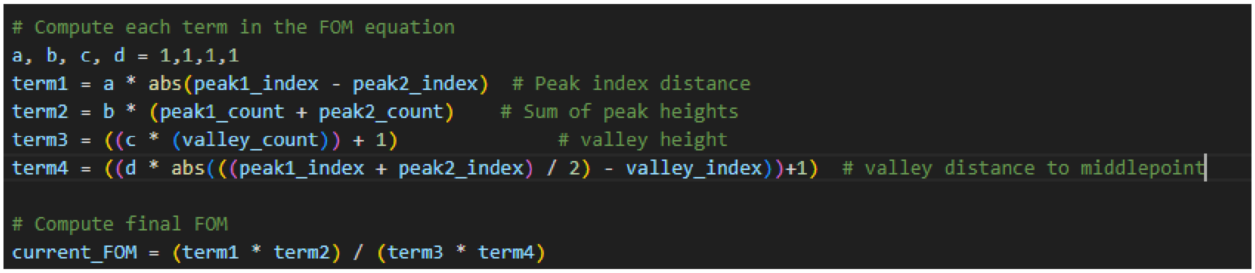
CSCI 8820 Computer Vision and Pattern Recognition
Alex Darwiche HW3

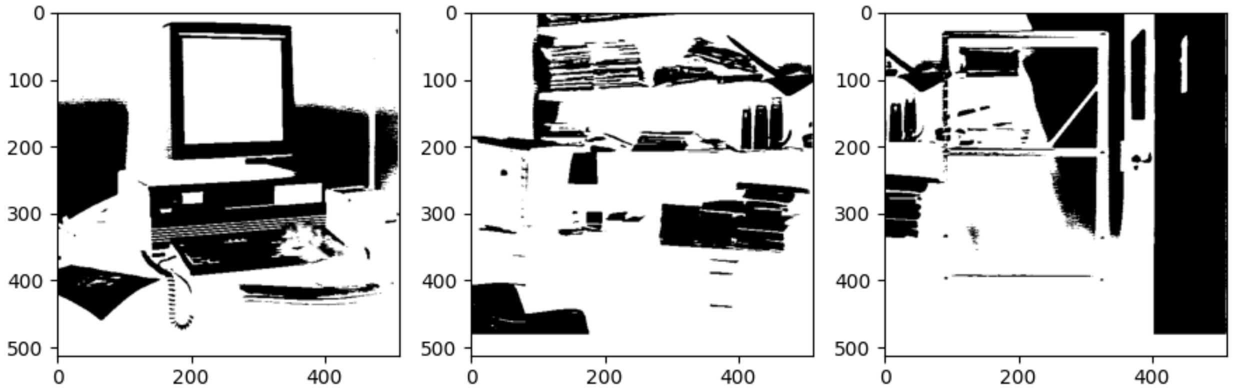
3/28/2025

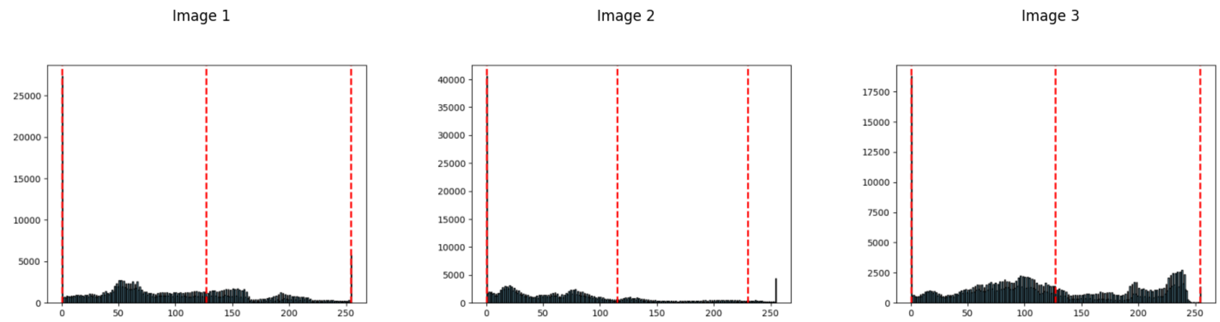
Question 1: Peakiness Detection

Explanation: To determine the peaks and valley chosen in this algorithm, I used a figure of merit that weighs 4 different things against one another. First, I was looking to maximize the 1) separation between the peaks and 2) the height of the peaks. Next, I wanted to minimize the 3) the height of the valley and 4) the difference between the middle point of the peaks and the index of the valley. Each of these constituted a "term" in my figure of merit, with each getting its own constant to control "how much we care" about that specific term. The terms to maximize were put in the numerator and the terms to minimize were