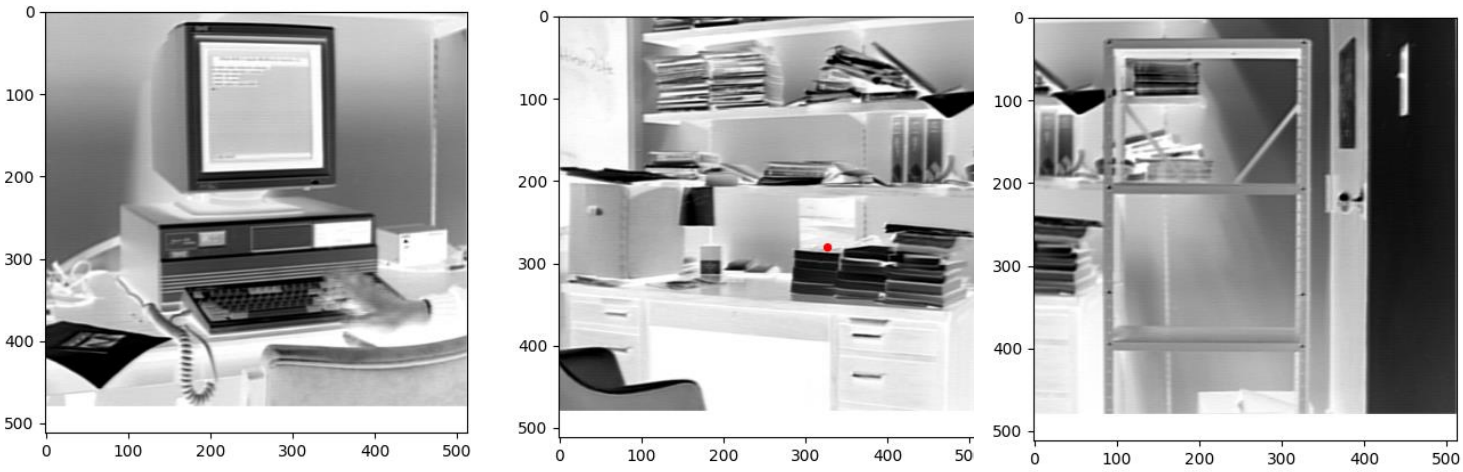
CSCI 8820 Computer Vision and Pattern Recognition
Alex Darwiche HW3

3/28/2025

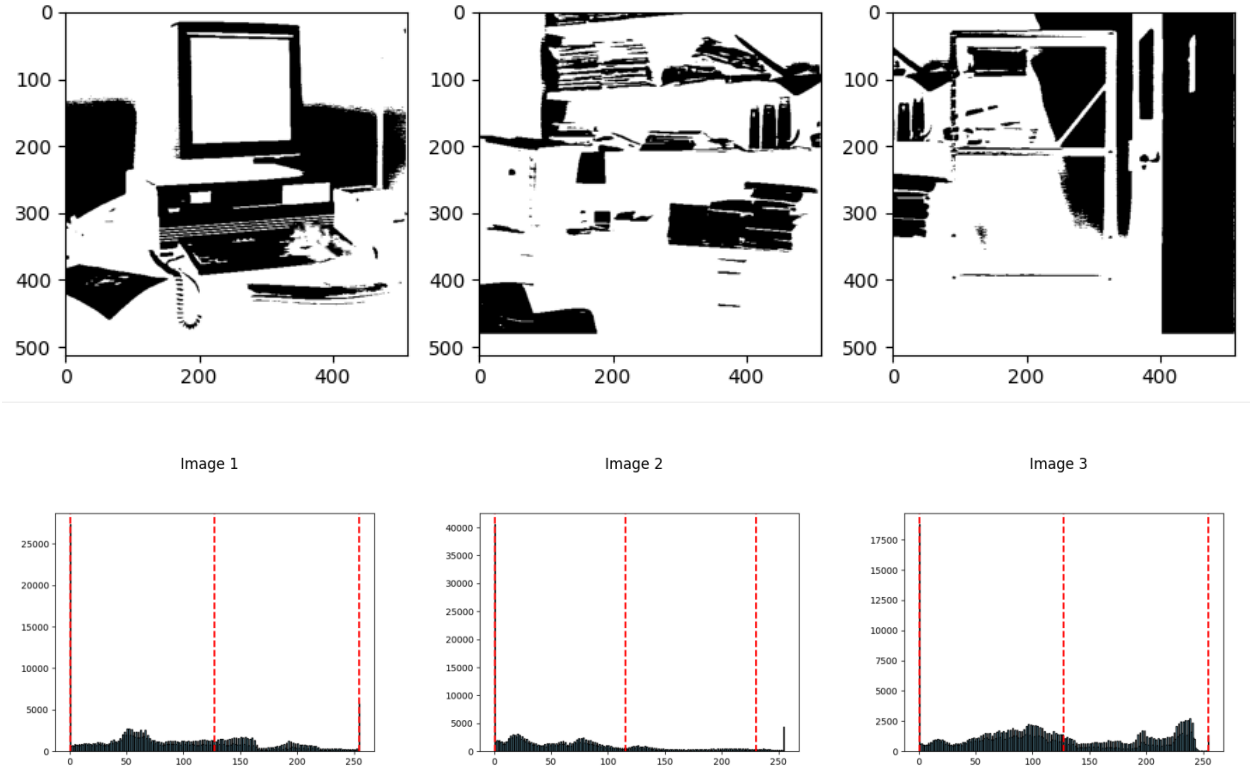
Original grayscale images:



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.

