_img, N):
    # Create canvas for output image
    output_channel = np.zeros_like(grey_img)

    # Pad the image with extra 0s as necessary
    padded_img = np.pad(grey_img, pad_width=N, mode="constant", constant_values=0)

    # Create an N by N window of the nearby pixels with 0 padding
    # Instead of using a double for loop which is slow, I do this through numpy vectorized functions
    # This will return a numpy array of shape: (H, W, 2N + 1, 2N + 1) (list of windows)
    # We do 2N + 1 since we want N on each direction from the input pixel
    window_shape = (2*N + 1, 2*N + 1)
    windows = np.lib.stride_tricks.sliding_window_view(padded_img, window_shape)

    # Substract the window's mean value
    window_means = np.mean(windows, axis=(-2, -1))
    windows = windows - window_means[:,:, None, None]

    # Compute the variance within each window
    window_variances = np.var(windows, axis=(-2, -1))

    # Normalize the variance to 0-1 range and scale to 255 for BW image
    output_channel = (window_variances - window_variances.min()) / (window_variances.max() - window_variances.min()) * 255

    # Return a new array with the within pixel variance as the center pixel value
    # These values are normalized to 0-255
    return output_channel.astype(np.uint8)


def get_contours(img):
    window_shape = (3,3)
    windows = np.lib.stride_tricks.sliding_window_view(img, window_shape)

    # We only want to look at windows where the center pixel is 1:
    center_pixels = windows[:, :, 1, 1]
    center_pixel_mask = center_pixels == 1

    # Create a mask that will look in each window and check for a 0 pixel
    # If there is a 0, it will return a 1, otherwise return a 0
    zero_mask = np.any(windows == 0, axis=(2, 3)).astype(int)

    final_mask = np.logical_and(center_pixel_mask, zero_mask)

    # Convert to grey scale (0->255)
    return final_mask * 255

def run_closing(contour, kernel_size):
    # Closing is dillation followed by erosion
    kernel = np.ones((kernel_size, kernel_size), dtype=np.uint8)

    contour = cv2.dilate(contour.astype(np.float32), kernel, iterations=3)
    contour = cv2.erode(contour.astype(np.float32), kernel, iterations=1)
    return contour

    def run_Otsus_algorithm_iterative(iters, save_path, invert, img_path=None, img=None, img_name="", histogram_folder=""):
    """
        Runs an iterative version of Otsus algorithm. Will print the following results:
            - Seperate masks for each channel dimension
            - A joined mask highlighting the RGB extractions for each mask
            - The histogram with variance values at each tested threshold for Otsu's algorithm.

    Args:
```

```
        iters (int): number of iters to run Otsus
        save_path (string): path pointing to where you want the images to be saved
        img_path (string, optional): Path to the image. Defaults to None.
        img (OpenCV Image Object, optional): OpenCV BGR Image. Defaults to None.
        img_name (str, optional): NameID for image printouts. Defaults to "".
    """

    print("Working on the following image:", img_name)
    if img_path != None:
        channels = get_bgr_split_images(img_path)
    else:
        channels = cv2.split(img)

    final_rgb_img = np.zeros((channels[0].shape[0], channels[0].shape[1], 3), dtype=np.uint8)
    final_mask = np.ones_like(channels[0], dtype=np.uint8)

    for i, channel in enumerate(channels):
        flattened_channel = channel.flatten()
        # Channel I am working on:
        channel_name = ["_blue", "_green", "_red"][i]
        print("Working on the following channel: ", channel_name)

        for iter in range(iters[i]):
            # As I run Otsu's algorithm, more of my pixels become 0 due to the masking.
            optimal_threshold = get_Otsus_segmentation(flattened_channel, img_name, iter, histogram_folder)

            # Print the optimal threshold:
            print("Optimal Threshold: ", optimal_threshold)

            # If the values in the channel are lessthan the threshold -> replace it with 0 in the bw_img or do the opposite.
            # This choice can be made depending on whether the foreground is a bright object, or a dark one.
            # If foreground is closer to white, you want to keep Class 1 (brighter pixels), else keep class 0 (darker pixels)
            if not invert:
                bw_img = np.zeros(channel.shape, dtype=np.uint8)
                bw_img[channel<optimal_threshold] = 255
            else:
                bw_img = np.ones_like(channel) * 255
                bw_img[channel<optimal_threshold] = 0

