University of Nevada, Reno

CS 474 — Image Processing

Assignment #2

Joshua Gleason

Instructor:
Dr. George Bebis

September 23, 2010

Contents

1 Introduction **2**

2 Histogram Equalization **2**

3 Histogram Specification **3**

4 Median Filtering **3**

5 Source Code **4**
5.1 generic.h 4
5.2 equalize.cc 8
5.3 spec.cc 9
5.4 median.cc 10

6 Images **12**

List of Figures

1 Histogram Equalization lenna.pgm 12
2 Histogram Equalization sf.pgm 13
3 Histogram Equalization lax.pgm 13
4 Histogram Specification lax.pgm 14
5 Histogram Specification lenna.pgm 15
6 Histogram Specification sf.pgm 16
7 Median Filter lenna.pgm 17
8 Median Filter peppers.pgm 18

1 Introduction

Poor lighting and other global conditions can reduce the quality of images. This project examines a few different methods of image enhancement.

The first method is global histogram equalization. Histogram equalization is especially useful for increasing contrast in images. This works well when an image has poor lighting, where contrast values sit close to each other.

Another method which was implemented is a global histogram specification, also commonly known as histogram matching. Histogram specification differs from equalization in that rather than matching with a uniform histogram, the histogram is matched with another histogram which is specified by the user. This can be useful when an image with similar contrast to another image is desired.

Finally the method of local median filtering is used to reduce salt and pepper noise from images. For median filtering the values in the vicinity of a pixel are examined, and the median value determined. This value then replaces the pixel at the center of the area being observed. For comparison purposes, mean filtering was also implemented. This method examines the pixels surrounding a pixel and determines the mean gray value. This mean then replaces the value of the pixel at the center of the neighborhood.

2 Histogram Equalization

Equalization was implemented by first building a function which calculates the histogram of an image. The histogram is built by simply adding one to the bin corresponding to that pixel value for every pixel in the image. Every bin is then divided by the total number of pixels in the image to normalize it. The histogram $p(r_k)$ can be defined as follows.

$$p(r_k) = \frac{n_k}{N}$$

k is the pixel intensity, r_k is the bin in our histogram that represents k, n_k is the number of pixels with intensity k in the image, and N is the number of total pixels, or the size of the image.

Once the histogram is calculated, calculating the transformation function s_k was done by creating a running total of the values in the histogram, and then de-normalizing by multiplying by the maximum pixel value. This is defined as follows.

$$s_k = T(r_k) = 255 \sum_{j=0}^{k} p(r_j) = 255 \sum_{j=0}^{k} \frac{n_j}{N}$$

The next step was to simply map all the pixels from the original image into $T(r_k)$. This was done quite easily since s_k was already de-normalized to have a range from 0 to 255.

This method worked remarkably well for being such a simple process. The results in Fig. [1, 2, 3] show the results of histogram equalization. In Fig. [1] the results are subtle, but the image looks like it has better contrast. In Fig. [2] there is a great improvement in the quality of the image. However in Fig. [3] it seems as if the contrast has been expanded just a little too much. The image looks much too bright in some areas, and too dark in others.

3 Histogram Specification

Matching the histogram for the image with another histogram is similar to histogram equalization except that instead of mapping the histogram to a constant distribution, it is mapped to model another histogram. This histogram could be any histogram, including one taken from another image.

For implementation, the histogram of the image is calculated, and then the equalization function is calculated for both the input image, and the specification histogram that was passed as the other parameter to the function. These steps use the same functions created in Sect. 2.

We will call the equalization function for the input image $T(r_k)$ and the equalization function of the specification histogram $S(r_k)$. To create the new transformation, for each grey level x, we find $T(x)$ and then find the value y such that $S(y)$ matches $T(x)$ as close as possible. Then for the value of our new transform R, we define it such that $R(x) = y$. Once this transformation function is complete, we use the same function as we did in Sect. 2 to calculate our output image.

This method worked well for correcting contrast issues with images, assuming we had another image that had the desired contrast. In Fig. [4b] we can see that the contrast seems much more appropriate than it did for Equalization (Fig. [3]). It also has some other interesting results, such as in Fig. [5c] where the image has been specified with the histogram from the input image in Fig. [4]. If the two images are compared we can see that the contrast is nearly identical. This could be a very useful property when using images where illumination effects are required. If histogram specification were used, the contrast between two images taken for the same purpose could be made closer to each other, which may strengthen a particular vision algorithm or something similar.

4 Median Filtering

Salt and pepper noise is very difficult to deal with using linear filters (such as mean). However this noise can be greatly reduced when using a median filter. This can be understood intuitively. If there are approximately an equal number of white (salt) and black (pepper) values in a particular vicinity, then the median value will no be either of these values. In fact it will be one of the in-between values that was in the original image. When using a mean, although some values are kept from the original image, the weights of these original pixels are overpowered by the white and black values. For this reason median filtering is the filter of choice when dealing with salt and pepper noise.