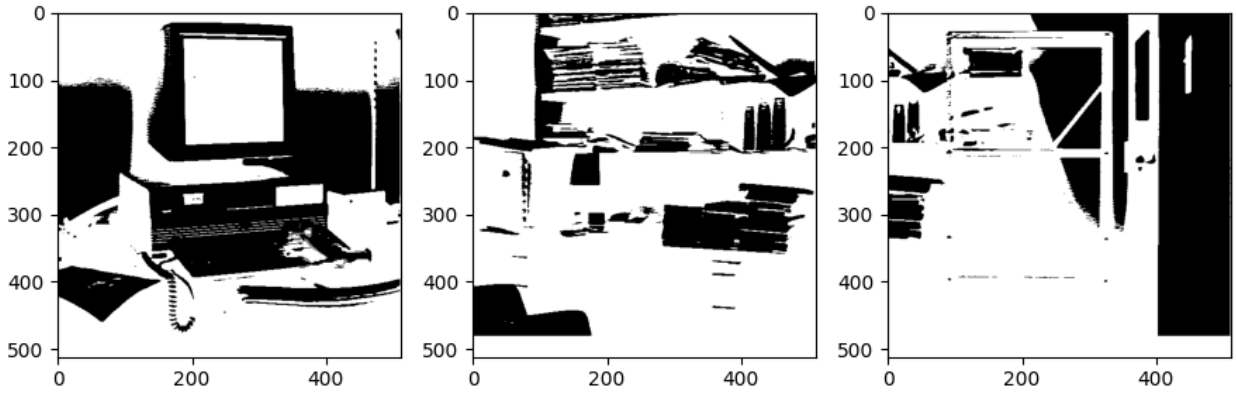
```
# Compute each term in the FOM equation
a, b, c, d = 1,1,1,1
term1 = a * abs(peak1_index - peak2_index)  # Peak index distance
term2 = b * (peak1_count + peak2_count)    # Sum of peak heights
term3 = ((c * (valley_count)) + 1)              # valley height
term4 = ((d * abs(((peak1_index + peak2_index)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

```python
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)
```



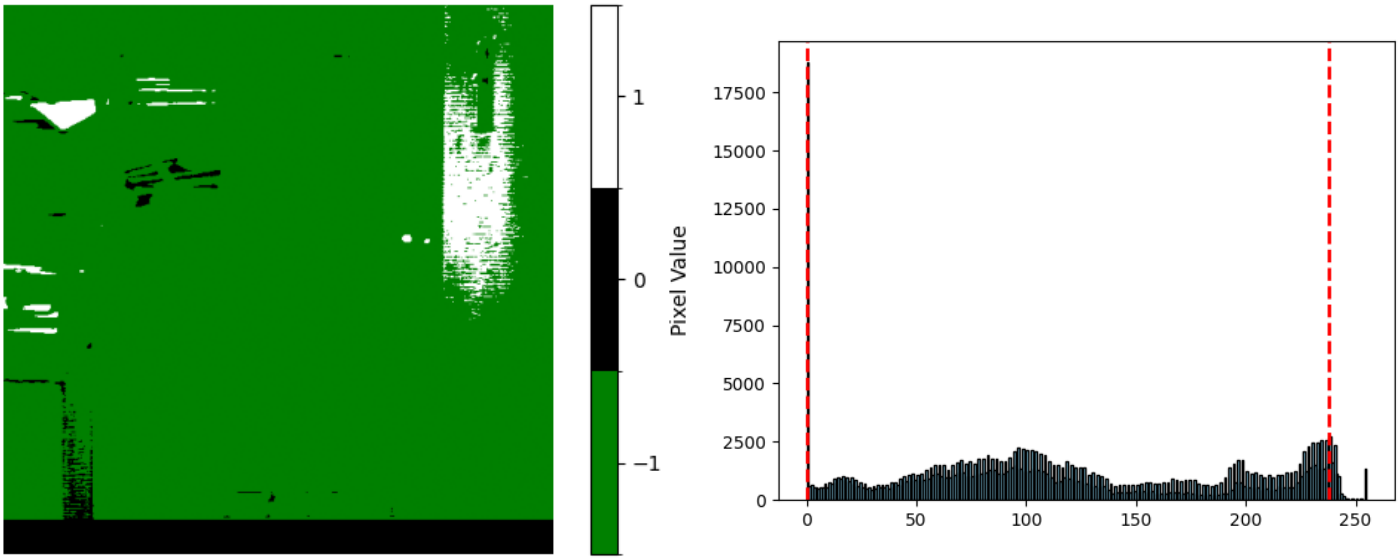
Question 3: Dual Thresholding

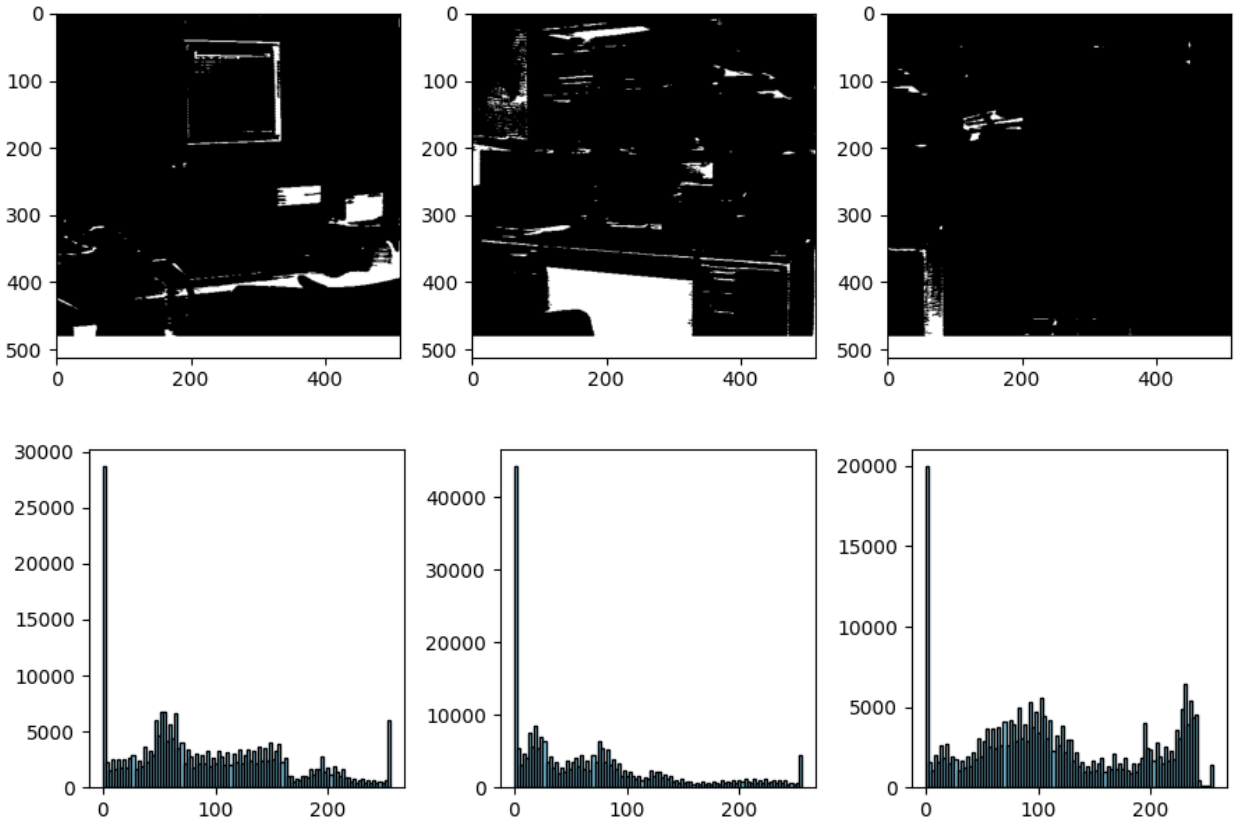
Explanation: For this problem, we first need to come up with an "automatic" way to find initial thresholds from the histogram of gray levels. There are 2 ways that I tested to do this. The first, I believe we talked about this in class, is to find the top 2 "modes" of the distribution. If we make the 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).

