Deev Patel & Wei Tong

CS 474: Image Processing & Interpretation

Monday, September 24, 2018

Programming Assignment # 1

Sampling, Quantization, & Equalization

Technical Discussion

(1) Sampling:

The sampling program asks the user what size they want to change the input image to. Only the resolutions specified in the assignment (128x128, 64x64, and 32x32) are allowed. For each subsequent sampling level starting at 128x128, the program takes every other pixel value in every other row and copies its value into the pixels to its right, directly underneath it, and diagonally below. Essentially, the image is first subdivided with non-overlapping 2x2 squares. Then, inside each square, the top left value is copied to all three other locations. This method is efficient because it skips the step of having to copy into a smaller array before having to recopy to a larger array when displaying.

(2) Quantization:

The goal of the quantization program is to lower the amount of gray values, or quantization levels, in an image. By default, the program assumes an input images' gray values range from 0 to 255 inclusive. Then, based on this assumption, the program is then able to lower the number of gray values to 128, 32, 8 or 2. Lowering the number of gray values to 2 is done by setting each pixel value to 0 if it was less than 128, or setting it to 255 if it was greater than or equal to 128. Lowering the number of gray values to 8 is done by first dividing the integer value of each pixel by 18.2. After dividing by 18.2, the integer remaining (since in C++ the decimals are dropped) has its parity checked. Odd numbers are rounded up to the next even number. This number is then multiplied by 18.2 and rounded to achieve the final gray value. This way, all the gray values would be reduced to only include values from the 8 member set of {0, 36, 73, 109, 146, 182, 219, 255}. The 18.2 number was obtained by dividing 255 by 14. The “magic” number 14 was obtained by taking the number of desired sections, subtracting 1, and multiplying by 2. The same concept is applied when reducing the number of gray values to 32, except pixel values are divided by 4.113, which was obtained by dividing 255 by 62. Finally, to lower the number of gray levels to 128, the program simply keeps each pixel value if its even and subtracts 1 from the value if its odd.

(3) Equalization:

Histogram equalization seeks to take the probability density function (pdf) of an image’s quantization values, or its normalized histogram, and transform it into a uniform distribution. This is done in hopes of spreading out the quantization values over the entire range of possibilities and giving the image more contrast. The technique is especially effective when the image is very bright or dark since redistributing the values makes it easier to visually discern different objects/places in the image.

Mathematically, histogram equalization is achieved by using the probability distribution function (also known as the cumulative distribution function or cdf) of an image as a mapping for its normalized histogram. This can be proven, at least in the continuous case, to produce a uniform distribution using probability theory. From probability theory, we know that (Eq 1) holds true for a mapping between random variables Y and X defined by Y = T(X) where f_Y and f_X are the respective probability density functions.

$$\mathbf{f_Y(y) = \left[f_X(x)\,\frac{dX}{dY}\right]_{X\,=\,T^{-1}(Y)}} \qquad \textbf{(Eq 1)}$$

However, in the special case where we define the transformation T(X) to be the probability distribution function of X, as shown in (Eq2), when can simplify (Eq1) to obtain (Eq3) using the fundamental theorem of Calculus.

$$Y = cdf(\mathbf{X}) = \int_{-\infty}^{X} f_{\mathbf{X}}(\mathbf{w})\, \mathbf{d}w \qquad (\mathbf{Eq\ 2})$$

$$\mathbf{f}_{\mathbf{Y}}(\mathbf{y}) = \left[\mathbf{f}_{\mathbf{X}}(\mathbf{x}) \frac{\mathbf{1}}{\mathbf{f}_{\mathbf{X}}(\mathbf{x})}\right]_{\mathbf{X} = \mathbf{T}^{-1}(\mathbf{Y})} = [\mathbf{1}]_{\mathbf{x} = \mathbf{T}^{-1}(\mathbf{Y})} = \mathbf{1} \quad (\mathbf{Eq\ 3})$$

Since f_Y is always equal to 1, it is uniform by definition. So, we can use the probability distribution function as the mapping to transform a normalized histogram into a uniform distribution. It is important to note that this doesn’t always work perfectly in practice. This is because, in the real world, we cannot always deal with continuous functions.