put in the denominator, and I added +1 to the denominators to ensure they were never 0. Below you will see the Python code that shows how I built this figure of merit, using the same important 1, for each term in this instance. Once you determine the 2 peaks and valley that maximize this figure of merit, you can use the valley to threshold between foreground and background.



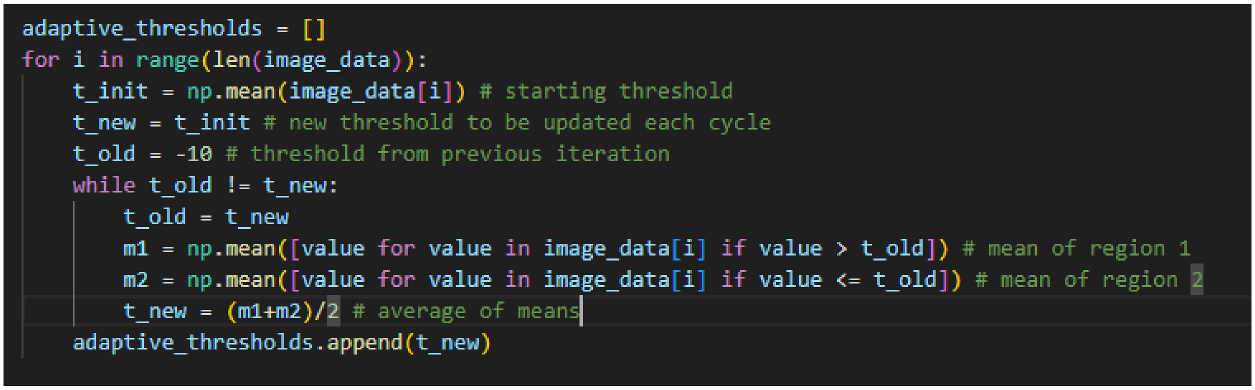
```
# Compute each term in the FOM equation
a, b, c, d = 1,1,1,1
term1 = a * abs(peak1_index - peak2_index)  # Peak index distance
term2 = b * (peak1_count + peak2_count)    # Sum of peak heights
term3 = ((c * (valley_count)) + 1)             # valley height
term4 = ((d * abs(((peak1_index + peak2_index) / 2) - valley_index))+1)  # valley distance to middlepoint

# Compute final FOM
current_FOM = (term1 * term2) / (term3 * term4)
```



