Assignment 2

CP467: Image Processing & Recognition
Professor: Dr. Zia Ud Din
Due Date: Friday October 20, 2023. 11:59 p.m.

Chandler Mayberry 190688910

Table of Contents

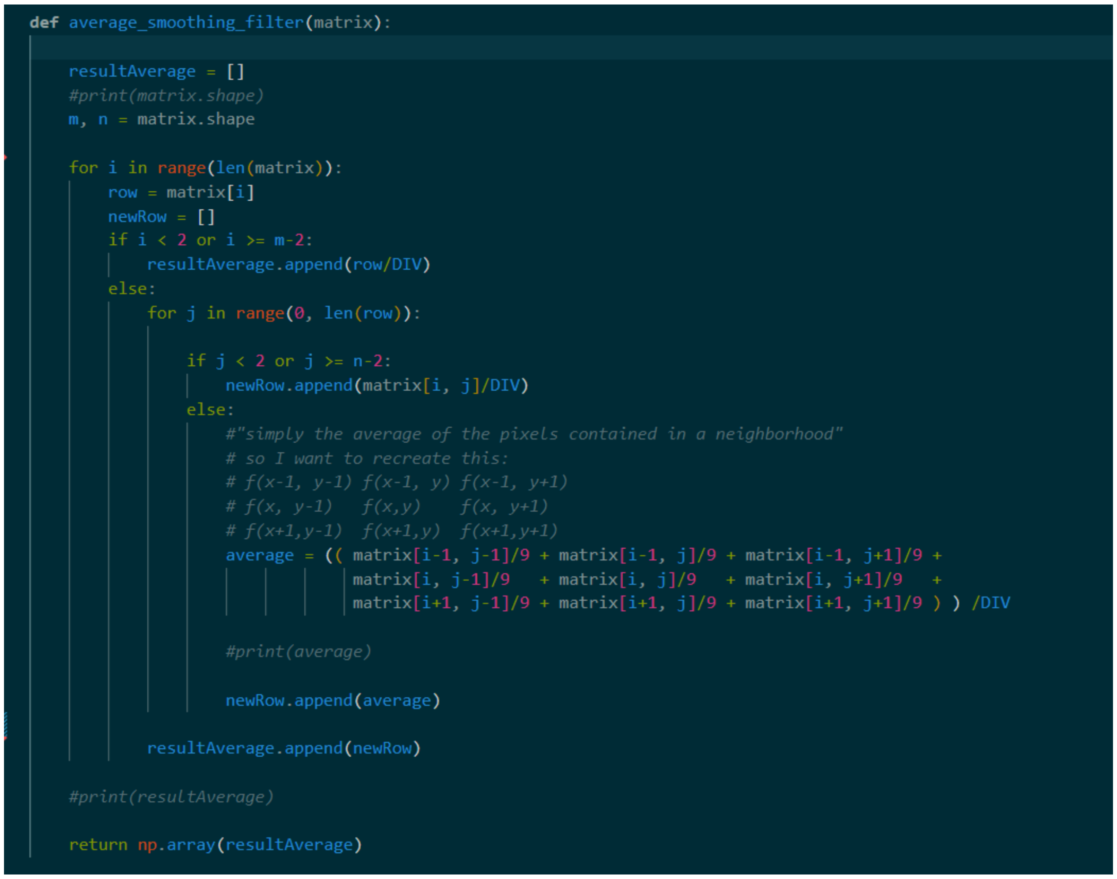
Introduction

All the code for this assignment may be run using the a02_190688910.py file in the source code folder. Running this file will perform the operations and display the resulting images to the user using python OpenCV.imshow() method.

Task 1

Averaging Smoothing Filter (Filter size: 3*3)

An averaging smoothing filter is a simple image blurring technique to reduce the sharpness and bridge the smaller details of an image. This operation is rather simple, the new value for each pixel in the image is the average of its surrounding neighborhood (adjacent pixel values). In the case of the operation implemented below, the filter size specified was 3*3. See the code below:

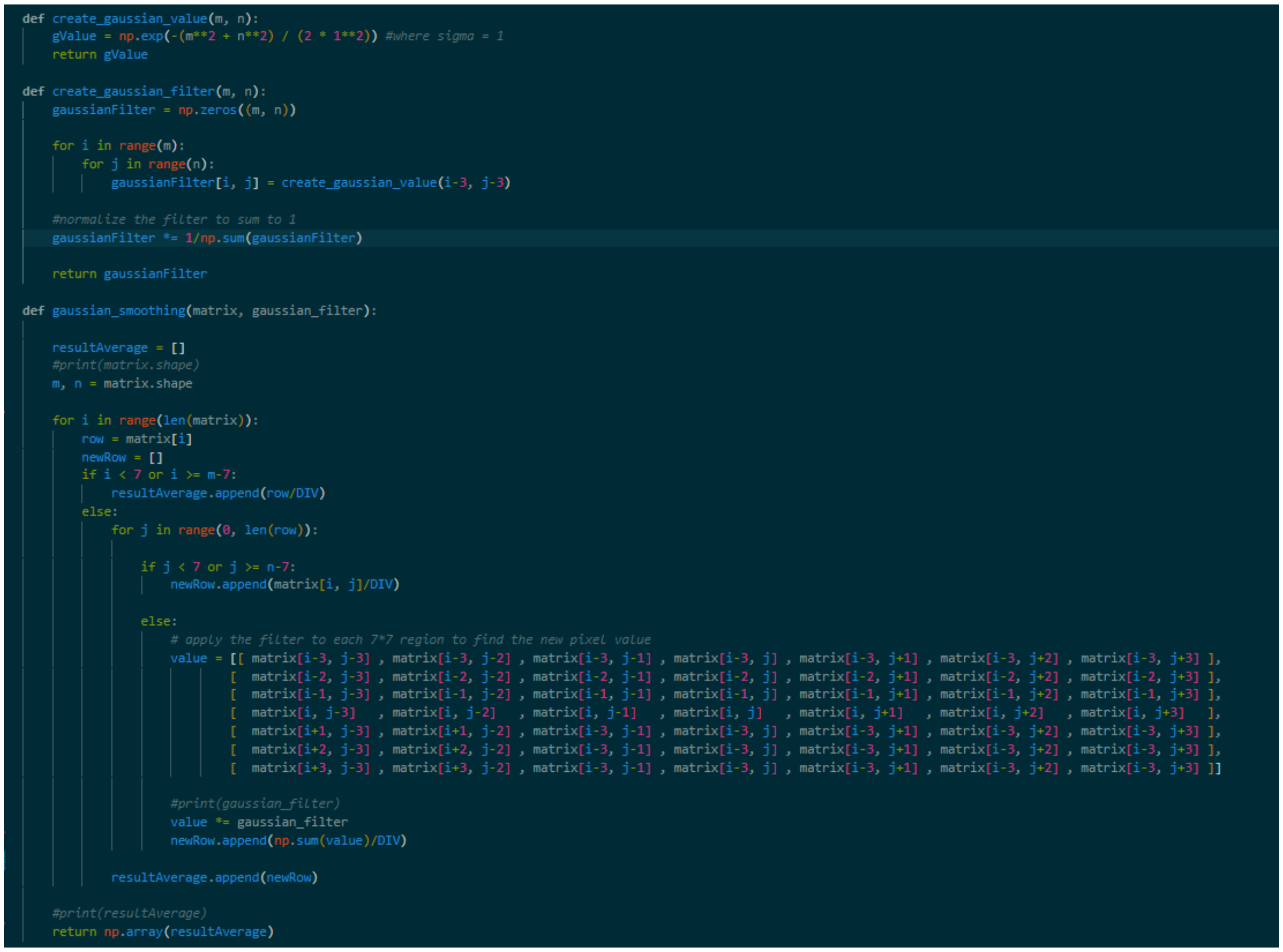


```python
def average_smoothing_filter(matrix):

    resultAverage = []
    #print(matrix.shape)
    m, n = matrix.shape

    for i in range(len(matrix)):
        row = matrix[i]
        newRow = []
        if i < 2 or i >= m-2:
            resultAverage.append(row/DIV)
        else:
            for j in range(0, len(row)):

                if j < 2 or j >= n-2:
                    newRow.append(matrix[i, j]/DIV)
                else:
                    #"simply the average of the pixels contained in a neighborhood"
                    # so I want to recreate this:
                    # f(x-1, y-1) f(x-1, y) f(x-1, y+1)
                    # f(x, y-1)   f(x,y)    f(x, y+1)
                    # f(x+1,y-1)  f(x+1,y)  f(x+1,y+1)
                    average = (( matrix[i-1, j-1]/9 + matrix[i-1, j]/9 + matrix[i-1, j+1]/9 +
                                 matrix[i, j-1]/9   + matrix[i, j]/9   + matrix[i, j+1]/9   +
                                 matrix[i+1, j-1]/9 + matrix[i+1, j]/9 + matrix[i+1, j+1]/9 ) ) /DIV

                    #print(average)

                    newRow.append(average)

            resultAverage.append(newRow)

    #print(resultAverage)

    return np.array(resultAverage)
```

Figure 1: Averaging Smoothing Filter Implementation

Gaussian Smoothing Filter (Filter size: 7*7, sigma=1, mean=0)

A Gaussian smoothing filter is another smoothing technique that reduces the noise within an image and produces a blurred image. This was introduced as a preprocess to edge detection in an image as it reduces the amount of noise (large value differences in neighboring pixels). The logic is relatively straight forward, for a 7*7 region with sigma value one and a mean of zero a normalized height map matrix is created as seen on slide 23-24, Lecture 5. This height map essentially gradients the values from lighter to darker from the center to edges of each pixel matrix (size 7*7). This height map is then applied to all the pixels and adjacent neighbors in the matrix, causing smoother transitions between gray values in the resulting image. See the code below:

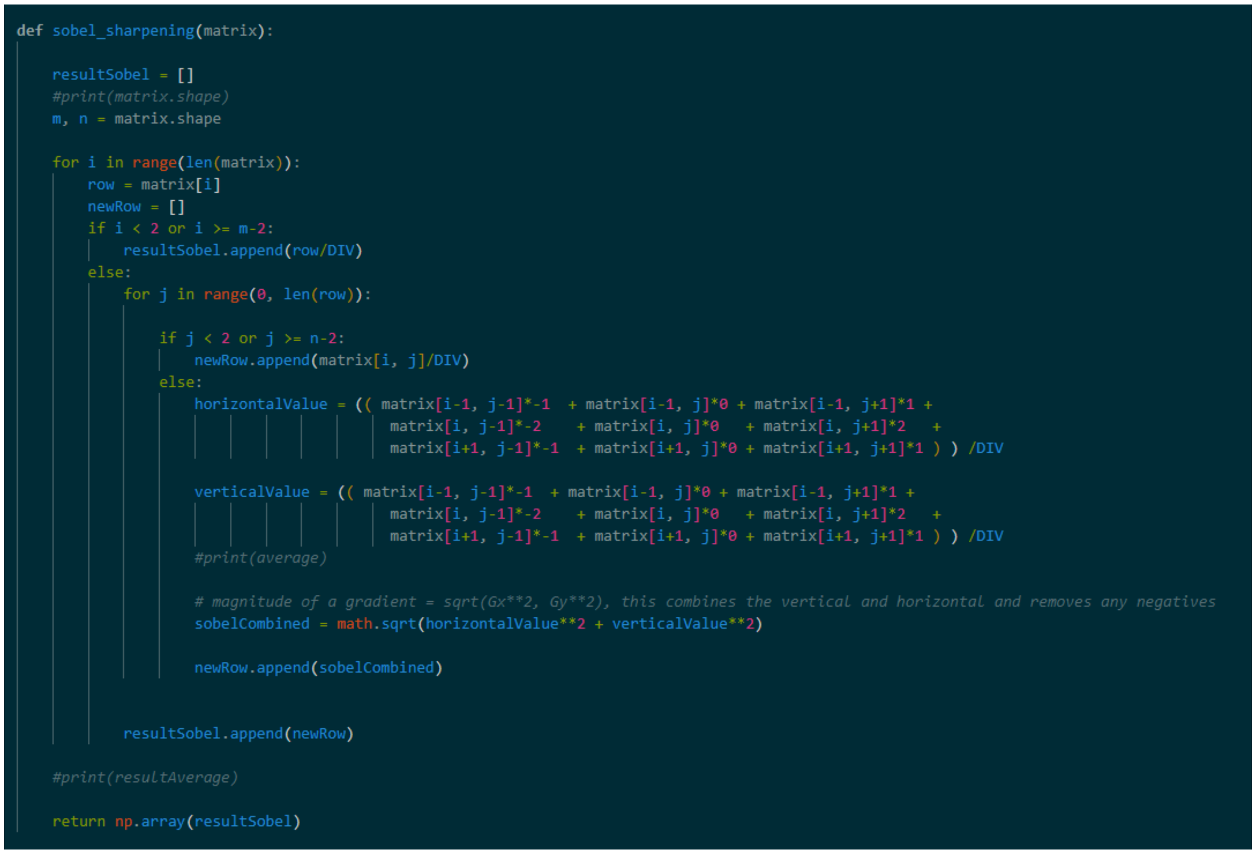


```
def create_gaussian_value(m, n):
    gValue = np.exp(-(m**2 + n**2) / (2 * 1**2)) #where sigma = 1
    return gValue

def create_gaussian_filter(m, n):
    gaussianFilter = np.zeros((m, n))

    for i in range(m):
        for j in range(n):
            gaussianFilter[i, j] = create_gaussian_value(i-3, j-3)

    #normalize the filter to sum to 1
    gaussianFilter *= 1/np.sum(gaussianFilter)

    return gaussianFilter

def gaussian_smoothing(matrix, gaussian_filter):

    resultAverage = []
    #print(matrix.shape)
    m, n = matrix.shape

    for i in range(len(matrix)):
        row = matrix[i]
        newRow = []
        if i < 7 or i >= m-7:
            resultAverage.append(row/DIV)
        else:
            for j in range(0, len(row)):

                if j < 7 or j >= n-7:
                    newRow.append(matrix[i, j]/DIV)

                else:
                    # apply the filter to each 7*7 region to find the new pixel value
                    value = [[ matrix[i-3, j-3] , matrix[i-3, j-2] , matrix[i-3, j-1] , matrix[i-3, j] , matrix[i-3, j+1] , matrix[i-3, j+2] , matrix[i-3, j+3] ],
                            [  matrix[i-2, j-3] , matrix[i-2, j-2] , matrix[i-2, j-1] , matrix[i-2, j] , matrix[i-2, j+1] , matrix[i-2, j+2] , matrix[i-2, j+3] ],
                            [  matrix[i-1, j-3] , matrix[i-1, j-2] , matrix[i-1, j-1] , matrix[i-1, j] , matrix[i-1, j+1] , matrix[i-1, j+2] , matrix[i-1, j+3] ],
                            [  matrix[i, j-3]   , matrix[i, j-2]   , matrix[i, j-1]   , matrix[i, j]   , matrix[i, j+1]   , matrix[i, j+2]   , matrix[i, j+3]   ],
                            [  matrix[i+1, j-3] , matrix[i+1, j-2] , matrix[i-3, j-1] , matrix[i-3, j] , matrix[i-3, j+1] , matrix[i-3, j+2] , matrix[i-3, j+3] ],
                            [  matrix[i+2, j-3] , matrix[i+2, j-2] , matrix[i-3, j-1] , matrix[i-3, j] , matrix[i-3, j+1] , matrix[i-3, j+2] , matrix[i-3, j+3] ],
                            [  matrix[i+3, j-3] , matrix[i+3, j-2] , matrix[i-3, j-1] , matrix[i-3, j] , matrix[i-3, j+1] , matrix[i-3, j+2] , matrix[i-3, j+3] ]]

                    #print(gaussian_filter)
                    value *= gaussian_filter
                    newRow.append(np.sum(value)/DIV)

            resultAverage.append(newRow)

    #print(resultAverage)
    return np.array(resultAverage)
```

Figure 2: Gaussian Smoothing Filter Implementation

Sobel Sharpening Filter (Filter size: 3*3, both horizontal and vertical)

The general purpose of the Sobel sharpening filter is to bring out finer detail within the image. This operation is simple, the Sobel filter is predefined both vertically and horizontally in a 3*3 matrix. Both the vertical and horizontal matrices are applied at each pixel and surrounding neighbors independently. In this operation it is required that both vertical and horizontal Sobel filtering is used, this combination is done by taking the magnitude of each value across both arrays. See the code below:



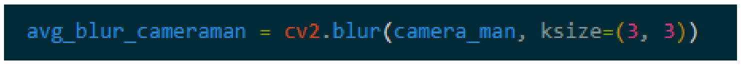
```python
def sobel_sharpening(matrix):

    resultSobel = []
    #print(matrix.shape)
    m, n = matrix.shape

    for i in range(len(matrix)):
        row = matrix[i]
        newRow = []
        if i < 2 or i >= m-2:
            resultSobel.append(row/DIV)
        else:
            for j in range(0, len(row)):

                if j < 2 or j >= n-2:
                    newRow.append(matrix[i, j]/DIV)
                else:
                    horizontalValue = (( matrix[i-1, j-1]*-1  + matrix[i-1, j]*0 + matrix[i-1, j+1]*1 +
                                         matrix[i, j-1]*-2    + matrix[i, j]*0   + matrix[i, j+1]*2   +
                                         matrix[i+1, j-1]*-1  + matrix[i+1, j]*0 + matrix[i+1, j+1]*1 ) ) /DIV

                    verticalValue = (( matrix[i-1, j-1]*-1  + matrix[i-1, j]*0 + matrix[i-1, j+1]*1 +
                                         matrix[i, j-1]*-2    + matrix[i, j]*0   + matrix[i, j+1]*2   +
                                         matrix[i+1, j-1]*-1  + matrix[i+1, j]*0 + matrix[i+1, j+1]*1 ) ) /DIV
                    #print(average)

                    # magnitude of a gradient = sqrt(Gx**2, Gy**2), this combines the vertical and horizontal and removes any negatives
                    sobelCombined = math.sqrt(horizontalValue**2 + verticalValue**2)

                    newRow.append(sobelCombined)


            resultSobel.append(newRow)

    #print(resultAverage)

    return np.array(resultSobel)
```