The implementation of the algorithm was very straight forward. Simply put the pixel, as well as its neighbors within a predefined distance, into an array. Then sort the array and retrieve the value in the center.

This method was used to attempt to reduce salt and pepper noise in images with 50% noise and 75% noise. In these experiments different sized filters were used, namely 3x3, 5x5, and 7x7. While the 7x7 filter reduced the most noise, it also lost the most detail in the image; although for mean the larger filters were even more distorted than the smaller ones. In Fig. [8] this can be observed. When looking at the original image with the 75% noise, it is difficult to believe that any useful information can be extracted from the image. For this reason it is quite impressive that the median filter is able to perform as well as it did.

5 Source Code

5.1 generic.h

```
#include <cv.h>
#include <highgui.h>
#include <fstream>
#include <ctime>
#include <cstdlib>
#include <algorithm>

namespace jdg
{
void calcHisto( IplImage* src, float* histo, int bins=256);
void saveHisto( char* fileName, float* histo, int bins=256 );
void calcEqualizeTransform( float* histo, float* func, int bins=256 );

enum Noise_Type{ SALT_AND_PEPPER };

// quantize from range [<min>, <max>] into [0,<vals>] then back
void quantize( IplImage* src, IplImage* dest, int min, int max, int vals )
{
  int width = src->width,
      height = src->height,
      old_range = max-min+1,
      val;
  int step = old_range/vals;

  for ( int i = 0; i < height; i++ )
    for ( int j = 0; j < width; j++ )
    {
      val = static_cast<int>( cvGetReal2D( src, i, j ) ) + min;
      val /= step;
      val = val*(old_range-1)/(vals-1);
      cvSetReal2D( dest, i, j, static_cast<double>(val) );
    }
}

void applyTransform( IplImage* img, float* func )
{
  int width = img->width,
      height = img->height;

  for ( int i = 0; i < height; i++ )
    for ( int j = 0; j < width; j++ )
      cvSetReal2D(
        img, i, j,
        round(func[static_cast<int>(cvGetReal2D(img, i, j))])
      );
}

// as of now only computes histogram for 8 bit grayscale images
// assumes histo is initialized to the size of bins
void calcHisto( IplImage* src, float* histo, int bins )
{
  int height = src->height,
      width = src->width,
      bin;
```

```
  float size = height * width;

  // zero the histogram
  for ( int i = 0; i < bins; i++ )
    histo[i] = 0.0;

  for ( int i = 0; i < height; i++ )
    for ( int j = 0; j < width; j++ )
    {
      bin = static_cast<int>(cvGetReal2D( src, i, j ));
      histo[bin]++;          // add to the histogram bin
    }

  // make histogram sum to 1
  for ( int i = 0; i < bins; i++ )
    histo[i] /= size;
}

void calcEqualizeTransform( float* histo, float* func, int bins )
{
  func[0] = histo[0] * (bins-1);
  for ( int i = 1; i < 256; i++ )
    func[i] = func[i-1] + histo[i] * (bins - 1);
}

void equalizeImage( IplImage* img )
{
  float histo[256];
  float func[256];

  calcHisto( img, histo );
  calcEqualizeTransform( histo, func );
  applyTransform( img, func );
}

void specifyImageHisto( IplImage* img, float* histo_spec )
{
  float histo_orig[256];

  // calculate the original histogram
  calcHisto( img, histo_orig );

  float func_orig[256];
  float func_spec[256];

  calcEqualizeTransform( histo_orig, func_orig );
  calcEqualizeTransform( histo_spec, func_spec );

  float func_new[256];

  for ( int i = 0; i < 256; i++ )
  {
    // starting value
    int start = 0;
    if ( i != 0 )
      start = func_new[i-1];
    func_new[i] = -1;
    for ( int j = start; j < 256 && func_new[i] < 0; j++ )
```

```
      if ( func_spec[j] > func_orig[i] || func_spec[j] == 255 )
        func_new[i] = std::max(0,j-1);
  }
  applyTransform( img, func_new );
}

void saveHisto( char* fileName, float* histo, int bins )
{
  std::ofstream fout(fileName);
  for ( int i = 0; i < bins; i++ )
    fout << histo[i] << ' ';
  fout.close();
}

void addSaltAndPepper( IplImage* img, int percentage )
{
  int height = img->height,
      width = img->width;

  for ( int i = 0; i < height; i++ )
    for ( int j = 0; j < width; j++ )
      if (rand() % 100 < percentage)
        cvSetReal2D( img, i, j, (rand() % 2) * 255.0 );
}

void medianFilter( IplImage*& img, int filter_size )
{
  int padding = (filter_size - 1) / 2;
  uchar* values = new uchar[filter_size*filter_size];

  IplImage* result = cvCreateImage( cvGetSize(img), img->depth, 1 );

