Programming Assignment 1

1

CS 474

Steven Hernandez and Adam Lychuk

Date due: 10/05/2020

Date handed in: 10/05/2020

Division of work:

Steven Hernandez implemented problems 1 and 3 on Sampling.cpp and equalization.cpp respectively.

Steven Hernandez wrote the theoretical, implementation, results and discussion of problems 1 and 3.

Adam Lychuk implemented problems 2 and 4.

Adam Lychuk wrote the theoretical, implementation, results and discussion of problems 2 and 4.

Theory

Part 1
The first part of the assignment consisted of using sub sampling by factors of 2, 4 and 8 in order to change the partial resolution within images. This had the program get every other pixel in every other row when the factor was 2. This method was applied to the factor of 4 and 8 as well. Hence, when the factor was 4, every other 4 pixels were collected in every other 4 rows. The samples were simply placed in a double array that was sized to the respective factor. Once this was done, the image would be set back to its original size by using the samples it took from the original image.

Part 2.
Image quantization squishes an image's signal to a smaller range of values often in order to compress the images size, or for an aesthetic toon shading effect. This is achieved by mapping the original uncompressed signal to a signal with lower fidelity. Mathematically optimization to achieve this mapping function can be done in a variety of ways, but a common brute force method is a simple linear transformation applied to each pixel. This linear transform is variable to the lower fidelity signal and is as follows in the discrete case:

$$f(x_1 \rightarrow x_n) = \frac{s}{l} \bullet \frac{v}{s/l}$$

Where:

s= source level
l=new level
v= value
x= each individual pixel

Part 3
The third part of the assignment used a method called histogram equalization which gives the image a better contrast by distributing the intensity of the histogram. In order to accomplish this, the probability distribution function was used which has the probability density of the pixel values being summed up so it slowly adds up to 255 and makes what seems like a linear growth. Once the new values are calculated, they are used to replace the existing values in the original image to the new equalized image.

$$F_X(x) = P(X \leq x) = \sum_{k=0}^{x} P(X = k)$$

Figure 1: Probability Distribution Function

Part 4
Histogram specification is a superset of histogram equalization and maps the histogram of the target image to another specified histogram. From there the target image is transformed using that morphed histogram to maximize the color space of the target image. This is different from

histogram equalization as we do not simply transform the original images histogram to maximize the color space of the target image. Finding this additional transformation requires us to make an intermediary transformation between histograms. The formula is as follows:

$$z = Q(r) = G^{-1}(s)\ where\ s = T(r)$$

G is the transformation of target images histogram to the specified histogram, so G inverse results in the target image transformed by the specified histogram. In practice finding G inverse requires mapping the cumulative histogram of T(r) which is v=G(z). We then map z to s and then r to z resulting in the transformed histogram that we map our image to.

Implementation

Part 1

For the sub sampling part of the assignment, the read and write functions for the PGM files provided by the professor were used along with the image.h file which contained the class "image". The main program was made in the Sampling.cpp which has two main parts: reading the image while selecting the right pixels to use for the sample and writing back the selected pixels to the new image. Two for-loops were used in order to go through both the rows and columns of the image being read while an if-statement decided if the current pixel value is to be stored for the sample. When a pixel value is chosen to be stored for the sample, it is placed in a double array that is correctly sized to store all the necessary values. This part was fairly easy to do; however, the tricky part was using the values and correctly placing them in the new image.

In order to accomplish this, the double for-loop was used once again to go through each pixel value of the original image along with counters to help move to the next index of the double array. The counters would increase in every iteration, but once they match the given sample factor they would reset back to zero and move on to the next element of the double array. This process helped go through every index of the double array and correctly place the values in the original image. Once that was done, the writeImage function would write the new image with the given name.

Part 2

After instantiating a new image object we step through the target image with two for loops. At each pixel we apply a linear transformation exactly as described in the theory section and take the output of that value and place it into the new image instantiation. We then return the new image object by returning a pointer to the dynamically allocated memory.

