Programming Assignment 1

CS 474

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

Alexander Novotny
50% Work

Matthew Lyman Page
50% Work

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

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

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.

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.

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,

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).

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.

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.
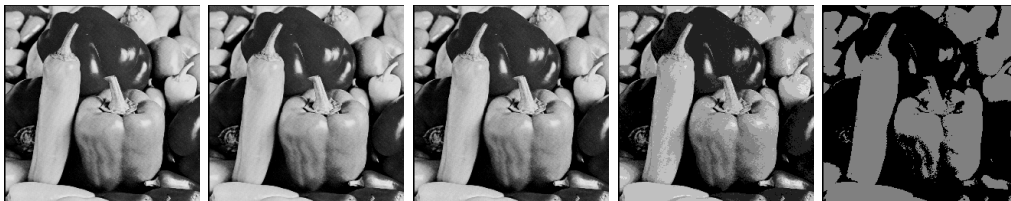

Figure 4: A comparison of `peppers.pgm` with varying quantization levels (From left to right: L=256, L=128, L=32, L=8, and L=2).

Overall, it appears that the effects of image quantization does not degrade an image as much as image sub-sampled. With both images, the effects of image sub-sampled become noticeable even after sub-sapling 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,

$$= f_X(F_X^{-1}(y)) \left| \frac{1}{F_X'(F_X^{-1}(y))} \right|$$

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

$$= 1,$$

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:

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

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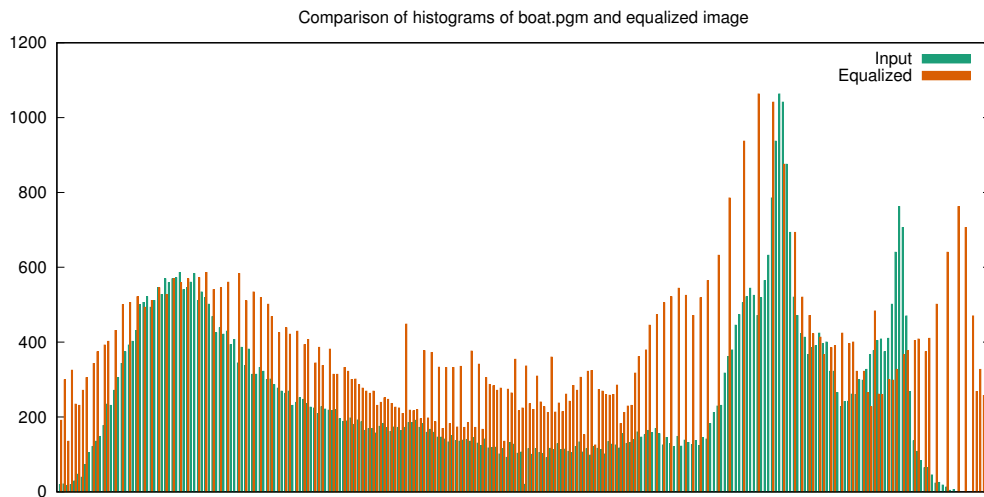
Figure 6: A comparison of histograms of `boat.pgm` and its equalized version

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.
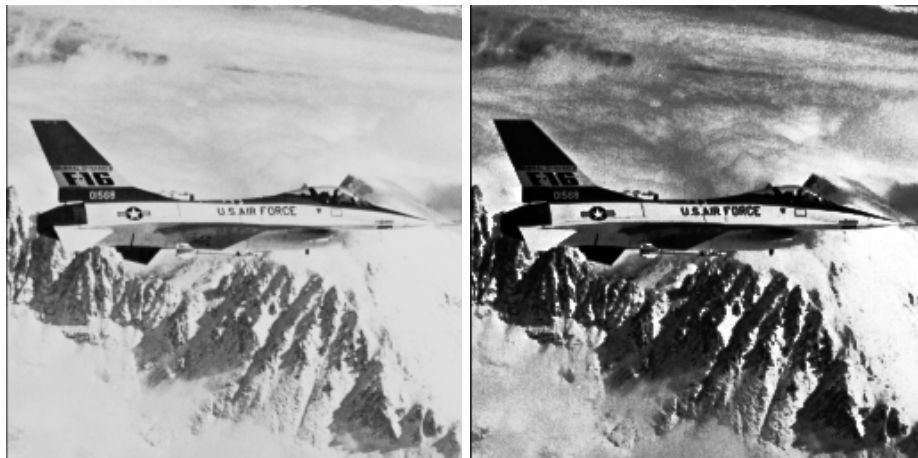

Figure 7: A comparison of `f_16.pgm` with its equalization (right).

