Ryan Miller
u1067596
Professor Whitaker
Digital Image Processing

Project 1 - Images, Introduction, and Histograms

Introduction

In the first project of CS6640, I explore basic digital image processing libraries in Python by implementing grey-level transformations and developing histograms on a variety of images. The goal of this project is to familiarize myself with the Python language and cover basic histogram processing of images. This report will be split up into four sections based on the tasks assigned in the project. In each section, the task at hand will first be introduced, followed by the reasoning and analysis behind the programming portion of the project. Note, the python scripts are attached in the assignment and are labeled by Question/Task. This first report will be more brief on the python reasoning than subsequent reports due to the simplicity of the programming portion of the assignment. A variety of images will then be presented to analyze the processing occurring in each task aiding in the analysis of the image processing theory. After all four sections have been presented, credit will be given to other people who contributed in some manner to the work I have presented.

Task 1

In task one, I am asked to complete a few basic tasks to ensure I can upload images from file directories using some imported image processing libraries from python. I utilize the read image function from the io module of the skimage library from python, a module that is used to read and write images of various formats. The shape of the image is output, defined as the 3D matrix that represents the number of pixels in its height and width by its depth (red, green, and blue). I then display the image using plt which plots the image with y and x axis numerical values to better understand the ordering of the pixels. Once the image is transformed to grayscale, the shape of the image reduces to a 2D matrix, getting rid of the red, green, and blue values. To write a function that converts the imported image to grayscale, I use the numpy dot command to return the image that is multiplied by weighted RGB values producing a grayscale image. The function is called color2gray and takes an image input and returns the image dotted with the weighted values.

The new image, the output of the grayscale function, is renamed to gray_img. To display this image, imshow is used from the matplotlib library, setting the cmap to grayscale. Although the picture is technically already grayscale after being passed through the function defined above, imshow by default uses a heatmap called viridis that displays image intensities and thus the grayscale mapping must be specified. Finally, the image is saved to the current directory using imsave in the io module of skimage.I present examples of input and output photos from Task 1.

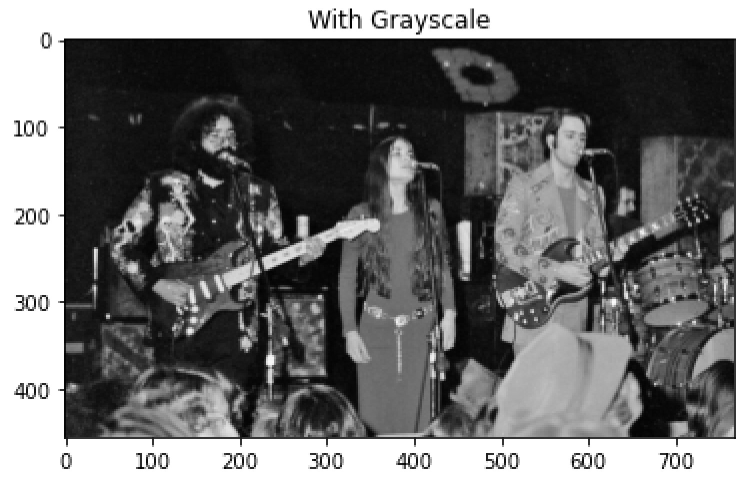

Figure 1: Grayscale image of Jerry Garcia, Donna Godchaux, and Bob Weir (left to right)

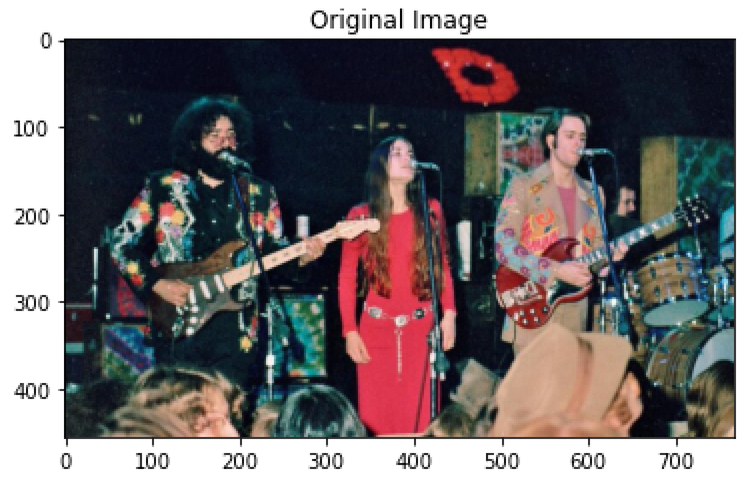

Figure 2: Original image of Jerry Garcia, Donna Godchaux, and Bob Weir (left to right)

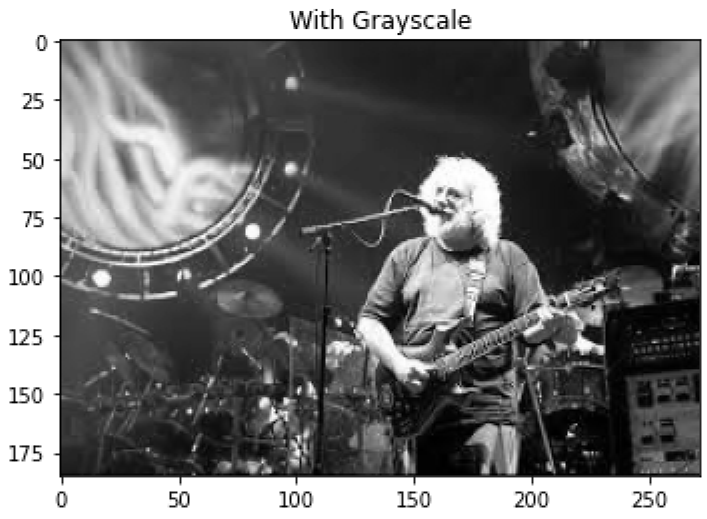

Figure 3: Grayscale image of Jerry Garcia

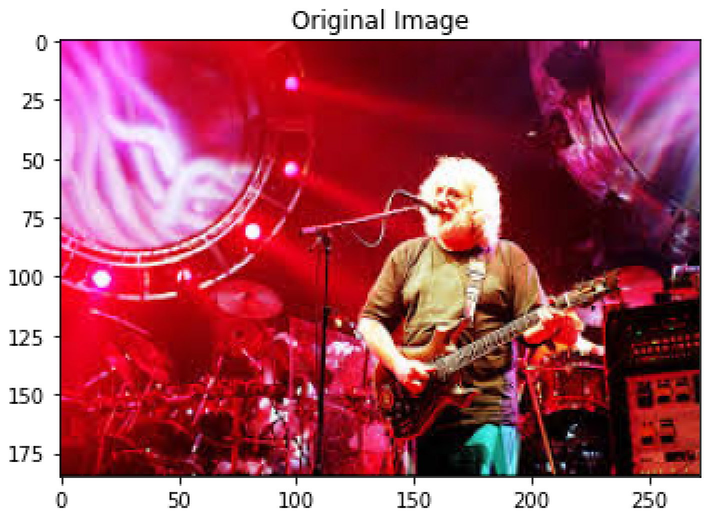

Figure 4: Original image of Jerry Garcia

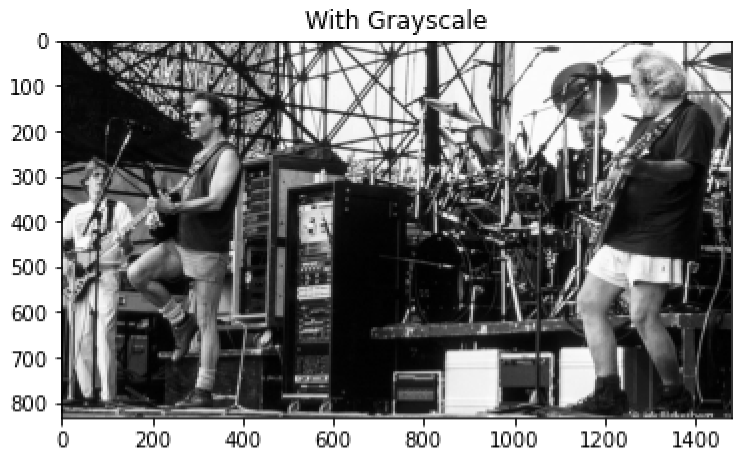

Figure 5: Grayscale image of Phil Lesh, Bob Weir, and Jerry Garcia (left to right)

