Programming Assignment

Project 1
University of Nevada, Reno
CS 474 - Image Processing and Interpretation
Professor: Dr. George Bebis, Ph.D.

Shawn Ray
Yi Jiang
Both Shawn Ray and Yi Jiang did about half of the coding and half of the report for this project.

Due: September 22, 2021

Contents

1 Theory

1.1 Image Sampling

The approach that will be discussed in this section is that of sub-sampling. Sub-sampling is a method used in image processing that involves taking an image and compressing it by a factor of its *sub-sample.* If an image is represented as a matrix of the pixel values, then we factor down said matrix by the sub-sample.

Given a matrix M, populated with arbitrary values and a dimension of 4x3:

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 \end{bmatrix}$$

A reduction of the size of this image can be produced by sub-sampling. For example, sub-sampling this matrix by 2 would mean that every 2 pixels are taken into the final matrix N.

$$N = \begin{bmatrix} 1 & 3 \\ 7 & 9 \end{bmatrix}$$

The matrix N would be the result after sub-sampling M by a factor of 2, giving a final dimension of 2x2. In essence, the new matrix's number of rows, X, and number of columns, Y, are derived from the division of the original matrix's X and Y by the sub-sampling factor.

Sub-sampling is a method of image compression with loss, as pixels within the matrix are discarded. Resizing of the matrix would result in losses to the quality of the image.

1.2 Image Quantization

Image Quantization is the method to which a *PGM* image can receive different gray levels. Recall that a PGM image stands for *portable gray map.* A PGM image will consist of a header; this header will contain information regarding what type of format, the rows and columns, as well as the maximum quantization value. This maximum level is the gray level of the image. It denotes how gray a PGM image can be. Image Quantization modifies this maximum level.

Take a matrix M denoted by

$$M = \begin{bmatrix} 135 & 2 & 35 & 41 \\ 37 & 47 & 115 & 216 \\ 172 & 38 & 19 & 100 \end{bmatrix}$$

Assume M would have the standard gray level of 255 as the maximum expression that these pixels can have. This gray level can be modified down to a lower level by a process known as Quantization.

Every pixel has a level known as the *bpp.* The bpp is the number of bits-per-pixel required to store said pixel. L determines the number of quantization levels, where $k = bpp$ and is represented by the power of 2:

$$L = 2^k \tag{1}$$

The range of pixel values is $[0, L-1]$. In a PGM Image, the maximum quantization level, denoted by Q, is represented by:

$$Q = L - 1 \tag{2}$$

Through the process of quantization, the new quantization level Q for the image will be the current maximum gray level for the image matrix. Let us denote p_i as the original pixel value from the PGM image and p_i' as the new quantized pixel value. This pixel value must be an integer. In addition, k denotes the original bpp and k' denotes the new bpp.

$$p_i' = \frac{p_i}{L_{old}/L_{new}} = \frac{p_i}{2^{k-k'}} \tag{3}$$

The denominator shows us the number of levels from the original divided by the number of levels for the new image, or 2^{8-k}, assuming the standard PGM image has a maximum gray value of 255 and $L_{old} = 2^8 = 256$ was the number of levels from the original image.

To show this in action, let us work with our matrix M. Using Equation 1, we apply a quantization with bpp of 7, giving us the number of gray levels in our new image:

$$L = 2^7 = 128$$

Recall that levels start from 0, therefore applying a level of 127 instead of 128. Then, the quantization level, according to Equation 2, is given:

$$Q = 128 - 1 = 127$$

Applying quantization on the matrix, M would become

$$M = \begin{bmatrix} 67 & 1 & 17 & 20 \\ 18 & 23 & 57 & 108 \\ 86 & 19 & 9 & 50 \end{bmatrix}$$

where the pixel value 67 was calculated using Equation 3:

$$\frac{135}{2^{8-7}} = \frac{135}{2} = 67.5 = 67$$

So, the pixel values range from $[0, 1, 2, ..., 127]$ with $Q = 127$. However, if we want our maximum quantization level to still be 255 and have pixel values range from $[0, 2, 4, ..., 254]$, then we could take our matrix M and perform the following calculation:

$$p_i' = p_i * 2^{k-k'} \tag{4}$$

As such, M, with only 128 levels instead of 256, would become

$$M = \begin{bmatrix} 134 & 2 & 34 & 40 \\ 36 & 46 & 114 & 216 \\ 172 & 38 & 18 & 100 \end{bmatrix}$$

After applying the transformation to the original matrix, we are given a new matrix that is a quantization of the original.

The below shows an original matrix N from $L = 256$ with $Q = 255$ quantized to $L = 2$ with $Q = 1$ and finally, $L = 2$ with $Q = 255$.

$$N = \begin{bmatrix} 45 & 81 & 29 & 154 \\ 129 & 117 & 36 & 77 \\ 100 & 135 & 1 & 8 \\ 25 & 67 & 254 & 191 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 128 \\ 128 & 0 & 0 & 0 \\ 0 & 128 & 0 & 0 \\ 0 & 0 & 128 & 128 \end{bmatrix}$$

1.3 Histogram Equalization

A histogram is a graphical representation of a set of data. In the topic discussed here, the histogram is used to represent the frequencies of each gray level found in the PGM image. This is found by taking in each pixel. Recall that an image is just a matrix with pixel values associated to each row and column; where the frequencies are how often that pixel value occurs.

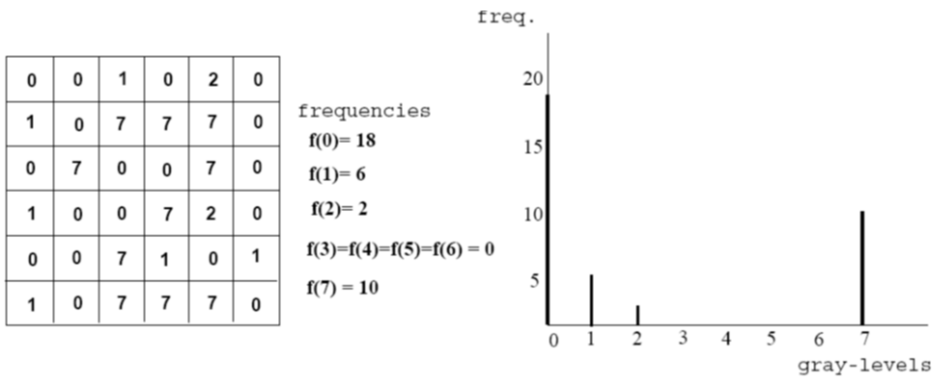


Figure 1: Histogram representation of the frequencies of a sample image.

