CS 474 Image Processing and Interpretation

Fall 2021 - Dr. George Bebis

Programming Assignment 1 Report

Written by Joseph Trierweiler and Logan Leavitt

Joseph was responsible for Parts 1, and 4

Logan was responsible for Parts 2, and 3

Cover Page - 5/5
Theory - 16/20
Implementation - 9/10
Results - 48/50
Code - 5/5
Organization - 10/10

Overall - 93/100
Very good choice to superimpose the histograms on the same plot for comparisons.

I suspect that if you had both gone over the report together, you would have noticed that the output histograms in Part 4 (correct) and the output histograms in Part 3 (incorrect) did not match. Make sure to go over the entire report together so that 1) you can spot each others' errors and 2) things are consistent - the size and look of the figures between the two parts varied a lot!

Theory

This assignment was centered around the PGM format and Intensity Transformations. PGM stands for 'portable gray map.' It's an image file format that utilizes a header and a body made up of data defined from its header. The first two characters define its image and storage type. The next line contains the number of columns (M) and rows (N) of an image. The final line of the header contains the quantization level (Q) of the image. Files containing functions for reading and writing .pgm files were provided for us by our professor.

Part 1 demonstrated 'image sampling,' or how many pixels are in an image. We were asked to "Write a program to change the spatial resolution from 256 x 256 to 128 x 128, 64 x 64, and 32 x 32 pixels using sub-sampling by a factor of 2, 4, and 8 correspondingly." We were also then asked to resize the sub-sampled images back up their original size in order to demonstrate the effects of the transformation. The theory for the implementation was to take every 2nd, 4th, or 8th pixel in the image, and throw out the other pixels in-between. This way we would be left with a smaller image with less samples. It would continue, however, to have the same structure as the original image.

Part 2 demonstrated 'image quantization,' the range of values used to color each pixel. It explores how the number of possible gray values affects image quality. The given images have 256 possible gray values, and in part 2 we modify those images to have quantization levels of 128, 32, 8, 2. The purpose of the exercise is to see how many details are lost at each quantization level.

Part 3 demonstrated the transformation 'histogram equalization,' which stretches the quantization levels of an image. The theory behind histogram equalization is that by redistributing the gray-level values of the image uniformly, we can improve the image's quality.

-4

This is wrong. There is no Gaussian/Normal distribution involved. Instead, there should be a cumulative distribution function used somehow.

This is especially true for low contrast images, since when histogram equalization is applied, it stretches the range of values that the image has, increasing its contrast. Histogram equalization is accomplished by using the probability density function (pdf) to compute the probabilities of each gray-level occurring and plotting it to a Gaussian distribution. This will change each pixel's gray levels depending on its relation to the curve, normally distributing them until the image's histogram resembles one that is more uniform.

Part 4 demonstrated 'histogram specification,' which stretches the quantization levels of an image in a non-uniform way. For certain images, a uniform distribution might not be the most effective way to enhance details in an image. Histogram specification allows one to modify the histogram of an image to approximate any other given histogram. To achieve histogram specification, histogram equalization is used. Suppose we have transformations T and G which equalize their appropriate histograms. To modify the input histogram to approximate the specified histogram, we apply $G^{-1}(T(s))$ to each pixel. This works since they are both equalizing transformations.

I'm not sure this makes sense.

Expand on this a bit - why is it important that they are both equalizing transformations?

Implementation

The CPP files showing the implementation of reading and writing of PGM files were primarily written by Professor Bebis. These files include image.cpp, image.h, ReadImage.cpp, ReadImageHeader.cpp, and WriteImage.cpp. The methods from these files were used extensively for each part, so it was pertinent that there was a high level description for the ones we most commonly used. Each PGM image was stored as an 'ImageType' object containing the information given to us by the PGM file format. This information includes the image's dimensions (M and N) and the image levels (Q). It also includes the Image's data for each pixel, such as x-y coordinates and what gray-level the pixel is. Therefore, when trying to transform an image, one would create an ImageType object, read the image, transform the image using the

ImageType object's get and set functions, and then write the image back into being a pgm file. The following paragraphs describe the implementations of the transformations.