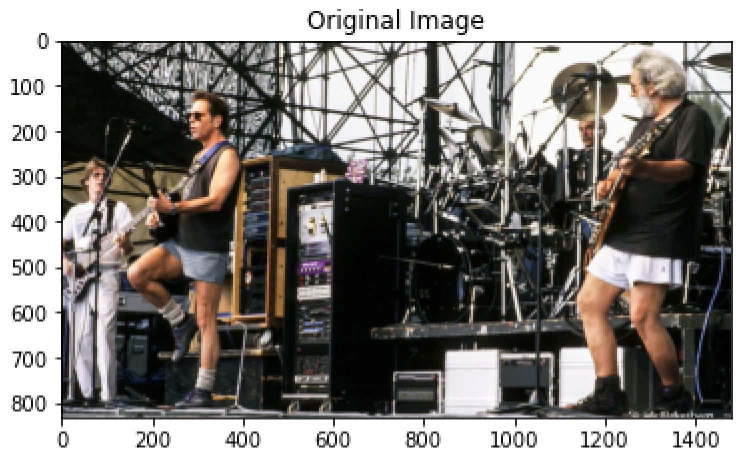

Figure 6: Original image of Phil Lesh, Bob Weir, and Jerry Garcia (left to right)

Task 2

In Task 2, I am asked to write a function from scratch that builds a histogram of an image based on its grayscale pixel values. The function returns a 2D array with the first column being the histogram bin values and where the second column is the bin counts. Without explicitly asking to use our original grayscale function from part 1, I decided to use the builtin python function rgb2gray to first convert images to grayscale before being passed through my function. In first analyzing the needs of the function, I knew that I would eventually have to index through the image matrix and record pixel numerical values. Thus, I knew I would need to iterate using multiple for loops and would thus need to first extract image dimensions upon which to iterate through. Thus, the first thing my function does is extract the image height and width using the

.shape function. I then take the max and min values of the image array to be able to start splitting up my bin values. For example, if I had an image with pixel values ranging from 0 to 255, I would need 256 bins to be able to accurately split up all the pixel data. Although bins are not always 1 unit large, but can become ranges, I took the bin size to be a constant value of 1.

To find the frequency of pixel numeric values, I constructed a series of 3 nested for loops. The outer loop iterates through the min and max bin count values. For the example I gave above, the outer for loop would iterate from 0 to 255 to be used to match the pixel values against. The inner two loops are used to iterate through the image where j and k iterate from 0 to the height and length, respectively. The innermost nested portion of the loop then checks to see if the image value at j and k are equal to the outer loop value. If it is, then the bin count array increments for that given value in the array. For example, if the image at j and k is equal to 13, then the bin count at i = 13 will be incremented. This methodology is used to count the frequency of the numbers and store that data in an array. Since the images being passed to this function are quite large, this portion of the code can take a while to complete.

After the for loops have finished running, small numpy operations are completed to bring the two arrays together to form a 2 column matrix of data as requested upon return of the function. Outside of the function, I pass in a grayscale image created using the rgb2gray function before extracting bin and frequency data for plotting. It must be noted that I consider myself a novice in python programming and the methodology and syntax used in this function are not efficient and should only be referenced if computational efficiency is not of concern.

The following grayscale images range from 0 to 255 and thus 0 corresponds to black and 255 corresponds to white.