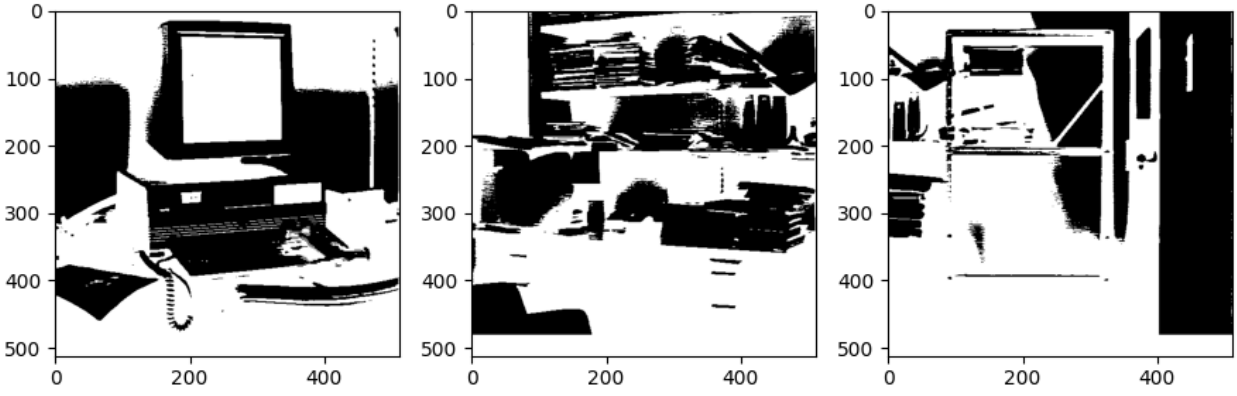
```
# loop through all points in region 1 (foreground)
while count < len(r1):

    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
```

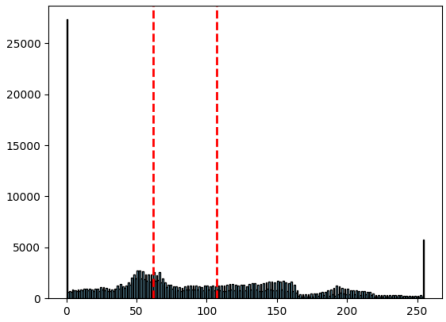
Bad Initial Results Below:

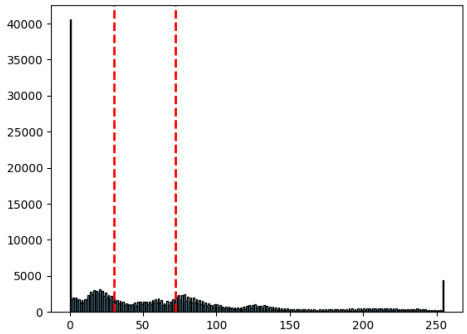


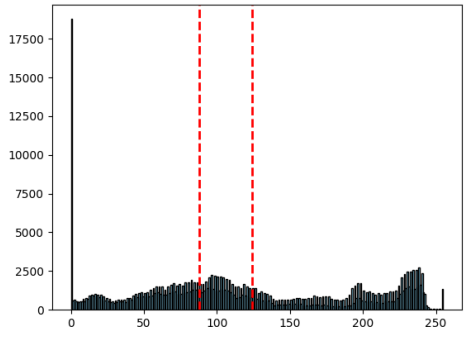


(Better Results: Method 2) Explanation: The second approach you can use to come up with an automatic threshold is to use p-tile or percentiles. Essentially, we can choose percentiles to form our region 1 and region 3 thresholds, then grow from there. In the below example, I used percentiles 40 and 60. 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.



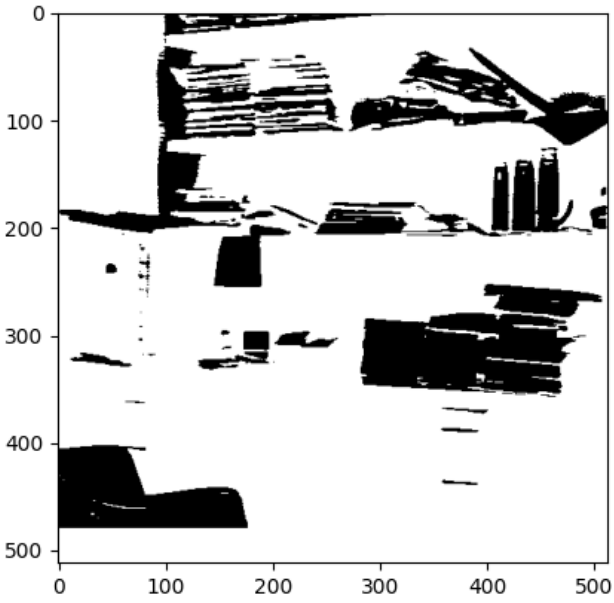


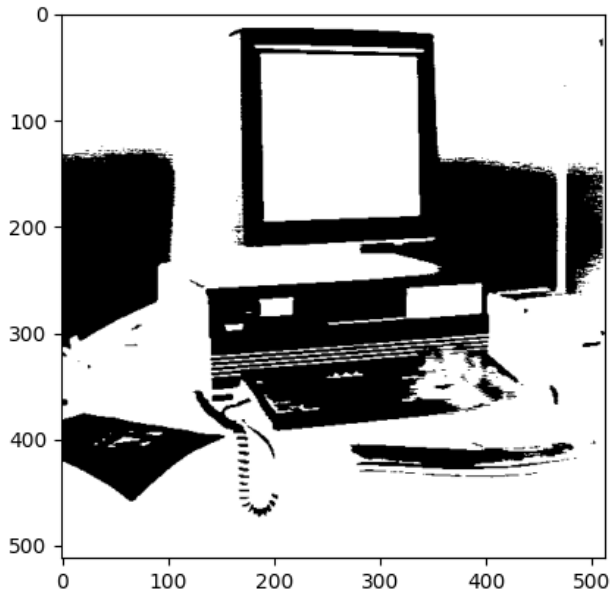


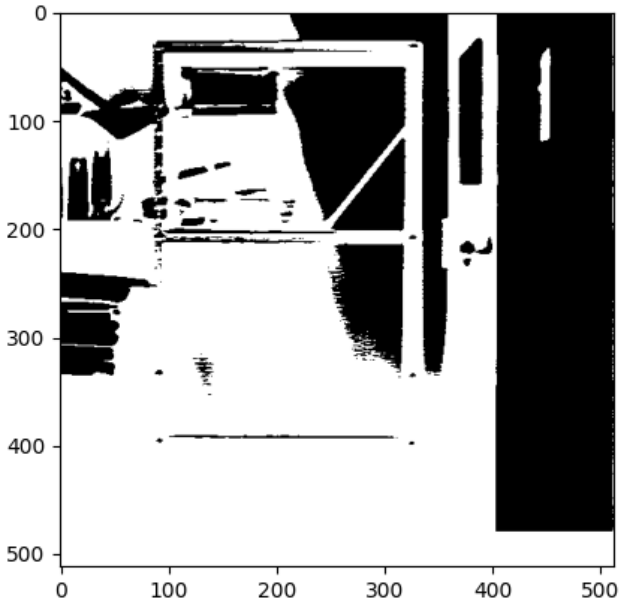


Individual Images (in case the ones above are difficult to see):

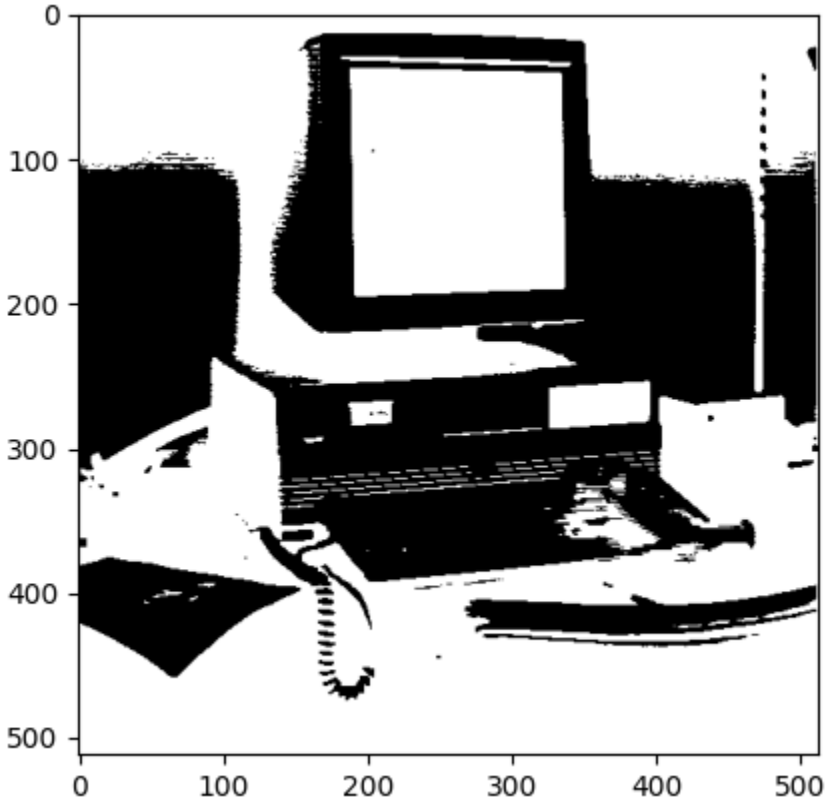
Question 1 Images:

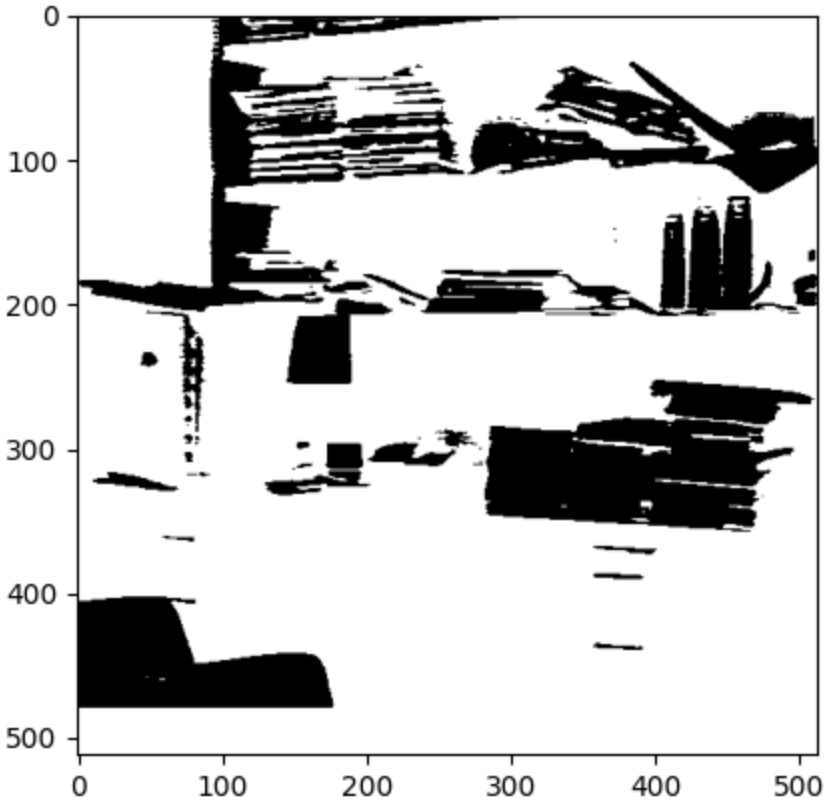

0
100
200
300
400
500
0
100
200
300
400
500

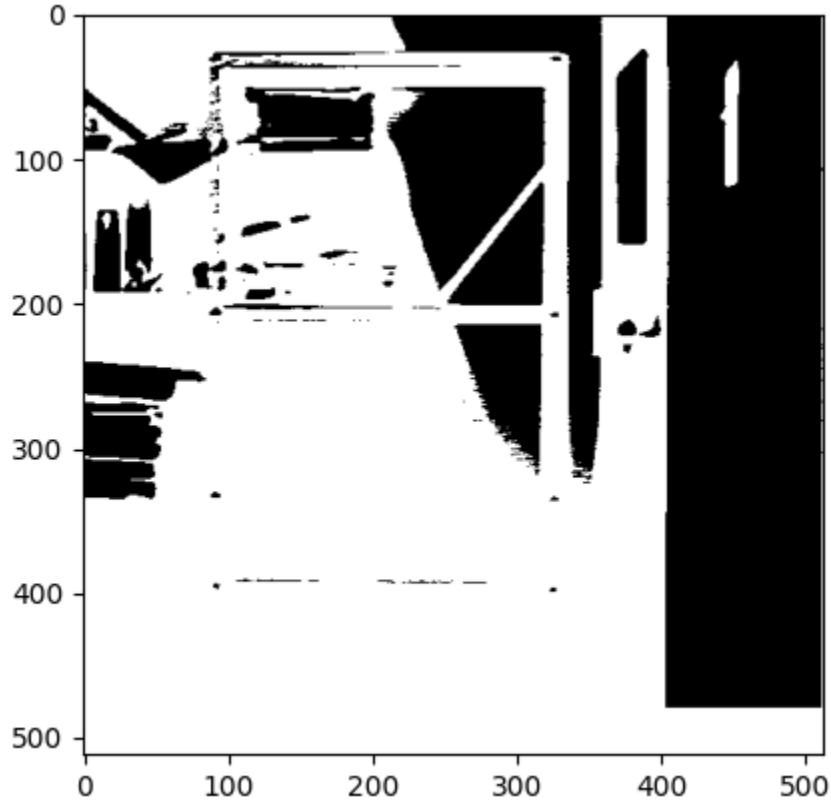

0
100
200
300
400
500
0
100
200
300
400
500



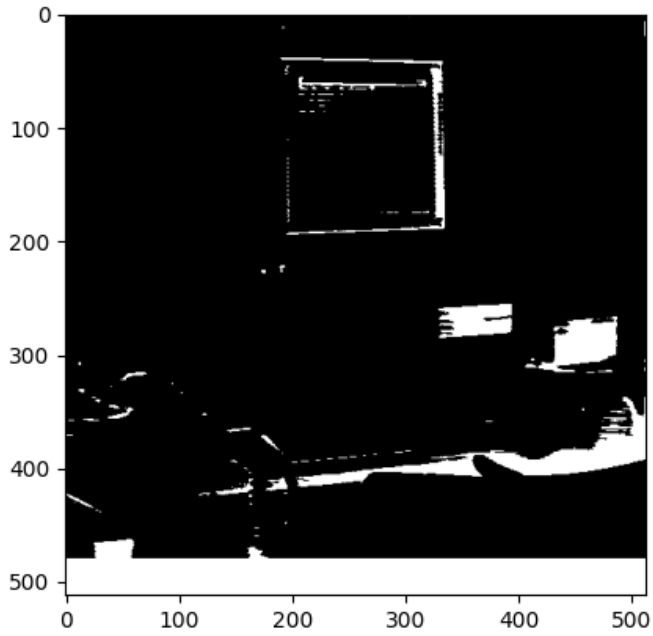
Question 2 Images:

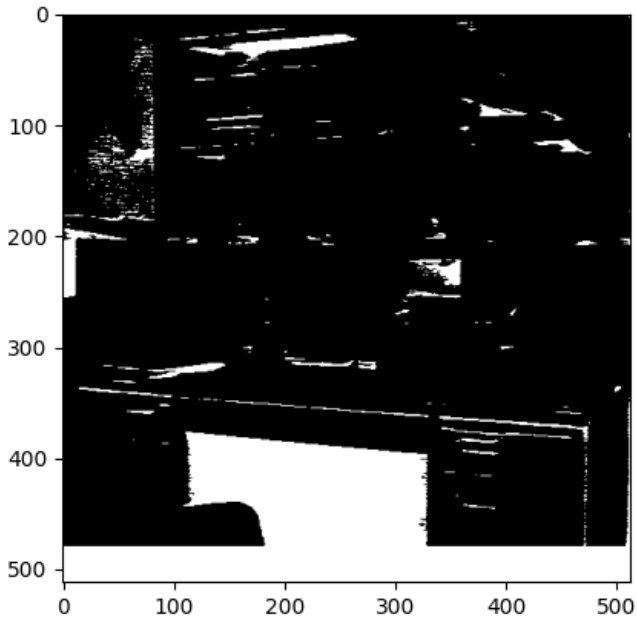


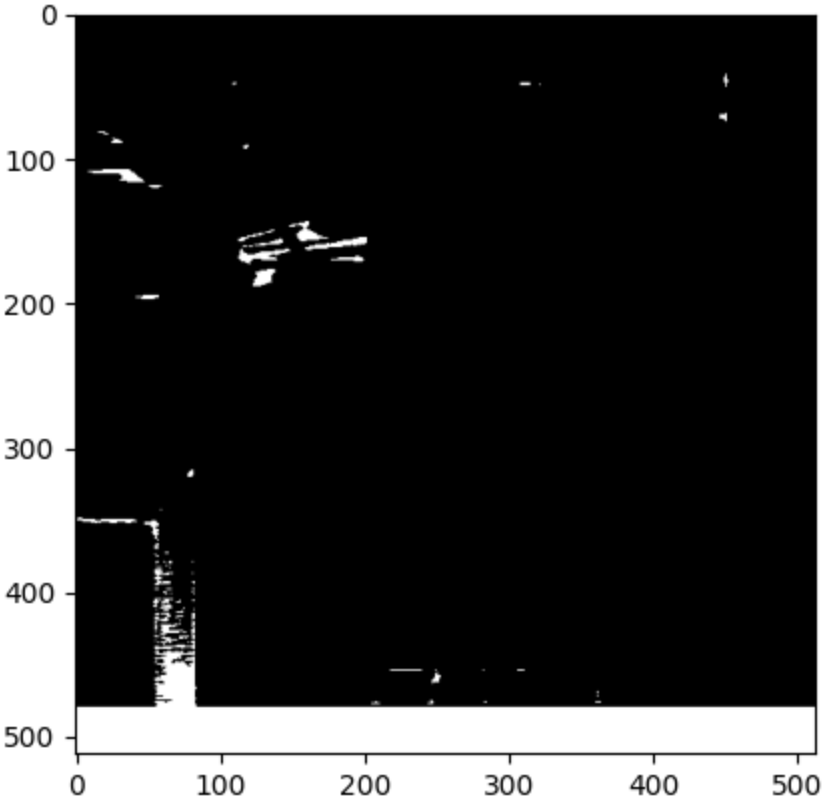

0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500

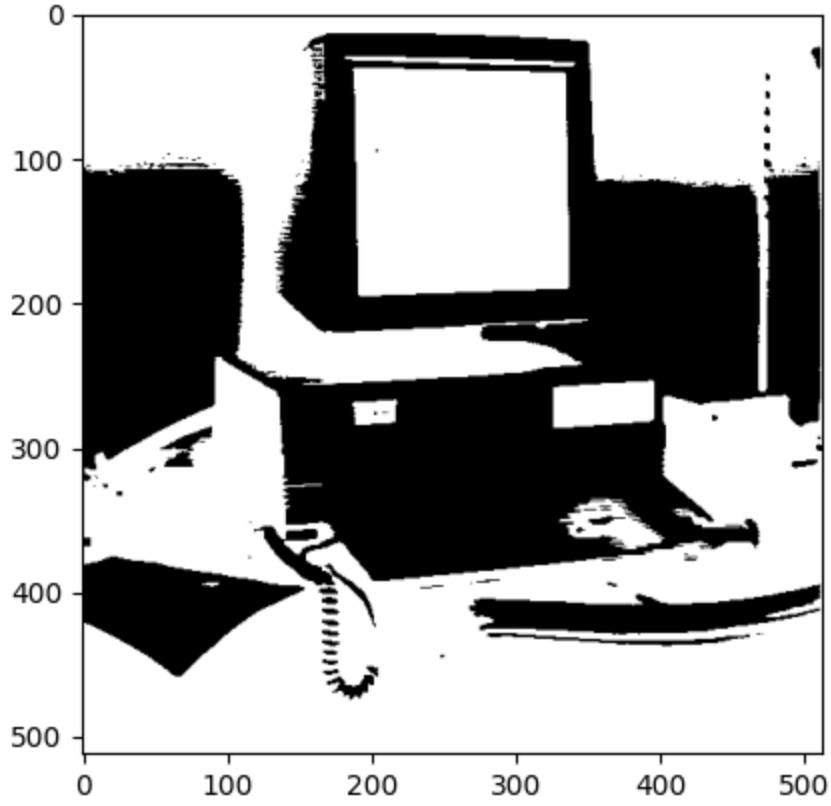
Question 3 Method 1 Images:

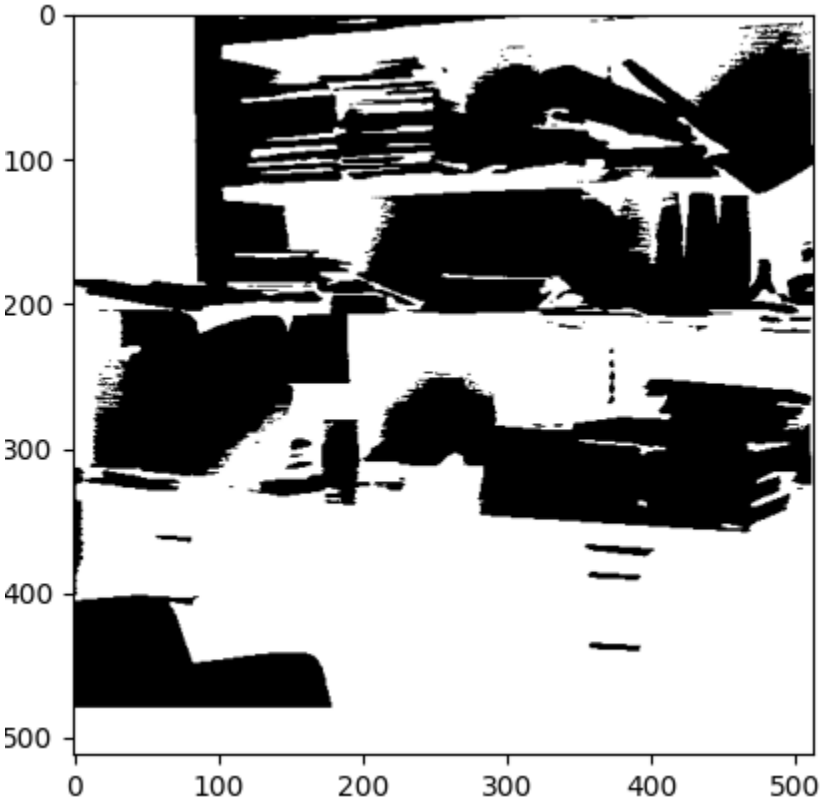


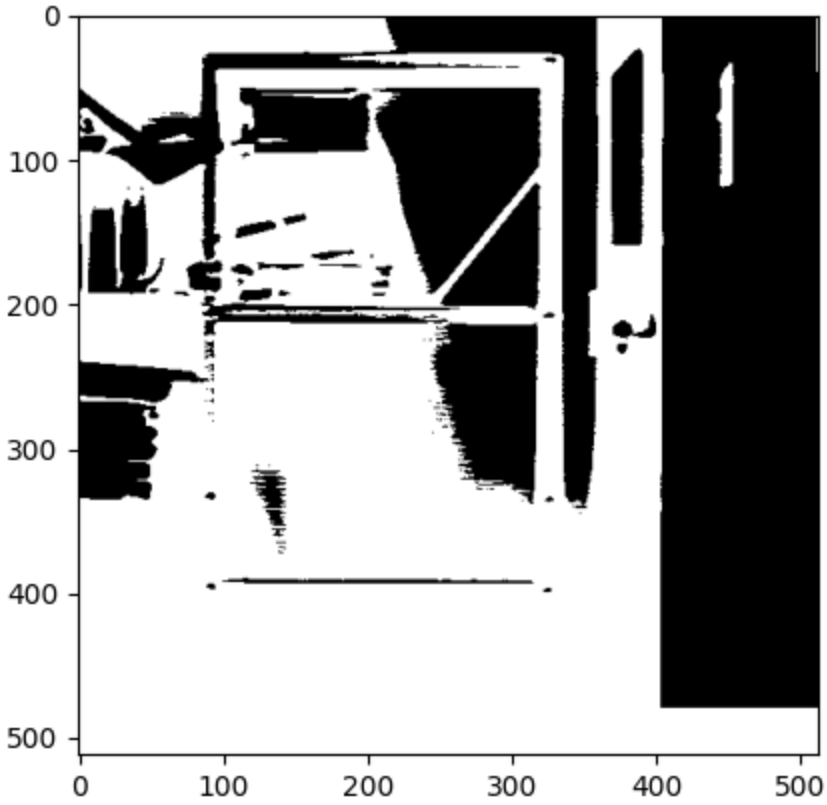




Question 3 Method 2 Images:


0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500



CODE BELOW:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math

def show_image(img,cmap_str='gray_r'):
    norm = plt.Normalize(vmin=0, vmax=1)  # Normalize so that only positive values are highlighted
    plt.imshow(img, cmap=cmap_str,norm=norm)
```

```
    plt.show()

def show_image_2(img,cmap_str='gray_r'):
    norm = plt.Normalize(vmin=0, vmax=255)  # Normalize so that only positive values are highlighted
    plt.imshow(img, cmap=cmap_str,norm=norm)
    plt.show()

def find_peaks(counts, bins):
    peaks = []
    indices = []
    for i in range(len(counts)):
        if i == 0:
            if counts[i] >= counts[i+1]:
                peaks.append(counts[i])
                indices.append(bins[i])
        elif i == len(counts)-1:
            if counts[i] >= counts[i-1]:
                peaks.append(counts[i])
                indices.append(bins[i])
        elif counts[i] >= counts[i+1] and counts[i] >= counts[i-1]:
            peaks.append(counts[i])
            indices.append(bins[i])
    return peaks, indices