Part 1's transformation was to reduce the spatial resolution of an image, and then resize the image back to the original size. The implementation of the code can be found in the 'prob1' folder and in the 'Sampling.cpp' file. The image is read, and then traversed via a nested for loop. Starting at the upper leftmost pixel (0x0), the loop checks if both of the indices of the image are evenly divisible by the given factor with a modulo operator. Pixels that are not, are overwritten by a previous pixel gray-level value. The newly overwritten file is then output into a .pgm file specified by the user.

Part 2's transformation modified the number of quantization levels of the image. While one could do this by modifying the Q value in the .pgm format, we instead keep Q=256 and manually compress the number of grey values. For 128 gray levels, we use the gray levels 0, 2, 4, 6, ..., 254. For 64 gray levels, we use the gray levels 0, 4, 8, ..., 252, and so forth. This is accomplished by reducing each pixel value to the closest gray value below it (e.g 7 -> 4 in the case of 64 gray levels). The implementation of this procedure is found under the 'prob2' folder in the 'prob2.cpp' file.

Part 3's transformation was to use histogram equalization to stretch the quantization of a given image. The implementation of the code can be found in the 'prob3' folder and in the 'Equalize.cpp' file. Creating the histogram of an image is given by the createHistogram function. It takes in an image, and then plots its pixel values into a 1D int array. Every time a pixel value's gray level is read, it's corresponding level in the 1D array is iterated by 1. This creates a histogram containing every pixel's gray level value. Equalizing a histogram is done with the equalizeHistogram function. It equalizes the given histogram by iteratively applying the

-1 probability density function formula and then writing its output to a new 1D array containing the equalized histogram. This equalized histogram can then be used to redistribute the gray levels

Which formula is this? Formulae mentioned in implementation would be good candidates to include in the theory section.

I think this is missing a step. You should not be directly calculating the equalized histogram. Instead, apply a pixel transformation to the input image, and then recalculate the histogram of the resultant image. This will be the equalized histogram.

of a given image by taking each pixel value from the old image and then overwriting it with the index of the pixel's gray level in the equalized histogram array.

Part 4's transformation takes two images, and transforms the first image so that it's histogram resembles the second image. To accomplish this, some code is reused from Part 2. Specifically, the code for creating an image histogram and the transformation is used. Because of this, many of the details of implementation are similar to Part 2. As described in the Theory section, we use the equalizing transformations T, G for the two histograms to create a new transformation $G^{-1}T$. Constructing the final transformation is the main piece of new functionality introduced in Part 4. In an ideal scenario, we can compute $r = G^{-1}(T(s))$ by manually searching for an r such that $G(r) = T(s)$. However, it is not always the case that such an r exists when working with discrete data. Instead, we find the closest approximation by searching for r that minimizes $|G(r) - T(s)|$, which is implemented with linear search. The code that implements this is found in the folder 'prob4' and in the file 'prob4.cpp'. The input images to be used are specified via the command line (e.g. "prob4 input.pgm specified.pgm")

This is a great observation, and would be good to talk about more in the theory section.

This method of finding r works, but it is probably more work than is necessary. Finding the first r (in ascending order) such that G(r) > T(s) or the last r such that G(r) < T(s) should be fine and reduces the amount of the array you will have to search through.

Results

Part 1

Figures 1 and 2 show the results of reducing the sampling of an image. These reference images were provided by Professor Bebis for demonstrating our results.