Part 3

This program used the histogram equalization technique to give a better contrast to the given image. In order to do this, three separate arrays were used to store, process and pass the pixel values of the given image. It starts by using a double loop to read the image while using another for-loop to count from 0-255, the minimum and maximum pixel values, and stores the value in the correct index of the first array that has exactly 256 elements. The next step uses a float variable with the value of the number of pixels in the image and the second array, also with 256 elements, to store the normalized frequencies once each of the elements of the first array is divided by the number of pixels. After this is done, the third array is used to apply the probability distribution function and sums up the frequencies of the second array. The third array then goes through another for-loop to multiply the values by 255, the max pixel value, and rounds down each value by using the floor function. Lastly, the new values are placed in the image by using a double for-loop and uses the original value of the pixel as the index of the array that holds the new information.

Part 4

The histogramSpec() function calls helper functions to perform the full specification. We get the histogram for each image using histogram() which simply sums 1/number of pixels for each gray level and stores the values in an array. The cumulative histogram is then created using the cumulativeHist() function which sums those values for each grey level using a for loop. Histogram spec uses both these helper functions on the target image and the image provided solely for its histogram. The resulting histograms are then sent to the inverseMap() function which creates the map of integer values as an array. That array is then used in mapImage() to make the final resulting image which is then returned.

Results and Discussion

Part 1

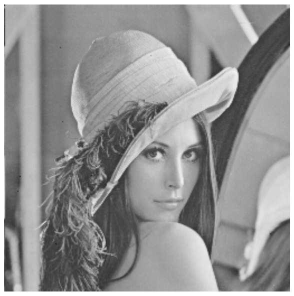


Figure 2: Original Lenna.pgm

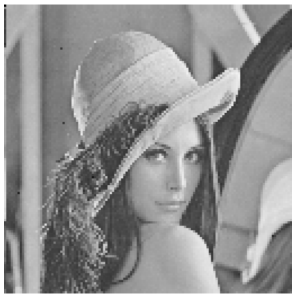


Figure 3: Factor 2 Lenna

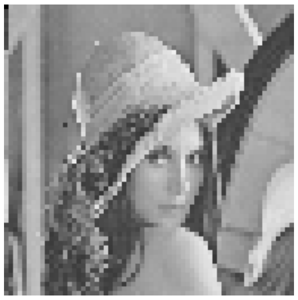


Figure 4: Factor 4 Lenna

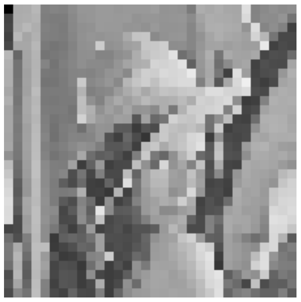


Figure 5: Factor 8 Lenna

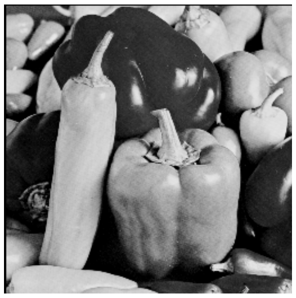


Figure 6: Original peppers.pgm

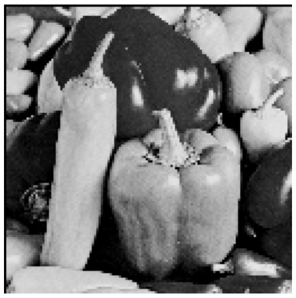


Figure 7: Factor 2 peppers

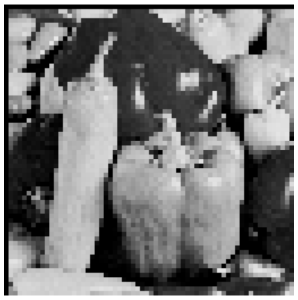


Figure 8: Factor 4 peppers

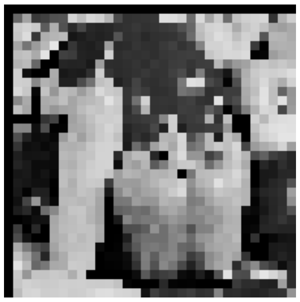
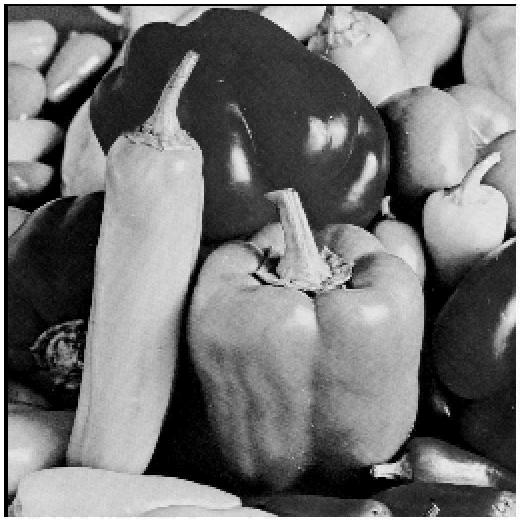