As can be seen in Figure 1, the pixel value of 7 occurs 10 times, therefore its frequency is 10 and this is what the histogram represents. Histogram equalization attempts to *equalize* or level out the frequencies in more normal distributions. To find the equalized histogram data, one would sum over all the probabilities obtained by taking the value divided by the dimensions of the input image. In essence it follows:

$$P_r(r_k) = n_k/(MN) \tag{5}$$

where M is the number of rows and N is the number of columns of the input matrix. Given a 64x64 image, MN would be $64 * 64 = 4096$.

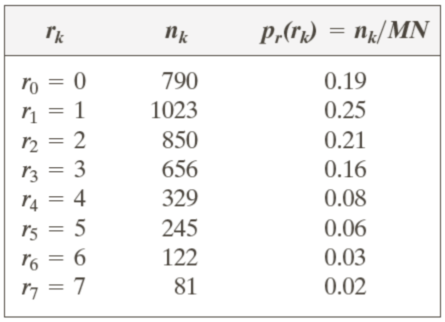
To perform histogram equalization, we have to compute for the transformation by the $s = T(r)$ function. The below shows how to get s for each gray level k.

$$s_k = \sum_{i=0}^{k} P_r(r_i) \tag{6}$$

Let us work with a sample 3 bit 64x64 image to show the technique at hand. Figure 2 shows us $r_k =$ the gray levels, $n_k =$ the frequencies for each gray level, and $P_r(r_k) =$ the probabilities of each frequency out of the total number of pixel data.

To use Equation 6, we will show the value of $s_0, s_1,$ and s_2:

$$\begin{aligned}
s_0 &= P_r(r_0) = 0.19 * 7 = 1.35 = 1 \\
s_1 &= P_r(r_0) + P_r(r_1) = (0.19 + 0.25) * 7 = 3.10 = 3 \\
s_2 &= P_r(r_0) + P_r(r_1) + P_r(r_2) = (0.19 * 0.25 * 0.21) * 7 = 4.55 = 5
\end{aligned}$$



r_k	n_k	$p_r(r_k) = n_k/MN$
$r_0 = 0$	790	0.19
$r_1 = 1$	1023	0.25
$r_2 = 2$	850	0.21
$r_3 = 3$	656	0.16
$r_4 = 4$	329	0.08
$r_5 = 5$	245	0.06
$r_6 = 6$	122	0.03
$r_7 = 7$	81	0.02

Figure 2: A 3 bit 64x64 image with 8 gray levels [0,7] and the probabilities of each frequency.

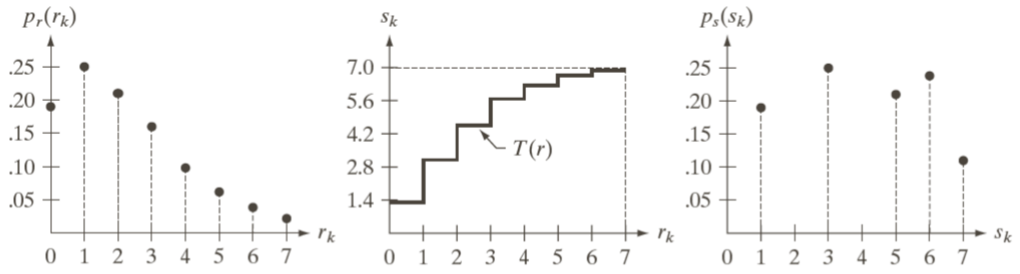


Figure 3: The process of using an input histogram and applying s=T(r) to get the equalized histogram.

This results in $s = [1, 3, 5, 6, 6, 7, 7, 7]$. Now, we just need to look at $P_r(r_k)$ and use it on $P_s(s_k)$. For example, $r_0 = 0, s_0 = 1, P_s(s_0) = 0.19$, but when $s_3 = s_4 = 6$, we need to get $P_s(s_k) = P_r(r_3) + P_r(r_4) = 0.16 + 0.08 = 0.24$.

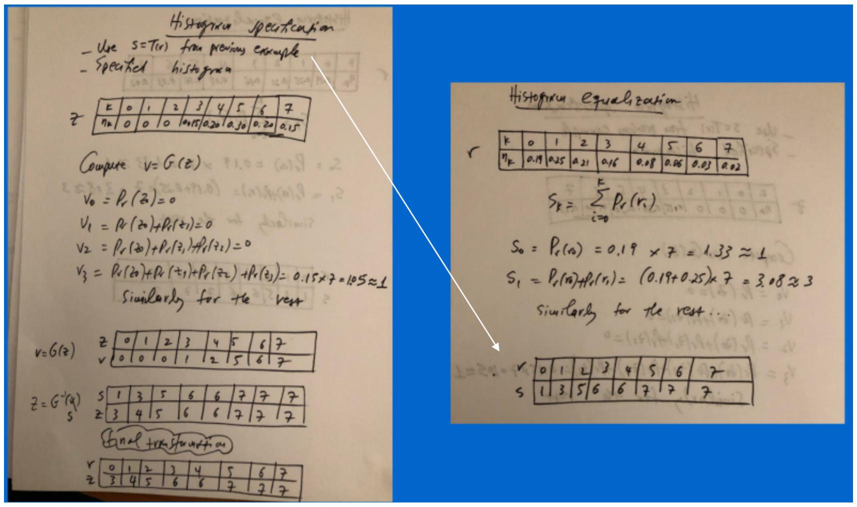
In the end, we get Figure 3 by computing all of $P_s(s_k)$, in which we use to plot the new histogram.

1.4 Histogram Specification

Histogram equalization yields an image with a uniform probability distribution function, yet sometimes non-uniform probability distribution functions can yield better results. This brings up the topic of Histogram Specification. As the name suggests, histogram specification attempts to find a histogram to match a specific histogram, unlike histogram equalization which just equalizes the frequencies present in the histogram.

