97+10

Image Enhancement through Point Processing (September 16, 2011)

Ali Ghezawi

ECE 572 – Digital Image Processing

***Abstract*— This project dealt with numerous point-processing techniques for image enhancement. Further, it worked with histograms as an effective tool for deciding how to and for actually making an image appear better. Histogram equalization and image averaging to remove Gaussian noise with zero mean was also discussed.**

I. Questions and Answers

A. An explanation of the difference between image sampling and image quantization. From two aspects, define image resolutions.

Sampling is the digitization of an image in the spatial domain whereas quantization is the digitization of an image in its amplitude or grayscale (for a grayscale image). Image resolution can be thought of both in space or in amplitude. That is, sampling determines the smallest seeable detail (in space), and quantization determines the smallest difference in intensities (in amplitude). Both contribute to the overall quality or 'resolution' of an image.

B. Explain pros and cons between vector representation and bitmap representation.

Vectors can show simple geometric shapes in a compact, scalable fashion easily, but it struggles to show complex combinations of shapes and colors. Bitmaps can show complex combinations of shapes and colors easily and can be converted to other formats easily, but it has trouble scaling since it loses quality. Vectors can be smaller than bitmaps for images with simple shapes whereas bitmaps can be smaller than vectors for complex images. Vectorization of an image can reveal boundaries in it automatically. image filters on bitmap.... -1

C. Comment on the effect by using different parameters of the power-law transformation. Comment on the similarity/difference between log transformation and power-law transformation. Can power-law transformation replace log transformation?

When $\gamma = 1$, the power-law transformation is just a scaled identity transformation, equaling the identity transformation when $c = 1$ too. As γ shrinks below 1, the transformation more aggressively maps a small range of darks into a larger range, also mapping a large range lights into a smaller range. As γ grows beyond 1, the transformation more aggressively maps a small range of lights into a larger range, also mapping a large range of darks into a smaller range.

The power-law transformation can be similar to the log transformation for certain $\gamma < 1$. However, it can also be very different since γ can be adjusted to make so many transformation curves, giving it greater versatility. The log transform specializes in compressing the dynamic range of an image, and the log transformation can never be mimicked for all input values by the power-law transformation. Thus, its replacement is not possible.

D. Describe what bit-plane slicing does. Does the most significant bit always contribute the most?

Bit-plane slicing generates images with ~~grayscale~~ binary values of either black or white that show the contribution of a certain bit. If white, the bit contributes to that location, and if black, it does not contribute to that location. A Gedankenexperiment can be created to disprove the theory that the most significant bit (MSB) always contributes the most to the image. If we define a new image as the bit-plane slice of some arbitrary image, showing the contribution by the first bit, then in all cases where the image has more than one bit, it is impossible for the MSB to be greater than the LSB. Hence, the theory is not true in general.

E. Comment on the difference (pros/cons) between contrast stretching and histogram equalization. Can they replace each other?

Contract stretching (CS) changes an image by stretching its intensities linearly. CS has more parameters for tuning, giving it the possibility to output precisely what the user wants, yet having parameters can also be a con since they must be tuned in the first place. CS tends to have a softer effect than histogram equalization (HE). Though this can be a con, it can also be a pro since the dramatic effect of HE is often bad – HE tends to produce an artificial graininess when the histogram is clumped together with few intensities in the image. The linearity, though, can be an issue when it restricts the contrast enhancing effect by making it impossible to stretch the contrast too much without obliterating the darkest or lightest detail by linearly pushing them all toward the floor or ceil, respectively.

HE, on the other hand, works to craft a probability distribution function as uniform as possible. HE has no parameters, giving it the pro of ease of use. However, the lack of tuning can be unwanted if the output is undesirable since no tuning can rectify the disappointing result. HE can also suffer due to the algorithm working on the entire image. For

example, a region with globally dark yet locally contrasting darkness can remain undetectably dark in the output. The use of a local HE can remedy this problem.

Since CS works better than HE in certain applications and vise-versa, no replacement is possible.

F. Show the derivation for histogram equalization

Let

$$s = T(r) \tag{1}$$

where s is the new intensity, r is the old intensity, T is a mapping function whose goal is to bring s's normalized histogram (or probability density function (PDF)) to uniformity. It is also assumed that s and r are continuous with a continuous PDF.