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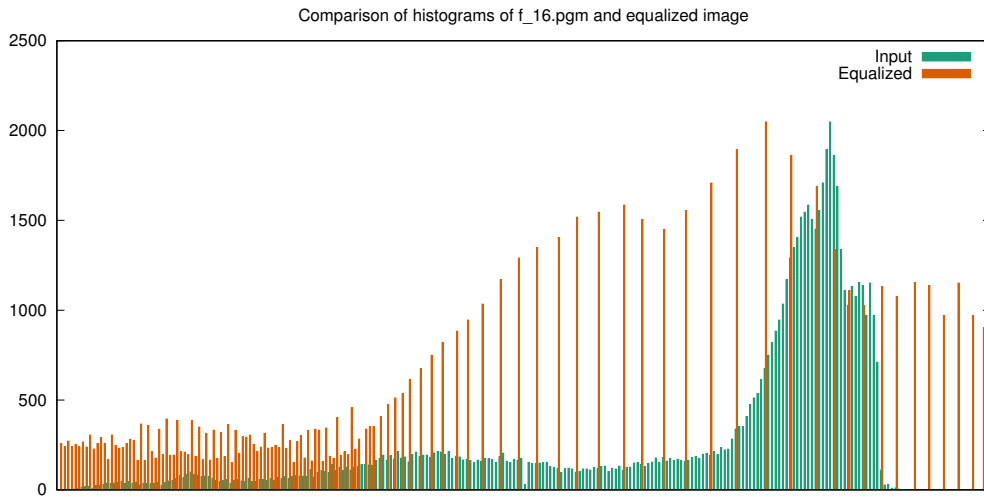
Figure 8: A comparison of histograms of `f_16.pgm` and its equalized version

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.

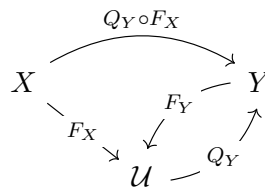Figure 9: A commutative diagram showing how to map from one distribution to another using their CDFs.

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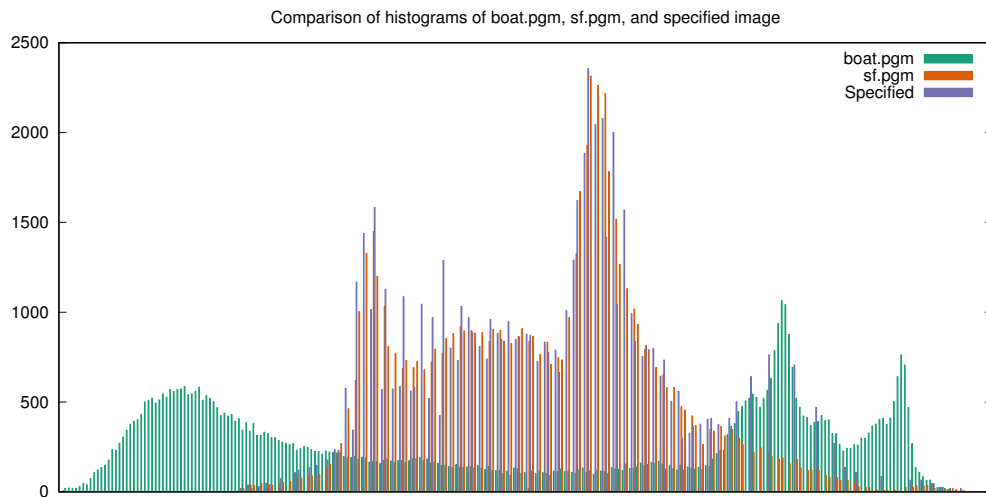
Figure 11: A comparison of histograms of `boat.pgm`, `sf.pgm`, and the specified output as seen in fig. 10.

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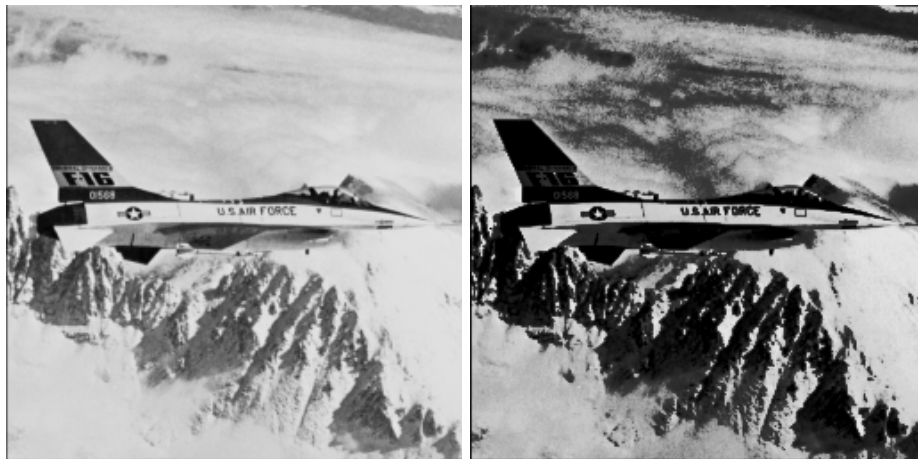
Figure 12: A comparison of `f_16.pgm` with its specification to `peppers.pgm` (right).

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-nov
o/CS474-PA1`.

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

```cpp
// 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);