def find_valleys(counts, bins):
    valleys = []
    indices = []
    for i in range(len(counts)):
        if i == 0:
            if counts[i] < counts[i+1]:
                valleys.append(counts[i])
                indices.append(bins[i])
        elif i == len(counts)-1:
            if counts[i] <= counts[i-1]:
                valleys.append(counts[i])
                indices.append(bins[i])
        elif counts[i] <= counts[i+1] and counts[i] <= counts[i-1]:
            valleys.append(counts[i])
            indices.append(bins[i])
    return valleys, indices

def find_best_combo(peak_indices, peak_counts, valley_indices, valley_counts):
    best_FOM = -10000

    for index_i, i in enumerate(peak_indices):
        for index_j, j in enumerate(peak_indices):
            # Find peak information
            peak1_index, peak2_index = i, j
            peak1_count, peak2_count = peak_counts[index_i], peak_counts[index_j]
            # Confirm that these peaks are correctly located around each other
            if peak1_index != peak2_index and peak1_index < peak2_index:
                for index_k, k in enumerate(valley_indices):
                    # Find Valley information
```

```
            valley_index, valley_count = k, valley_counts[index_k]
            if valley_index > peak1_index and valley_index < peak2_index and peak1_count > valley_count
and peak2_count > valley_count:

                # 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)

                if current_FOM > best_FOM:

                  best_FOM = current_FOM
                  best_combo = [peak1_index.copy(), valley_index.copy(), peak2_index.copy()]
                  # print(f"Best FOM: {best_FOM:.10f}, Peak 1 Index: {best_combo[0]:.2f}, Valley Index:
{best_combo[1]:.2f}, Peak 2 Index: {best_combo[2]:.2f}")
                  # print(f"Part 1 (a * term1): {term1:.2f}")
                  # print(f"Part 2 (b * term2): {term2:.2f}")
                  # print(f"Part 3 (c * (term3 + 1)): {term3:.2f}")
                  # print(f"Part 4 (d * (term4 + 1)): {term4:.2f}")
                  # print(f"Valley Count: {valley_count:.1f}")
                  # print(f"Peak1 Count: {peak1_count:.1f}")
                  # print(f"Peak2 Count: {peak2_count:.1f}")
                  # print(f"Current FOM: {current_FOM:.10f}")
  return best_combo

 #### Run Code ####

# Parameter Definitions
file_path1 = "img/test1.img"
file_path2 = "img/test2.img"
file_path3 = "img/test3.img"

files = [file_path1,file_path2,file_path3]
width, height = 512, 512
header_size = 512
image_data = []
combos = []

# Read the file(s)
for i in range(len(files)):