            # Fill in the channels for the final image:
            final_rgb_img[:, :, i] = bw_img

            # Save the masked image to a file
            cv2.imwrite(save_path + "RGB_Seperate/Iter" + str(iter) + "/" + img_name + channel_name + ".jpg", bw_img)

            # Update channel to new image for next iteration:
            # Use the bw_img as a mask on top of the old channel
            # channel = channel * (bw_img // 255)
            flattened_channel = flattened_channel[flattened_channel < optimal_threshold]


        # Final mask is a logical and of the masks from all of the final masks after all iterations of thresholding
        final_mask = np.logical_and(bw_img // 255, final_mask).astype(np.uint8)

    # Save the image with the combined masks:
    final_mask_img = final_mask * 255
    cv2.imwrite(save_path + "Final_Images/" + img_name + ".jpg", final_mask_img)

    # Save the final image in RBG with all channels masked seperately:
    cv2.imwrite(save_path + "RGB_Final_Masks/" + img_name + ".jpg", final_rgb_img)

# Calling code:
input_folder = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/pics/"
img_paths = [input_folder+img_name for img_name in os.listdir(input_folder)]

# Otsu's Algorithm for RGB images:
```

```
# Images are in the following order: Climb, Flower, dog_small, RVL
iters = [[3,3,4],
         [1,1,1],
         [3,3,3],
         [2,2,2]]
invert = [False, True, False, False]
for i, img_path in enumerate(img_paths):
    img_name = img_path.split("/")[-1][:-4]
    run_Otsus_algorithm_iterative(iters=iters[i], save_path=rgb_path, invert=invert[i],
    img_path=img_path, img=None, img_name=img_name, histogram_folder="RGB/")


# Calling code for Texture Contouring
# Images are in the following order: Climb, Flower, dog_small, RVL
iters = [[1,1,1],
            [2,2,2],
            [1,1,1],
            [3,2,2]]
invert = [True, True, True, True]
window_sizes = [[3, 5, 7],
                [3, 5, 7],
                [5, 4, 3],
                [1, 2, 3]]

# Otsu's algorithm based on texture:
for i, img_path in enumerate(img_paths):
    img_name = img_path.split("/")[-1][:-4]
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    final_mask = np.ones_like(img, dtype=np.uint8)

    # Initialize image to fill in with texture data
    new_img = np.zeros((img.shape[0], img.shape[1], 3))

    for channel_idx ,N in enumerate(window_sizes[i]):
        # Run Texture Otsu on the image:
        channel = get_texture_ostu(img, N)

        # Input the texture Otsu outputs as channels for BGR Otsu
        new_img[:,:,channel_idx] = channel
    # I now have a new image that I need to segment using BGR Otsu's Algorithm
    run_Otsus_algorithm_iterative(iters=iters[i], save_path=texture_path, invert=invert[
    i], img_path=None, img=new_img, img_name=img_name, histogram_folder="Texture/")


# Calling code for contouring based on the images with closing.
final_image_dirs = ["/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/Output_Images/
    Texture_Segmentation/Final_Images/",
                    "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW6/Output_Images/RGB_Otsus/
    Final_Images/"]
types = ["texture", "rgb"]
for type, final_img_dir in zip(types, final_image_dirs):
    for img_name in os.listdir(final_img_dir):
        # Run the contouring on that image

        # Open the image:
        img = cv2.imread(final_img_dir + img_name)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Get the image contour:
        contour = get_contours(img)

        # Save the images:
        cv2.imwrite(contour_path+img_name[:-4] + "_" + type + ".jpg", contour)

        # Run Closing on the images:
        closed_contour = run_closing(contour, 3)
        cv2.imwrite(contour_path+"Post_Closing/" + img_name[:-4] + "_" + type + ".jpg",
    closed_contour)
```