```
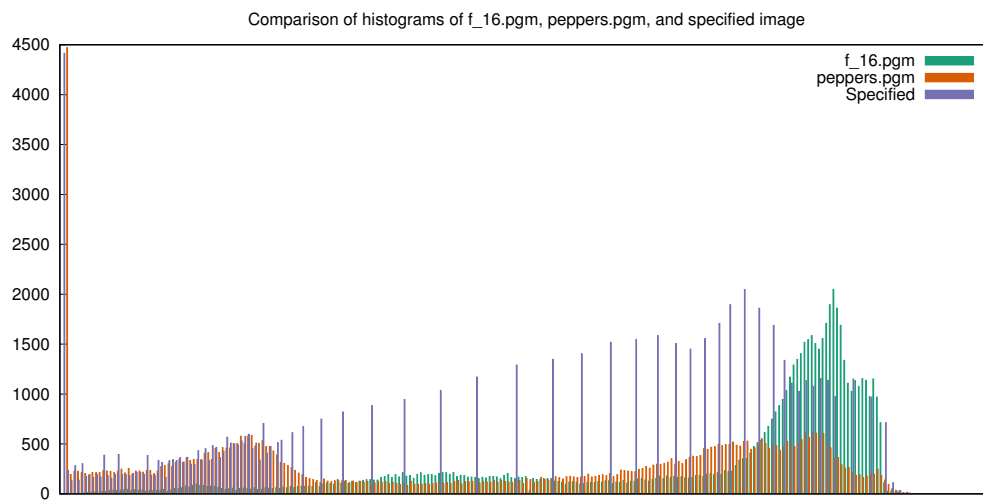
```cpp
37      // Output to stream (such as file)
38      friend std::ostream &operator<<(std::ostream &out, const Image &im);
39
40      // Pixel access - works like 2D array i.e. image[i][j]
41      pixelT *operator[](unsigned i);
42      const pixelT *operator[](unsigned i) const;
43      Image &operator=(const Image &rhs);   // Assignment
44      Image &operator=(Image &&rhs);        // Move
45
46      // Read-only properties
47      pixelT *const &pixels  = pixelValue;
48      const unsigned &rows   = M;
49      const unsigned &cols   = N;
50      const unsigned &maxVal = Q;
51
52  private:
53      Image(unsigned, unsigned, unsigned, pixelT *);
54      unsigned M, N, Q;
55      pixelT *pixelValue;
56  };
57
58  std::ostream &operator<<(std::ostream &out, const Image::Header &head);
```

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

```cpp
1   // Common/image.cpp
2   #include "image.h"
3
4   #include <cassert>
5   #include <cstdlib>
6   #include <exception>
7
8   Image::Image() : Image(0, 0, 0, nullptr) {}
9
10  Image::Image(unsigned M, unsigned N, unsigned Q) : Image(M, N, Q, new
    ↪   Image::pixelT[M * N]) {}
11
12  Image::Image(const Image& oldImage) : Image(oldImage.M, oldImage.N, oldImage.Q) {
13      for (unsigned i = 0; i < M * N; i++) { pixelValue[i] = oldImage.pixelValue[i]; }
14  }
15
16  // Move constructor - take old image's pixel values and make old image invalid
17  Image::Image(Image&& oldImage) : Image(oldImage.M, oldImage.N, oldImage.Q,
    ↪   oldImage.pixelValue) {
18      oldImage.M = oldImage.N = oldImage.Q = 0;
19      oldImage.pixelValue                  = nullptr;
20  }
21
22  Image::Image(unsigned M, unsigned N, unsigned Q, pixelT* pixels)
23      : M(M), N(N), Q(Q), pixelValue(pixels) {}
24
25  Image::~Image() {
```

```cpp
26      if (pixelValue != nullptr) { delete[] pixelValue; }
27    }
28
29    // Slightly modified version of readImage() function provided by Dr. Bebis
30    Image Image::read(std::istream& in) {
31      int N, M, Q;
32      unsigned char* charImage;
33      char header[100], *ptr;
34
35      static_assert(sizeof(Image::pixelT) == 1,
36                    "Image reading only supported for single-byte pixel types.");
37
38      // read header
39      in.getline(header, 100, '\n');
40      if ((header[0] != 'P') || (header[1] != '5')) { throw std::runtime_error("Image
     ↪  is not PGM!"); }
41
42      in.getline(header, 100, '\n');
43      while (header[0] == '#') in.getline(header, 100, '\n');
44
45      N = strtol(header, &ptr, 0);
46      M = atoi(ptr);
47
48      in.getline(header, 100, '\n');
49      Q = strtol(header, &ptr, 0);
50
51      if (Q > 255) throw std::runtime_error("Image cannot be read correctly (Q >
     ↪  255)!");
52
53      charImage = new unsigned char[M * N];
54
55      in.read(reinterpret_cast<char*>(charImage), (M * N) * sizeof(unsigned char));
56
57      if (in.fail()) throw std::runtime_error("Image has wrong size!");
58
59      return Image(M, N, Q, charImage);
60    }
61
62    // Slightly modified version of writeImage() function provided by Dr. Bebis
63    std::ostream& operator<<(std::ostream& out, const Image& im) {
64      static_assert(sizeof(Image::pixelT) == 1,
65                    "Image writing only supported for single-byte pixel types.");
66
67      out << "P5" << std::endl;
68      out << im.N << " " << im.M << std::endl;
69      out << im.Q << std::endl;
70
71      out.write(reinterpret_cast<char*>(im.pixelValue), (im.M * im.N) *
     ↪  sizeof(unsigned char));
72
73      if (out.fail()) throw std::runtime_error("Something failed with writing
     ↪  image.");
74    }
```

```
75
76   Image& Image::operator=(const Image& rhs) {
77     if (pixelValue != nullptr) delete[] pixelValue;
78
79     M = rhs.M;
80     N = rhs.N;
81     Q = rhs.Q;
82
83     pixelValue = new pixelT[M * N];
84
85     for (unsigned i = 0; i < M * N; i++) pixelValue[i] = rhs.pixelValue[i];
86
87     return *this;
88   }
89
90   Image& Image::operator=(Image&& rhs) {
91     if (pixelValue != nullptr) delete[] pixelValue;
92
93     M          = rhs.M;
94     N          = rhs.N;
95     Q          = rhs.Q;
96     pixelValue = rhs.pixelValue;
97
98     rhs.M = rhs.N = rhs.Q = 0;
99     rhs.pixelValue         = nullptr;
100
101    return *this;
102  }
103
104  Image::pixelT* Image::operator[](unsigned i) {
105    return pixelValue + i * N;
106  }
107
108  const Image::pixelT* Image::operator[](unsigned i) const {
109    return pixelValue + i * N;
110  }
111
112  // Slightly modified version of readImageHeader() function provided by Dr. Bebis
113  Image::Header Image::Header::read(std::istream& in) {
114    unsigned char* charImage;
115    char header[100], *ptr;
116    Header re;
117
118    // read header
119    in.getline(header, 100, '\n');
120    if ((header[0] == 'P') && (header[1] == '5')) {
121      re.type = GRAY;
122    } else if ((header[0] == 'P') && (header[1] == '6')) {
123      re.type = COLOR;
124    } else
125      throw std::runtime_error("Image is not PGM or PPM!");
126
127    in.getline(header, 100, '\n');
```

```cpp
128      while (header[0] == '#') in.getline(header, 100, '\n');
129
130      re.N = strtol(header, &ptr, 0);
131      re.M = atoi(ptr);
132
133      in.getline(header, 100, '\n');
134
135      re.Q = strtol(header, &ptr, 0);
136
137      return re;
138    }
139
140    std::ostream& operator<<(std::ostream& out, const Image::Header& head) {
141      switch (head.type) {
142        case Image::Header::Type::COLOR:
143          out << "PPM Color ";
144          break;
145        case Image::Header::Type::GRAY:
146          out << "PGM Grayscale ";
147      }
148      out << "Image size " << head.M << " x " << head.N << " and max value of " <<
         head.Q << ".";
149    }
```