  for ( int i = 0; i < img->height; i++ )
    for ( int j = 0; j < img->width; j++ )
    {
      // itterate accross local region
      static int low_i, up_i, low_j, up_j, index;
      low_i = i-padding;
      up_i = i+padding;
      low_j = j-padding;
      up_j = j+padding;
      index = 0;
      // build list
      for ( int f_i = low_i; f_i <= up_i; f_i++ )
        for ( int f_j = low_j; f_j <= up_j; f_j++ )
        {
          if ( f_i < 0 || f_j < 0 || f_i >= img->height || f_j >= img->
             width )
            values[index] = 0;  // pad with zeros
          else
            values[index] = static_cast<unsigned char>(
              cvGetReal2D( img, f_i, f_j ));
          index++;
        }
      // sort the values
      std::sort(values, values+(filter_size*filter_size));
      // set to median
      cvSetReal2D( result, i, j,
```

```
        static_cast<double>(values[(filter_size*filter_size-1)/2]));
    }

  cvReleaseImage( &img );
  delete [] values;

  img = result;
}

void meanFilter( IplImage*& img, int filter_size )
{
  int padding = (filter_size - 1) / 2;
  IplImage* result = cvCreateImage( cvGetSize(img), img->depth, 1 );

  for ( int i = 0; i < img->height; i++ )
    for ( int j = 0; j < img->width; j++ )
    {
      // itterate accross local region
      static int low_i, up_i, low_j, up_j;
      static double total;
      low_i = i-padding;
      up_i = i+padding;
      low_j = j-padding;
      up_j = j+padding;
      total = 0;
      for ( int f_i = low_i; f_i <= up_i; f_i++ )
        for ( int f_j = low_j; f_j <= up_j; f_j++ )
          if ( !(f_i < 0 || f_j < 0 || f_i >= img->height || f_j >= img->
              width) )
            total += cvGetReal2D( img, f_i, f_j );
      // set to mean
      double mean = total / (filter_size * filter_size);
      cvSetReal2D( result, i, j, mean );
    }

  cvReleaseImage( &img );
  img = result;
}

void medianFilterBetter( IplImage*& img, int filter_size, int thresh )
{
  int padding = (filter_size - 1) / 2;
  uchar* values = new uchar[filter_size*filter_size];

  IplImage* result = cvCreateImage( cvGetSize(img), img->depth, 1 );

  for ( int i = 0; i < img->height; i++ )
    for ( int j = 0; j < img->width; j++ )
    {
      // itterate accross local region
      static int low_i, up_i, low_j, up_j, index;
      low_i = i-padding;
      up_i = i+padding;
      low_j = j-padding;
      up_j = j+padding;
      index = 0;
      // build list
      for ( int f_i = low_i; f_i <= up_i; f_i++ )
```

```
        for ( int f_j = low_j; f_j <= up_j; f_j++ )
        {
          if ( f_i < 0 || f_j < 0 || f_i >= img->height || f_j >= img->
              width )
            values[index] = 0;  // pad with zeros
          else
            values[index] = static_cast<unsigned char>(
              cvGetReal2D( img, f_i, f_j ));
          if ( values[index] >= thresh && values[index] <= 255-thresh )
            index++;
        }
      // sort the values
      int median = 127;
      if ( index != 0 )
      {
        std::sort(values, values+(index-1));
        median = values[index/2];
      }

      // set to median
      cvSetReal2D( result, i, j, static_cast<double>(median));
    }

  cvReleaseImage( &img );
  delete [] values;

  img = result;
}

}
```

5.2 equalize.cc

```
#include <iostream>
#include <fstream>
#include <cv.h>
#include <highgui.h>

#include "generic.h"

//#define USE_OCV

using namespace std;

int main(int argc, char *argv[])
{
  if ( argc < 2 )
  {
    cout << "Usage: ./program <original image> {output filename} "
            "{output histogram filename} {original histogram filename} " <<
                endl
         << "Parameters in {} are optional" << endl;
    return -1;
  }

  IplImage* orig = cvLoadImage( argv[1], 0 );
  IplImage* equalized = cvCloneImage( orig );
```

```
  jdg::equalizeImage( equalized );

  // save results
  float temp_histo[256];
  if ( argc >= 3 )
    cvSaveImage( argv[2], equalized );
  if ( argc >= 4 )
  {
    jdg::calcHisto( equalized, temp_histo );
    jdg::saveHisto( argv[3], temp_histo );
  }
  if ( argc >= 5 )
  {
    jdg::calcHisto( orig, temp_histo );
    jdg::saveHisto( argv[4], temp_histo );
  }

  // display results
  cvNamedWindow("Original");
  cvNamedWindow("Equalized");
  cvShowImage("Original", orig );
  cvShowImage("Equalized", equalized );
  cvWaitKey();
  cvReleaseImage( &orig );
  cvReleaseImage( &equalized );

  return 0;
}
```