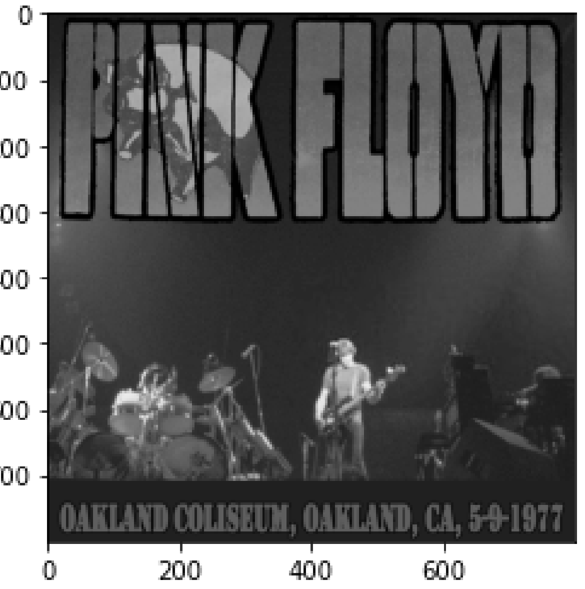

Figure 7: Grayscale image of Pink Floyd Post from 1977

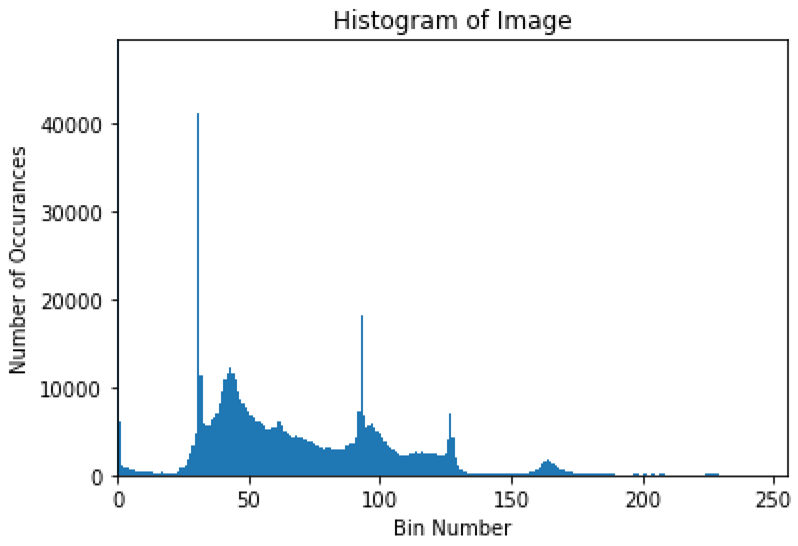

Figure 8: Histogram representation of Figure 7

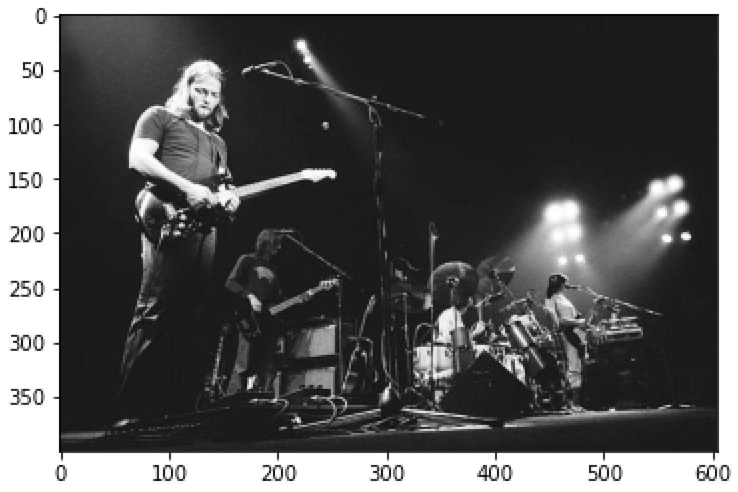

Figure 9: Grayscale image of Pink Floyd

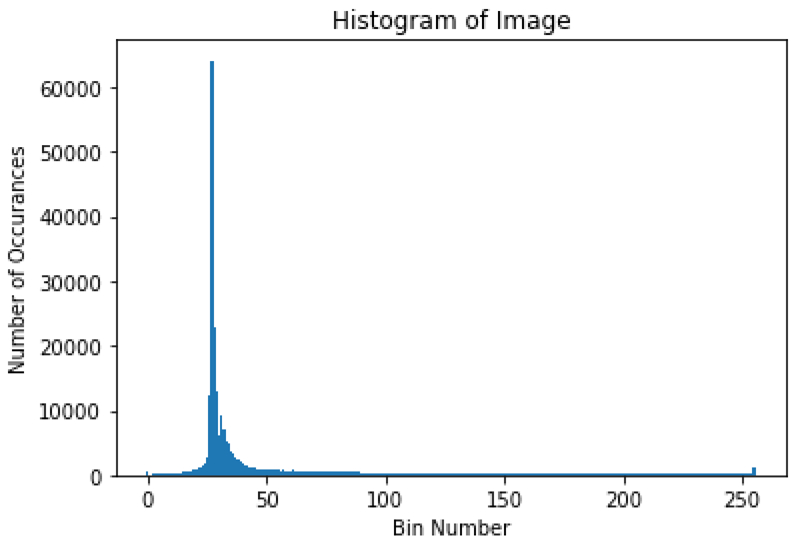

Figure 10: Histogram representation of Figure 9

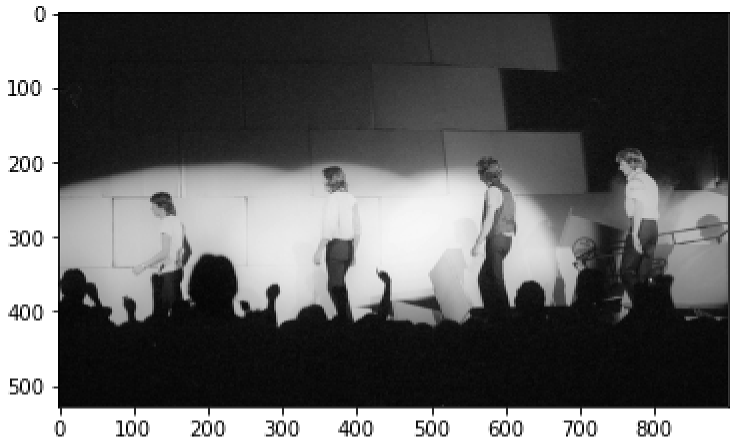

Figure 11: Pink Floyd grayscale image

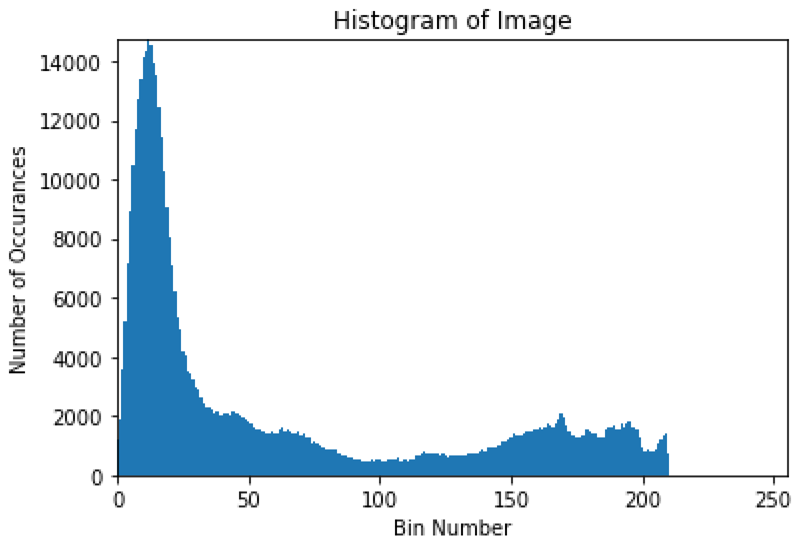

Figure 12: Histogram representation of Figure 11

The shape of the histograms presented above relate to the appearance of the original image. Since the histogram reflects the frequency of an intensity level, inferences can be made on the image by purely analyzing the histogram. For example, the histogram presented in Figure 10 is very narrow and lacks a large distribution. The values are concentrated heavily around 35, which on a 0 to 255 scale, appears very dark, nearly black. There is also a small spike near 255 signaling there is a small concentration of pure white in the photo. Upon looking at the image presented in Figure 9, our prediction is very spot on. Due to the all black background, there is no surprise that a large majority of intensity levels hover in the dark regions.

Similar analysis can easily be applied to Figures 11 and 12. Based on Figure 12, there appears to be large areas of very dark black with a decent amount of pixels that also take on a white value. Looking at Figure 11 affirms this analysis, with the crowd appearing all black and the lights on stage giving rise to the increase in distribution nearer the higher end of the 255 range. Although the range of grey intensities is low, they still appear in the background behind the spotlights of the image and thus can be made out on the histogram near the middle of the 0-255 range.

Task 3

In Task 3, I am asked to define a function that performs double sided thresholding on images to define regions before visualizing results via histograms and resulting images. Using built in libraries, I am then asked to perform flood fill and connected components on the thresholded images before visualizing results. Per the TA's instructions for this portion of the assignment, the inputs to the function are again assumed to be grayscale images and since the arguments for the function are not specified, the function sets thresholds within itself and does not take them to be arguments.

A thresholding function maps an image to binary values 0 and 1 depending on the pixel numeric value. A double sided thresholding function does this exact process, but twice over. It takes a range of values to be 1 and another range of values to be zero. The function T(r) resembles that of a square wave signal where the range of values taking on 0 and 1 are up to the programmer. To perform this task, I used a variety of builtin python libraries, including io, color, and measure from skimage as well as matplotlib and flood_fill from skimage.segmentation to do the flood fill portion of the task.

In order to implement this function, I knew I would again need to index through the image pixel values to either map them to the corresponding 1 or 0. Thus, a similar pair of nested loops as used in the last task are used again to iterate j and k in the range of the image height and width. Thus, the first thing the function does is extract information on the input image size. Threshold values are then set by the user defining the function and up for change to view the effects of thresholding on images. Then, using the nested loops and if statements, the image pixel values are checked to be inside or out of the thresholding range and set to 0 and 1. The thresholds are set up so that threshold 1 should be less than threshold 2. Thus, the first if statement checks if the pixel value is less than threshold 1 or greater than threshold 2. If the pixel is not, then the image pixel value is mapped to 255 (HIGH). If the image value is less than threshold 1 and greater than threshold 2, then the image pixel value is mapped to 0 (LOW). The resulting image is then returned from the function for analysis.

Thresholding sends all pixel values to black or white and the decided thresholds set by the user determine how much of the picture is sent to which color. The larger my threshold range that gets set to 255, the more pixels that get mapped to white (see Figures, 13, 14, and 15).

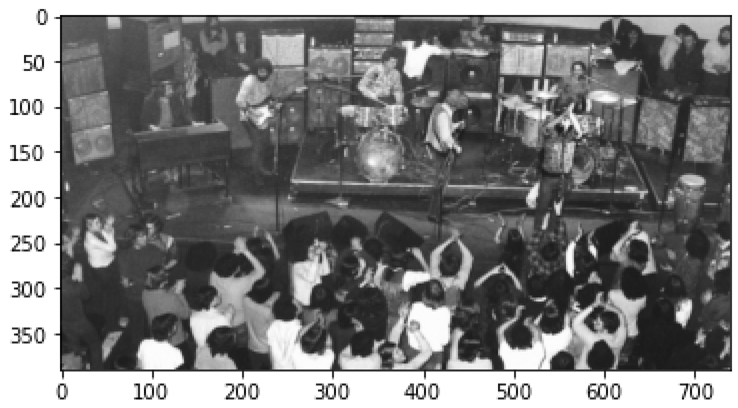

Figure 13: Original photo of Grateful Dead with Duane and Greg Allman

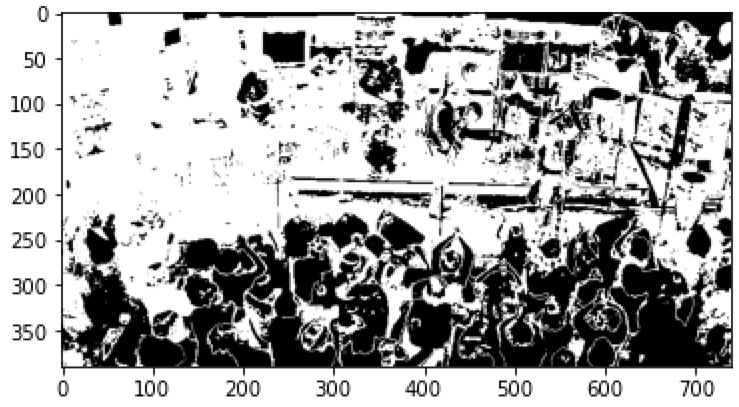

Figure 14: Figure 13 with low threshold set at 50 and high threshold set at 150

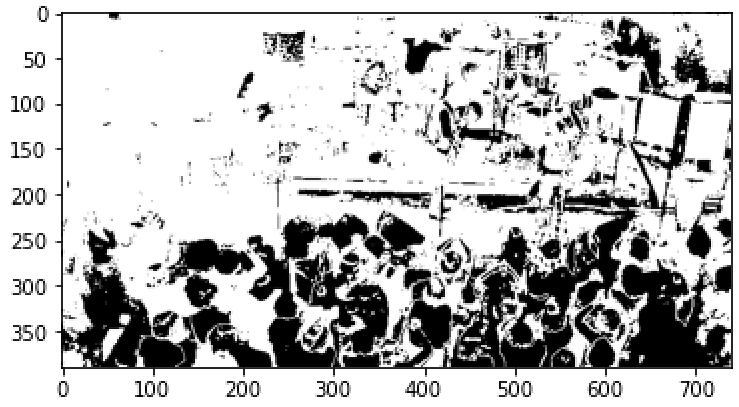

Figure 15: Figure 13 with low threshold set at 50 and high threshold set at 200

From these figures, it is clear that when I increase the effective range of the values mapped to a high value, the more amount of whitespace I see in the resulting image. One other thing that is very interesting is that the majority of black space near the bottom of the photo in Figure 15 actually originates from the white tee shirts in the crowd. This makes sense since the white pixel values are generally very high, close to 255, which is outside of our threshold range and thus mapped to completely black. In this sense, the double sided threshold is mapping black to black and white to black and only the in-between values (decided by the threshold but in between pure black and white, meaning some value of gray) to white. To further analyze this phenomena, lets step away from the rather complex musical concert photos and look at a rather simple black and white photo.