  with open(files[i], "rb") as f:
    f.seek(header_size)
    image_data.append(list(f.read()))

# # Reshape into 2D array
# import pdb;pdb.set_trace()
# image_array = [image_data[i].reshape((height, width)) for i in range(len(files))]
```

```
# image_array = [[item for sublist in image_array[i] for item in sublist] for i in range(len(image_array))]

for i in range(len(image_data)):
  image = image_data[i]

  bins_count = 255
  # Plotting a basic histogram
  counts, bins, patches = plt.hist(image, bins=bins_count, color='skyblue', edgecolor='black')

  # plt.show()
  peak_counts, peak_indices = find_peaks(counts, bins)
  valley_counts, valley_indices = find_valleys(counts, bins)

  combo = find_best_combo(peak_indices,peak_counts,valley_indices,valley_counts)

  for val in combo:
    plt.axvline(val, color='red', linestyle='dashed', linewidth=2)
  plt.savefig('img/img_'+str(i)+'.png')
  # plt.show()
  plt.close()

  combos.append(combo)

# print(len(combos))
# for i in combos:
#   print(i)

import copy
image_data_binary = copy.deepcopy(image_data)
for i in range(len(image_data)):
  for j in range(len(image_data[i])):
    if image_data[i][j] >= combos[i][1]:
      image_data_binary[i][j] = 1
    else:
      image_data_binary[i][j] = 0


# Reshape into 2D array
image_data_binary = [np.array(image_data_binary[i]).reshape((height, width)) for i in range(len(files))]
image_data_binary = [[item for sublist in image_data_binary[i] for item in sublist] for i in
range(len(image_data_binary))]

for i in image_data_binary:
  show_image(np.array(i).reshape((height,width)))

# # Create a subplot with the number of images you have (assuming 3 images for this example)
# num_images = len(image_data_binary)
# fig, axes = plt.subplots(1, num_images, figsize=(num_images * 3, 3))  # Adjust size as needed

# # Ensure axes is iterable if there's only one image
# if num_images == 1:
#   axes = [axes]
```

```
# # Loop over each image and display it
# for i, ax in enumerate(axes):
#     ax.imshow(np.array(image_data_binary[i]).reshape(height,width), cmap='gray_r')

# plt.tight_layout()  # Adjust layout to make sure images are spaced nicely
# plt.show()

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# # Load images
# img1 = mpimg.imread("img/img_0.png")
# img2 = mpimg.imread("img/img_1.png")
# img3 = mpimg.imread("img/img_2.png")

# # Create figure and subplots
# fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# # Display first image
# axes[0].imshow(img1)
# axes[0].axis("off")  # Hide axes
# axes[0].set_title("Image 1")

# # Display second image
# axes[1].imshow(img2)
# axes[1].axis("off")
# axes[1].set_title("Image 2")

# # Display third image
# axes[2].imshow(img3)
# axes[2].axis("off")
# axes[2].set_title("Image 3")

# # Adjust layout and show
# plt.tight_layout()
# plt.show()

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)