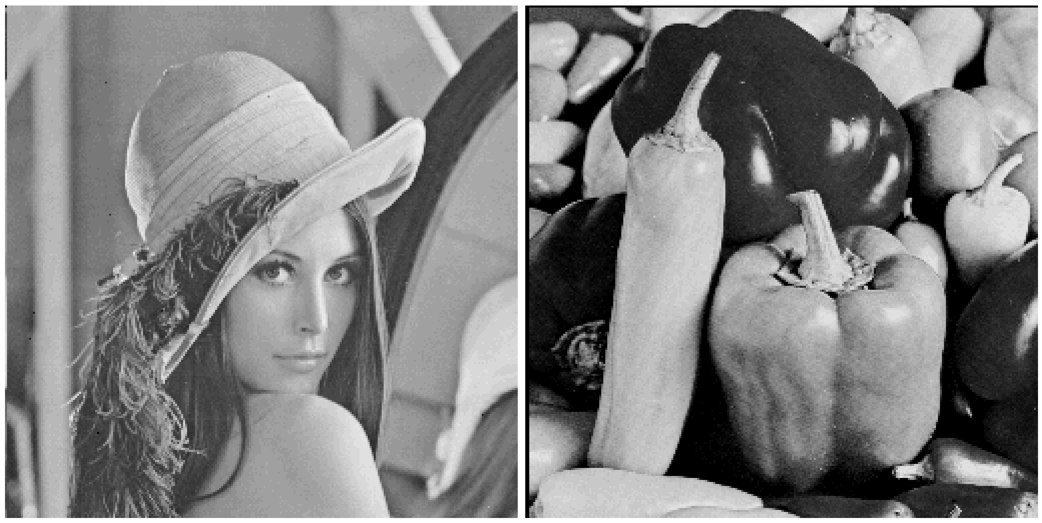
Figure 9: Factor 8 peppers

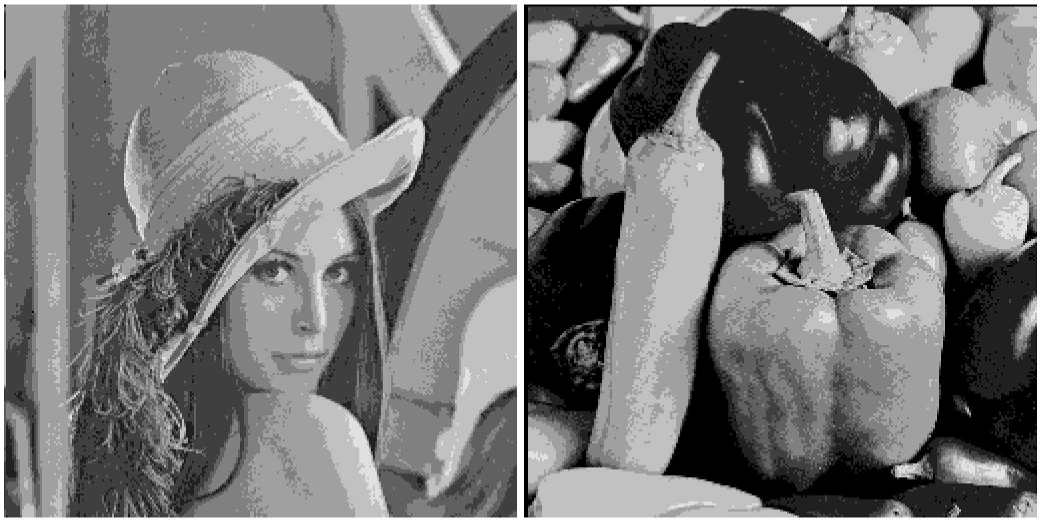
Figure 3 and figure 7 are the factor 2 versions of their original images and while there isn't considerable difference in quality unless you look very closely at the picture, it still has some things that can show the difference in quality such as the edges of the vegetables or the outline of Lenna. Sampling into factor 4, figure 4 and figure 8 start showing even more of a pixelated image as the smaller details of the image are hard to discern. Finally, figure 5 and figure 9 really makes the peppers and Lenna look very blurry since they are factored by 8.