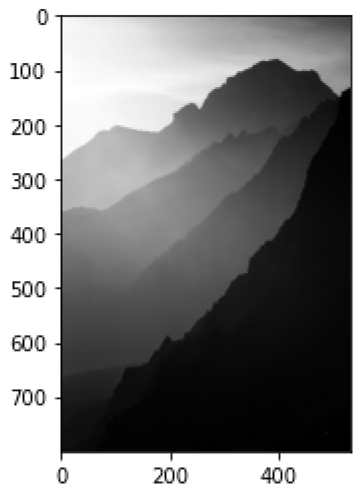

Figure 16: Original photo

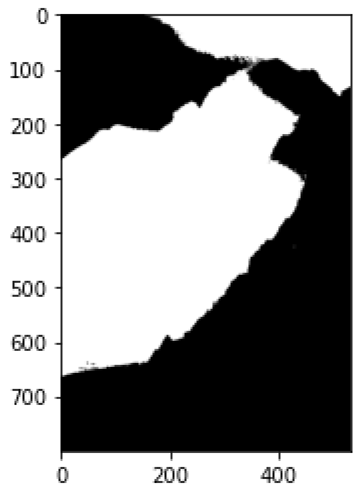

Figure 17: Thresholded image presented in Figure 16 - high threshold set at 200 and low at 50

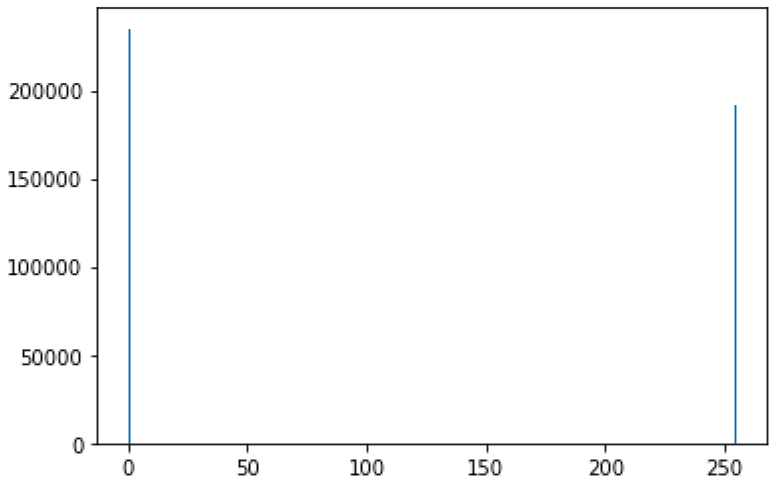

Figure 18: Histogram for Figure 17

From Figure 16 and 17, we again see that the new white color regions are actually those that started off as the lightest and darkest in the original photo. The middle values (some value of gray) are being mapped to white. To decrease the amount of gray being mapped to white, I think further decrease my range of values being mapped to high. In Figure 19, I present the same image with threshold values at 125 and 200, so that pixel values between these thresholds are mapped to high. We also see from the resulting histogram, a larger number of pixels sent to black which is evident in the resulting image.

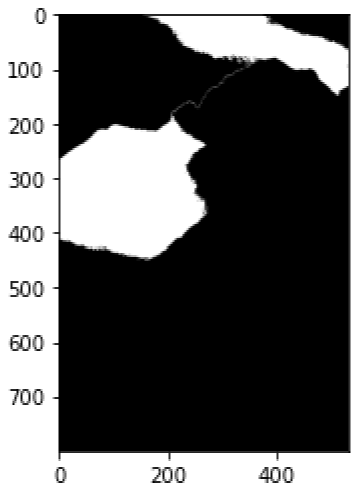

Figure 19: Figure 16 mapped with low and high threshold values set to 125 and 200, respectively

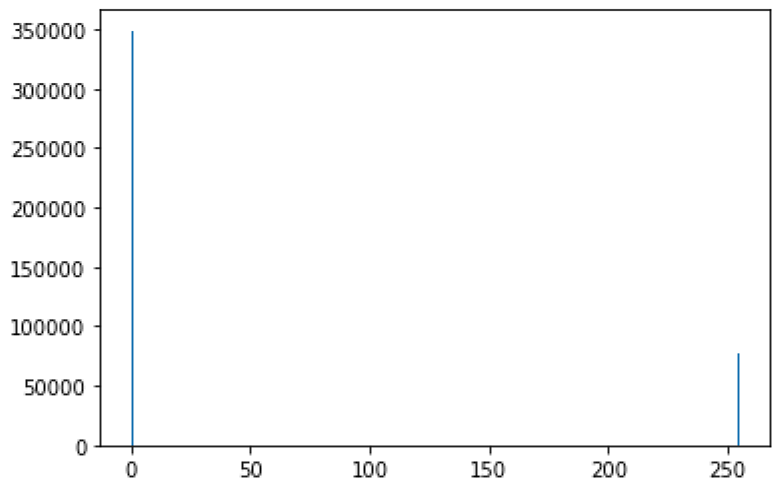

Figure 20: Histogram for Figure 19

The second part of this task has us perform flood fill and connected components on the thresholded images. The flood fill algorithm works much like the paint bucket tool in microsoft paint. You can choose wear to click the paint buck and the algorithm will go through the neighboring pixels to find pixels it is connected to and has the same color as before painting off subsequent pictures. In the flood fill built in function from skimage, you must specify a few parameters before implementation. First, you feed the function your image, while also specifying the seed point - your "paint bucket" click point, the grayscale value of your "paint bucket", and a tolerance value signaling how close in intensity the neighboring pixels need to be to be grouped in with your seed point. Since we have already mapped all of our image values to either 0 or 255, the threshold does not make much of a difference in our examples, unless it is set at 256. In the following examples, I set the threshold to a random value less than 255 and greater than 0. Thus, the only varying parameters to be mentioned are that of the seed point and the resulting paint intensity value.

The second portion of this task asks us to then perform connected components on the thresholded images and remove connected components that are smaller than a certain specified size before presenting the resulting image. I implemented this portion of the project using the measure library from skimage. My code takes the flood filled image and measures all the labels and then uses remove_all_objects from the morphology section of the library skimage to remove objects smaller than a specified size before graphing with the color map set to "nipy spectral" to view the different connected component portions is various colors.

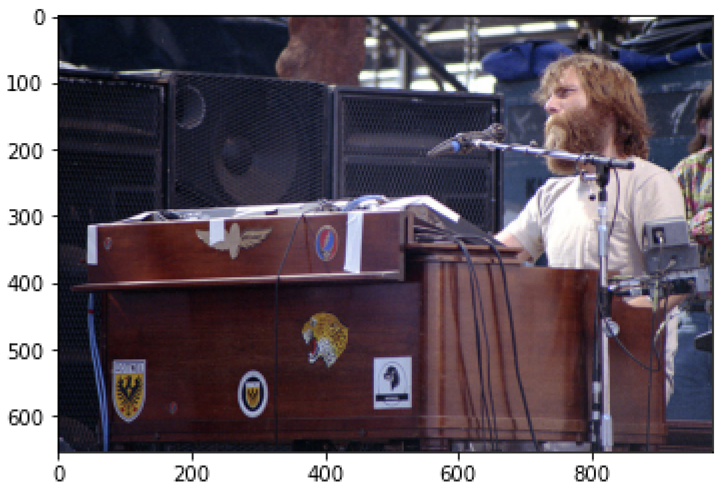

Figure 21: Original photo of Brent Mydland

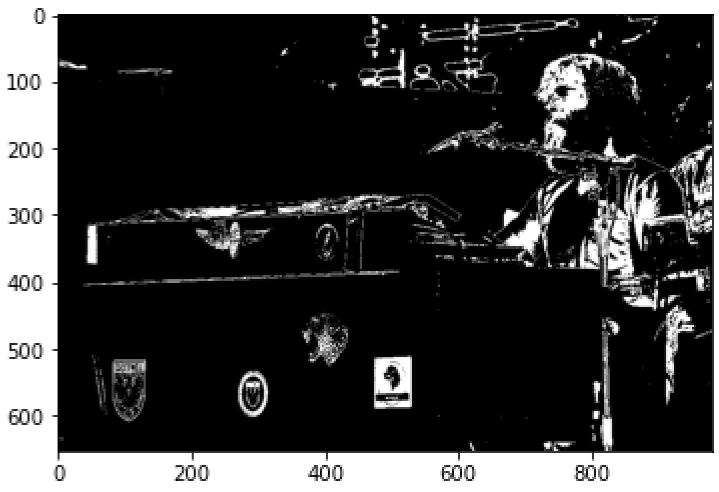

Figure 22: Resulting threshold image using low and high thresholds of 125 and 200, respectively

