Programming Assignment 1

CS 474

`https://github.com/alexander-novo/CS474-PA1`

Alexander Novotny	Matthew Page
50% Work	50% Work
Sections 3,4	Sections 1,2

Due: October 5, 2020
Submitted: October 4, 2020

Contents

1 Image Sampling

1.1 Theory

Image sampling is one of two key processes during image acquisition and digitization. Intuitively, sampling can be defined as the process whereby a continuous image is transformed such that the 2D locations of an image are represented by a discrete coordinate system. Mathematically this can be defined as a map

$$m : f(s,t) \to f(x,y)$$

where s,t are non-negative real values and x,y are non-negative integers. In this formalism an image is viewed as a 2D function, where the domain defines the location of the image and the range defines the possible pixel values of the image.

The quality of a digital image depends heavily on the number of samples taken. An image with more samples will in general appear more alike to the original continuous image, however it will require more memory to store an image as the number of samples increases. If an image is under sampled, then certain artifacts may appear in the newly created digital image and aliasing may occur. The spatial resolution of an image is a measure of how well sampled an image is, and helps quantify image quality in terms of pixels per unit of distance.

1.2 Implementation

In order to implement image sampling, a pgm image was first read into the program using command line arguments, along with the output image location and sub sampling factor. In order to sample the image, image pixels were iterated over using a nested for loop, however, every n pixels were skipped, where n is the sub-sample factor. For every n pixels, its value was stored and a second set of nested for loops were used in order to modify a $k \times k$ window of pixels, such that each pixel in the window became the same value as the sampled image. This process continues until the end of the image is reached. This approach was taken so that the image would maintain the same number of pixels for comparison and visualization purposes. The main data structure used was the image class, used to represent the original image. Each value of the image was modified in place to create the sub-sampled image.

1.3 Results and Discussion

The results of the sub-sampling algorithm are showcased using the `lenna.pgm` and `peppers.pgm` images. Figure 1 used the `lenna.pgm` image to demonstrate the effect of sub-sampling. According to the figure, the effects of sub-sampling become prominent immediately, particularly along the edges within the image where aliasing can be seen. Once the image is sub-sampled by a factor of four, the details of the image begin to be obscured. This is especially shown in the features of the woman's face and hair, where the details become less clear. Lastly, the image sub-sampled by a factor of eight loses nearly all of its finer details, and only the larger, more general features of the image remain identifiable.

The same sub-sampling process was conducted with the peppers image, which are shown in fig. 2. Similarly to the `lenna.pgm` image, the image quality begins to degrade after sub-sampling by a factor of two. After sub-sampling by a factor of four, the aliasing on the pepper's edges becomes more evident and the original texture of the peppers becomes more distorted. Finally, the image sub-sampled by a factor of eight loses most of the small details and it becomes difficult to make out the original image.

Figure 1: A comparison of `lenna.pgm` with varying sub-sampled images, scaled accordingly (From left to right: 256 x 256, 128 x 128, 64 x 64, 32 x 32).

Figure 2: A comparison of `peppers.pgm` with varying subsampled images, scaled accordingly (From left to right: 256 x 256, 128 x 128, 64 x 64, 32 x 32).

Based on the sampling experiments conducted, it appears that sampling has an immediate effect on the pgm images, with the smoothness of the edges being lost initially, followed by smaller details within the image, and finally the image loses most of its original features. Although details may be lost, sub-sampling may provide an opportunity to reduce the memory requirements of an image at a cost of quality. Further comparisons between sub-sampling and the technique of image quantization will be discussed at the end of the following section.

2 Image Quantization

2.1 Theory

Image quantization is the second major process when digitizing an image. As opposed to image sampling, which aims to discretize the image coordinates or pixel locations, quantization is responsible for discretizing the pixel values from a continuous value to finite integer values. Using the same model of an image as a 2D function, quantization is equivalent to transforming the output, or range, of the image function $f(x, y)$ from the set of non-negative real values to a finite set of non-negative integer values. Similar to sampling, quantization is required in order to properly store an image on a digital system by limiting the required number of parameters needed to represent an image, thereby saving memory or disk space.

The number of integers used for the pixel values is defined by the quantization level L. Typically this value is power of 2, and determines the range of possible pixel values from 0 up to $L - 1$. In general, the higher the quantization level is, the higher the quality of the image will be, due to more gray level values being possible at each pixel location. The measure of the number of possible discernible pixel or gray level values is defined as the intensity resolution of an image.

2.2 Implementation

Image quantization also used command line arguments to read the input image, output image location, and quantization level. After reading the image, each pixel is iterated over and discretized according to the new quantization level.

The main data structures used in this algorithm include the image class and a vector of new pixel values between 0 and 255. These values were calculated by obtaining a pixel value offset using the equation

$$\text{offset} = \frac{256}{L}.$$

Therefore, a quantization level of 2 gives an offset of 128. Using this offset, the ith pixel value is calculated as $i \cdot \text{offset}$ for $0 \geq i \geq$ quantization level. Each pixel value was converted using

$$p_{new} = newPixelValues[\lfloor p_{old}/\text{offset} \rfloor]$$

where $newPixelValues$ is a vector of size L containing the new gray level values of the image.

2.3 Results and Discussion

The quantization algorithm was performed on the two images `peppers.pgm` and `lenna.pgm`, both of whose original quantization level were 256. The results of quantizing `lenna.pgm` are shown in fig. 3. According to the figure, there is little noticeable difference from quantizing the image down to 128 and 32 gray level values. However, as seen in the figure, once the quantization level dropped to 8, noticeable differences begin to appear in the shading of the image. At quantization level two, the image essentially becomes a binary image, with much of the shading detail lost.

Figure 3: A comparison of `lenna.pgm` with varying quantization levels (From left to right: L=256, L=128, L=32, L=8, and L=2).

Figure 4 demonstrates a similar quantization process for the `pepper.pgm` image. With this image, there are very little noticeable differences for the level 128 and 32 quantized images compared to the original. At quantization level 8, the differences become prominent, especially for the two large peppers featured at the center of the image. Lastly, the level two quantized pepper image on the right resembles a binary image, with minimal detail compared to the other images.

Figure 4: A comparison of `peppers.pgm` with varying quantization levels (From left to right: L=256, L=128, L=32, L=8, and L=2).

Overall, it appears that the effects of image quantization does not degrade an image as much as sub-sampling. With both images, the effects of image sub-sampling become noticeable even after sub-sampling by a factor of two, whereas in image quantization the effects do not drastically detract from the quality of the image until the quantization level of 8 is used. As a consequence, in the case of memory limitations it would be advisable to first reduce the quantization level of an image rather than sub-sample it, as this would lead to the fewest defects while reducing the memory requirements of an image.