Part 2

i.




ii.


iii.

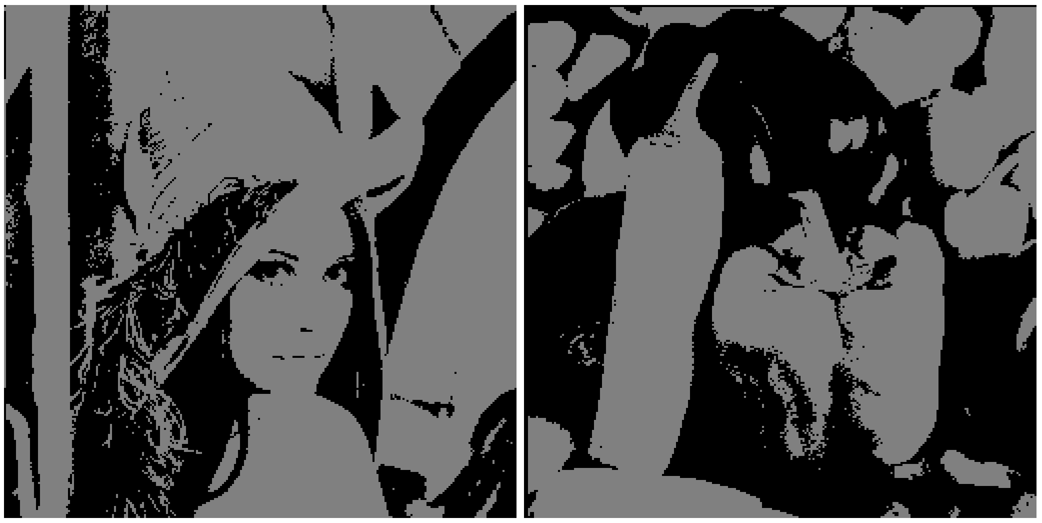

iiii.

Figure 10: Lenna and Peppers images arranged in order of quantization

In the above figure the quantized images are shown side by side ordered by the number of grey levels starting at 256 and decreasing to 128, 32, 8, and 2 respectively. It is easy to visually see a decrease in fidelity as less grey levels are used, in particular at levels 8 and 2 where a dramatic change in appearance has occurred. You can also see that both 32 and 128 are not noticeably different when looking from afar. With the memory saved compressing the images it may be worth it when there is no noticeable difference.