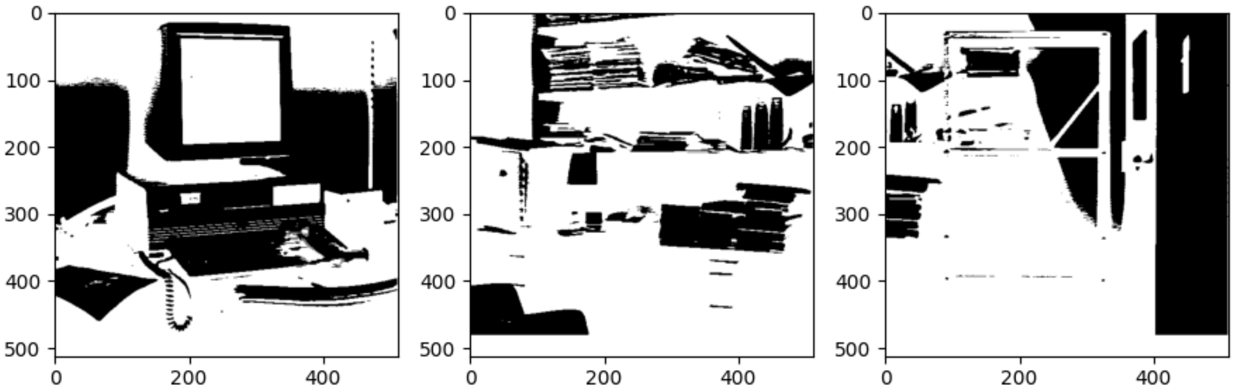


Question 2: Iterative Thresholding

Explanation: For this question, I simply started by finding the mean of the grayscales of the image. I call this T_init. Values below T_init are in region 1 and values above T_init are in region 2. I then find the mean of each region, then t_new if the average of these 2 means. I continue this process iteratively, until the t_new stops changing. The code below will show the process of finding these thresholds for each of the 3 input images.

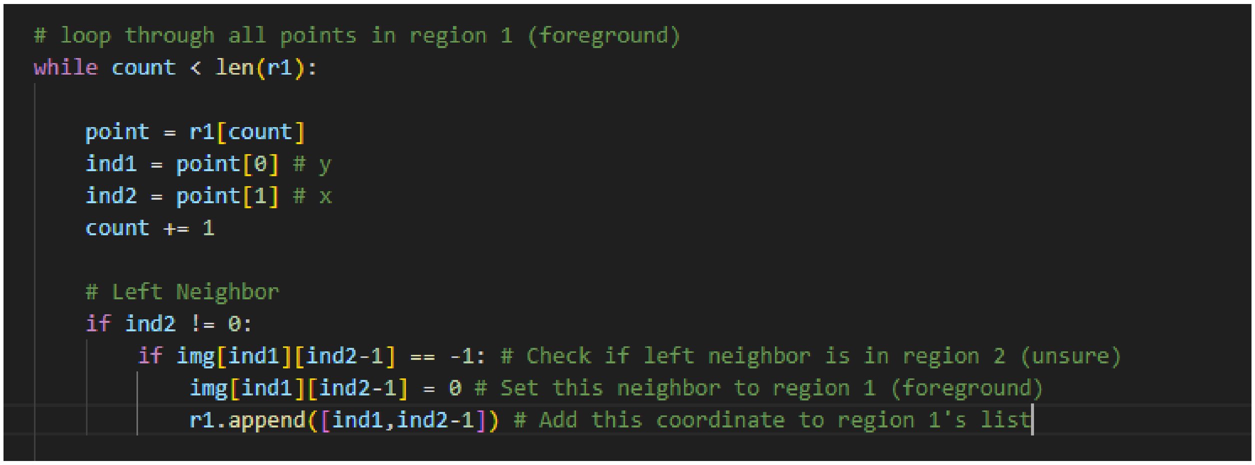


```
adaptive_thresholds = []
for i in range(len(image_data)):
    t_init = np.mean(image_data[i]) # starting threshold
    t_new = t_init # new threshold to be updated each cycle
    t_old = -10 # threshold from previous iteration
    while t_old != t_new:
        t_old = t_new
        m1 = np.mean([value for value in image_data[i] if value > t_old]) # mean of region 1
        m2 = np.mean([value for value in image_data[i] if value <= t_old]) # mean of region 2
        t_new = (m1+m2)/2 # average of means
    adaptive_thresholds.append(t_new)
```