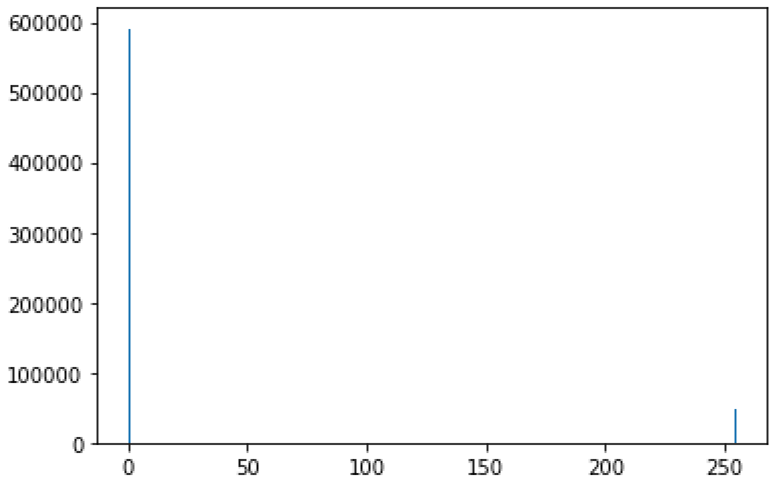

Figure 23: Histogram of Figure 22

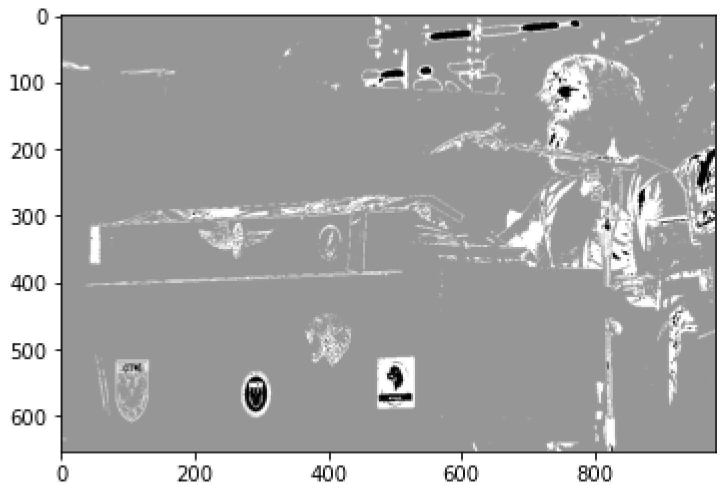

Figure 24: Resulting flood fill image of Figure 22, seed point at (200, 300) with fill value of 150

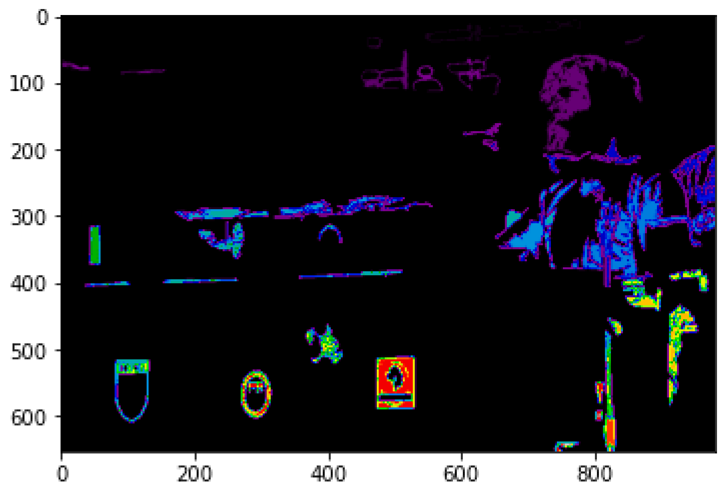

Figure 25: Connected components of Figure 22 with tolerance of 100 pixels

From Figure 24, we see the output of the flood fill portion of this task, whose theoretical explanation was presented earlier. For this exact image, the seed point was taken to be at (200, 300) with the intensity fill value of 150. This fill value explains why much of the image now appears gray. An intensity value of 150 lies between 0 and 255 and thus represents a gray color. Thus, from the resulting image, we can deduce that the seed point lies just outside of Brent's organ and thus connects together the background of the image. To really see the effect of the connected components, I probed the image to find an exact pixel value inside one of the stickers on the front of his organ. This resulting image is presented in Figure 26.

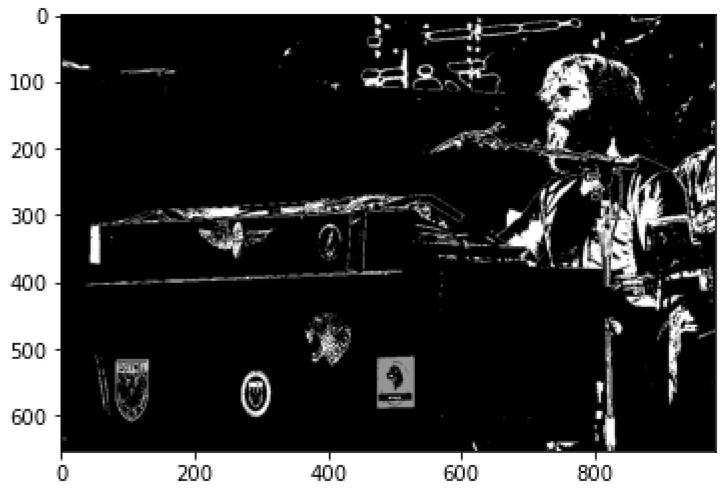

Figure 26: Flood fill implemented at bottom middle sticker on origin at seed point (550, 475)

From looking at 26, we now see that the "gray" fill is only done to the single sticker. This is because I strategically took the seed point to be the white space inside this sticker and thus the only pixels being filled are the original white values making up the background of the sticker.

This change does not have an effect on the connected components portion of the image because the subsequent connected components of the entire image are still the same. But if I wanted to change the size of the connected components that make the cut and are output as color, I can change the second input to remove_all_components. Currently, as presented in Figure 25, the cutoff is 100 pixels. If I increase this to 250 pixels, the resulting image is formed:

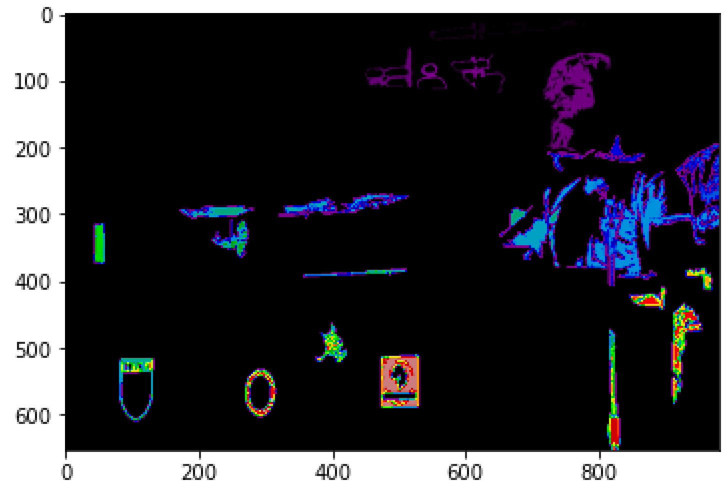

Figure 27: Resulting connected components from shutoff component size of 250 pixels

It is clear the regions that make the cut are fewer and some regions are now neglected and turned to blackspace.