image_data_binary2 = copy.deepcopy(image_data)
for i in range(len(image_data)):
  for j in range(len(image_data[i])):
    if image_data[i][j] >= adaptive_thresholds[i]:
```

```
        image_data_binary2[i][j] = 1
      else:
        image_data_binary2[i][j] = 0


# # Reshape into 2D array
# image_data_binary2 = [np.array(image_data_binary2[i]).reshape((height, width)) for i in
range(len(files))]
# image_data_binary2 = [[item for sublist in image_data_binary2[i] for item in sublist] for i in
range(len(image_data_binary2))]

for i in image_data_binary2:
  show_image(np.array(i).reshape((height,width)))

# # Create a subplot with the number of images you have (assuming 3 images for this example)
# num_images = len(image_data_binary)
# fig, axes = plt.subplots(1, num_images, figsize=(num_images * 3, 3))  # Adjust size as needed

# # Ensure axes is iterable if there's only one image
# if num_images == 1:
#    axes = [axes]

# # Loop over each image and display it
# for i, ax in enumerate(axes):
#    ax.imshow(np.array(image_data_binary2[i]).reshape(height,width), cmap='gray_r')

plt.tight_layout()  # Adjust layout to make sure images are spaced nicely
plt.show()

def find_2_highest_peaks(peaks_counts,peak_indices):
  total_values = np.sort(peaks_counts)[-2:]
  top_peaks = [[peak_indices[ind],i] for ind,i in enumerate(peaks_counts) if i in total_values]
  return top_peaks

dual_thresholds = []
image_data3 = copy.deepcopy(image_data)
image_data4 = [np.array(image_data3[i]).reshape((height, width)) for i in range(len(files))]


for i in range(len(image_data3)):
  bins_count = 255
  # Plotting a basic histogram
  counts, bins, patches = plt.hist(image_data3[i], bins=bins_count, color='skyblue', edgecolor='black')
  # plt.show()
  peaks_counts, peak_indices = find_peaks(counts, bins)
  import os
  top_peaks = find_2_highest_peaks(peaks_counts,peak_indices)
  t1 = top_peaks[0][0]
  t2 = top_peaks[1][0]
  import time
  vals = [t1,t2]
  for val in vals:
    plt.axvline(val, color='red', linestyle='dashed', linewidth=2)
```

```
plt.show()
# Make the program wait for 1 second
time.sleep(1)
plt.close()

img = copy.deepcopy(image_data4[i])
r1 = []
r2 = []
r3 = []
for ind1, row in enumerate(img):
  for ind2, col in enumerate(row):
    if img[ind1][ind2] <= t1:
      img[ind1][ind2] = 0
      r1.append([ind1,ind2])
    elif img[ind1][ind2] >= t2:
      img[ind1][ind2] = 1