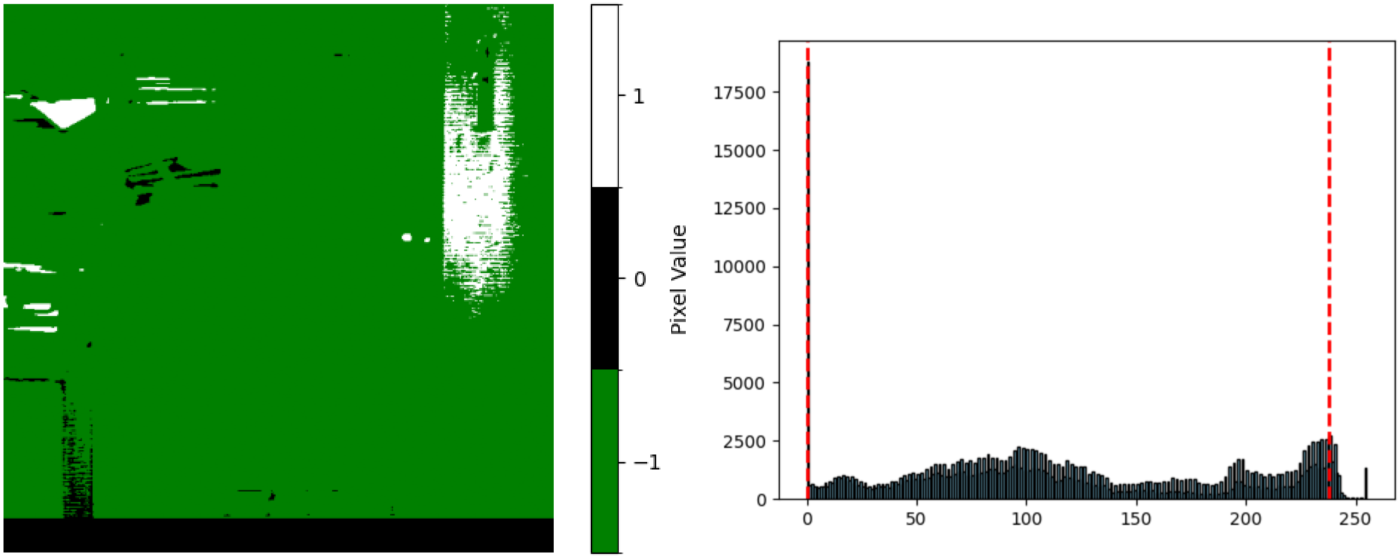
Question 3: Dual Thresholding

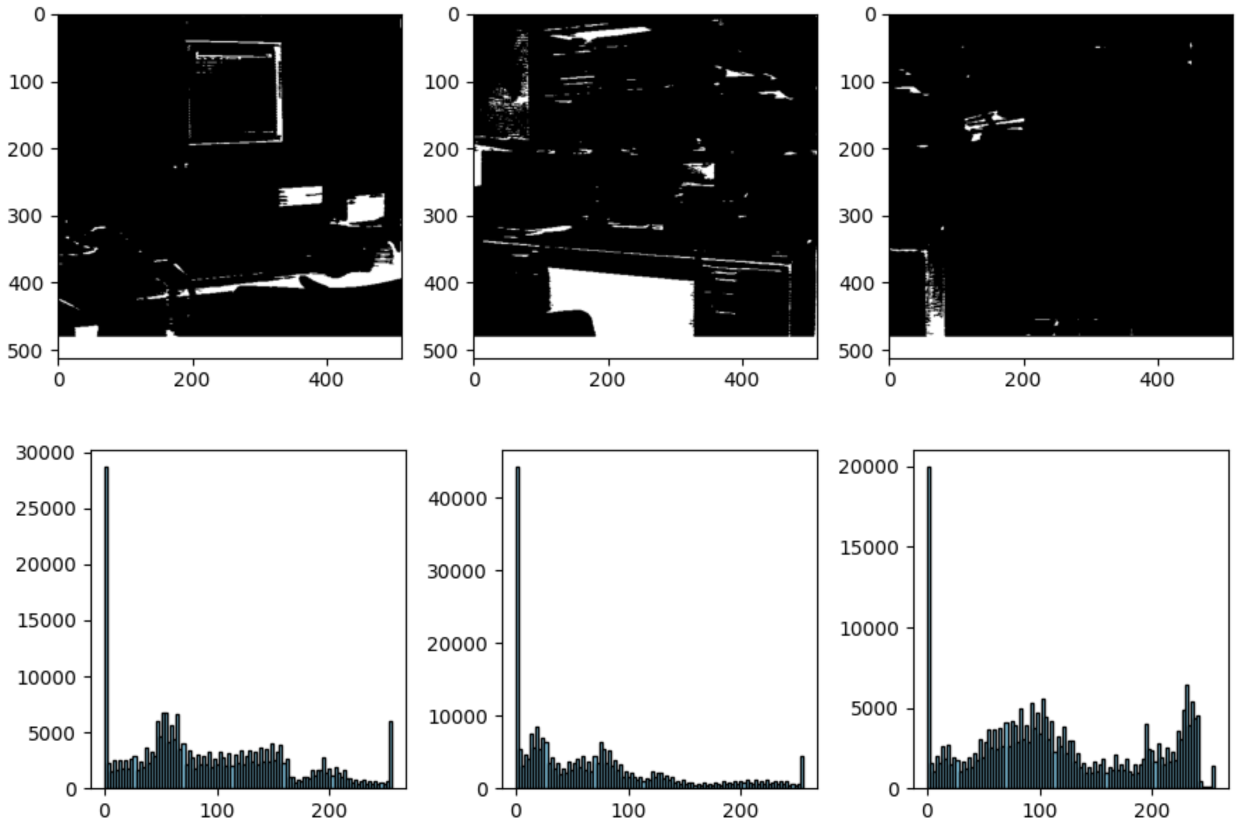
Explanation: For this problem, we first need to come up with an "automatic" way to find initial thresholds from the histogram of gray levels. There are 2 ways that I tested to do this. The first, I believe we talked about this in class, is to find the top 2 "modes" of the distribution. If we make the assumption that the gray scale levels are bi-modal, then we can treat these 2 modes as the 100%-foreground and background pixels. Anything below the first peak is in region 1 (foreground). Anything above the second peak is in region 3 (background). The pixels in the middle are then considered region 2 (unsure). I then looped through all the pixels in r1 and grew them outwards if they had 4-neighbors in region 2. If a neighbor of region 1 pixels is in region 2, I added those to the end of region 1 array. This mimics a recursive solution where we continue spreading out as long as there are "unsure" pixels, growing region 1. Once we can no longer grow region 1, we then make