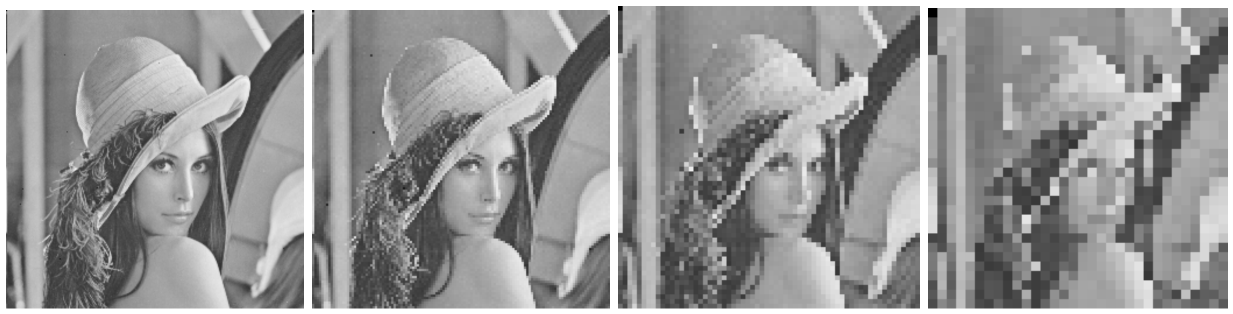


Figure 1 is the results of using 'Sampling.cpp' on the target image 'lenna.pgm' using the factors 0,2,4, and 8 respectively

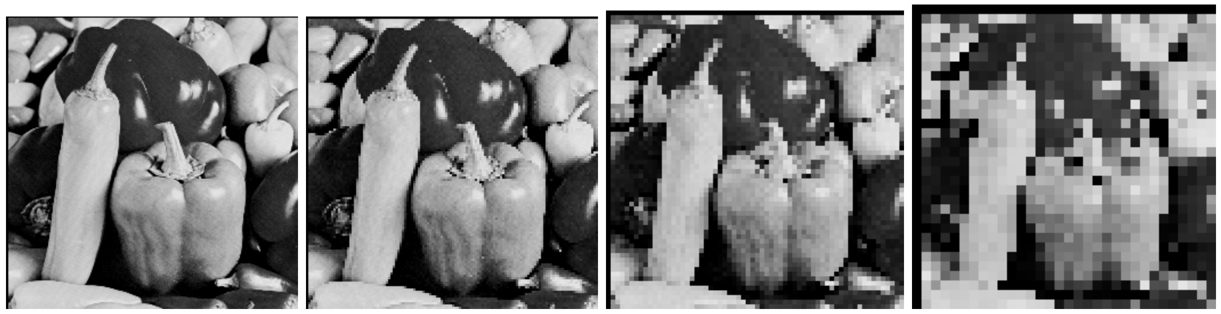


Figure 2 is the results of using 'Sampling.cpp' on the target image 'peppers.pgm' using the factors 0,2,4, and 8 respectively

As can be seen in figures 1 and 2, reducing the sampling of an image reduces the image's quality. As the sub-sampled pixels grow larger, it becomes more obvious the amount of data we've lost from the image. It becomes harder to distinguish the edges of the subject from the background in figure 1. In figure 2 it becomes almost impossible to distinguish what vegetables we're looking at by the time we get to the 8 sub-sampled image.

Part 2

The figures below showcase the results of our code for part 2. One can notice that a surprising amount of detail is maintained when the number of gray levels is reduced. While the image quality does suffer, a face can be made out in ‘lenna.pgm’ even when there are only two gray levels.

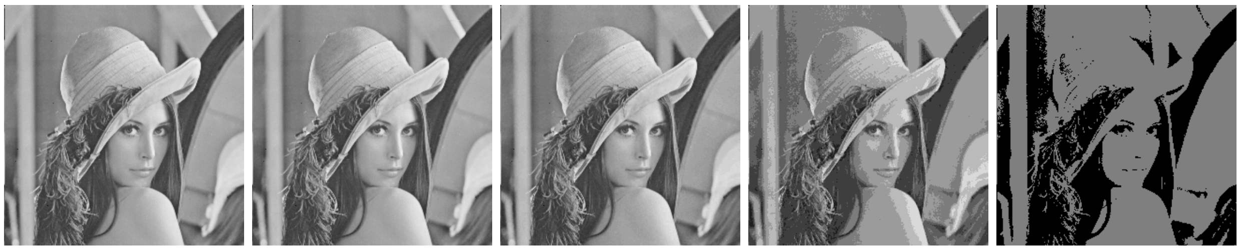


Figure 3 is the results of varying the number of gray levels in an lenna.pgm with L=256, 128, 32, 8, 2 respectively