Histogram specification follows off the results of histogram equalization by computing $s = T(r)$ to get the s array. It also takes in a specified histogram as input which we denote as z and compute $v = G(z)$. After that, we have to compute $z' = G^-1(s)$. This computation uses v and s. For example, when $s_0 = 1$ and we want to get z'_0, we want to look at which index in z gives us 1 in v. It turns out that happens when $z_3 = 1$, so we will put 3 into z'_0.

Figure 4 demonstrates how one would compute the resulting histogram from the data given a specified target. In Figure 5, the upper left histogram comes from the original image. The upper right image shows the specified histogram. The lower left shows the computation of $v = G(z)$. The lower right shows the resulted histogram after applying our previous $s = T(r)$ into $z' = G^-1(s)$.



Histogram specification

- Use $s = T(r)$ from previous example
- Specified histogram

z

k	0	1	2	3	4	5	6	7
n_k	0	0	0	0.15	0.20	0.30	0.20	0.15

Compute $v = G(z)$

$v_0 = P_r(z_0) = 0$

$v_1 = P_r(z_0) + P_r(z_1) = 0$

$v_2 = P_r(z_0) + P_r(z_1) + P_r(z_2) = 0$

$v_3 = P_r(z_0) + P_r(z_1) + P_r(z_2) + P_r(z_3) = 0.15 \times 7 = 1.05 \approx 1$

Similarly for the rest

$v = G(z)$

z	0	1	2	3	4	5	6	7
v	0	0	0	1	2	5	6	7

$z = G^{-1}(s)$

s	1	3	5	6	6	7	7	7
z	3	4	5	6	6	7	7	7

Final transformation

r	0	1	2	3	4	5	6	7
z	3	4	5	6	6	7	7	7

Histogram equalization

r

k	0	1	2	3	4	5	6	7
n_k	0.19	0.25	0.21	0.16	0.08	0.06	0.03	0.02

$$s_k = \sum_{i=0}^{k} P_r(r_i)$$

$s_0 = P_r(r_0) = 0.19 \times 7 = 1.33 \approx 1$

$s_1 = P_r(r_0) + P_r(r_1) = (0.19 + 0.25) \times 7 = 3.08 \approx 3$

Similarly for the rest...

r	0	1	2	3	4	5	6	7
s	1	3	5	6	6	7	7	7

Figure 4: Histogram specification demonstrated by hand. The right is histogram equalization prior to specification.

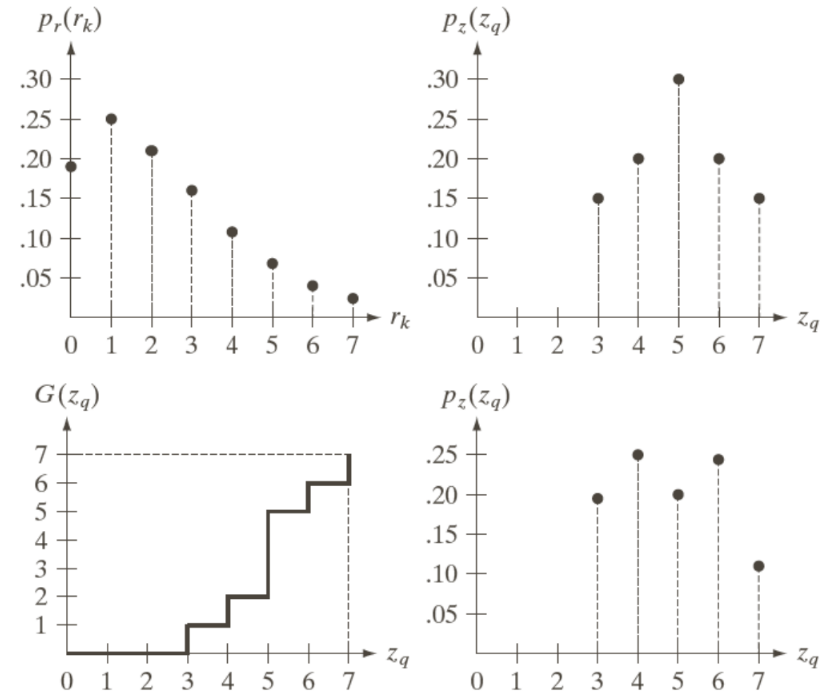


Figure 5: A standard histogram representation of a image on the left, and its target specific histogram on the right.

2 Implementation

2.1 Image Sampling

The implementation of Image Sampling will follow that described in the theory section of this document. Recall that an image is just a matrix of stored pixel values, these pixel values will not change.

The main program declares $N = M = L = 256,$ and $Q = 255.$ We work with the images `"lenna.pgm"` and `"peppers.pgm"`, so the program calls `ImageType` class to instantiate the two image objects with N, M, Q.

The `imageSampling` function is called, taking in the image filename, the `ImageType` object, and the sub-sampling factor. This function calls `readImageHeader` to store N, M, Q and `readImage` to read in the data contents of the image file, which now our `ImageType` reference has actual data. The program makes another `ImageType` object for the sub-sampled image by dividing M and N by the factor - let us call it sf - but keeping the Q.

Sub-sampling is performed in a nested loop that iterates every sf times and increments the new image's count, using `getPixelVal` from the original image and setting it to the new image using `setPixelVal`. The last step simply uses `writeImage` to write the new image.

Since the assignment for this part also asks for resizing the image back to 256x256, that is also done with the call to the `imageSizing` function. This function does the same thing the `imageSampling` function does in the first part, but instead assigns M and N by multiplying the size factor - let us call it sf. To perform resizing, a nested loop was used where we get the original image's pixel, which was taken from the current position multiplied with sf, and setting it to the new image's current pixel.

2.2 Image Quantization

The implementation of Image Quantization will follow that described in the theory section of this document. Recall that an image is just a matrix of stored pixel values, these pixel values will not change.

The main program declares $N = M = L = 256,$ and $Q = 255.$ We work with the images `"lenna.pgm"` and `"peppers.pgm"`, so the program calls `ImageType` class to instantiate the two image objects with N, M, Q.

The `imageQuantization` function is called, taking in the image filename, the `ImageType` object, the bpp, and the mode. In here, when `mode=0`, we are quantizing it to $Q = L - 1$, but `mode=1` will quantize to $Q = 255$.

Inside `imageQuantization`, it fills data into the `ImageType` reference after calling `readImage`. The quantization process is performed with a nested loop that gets the original pixel value and, depending on which mode was passed in, computes Equation 3 or Equation 4 to get the quantized value. This value is set to the new image. The last step calls `writeImage` to write the new image.