Listing 3: Implementation file for the `Histogram` supporting library.

```cpp
1    // Common/histogram_tools.cpp
2    #include "histogram_tools.h"
3
4    #include <algorithm>
5    #include <iostream>
6
7    void Histogram::print(unsigned* histogram, unsigned bins, unsigned width, unsigned
       height) {
8      // An adjusted histogram, which has been binned
9      unsigned* binnedHistogram = new unsigned[width];
10     // Maximum number of original bins represented by each new bin
11     // Each bin is this size, except maybe the last bin (which may be smaller)
12     unsigned binSize = 1 + (bins - 1) / width;
13     // The maximum number of observations in all bins
14     unsigned maxBin = 0;
15
16     // Calculate new binnedHistogram and maxBin
17    #pragma omp parallel for reduction(max : maxBin)
18     for (unsigned i = 0; i < width; i++) {
19       binnedHistogram[i] = 0;
20       for (unsigned j = binSize * i; j < binSize * (i + 1) && j < bins; j++) {
21         binnedHistogram[i] += histogram[j];
22       }
23       maxBin = std::max(binnedHistogram[i], maxBin);
24     }
25
```

```cpp
26      // The maximum number of observations each tick can represent
27      // May represent as few as 1, if present on the top of a histogram bar
28      unsigned tickSize = 1 + (maxBin - 1) / height;
29
30      for (unsigned i = 1; i <= height; i++) {
31        unsigned threshold = (height - i) * tickSize;
32        for (unsigned j = 0; j < width; j++) {
33          if (binnedHistogram[j] > threshold)
34            std::cout << '*';
35          else
36            std::cout << ' ';
37        }
38        std::cout << '\n';
39      }
40
41      delete[] binnedHistogram;
42    }
```

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

```cpp
1   #include <iostream>
2   #include <fstream>
3   #include <sstream>
4
5   #include "../Common/image.h"
6
7   /*
8     Subsamples and image based on the sampling factor
9     @Param: image - the input image that will be sampled
10    @Param: subsample_factor - the factor by which to sample
11    @Return: void
12  */
13  void subsample_image(Image& image, int subsample_factor){
14
15    // iterate through image to get the sample
16    for(int i=0; i<image.cols; i += subsample_factor){
17        for(int j=0; j<image.rows; j += subsample_factor) {
18
19            // save the sampled pixel
20            int pixelSample = image[i][j];
21
22            // Modify neighbor pixels to match the sampled pixel
23            for (int k = 0; k < subsample_factor; k++)
24            {
25              for (int l = 0; l < subsample_factor; l++)
26              {
27                image[i + k][j + l] = pixelSample;
28              }
29            }
30        }
31      }
32  }
```

```
33
34    int main(int argc, char** argv) {
35
36        int M, N, Q;
37        bool type;
38        int val;
39        int subsample_factor;
40        std::istringstream ss(argv[3]);
41
42        // Get sampling factor, error checking
43        if(ss >> subsample_factor) {
44          if(256 % subsample_factor != 0 || subsample_factor > 256){
45             std::cout << "Error: Subsample factor should be power of 2 less than 256"
                  ↪   << std::endl;
46             return 1;
47
48          }
49        }
50
51        //read image
52        std::ifstream inFile(argv[1]);
53
54         Image image = Image::read(inFile);
55
56      std::cout << "Question 1: Sampling." << std::endl;
57
58        // sample the image
59        subsample_image(image, subsample_factor);
60
61        // Save output image
62        std::ofstream outFile;
63        outFile.open(argv[2]);
64        outFile << image;
65        outFile.close();
66
67        return 0;
68    }
```