5.3 spec.cc

```
#include <iostream>
#include <fstream>
#include <cv.h>
#include <highgui.h>
#include "generic.h"

using namespace std;

int main(int argc, char *argv[])
{
  if ( argc < 3 )
  {
    cout << "Usage: ./program <original image> <new image> {output filename
        } "
             "{output histogram filename} {original histogram filename} "
             "{matching histogram filename}" << endl
         << "Parameters in {} are optional" << endl;
    return -1;
  }

  IplImage* orig = cvLoadImage(argv[1],0);
  IplImage* match = cvLoadImage(argv[2],0);
  IplImage* speced = cvCloneImage( orig );

  // calculate histogram to match original with
  float histo[256];
  jdg::calcHisto( match, histo );
```

```
  // compute histogram specification on image
  jdg::specifyImageHisto( speced, histo );

  // save results
  float temp_histo[256];
  if ( argc >= 4 )  // save specified image
    cvSaveImage( argv[3], speced );
  if ( argc >= 5 )  // save specified histogram
  {
    jdg::calcHisto( speced, temp_histo );
    jdg::saveHisto( argv[4], temp_histo );
  }
  if ( argc >= 6 )  // save original histogram
  {
    jdg::calcHisto( orig, temp_histo );
    jdg::saveHisto( argv[5], temp_histo );
  }
  if ( argc >= 7 )  // save new histogram
    jdg::saveHisto( argv[6], histo );

  // display results
  cvNamedWindow("original");
  cvNamedWindow("match with");
  cvNamedWindow("specified");
  cvShowImage("original", orig);
  cvShowImage("match with", match);
  cvShowImage("specified", speced);
  cvWaitKey();
  cvReleaseImage( &orig );
  cvReleaseImage( &match );
  cvReleaseImage( &speced );
  cvDestroyWindow("original");
  cvDestroyWindow("match with");
  cvDestroyWindow("specified");

  return 0;
}
```