all the remaining pixels join region 3 (background). **This approach did not seem to be successful for this set of images, likely due to the modes being relatively far to the edges of the grayscale histogram. The image with green below, shows the "unsure" region in one of the images. This illustrates how few pixels can get initially grouped as r1 or r3 in the initial pass. When most pixels are in r2, the algorithm will simply grow into the green space as far as it can, leaving a completely mono-color image (as seen below). The second approach achieves much better results (p-tile method).**



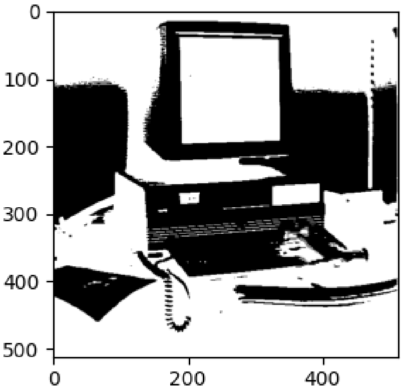
```
# loop through all points in region 1 (foreground)
while count < len(r1):

    point = r1[count]
    ind1 = point[0] # y
    ind2 = point[1] # x
    count += 1

    # Left Neighbor
    if ind2 != 0:
        if img[ind1][ind2-1] == -1: # Check if left neighbor is in region 2 (unsure)
            img[ind1][ind2-1] = 0 # Set this neighbor to region 1 (foreground)
            r1.append([ind1,ind2-1]) # Add this coordinate to region 1's list
```