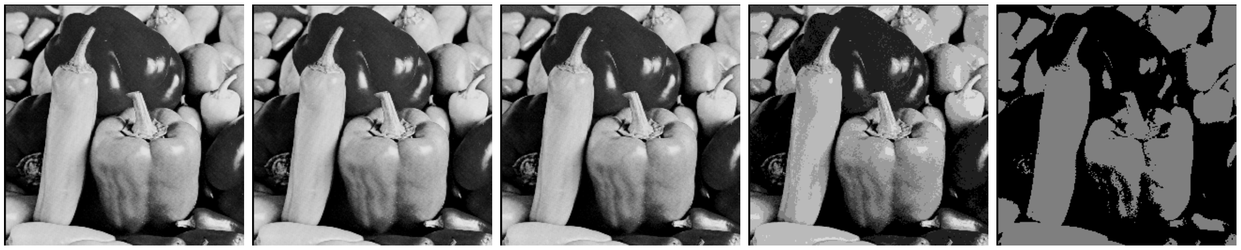


Figure 4 is the results of varying the number of gray levels in an peppers.pgm with L=256, 128, 32, 8, 2 respectively

Part 3

The figures below demonstrate the results of applying the histogram equalization transformation.

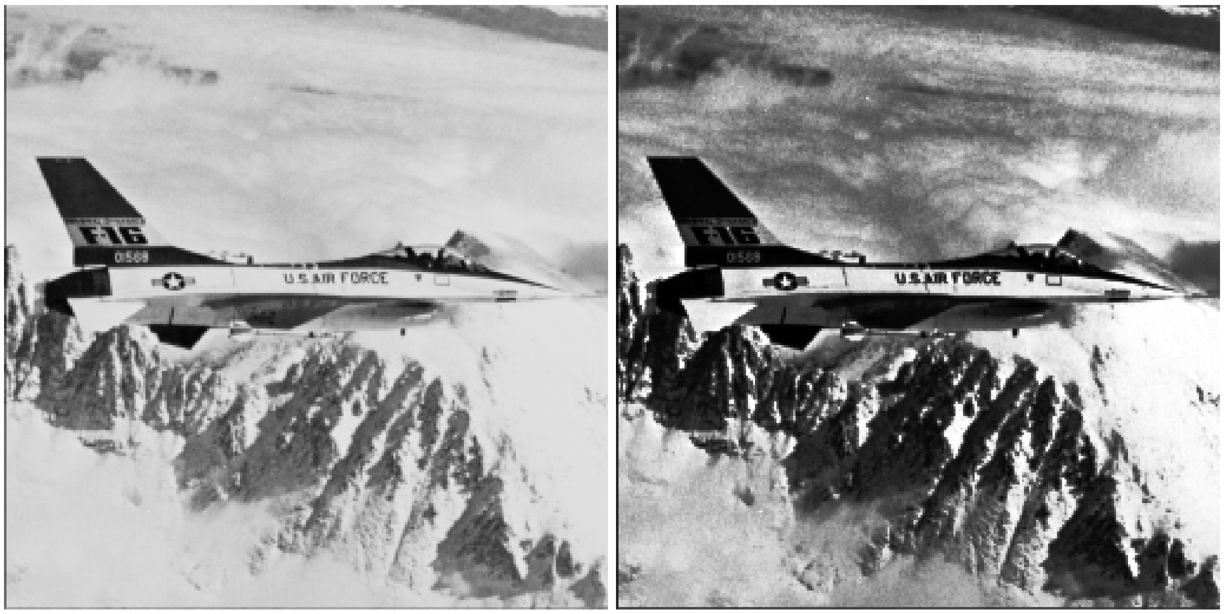
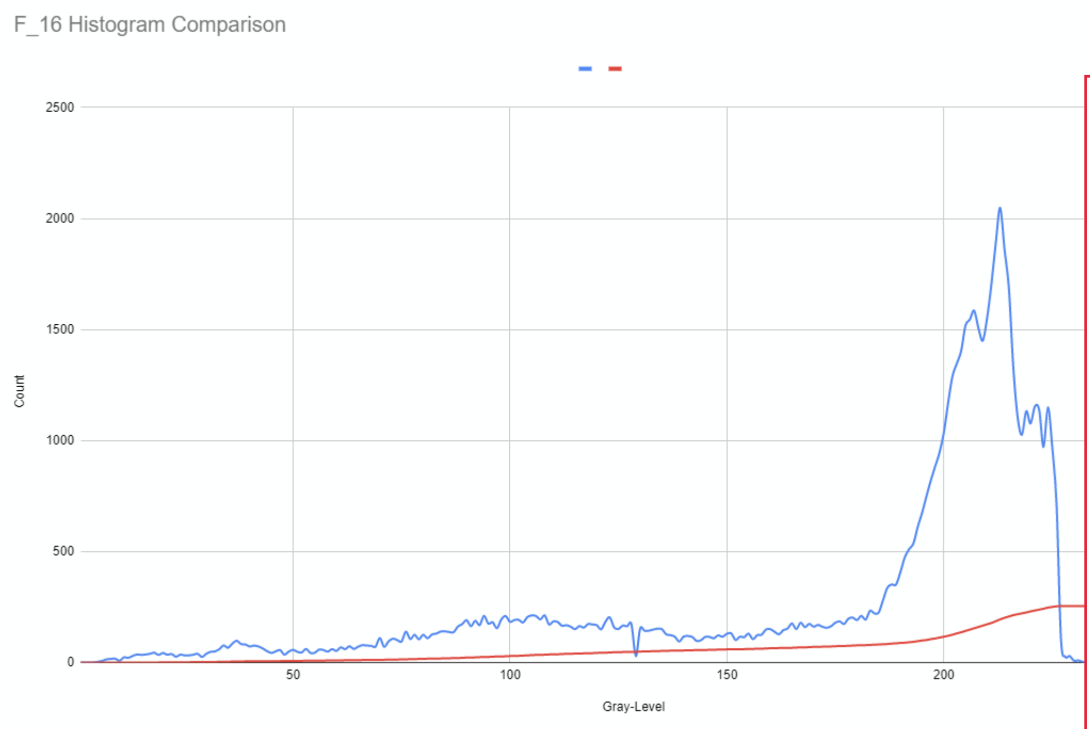