Part 3

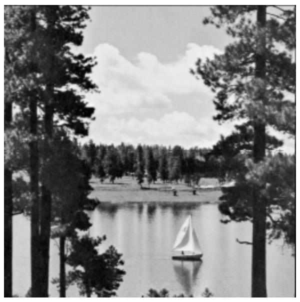


Figure 11: Original boats.pgm

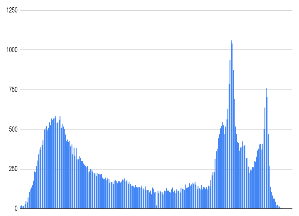


Figure 12:His. boats.pgm

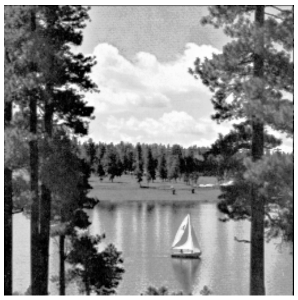


Figure 13:Equ. Boats.pgm

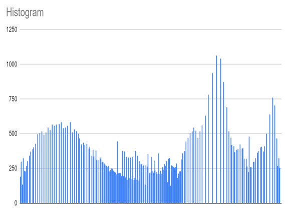


Figure 14: His. Equ. boats.pgm

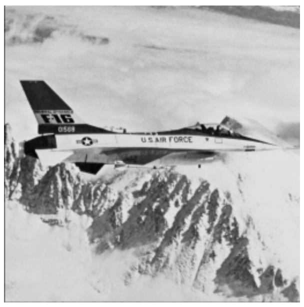


Figure 15: Original f_16.pgm

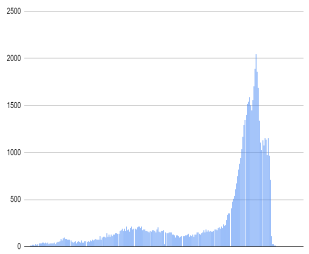


Figure 16: His. f_16.pgm

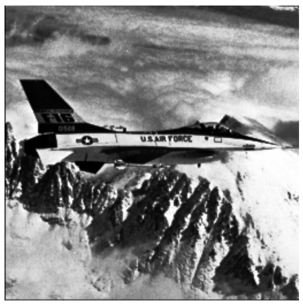


Figure 17: Equalized f_16.pgm

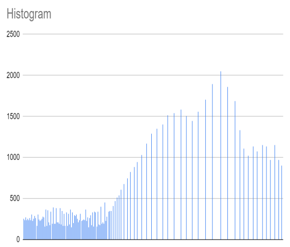


Figure 18:His. Equ. f_16.pgm

Histogram equalization is supposed to help with the contrast in images and it did not disappoint. The boats.pgm seen in figure 10 seem to have a high contrast which is reflected by seeing the histogram in figure 11. Once the histogram equalization technique is applied, it does not seem to show a big difference unless one takes a closer look to the clouds and the leaves where they used to be darker in figure 12. The histogram shown in the figure 13 shows how the pixel values were spread out due to the technique, but still retains most of the shape it originally had. The next image, figure 14, is a fighter jet that is high above the clouds and its histogram shows it is clearly a bright image. After applying the technique, it really shows the outlines on the clouds much more clearly compared to the ocean of white. Its equalized histogram, figure 17, shows how the values were spread out to help give the image a better contrast, but it seems it may have gone a bit overboard with the darker values.