      r3.append([ind1,ind2])
    else:
      img[ind1][ind2] = -1
      r2.append([ind1,ind2])

count = 0


import matplotlib.colors as mcolors
# Define a colormap for the 3 values
cmap = mcolors.ListedColormap(["green", "black", "white"])
bounds = [-1.5, -0.5, 0.5, 1.5]  # Define the bounds for each value
norm = mcolors.BoundaryNorm(bounds, cmap.N)  # Normalize values to colormap

# Plot the image
plt.imshow(img, cmap=cmap, norm=norm)
plt.colorbar(ticks=[-1, 0, 1], label="Pixel Value")  # Add colorbar for reference
plt.axis('off')  # Hide axis for better visualization
plt.title("Custom Image with Values -1, 0, 1")
plt.show()
# loop through all points in region 1 (foreground)
while count < len(r3):

  point = r3[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] = 1 # Set this neighbor to region 1 (foreground)
      r3.append([ind1,ind2-1]) # Add this coordinate to region 1's list

  # Top
  if ind1 != 0:
    if img[ind1-1][ind2] == -1:
```

```
            img[ind1-1][ind2] = 1
            r3.append([ind1-1,ind2])
        # Right
        if ind2 != 511:
          if img[ind1][ind2+1] == -1:
            img[ind1][ind2+1] = 1
            r3.append([ind1,ind2+1])

        # Right
        if ind1 != 511:
          if img[ind1+1][ind2] == -1:
            img[ind1+1][ind2] = 1
            r3.append([ind1+1,ind2])
    for ind1, row in enumerate(img):
      for ind2, col in enumerate(row):
        if img[ind1][ind2] == -1:
          img[ind1][ind2] = 0

    image_data4[i] = img

for i in image_data4:
    show_image(np.array(i))

# Create a subplot with the number of images you have (assuming 3 images for this example)
num_images = len(image_data4)
fig, axes = plt.subplots(1, num_images, figsize=(num_images * 3, 3))  # Adjust size as needed

# Ensure axes is iterable if there's only one image
if num_images == 1:
    axes = [axes]

# Loop over each image and display it
for i, ax in enumerate(axes):
    ax.imshow(np.array(image_data4[i]).reshape(height,width), cmap='gray_r')

plt.tight_layout()  # Adjust layout to make sure images are spaced nicely
plt.show()

# Create a subplot with the number of images you have (assuming 3 images for this example)
num_images = len(image_data4)
fig, axes = plt.subplots(1, num_images, figsize=(num_images * 3, 3))  # Adjust size as needed

# Ensure axes is iterable if there's only one image
if num_images == 1:
    axes = [axes]

# Loop over each image and display it
for i, ax in enumerate(axes):
    ax.hist(image_data3[i], bins=bins_count, color='skyblue', edgecolor='black')

plt.tight_layout()  # Adjust layout to make sure images are spaced nicely
plt.show()
```

```