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

```
1     #include <iostream>
2     #include <fstream>
3     #include <sstream>
4     #include <vector>
5
6     #include "../Common/image.h"
7
8     /*
9       Quantizes an image based on the quantization level
10      @Param: image - the input image that will be quantized
11      @Param: quantization_level - the number of gray level values to use
12      @Return: void
```

```
13   */
14   void quantize_image(Image& image, int quantization_level){
15
16     int offset = 256 / quantization_level;
17
18     // New set of possible pixel values
19     std::vector<int> newPixelValues;
20
21     // calculate new values for pixels
22     for (int i = 0; i < quantization_level; i++)
23         newPixelValues.push_back(i * offset);
24
25
26     // For each pixel
27     for(int i=0; i<image.cols; i++) {
28         for(int j=0; j<image.rows; j++) {
29
30             // current pixel
31             int pixelValue = image[i][j];
32
33             // index for new pixel value
34             int index = pixelValue / offset;
35
36             // update image pixel
37             image[i][j] = newPixelValues[index];
38         }
39     }
40   }
41
42   int main(int argc, char** argv) {
43     int M, N, Q;
44     bool type;
45     int val;
46     int quantization_level;
47     std::istringstream ss(argv[3]);
48
49     // GEt quantization level
50     if(ss >> quantization_level) {
51       if(quantization_level > 256){
52         std::cout << "Error: Quantization level should be less than 256" <<
                ↪   std::endl;
53         return 1;
54
55       }
56     }
57
58     // Read original image
59     std::ifstream inFile(argv[1]);
60
61     Image image = Image::read(inFile);
62
63     std::cout << "Question 2: Quantization." << std::endl;
64
```

```cpp
65      // Quantize the image
66      quantize_image(image, quantization_level);
67
68      // Save output image
69      std::ofstream outFile;
70      outFile.open(argv[2]);
71      outFile << image;
72      outFile.close();
73
74      return 0;
75    }
```

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

```cpp
1    // Q3-Equalization/main.cpp
2    #include <cstring>
3    #include <fstream>
4    #include <iostream>
5    #include <map>
6    #include <mutex>
7
8    #include "../Common/histogram_tools.h"
9    #include "../Common/image.h"
10
11   // Struct for inputting arguments from command line
12   struct Arguments {
13     char *inputImagePath, *outImagePath;
14     Image inputImage;
15     std::ofstream outFile;
16     unsigned histogramWidth = 64, histogramHeight = 10;
17     bool plot = false;
18     std::ofstream plotFile;
19   };
20
21   void equalize(Arguments& arg);
22   bool verifyArguments(int argc, char** argv, Arguments& arg, int& err);
23   void printHelp();
24
25   int main(int argc, char** argv) {
26     int err;
27     Arguments arg;
28
29     if (!verifyArguments(argc, argv, arg, err)) { return err; }
30
31     equalize(arg);
32
33     return 0;
34   }
35
36   void equalize(Arguments& arg) {
37     unsigned* histogram    = new unsigned[arg.inputImage.maxVal + 1];
38     unsigned* newHistogram = new unsigned[arg.inputImage.maxVal + 1];
```

```
39      unsigned* cdf            = new unsigned[arg.inputImage.maxVal + 1];
40      std::mutex* locks        = new std::mutex[arg.inputImage.maxVal + 1];
41      // Initialise histogram bins to be empty
42    #pragma omp parallel for
43      for (unsigned i = 0; i <= arg.inputImage.maxVal; i++) {
44        histogram[i] = newHistogram[i] = 0;
45      }
46
47      // Create histogram
48    #pragma omp parallel for
49      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
50        unsigned bin = arg.inputImage.pixels[i];
51        locks[bin].lock();
52        histogram[bin]++;
53        locks[bin].unlock();
54      }
55
56      // Calculate CDF
57      cdf[0] = histogram[0];
58      for (unsigned i = 1; i <= arg.inputImage.maxVal; i++) {
59        cdf[i] = cdf[i - 1] + histogram[i];
60      }
61
62      // Tranform image with the CDF
63    #pragma omp parallel for
64      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
65        Image::pixelT& pixelVal = arg.inputImage.pixels[i];
66        pixelVal                = cdf[pixelVal] * arg.inputImage.maxVal /
67                   (arg.inputImage.rows * arg.inputImage.cols);
68      }
69
70      // Write new transformed image out
71      arg.outFile << arg.inputImage;
72      arg.outFile.close();
73
74      // Calculate histogram of new image
75    #pragma omp parallel for
76      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
77        unsigned bin = arg.inputImage.pixels[i];
78        locks[bin].lock();
79        newHistogram[bin]++;