Part 4

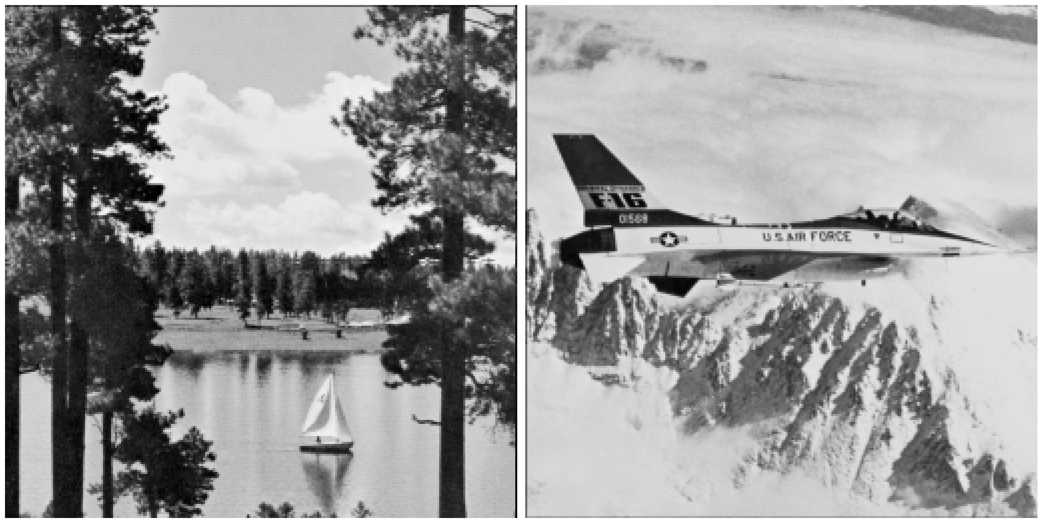


Figure 19: Left boat.pgm Right f_16.pgm

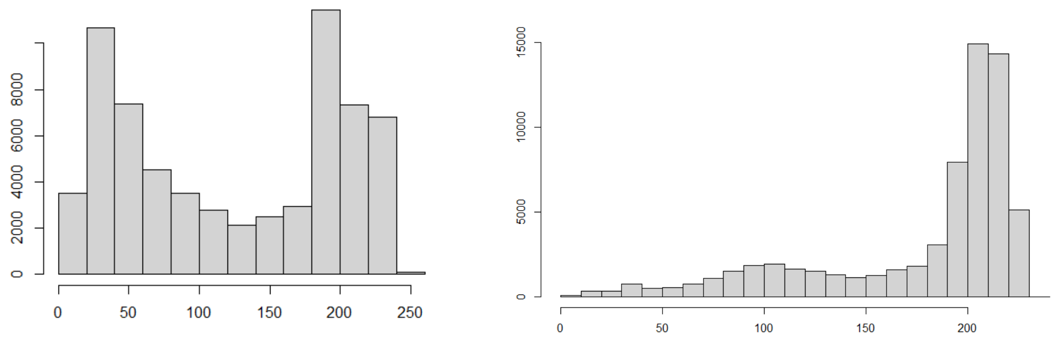


Figure 20: Left boat.pgm histogram Right f_16.pgm histogram

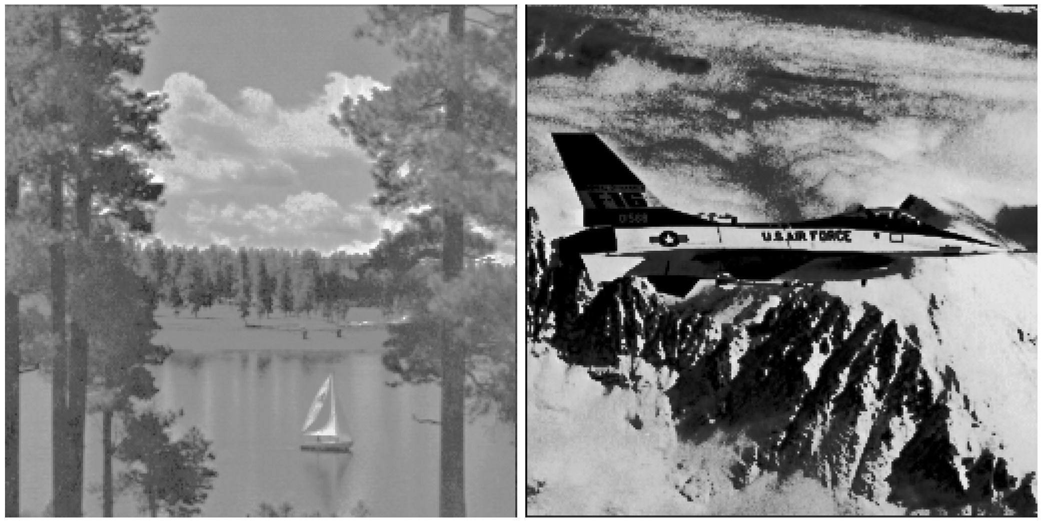
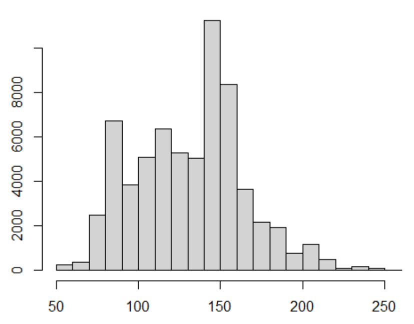


Figure 21: Left boat.pgm after specified histogram transformation Right boat.pgm after specified histogram transformation



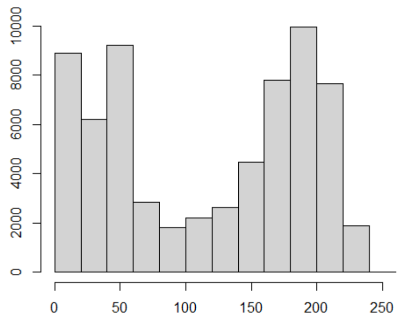


Figure 22: Left boat.pgm histogram after transformation Right f_16.pgm histogram after transformation

In the above figures are the before and after specification images for the boat and f_16 images as well as before and after for their histograms. Interestingly the color profiles match the color profiles of the images the specified histogram was drawn from, the sf and peppers image. This makes sense as we are essentially equalizing the histogram and then fitting it to the histogram of the other images. Since histograms are a representation of gray levels it makes sense that the images have similar contrast to the specified histogram.