Figure 5 is the results of using ‘Equalize.cpp’ on the target image ‘f_16.pgm’ (original on the left, result on the right)

As you can see from figure 5, there is a lot more detail present in the resulting image. Especially in the background, where the clouds stop and the mountains begin. The low contrast parts of the image are now higher, which brings out details that were once lost.



-1

First of all, a line graph is a poor choice of graph for histograms. Line graphs are used to infer values in-between the ones plotted, but histograms explicitly do not have values between the ones plotted. Try using a bar chart (or finding one specifically for histograms - those are common).

Secondly, the equalized histogram does not depict a histogram. As noted in the implementation section, it looks like you computed the cumulative distribution function, which then needs to be used to apply a pixel transformation to the image. Only then can you calculate the equalized histogram from the resulting image. Equalized histograms have a major flaw in that peaks cannot become smaller, so if you superimpose the two histograms you should be able to see the same peaks , just moved elsewhere in the histogram. This is not evident here.

Figure 6 is a plot of the histogram data comparing the original image's histogram (blue) to the equalized histogra

Figure 6 explains the results of our histogram equalization. Notice how the red line, rep the grey levels used in the result from figure 5, is more evenly distributed than the origi note the areas where the red line surpasses the blue line. That unused range from the image became used, allowing for the details that lived in that range to stand out more figure 5.