2.3 Histogram Equalization

The implementation of Histogram Equalization will follow that described in the theory section of this document. Recall that an image is just a matrix of stored pixel values, these pixel values will not change.

The main program declares $N = M = L = 256$, and $Q = 255$. We work with the images `"boat.pgm"` and `"f_16.pgm"`, so the program calls `ImageType` class to instantiate the two image objects with N, M, Q. Arrays of `double` type are declared to store the $P_r(r_k)$ for each of the images.

The `getHistogram` function is called, taking in the image filename, the `ImageType` object, and its assigned $P_r(r_k)$ array. This function mainly reads data by calling `readImage` and stores the frequencies of each gray level in the image. The frequencies are then used to get the probabilities, which ultimately is stored into the passed in probability array.

Now that the specific `ImageType` object has actual data and its $P_r(r_k)$ array has probabilities, the program calls `equalizeImage`, which uses the same arguments. This function computes $s = T(r)$, and the process goes according to the steps defined in the theory section.

It would grab the probability of the current element and accumulate it with the previous probabilities. However, they should also be multiplied with the number of levels in the image. The stored result is rounded and goes to the s array. The probability would be grabbed from the same index and be placed into the $P_s(s_k)$ array.

The program also calls `printHistogram` that accepts the filename and the probability array. This will generate a text file with gray levels and the probabilities, separated by commas.

Finally, the program has a nested for loop to equalize the original image. It gets the original pixel value and uses that as an index in the s array to set the new image pixel. This image gets written out.

2.4 Histogram Specification

The implementation of Histogram Specification will follow that described in the theory section of this document. Recall that an image is just a matrix of stored pixel values, these pixel values will not change.

The main program declares $N = M = L = 256$, and $Q = 255$. We work with the images `"boat.pgm"`, `"f_16.pgm"`, `"sf.pgm"`, and `"peppers.pgm"`, so the program calls `ImageType` class to instantiate the four image objects with N, M, Q. Arrays of `double` type are declared to store the $P_r(r_k)$ for each of the images.

Like histogram equalization, the `getHistogram` function is called for each `ImageType` object before calling the `specifyImage` function, but this time it also accepts a parameter for the specified image's probabilities. The first part of `specifyImage` does the same thing `equalizeImage` does because we need to compute $s = T(r)$.

The next step computes for $v = G(z)$. This performs the same steps as $s = T(r)$ but instead accumulates the probabilities of the specified image and stores to the v array.

The next step computes for $z' = G^{-}1(s)$. A nested for loop calls the `match` function that returns the closest match of s_k in v to v_k. v_k is used on a call to `indexOf`, which grabs the index of v_k in z to z_k. After this call, that result is saved to the z' array. The probabilities are taken.

`printHistogram` is called three times to generate the original histogram, the specified histogram, and the resulting histogram. The image is also specified through getting the original pixel value and uses that as an index in the z' array to set the new image pixel. The image is then written out.

3 Results and Discussion

3.1 Image Sampling

Figure 6 shows the results of the sub-sampling procedure applied to the `"lenna.pgm"` image. Figure 7 shows the results of the sub-sampling procedure applied to the `"peppers.pgm"` image.

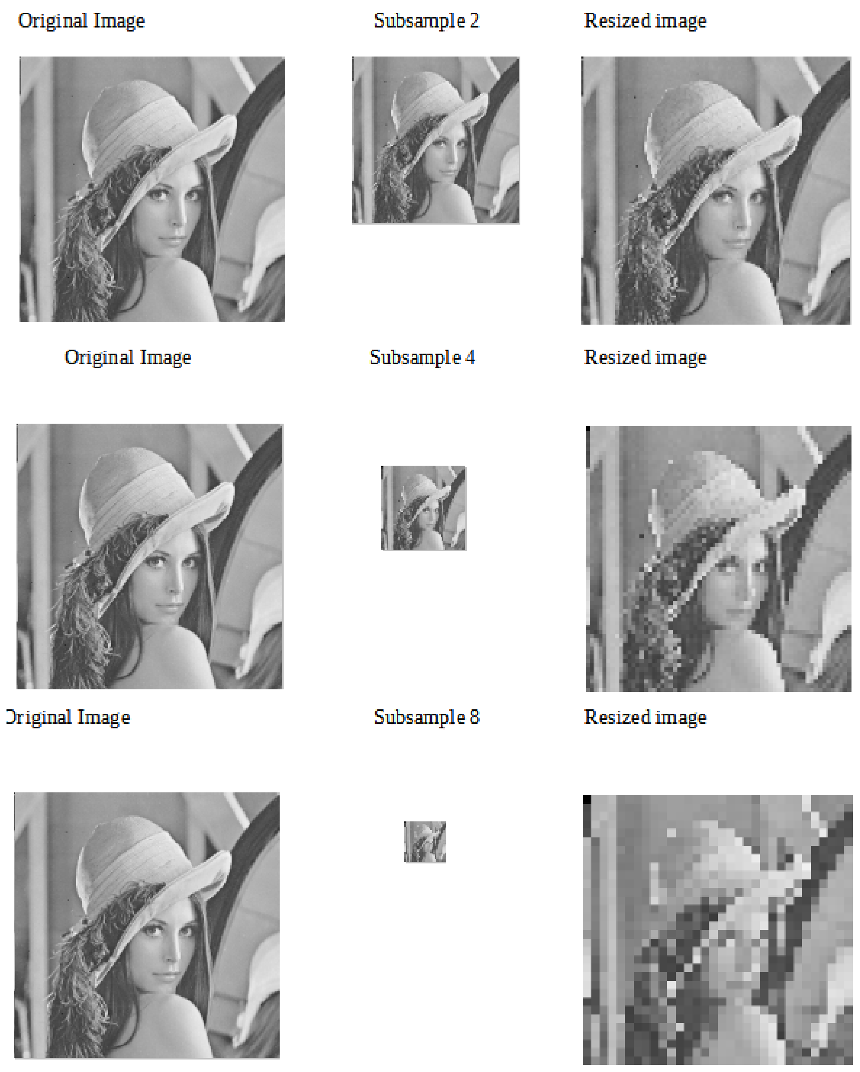


Figure 6: The results of sub-sampling the *lenna* image under 4 different factors and resized back to 256x256.