Source Code:

Part 1

```
// Steven Hernandez
// CS 474
// 9/24/2020
// assignment 1 part 1
#include <iostream>
#include <fstream>
using namespace std;
#include "image.h"

int readImageHeader(char[], int&, int&, int&, bool&);
int readImage(char[], ImageType&);
int writeImage(char[], ImageType&);
int main(int argc, char *argv[])
{
 int i, j;
 int M, N, Q;
 bool type;
 int val;
 int sample;
 int resolution;


 // read image header
 readImageHeader(argv[1], N, M, Q, type);

 // allocate memory for the image array

 ImageType image(N, M, Q);

 // read image
 readImage(argv[1], image);

 cout << "Enter image sub-sample factor (2, 4 or 8): ";
 cin >> sample;
 resolution = 256 / sample;

 //I made a double array that is sized on the N M of the
factored image
 int SampleArr[resolution][resolution];
 //These two variables are used to change index of double
array
 int arrayi = 0, arrayj = 0;

/////////// Sub Sample
//////////////////////////////////////////////////////////
```

```
//two for loops to get each pixel value
 for(i = 0; i < N; i++)
 {
   for(j = 0; j < M; j++)
   {
     image.getPixelVal(i,j, val);
     //uses % to decide if the pixel value needs
     // to be stored into double array "Samplearray"
     if((i % sample == 0) && (j % sample == 0))
     {
       SampleArr[arrayi][arrayj] = val;

       arrayj++;
       if(arrayj == resolution)
       {
         arrayj = 0;
         arrayi++;
       }
     }
   }
 }

 int counteri = 0, counterj = 0;
 int SampleCounterj = 0, SampleCounteri = 0;

 for(i = 0; i < N; i++)
 {
   for(j = 0; j < M; j++)
    {
    //sets corresponding pixel value
    image.setPixelVal(i,j, SampleArr[counteri][counterj]);

    //Increases SampleCounter which counts how many
iterations
    //before going to next index of Samplearray
    SampleCounterj++;
    //Resets SampleCounter if equal to sample factor
    if(SampleCounterj == sample)
     {
      counterj++;
      SampleCounterj = 0;
     }
    }

   counterj = 0;
   SampleCounteri++;
   if(SampleCounteri == sample)
    {
     counteri++;
     SampleCounteri = 0;
    }
 }
////////    End of sub sampling
////////////////////////////////////////////////

 // write image
 writeImage(argv[2], image);

 return (1);
}
```

Part 2

```
ImageType* quantize(ImageType& image, int type) {
  int rows, cols, levels, val;
  image.getImageInfo(rows, cols, levels);
  ImageType * out_image = new ImageType(rows, cols, levels);
  for (int i=0; i<rows; ++i) {
    for (int j=0; j<cols; ++j) {
      image.getPixelVal(i,j,val);
      out_image->setPixelVal(i, j, (256/type) * (val / (256/type)));
    }
  }
  return out_image;
}
```