80        locks[bin].unlock();
81      }
82
83      // Print histograms
84      std::cout << "\nHistogram of input image \"" << arg.inputImagePath << "\":\n";
85      Histogram::print(histogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
86                       arg.histogramHeight);
87
88      std::cout << "\nHistogram of output image \"" << arg.outImagePath << "\":\n";
89      Histogram::print(newHistogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
90                       arg.histogramHeight);
91
```

```cpp
92      // Print histogram data for plot file
93      if (arg.plot) {
94        arg.plotFile << "Image Input Equalized\n";
95        for (unsigned i = 0; i <= arg.inputImage.maxVal; i++) {
96          arg.plotFile << i << "    " << histogram[i] << "    " << newHistogram[i]
97                       << '\n';
98        }
99        arg.plotFile.close();
100     }
101
102     delete[] histogram;
103     delete[] newHistogram;
104     delete[] cdf;
105     delete[] locks;
106   }
107
108   bool verifyArguments(int argc, char** argv, Arguments& arg, int& err) {
109     // If there are not the minimum number of arguments, print help and leave
110     if (argc < 2 ||
111         (argc < 3 && strcmp(argv[1], "-h") && strcmp(argv[1], "--help"))) {
112       std::cout << "Missing operand.\n";
113       err = 1;
114       printHelp();
115       return false;
116     }
117
118     // If the user asks for the help menu, print help and leave
119     if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
120       printHelp();
121       return false;
122     }
123
124     // Find optional argument switches
125     for (unsigned i = 3; i < argc; i++) {
126       if (!strcmp(argv[i], "-width")) {
127         if (i + 1 >= argc) {
128           std::cout << "Missing width";
129           err = 1;
130           printHelp();
131           return false;
132         }
133
134         arg.histogramWidth = strtoul(argv[i + 1], nullptr, 10);
135         if (arg.histogramWidth == 0) {
136           std::cout << "Width \"" << argv[i + 1]
137                     << "\" could not be recognised as a positive integer.";
138           err = 2;
139           return false;
140         }
141
142         i++;
143       } else if (!strcmp(argv[i], "-height")) {
144         if (i + 1 >= argc) {
```

```
145            std::cout << "Missing height";
146            err = 1;
147            break;
148          }
149
150          arg.histogramHeight = strtoul(argv[i + 1], nullptr, 10);
151          if (arg.histogramHeight == 0) {
152            std::cout << "Height \"" << argv[i + 1]
153                      << "\" could not be recognised as a positive integer.";
154            err = 2;
155            return false;
156          }
157
158          i++;
159        } else if (!strcmp(argv[i], "-p")) {
160          if (i + 1 >= argc) {
161            std::cout << "Missing plot output file";
162            err = 1;
163            break;
164          }
165
166          arg.plot = true;
167          arg.plotFile.open(argv[i + 1]);
168
169          if (!arg.plotFile) {
170            std::cout << "Plot file \"" << argv[i + 1]
171                      << "\" could not be opened";
172            err = 2;
173            return false;
174          }
175
176          i++;
177        }
178      }
179
180      // Required arguments
181      arg.inputImagePath = argv[1];
182      std::ifstream inFile(argv[1]);
183      try {
184        arg.inputImage = Image::read(inFile);
185      } catch (std::exception& e) {
186        std::cout << "Image \"" << argv[1] << "\"failed to be read: \"" << e.what()
187                  << "\"\n";
188        err = 2;
189        return false;
190      }
191
192      arg.outImagePath = argv[2];
193      arg.outFile.open(argv[2]);
194      if (!arg.outFile) {
195        std::cout << "Could not open \"" << argv[2] << "\"\n";
196        err = 2;
197        return false;
```

```cpp
198      }
199
200      return true;
201    }
202
203    void printHelp() {
204      std::cout
205          << "Usage: equalize <image> <output> [options]   (1)\n"
206          << "   or: equalize -h                           (2)\n\n"
207          << "(1) Take an image file as input, equalize its histogram,\n"
208          << "    and write new image to output file. Displays the original\n"
209          << "    histogram and the new equalized histogram.\n"
210          << "(2) Print this help menu\n\n"
211          << "Options:\n"
212          << "  -width <width>    Number of visual histogram bins\n"
213          << "  -height <height>  Height of visual histogram (in lines)\n"
214          << "  -p <file>         Send histogram plotting data to a file for
              ↪  gnuplot\n";
215    }
```