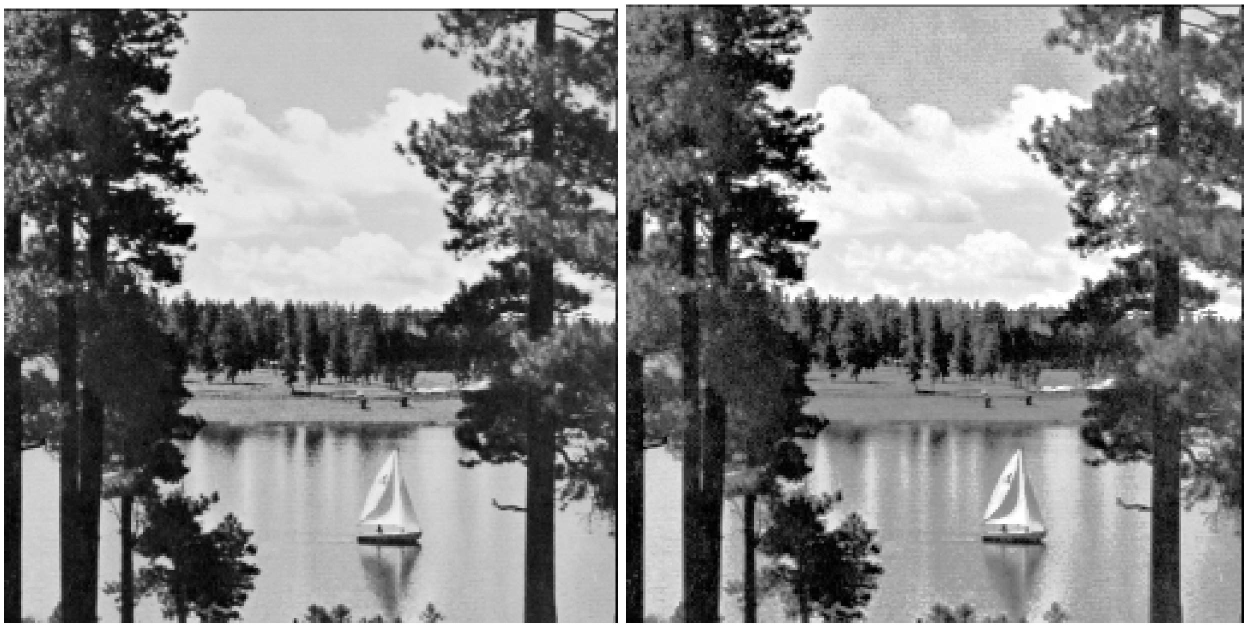
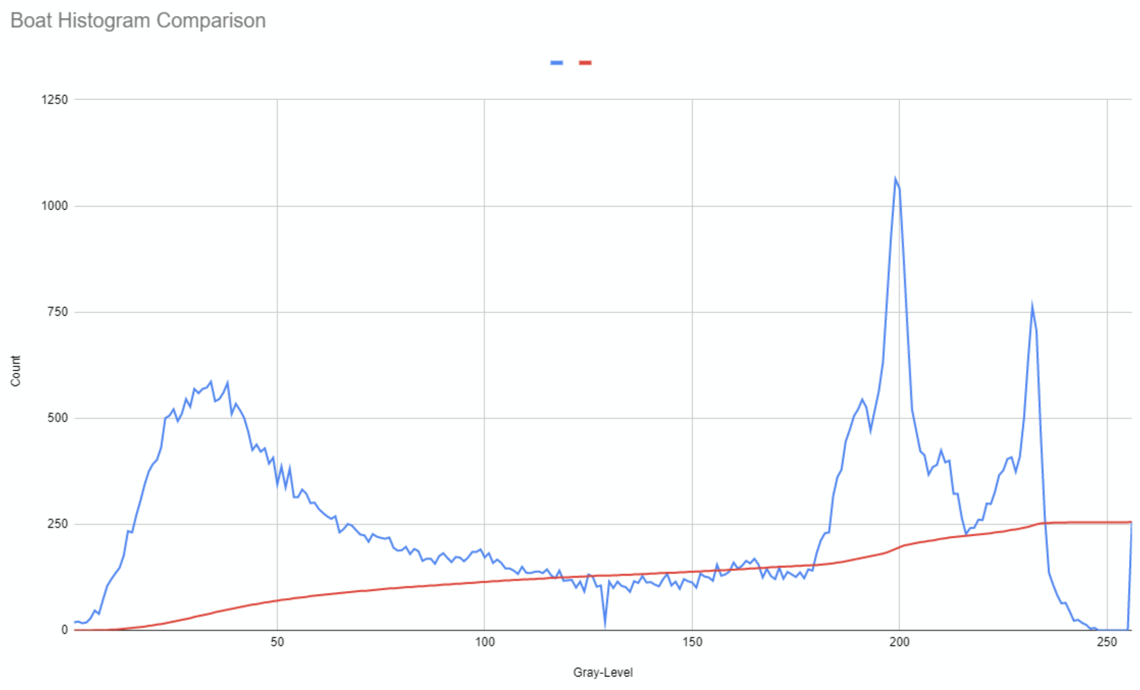


Figure 7 is the results of using 'Equalize.cpp' on the target image 'boat.pgm' (original on the left, result on the right)

Figure 7 also demonstrates the result of using histogram equalization on an image. If you look at the lake the boat rests on, you can see more details in the water and in the clouds in the result than the original image.