3 Histogram Equalization

3.1 Theory

It is desirable to have high contrast in an image, as it allows you (and a computer vision algorithm) to pick out details more easily. In general, images whose histograms have a uniform distribution tend to have high contrast - especially when compared with images with a central mode (represented by a central "hump" in the histogram). To convert a (continuously distributed) random variable X to a uniform distributed random variable Y, we simply apply the transformation

$$Y = F_X(X),$$

where F_X is the CDF (cumulative distribution function) of X. As a transformation of the variable X, we know then that the PDF (probability density function) of Y is

$$f_Y(y) = f_X(F_X^{-1}(y))\left|\frac{\mathrm{d}}{\mathrm{d}y}F_X^{-1}(y)\right|,$$

and by the inverse function theorem of calculus,

$$\begin{aligned}
&= f_X(F_X^{-1}(y))\left|\frac{1}{F_X'(F_X^{-1}(y))}\right| \\
&= f_X(F_X^{-1}(y))\left|\frac{1}{f_X(F_X^{-1}(y))}\right| \\
&= 1,
\end{aligned}$$

so $Y \sim \mathcal{U}(0,1)$. Of course, this only applies to continuous random variables, but we hope that a similar behaviour can be observed in discrete random variables. Unfortunately, since all pixels fall into a certain number of "bins" in the image's histogram (based on the quantization level of the image), the transform can't decrease the number of pixels in a bin. Instead, it can only spread bins out in the histogram and consolidate multiple bins into one, increasing the number of pixels in a bin. Therefore, if there are noticeable modes in the original image's histogram, there will still be noticeable modes in the equalized histogram. As well, image quality will drop due to the spreading out and consolidating of bins effectively quantizing the image.

Since this behaviour comes from the approximation of continuous distributions by discrete distributions, the better an approximation is, the less noticeable these effects become. Therefore, increasing the quantization level of an image will make the process more effective, causing the output image's histogram to be more like a uniform distribution.

3.2 Implementation

An array of integers is used for the image's histogram, which is calculated by looping over the image's pixels and incrementing the bin whose index is given by the pixel's intensity value. Then, the CDF is calculated by iteratively summing over the calculated histogram, using the recurrence relation for discrete CDFs:

$$\begin{aligned} F_X(x) &= \sum_{i=-\infty}^{x} P(X = x) \\ &= P(X = x) + \sum_{i=-\infty}^{x-1} P(X = x) \\ &= P(X = x) + F_X(x-1), \end{aligned} \tag{1}$$

where $P(X = x)$ is the histogram value of the intensity x. Since pixel intensities have finitely many values (and therefore a minimum value), $F_X(x-1) = 0$ for some x (the minimum intensity) and $F_X(x) = P(X = x)$.

The CDF is never converted to its normalized version. Instead, when applying the transformation, each resulting transformed pixel is multiplied by the normalization constant. This is to prevent accumulation of round-off errors until the final integer pixel value is calculated.

Since the calculation of the original histogram and the transformation is embarrassingly parallel, OpenMP is used to parallelize.

The source code for this implementation can be found in listing 6.

3.3 Results and Discussion

Figure 5 shows the result of applying the algorithm to the image `boat.pgm`. There is a noticeable difference in contrast - especially in the water, which is much clearer, and the shadows on the sail. However, there is some noise introduced in the sky, and loss of detail on the coast.

Figure 5: A comparison of `boat.pgm` with its equalization (right).

Figure 6 compares the original histogram of `boat.pgm` with the histogram of the new equalized image. As discussed in section 3.1, the new histogram is not that of a uniform distribution, but there are some notable improvements over the original histogram. Firstly, the bins concentrated around the various modes have become sparser, so while the modes still exist with the same number of pixels in their bins(as discussed earlier), there are fewer pixels in the region of the bin. As well, a couple of the modes have spread out, making them easier to differentiate between. Finally, the bins in lower regions of the histogram have concentrated so they aren't as low compared to the modes.

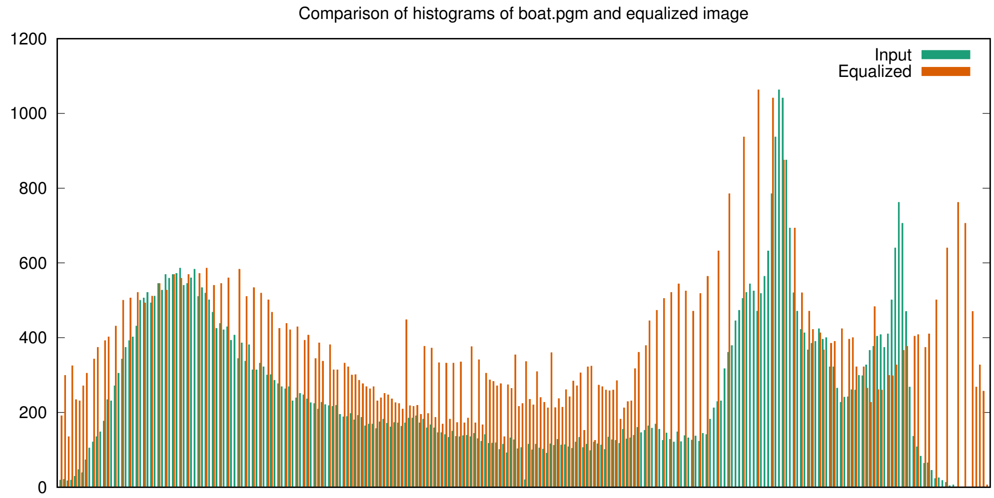

Figure 6: A comparison of histograms of `boat.pgm` and its equalized version

Figure 7 shows the result of applying the algorithm to the image `f_16.pgm`. There's a drastic increase in contrast in the clouds, but the results around the text on the plane are a mixed bag - the "U.S. AIR FORCE" text in the middle of the plane has good increase in contrast, while the "F-16" text on the tail has a decrease in contrast. As well, there is loss of detail on the mountains and the aberration along the left and lower rims of the image.

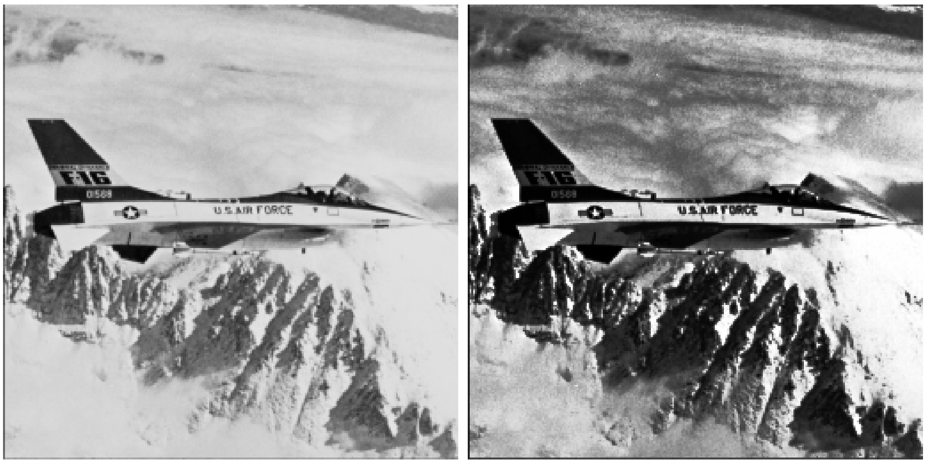

Figure 7: A comparison of `f_16.pgm` with its equalization (right).

Figure 8 compares the original histogram of `f_16.pgm` with the histogram of the new equalized image. The sparseness of bins and consolidation of bins is more apparent than in the previous example, especially around the mode of the image. This probably accounts for the loss of detail in the image, since most of the notable loss of detail happened in brighter regions of the image. These regions all got placed into the same bin, causing them to lose contrast and detail.