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

```cpp
1    // Q4-Specification/main.cpp
2    #include <cstring>
3    #include <fstream>
4    #include <iostream>
5    #include <mutex>
6    #include <regex>
7    #include <vector>
8
9    #include "../Common/histogram_tools.h"
10   #include "../Common/image.h"
11
12   // Struct for inputting arguments from command line
13   struct Arguments {
14     char *inputImagePath, *outImagePath, *histogramPath;
15     Image inputImage;
16     std::ifstream histogramFile;
17     std::ofstream outFile;
18     unsigned histogramWidth = 64, histogramHeight = 10;
19     bool plot = false;
20     std::ofstream plotFile;
21   };
22
23   int specify(Arguments& arg);
24   void printHistogram(unsigned* histogram, const Arguments& arg);
25   bool verifyArguments(int argc, char** argv, Arguments& arg, int& err);
26   void printHelp();
27
28   int main(int argc, char** argv) {
29     int err;
30     Arguments arg;
```

```cpp
31
32     if (!verifyArguments(argc, argv, arg, err)) { return err; }
33
34     return specify(arg);
35   }
36
37   int specify(Arguments& arg) {
38     unsigned* histogram    = new unsigned[arg.inputImage.maxVal + 1];
39     unsigned* newHistogram = new unsigned[arg.inputImage.maxVal + 1];
40     unsigned* cdf          = new unsigned[arg.inputImage.maxVal + 1];
41     std::mutex* locks      = new std::mutex[arg.inputImage.maxVal + 1];
42     std::vector<unsigned> targetHistogram;
43     unsigned targetPixels = 0;  // Number of pixels in the target histogram
44
45     // Initialise histogram bins to be empty
46   #pragma omp parallel for
47     for (unsigned i = 0; i <= arg.inputImage.maxVal; i++) {
48       histogram[i] = newHistogram[i] = 0;
49     }
50
51     // Create input image histogram
52   #pragma omp parallel for
53     for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
54       unsigned bin = arg.inputImage.pixels[i];
55       locks[bin].lock();
56       histogram[bin]++;
57       locks[bin].unlock();
58     }
59
60     // Start with enough space to hold our input image. If we need more, we can get
61     // more, but we're probably working with similarly-valued images.
62     targetHistogram.reserve(arg.inputImage.maxVal + 1);
63
64     // Read in target histogram
65     std::string line;
66     std::regex rHistogram("^([[:digit:]]+)[[:space:]]+([[:digit:]]+).*");
67     std::smatch matches;
68     while (arg.histogramFile) {
69       std::getline(arg.histogramFile, line);
70       if (!std::regex_match(line, matches, rHistogram)) continue;
71
72       if (stoul(matches[1].str()) != targetHistogram.size()) {
73         std::cout << "Error in reading histogram file \"" << arg.histogramPath
74                   << "\":\n"
75                   << "Bucket \"" << stoul(matches[1].str())
76                   << "\" was expected to be \"" << targetHistogram.size()
77                   << "\".";
78       }
79
80       targetHistogram.push_back(stoul(matches[2].str()));
81       targetPixels += targetHistogram.back();
82     }
83     arg.histogramFile.close();
```

```
84
85        // Calculate CDFs
86        std::vector<unsigned> targetCDF(targetHistogram.size());
87        cdf[0]       = histogram[0];
88        targetCDF[0] = targetHistogram[0];
89
90        for (unsigned i = 1; i <= arg.inputImage.maxVal; i++) {
91          cdf[i] = cdf[i - 1] + histogram[i];
92        }
93        for (unsigned i = 1; i < targetHistogram.size(); i++) {
94          targetCDF[i] = targetCDF[i - 1] + targetHistogram[i];
95        }
96
97    // Tranform input image with its CDF and inverse CDF of target histogram
98    #pragma region CDF transformation
99        // Separate cases for if the images have different dimensions/maxVal, to make
100       // calculation easier
101       if (arg.inputImage.maxVal == targetHistogram.size() - 1) {
102         if (arg.inputImage.rows * arg.inputImage.cols == targetPixels) {
103   #pragma omp parallel for
104           for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
105             Image::pixelT& pixelVal = arg.inputImage.pixels[i];
106
107             unsigned inversePixel = cdf[pixelVal];
108
109             pixelVal = targetCDF.rend() -
110                        std::lower_bound(targetCDF.rbegin(), targetCDF.rend(),
111                                         inversePixel, std::greater<unsigned>());
112           }
113         } else {
114   #pragma omp parallel for
115           for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
116             Image::pixelT& pixelVal = arg.inputImage.pixels[i];
117
118             unsigned inversePixel = cdf[pixelVal] * targetPixels /
119                                     (arg.inputImage.rows * arg.inputImage.cols);
120             pixelVal = targetCDF.rend() -
121                        std::lower_bound(targetCDF.rbegin(), targetCDF.rend(),
122                                         inversePixel, std::greater<unsigned>());
123           }
124         }
125       } else if (arg.inputImage.rows * arg.inputImage.cols == targetPixels) {
126   #pragma omp parallel for
127         for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
128           Image::pixelT& pixelVal = arg.inputImage.pixels[i];
129
130           unsigned inversePixel =
131               cdf[pixelVal] * arg.inputImage.maxVal / (targetHistogram.size() - 1);
132           pixelVal = targetCDF.rend() -
133                      std::lower_bound(targetCDF.rbegin(), targetCDF.rend(),
134                                       inversePixel, std::greater<unsigned>());
135         }
136       } else {
```

```cpp
137        // In this case, we need to do math with ull because of the multiplications
138        // overflowing The result after division should fit within an unsigned, though
139  #pragma omp parallel for
140      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
141          Image::pixelT& pixelVal = arg.inputImage.pixels[i];
142
143          unsigned inversePixel = ((unsigned long long) cdf[pixelVal]) *
144                                  arg.inputImage.maxVal * targetPixels /
145                                  (arg.inputImage.rows * arg.inputImage.cols *
146                                   (targetHistogram.size() - 1));
147          pixelVal = targetCDF.rend() -
148                     std::lower_bound(targetCDF.rbegin(), targetCDF.rend(),
149                                      inversePixel, std::greater<unsigned>());
150      }
151    }
152  #pragma endregion CDF transformation
153
154      // Write new transformed image out
155      arg.outFile << arg.inputImage;
156      arg.outFile.close();
157