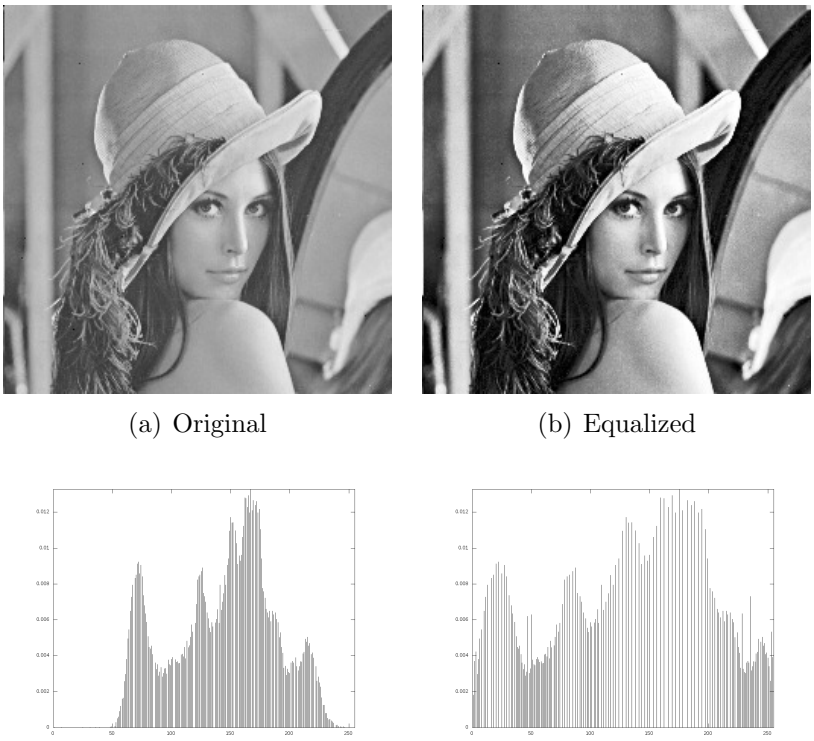
5.4 median.cc

```
#include <iostream>
#include "generic.h"

using namespace std;

int main(int argc, char *argv[])
{
  // check arguments
  if ( argc < 2 )
  {
    cout << "Usage: ./program <image filename> {filter size=3} "
            "{salt/pepper percentage=50} {median filename} "
            "{mean filename} {salt/pepper filename}" << endl
         << "Parameters in {} are optional" << endl;
    return -1;
  }
```

```
// get filter_size and salt/pepper percentage
int filter_size = 3,
    pepper_percentage = 50;

if ( argc >= 3 )
{
  filter_size = atoi(argv[2]);
  filter_size += filter_size % 2 - 1; // ensure value is odd
}
if ( argc >= 4 )
  pepper_percentage = atoi(argv[3]);

// initialize images
IplImage* orig = cvLoadImage( argv[1], 0 );
IplImage* salt_pepper = cvCloneImage( orig );

// add salt and pepper to original
jdg::addSaltAndPepper( salt_pepper, pepper_percentage );

// intialize median and mean images
IplImage* median = cvCloneImage( salt_pepper );
IplImage* median_better = cvCloneImage( salt_pepper );
IplImage* mean = cvCloneImage( salt_pepper );

// run filters on median and mean
jdg::medianFilter( median, filter_size );
jdg::meanFilter( mean, filter_size );

// better median filter which ignores values <= 0+1 and >= 255-1 when
// calculating median
jdg::medianFilterBetter( median_better, filter_size, 1 );

// save results
if ( argc >= 5 )
  cvSaveImage( argv[4], median );
if ( argc >= 6 )
  cvSaveImage( argv[5], mean );
if ( argc >= 7 )
  cvSaveImage( argv[6], salt_pepper );

// display results
cvNamedWindow("Original");
cvNamedWindow("Salt and Pepper");
cvNamedWindow("Mean");
cvNamedWindow("Median(Better)");
cvNamedWindow("Median");

cvShowImage("Original", orig);
cvShowImage("Salt and Pepper", salt_pepper);
cvShowImage("Mean", mean);
cvShowImage("Median(Better)", median_better);
cvShowImage("Median", median);

cvWaitKey();

// clean up memory
cvDestroyWindow("Original");
cvDestroyWindow("Salt and Pepper");
```

```
    cvDestroyWindow("Mean");
    cvDestroyWindow("Median(Better)");
    cvDestroyWindow("Median");

    cvReleaseImage(&orig);
    cvReleaseImage(&salt_pepper);
    cvReleaseImage(&mean);
    cvReleaseImage(&median_better);
    cvReleaseImage(&median);

    return 0;
}
```

6 Images

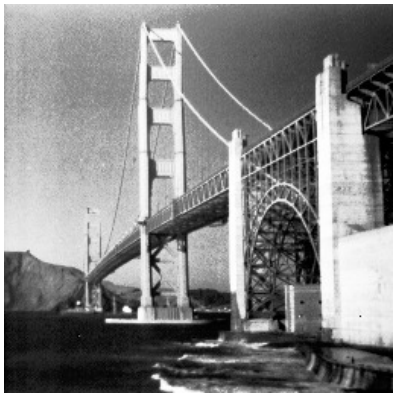


(a) Original (b) Equalized

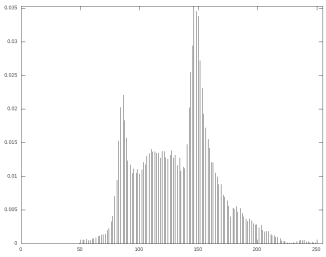
(c) Original Histogram (d) Equalized Histogram