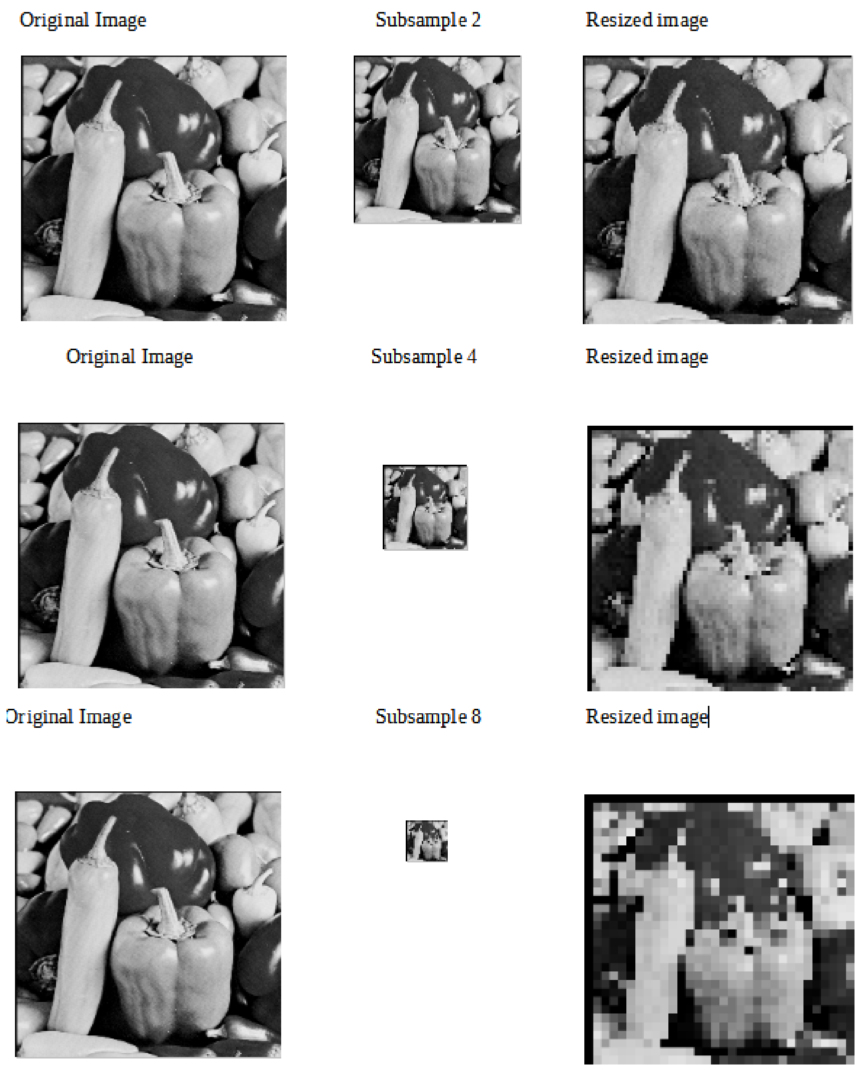


Figure 7: The results of sub-sampling the *peppers* image under 4 different factors and resized back to 256x256.

A major finding about image sampling on a PGM image was that we needed only a sub-sample factor to reduce the rows and columns. It was interesting to see that we only needed to take x many pixels from the original image and then had them on a smaller image size. The images did not seem to lose any major details too.

Resizing the image was challenging as well, but it turned out that it was possible by filling in the same values x many times to the right and to the bottom (forming a square before the next set of values), which was retrieved by calculating the current M position multiplied by a factor of the sub-sample.

3.2 Image Quantization

Figure 8 shows the results of the Image Quantization procedure applied to the "lenna.pgm" image. Figure 9 shows the results of the Image Quantization procedure applied to the "peppers.pgm" image.

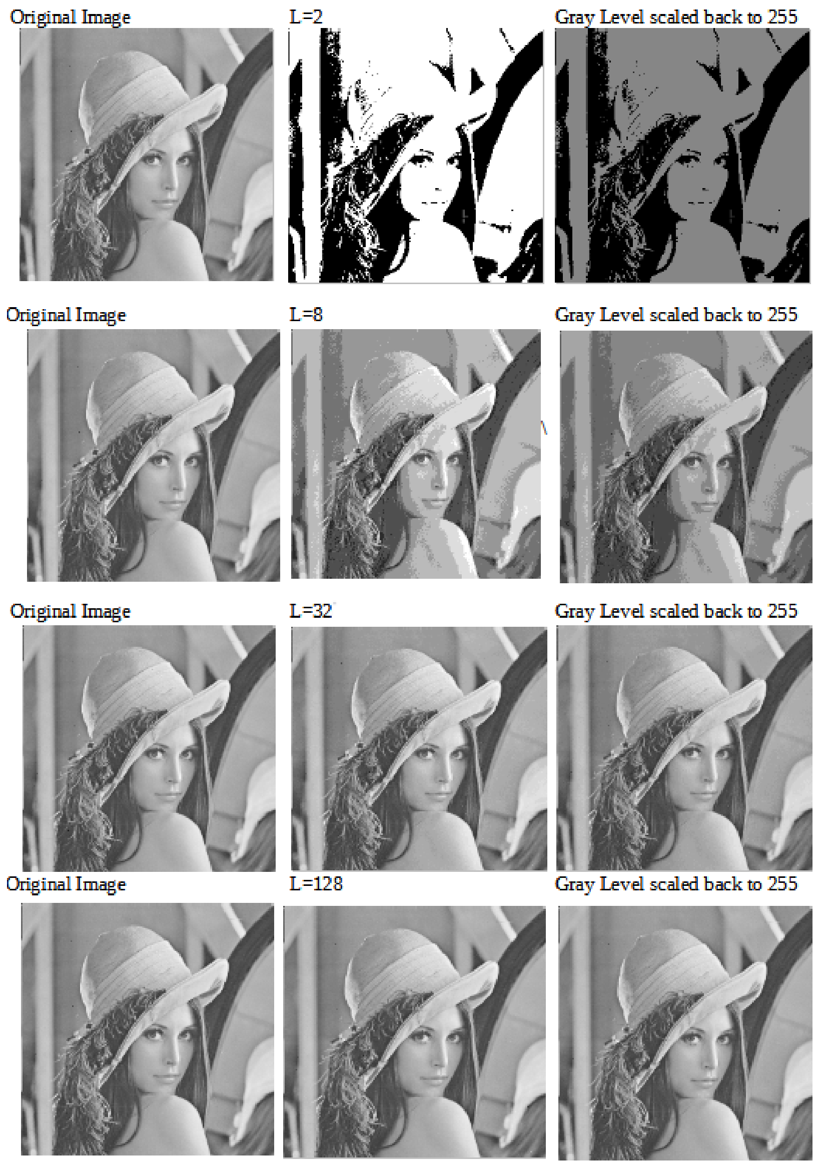

Original Image
L=2
Gray Level scaled back to 255
Original Image
L=8
Gray Level scaled back to 255
Original Image
L=32
Gray Level scaled back to 255
Original Image
L=128
Gray Level scaled back to 255