Results

(1) Sampling:

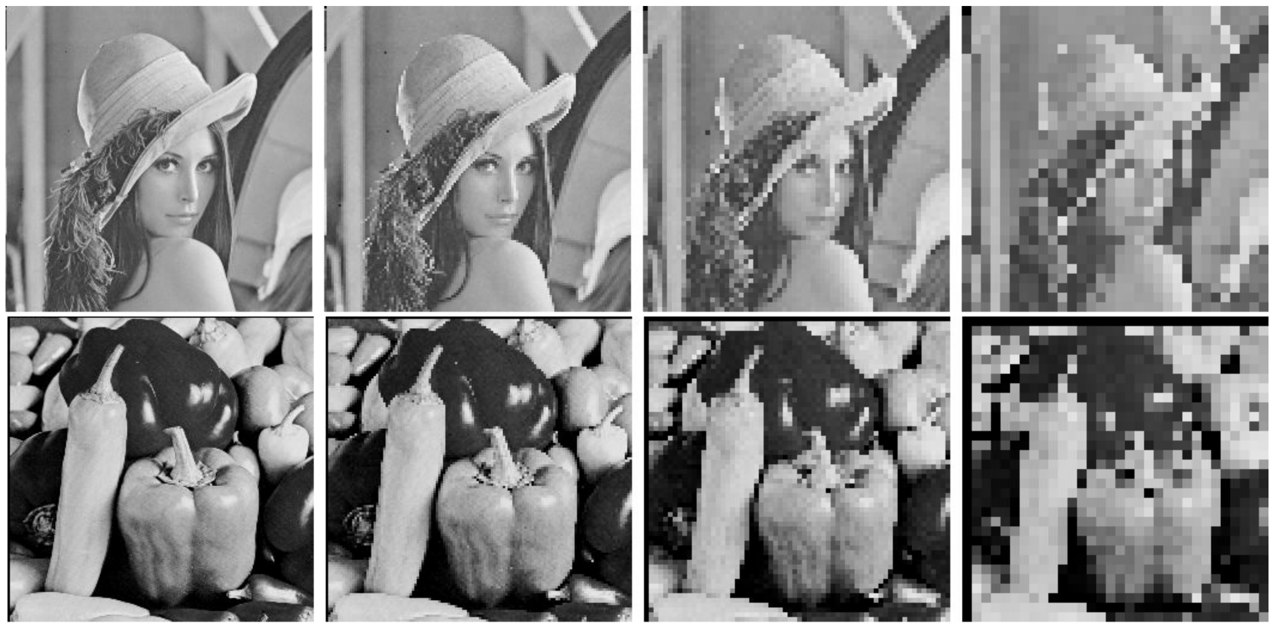


Figure 1: Two images used to test the sampling function. The leftmost image is the original 256x256. It is followed by 128x128, 64x64, and 32x32 versions.

(2) Quantization:

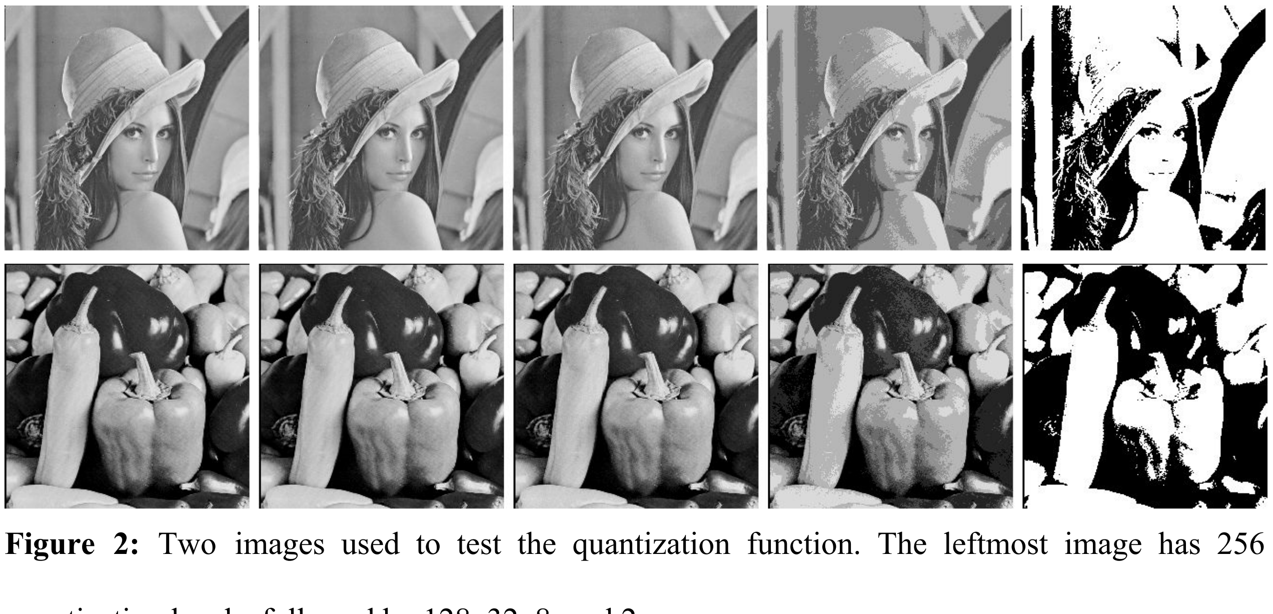


Figure 2: Two images used to test the quantization function. The leftmost image has 256 quantization levels, followed by 128, 32, 8, and 2.

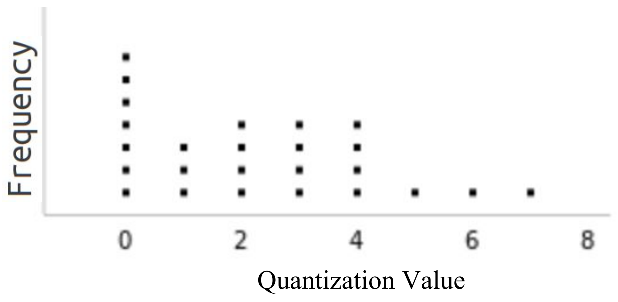
(3) Equalization:

Original Data

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

Equalized Data

6	5	6	6	1
5	1	5	1	3
7	3	1	6	2
6	2	5	1	3
1	6	3	2	1



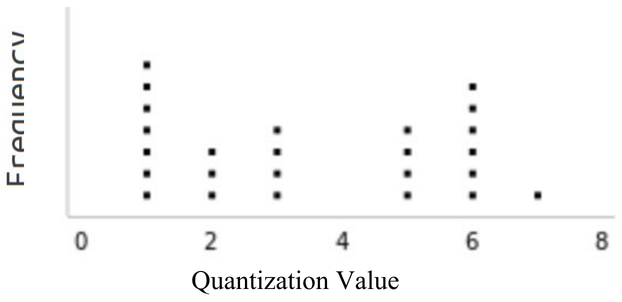


Figure 3: A 5x5 test "image" used to debug/verify the histogram equalization program. On the top left are the pixel values of the original image that were fed into the program. On the top right are the pixel values of the equalized image produced by the program. Below each image is the corresponding histogram for reference.