-1

Figure 8 is a plot of the histogram data comparing the original image's histogram (blue) to the equalized histogram (red)

Figure 8 tells a very similar story to figure 6. The red line is more linear than our original blue line, representing the equalization of the distribution of quatitization. Some of the missing levels that lived between 100 and 150 got brought back, as well as the high gray levels found above 200. These plots are useful for visualizing the stretch that occurs during histogram equalization.

Part 4

Below are the results of histogram specification. Theoretically, we would expect the resulting image to take on similar tones to the image used for the specified histogram. In both test cases, the resulting images do somewhat resemble the specified image.

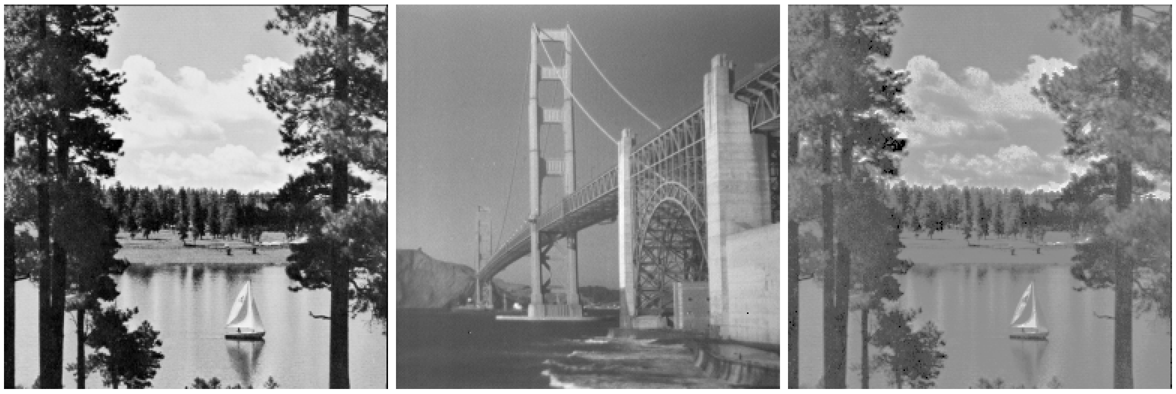
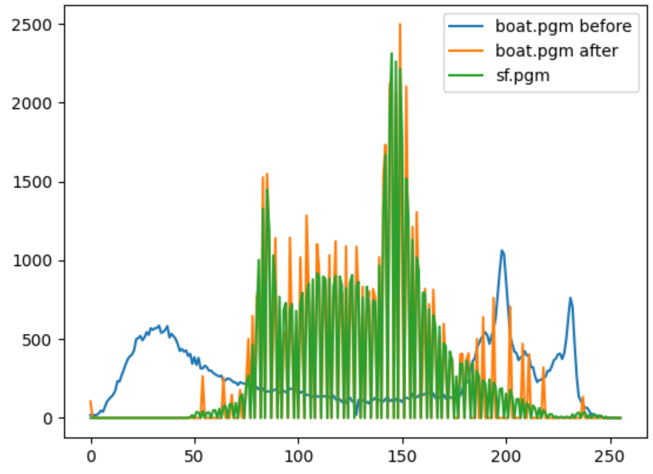


Figure 9 is the results(right) of applying histogram specification on boat.pgm(left) with sf.pgm(middle) being used for the specified histogram.

Notice how the after histograms in this section look like the input histograms, unlike in the previous section.



The choice to super-impose the histograms on the same plot was a very good one - it's very easy to see how the histogram changed by the transformation and how similar it is to the specified histogram.