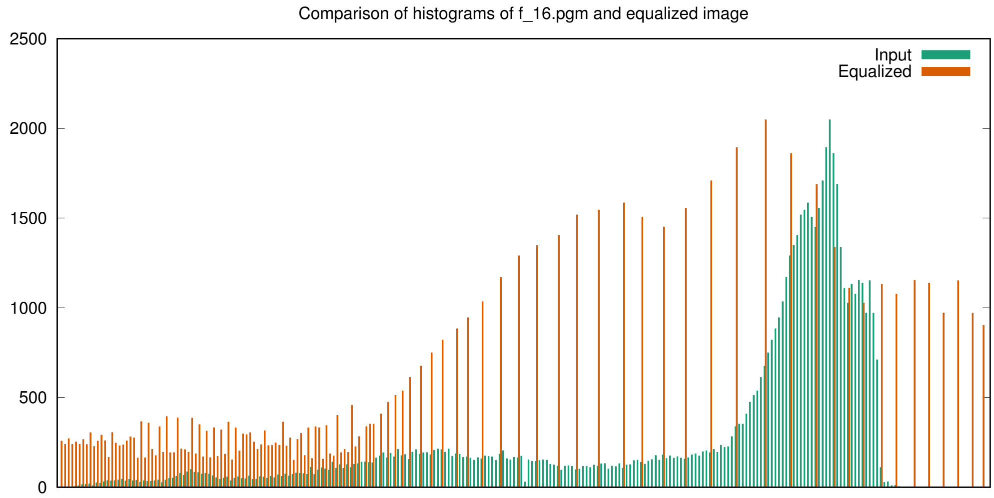

Figure 8: A comparison of histograms of `f_16.pgm` and its equalized version

The source code for generating these histogram comparison figures can be found in listing 8.

4 Histogram Specification

4.1 Theory

As demonstrated in section 3.3, we can transform images to have a similar distribution of pixel intensities as a uniform distribution by using the CDF of the image. Since CDFs are monotonically increasing, the pre-image of a single point is always a continuous interval, and when the CDF is strictly increasing (as is usually the case), the pre-image of a single point is also a single point. In this way, we can naturally define an "inverse" CDF

$$Q_X(x) = \inf F_X^{-1}(\{x\}), \tag{2}$$

known as the "quantile" function. Note that for discrete distributions, the infimum is equal to the minimum, and is chosen because because CDFs are right-continuous (so in a discrete distribution, the minimum is always a possible value). Using the quantile function, we can transform a uniform distribution back to the original distribution. In this way, for two distributions X and Y, F_X transforms X to a uniform distribution and Q_Y transforms a uniform distribution back to Y, so $Q_Y \circ F_X$ transforms X to Y, demonstrated in fig. 9.

$$\begin{tikzcd}
X \arrow[rr, bend left, "Q_Y\circ F_X"] \arrow[dr, "F_X"'] & & Y \arrow[dl, bend right, "F_Y"'] \\
& \mathcal{U} \arrow[ur, bend right, "Q_Y"'] &
\end{tikzcd}$$

Figure 9: A commutative diagram showing how to map from one distribution to another using their CDFs.

In this way, we can perform an image transformation similar to histogram equalization, but with a supplied distribution instead of just a uniform distribution. This can be used for a similar effect as histogram equalization, but with a handpicked distribution to help avoid detail loss like in histogram equalization.

Similarly to histogram equalization, the math isn't exact for discrete distributions, but we can think of discrete distributions as approximations of continuous distributions. There are similar problems with this approximation as with the one made for equalization, but these problems can be lessened with increased quantization levels.

4.2 Implementation

The CDFs of the input image and input histogram are calculated in the same way as detailed in section 3.2. Then, each pixel's intensity is normalized with the input image's normalization constant and unnormalized with the input histogram's normalization constant. This step can be avoided if the images share a normalization constant - notably, when they are the same size and quantization levels. The quantile function (eq. (2)) is then calculated on the unnormalized value using a binary search on the input histogram's CDF. This is done, rather than calculating the quantile function for every possible value, to save memory and because a binary search can take advantage of the sorted nature of the calculated CDF.

The source code for this implementation can be found in listing 7.

4.3 Results and Discussion

Figure 10 shows the results of specifying `boat.pgm` to `sf.pgm`'s histogram. Since `sf.pgm` has much less contrast than `boat.pgm` and fewer extremely dark/light pixels, there is a huge loss of detail.

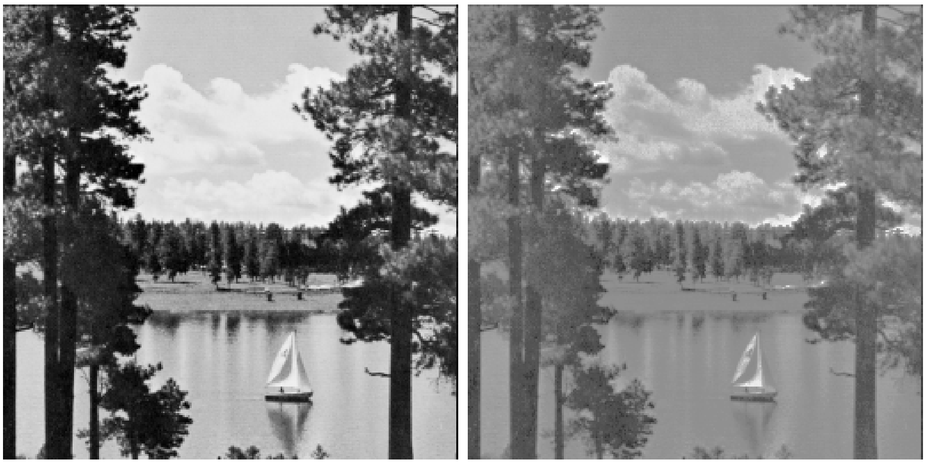

Figure 10: A comparison of `boat.pgm` with its specification to `sf.pgm` (right).

Figure 11 compares the histograms of `boat.pgm`, `sf.pgm`, and the specified output above. As can be seen, the algorithm does a much better job at matching the given histogram than matching a uniform distribution. From just a cursory glance at the comparison, it is hard to tell the output image's histogram apart from the input histogram. This is because the input histogram occupies a narrower band of intensities and has higher peaks than the input image's histogram, allowing the algorithm to consolidate bins effectively. This can be demonstrated by the areas where the algorithm fails to transform the histogram well - near the right side of the histogram - since this region of the input image's histogram is much larger than the input histogram. The reverse transformation would likely not be nearly as successful.