158      // Calculate histogram of new image
159  #pragma omp parallel for
160      for (unsigned i = 0; i < arg.inputImage.rows * arg.inputImage.cols; i++) {
161          unsigned bin = arg.inputImage.pixels[i];
162          locks[bin].lock();
163          newHistogram[bin]++;
164          locks[bin].unlock();
165      }
166
167      // Print histograms
168      std::cout << "\nHistogram of input image \"" << arg.inputImagePath << "\":\n";
169      Histogram::print(histogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
170                       arg.histogramHeight);
171
172      std::cout << "\nInput histogram \"" << arg.histogramPath << "\":\n";
173      Histogram::print(&targetHistogram.front(), targetHistogram.size(),
174                       arg.histogramWidth, arg.histogramHeight);
175
176      std::cout << "\nHistogram of output image \"" << arg.outImagePath << "\":\n";
177      Histogram::print(newHistogram, arg.inputImage.maxVal + 1, arg.histogramWidth,
178                       arg.histogramHeight);
179
180      // Print histogram data for plot file
181      if (arg.plot) {
182          arg.plotFile << "Source Input-Image Input-Histogram Specified\n";
183          unsigned i;
184          for (i = 0; i <= arg.inputImage.maxVal && i < targetHistogram.size(); i++) {
185              arg.plotFile << i << "    " << histogram[i] << "     "
186                           << targetHistogram[i] << "    " << newHistogram[i] << '\n';
187          }
188          arg.plotFile.close();
189      }
```

```
190
191      delete[] histogram;
192      delete[] newHistogram;
193      delete[] cdf;
194      delete[] locks;
195
196      return 0;
197    }
198
199    bool verifyArguments(int argc, char** argv, Arguments& arg, int& err) {
200      if (argc < 2 ||
201          (argc < 4 && strcmp(argv[1], "-h") && strcmp(argv[1], "--help"))) {
202        std::cout << "Missing operand.\n";
203        err = 1;
204        printHelp();
205        return false;
206      }
207
208      if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "--help")) {
209        printHelp();
210        return false;
211      }
212
213      // Find optional argument switches
214      for (unsigned i = 4; i < argc; i++) {
215        if (!strcmp(argv[i], "-width")) {
216          if (i + 1 >= argc) {
217            std::cout << "Missing width";
218            err = 1;
219            printHelp();
220            return false;
221          }
222
223          arg.histogramWidth = strtoul(argv[i + 1], nullptr, 10);
224          if (arg.histogramWidth == 0) {
225            std::cout << "Width \"" << argv[i + 1]
226                      << "\" could not be recognised as a positive integer.";
227            err = 2;
228            return false;
229          }
230
231          i++;
232        } else if (!strcmp(argv[i], "-height")) {
233          if (i + 1 >= argc) {
234            std::cout << "Missing height";
235            err = 1;
236            break;
237          }
238
239          arg.histogramHeight = strtoul(argv[i + 1], nullptr, 10);
240          if (arg.histogramHeight == 0) {
241            std::cout << "Height \"" << argv[i + 1]
242                      << "\" could not be recognised as a positive integer.";
```

```
243          err = 2;
244          return false;
245        }
246
247        i++;
248      } else if (!strcmp(argv[i], "-p")) {
249        if (i + 1 >= argc) {
250          std::cout << "Missing plot output file";
251          err = 1;
252          break;
253        }
254
255        arg.plot = true;
256        arg.plotFile.open(argv[i + 1]);
257
258        if (!arg.plotFile) {
259          std::cout << "Plot file \"" << argv[i + 1]
260                    << "\" could not be opened";
261          err = 2;
262          return false;
263        }
264
265        i++;
266      }
267    }
268
269    // Required arguments
270    arg.inputImagePath = argv[1];
271    std::ifstream inFile(argv[1]);
272    try {
273      arg.inputImage = Image::read(inFile);
274    } catch (std::exception& e) {
275      std::cout << "Image \"" << argv[1] << "\"failed to be read: \"" << e.what()
276                << "\"\n";
277      err = 2;
278      return false;
279    }
280
281    arg.histogramPath = argv[2];
282    arg.histogramFile.open(argv[2]);
283    if (!arg.histogramFile) {
284      std::cout << "Could not open \"" << argv[2] << "\"\n";
285      err = 2;
286      return false;
287    }
288
289    arg.outImagePath = argv[3];
290    arg.outFile.open(argv[3]);
291    if (!arg.outFile) {
292      std::cout << "Could not open \"" << argv[3] << "\"\n";
293      err = 2;
294      return false;
295    }
```

```cpp
296
297      return true;
298    }
299
300    void printHelp() {
301      std::cout
302          << "Usage: specify <image> <histogram> <output> [options]    (1)\n"
303          << "   or: specify -h                                         (2)\n\n"
304          << "(1) Take an image file as input, change its histogram to the\n"
305          << "    specified histogram, and write new image to output file.\n"
306          << "    Displays the original histogram and the new equalized histogram.\n"
307          << "    Histogram files can be obtained by running 'equalize' with\n"
308          << "    the -p flag set (or 'specify' with the -p falg set).\n"
309          << "(2) Print this help menu\n\n"
310          << "Options:\n"
311          << "  -width <width>    Number of visual histogram bins\n"
312          << "  -height <height>  Height of visual histogram (in lines)\n"
313          << "  -p <file>         Send histogram plotting data to a file for
               ↪  gnuplot\n";
314    }
```

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

```gnuplot
1    # Q3-Equalization/plot-histograms.plt
2    # A gnuplot plotting file to plot the two histograms of data from equalize with
     ↪   the -p switch
3    if (!exists("outfile")) outfile='plot.eps'
4
5    if (!exists("imageName")) {
6      set title "Comparison of histograms of input and output (equalized) images"
7    } else {
8      set title "Comparison of histograms of " . imageName . " and equalized image"
       ↪   noenhanced
9    }
10
11   set terminal postscript eps enhanced color size 6,3
12   set output outfile
13   set style data histogram
14   set style histogram cluster gap 1
15   set style fill solid
16   set boxwidth 0.9
17
18   unset xtics
19
20   # Colors chosen using ColorBrewer 2.0 qualitative scheme "Dark2"
21   # https://colorbrewer2.org/#type=qualitative&scheme=Dark2&n=3
22   plot infile using 2:xtic(1) ti col linecolor rgb "#1b9e77",\
23     '' 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"
```