However, it is held back by the choice of plot (like above) - all of those valleys are unnecessary to show and them combined with the line thickness means it is difficult to see the boat.pgm after histogram. If you used a box plot instead, you could have grouped the histograms such that each one was visible.

As well, this one is so low resolution and the line thickness is so large that it is difficult to see what is going on. Try lowering the line thickness and either export to higher resolution or to PDF (Google sheets supports exporting to PDF). Also make it a bit larger, to match the previous part.

Figure 10 showcases the corresponding histograms for boat.pgm, sf.pgm and boat.pgm after histogram specification.

Figure 10 is a clear example of histogram specification working as intended. While the original histogram for 'boat.pgm' does not resemble the histogram for 'sf.pgm', the transformed histogram does.

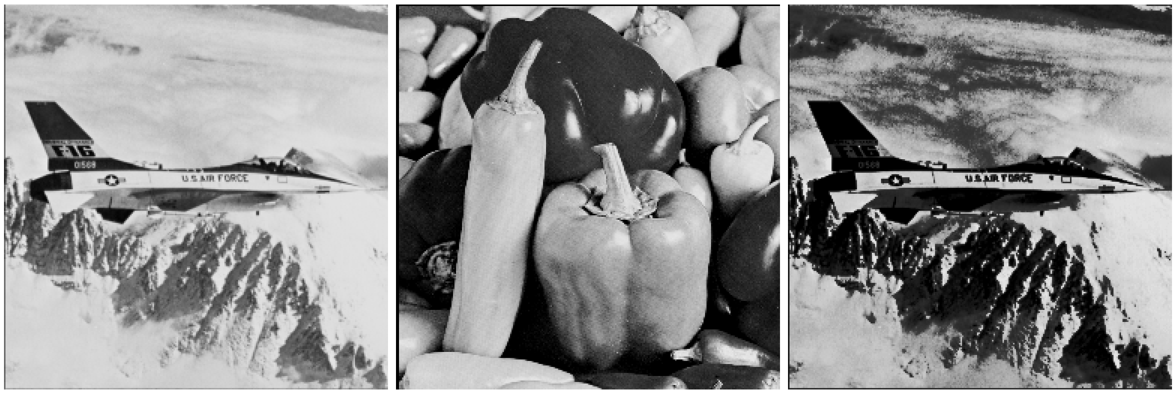
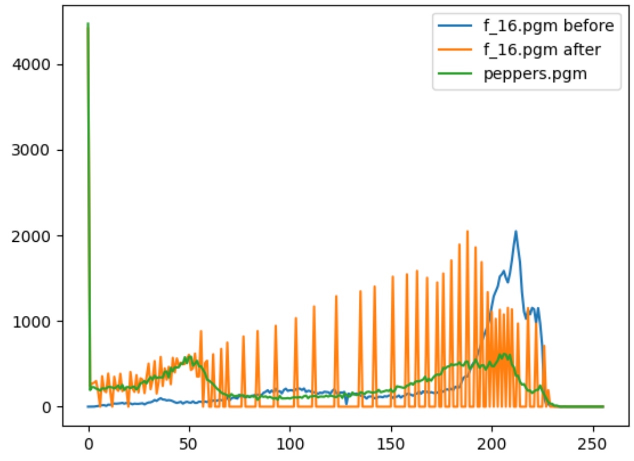


Figure 11 is the results(right) of applying histogram specification on f_16.pgm(left) with peppers.pgm(middle) being used for the specified histogram.



This has a similar problem to above - notice how you can't see the orange spike on the left because it is covered by the green spike. For me (the grader) - I am looking to see the orange spike on the left as it is a defining characteristic of a correctly specified histogram. Instead, I just have to assume it is there.

Otherwise good plot.

Figure 12 showcases the corresponding histograms for f_16.pgm, peppers.pgm and f_16.pgm after histogram specification.

In figure 12, the transformed histogram somewhat matches the shape of the specified histogram, but appears much larger because the grey values are concentrated in large spikes. This could be explained by the discrete nature of the data. However, we can still conclude that the transformed histogram still effectively models the specified histogram.