Since they have PDFs, r and s can be thought of as random variables between 0 and L – 1 where S is uniform, R is arbitrary in its distribution, and L is the greatest intensity. Their PDFs are deterministically related since S is a function of R as

$$p_S(s) = p_R(r)\left|\frac{dr}{ds}\right| \tag{2}$$

where monotonicity in T is assumed to guarantee a unique mapping and to guarantee ds/dr is nonnegative. Solving for ds/dr gives

$$\left|\frac{ds}{dr}\right| = \frac{ds}{dr} = \frac{p_R(r)}{p_S(s)} \tag{3}$$

where p_S is uniform by definition and therefore a constant as defined earlier over the interval defined earlier. So its value can be found simply as

$$\begin{aligned} p_S(s) &= c \\ \int_{-\infty}^{\infty} p_S(s)ds &= (L-1)c = 1 \\ p_S(s) &= \frac{1}{L-1} \end{aligned} \tag{4}$$

Integrating both sides of (3) with respect to r and inserting the result from (4) then results in

$$s = T(r) = (L-1)\int_{-\infty}^{r} p_R(w)dw \tag{5}$$

So the transform, in continuous time, is the cumulative distribution of R evaluated at r and scaled by $L – 1$. The discrete analog is then

$$s = T(r) = \frac{L-1}{n_1 n_2}\sum_{w=0}^{r} H[w] \tag{6}$$

w=0,1,2,.....,L-1 -1

where L is the number of intensities, n_1 is the number of rows, n_2 is the number of columns, and H is a histogram of the original image. That is, H divided by the area of the image approximates a probability density function whose sum is a cumulative distribution function.

G. Prove Equations 2.6-6 and 2.6-7 on page 75 to help understand the image averaging algorithm for enhancement.

Let us first define a single random variable f_{Gn} whose effect is the sum of n Gaussian random variables X whose means are zero and standard deviations are σ.

$$f_{G_n}(g) = f_{X_1}(g) * f_{X_2}(g) \ldots f_{X_n}(g) \tag{7}$$

Using the Fourier transform to work with multiplication instead of convolution, we find

$$\begin{aligned} F_{G_n}(f) &= \prod_{i=1}^{n} \exp\left(-2\pi^2\sigma^2 f\right) \\ F_{G_n}(f) &= \exp\left(-2\pi^2\sigma^2 f \sum_{i=1}^{n} 1\right) \\ F_{G_n}(f) &= \exp\left(-2\pi^2 n\sigma^2 f\right) \end{aligned} \tag{8}$$

We take the inverse Fourier transform to find the resulting PDF as

$$\begin{aligned} f_{G_n}(g) &= \mathcal{F}^{-1}\left\{\exp\left(-2\pi^2 n\sigma^2 f\right)\right\} \\ f_{G_n}(g) &= \frac{1}{\sqrt{2\pi n\sigma^2}}\exp\left(-\frac{g}{2n\sigma^2}\right) \end{aligned} \tag{9}$$

which is the PDF of a Gaussian random variable whose mean is zero and variance is $n\sigma^2$.

If we say we have n images with noise added to them described by X, we can write the average of these n noisy images as

$$\bar{m}(x,y) = \frac{nm(x,y) + f_{G_n}(g)}{n} = m(x,y) + \frac{f_{G_n}(g)}{n} \tag{10}$$

where m is the original image. The expectation then is

$$\begin{aligned} E\{\bar{m}(x,y)\} &= E\left\{m(x,y) + \frac{f_{G_n}(g)}{n}\right\} \\ E\{\bar{m}(x,y)\} &= E\{m(x,y)\} + \frac{1}{n}E\{f_{G_n}(g)\} \end{aligned} \tag{11}$$

where the expectation operator's linearity made it possible to distribute the operator onto the two terms and factor out the reciprocal of *n*. The expectation of a Gaussian random variable is its mean defined by its distribution. Equation (9) shows the mean of the sum of *n X*s is zero, so that expectation in (11) of f_{Gn} reduces to zero. Further, the expectation of a constant is just that same constant, so the expectation of the constant image is the image itself. Hence,

$$E\left\{\bar{m}(x,y)\right\} = m(x,y) \tag{12}$$

The variance can also be found with

$$V\left\{\bar{m}(x,y)\right\} = V\left\{m(x,y) + \frac{f_{G_n}(g)}{n}\right\} \tag{13}$$

A constant added within the variance operator does not change the result, so the constant image is removed from within the operator. Further, constants can be factored out of the variance operator if squared. Thus, we have

$$V\left\{\bar{m}(x,y)\right\} = \frac{1}{n^2} V\left\{f_{G_n}(g)\right\} \tag{14}$$

And the variance of a Gaussian is defined by the variance in its PDF. Hence, by looking at (9), we have

$$V\left\{\bar{m}(x,y)\right\} = \frac{1}{n^2} n\sigma^2 = \frac{\sigma^2}{n} \tag{15}$$