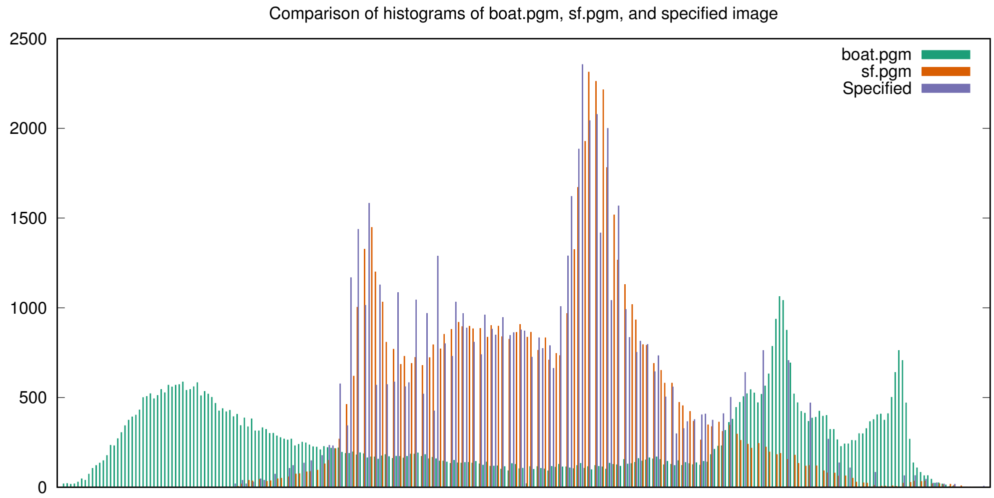

Figure 11: A comparison of histograms of `boat.pgm`, `sf.pgm`, and the specified output as seen in fig. 10.

Figure 12 shows the result of specifying `f_16.pgm` to `peppers.pgm`'s histogram. Since `peppers.pgm` has a large number of of purely black pixels, a similar amount of pixels in `f_16.pgm` are converted to be purely black and there is a resulting large amount of detail loss - especially around darker regions.

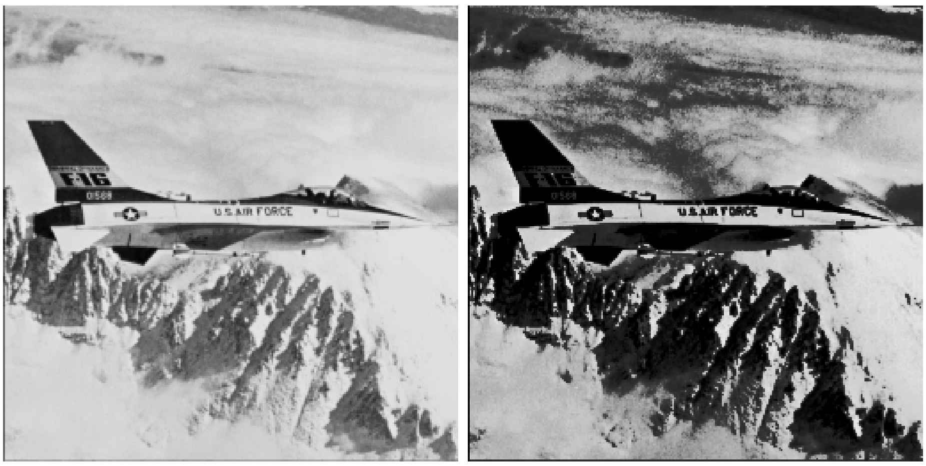

Figure 12: A comparison of `f_16.pgm` with its specification to `peppers.pgm` (right).

Figure 13 compares the histograms of `f_16.pgm`, `peppers.pgm`, and the specified output above. This gives a better look at the strengths and weaknesses of the algorithm - wherever the input image's histogram is below the input histogram (such as to the left), the algorithm succeeds in replicating the input histogram. In other regions (such as to the middle and right), the algorithm can only space out the bins to make the average density in a region similar.

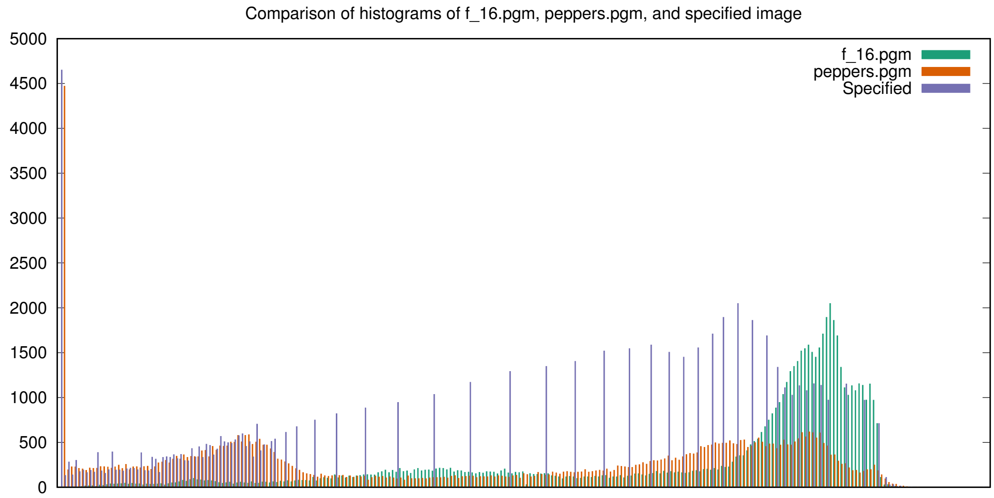

Figure 13: A comparison of histograms of `f_16.pgm`, `peppers.pgm`, and the specified output as seen in fig. 12.

Code Listings

Source code can also be found on the project's GitHub page: `https://github.com/alexander-novo/CS474-PA1`.

Listing 1: Header file for the common `Image` class.

```
// Common/image.h
#pragma once

#include <iostream>

class Image {
public:
  // The type that is used for the value of each pixel
  // As of right now, read and operator<< only work if it is one byte large
  typedef unsigned char pixelT;
  // Struct for reading just the header of an image
  struct Header {
    enum Type {
      COLOR,
      GRAY,
    } type;

    unsigned M, N, Q;

    // Read header from file
    // Throws std::runtime_error for any errors encountered,
    // such as not having a valid PGM/PPM header
    static Header read(std::istream &in);
  };

  Image();
  Image(unsigned, unsigned, unsigned);
  Image(const Image &);  // Copy constructor
  Image(Image &&);       // Move constructor
  ~Image();

  // Read from stream (such as file)
  // Throws std::runtime_error for any errors encountered,
  // such as not being a valid PGM image
  static Image read(std::istream &in);

```

```
  // Output to stream (such as file)
  friend std::ostream &operator<<(std::ostream &out, const Image &im);

  // Pixel access - works like 2D array i.e. image[i][j]
  pixelT *operator[](unsigned i);
  const pixelT *operator[](unsigned i) const;
  Image &operator=(const Image &rhs);  // Assignment
  Image &operator=(Image &&rhs);       // Move

  // Read-only properties
  pixelT *const &pixels  = pixelValue;
  const unsigned &rows   = M;
  const unsigned &cols   = N;
  const unsigned &maxVal = Q;

private:
  Image(unsigned, unsigned, unsigned, pixelT *);
  unsigned M, N, Q;
  pixelT *pixelValue;
};

std::ostream &operator<<(std::ostream &out, const Image::Header &head);
```