Task 4

In task four, I am asked to perform histogram equalization on a variety of images and show the results before performing local histogram equalization. We have already gone over the theory behind histograms earlier in this report. Histogram equalization acts to adjust the contrast of an image based on its histogram by increasing the global contrast. In histograms, this will result in a subsequent distribution of the intensity values. If we were to graph the cumulative distribution function for the histogram, we would expect to see a straight line with constant slope. To implement this task in Python, I utilized a few libraries and modules. I first take the image in and show the original picture and the original histogram using the hist module from matplotlib. I then calculate the cdf for the histogram using the histogram module in numpy to return the count and bins for the histogram. To calculate the cdf, I simply sum the probability density and then use cumsum from numpy to sum along an axis. I then plot the cdf to ensure that the histogram has equalized as expected. For the following images, I present the original image, the original histogram, the original cdf and then the subsequent equalized photo, histogram, and cdf.

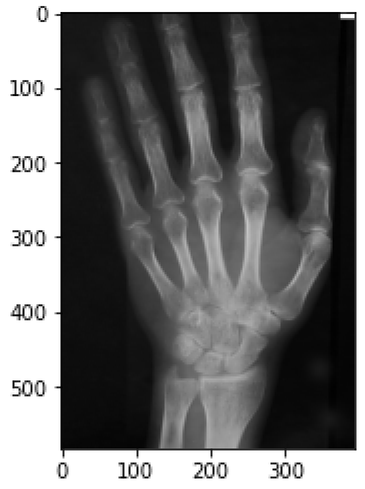

Figure 28: Original image used in analysis for task 4

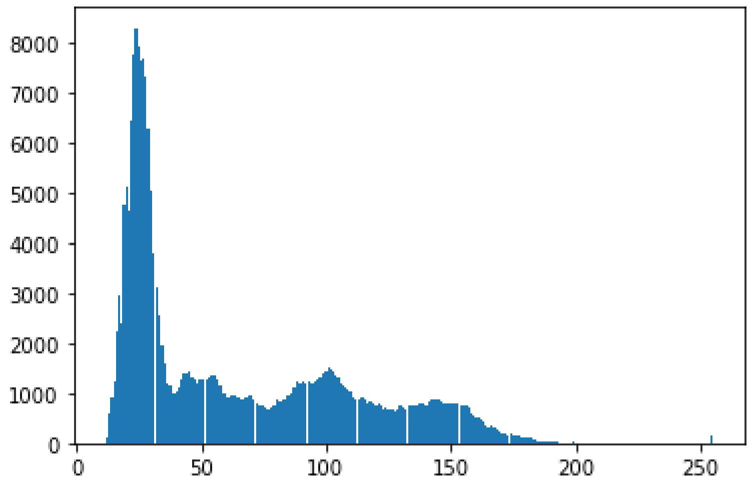

Figure 29: Histogram for the original image presented in Figure 29

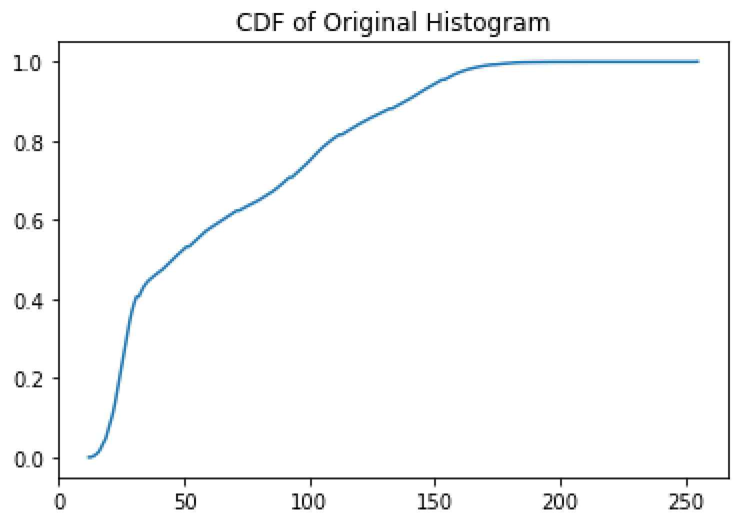

Figure 30: Cumulative distribution of histogram presented in Figure 29

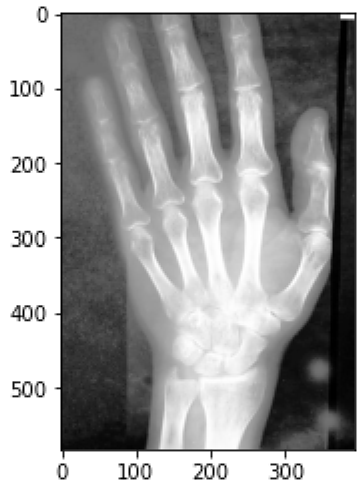

Figure 31: Resulting image after histogram equalization

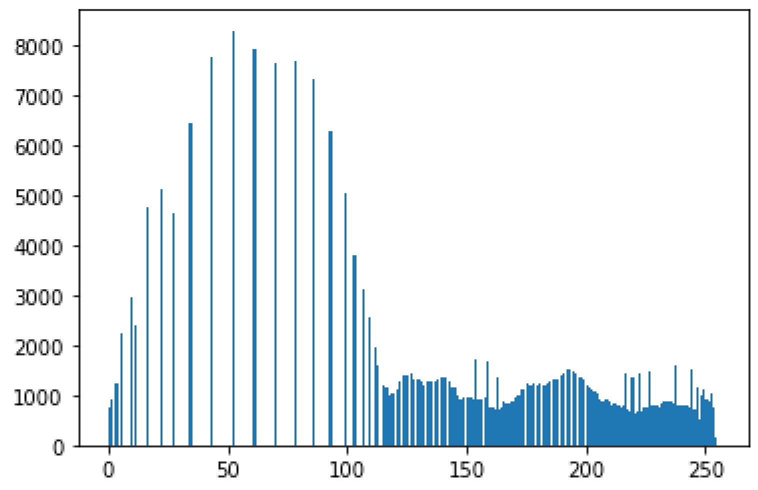

Figure 32: Resulting histogram after histogram equalization

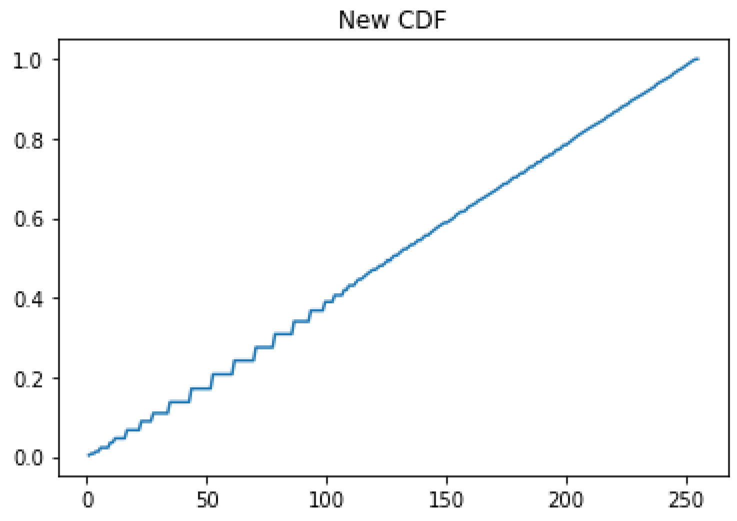

Figure 33: New CDF after histogram equalization

From Figures 29 and 32, we see the effect of the histogram equalization, where the called function acts to spread out the density of intensity values over the entire range of possible values. This is further shown in the CDF graphs, Figures 30 and 33. Figure 30 represents the original CDF which is shown to be front heavy; a lot of values presented in histogram in Figure 29 are nearer 0 than 255. But after the equalization, the distribution resembles that of a straight line where the intensity values in the histogram are evenly distributed. I will present similar actions carried out on another image to show for the purpose of this presentation but no more analysis will be given since there aren't any parameters to be altered and thus the results match those just presented in Figures 28 to 33.

