Sean Carda
PID: A59009786
ECE 253 – Image Processing
November 7, 2021
Homework 2

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.

By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

Problem 1:
Code:

```python
# Define a function for adaptive histogram equalization.
def AHE(im, win_size):
    # For looping purposes, grab the dimensions of the image.
    im = im[:, :, 0]
    im_size = im.shape

    # Also, instantiate an empty output image of the same dimensions as the input image.
    output = np.zeros([im_size[0], im_size[1]])

    # Grab the offset produced by the win_size.
    offset = int((win_size - 1) / 2)

    # Pad the image on all four sides such that the edges of the image may be operated on.
    im_pad = np.pad(im, offset, mode='symmetric')

    # Begin looping through the image.
    print("Starting to loop...")
    for x in range(0, im_size[0]):
        for y in range(0, im_size[1]):
            # Fetch the region of interest based on the current x, y coordinate.
            #region = im_pad[(x - offset):(x + offset), (y - offset):(y + offset)]
            region = im_pad[(x):(x + 2*offset), (y):(y + 2*offset)]

            # Calculate the rank of the coordinate by summing the boolean matrix comparing
            # im(x, y) to region.
            rank = sum(sum(im[x, y] > region))

            # Load the output with a pixel dependent on the window size and the rank counter.
            output[x, y] = round((rank * 255) / (win_size * win_size))

    # Return the equalized image.
    return np.uint8(output)

# File path to the beach image.
file_path = r"beach.png"

# Begin by importing an image.
A = cv2.imread(file_path)
```

```python
# Show the original image.
cv2.imshow('Beach', A)
cv2.waitKey(0)

# For each window size, compute the
for i in [33, 65, 129]:
    # Test the function.
    print("Equalizing the image...")
    B = AHE(cv2.imread(file_path), i)
    print("Equalization complete!")

    # Show the equalized image.
    cv2.imshow(('Beach_AHE_' + str(i)), B)
    cv2.waitKey(0)

    # Save the computed image.
    file_name = 'beach_image_win_' + str(i) + '.png'
    cv2.imwrite(file_name, B)


# Show the difference in performance between AHE and HE by using opencv to compute
# the HE image.
A = cv2.imread(file_path)
B = cv2.equalizeHist(A[:, :, 0])

cv2.imshow('Beach_AHE', B)
cv2.waitKey(0)
cv2.imwrite(('beach_image_he.png'), B)
```

Output:

Original Image:

AHE Image with a Window of Size 33:

AHE Image with a Window of Size 65:

AHE Image with a Window of Size 129:

Simple HE Image:

Questions:
a)
While the AHE image, aesthetically, does not look as pleasing as the original due to the wide spread of pixel intensities, the AHE image reveals significantly more information about the original image. Things such as the texture of the ground, the building, clouds in the sky, and even the fact that there exists a fourth person in the building, are revealed in the AHE image. The HE image is aesthetically more pleasing than the original due to the improved spread and balance of pixel intensities but does not reveal much more information that we could not already ascertain from the original.

b)
If we consider which enhancement method improves the general appearance of the given image, the HE approach is superior due to the smoother spread of pixel intensities throughout the image. If we consider which enhancement method is superior for something like a machine to read, the AHE image is superior due to the fact that AHE image contains more meaningful information. However, it is not necessarily true that this will hold for any image; it is possible that the AHE algorithm will distort the given image depending on the window size, and therefore requires more testing to reach a concrete conclusion.

Problem 2:

Part i:

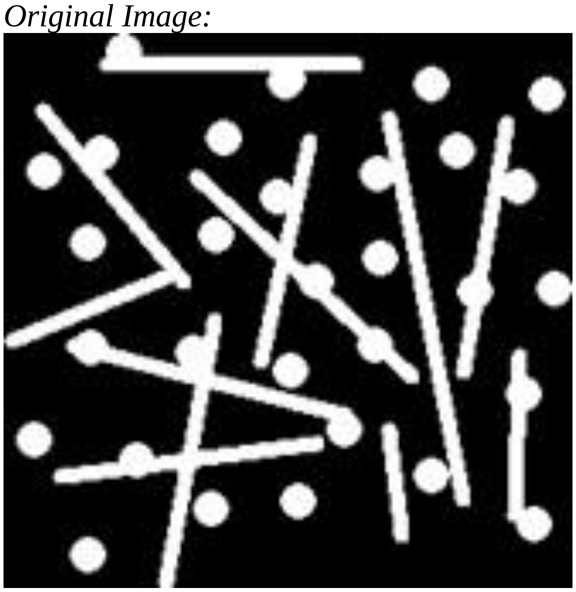
Code:

```python
#------
# i)
#------
cv2.destroyAllWindows()

# a)

# First, load the shapes image.
file_path = r'circles_lines.jpg'
shapes = (cv2.imread(file_path))
shapes_en = cv2.resize(shapes, (501, 489), interpolation=cv2.INTER_AREA)
cv2.imwrite('shapes_enlarged.jpg', shapes_en)
cv2.imshow('Shapes', shapes_en)
cv2.waitKey(0)

# Turn the image into a binary image for morphological operations. Here, pixels with intensities
# above 130 are given a value of 1. Otherwise, give a value of 0. For binary purposes, divide
# by 255 to have values of 0 and 1.
shapes_binary = np.uint8(np.where(shapes[:, :, 0] > 130, 255, 0) / 255)
bin_shapes_enlarge = cv2.resize(shapes_binary * 255, (501, 489), interpolation=cv2.INTER_AREA) # Enlarge
cv2.imwrite('circles_binary.jpg', bin_shapes_enlarge)
cv2.imshow('Shapes_Binary', bin_shapes_enlarge)
cv2.waitKey(0)

# First, instantiate a structure element for the morphological operation.
element = np.array([[0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
                    [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                    [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                    [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                    [0, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
                    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0]])


# Perform opening with the structure element defined above.
circles = morphology.opening(shapes_binary, element)

# Show the result.
circles_en = cv2.resize(circles * 255, (501, 489), interpolation=cv2.INTER_AREA)
cv2.imwrite('circles_opened.jpg', circles_en)
cv2.imshow('Circles', circles_en)
cv2.waitKey(0)

# b)

# Now, label the circles in the morphed image.
labeled_circles = ndimage.measurements.label(circles)

# Plot the labeled objects.
plt.imshow(labeled_circles[0])
plt.colorbar()
plt.xlabel('Columns')
plt.ylabel('Rows')
plt.title('Circles Colored by Connected Component Analysis')
plt.show()

# c)

# Method for computing the centroids and areas of a list of given shapes.
def compute_object_stats(object_original, object_matrix, object_count, quantity):

    # Instantiate a dictionary in which the centroid coordinates and areas will be stores.
    object_dictionary = {}

    # Grab the size of the input object matrix.
    object_matrix_size = object_matrix.shape

    # For every object, calculate its centroid and area.
    for i in range(1, object_count + 1):
        # First, calculate the area of the object.
        valid_elements = (object_matrix == i)

        # If the given quantity command is length, then compute the lengths of the objects.
        if quantity == 'length':
            max_length = 0
            for c in range(0, object_matrix_size[1]):
                length = sum(valid_elements[:, c])
                if length > max_length:
                    max_length = length
            measure = max_length

        # Otherwise, calculate the areas of the objects.
        else:
            measure = sum(sum(valid_elements))

        # Now, find all the x and y coordinates where the matrix has valid elements.
        # then, calculate the mean of those coordinates.
        r_list = []
        c_list = []
```

```python
        # Locate valid coordinates.
        for r in range(0, object_matrix_size[0]):
            for c in range(0, object_matrix_size[1]):
                if valid_elements[r, c] > 0:
                    r_list.append(r)
                    c_list.append(c)

        # Take their mean.
        r_mean = round(mean(r_list))
        c_mean = round(mean(c_list))

        # Load the area and centroid as a new value in the dictionary.
        object_dictionary['Object ' + str(i)] = [measure, [r_mean, c_mean]]

    # Create a new image and verify that the centroids have been calculated correctly.
    centroid_im = np.zeros([object_matrix_size[0], object_matrix_size[1], 3])
    centroid_im[:, :, 0] = object_original
    centroid_im[:, :, 1] = object_original
    centroid_im[:, :, 2] = object_original

    # For all the objects in the dictionary, grab the calculated coordinates and set the values
    # of the original image at those coordinates to blue.
    for key in object_dictionary:
        coordinates = (object_dictionary.get(key))[1]
        for r in range(0, object_matrix_size[0]):
            for c in range(0, object_matrix_size[1]):
                if r == coordinates[0] and c == coordinates[1]:
                    centroid_im[r, c, 1] = 0
                    centroid_im[r, c, 2] = 0

    # Return both the centroid image as well as the object dictionary containing the relevant
    # info on all of the objects in the image.
    return [centroid_im, object_dictionary]


# Method for printing the computed shape stats.
def print_stats(object_dict, measure):
    print("\n--Object--\t--" + measure + "--\t--Centroid [row, col]--")
    for key in object_dict:
        data = object_dict.get(key)
        print(key + ": \t" + str(data[0]) + "\t\t" + str(data[1]))

# Compute the object stats for the circle objects.
circle_stats = compute_object_stats(circles, labeled_circles[0], labeled_circles[1], 'area')

# Print the stats computed for the circles.
print_stats(circle_stats[1], 'Area')

# Display the positions of the centroids to verify that the centroids match the positions of the shapes.
centroid_en = cv2.resize(255 * circle_stats[0], (501, 489), interpolation=cv2.INTER_AREA) # Enlarge.
cv2.imwrite('circle_centroids.jpg', centroid_en)
cv2.imshow('Centroids for Circles', centroid_en)
cv2.waitKey(0)
```

Output:

Original Image:



Binary Image:

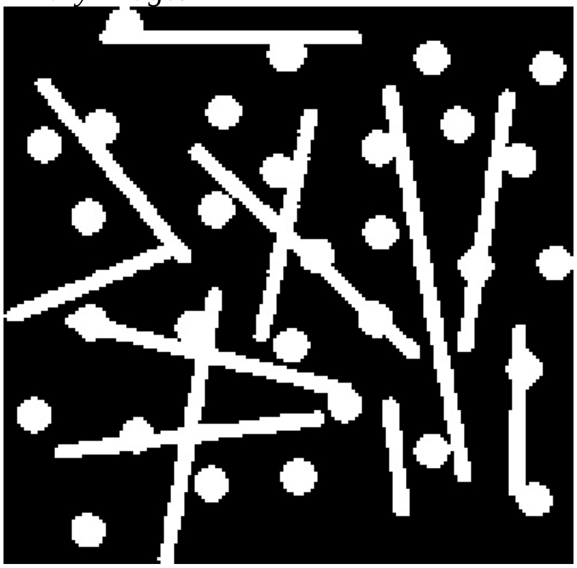
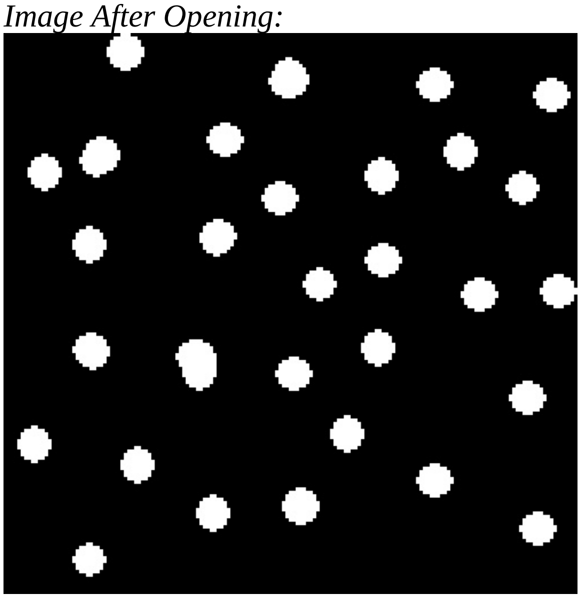


Image After Opening:



Structuring Element Used:

For this problem, the structuring element was a disk of radius 5. The matrix itself was a 10 x 10 matrix, with ones in places where pixels in the binary image should be kept, and zeros where pixels in the binary image should be removed. This structuring element was successful in removing most of the lines in the image. However, some of the preserved circles have areas larger than others due to the fact that some circles were intersecting with lines.