Listing 2: Implementation file for the common `Image` class.

```
// Common/image.cpp
#include "image.h"

#include <cassert>
#include <cstdlib>
#include <exception>

Image::Image() : Image(0, 0, 0, nullptr) {}

Image::Image(unsigned M, unsigned N, unsigned Q) : Image(M, N, Q, new
↪  Image::pixelT[M * N]) {}

Image::Image(const Image& oldImage) : Image(oldImage.M, oldImage.N, oldImage.Q) {
  for (unsigned i = 0; i < M * N; i++) { pixelValue[i] = oldImage.pixelValue[i]; }
}

// Move constructor - take old image's pixel values and make old image invalid
Image::Image(Image&& oldImage) : Image(oldImage.M, oldImage.N, oldImage.Q,
↪  oldImage.pixelValue) {
  oldImage.M = oldImage.N = oldImage.Q = 0;
  oldImage.pixelValue                  = nullptr;
}

Image::Image(unsigned M, unsigned N, unsigned Q, pixelT* pixels)
    : M(M), N(N), Q(Q), pixelValue(pixels) {}

Image::~Image() {
```

```
  if (pixelValue != nullptr) { delete[] pixelValue; }
}

// Slightly modified version of readImage() function provided by Dr. Bebis
Image Image::read(std::istream& in) {
  int N, M, Q;
  unsigned char* charImage;
  char header[100], *ptr;

  static_assert(sizeof(Image::pixelT) == 1,
                "Image reading only supported for single-byte pixel types.");

  // read header
  in.getline(header, 100, '\n');
  if ((header[0] != 'P') || (header[1] != '5')) { throw std::runtime_error("Image is not PGM!"); }

  in.getline(header, 100, '\n');
  while (header[0] == '#') in.getline(header, 100, '\n');

  N = strtol(header, &ptr, 0);
  M = atoi(ptr);

  in.getline(header, 100, '\n');
  Q = strtol(header, &ptr, 0);

  if (Q > 255) throw std::runtime_error("Image cannot be read correctly (Q > 255)!");

  charImage = new unsigned char[M * N];

  in.read(reinterpret_cast<char*>(charImage), (M * N) * sizeof(unsigned char));

  if (in.fail()) throw std::runtime_error("Image has wrong size!");

  return Image(M, N, Q, charImage);
}

// Slightly modified version of writeImage() function provided by Dr. Bebis
std::ostream& operator<<(std::ostream& out, const Image& im) {
  static_assert(sizeof(Image::pixelT) == 1,
                "Image writing only supported for single-byte pixel types.");

  out << "P5" << std::endl;
  out << im.N << " " << im.M << std::endl;
  out << im.Q << std::endl;

  out.write(reinterpret_cast<char*>(im.pixelValue), (im.M * im.N) * sizeof(unsigned char));

  if (out.fail()) throw std::runtime_error("Something failed with writing image.");
}
```

```

Image& Image::operator=(const Image& rhs) {
  if (pixelValue != nullptr) delete[] pixelValue;

  M = rhs.M;
  N = rhs.N;
  Q = rhs.Q;

  pixelValue = new pixelT[M * N];

  for (unsigned i = 0; i < M * N; i++) pixelValue[i] = rhs.pixelValue[i];

  return *this;
}

Image& Image::operator=(Image&& rhs) {
  if (pixelValue != nullptr) delete[] pixelValue;

  M          = rhs.M;
  N          = rhs.N;
  Q          = rhs.Q;
  pixelValue = rhs.pixelValue;

  rhs.M = rhs.N = rhs.Q = 0;
  rhs.pixelValue        = nullptr;

  return *this;
}

Image::pixelT* Image::operator[](unsigned i) {
  return pixelValue + i * N;
}

const Image::pixelT* Image::operator[](unsigned i) const {
  return pixelValue + i * N;
}

// Slightly modified version of readImageHeader() function provided by Dr. Bebis
Image::Header Image::Header::read(std::istream& in) {
  unsigned char* charImage;
  char header[100], *ptr;
  Header re;

  // read header
  in.getline(header, 100, '\n');
  if ((header[0] == 'P') && (header[1] == '5')) {
    re.type = GRAY;
  } else if ((header[0] == 'P') && (header[1] == '6')) {
    re.type = COLOR;
  } else
    throw std::runtime_error("Image is not PGM or PPM!");

  in.getline(header, 100, '\n');
```

```
  while (header[0] == '#') in.getline(header, 100, '\n');

  re.N = strtol(header, &ptr, 0);
  re.M = atoi(ptr);

  in.getline(header, 100, '\n');

  re.Q = strtol(header, &ptr, 0);

  return re;
}

std::ostream& operator<<(std::ostream& out, const Image::Header& head) {
  switch (head.type) {
    case Image::Header::Type::COLOR:
      out << "PPM Color ";
      break;
    case Image::Header::Type::GRAY:
      out << "PGM Grayscale ";
  }
  out << "Image size " << head.M << " x " << head.N << " and max value of " <<
    ↪ head.Q << ".";
}
```

Listing 3: Implementation file for the Histogram supporting library.

```
// Common/histogram_tools.cpp
#include "histogram_tools.h"

#include <algorithm>
#include <iostream>

void Histogram::print(unsigned* histogram, unsigned bins, unsigned width, unsigned
↪ height) {
  // An adjusted histogram, which has been binned
  unsigned* binnedHistogram = new unsigned[width];
  // Maximum number of original bins represented by each new bin
  // Each bin is this size, except maybe the last bin (which may be smaller)
  unsigned binSize = 1 + (bins - 1) / width;
  // The maximum number of observations in all bins
  unsigned maxBin = 0;

  // Calculate new binnedHistogram and maxBin
#pragma omp parallel for reduction(max : maxBin)
  for (unsigned i = 0; i < width; i++) {
    binnedHistogram[i] = 0;
    for (unsigned j = binSize * i; j < binSize * (i + 1) && j < bins; j++) {
      binnedHistogram[i] += histogram[j];
    }
    maxBin = std::max(binnedHistogram[i], maxBin);
  }

```

```
  // The maximum number of observations each tick can represent
  // May represent as few as 1, if present on the top of a histogram bar
  unsigned tickSize = 1 + (maxBin - 1) / height;

  for (unsigned i = 1; i <= height; i++) {
    unsigned threshold = (height - i) * tickSize;
    for (unsigned j = 0; j < width; j++) {
      if (binnedHistogram[j] > threshold)
        std::cout << '*';
      else
        std::cout << ' ';
    }
    std::cout << '\n';
  }

  delete[] binnedHistogram;
}
```

Listing 4: Implementation file for the `sample` program.

```
#include <iostream>
#include <fstream>
#include <sstream>

#include "../Common/image.h"

/*
  Subsamples and image based on the sampling factor
  @Param: image - the input image that will be sampled
  @Param: subsample_factor - the factor by which to sample
  @Return: void
*/
void subsample_image(Image& image, int subsample_factor){

  // iterate through image to get the sample
  for(int i=0; i<image.cols; i += subsample_factor){
       for(int j=0; j<image.rows; j += subsample_factor) {

         // save the sampled pixel
         int pixelSample = image[i][j];

         // Modify neighbor pixels to match the sampled pixel
         for (int k = 0; k < subsample_factor; k++)
         {
           for (int l = 0; l < subsample_factor; l++)
           {
             image[i + k][j + l] = pixelSample;
           }
         }
      }
    }
}
```

```

int main(int argc, char** argv) {

   int M, N, Q;
   bool type;
   int val;
   int subsample_factor;
   std::istringstream ss(argv[3]);

   // Get sampling factor, error checking
   if(ss >> subsample_factor) {
     if(256 % subsample_factor != 0 || subsample_factor > 256){
       std::cout << "Error: Subsample factor should be power of 2 less than 256"
        ↪  << std::endl;
       return 1;

     }
   }

   //read image
   std::ifstream inFile(argv[1]);

   Image image = Image::read(inFile);

  std::cout << "Question 1: Sampling." << std::endl;

  // sample the image
  subsample_image(image, subsample_factor);

  // Save output image
  std::ofstream outFile;
  outFile.open(argv[2]);
  outFile << image;
  outFile.close();

  return 0;
}
```