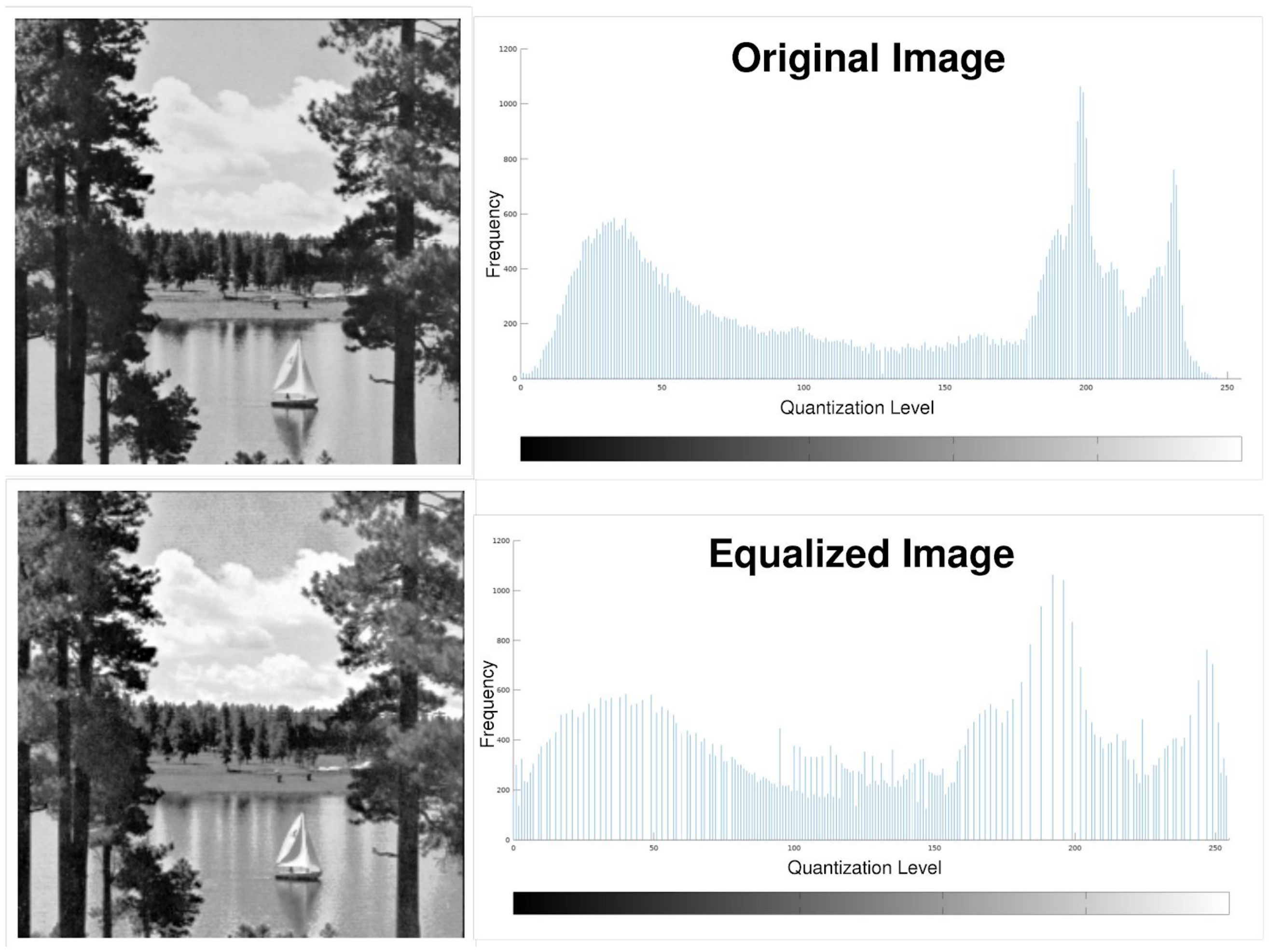


Figure 4: The original "boat" image and its histogram (on top) vs the equalized version and its histogram (on bottom).