Image After Connected Component Analysis (with color-bar):

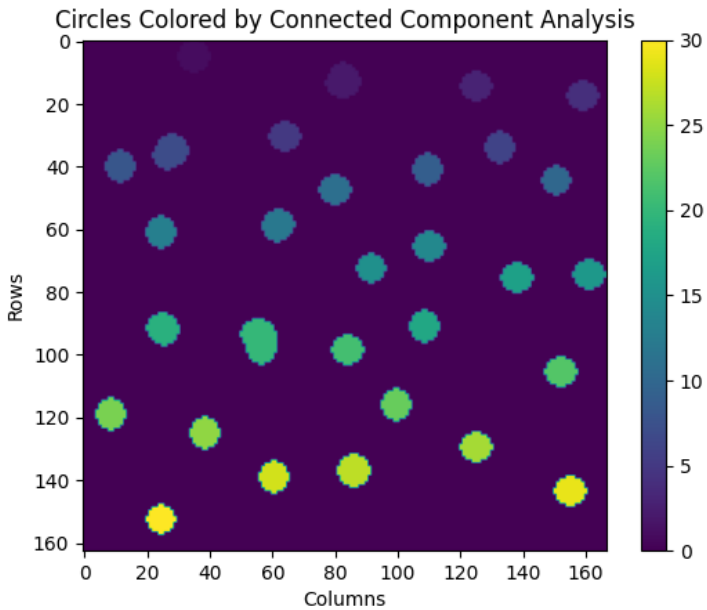


Image After Calculating Centroids:

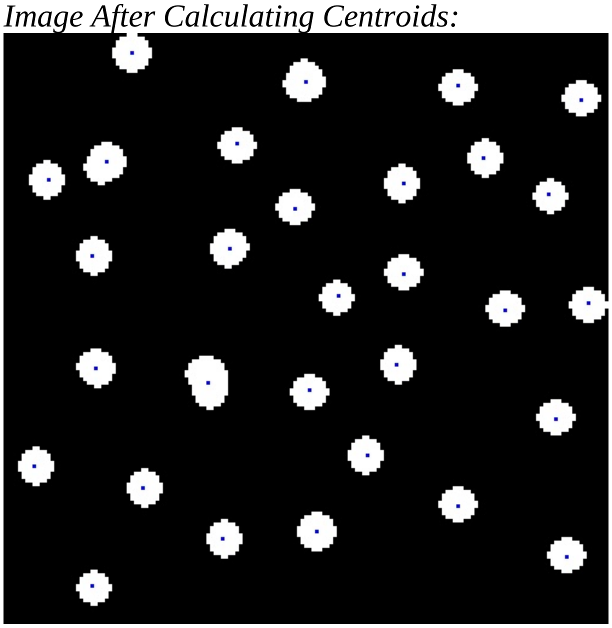


Table of Centroid and Area Values:

--Object--	--Area--	--Centroid [row, col]--
Object 1:	89	[5, 35]
Object 2:	102	[13, 83]
Object 3:	78	[14, 125]
Object 4:	78	[18, 159]
Object 5:	78	[30, 64]
Object 6:	78	[34, 132]
Object 7:	100	[35, 28]
Object 8:	78	[40, 12]
Object 9:	78	[41, 110]
Object 10:	68	[44, 150]
Object 11:	78	[48, 80]
Object 12:	85	[59, 62]
Object 13:	78	[61, 24]
Object 14:	78	[66, 110]
Object 15:	68	[72, 92]
Object 16:	78	[74, 161]
Object 17:	78	[76, 138]
Object 18:	78	[91, 108]
Object 19:	85	[92, 25]
Object 20:	132	[96, 56]
Object 21:	78	[98, 84]
Object 22:	78	[106, 152]
Object 23:	78	[116, 100]
Object 24:	78	[119, 8]
Object 25:	78	[125, 38]
Object 26:	78	[130, 125]
Object 27:	89	[137, 86]
Object 28:	78	[139, 60]
Object 29:	78	[144, 155]
Object 30:	68	[152, 24]

Part ii:

Code:

```
#------
# ii)
#------
cv2.destroyAllWindows()

# First, load the shapes image.
file_path = r'lines.jpg'
shapes = (cv2.imread(file_path))
shapes_en = cv2.resize(shapes, (462, 552), interpolation=cv2.INTER_AREA)
cv2.imwrite('shapes_enlarged.jpg', shapes_en)
cv2.imshow('Original Lines Image', shapes_en)
cv2.waitKey(0)

# Turn the image into a binary image for morphological operations. Here, pixels with intensities
# above 130 are given a value of 1. Otherwise, give a value of 0. For binary purposes, divide
# by 255 to have values of 0 and 1.
shapes_binary = np.uint8(np.where(shapes[:, :, 0] > 130, 255, 0) / 255)
bin_shapes_enlarge = cv2.resize(shapes_binary * 255, (462, 552), interpolation=cv2.INTER_AREA) # Enlarge
cv2.imwrite('binary_lines.jpg', bin_shapes_enlarge)
cv2.imshow('Binary Lines', bin_shapes_enlarge)
cv2.waitKey(0)

# a)

# First, instantiate a structure element for the morphological operation.
element = np.ones([13, 1])

#lines = morphology.opening(shapes_binary_filter, element)
lines = morphology.opening(shapes_binary, element)

# Show the result.
lines_en = cv2.resize(lines * 255, (462, 552), interpolation=cv2.INTER_AREA)
cv2.imwrite('lines_en.jpg', lines_en)
cv2.imshow('Lines', lines_en)
cv2.waitKey(0)

# b)

# Now, label the circles in the morphed image.
labeled_lines = ndimage.measurements.label(lines)

# Plot the labeled objects.
plt.imshow(labeled_lines[0])
plt.colorbar()
plt.xlabel('Columns')
plt.ylabel('Rows')
```

```
plt.title('Lines Colored by Connected Component Analysis')
plt.show()

# c)

# Compute the object stats for the vertical line objects.
line_stats = compute_object_stats(lines, labeled_lines[0], labeled_lines[1], 'length')

# Print the stats computed for the lines.
print_stats(line_stats[1], 'Length')

# Display the positions of the centroids to verify that the centroids match the positions of the shapes.
centroid_en = cv2.resize(255 * line_stats[0], (462, 552), interpolation=cv2.INTER_AREA) # Enlarge.
cv2.imwrite('centroids_lines.jpg', centroid_en)
cv2.imshow('Centroids for Lines', centroid_en)
cv2.waitKey(0)
```