Listing 5: Implementation file for the `quantize` program.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>

#include "../Common/image.h"

/*
  Quantizes an image based on the quantization level
  @Param: image - the input image that will be quantized
  @Param: quantization_level - the number of gray level values to use
  @Return: void
```

```
*/
void quantize_image(Image& image, int quantization_level){

  int offset = 256 / quantization_level;

  // New set of possible pixel values
  std::vector<int> newPixelValues;

  // calculate new values for pixels
  for (int i = 0; i < quantization_level; i++)
      newPixelValues.push_back(i * offset);


  // For each pixel
  for(int i=0; i<image.cols; i++) {
       for(int j=0; j<image.rows; j++) {

         // current pixel
         int pixelValue = image[i][j];

        // index for new pixel value
         int index = pixelValue / offset;

        // update image pixel
        image[i][j] = newPixelValues[index];
      }
    }
}

int main(int argc, char** argv) {
  int M, N, Q;
   bool type;
   int val;
   int quantization_level;
   std::istringstream ss(argv[3]);

  // GEt quantization level
   if(ss >> quantization_level) {
     if(quantization_level > 256){
       std::cout << "Error: Quantization level should be less than 256" << std::endl;
       return 1;

     }
   }

  // Read original image
   std::ifstream inFile(argv[1]);

   Image image = Image::read(inFile);

  std::cout << "Question 2: Quantization." << std::endl;

```

```cpp
  // Quantize the image
  quantize_image(image, quantization_level);

  // Save output image
  std::ofstream outFile;
  outFile.open(argv[2]);
  outFile << image;
  outFile.close();

  return 0;
}
```

Listing 6: Implementation file for the `equalize` program.

```cpp
// Q3-Equalization/main.cpp
#include <cstring>
#include <fstream>
#include <iostream>
#include <map>
#include <mutex>

#include "../Common/histogram_tools.h"
#include "../Common/image.h"

// Struct for inputting arguments from command line
struct Arguments {
  char *inputImagePath, *outImagePath;
  Image inputImage;
  std::ofstream outFile;
  unsigned histogramWidth = 64, histogramHeight = 10;
  bool plot = false;
  std::ofstream plotFile;
};

void equalize(Arguments& arg);
bool verifyArguments(int argc, char** argv, Arguments& arg, int& err);
void printHelp();

int main(int argc, char** argv) {
  int err;
  Arguments arg;

  if (!verifyArguments(argc, argv, arg, err)) { return err; }

  equalize(arg);

  return 0;
}

void equalize(Arguments& arg) {
  unsigned* histogram    = new unsigned[arg.inputImage.maxVal + 1];
  unsigned* newHistogram = new unsigned[arg.inputImage.maxVal + 1];
```

```
  unsigned* cdf          = new unsigned[arg.inputImage.maxVal + 1];
  std::mutex* locks      = new std::mutex[arg.inputImage.maxVal + 1];
  // Initialise histogram bins to be empty
#pragma omp parallel for
  for (unsigned i = 0; i <= arg.inputImage.maxVal; i++) {
    histogram[i] = newHistogram[i] = 0;
  }

  // Create histogram
#pragma omp parallel for
  for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
    unsigned bin = arg.inputImage.pixels[i];
    locks[bin].lock();
    histogram[bin]++;
    locks[bin].unlock();
  }

  // Calculate CDF
  cdf[0] = histogram[0];
  for (unsigned i = 1; i <= arg.inputImage.maxVal; i++) {
    cdf[i] = cdf[i - 1] + histogram[i];
  }

  // Tranform image with the CDF
#pragma omp parallel for
  for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
    Image::pixelT& pixelVal = arg.inputImage.pixels[i];
    pixelVal                = cdf[pixelVal] * arg.inputImage.maxVal /
               (arg.inputImage.rows * arg.inputImage.cols);
  }

  // Write new transformed image out
  arg.outFile << arg.inputImage;
  arg.outFile.close();

  // Calculate histogram of new image
#pragma omp parallel for
  for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
    unsigned bin = arg.inputImage.pixels[i];
    locks[bin].lock();
    newHistogram[bin]++;
    locks[bin].unlock();
  }

  // Print histograms
  std::cout << "\nHistogram of input image \"" << arg.inputImagePath << "\":\n";
  Histogram::print(histogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
                   arg.histogramHeight);

  std::cout << "\nHistogram of output image \"" << arg.outImagePath << "\":\n";
  Histogram::print(newHistogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
                   arg.histogramHeight);

```

```
  // Print histogram data for plot file
  if (arg.plot) {
    arg.plotFile << "Image Input Equalized\n";
    for (unsigned i = 0; i <= arg.inputImage.maxVal; i++) {
      arg.plotFile << i << "    " << histogram[i] << "    " << newHistogram[i]
                   << '\n';
    }
    arg.plotFile.close();
  }

  delete[] histogram;
  delete[] newHistogram;
  delete[] cdf;
  delete[] locks;
}

bool verifyArguments(int argc, char** argv, Arguments& arg, int& err) {
  // If there are not the minimum number of arguments, print help and leave
  if (argc < 2 ||
      (argc < 3 && strcmp(argv[1], "-h") && strcmp(argv[1], "--help"))) {
    std::cout << "Missing operand.\n";
    err = 1;
    printHelp();
    return false;
  }

  // If the user asks for the help menu, print help and leave
  if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
    printHelp();
    return false;
  }

  // Find optional argument switches
  for (unsigned i = 3; i < argc; i++) {
    if (!strcmp(argv[i], "-width")) {
      if (i + 1 >= argc) {
        std::cout << "Missing width";
        err = 1;
        printHelp();
        return false;
      }

      arg.histogramWidth = strtoul(argv[i + 1], nullptr, 10);
      if (arg.histogramWidth == 0) {
        std::cout << "Width \"" << argv[i + 1]
                  << "\" could not be recognised as a positive integer.";
        err = 2;
        return false;
      }

      i++;
    } else if (!strcmp(argv[i], "-height")) {
      if (i + 1 >= argc) {
```

```
        std::cout << "Missing height";
        err = 1;
        break;
      }

      arg.histogramHeight = strtoul(argv[i + 1], nullptr, 10);
      if (arg.histogramHeight == 0) {
        std::cout << "Height \"" << argv[i + 1]
                  << "\" could not be recognised as a positive integer.";
        err = 2;
        return false;
      }

      i++;
    } else if (!strcmp(argv[i], "-p")) {
      if (i + 1 >= argc) {
        std::cout << "Missing plot output file";
        err = 1;
        break;
      }

      arg.plot = true;
      arg.plotFile.open(argv[i + 1]);

      if (!arg.plotFile) {
        std::cout << "Plot file \"" << argv[i + 1]
                  << "\" could not be opened";
        err = 2;
        return false;
      }

      i++;
    }
  }

  // Required arguments
  arg.inputImagePath = argv[1];
  std::ifstream inFile(argv[1]);
  try {
    arg.inputImage = Image::read(inFile);
  } catch (std::exception& e) {
    std::cout << "Image \"" << argv[1] << "\"failed to be read: \"" << e.what()
              << "\"\n";
    err = 2;
    return false;
  }

  arg.outImagePath = argv[2];
  arg.outFile.open(argv[2]);
  if (!arg.outFile) {
    std::cout << "Could not open \"" << argv[2] << "\"\n";
    err = 2;
    return false;
```