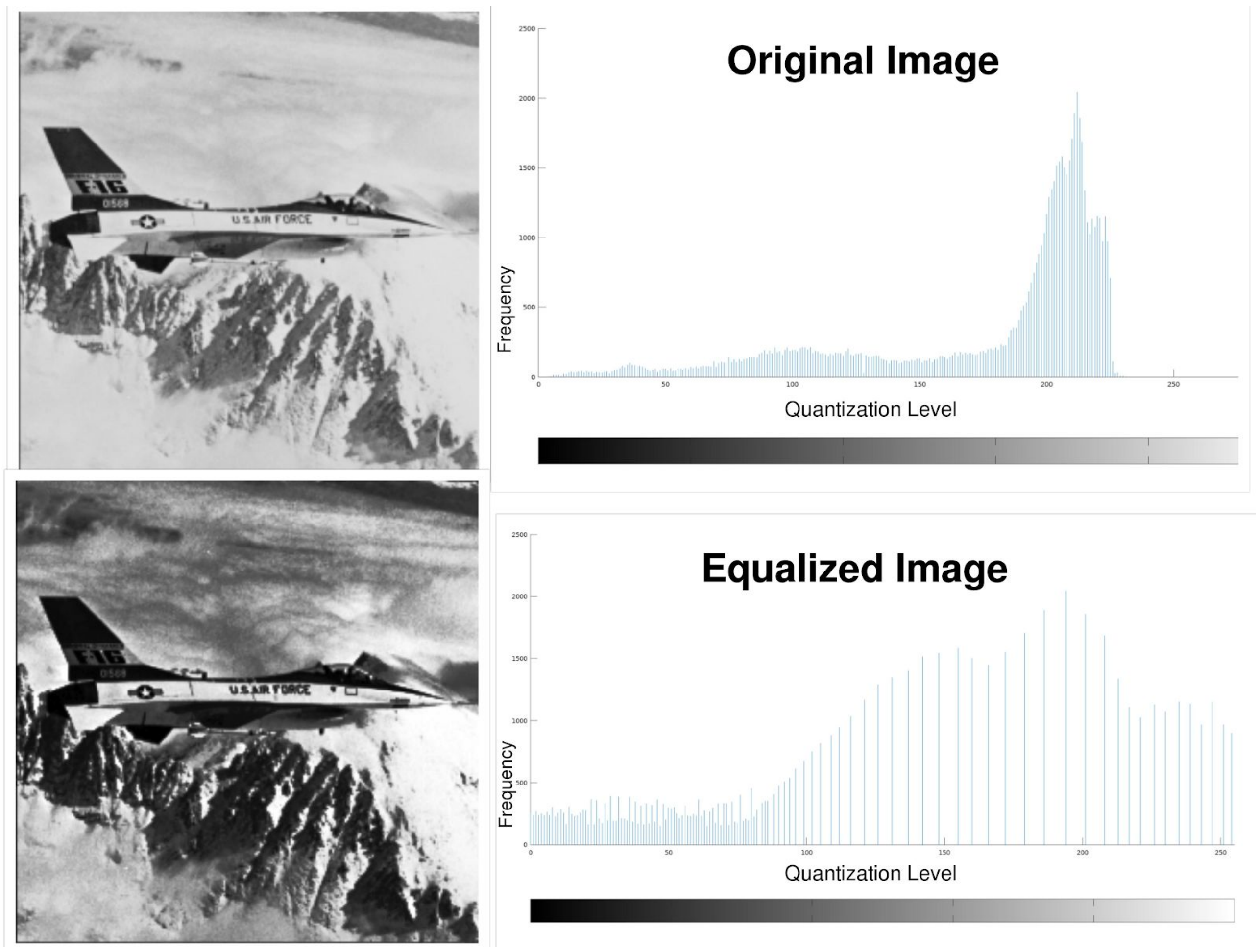


Figure 5: The original “f_16” image and its histogram (on top) vs the equalized version and its histogram (on bottom)

Discussion

(1) Sampling:

Changing the spatial resolution to 128x128 yielded an immediate discernable difference. The edges and curves of both images seemed much more jagged compared to the original. However, they are much less noticeable from a distance. Changing the resolution to 64x64 intensified this effect. The hair in the upper row of Fig. 1 could be easily mistaken for some other scraggly object, while finer details of the peppers were completely gone. The 32x32 resolution found on the far right of Fig. 1 is so pixelated that it is only possible to tell that there is some food in the bottom image and quite possibly a human in the top image.