Figure 1: Histogram Equalization lenna.pgm

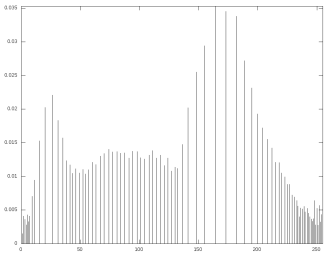
(a) Original



(b) Equalized



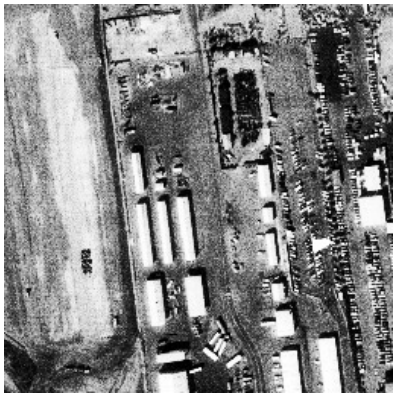
(c) Original Histogram



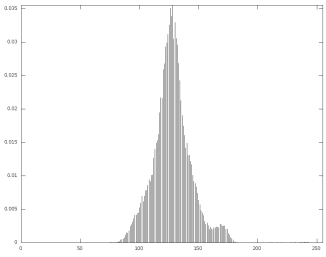
(d) Equalized Histogram

Figure 2: Histogram Equalization sf.pgm

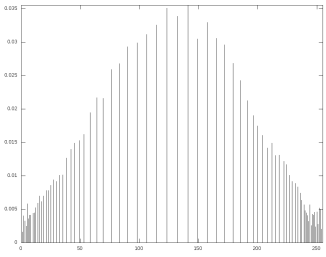
(a) Original



(b) Equalized



(c) Original Histogram

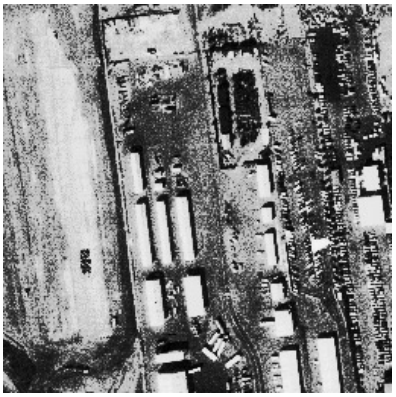


(d) Equalized Histogram

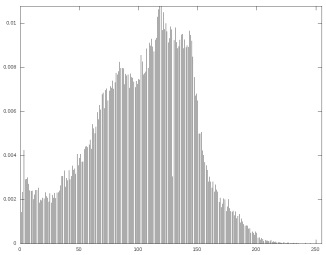
Figure 3: Histogram Equalization lax.pgm