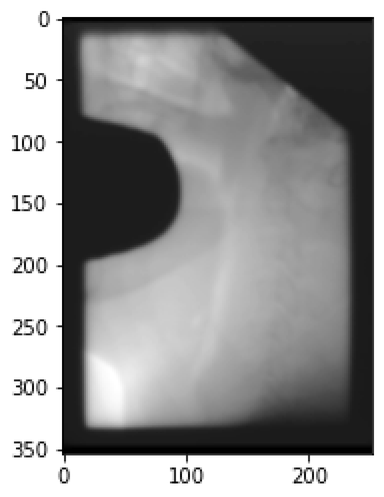

Figure 34: Original photo used for histogram equalization

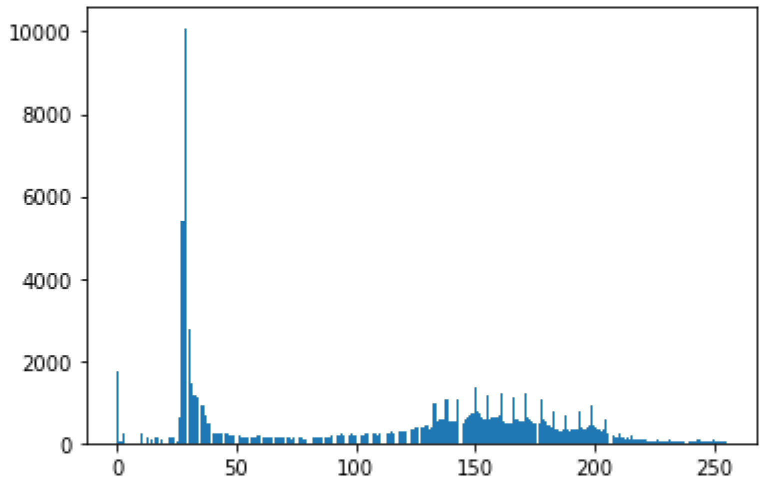

Figure 35: Histogram for Figure 34

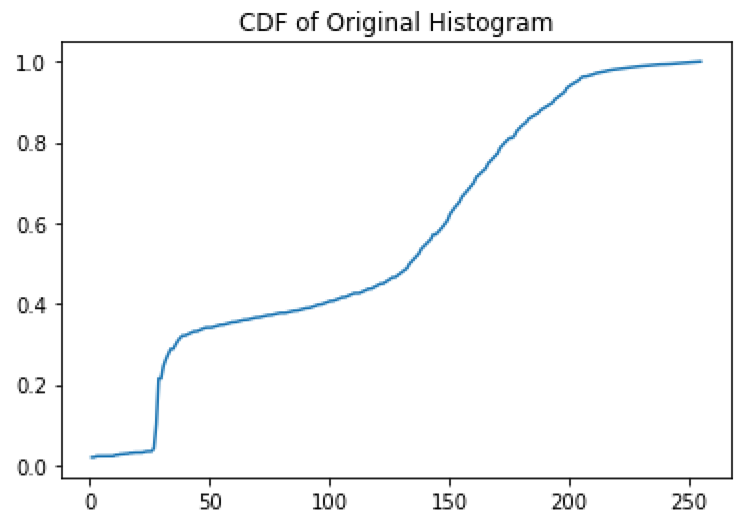

Figure 36: CDF for Figure 34

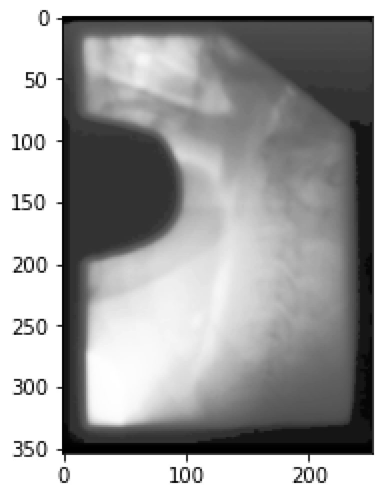

Figure 37: Image after histogram equalization

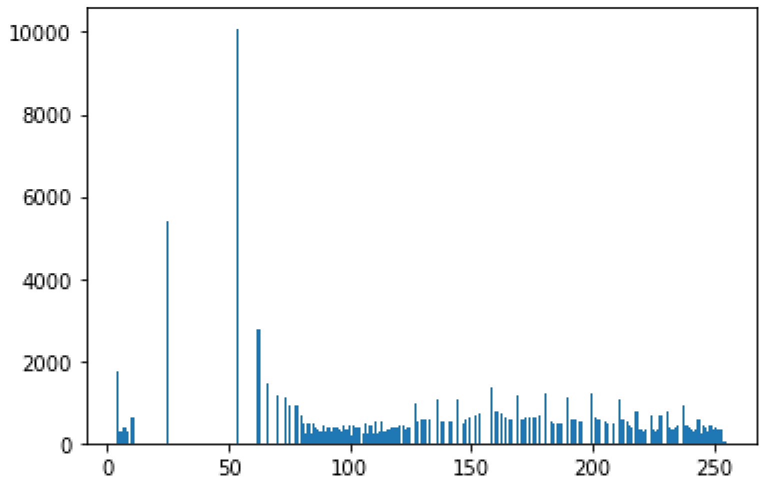

Figure 38: Histogram after histogram equalization

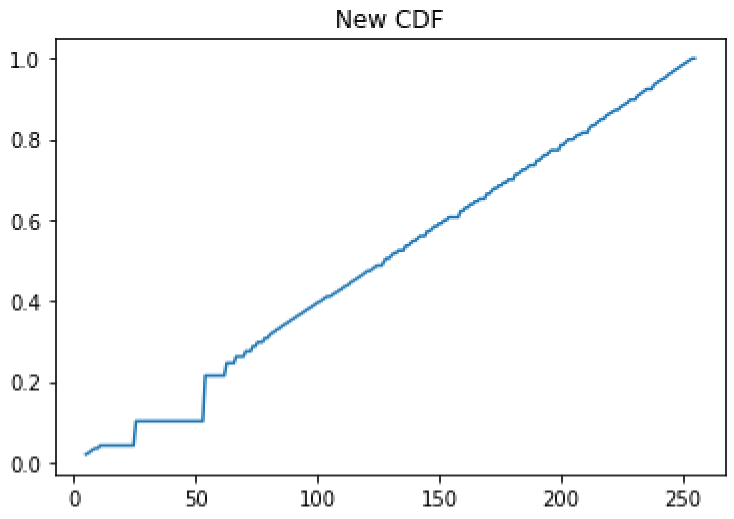

Figure 39: CDF after histogram equalization

The last portion of this task asks us to perform adaptive histogram equalization on a variety of images. To make the problem slightly more interesting, I implemented a function that performs contrast limited adaptive histogram equalization (CLAHE). CLAHE is a variant of adaptive histogram equalization where the contrast amplification is limited to reduce noise amplification. The clipping size and kernel size can be fed into the function performing the clipped adaptive histogram equalization. To implement this portion of the task in my code, I simply imported the equalize_adapthist from skimage.exposure and input the various parameters for the clipping. I then decided to view the results by also displaying the cdf curve as outlined earlier in this task.

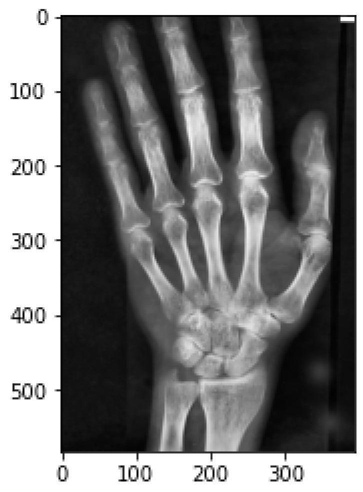

Figure 40: CLAHE applied to Figure 28 with clip limit set to 0.01

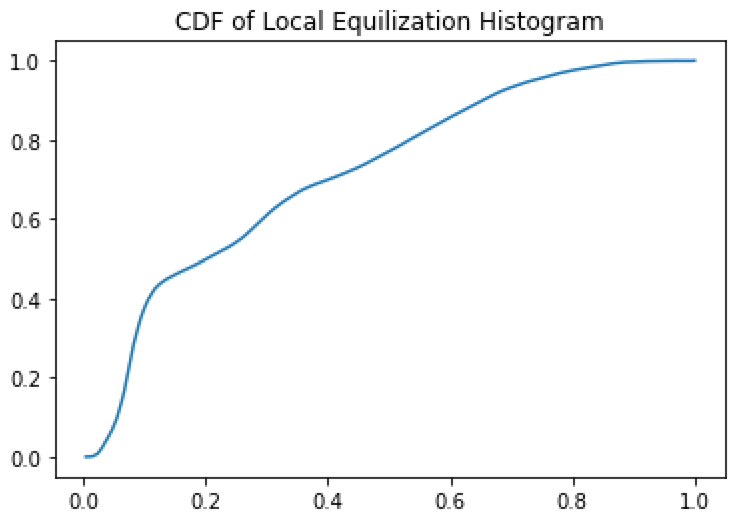

Figure 41: CDF from Figure 38

Upon comparing Figure 40 to Figure 28, it is clear that there is not much of a change in the image, except for slight increase in contrast. Upon looking at the resulting histogram after applying the adaptive histogram, it is showing a histogram that is nearly identical to the original presented in Figure 29. Upon looking further at the documentation online, this is expected as the clip limit is set between 0 and 1 (normalized) with 1 being higher contrast. Thus, to show this variable, I subject the same figure to a clipping value of 0.5.