Figure 3: Sobel Sharpening Filter Implementation

Task 2

Averaging Smoothing Filter Built-in (Same as Task 1)

Averaging smoothing filter using built-in function. See the code below:

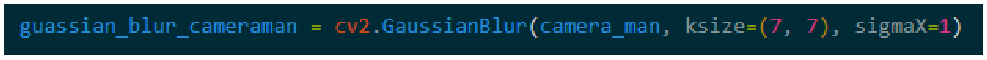


```python
avg_blur_cameraman = cv2.blur(camera_man, ksize=(3, 3))
```

Figure 4: Averaging Smoothing Filter Built-in

Gaussian Smoothing Filter Built-in (Same as Task 1)

Gaussian smoothing filter using built-in function. See the code below:

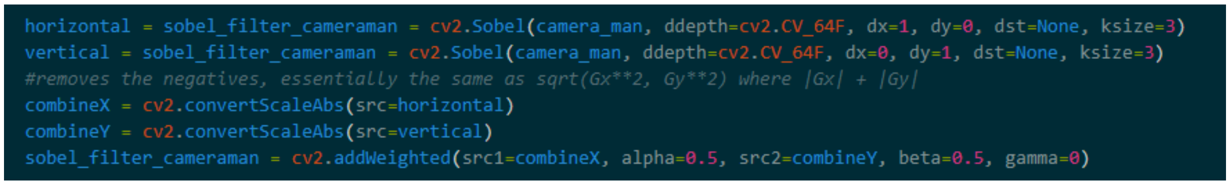


```
guassian_blur_cameraman = cv2.GaussianBlur(camera_man, ksize=(7, 7), sigmaX=1)
```

Figure 5: Gaussian Smoothing Filter Built-in

Sobel Sharpening Filter Built-in (Same as Task 1)

Sobel sharpening filter using built-in functions. See the code below:



```
horizontal = sobel_filter_cameraman = cv2.Sobel(camera_man, ddepth=cv2.CV_64F, dx=1, dy=0, dst=None, ksize=3)
vertical = sobel_filter_cameraman = cv2.Sobel(camera_man, ddepth=cv2.CV_64F, dx=0, dy=1, dst=None, ksize=3)
#removes the negatives, essentially the same as sqrt(Gx**2, Gy**2) where |Gx| + |Gy|
combineX = cv2.convertScaleAbs(src=horizontal)
combineY = cv2.convertScaleAbs(src=vertical)
sobel_filter_cameraman = cv2.addWeighted(src1=combineX, alpha=0.5, src2=combineY, beta=0.5, gamma=0)
```

Figure 6: Sobel Sharpening Filter Built-in

Comparison of Task 1 and Task 2 Results

Comparing the results of the built-in functions to those that I implemented, it is extremely clear that the built-in operations are significantly faster than the scratch implementation of the algorithms. Either than that, it seems the images produced for Gaussian and Averaging smoothing were nearly identical to the built-ins from visual inspection. The only difference I see are the results of the Sobel sharpening filter, where the built-in does a significantly better job than that of the scratch implementation. This is perhaps due to my combination of values after calculating two separate matrix values for horizontal and vertical, where the built-in may do something cleverer to combine the images equally in cv2.addWeighted. Please see results below and in the folder included to see enlarged results:

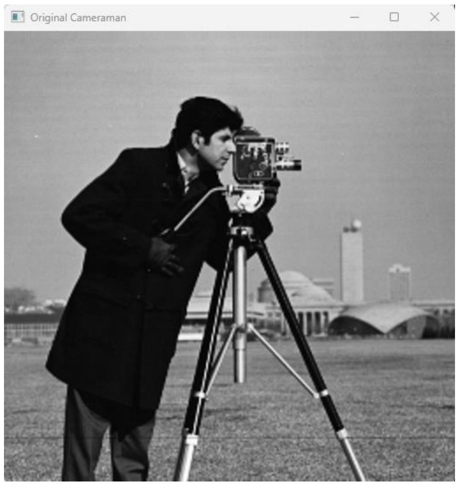


Figure 7: Original Cameraman

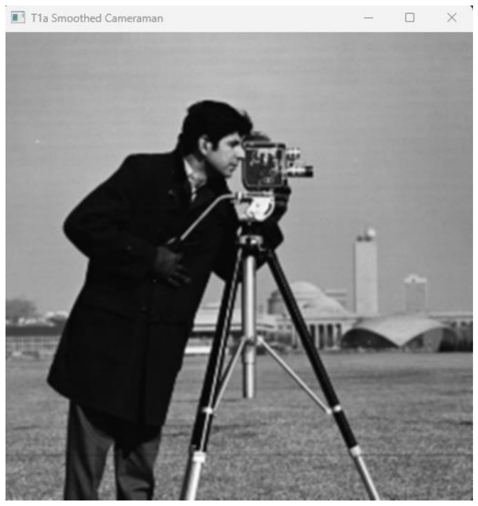


Figure 8: Averaging Smoothing Filter Result

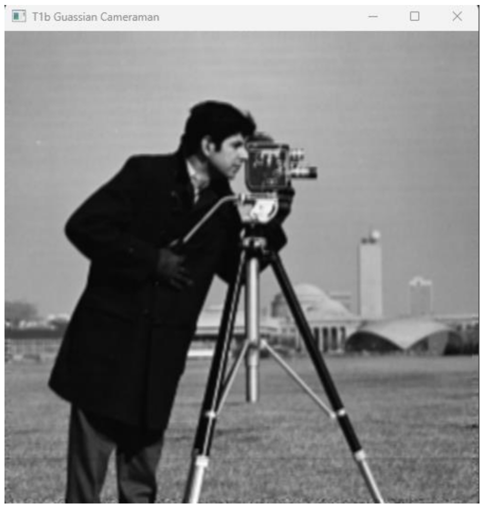


Figure 9: Gaussian Smoothing Filter Result

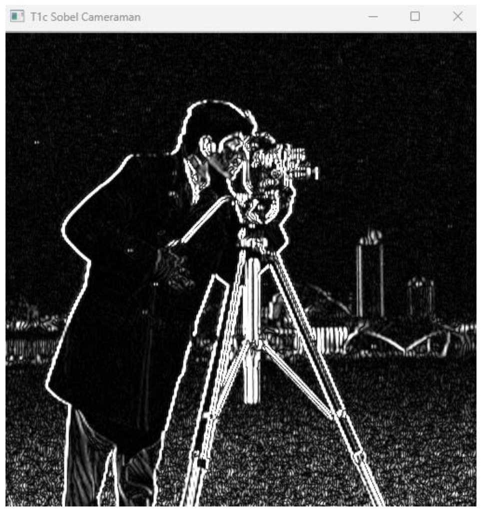


Figure 10: Sobel Sharpening Filter Result

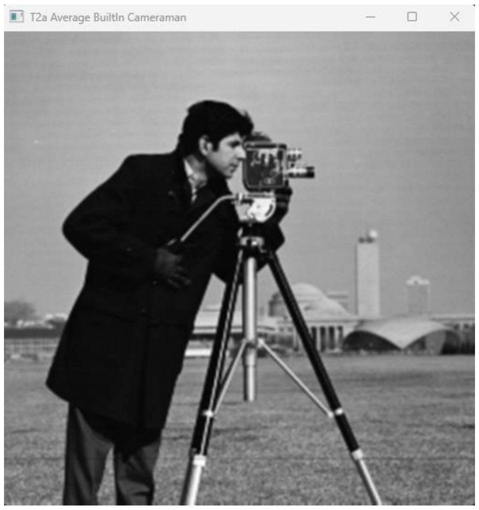


Figure 11: Averaging Smoothing Built-in

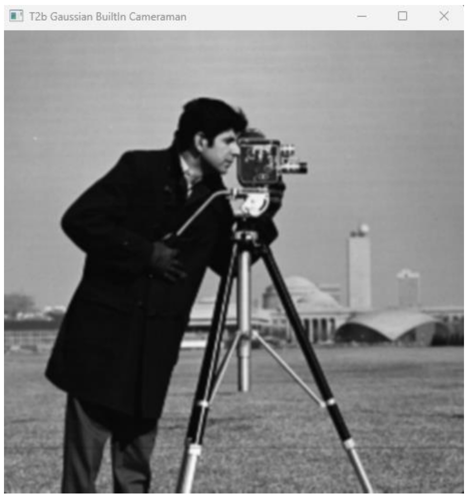


Figure 12: Gaussian Smoothing Built-in

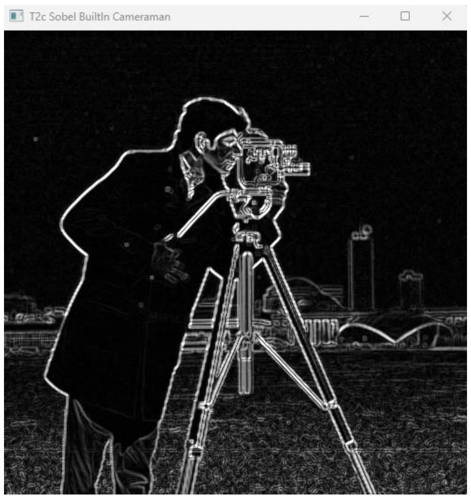
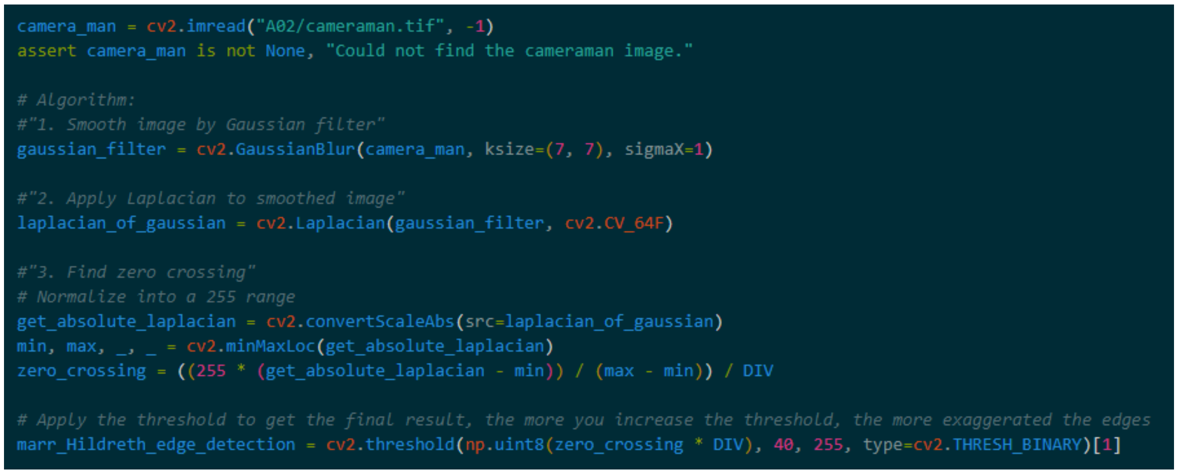


Figure 13: Sobel Sharpening Built-in

Task 3

Marr-Hildreth Edge Detector

Marr-Hildreth edge detector is an edge detection algorithm that results in a black and white image, where the white values indicate the edges of objects in an image. Marr-Hildreth was implemented with built-ins as follows from the slides; first smooth the image using a gaussian filter, then apply Laplacian and combine. Then normalize into 255 range, get the zero crossings, and apply a threshold within bounds 0-255 to get the result. The threshold I finalized was 40-255 as it produced the best edge detection result. See the code and result below:



```python
camera_man = cv2.imread("A02/cameraman.tif", -1)
assert camera_man is not None, "Could not find the cameraman image."

# Algorithm:
#"1. Smooth image by Gaussian filter"
gaussian_filter = cv2.GaussianBlur(camera_man, ksize=(7, 7), sigmaX=1)

#"2. Apply Laplacian to smoothed image"
laplacian_of_gaussian = cv2.Laplacian(gaussian_filter, cv2.CV_64F)

#"3. Find zero crossing"
# Normalize into a 255 range
get_absolute_laplacian = cv2.convertScaleAbs(src=laplacian_of_gaussian)
min, max, _, _ = cv2.minMaxLoc(get_absolute_laplacian)
zero_crossing = ((255 * (get_absolute_laplacian - min)) / (max - min)) / DIV

# Apply the threshold to get the final result, the more you increase the threshold, the more exaggerated the edges
marr_Hildreth_edge_detection = cv2.threshold(np.uint8(zero_crossing * DIV), 40, 255, type=cv2.THRESH_BINARY)[1]
```