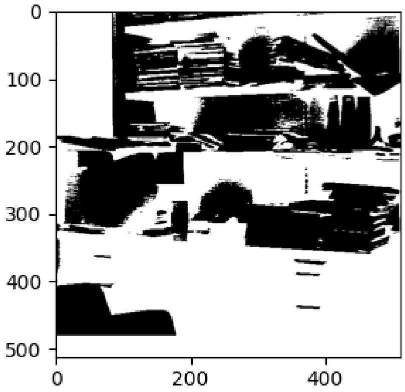
Bad Initial Results Below:

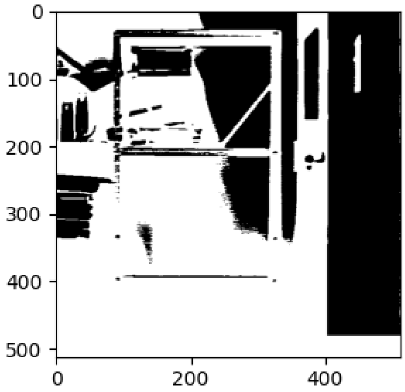

1
0
−1
Pixel Value
17500
15000
12500
10000
7500
5000
2500
0
0
50
100
150
200
250

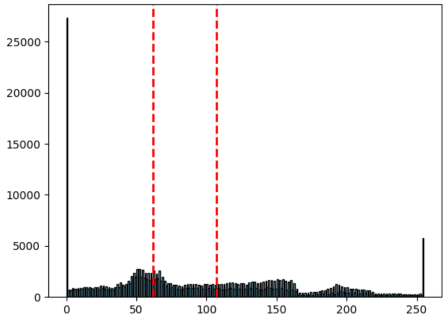

0
100
200
300
400
500
0
200
400
30000
25000
20000
15000
10000
5000
0
100
200
40000
30000
20000
10000
20000
15000
10000
5000

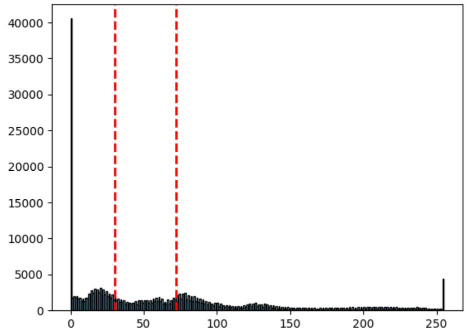
(Better Results: Method 2) Explanation: The second approach you can use to come up with an automatic threshold is to use p-tile or percentiles. Essentially, we can choose percentiles to form our region 1 and region 3 thresholds, then grow from there. In the below example, I used percentiles 40 and 60. These provided the bounds on the initial pixels, and then I grew the region 1 until it could not grow anymore. This appears to do a good job of separating the foreground and backgrounds from each other.

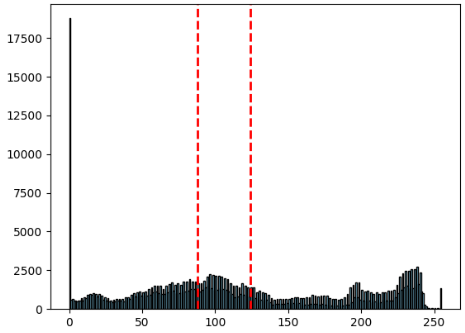

0
100
200
300
400
500
0
200
400


0
100
200
300
400
500
0
200
400


0
100
200
300
400
500
0
200
400


25000
20000
15000
10000
5000
0
0
50
100
150
200
250


40000
35000
30000
25000
20000
15000
10000
5000
0
0
50
100
150
200
250


17500
15000
12500
10000
7500
5000
2500
0
0
50
100
150
200
250