Part 3

```
// Steven Hernandez
// CS 474
// 9/24/2020
// assignment 1 part 3
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;


#include "image.h"

int readImageHeader(char[], int&, int&, int&, bool&);
int readImage(char[], ImageType&);
int writeImage(char[], ImageType&);

int main(int argc, char *argv[])
{
 int i, j;
 int M, N, Q;
 bool type;
 int val;
 int sample;
 int resolution;


 // read image header
 readImageHeader(argv[1], N, M, Q, type);

 // allocate memory for the image array

 ImageType image(N, M, Q);

 // read image
 readImage(argv[1], image);

////////////////////////////////////////////////////////////
/////////////////////
//Counts how many of a certain pixel value there is 0-255
 int EqualArr[256] = {0};

 for(i = 0; i < N; i++)
 {
   for(j = 0; j < M; j++)
   {
     image.getPixelVal(i,j,val);             //gets the
next pixel and its value
     for(int a = 0; a < 255; a++)  //loops from 0-255 to
find the value of pixel
     {
       if(val == a) //Once the value is found, it adds 1 to
the frequency of its
       {            // respective value index within the
array
         EqualArr[a] += 1;
       }
     }
   }
 }

float NumOfPixels = N * M;
float NormalizedArr[256] = {0};

// now get the normailized frequencies of the array
EqualArr divided by the
//total number of pixels in the whole picture
for(int a = 0; a < 256; a++)
{
  NormalizedArr[a] = EqualArr[a] / NumOfPixels;
}

// S=T(r)
float TArr[256] = {0};

//Probability distribution function
//sum up the frequencies, ex:S(1) = P(0) + P(1), S(2) =
P(0) + P(1) + P(2),
for(int a = 0; a < 256; a++)
{
  for(int b = 0; b < a+1; b++)
  {
    TArr[a] += NormalizedArr[b];
  }
}
```

```
for(int a = 0; a < 256; a++)
{
  //The values get multiplied by 255 to get them back to
the values 0-255
  TArr[a] *= 255;
  //Truncate, I rounded to preceding num using the floor
function
  TArr[a] = floor(TArr[a]);
}

for(i = 0; i < N; i++)
{
  for(j = 0; j < M; j++)
  {
    //call getPixelVal just to get 'val'
    image.getPixelVal(i,j,val);
    //replace original values with new equalized values and
DONE!
    image.setPixelVal(i,j,TArr[val]);
  }
}
/////////  End of Histogram  equalization
////////////////////////////////////
 // write image
 writeImage(argv[2], image);

 return (1);
}
```

PART 4

```
void PrintHistogramValues(int N, int M, ImageType &image){
  int i,j,val;
  ofstream myfile1;
  myfile1.open("hist.csv");

  for (i=0; i<N;i++)
  {
    for (j=0; j<M; j++)
    {
      image.getPixelVal(i,j,val);
      myfile1 << val << ",";
    }
    myfile1 << endl;
  }
  myfile1.close();
}

float * histogram(ImageType image) {
  int rows, cols, bytes;
  image.getImageInfo(rows, cols, bytes);
  int total_pixels = rows * cols;
  float * hist = new float[256];
  memset(hist, 0, sizeof(float)*256);
  int dummy = 0;
  for (int i=0; i<rows; ++i) {
    for (int j=0; j<cols; ++j) {
      if(image.getPixelVal(i,j, dummy) < 256 && image.getPixelVal(i,j,
dummy) >= 0)
        hist[image.getPixelVal(i,j, dummy)] += 1.0f/(float)total_pixels;
    }
  }
  return hist;
}

float * cumulativeHist(const float * hist) {
  float * cumulativehist;
  float sum = 0;
  cumulativehist = new float[256];
  memset(cumulativehist, 0, sizeof(float)*256);
  for (int i=0; i<256; ++i) {
    sum += hist[i];
    cumulativehist[i] = sum;
  }
  return cumulativehist;
}

int * inverseMap(const float * in_hist, const float * out_hist) {
  int * map = new int[256];
  float var = 0, prev_var = 256;

  for (int i=0; i<256; ++i) {
    for(int j=0; j<256; ++j) {
      var = std::abs(out_hist[j] - in_hist[i]);
      if (var > prev_var) {
        map[i] = j-1;
        break;
      } else if (j==255) {
        map[i] = 255;
      }
      prev_var = var;
    }
    prev_var = 256;
  }
  return map;
}

ImageType * mapImage(const int * map, ImageType image) {
  int rows, cols, bytes;
  image.getImageInfo(rows, cols, bytes);
  ImageType * out_image = new ImageType(rows, cols, bytes);
  int dummy = 0;
  for (int i=0; i<rows; ++i) {
    for (int j=0; j<cols; ++j) {
      out_image->setPixelVal(i, j, map[image.getPixelVal(i,j, dummy)]);
      std::cout << out_image->getPixelVal(i,j, dummy) << std::endl;
    }
  }

  return out_image;
}


ImageType* histogramSpec(ImageType& im, ImageType& imSpec) {
  float * in_hist = histogram(im);
  float * out_hist = histogram(imSpec);
  float * in_cumulative_hist = cumulativeHist(in_hist);
  float * out_cumulative_hist = cumulativeHist(out_hist);
  int * map = inverseMap(in_cumulative_hist, out_cumulative_hist);
  ImageType * mapped_image = mapImage(map, im);
  //writeImage("test.pgm", *mapped_image);
  return mapped_image;
}
```