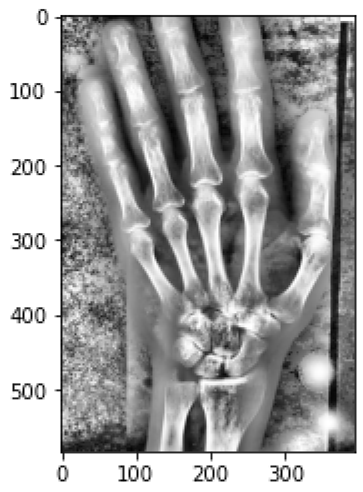

Figure 42: CLAHE image with clipping set to 0.5

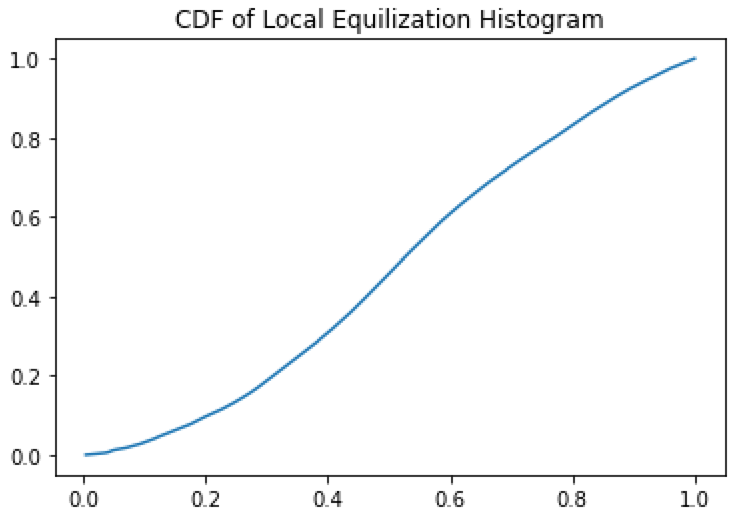

Figure 43: CDF from Figure 40

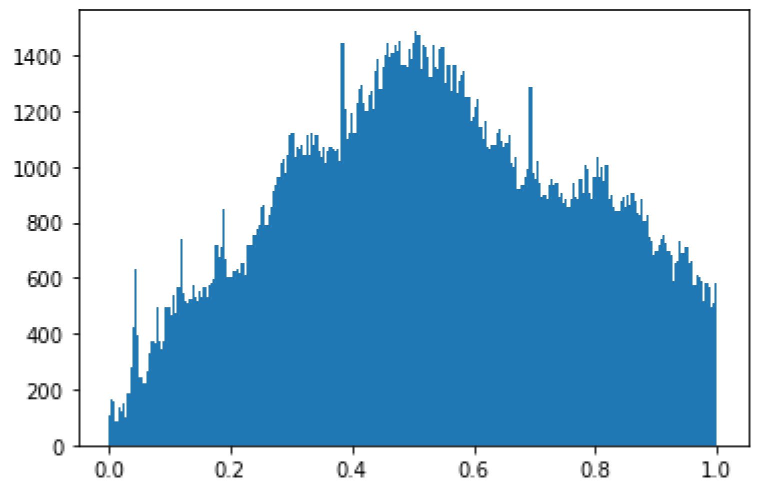

Figure 44: Histogram aftering applying CLAHE

As expected, the histogram distribution of values has been spread out across normalized values 0 to 1. Out of pure curiosity a clipping value was set to 1, which is presented in Figure 45 to be exactly the same as the image with clipping value of 0.5.

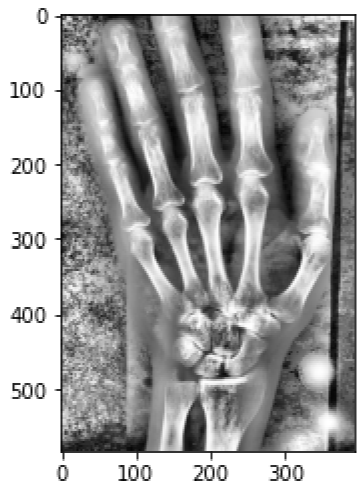

Figure 45: Clipping set to 1

As shown in Figure 45, the actual image does not change much past 0.5. Upon rethinking about this idea, it actually makes sense since a clipping value of 0.5 is clipping a majority of the distribution already present, so further increasing the clipping limit has little to no effect on the total amount of data being redistributed.

Extra Note

Upon subsequent testing of my functions and the above outlined tasks on random images from the internet, I have realized a few things that I feel I should briefly speak on and explain implementation of in the programming portion of this assignment. Grayscale images can be represented in multiple different ways. In the end, the theoretical concepts of the images are the same no matter what way they are represented. In general, I have done this project assuming that the grayscale image being input to the function is a 2D integer array with intensity values from 0 to 255, where 0 represents completely black and 255 being completely white. This is just one possible way to represent an image. After further checking of images online, instead of representing the range of 0 to 255, one could also represent the range from 0 to 1 with large precision floating point numbers. The intensity mapping from 0 to 1 is still completely valid for that of 0 to 255. Thus, what I subsequently had to do after writing large portions of code to take into account strictly images with range 0 to 255 in integer values, is that I actually had to go in and see if the image had range 0 to 1 and make the appropriate adjustments. The easiest way to do this that I came up with was to check the modulus of the values and if the pixel value mod 1 is not zero, then I know I am not dealing with an integer and subsequent changes can be made to format the image in the necessary way for my functions. It must be noted that this is by no means incorrect but it must be considered more carefully in following projects.

Credit:

I could not have been able to do the following project without the work of a few people and many online forums where countless people have contributed their own work. I want to thank Dr. Ross Whitaker and Jadie Adams for helping guide me on a few problems. I will present the rest of the credits in URL format.

Task 1:

- Dr. Ross Whitaker
- Julie Adams

Task 2:

- https://stackoverflow.com/questions/39805697/skimage-why-does-rgb2gray-from-skimage-color-result-in-a-colored-image
- https://www.tutorialspoint.com/how-to-remove-gaps-between-bars-in-matplotlib-bar-chart
- https://stackoverflow.com/questions/59577778/matplotlib-weekly-bars-are-too-thin-when-width1-0-too-thick-when-width-1-0
- https://www.youtube.com/watch?v=-mF8yrw0f6g
- https://numpy.org/doc/stable/reference/generated/numpy.ndarray.size.html
- https://stackoverflow.com/questions/15675667/column-matrix-representation-in-python
- https://www.w3schools.com/python/numpy/numpy_array_indexing.asp

Task 3:

- https://matplotlib.org/stable/tutorials/colors/colormaps.html
- https://scikit-image.org/docs/stable/api/skimage.filters.html#skimage.filters.gaussian
- https://scipy-lectures.org/packages/scikit-image/auto_examples/plot_labels.html
- https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_floodfill.html
- https://towardsdatascience.com/histograms-in-image-processing-with-skimage-python-be5938962935
- https://chrischow.github.io/365DaysOfDS/cheatsheets/skimage-cheatsheet
- remove_small_objects - skimage - Python documentation - Kite
- Module: measure — skimage v0.19.0.dev0 docs (scikit-image.org)
- Module: morphology — skimage v0.18.0 docs (scikit-image.org)

Task 4:

- https://scikit-image.org/docs/dev/api/skimage.filters.rank.html#skimage.filters.rank.equalize
- https://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_local_equalize.html
- https://scikit-image.org/docs/dev/api/skimage.morphology.html#skimage.morphology.disk
- https://www.kite.com/python/answers/how-to-show-two-figures-at-once-in-matplotlib-in-python
- https://www.w3schools.com/python/matplotlib_labels.asp
- https://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_equalize.html
- https://www.geeksforgeeks.org/how-to-calculate-and-plot-a-cumulative-distribution-function-with-matplotlib-in-python/
- Module: exposure — skimage v0.19.0.dev0 docs (scikit-image.org)
- numpy.cumsum — NumPy v1.21 Manual