Figure 8: The results of applying quantization to the *lenna* image under 4 different levels, as well as using that number of levels with a maximum quantization level of $Q = 255$.

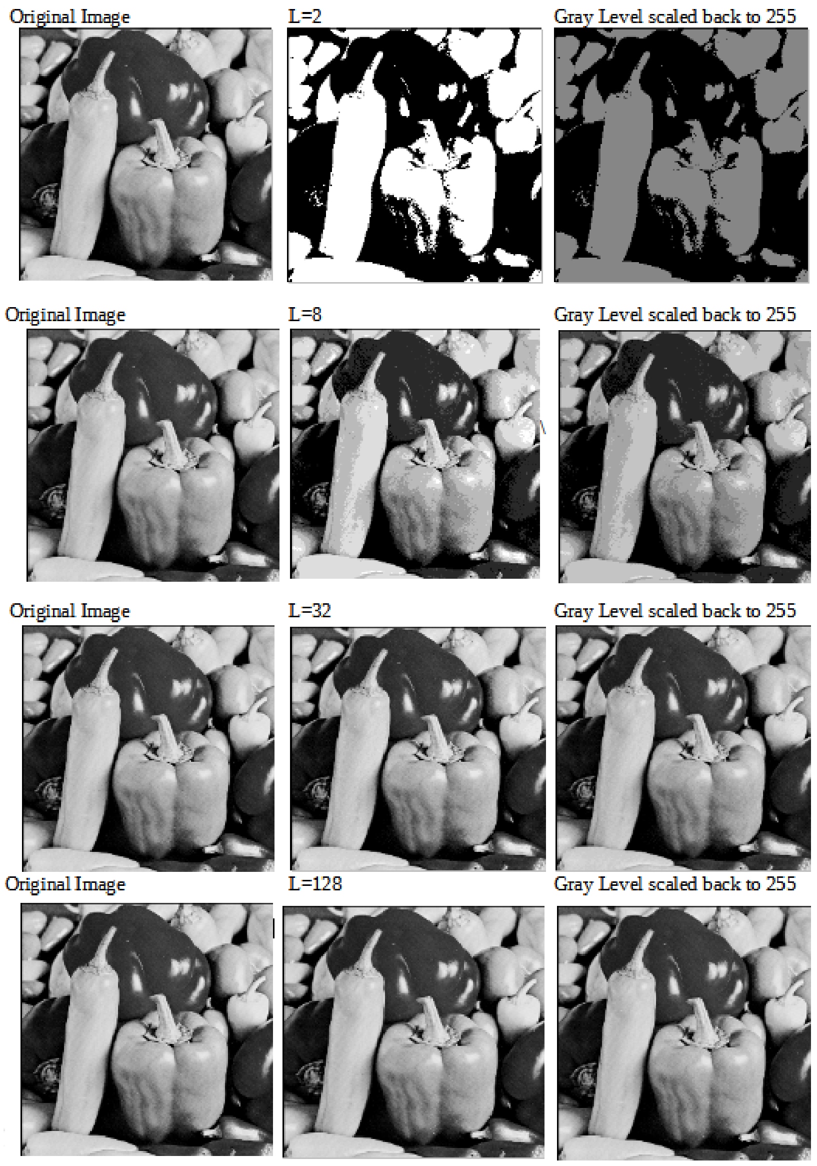

Original Image
L=2
Gray Level scaled back to 255
Original Image
L=8
Gray Level scaled back to 255
Original Image
L=32
Gray Level scaled back to 255
Original Image
L=128
Gray Level scaled back to 255

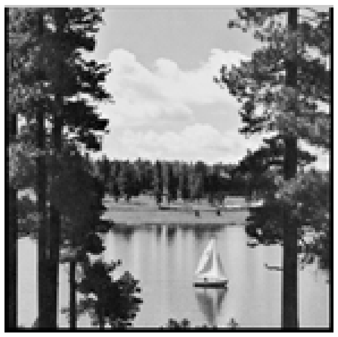
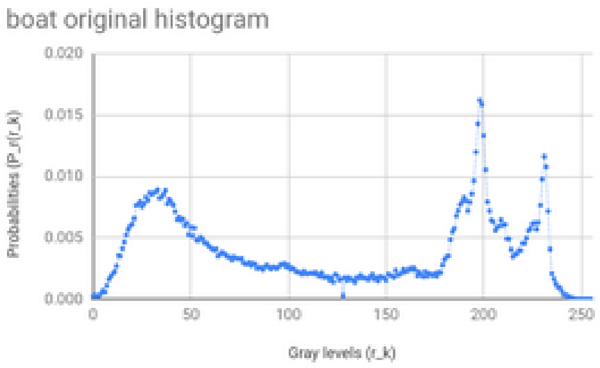
Figure 9: The results of applying quantization to the *peppers* image under 4 different levels, as well as using that number of levels with a maximum quantization level of $Q = 255$.

The original image was quantized down to $L = 2, 8, 32,$ and 128. As can see, the second column shows all images when $Q = L - 1$. For example, when $L = 2$, it only had two gray levels: black and white. The others had a few levels spread out. The third column shows all images when $Q = 255$ instead of $Q = L-1$.

The major findings of the image quantization are to be expected. When the gray levels were $L = 2$, one would expect the image to appear to have high contrast, or not much gray. As the gray levels increased, for example, in $L = 128$, the image appeared to be closer to the original and, in turn, when scaled back would appear closer to the original.