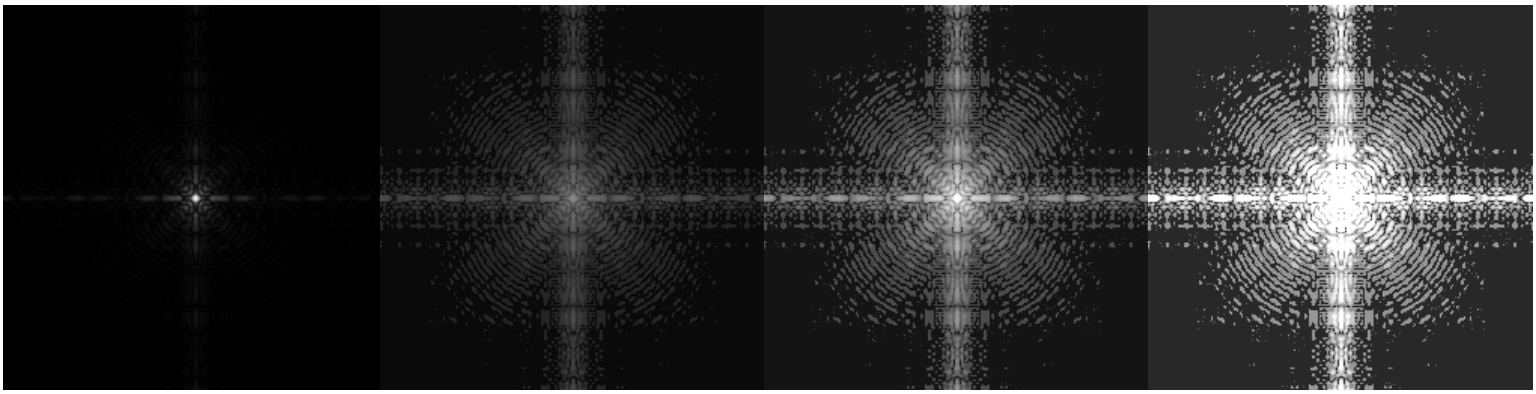
II. Result

A. Task 1.1

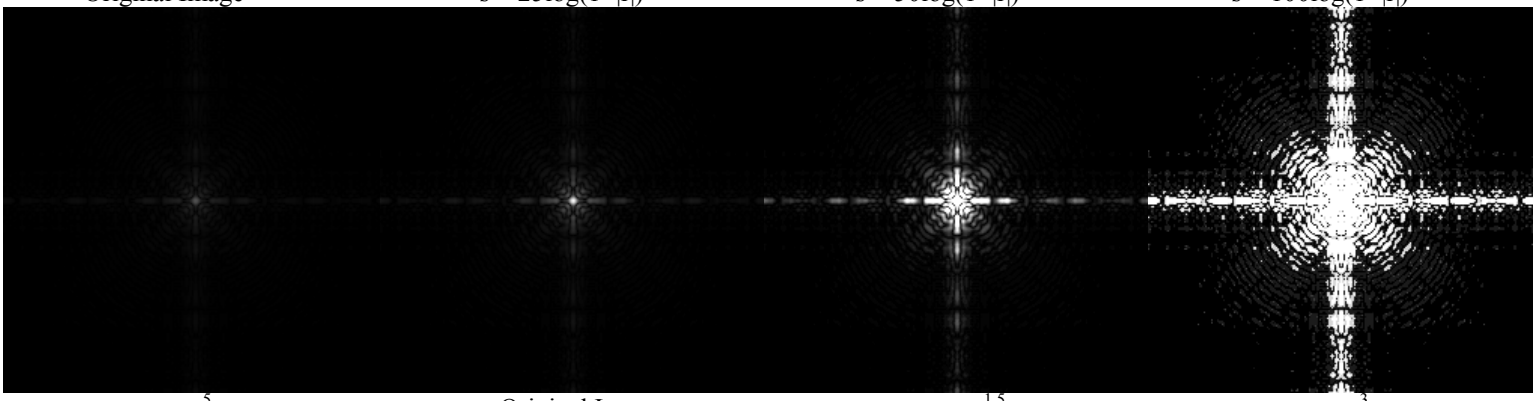
Original Image — Subsampled with $s = 4$