(a) Original

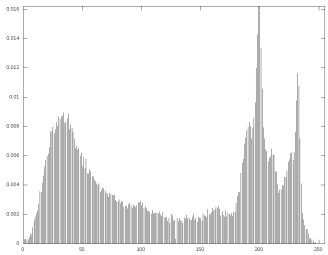
(b) After Specification 1



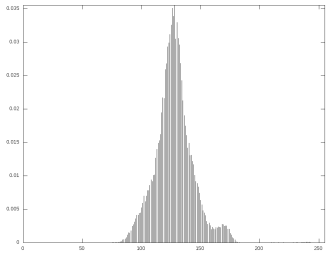
(c) After Specification 2



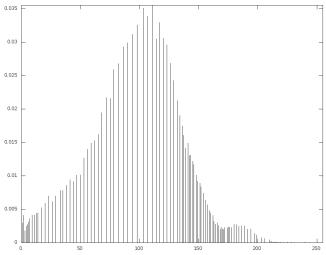
(d) Specification Histogram 1



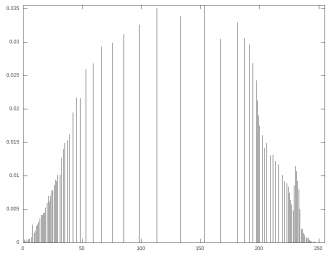
(e) Specification Histogram 2



(f) Original Histogram



(g) Histogram After 1



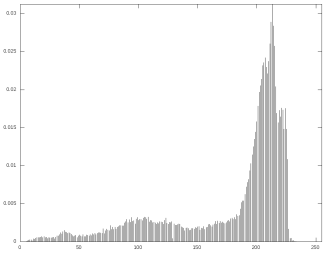
(h) Histogram After 2

Figure 4: Histogram Specification lax.pgm

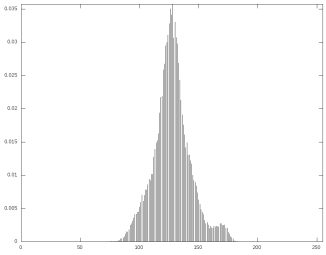
(a) Original

(b) After Specification 1

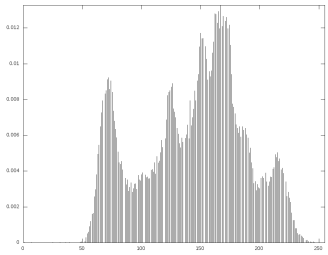
(c) After Specification 2



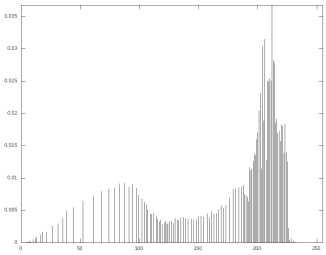
(d) Specification Histogram 1



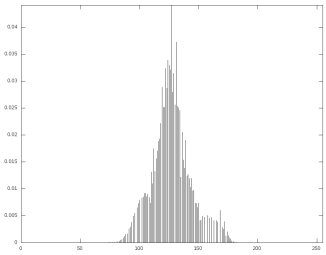
(e) Specification Histogram 2



(f) Original Histogram



(g) Histogram After 1



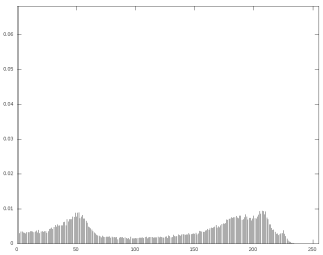
(h) Histogram After 2

Figure 5: Histogram Specification lenna.pgm

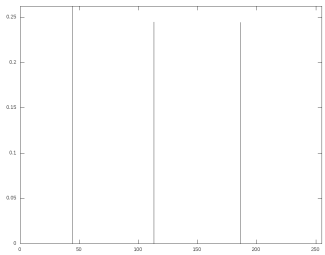
(a) Original

(b) After Specification 1

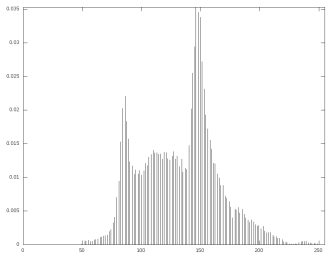
(c) After Specification 2



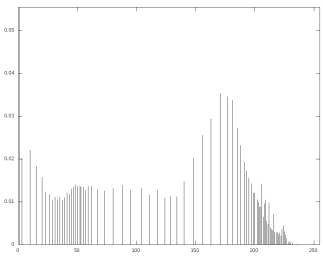
(d) Specification Histogram 1



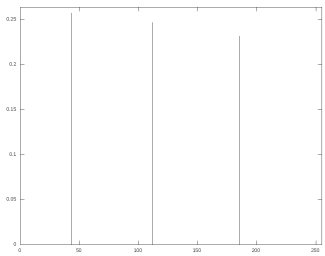
(e) Specification Histogram 2



(f) Original Histogram



(g) Histogram After 1

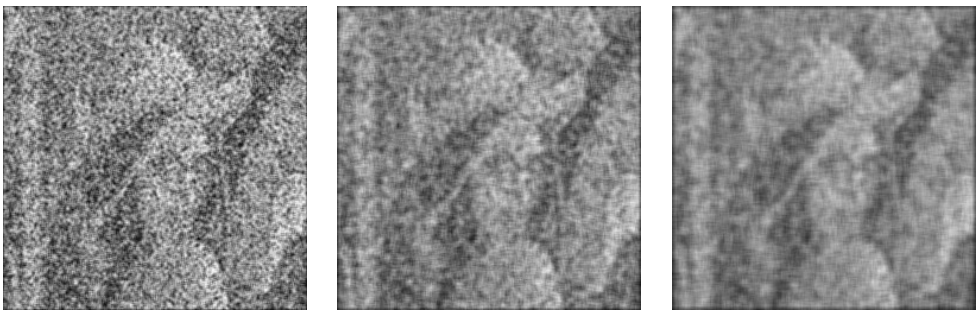


(h) Histogram After 2

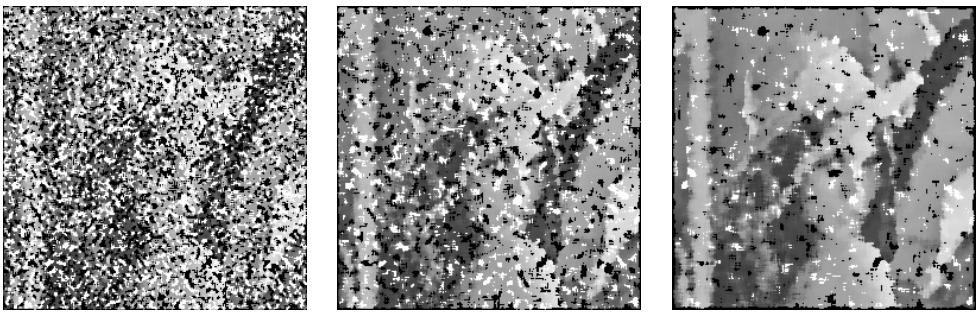
Figure 6: Histogram Specification sf.pgm

(a) Original (b) 50% Salt and Pepper (c) 75% Salt and Pepper

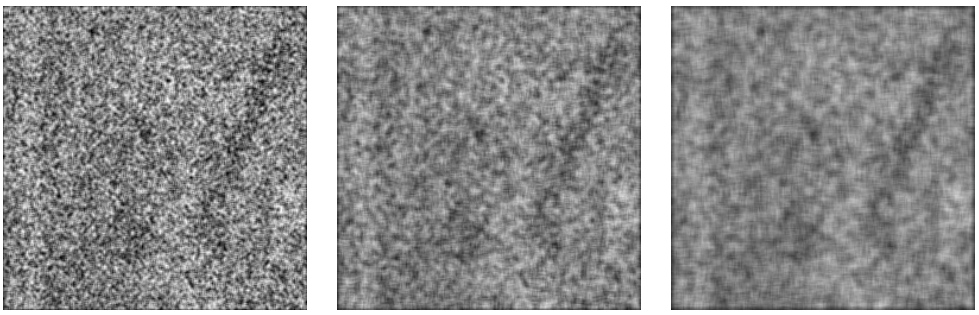
(d) 50% 3x3 Median Filter (e) 50% 5x5 Median Filter (f) 50% 7x7 Median Filter