Figure 14: Marr-Hildreth Edge Detector Built-ins

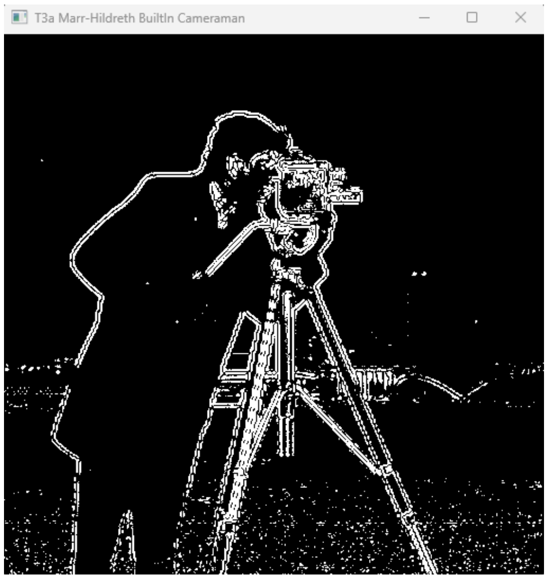
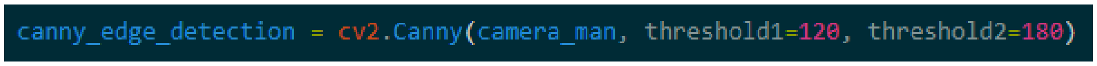


Figure 15: Marr-Hildreth Result

Canny Edge Detector

Canny edge detector is also an edge detection algorithm that produces similar results to Marr-Hildreth. This is the most universally used edge detector in image processing as well as computer vision as stated in Lecture 5. The threshold I found to be the best was between 120-180, where the camera man and his camera are outlined without too much detail. See the code and result below:



```
canny_edge_detection = cv2.Canny(camera_man, threshold1=120, threshold2=180)
```

Figure 16: Canny Built-in

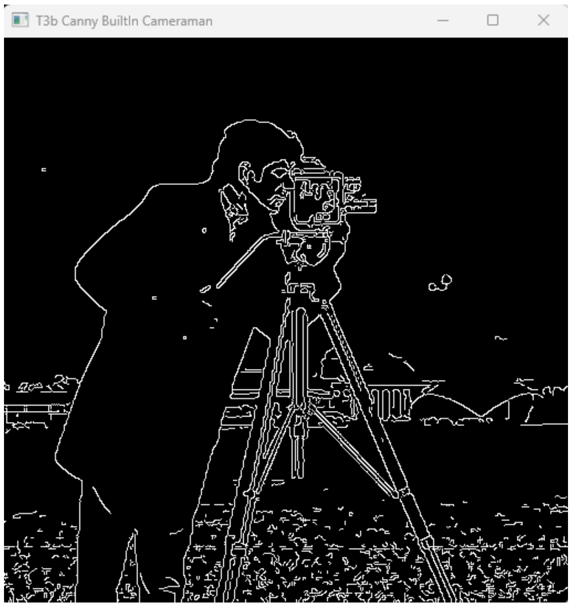


Figure 17: Canny Result

Compare Results of Edge Detectors

As seen above, the results for the Marr-Hildreth implementation using built-ins are far worse than the result of the Canny built-in. This is perhaps due to an incorrect implementation of Marr-Hildreth or truly representing the significant improvement using Canny edge detection is. I found that no matter the threshold values used in Marr-Hildreth the image would result with a lot of noise in terms of random white pixels and poor localization of edges (keeping straight lines). The higher the threshold resulted in more exaggerated edges and less detail in the image for Marr-Hildreth. In comparison, Canny results in a nice edge map where the edges are well defined with a lot less noise regardless of a specific threshold.