(2) Quantization:

Between the original images and the 128 gray level images, found second to the left in Fig. 2, it is very hard to find any differences unless the images are inspected closely with a good monitor. However, the changes become quite a bit more noticeable on the 32 gray level images, found in the center of Fig. 2. The black pepper situated on top of the main pepper looks darker than the original and 128 level image. The difference on 8 level image is very noticeable, even from a distance. There is peppering throughout the image due to a lack of shade variety. The difference on the 2 gray level image is very obvious because it is just black and white. The 128 gray levels seem to work fine for reducing the number of gray levels. The changes are not obvious at all, and only appear when comparing side by side with the original. 32 gray levels also works fine, but the images begin to look noticeably darker than the original.

(3) Equalization:

In terms of implementation, the histogram equalization algorithm worked as required. This was verified by testing the program with the test dataset shown in Fig. 3. The results, also shown in Fig. 3, matched both mathematically with the output numbers calculated by hand and visually with the fairly uniform histogram of the output data.

Visually, the equalization process created more detailed, defined images. In Fig. 4, we can see the process brought out details in previously smooth areas of the “boat” image. Notably, this happened in originally dark and light areas. This is because, as the histogram shows, the original image simultaneously had a large number of pixels with values in a low range (around 10-25) and a high range (180-240). As a result, in the equalized image, we can discern various details inside what was once a fairly uniformly colored sky. Notably, the clouds are much more defined with a deeper contrast in regards to the rest of the sky. On the other end of the spectrum, we can see different shades inside originally dark shaded regions. Specifically, the tree trunks on the bottom right and left of the image are no longer completely black. Beyond this stretching of the low and high regions, there is not much more visually different about the equalized image. Again, this is a direct result of the bimodal nature of the original histogram.

The equalization process was much more evident on the “F-16” image, as shown in Fig. 5. Here, we can again see how the process brings out more details in originally smooth, uniform areas. In this case the snowy areas had their values stretched, resulting in dramatically more visible details in place of the previously bland, white snow. This image specifically was highly affected by the equalization processes because, as the original histogram shows, the full range of values were not originally used. In fact, as the top right of Fig. 5 shows, there were almost no pixel values above 230 in the original “f_16” image. As a result, the equalization process darkened a lot of the image when the values were stretched to spread across the entire interval.

Analytically, it is clear that histogram equalization does not produce perfectly normal histograms. This can be seen with clarity in the bottom right of Fig. 5, where the equalized histogram is much taller on the right half. This discrepancy is due to the real world limitation of having to use a discrete number of quantization levels. However, despite not being perfect, the transformation’s effects are evident in almost all domains of analysis. Moreover, the increase in uniformity helps bring out contrast and detail in otherwise uneventful regions.

Program Listings

(1) Sampling:

```
for(int i = 0; i < N; i += loopSkipper){
     for(int j = 0; j < M; j += loopSkipper){
           image.getPixelVal(i, j, myValue);
           for(int k = 0; k < loopSkipper; k++){
                for(int l = 0; l < loopSkipper; l++){
                     image.setPixelVal(i + k, j + l, myValue);
                }
          }
     }
}
```