3.3 Histogram Equalization

Figure 10 shows the equalization of the "boat.pgm" image. Figure 11 shows the equalization of the "f_16.pgm" image.

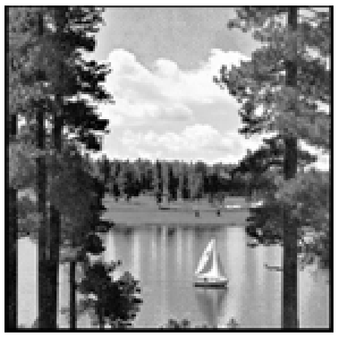
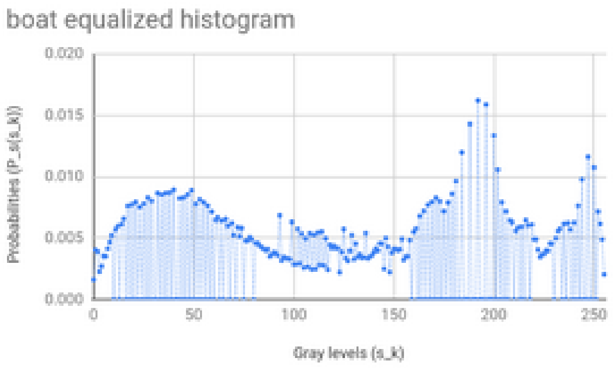
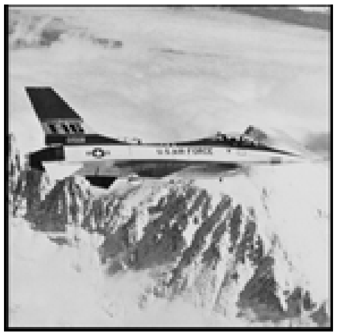
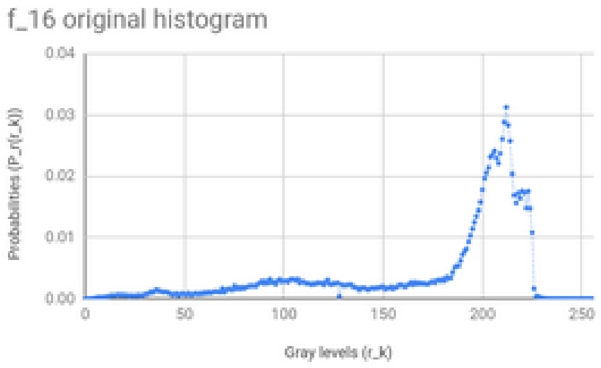



Figure 10: The original *boat* image and its histogram is displayed at the top. The equalized image and its histogram is displayed at the bottom.

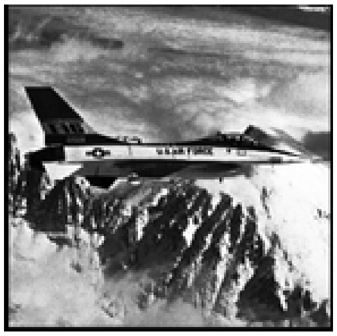
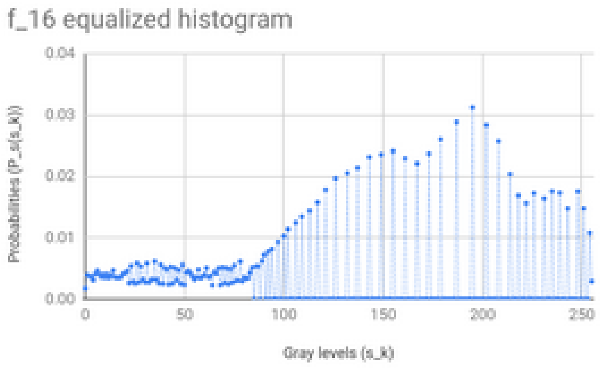



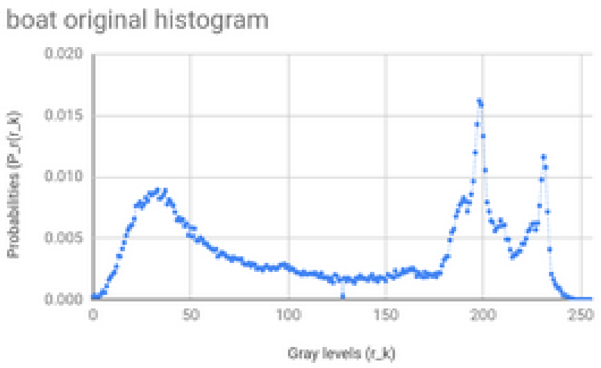
Figure 11: The original *f_16* image and its histogram is displayed at the top. The equalized image and its histogram is displayed at the bottom.

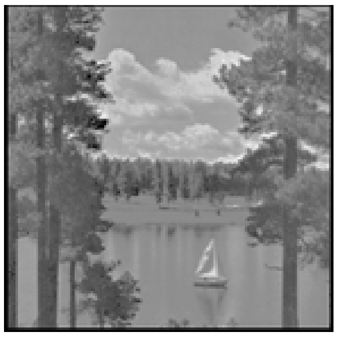
There were many major findings for this part. First, we had to use a test image to see if the computations worked or not. We used the values from the example shown in Figure 2. The results turned out correct when printing the s array of values. Second, equalizing the image by taking what was in the s array, as compared to using the k levels as the index, seemed like a challenge since the resulting image did not look that different.

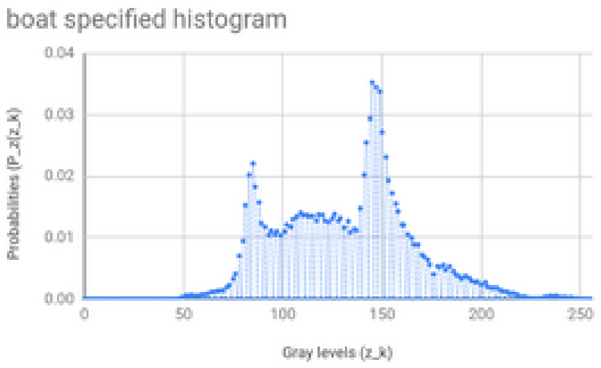
However, when we put the printed values of the histogram on Excel, we were able to find the results satisfying. As can see from the two figures, the images were equalized to a fairly nice level. We also tested the trend line on Excel, which were not shown in the above figures. The result was a a linear and almost flat line across the levels of the histogram. This meant that the histograms spread the gray levels almost evenly but still kept the shape of the original histograms.