```
  }

  return true;
}

void printHelp() {
  std::cout
      << "Usage: equalize <image> <output> [options]    (1)\n"
      << "   or: equalize -h                            (2)\n\n"
      << "(1) Take an image file as input, equalize its histogram,\n"
      << "    and write new image to output file. Displays the original\n"
      << "    histogram and the new equalized histogram.\n"
      << "(2) Print this help menu\n\n"
      << "Options:\n"
      << "  -width <width>    Number of visual histogram bins\n"
      << "  -height <height>  Height of visual histogram (in lines)\n"
      << "  -p <file>         Send histogram plotting data to a file for gnuplot\n";
}
```

Listing 7: Implementation file for the `specify` program.

```
// Q4-Specification/main.cpp
#include <cstring>
#include <fstream>
#include <iostream>
#include <mutex>
#include <regex>
#include <vector>

#include "../Common/histogram_tools.h"
#include "../Common/image.h"

// Struct for inputting arguments from command line
struct Arguments {
  char *inputImagePath, *outImagePath, *histogramPath;
  Image inputImage;
  std::ifstream histogramFile;
  std::ofstream outFile;
  unsigned histogramWidth = 64, histogramHeight = 10;
  bool plot = false;
  std::ofstream plotFile;
};

int specify(Arguments& arg);
void printHistogram(unsigned* histogram, const Arguments& arg);
bool verifyArguments(int argc, char** argv, Arguments& arg, int& err);
void printHelp();

int main(int argc, char** argv) {
  int err;
  Arguments arg;
```

```

  if (!verifyArguments(argc, argv, arg, err)) { return err; }

  return specify(arg);
}

int specify(Arguments& arg) {
  unsigned* histogram    = new unsigned[arg.inputImage.maxVal + 1];
  unsigned* newHistogram = new unsigned[arg.inputImage.maxVal + 1];
  unsigned* cdf          = new unsigned[arg.inputImage.maxVal + 1];
  std::mutex* locks      = new std::mutex[arg.inputImage.maxVal + 1];
  std::vector<unsigned> targetHistogram;
  unsigned targetPixels = 0;  // Number of pixels in the target histogram

  // Initialise histogram bins to be empty
#pragma omp parallel for
  for (unsigned i = 0; i <= arg.inputImage.maxVal; i++) {
    histogram[i] = newHistogram[i] = 0;
  }

  // Create input image histogram
#pragma omp parallel for
  for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
    unsigned bin = arg.inputImage.pixels[i];
    locks[bin].lock();
    histogram[bin]++;
    locks[bin].unlock();
  }

  // Start with enough space to hold our input image. If we need more, we can get
  // more, but we're probably working with similarly-valued images.
  targetHistogram.reserve(arg.inputImage.maxVal + 1);

  // Read in target histogram
  std::string line;
  std::regex rHistogram("^([[:digit:]]+)[[:space:]]+([[:digit:]]+).*");
  std::smatch matches;
  while (arg.histogramFile) {
    std::getline(arg.histogramFile, line);
    if (!std::regex_match(line, matches, rHistogram)) continue;

    if (stoul(matches[1].str()) != targetHistogram.size()) {
      std::cout << "Error in reading histogram file \"" << arg.histogramPath
                << "\":\n"
                << "Bucket \"" << stoul(matches[1].str())
                << "\" was expected to be \"" << targetHistogram.size()
                << "\".";
    }

    targetHistogram.push_back(stoul(matches[2].str()));
    targetPixels += targetHistogram.back();
  }
  arg.histogramFile.close();
```

```

  // Calculate CDFs
  std::vector<unsigned> targetCDF(targetHistogram.size());
  cdf[0]       = histogram[0];
  targetCDF[0] = targetHistogram[0];

  for (unsigned i = 1; i <= arg.inputImage.maxVal; i++) {
    cdf[i] = cdf[i - 1] + histogram[i];
  }
  for (unsigned i = 1; i < targetHistogram.size(); i++) {
    targetCDF[i] = targetCDF[i - 1] + targetHistogram[i];
  }

// Tranform input image with its CDF and inverse CDF of target histogram
#pragma region CDF transformation
  // Separate cases for if the images have different dimensions/maxVal, to make
  // calculation easier
  if (arg.inputImage.maxVal == targetHistogram.size() - 1) {
    if (arg.inputImage.rows * arg.inputImage.cols == targetPixels) {
#pragma omp parallel for
      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
        Image::pixelT& pixelVal = arg.inputImage.pixels[i];

        unsigned inversePixel = cdf[pixelVal];

        // Since there are 257 possible values (256 different "found" objects
        // + none found), clamp to 256 possible values
        pixelVal = std::max<unsigned>(
                       1, targetCDF.rend() -
                              std::lower_bound(targetCDF.rbegin(),
                                               targetCDF.rend(), inversePixel,
                                               std::greater<unsigned>())) -
                   1;
      }
    } else {
#pragma omp parallel for
      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
        Image::pixelT& pixelVal = arg.inputImage.pixels[i];

        unsigned inversePixel = cdf[pixelVal] * targetPixels /
                                (arg.inputImage.rows * arg.inputImage.cols);

        // Since there are 257 possible values (256 different "found" objects
        // + none found), clamp to 256 possible values
        pixelVal = std::max<unsigned>(
                       1, targetCDF.rend() -
                              std::lower_bound(targetCDF.rbegin(),
                                               targetCDF.rend(), inversePixel,
                                               std::greater<unsigned>())) -
                   1;
      }
    }
  } else if (arg.inputImage.rows * arg.inputImage.cols == targetPixels) {
```

```
#pragma omp parallel for
    for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
      Image::pixelT& pixelVal = arg.inputImage.pixels[i];

      unsigned inversePixel =
          cdf[pixelVal] * arg.inputImage.maxVal / (targetHistogram.size() - 1);

      // Since there are 257 possible values (256 different "found" objects
      // + none found), clamp to 256 possible values
      pixelVal =
          std::max<unsigned>(
              1, targetCDF.rend() -
                     std::lower_bound(targetCDF.rbegin(), targetCDF.rend(),
                                      inversePixel, std::greater<unsigned>())) -
          1;
    }
  } else {
    // In this case, we need to do math with ull because of the multiplications
    // overflowing The result after division should fit within an unsigned, though
#pragma omp parallel for
    for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
      Image::pixelT& pixelVal = arg.inputImage.pixels[i];

      unsigned inversePixel = ((unsigned long long) cdf[pixelVal]) *
                              arg.inputImage.maxVal * targetPixels /
                              (arg.inputImage.rows * arg.inputImage.cols *
                               (targetHistogram.size() - 1));
      // Since there are 257 possible values (256 different "found" objects
      // + none found), clamp to 256 possible values
      pixelVal =
          std::max<unsigned>(
              1, targetCDF.rend() -
                     std::lower_bound(targetCDF.rbegin(), targetCDF.rend(),
                                      inversePixel, std::greater<unsigned>())) -
          1;
    }
  }
#pragma endregion CDF transformation