Output:

Original Image:

Binary Image:

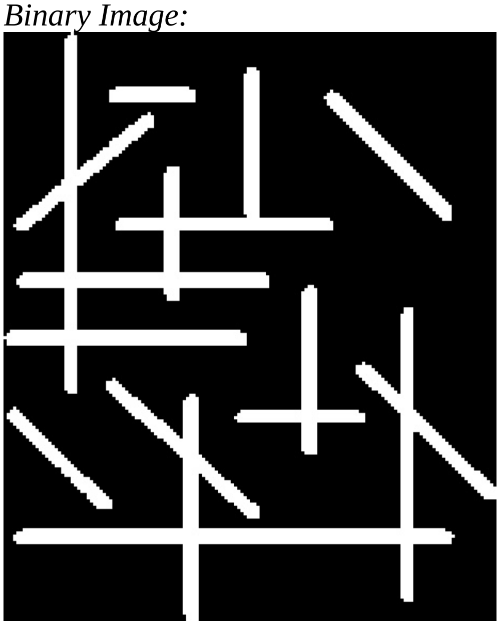
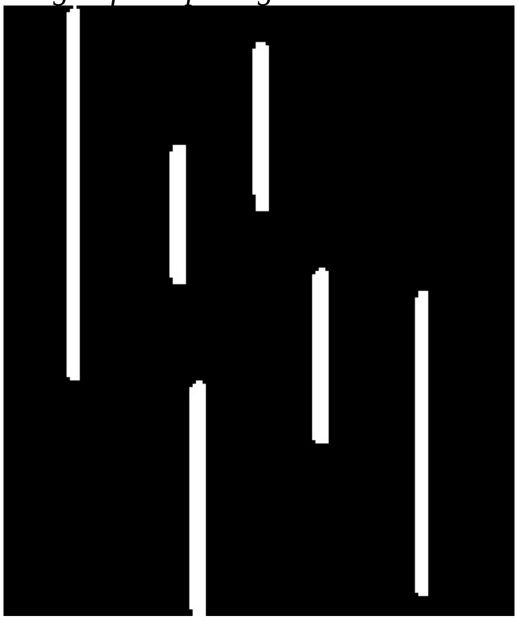


Image After Opening:



Structuring Element Used:
For this problem, the structuring element was a rectangle of length 13, and width 1. The matrix itself was a 13 x 1 matrix of all ones. No zeros were used in this structuring element since they would have been redundant – only the ones needed to fit in the image. This structuring element was very efficient in removing the horizontal and diagonal lines from the image.

Image After Connected Component Analysis (with color-bar):

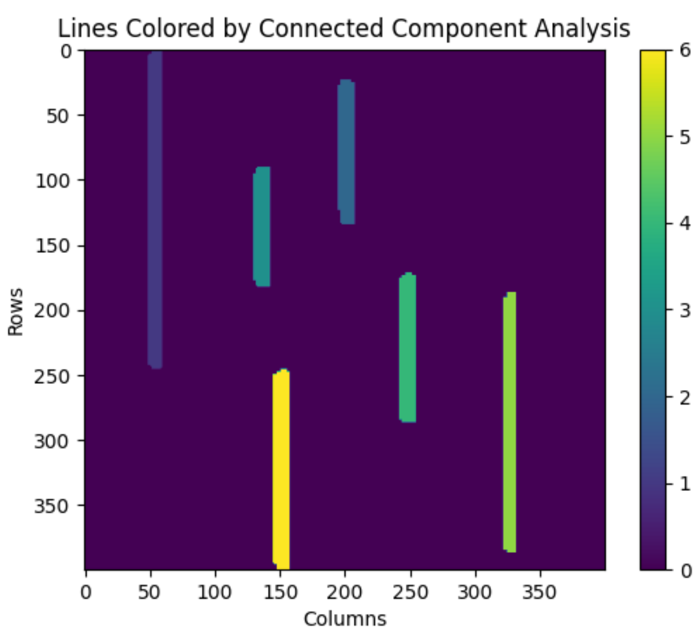


Image After Calculating Centroids:

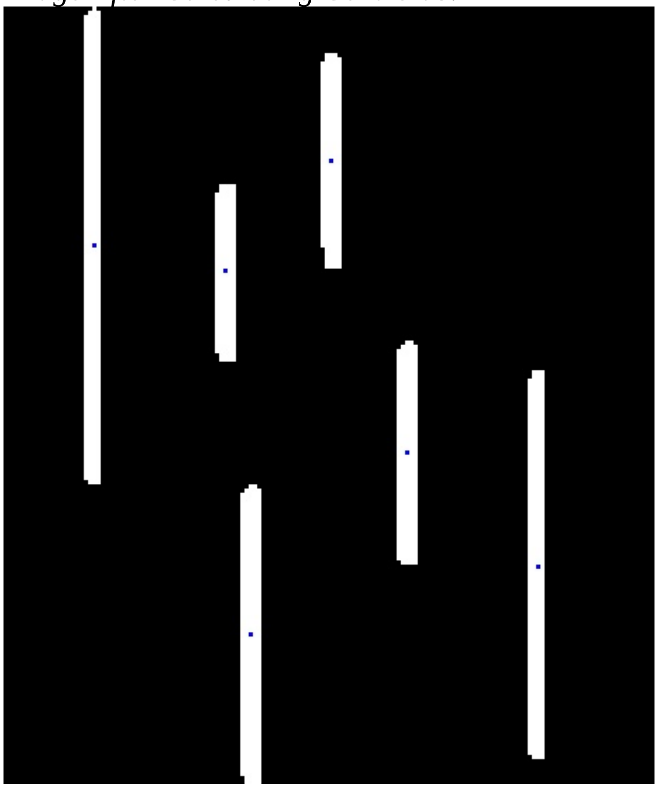


Table of Centroid and Area Values:

--Object--	--Length--	--Centroid [row, col]--
Object 1:	113	[56, 21]
Object 2:	51	[36, 77]
Object 3:	42	[62, 52]
Object 4:	53	[105, 95]
Object 5:	92	[132, 126]
Object 6:	71	[148, 58]

Problem 3:

Part i:

Code:

Part ii:

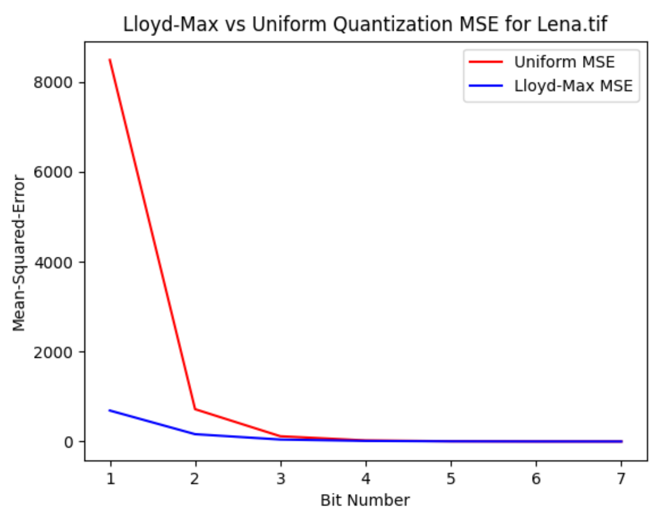
Code:

```python
#------
# ii)
#------

# This functions accepts an image and a bit value and computes the Lloyd-Max quantized image
# for the corresponding bit value.
def compute_lloyd_quant(im, s):
    # Flatten the lena image for the lloyds function to take as a input.
    dim = im.shape
    new_dim = [dim[0] * dim[1], 1]
    im_flat = np.reshape(im, new_dim)

    # Using the provided lloyd_python script, compute the necessary intensity partitions
    # and corresponding transformed intensity values.
    partition, codebook = lloyds(im_flat, [2**s])

    # For this new range of intensities, quantize the original image.
    output = np.zeros([dim[0], dim[1]])
    for r in range(0, dim[0]):
        for c in range(0, dim[1]):
            # Grab the current pixel value.
            pixel = im[r, c]

            # Setup a flag marker for when to stop searching for new values.
            flag = 0

            # Instantiate an index variable to keep track of the partition index.
            index = 0

            # While we have not reached the correct transform.
            while flag == 0:
                # If the current pixel value is less than the current partition bin, find all the
                # valid codebook values and find the value in the resulting list that is the max.
                if pixel < partition[index]:
                    vals = np.where(codebook < partition[index], codebook, 0)
                    output[r, c] = round(codebook[np.argmax(vals)])
                    flag = 1

                # If the current pixel value is greater than all partition bins, then we want to
                # transform that pixel into the largest codebook intensity.
                elif pixel > partition[len(partition) - 1]:
                    output[r, c] = round(codebook[2**s - 1])
                    flag = 1

                # Adjust the index in case the conditions above have not been met.
                index += 1

    # Return the quantized image.
    return np.uint8(output)


# Now, compute the mean squared error between the original image and the uniform quantized images
# and Lloyd-Max quantized images for bits 1 through 7. To do this, we will create a method
# that accepts an image and a quantization method and computes the MSE between the original image
# and the quantized image.
def compute_mse(im, quant_method):
    # Instantiate a dictionary to keep track of the MSE relative to the bit number.
    mse_vals = {}

    # Compute the quantizated image for every bit level. Instantiate an empty image set for
    # the quantization images.
    size = im.shape
    quantized_image_set = {}

    # Depending on the requested quantization method, compute the quantization images for the
    # 7 different bits.
    if quant_method == 'Lloyd':
        for i in range(1, 8):
            print('Computing Lloyd quantization for bit ' + str(i) + '...')
            quantized_image_set[i] = compute_lloyd_quant(im, i)
    else:
        for i in range(1, 8):
            print('Computing Uniform quantization for bit ' + str(i) + '...')
            quantized_image_set[i] = compute_uniform_quant(im, i)

    # For every quantized image, compute the MSE values.
    print('Computing MSE values...')
    for i in range(0, 7):
        # Grab the current quantized image.
        quant_image = np.uint64(quantized_image_set[i + 1])
        original_im = np.uint64(im)

        # Compute the error.
        error = (original_im - quant_image)

        error_squared = error**2

        # Sum the square of every term.
        squared_sum = sum(sum(error_squared))

        # Take the mean.
        mse = squared_sum / (size[0] * size[1])
```