(g) 50% 3x3 Mean Filter (h) 50% 5x5 Mean Filter (i) 50% 7x7 Mean Filter



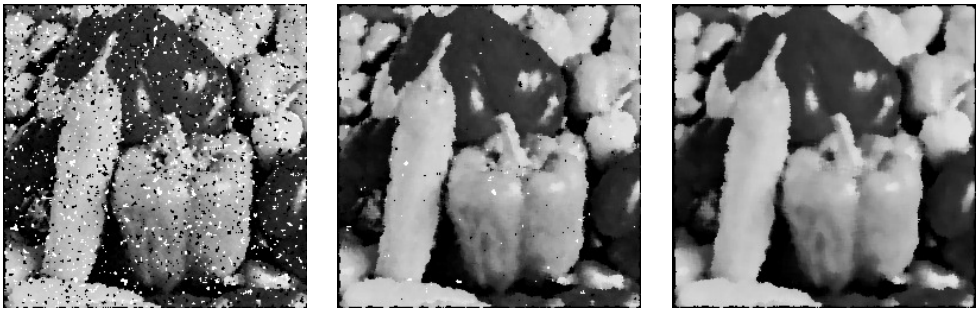
(j) 75% 3x3 Mean Filter (k) 75% 5x5 Mean Filter (l) 75% 7x7 Mean Filter



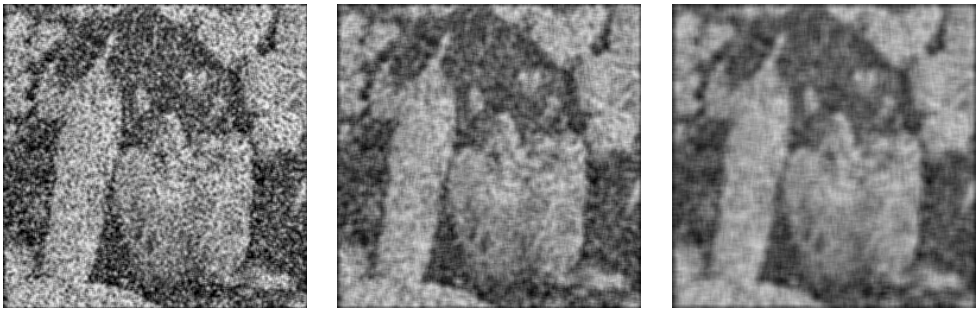
(m) 75% 3x3 Mean Filter (n) 75% 5x5 Mean Filter (o) 75% 7x7 Mean Filter

Figure 7: Median Filter lenna.pgm

(a) Original (b) 50% Salt and Pepper (c) 75% Salt and Pepper

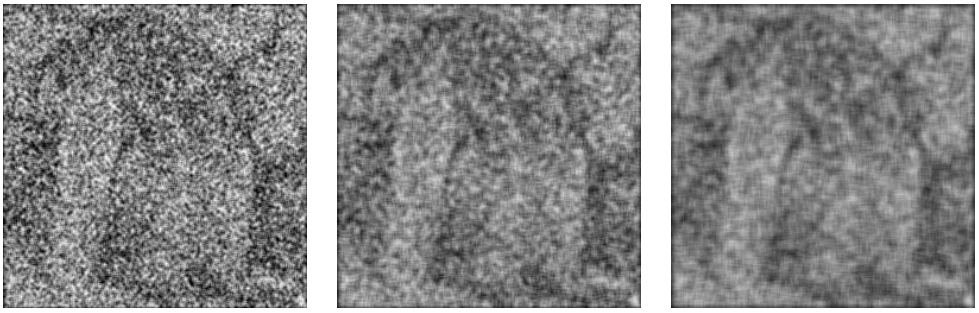


(d) 50% 3x3 Median Filter (e) 50% 5x5 Median Filter (f) 50% 7x7 Median Filter



(g) 50% 3x3 Mean Filter (h) 50% 5x5 Mean Filter (i) 50% 7x7 Mean Filter

(j) 75% 3x3 Mean Filter (k) 75% 5x5 Mean Filter (l) 75% 7x7 Mean Filter



(m) 75% 3x3 Mean Filter (n) 75% 5x5 Mean Filter (o) 75% 7x7 Mean Filter

Figure 8: Median Filter peppers.pgm