3.4 Histogram Specification

Figure 12 shows the specification of the `"boat.pgm"` image. Figure 13 shows the specification of the `"f_16.pgm"` image.







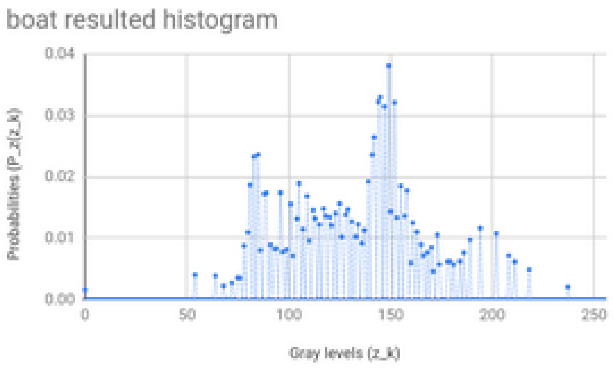
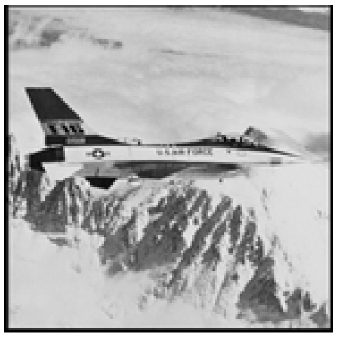
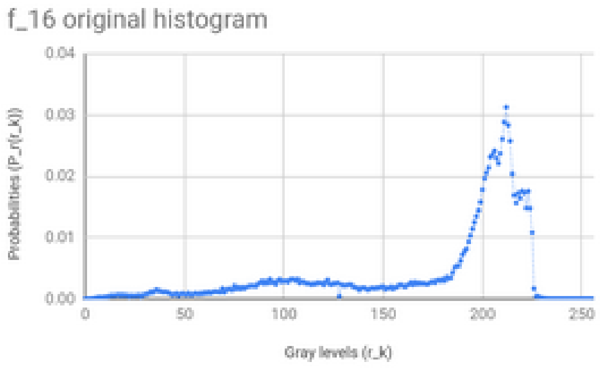
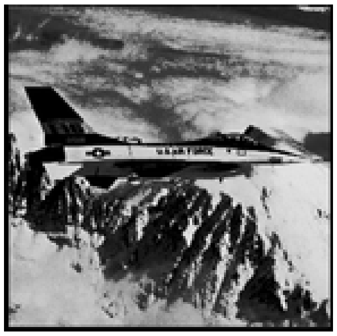


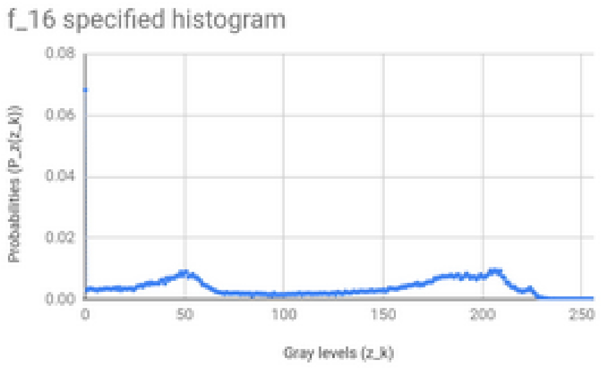
Figure 12: The original *boat* image and its histogram is displayed at the top. The specified histogram comes from the *sf* image. The resulted histogram is displayed at the bottom.

The `"sf.pgm"` image was overly grayed out, so the resulting image looks like it was covered in gray. As can see, the specified *sf* histogram, when used with the original histogram, made the gray levels wrap towards the middle.









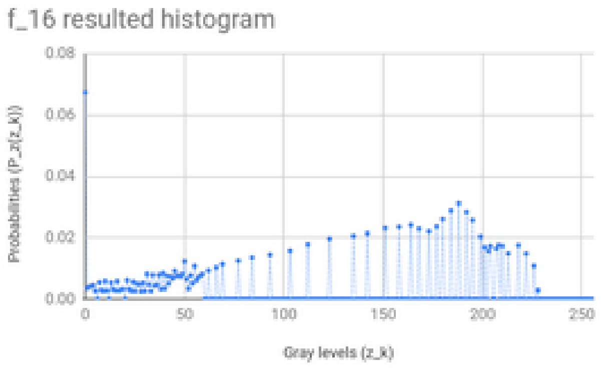


Figure 13: The original *f_16* image and its histogram is displayed at the top. The specified histogram comes from the *peppers* image. The resulted histogram is displayed at the bottom.

The `"peppers.pgm"` image had dark gray levels, so the resulting image becomes a bit black. As can see, the specified *peppers* histogram, when used with the original histogram, made the gray levels flatten down.

As a whole, histogram specification was very challenging to program for, and it is true that it is computationally expensive. Computing for $s = T(r)$ from the previous part was a tough start, but after getting that done, we were able to apply it to $v = G(z)$. The trick was getting $z' = G^{-}1(s)$ correct.

Because of trying to find the closest match to a gray level that might not be in the v array, we had to write a few helper functions to grab the value closest and also get the index of some value! In the end, the results matched with the test data using Figure 2, so we applied it with our `"boat.pgm"` and `"f_16.pgm"` images.

But, before we had to do that, we noticed that we needed to split the histogram data of getting their probabilities stored for each gray level. So, we used the `getHistogram` function for all four images (two original and two specified ones), which would help us retrieve and reuse such probabilities. These probabilities were important when passing into the `specifyImage` function, as we needed to work with both data for three computation steps.

The results, with a visual thanks to importing the generated text file dataset, were very surprising as well. Just like we learned in lecture, histogram specification can use a non-uniform pdf and still equalize the images in some ways while using a totally unrelated image as specification. The `"boat.pgm"` image looked very gray in the resulting image due to specifying the `"sf.pgm"` image, and the resulting histogram's new shape had combined the equalization of two histograms. The same goes to the `"f_16.pgm"` image, as it became more black due to specifying `"peppers.pgm"` image's gray levels.