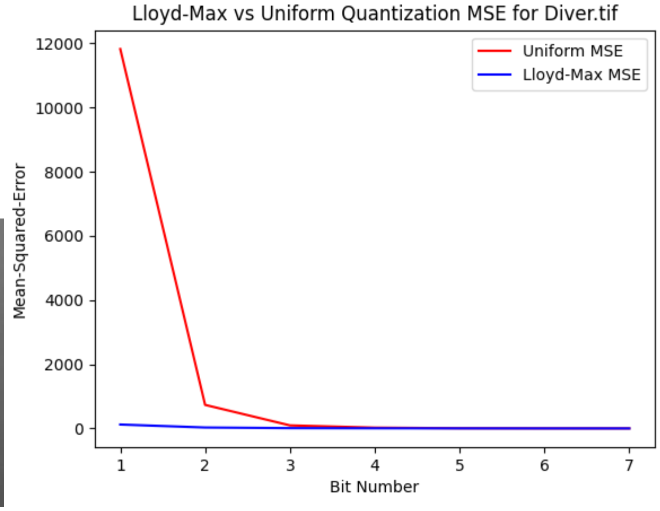
```python

            # Store in the corresponding bin.
            mse_vals[i] = mse

        # Return the values.
        print('--Computation for ' + quant_method + ' quantization complete!--')
        return mse_vals


# First, load the Lena512 image.
file_path = r'lena512.tif'
lena = (cv2.imread(file_path))

# Compute the MSE for the Lena image for both uniform and Lloyd quantization.
bit_numbers = [1, 2, 3, 4, 5, 6, 7]
lena_mse_unif = compute_mse(lena[:, :, 0], 'Uniform')
lena_mse_lloyd = compute_mse(lena[:, :, 0], 'Lloyd')

# Show the original image and plot the MSE values for both quantizers.
cv2.imshow('Lena Image', lena)
cv2.waitKey(0)
plt.plot(bit_numbers, lena_mse_unif.values(), color='red', label='Uniform MSE')
plt.plot(bit_numbers, lena_mse_lloyd.values(), color='blue', label='Lloyd-Max MSE')
plt.legend(loc="upper right")
plt.xlabel('Bit Number')
plt.ylabel('Mean-Squared-Error')
plt.title('Lloyd-Max vs Uniform Quantization MSE for Lena.tif')
plt.show()

# Second, load the Diver image.
file_path = r'diver.tif'
diver = (cv2.imread(file_path))

# Compute the MSE for the Lena image for both uniform and Lloyd quantization.
diver_mse_unif = compute_mse(diver[:, :, 0], 'Uniform')
diver_mse_lloyd = compute_mse(diver[:, :, 0], 'Lloyd')

# Show the original image and plot the MSE values for both quantizers.
cv2.imshow('Diver Image', diver)
cv2.waitKey(0)
plt.plot(bit_numbers, diver_mse_unif.values(), color='red', label='Uniform MSE')
plt.plot(bit_numbers, diver_mse_lloyd.values(), color='blue', label='Lloyd-Max MSE')
plt.legend(loc="upper right")
plt.xlabel('Bit Number')
plt.ylabel('Mean-Squared-Error')
plt.title('Lloyd-Max vs Uniform Quantization MSE for Diver.tif')
```

Output:

Original Lena Image and MSE Plot:



Original Diver Image and MSE Plot:



A likely reason for the stark gap in MSE between the uniform quantization and Lloyd-Max quantization methods, and perhaps a primary reason for why the Lloyd-Max quantizer outperforms the uniform quantizer, is due to the fact that the Lloyd-Max quantizer establishes intensity levels that are closer to the average intensity level of the original image than the uniform quantizer. This means that the gaps between the Lloyd-Max intensity levels will be smaller than the gab between the uniform intensity levels, resulting in lower error. This is the reason why the uniform quantizer produces such a large 1-bit MSE value – is that the 1-bit quantizer produces intensity levels at the most extreme ends of the pixel intensity spectrum (on a 8-bit scale, that is).

As the bits increase, the gap decreases between the two quantizers since the levels produced by both start to become closer to the levels in the 8-bit image, therefore reducing the error.

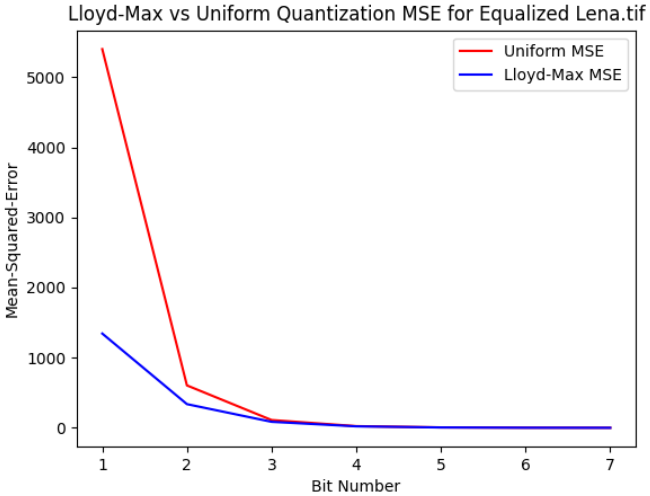
Part iii:

Code:

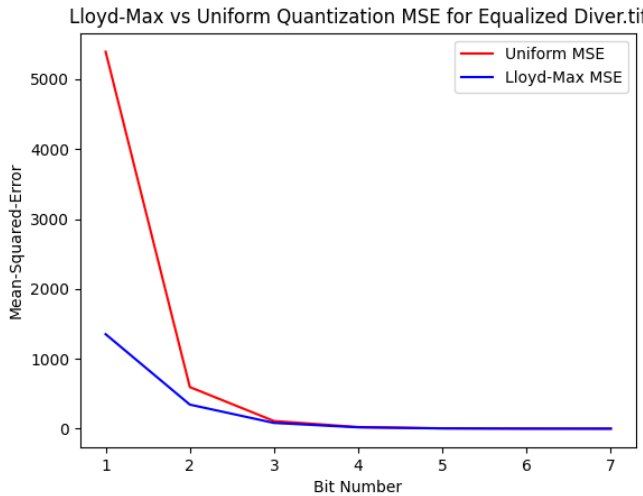
```python
#------
# iii)
#------

# Now, compute the MSE after equalizing the histograms
lena_eq = cv2.equalizeHist(lena[:, :, 0])

# Compute the MSE for the equalized Lena image for both uniform and Lloyd quantization.
bit_numbers = [1, 2, 3, 4, 5, 6, 7]
lena_eq_mse_unif = compute_mse(lena_eq, 'Uniform')
lena_eq_mse_lloyd = compute_mse(lena_eq, 'Lloyd')

# Show the original image and plot the MSE values for both quantizers.
cv2.imwrite('lena_eq.jpg', lena_eq)
cv2.imshow('Equalized Lena Image', lena_eq)
cv2.waitKey(0)
plt.plot(bit_numbers, lena_eq_mse_unif.values(), color='red', label='Uniform MSE')
plt.plot(bit_numbers, lena_eq_mse_lloyd.values(), color='blue', label='Lloyd-Max MSE')
plt.legend(loc="upper right")
plt.xlabel('Bit Number')
plt.ylabel('Mean-Squared-Error')
plt.title('Lloyd-Max vs Uniform Quantization MSE for Equalized Lena.tif')
plt.show()

# Equalize the diver image.
diver_eq = cv2.equalizeHist(diver[:, :, 0])

# Compute the MSE for the Lena image for both uniform and Lloyd quantization.
diver_eq_mse_unif = compute_mse(diver_eq, 'Uniform')
diver_eq_mse_lloyd = compute_mse(diver_eq, 'Lloyd')

# Show the original image and plot the MSE values for both quantizers.
cv2.imwrite('diver_eq.jpg', diver_eq)
cv2.imshow('Equalized Diver Image', diver_eq)
cv2.waitKey(0)
plt.plot(bit_numbers, diver_eq_mse_unif.values(), color='red', label='Uniform MSE')
plt.plot(bit_numbers, diver_eq_mse_lloyd.values(), color='blue', label='Lloyd-Max MSE')
plt.legend(loc="upper right")
plt.xlabel('Bit Number')
plt.ylabel('Mean-Squared-Error')
plt.title('Lloyd-Max vs Uniform Quantization MSE for Equalized Diver.tif')
plt.show()
```