  // Write new transformed image out
  arg.outFile << arg.inputImage;
  arg.outFile.close();

  // Calculate histogram of new image
#pragma omp parallel for
  for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
    unsigned bin = arg.inputImage.pixels[i];
    locks[bin].lock();
    newHistogram[bin]++;
    locks[bin].unlock();
  }

  // Print histograms
```

```cpp
  std::cout << "\nHistogram of input image \"" << arg.inputImagePath << "\":\n";
  Histogram::print(histogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
                   arg.histogramHeight);

  std::cout << "\nInput histogram \"" << arg.histogramPath << "\":\n";
  Histogram::print(&targetHistogram.front(), targetHistogram.size(),
                   arg.histogramWidth, arg.histogramHeight);

  std::cout << "\nHistogram of output image \"" << arg.outImagePath << "\":\n";
  Histogram::print(newHistogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
                   arg.histogramHeight);

  // Print histogram data for plot file
  if (arg.plot) {
    arg.plotFile << "Source Input-Image Input-Histogram Specified\n";
    unsigned i;
    for (i = 0; i <= arg.inputImage.maxVal && i < targetHistogram.size(); i++) {
      arg.plotFile << i << "    " << histogram[i] << "    "
                   << targetHistogram[i] << "    " << newHistogram[i] << '\n';
    }
    arg.plotFile.close();
  }

  delete[] histogram;
  delete[] newHistogram;
  delete[] cdf;
  delete[] locks;

  return 0;
}

bool verifyArguments(int argc, char** argv, Arguments& arg, int& err) {
  if (argc < 2 ||
      (argc < 4 && strcmp(argv[1], "-h") && strcmp(argv[1], "--help"))) {
    std::cout << "Missing operand.\n";
    err = 1;
    printHelp();
    return false;
  }

  if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
    printHelp();
    return false;
  }

  // Find optional argument switches
  for (unsigned i = 4; i < argc; i++) {
    if (!strcmp(argv[i], "-width")) {
      if (i + 1 >= argc) {
        std::cout << "Missing width";
        err = 1;
        printHelp();
        return false;
```

```
      }

      arg.histogramWidth = strtoul(argv[i + 1], nullptr, 10);
      if (arg.histogramWidth == 0) {
        std::cout << "Width \"" << argv[i + 1]
                  << "\" could not be recognised as a positive integer.";
        err = 2;
        return false;
      }

      i++;
    } else if (!strcmp(argv[i], "-height")) {
      if (i + 1 >= argc) {
        std::cout << "Missing height";
        err = 1;
        break;
      }

      arg.histogramHeight = strtoul(argv[i + 1], nullptr, 10);
      if (arg.histogramHeight == 0) {
        std::cout << "Height \"" << argv[i + 1]
                  << "\" could not be recognised as a positive integer.";
        err = 2;
        return false;
      }

      i++;
    } else if (!strcmp(argv[i], "-p")) {
      if (i + 1 >= argc) {
        std::cout << "Missing plot output file";
        err = 1;
        break;
      }

      arg.plot = true;
      arg.plotFile.open(argv[i + 1]);

      if (!arg.plotFile) {
        std::cout << "Plot file \"" << argv[i + 1]
                  << "\" could not be opened";
        err = 2;
        return false;
      }

      i++;
    }
  }

  // Required arguments
  arg.inputImagePath = argv[1];
  std::ifstream inFile(argv[1]);
  try {
    arg.inputImage = Image::read(inFile);
```

```
  } catch (std::exception& e) {
    std::cout << "Image \"" << argv[1] << "\"failed to be read: \"" << e.what()
              << "\"\n";
    err = 2;
    return false;
  }

  arg.histogramPath = argv[2];
  arg.histogramFile.open(argv[2]);
  if (!arg.histogramFile) {
    std::cout << "Could not open \"" << argv[2] << "\"\n";
    err = 2;
    return false;
  }

  arg.outImagePath = argv[3];
  arg.outFile.open(argv[3]);
  if (!arg.outFile) {
    std::cout << "Could not open \"" << argv[3] << "\"\n";
    err = 2;
    return false;
  }

  return true;
}

void printHelp() {
  std::cout
      << "Usage: specify <image> <histogram> <output> [options]   (1)\n"
      << "   or: specify -h                                        (2)\n\n"
      << "(1) Take an image file as input, change its histogram to the\n"
      << "    specified histogram, and write new image to output file.\n"
      << "    Displays the original histogram and the new equalized histogram.\n"
      << "    Histogram files can be obtained by running 'equalize' with\n"
      << "    the -p flag set (or 'specify' with the -p falg set).\n"
      << "(2) Print this help menu\n\n"
      << "Options:\n"
      << "  -width <width>    Number of visual histogram bins\n"
      << "  -height <height>  Height of visual histogram (in lines)\n"
      << "  -p <file>         Send histogram plotting data to a file for gnuplot\n";
}
```

Listing 8: `gnuplot` plotting file for generating two-histogram comparison plots. Used for generating comparison plots in section 3.3.

```
# Q3-Equalization/plot-histograms.plt
# A gnuplot plotting file to plot the two histograms of data from equalize with the -p switch
if (!exists("outfile")) outfile='plot.eps'

if (!exists("imageName")) {
  set title "Comparison of histograms of input and output (equalized) images"
} else {
  set title "Comparison of histograms of " . imageName . " and equalized image" noenhanced
}

set terminal postscript eps enhanced color size 6,3
set output outfile
set style data histogram
set style histogram cluster gap 1
set style fill solid
set boxwidth 0.9

unset xtics

# Colors chosen using ColorBrewer 2.0 qualitative scheme "Dark2"
# https://colorbrewer2.org/#type=qualitative&scheme=Dark2&n=3
plot infile using 2:xtic(1) ti col linecolor rgb "#1b9e77",\
  '' u 3 ti col linecolor rgb "#d95f02"
```

Listing 9: `gnuplot` plotting file for generating three-histogram comparison plots. Used for generating comparison plots in section 4.3.

```
# Q4-Specification/plot-histograms.plt
# A gnuplot plotting file to plot the three histograms of data from specify with the -p switch
if (!exists("outfile")) outfile='plot.eps'

if (!exists("imageName") || !exists("histoName")) {
  set title "Comparison of histograms of input image, input histogram, and output image"
} else {
  set title "Comparison of histograms of " . imageName . ", " . histoName . ", and specified image" noenhanced
}

set terminal postscript eps enhanced color size 6,3
set output outfile
set style data histogram
set style histogram cluster gap 1
set style fill solid
set boxwidth 0.9

unset xtics

# Colors chosen using ColorBrewer 2.0 qualitative scheme "Dark2"
# https://colorbrewer2.org/#type=qualitative&scheme=Dark2&n=3
plot infile using 2:xtic(1) ti imageName noenhanced linecolor rgb "#1b9e77",\
  '' u 3 ti histoName noenhanced linecolor rgb "#d95f02",\
  '' u 4 ti col linecolor rgb "#7570b3"
```