image_data5 = copy.deepcopy(image_data)
image_data6 = [np.array(image_data5[i]).reshape((height, width)) for i in range(len(files))]

for i in range(len(image_data5)):
  bins_count = 255
  # Plotting a basic histogram
  counts, bins, patches = plt.hist(image_data5[i], bins=bins_count, color='skyblue', edgecolor='black')
  # plt.show()
  peaks_counts, peak_indices = find_peaks(counts, bins)
  import os
  top_peaks = find_2_highest_peaks(peaks_counts,peak_indices)
  t1 = np.percentile(image_data5[i], 40)
  t2 = np.percentile(image_data5[i], 60)
  import time

  vals = [t1,t2]
  for val in vals:
    plt.axvline(val, color='red', linestyle='dashed', linewidth=2)
  plt.show()
  # Make the program wait for 1 second
  time.sleep(1)
  plt.close()

  img2 = copy.deepcopy(image_data6[i])
  r1 = []
  r2 = []
  r3 = []
  for ind1, row in enumerate(img2):
    for ind2, col in enumerate(row):
      if img2[ind1][ind2] <= t1:
        img2[ind1][ind2] = 0
        r1.append([ind1,ind2])
      elif img2[ind1][ind2] >= t2:
        img2[ind1][ind2] = 1
        r3.append([ind1,ind2])
      else:
        img2[ind1][ind2] = -1
        r2.append([ind1,ind2])

  count = 0

  # 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 img2[ind1][ind2-1] == -1: # Check if left neighbor is in region 2 (unsure)
          img2[ind1][ind2-1] = 0 # Set this neighbor to region 1 (foreground)
          r1.append([ind1,ind2-1]) # Add this coordinate to region 1's list

      # Top
      if ind1 != 0:
        if img2[ind1-1][ind2] == -1:
          img2[ind1-1][ind2] = 0
          r1.append([ind1-1,ind2])
      # Right
      if ind2 != 511:
        if img2[ind1][ind2+1] == -1:
          img2[ind1][ind2+1] = 0
          r1.append([ind1,ind2+1])

      # Right
      if ind1 != 511:
        if img2[ind1+1][ind2] == -1:
          img2[ind1+1][ind2] = 0
          r1.append([ind1+1,ind2])

    for ind1, row in enumerate(img2):
      for ind2, col in enumerate(row):
        if img2[ind1][ind2] == -1:
          img2[ind1][ind2] = 1

    image_data6[i] = img2

for i in image_data6:
    show_image(np.array(i))

# Create a subplot with the number of images you have (assuming 3 images for this example)
num_images = len(image_data6)
fig, axes = plt.subplots(1, num_images, figsize=(num_images * 3, 3))  # Adjust size as needed

# Ensure axes is iterable if there's only one image
if num_images == 1:
    axes = [axes]

# Loop over each image and display it
for i, ax in enumerate(axes):
    ax.imshow(np.array(image_data6[i]).reshape(height,width), cmap='gray_r')

plt.tight_layout()  # Adjust layout to make sure images are spaced nicely
plt.show()
```