Output:

Equalized Lena Image and MSE Plots:



Equalized Diver Image and MSE Plots:



Here, after equalization, the gap between the two quantizers has moderately decreased. It is clear that the uniform quantizer performs better after an image has been equalized, while the Lloyd-Max quantizer performs slightly weaker than when the images were not equalized.

A probable explanation for this could be that due to the increased contrast in the images after equalization. The increased contrast would benefit the uniform quantizer since there would exist more pixels on the lower ends of the 8-bit intensity spectrum, thus reducing the error between regions closer to intensities 0 and 255. This would not benefit the Lloyd-Max quantizer since the number of pixels closer to the Lloyd-Max intensity levels would decrease, and thus, the error would increase due to the distribution of intensities further away from the Lloyd-Max intensities.

Part iv:

I believe that the MSE of the Lloyd-Max quantizer stays close to zero for 7-bit quantization due to the fact that by nature, the Lloyd-Max quantizer locates the optimal threshold locations for a given image. The thresholds given by the 7-bit quantizer remain close to the concentrations of intensities resulting from the equalization since the quantizer is actively seeking those points.

Problem 4:

Part i:

Code:

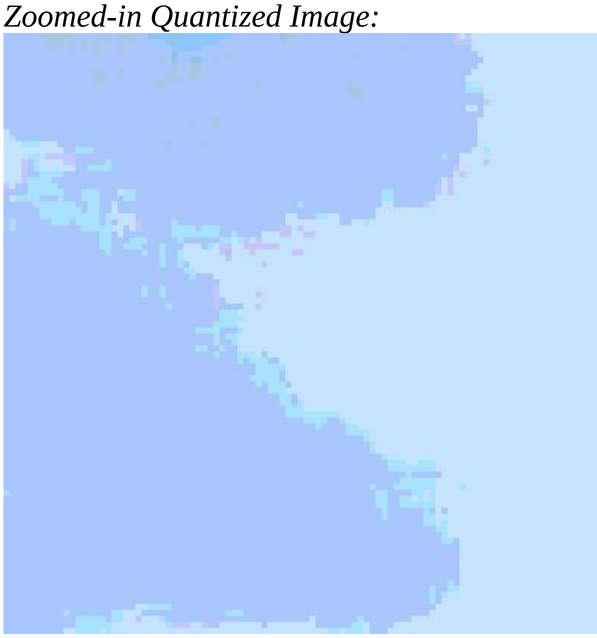
```
#------
# i)
#------

# Method which accepts an image and computes the uniform quantization for 10 intensity levels.
def compute_uniform_color_quant(im):
    # Compute the number of intensity values.
    levels = 10

    # Compute the new intensity values for the original pixel values.
    new_values = np.round((255 / (levels - 1)) * np.arange(levels))

    # Grab the size of the image.
    im_size = im.shape
    print(im_size)

    # Split the image into the three color bands and perform the quantization seperately
    # for each color layer.
    print('Computing uniform quantized values...')
    output = np.zeros([im_size[0], im_size[1], im_size[2]])
    for r in range(0, im_size[0]):
        for c in range(0, im_size[1]):
            # Calculate the difference in intensities between the current pixel value
            # and the new calculated intensities.
            difference_blue = abs(new_values - im[r, c, 0])
            difference_green = abs(new_values - im[r, c, 1])
            difference_red = abs(new_values - im[r, c, 2])

            # Find the minimum mapping from the old pixel value to the new intensity value.
            out_blue = new_values[np.argmin(difference_blue)]
            out_green = new_values[np.argmin(difference_green)]
            out_red = new_values[np.argmin(difference_red)]

            # Concatonate these values into the new output pixel.
            output[r, c, :] = [out_blue, out_green, out_red]

        if (r + 1) % 100 == 0:
            percent = round(100 * (1 + r) / im_size[0])
            print('--Computation Progress: ' + (str(percent) + '%'))

    # Return the final output image.
    return np.uint8(output)

# Load the geisel image.
file_path = r'geisel.jpg'
geisel = (cv2.imread(file_path))
cv2.imshow('Geisel Original', cv2.resize(geisel[476:576, 468:568], (400, 400), interpolation=cv2.INTER_AREA))
cv2.waitKey(0)

# Compute the uniform color quantization of the Geisel image.
geisel_uniform_quant = compute_uniform_color_quant(geisel)
cv2.imwrite('geisel_uniform_quant.jpg', geisel_uniform_quant)

# Show the resized image.
geisel_size = geisel.shape
geisel_uniform_quant_en = cv2.resize(geisel_uniform_quant[476:576, 468:568], (400, 400), interpolation=cv2.INTER_AREA)
cv2.imwrite('geisel_uniform_quant_en.jpg', geisel_uniform_quant_en)
cv2.imshow('Geisel Quantized Enhanced', geisel_uniform_quant_en)
cv2.imshow('Geisel Quantized', geisel_uniform_quant)
cv2.waitKey(0)
```