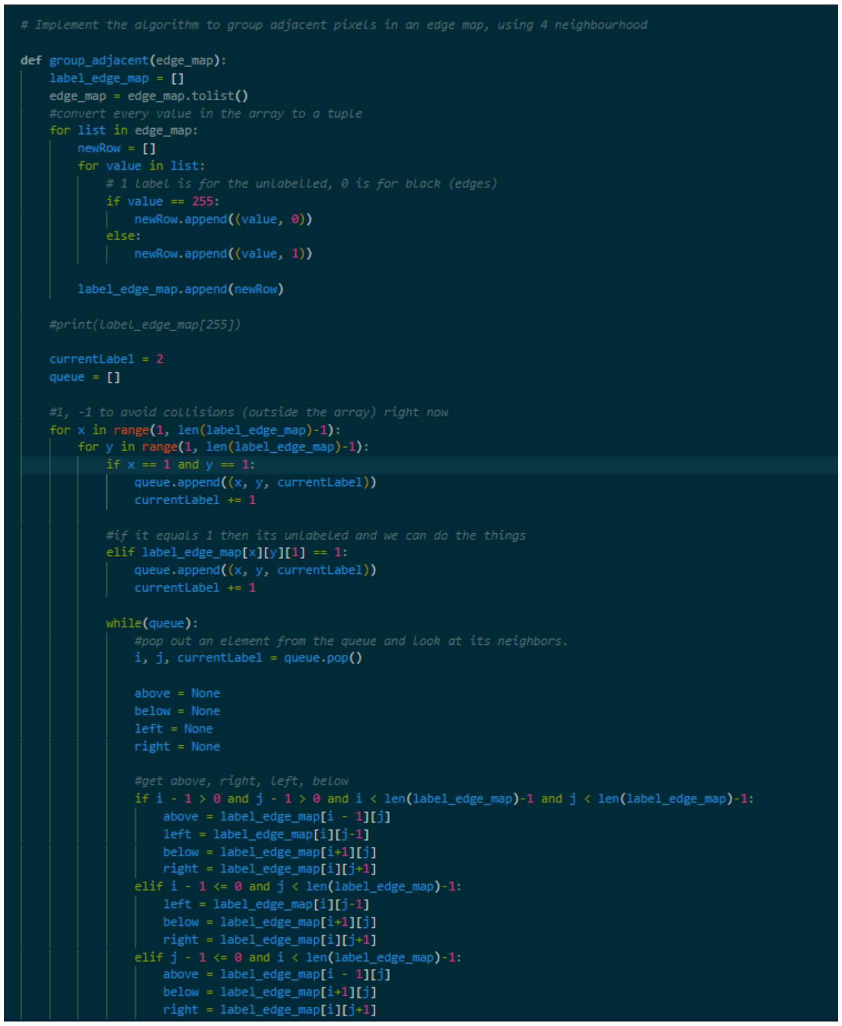
Task 4

Group Adjacent Pixels

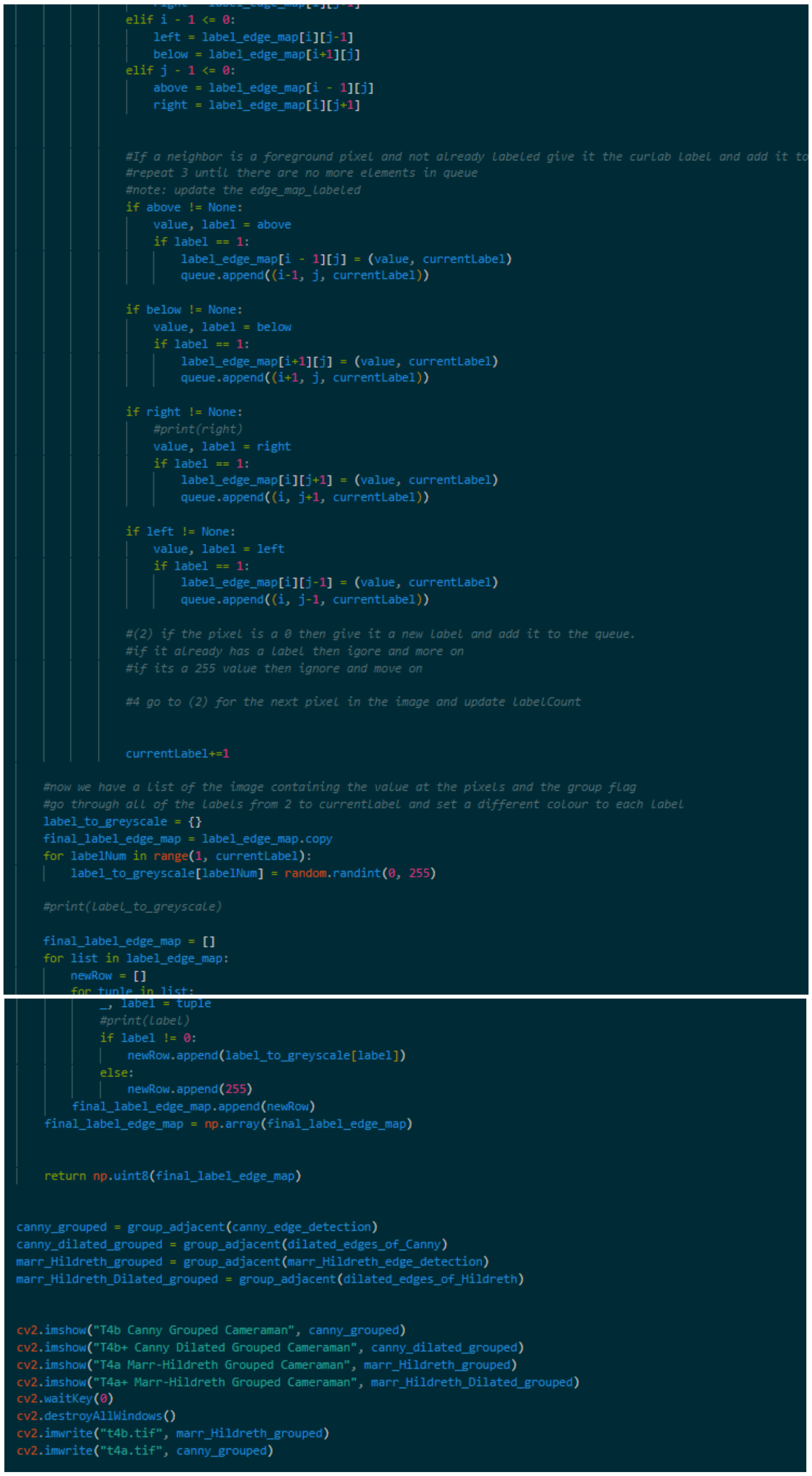
Group adjacent pixels is a simple concept that involves filling in color and/or labeling pixels based on their grouping between edges. For example, think of the bucket tool within Microsoft paint on each group of pixels in an image.

Note:

I noticed when I ran the function it seemed that the edge maps produced by both my Canny and Marr-Hildreth implementations ended up with gaps between edges causing large grouping of pixels between objects in the image (such as the cameraman and the foreground). Therefore, I wanted to increase the size of the edges to better show the grouping and applied a dilation function to the final result of Canny and Marr-Hildreth. See the code and results below:



```python
# Implement the algorithm to group adjacent pixels in an edge map, using 4 neighbourhood

def group_adjacent(edge_map):
    label_edge_map = []
    edge_map = edge_map.tolist()
    #convert every value in the array to a tuple
    for list in edge_map:
        newRow = []
        for value in list:
            # 1 label is for the unlabelled, 0 is for black (edges)
            if value == 255:
                newRow.append((value, 0))
            else:
                newRow.append((value, 1))

        label_edge_map.append(newRow)

    #print(label_edge_map[255])

    currentLabel = 2
    queue = []

    #1, -1 to avoid collisions (outside the array) right now
    for x in range(1, len(label_edge_map)-1):
        for y in range(1, len(label_edge_map)-1):
            if x == 1 and y == 1:
                queue.append((x, y, currentLabel))
                currentLabel += 1

            #if it equals 1 then its unlabeled and we can do the things
            elif label_edge_map[x][y][1] == 1:
                queue.append((x, y, currentLabel))
                currentLabel += 1

            while(queue):
                #pop out an element from the queue and look at its neighbors.
                i, j, currentLabel = queue.pop()

                above = None
                below = None
                left = None
                right = None

                #get above, right, left, below
                if i - 1 > 0 and j - 1 > 0 and i < len(label_edge_map)-1 and j < len(label_edge_map)-1:
                    above = label_edge_map[i - 1][j]
                    left = label_edge_map[i][j-1]
                    below = label_edge_map[i+1][j]
                    right = label_edge_map[i][j+1]
                elif i - 1 <= 0 and j < len(label_edge_map)-1:
                    left = label_edge_map[i][j-1]
                    below = label_edge_map[i+1][j]
                    right = label_edge_map[i][j+1]
                elif j - 1 <= 0 and i < len(label_edge_map)-1:
                    above = label_edge_map[i - 1][j]
                    below = label_edge_map[i+1][j]
                    right = label_edge_map[i][j+1]
```



```
                elif i - 1 <= 0:
                    left = label_edge_map[i][j-1]
                    below = label_edge_map[i+1][j]
                elif j - 1 <= 0:
                    above = label_edge_map[i - 1][j]
                    right = label_edge_map[i][j+1]


                #If a neighbor is a foreground pixel and not already labeled give it the curlab label and add it to
                #repeat 3 until there are no more elements in queue
                #note: update the edge_map_labeled
                if above != None:
                    value, label = above
                    if label == 1:
                        label_edge_map[i - 1][j] = (value, currentLabel)
                        queue.append((i-1, j, currentLabel))

                if below != None:
                    value, label = below
                    if label == 1:
                        label_edge_map[i+1][j] = (value, currentLabel)
                        queue.append((i+1, j, currentLabel))

                if right != None:
                    #print(right)
                    value, label = right
                    if label == 1:
                        label_edge_map[i][j+1] = (value, currentLabel)
                        queue.append((i, j+1, currentLabel))

                if left != None:
                    value, label = left
                    if label == 1:
                        label_edge_map[i][j-1] = (value, currentLabel)
                        queue.append((i, j-1, currentLabel))

                #(2) if the pixel is a 0 then give it a new label and add it to the queue.
                #if it already has a label then igore and more on
                #if its a 255 value then ignore and move on

                #4 go to (2) for the next pixel in the image and update labelCount


                currentLabel+=1

    #now we have a list of the image containing the value at the pixels and the group flag
    #go through all of the labels from 2 to currentLabel and set a different colour to each label
    label_to_greyscale = {}
    final_label_edge_map = label_edge_map.copy
    for labelNum in range(1, currentLabel):
        label_to_greyscale[labelNum] = random.randint(0, 255)

    #print(label_to_greyscale)

    final_label_edge_map = []
    for list in label_edge_map:
        newRow = []
        for tuple in list:
            _, label = tuple
            #print(label)
            if label != 0:
                newRow.append(label_to_greyscale[label])
            else:
                newRow.append(255)
        final_label_edge_map.append(newRow)
    final_label_edge_map = np.array(final_label_edge_map)


    return np.uint8(final_label_edge_map)


canny_grouped = group_adjacent(canny_edge_detection)
canny_dilated_grouped = group_adjacent(dilated_edges_of_Canny)
marr_Hildreth_grouped = group_adjacent(marr_Hildreth_edge_detection)
marr_Hildreth_Dilated_grouped = group_adjacent(dilated_edges_of_Hildreth)


cv2.imshow("T4b Canny Grouped Cameraman", canny_grouped)
cv2.imshow("T4b+ Canny Dilated Grouped Cameraman", canny_dilated_grouped)
cv2.imshow("T4a Marr-Hildreth Grouped Cameraman", marr_Hildreth_grouped)
cv2.imshow("T4a+ Marr-Hildreth Grouped Cameraman", marr_Hildreth_Dilated_grouped)
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.imwrite("t4b.tif", marr_Hildreth_grouped)
cv2.imwrite("t4a.tif", canny_grouped)
```

Figure 18: Group Adjacent Pixels Implementation

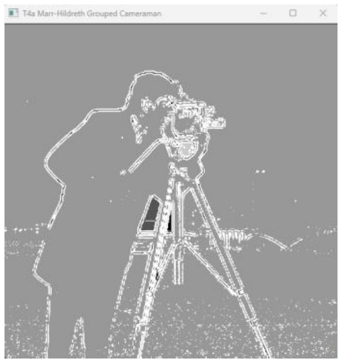


Figure 19: Marr-Hildreth Grouped Result

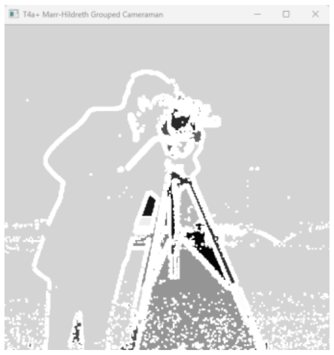


Figure 20: Marr-Hildreth Dilated Grouped Result

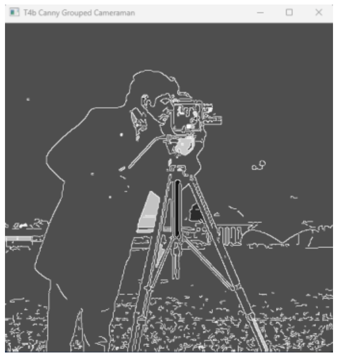


Figure 21: Canny Grouped Result

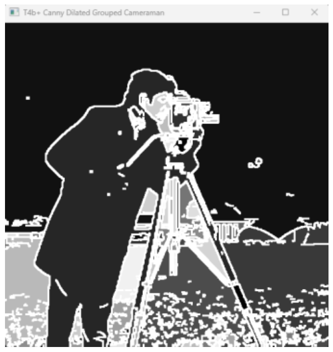


Figure 22: Canny Dilated Grouped Result

Comparison of Grouped Adjacent Pixel Results:

With both the dilated and un-dilated edge maps, Canny produces more connected regions than Marr-Hildreth. My Marr-Hildreth implementation had a lot of issues with random pixel placement and poor localization of edges as mentioned above. Canny edge detection also produced more image details in observation.