(2) Quantization:

```
for(int i = 0; i < N; i++){
     for(int j = 0; j < M; j++){
          image.getPixelVal(i, j, myValue);
          if(choice == 1){
               if(myValue % 2 == 1){
                    image.setPixelVal(i, j, myValue - 1);
               }
          }
          else if(choice == 2){
               myValue /= 4.113;
               if(myValue % 2 == 1){
                    myValue++;
               }
               roundedAnswer = myValue * 4.113;
               myValue = std::round(roundedAnswer);
               image.setPixelVal(i, j, myValue);
               /*
                * In future implementations, use formula to
                * obtain the "magic" number. Number is found
                * by dividing 255 by (number of sections - 1) * 2
                * Further details are explained in comment below
                */
          }
          else if(choice == 3){
               myValue /= 18.2;
               if(myValue %2 == 1){
                    myValue++;
               }
               roundedAnswer = myValue * 18.2;
               myValue = std::round(roundedAnswer);
               image.setPixelVal(i, j, myValue);

               /*
                * Values obtained by dividing 255 into 7 sections
                *       0, 36, 73, 109, 146, 182, 219, 255
                * Dividing by 14 finds the values midway between
                * the above values. 18.2 is the size of each 1/14
                * section, so dividing by it will give us which
                * 1/14 section the number is in. Rounding up odd
                * numbers to the next even number puts them in
                * the correct gray value when multiplied by 18.2.
                * Not rounding up puts it exactly at the numbers
                * midway between the 7 gray numbers
               */
          }
          else if(choice == 4){
               if(myValue < 128){
                    image.setPixelVal(i, j, 0);
               }
               else{
                    image.setPixelVal(i, j, 255);
               }
          }
          else{
               std::cout << "Invalid choice" << std::endl;
               return -1;
          }
      }
  }
```

(3) Equalization:

To equalize an image, simply read in the image and properly call each of the following functions in the correct order as listed below. Make sure to delete any dynamically allocated arrays returned by the functions when done.

```
//function to calculate and return unnormalized histogram representing the
//frequency of each gray scale value
int * computeHistogram(const ImageType & img) {
    int numRows, numCols, maxVal;
    img.getImageInfo(numRows, numCols, maxVal); //get details of image

    //create histogram array and initialize to zero
    int * histogram = new int[maxVal + 1];
    for (int i = 0; i <= maxVal; ++i)
        histogram[i] = 0;

    int tempVal;
    //iterate through every pixel
    for (int r = 0; r < numRows; ++r)
        for (int c = 0; c < numCols; ++c) {
            img.getPixelVal(r, c, tempVal); //read pixel value
            ++histogram[tempVal]; //increment frequency value in histogram
        }

    return histogram;
}

//function to normalize histogram and get pdf of the image - returns
//probability density function
double * computePDF(const int * histogram, int size, int samples) {
    double * normalized = new double[size]; //allocate array to hold pdf
    //normalize each value by dividing by number of samples
    for (int i = 0; i < size; ++i)
        normalized[i] = (double) histogram[i] / samples;

    return normalized;
}
```

```
//function to compute cdf of image by making each term the sum of itself and
//the previous one - returns probability distribution function
double * computeCDF(const double * data, int size) {
    double * cdf = new double[size]; //allocate array to hold cdf

    cdf[0] = data[0];
    for (int i = 1; i < size; ++i)
        cdf[i] = cdf[i - 1] + data[i];

    return cdf;
}

//function to equalize the given image using the given cdf
void equalize(ImageType & img, const double * histogramCDF) {
    int numRows, numCols, maxVal;
    img.getImageInfo(numRows, numCols, maxVal); //get details of image
    --maxVal; //account for 0 being lowest value

    int currVal;
    //iterate through every pixel
    for (int r = 0; r < numRows; ++r)
        for (int c = 0; c < numCols; ++c) {
            img.getPixelVal(r, c, currVal); //read each pixel value
            //change each pixel value using the cdf as the mapping
            img.setPixelVal(r, c, (int) (histogramCDF[currVal] * maxVal));
        }
}
```

Division of Work

The image sampling and quantization sections, of both the code and the report, were done by Wei. The histogram equalization sections, of both the code and the report, were done by Deev. We believe this was an and equal division of work because it made each person responsible for 50% of the points of the assignment.