Output:
Original Image:

Zoomed-in Original:

Quantized Image:

Zoomed-in Quantized Image:



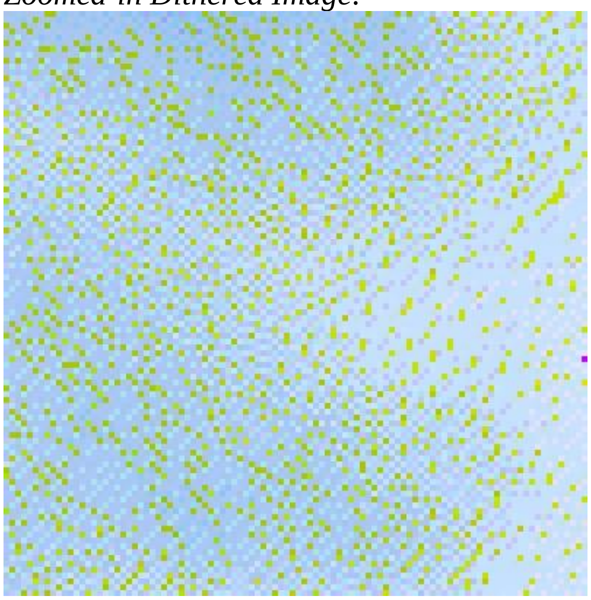
Part ii:

Code:

```python
#------
# ii)
#------

# Function which accepts an RGB pixel and computes a new pixel with the closest intensities
# for 10 new intensity levels.
def find_closest_pallate_color(pixel):
    # Compute a new pixel value given that we are quantizing to 10 intensity levels.
    new_values = np.round((255 / 9) * np.arange(10))

    # Compute new intensity values for all three color layers.
    red = new_values[np.argmin(abs(new_values - pixel[2]))]
    green = new_values[np.argmin(abs(new_values - pixel[1]))]
    blue = new_values[np.argmin(abs(new_values - pixel[0]))]

    # Concatonate the new pixel values.
    return [blue, green, red]

# Function which computes Floyd-Steinberg dithering quantization for a given RGB image. This
# function will compute the quantized image for 10 intensity levels.
def compute_floyd_steinberg_dither(im):
    # First, grab the dimensions of the image.
    size = im.shape

    # Now, pad the original image due to boundary computations in the F-S algorithm. This padding
    # only needs to be done for 1 pixel.
    im_pad = np.pad(im, ((1, 1), (1, 1), (0, 0)), mode='symmetric')

    # Instantiate an output image to return.
    output = np.int64(np.copy(im_pad))

    # Perform the algorithm.
    print('Computing Floyd-Steinberg quantized values...')
    for r in range(0, size[0]):
        for c in range(0, size[1]):
            # Grab the current pixel value from the given image.
            old_pixel = output[r, c, :]

            # Grab the new pixel based on the 'find_closest_pallete_color' function.
            new_pixel = find_closest_pallate_color(old_pixel)

            # Compute the error between the new pixel and the old pixel.
            error = old_pixel - new_pixel

            # Store the new pixel in the output image.
            output[r, c, :] = new_pixel

            # Now, compute the pixels surrounding the pixel we just stored.
            output[r + 1, c, :] = np.round(output[r + 1, c, :] + ((7 / 16) * error))
            output[r - 1, c + 1, :] = np.round(output[r - 1, c + 1, :] + ((3 / 16) * error))
            output[r, c + 1, :] = np.round(output[r, c + 1, :] + ((5 / 16) * error))
            output[r + 1, c + 1, :] = np.round(output[r + 1, c + 1, :] + ((1 / 16) * error))

        # Check progress of computation.
        if (r + 1) % 100 == 0:
            percent = round(100 * (1 + r) / size[0])
            print('--Computation Progress: ' + (str(percent) + '%'))

    return np.uint8(output)


# Calculate the Floyd-Steinberg dithering quantization for the Geisel image.
geisel_fs_quant = compute_floyd_steinberg_dither(geisel)
cv2.imwrite('geisel_fs_quant.jpg', geisel_fs_quant)

# Show the quantized image.
geisel_fs_quant_en = cv2.resize(geisel_fs_quant[476:576, 468:568], (400, 400), interpolation=cv2.INTER_AREA)
cv2.imwrite('geisel_fs_quant_en.jpg', geisel_fs_quant_en)
cv2.imshow('Geisel Dithered Enhanced', geisel_fs_quant_en)
cv2.imshow('Geisel Dithered', geisel_fs_quant)
cv2.waitKey(0)
```

Output:
Dithered Image:

Zoomed-in Dithered Image:



Questions:

1)

The biggest difference between the two images is that the uniformly quantized image possesses far less detail than the dithered image. And while it cannot be gleaned from far away, the dithered image has a “tiled” texture to it, where the pixels are weaved together to give the illusion of more intensity values than there really are, and thus “smoothens” the image. The texture of the quantized image is not as smooth, and the jumps between the intensity values are far more prominent due to the restrictions on the intensity levels. It should be noted that in this case, the colors on the dithering image look slightly stranger.

2)

The primary reason behind these differences is that the dithering algorithm propagates the error between the original and quantized pixels throughout the image. This is what leads to the “tiled” effect stated previously. The tiling gives the illusion that there are more intensity values than exist given the number of desired layers. This does not actually mean that there are an increased number of intensity layers than are desired.

The difference in terms of color is likely due to an error in propagating the pixel error throughout the image. As a result, some of the pixels appear distorted, and do not reflect the original colors of the image. This is likely not a flaw in the algorithm itself, but a flaw in managing the overflow of datatypes.