Original Image — Quantized with $q = 4$

B. Task 1.2

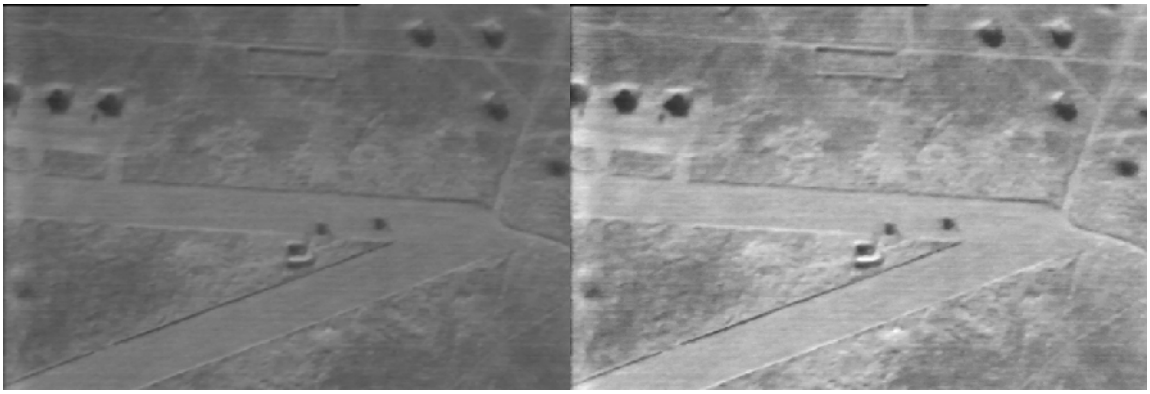


Original Image | s = 25log(1+|r|) | s = 50log(1+|r|) | s = 100log(1+|r|)

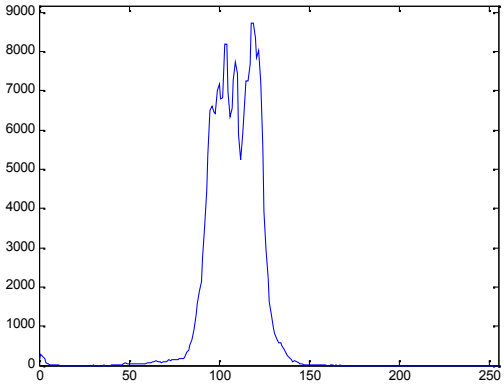


$s = r^{-5}$ | Original Image | $s = r^{1.5}$ | $s = r^{3}$

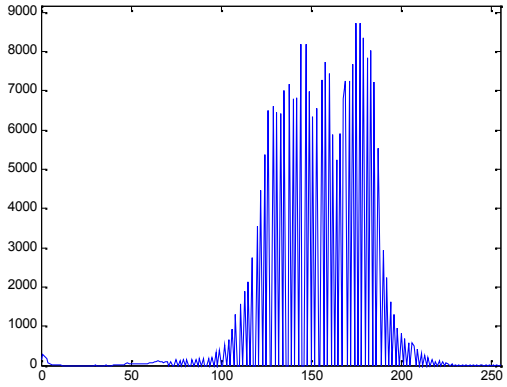
C. Task 1.3



Original Image | doCs bunker.pgm bunker2.pgm 1 0 70 2.25 -87.5 110 1.9 -49

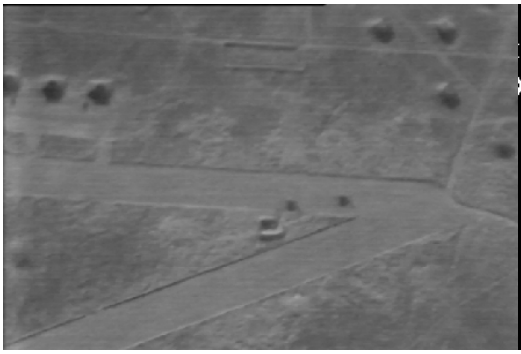


Original Histogram



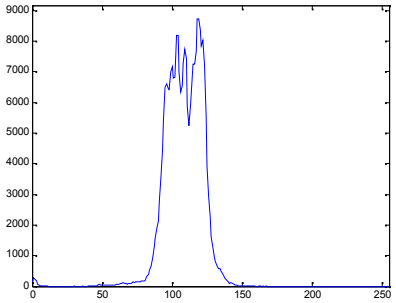
Contrast Stretched Histogram

D. Task 1.4



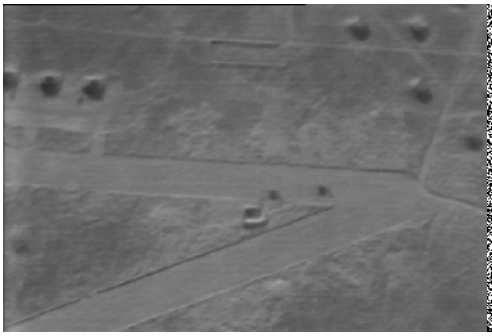
Original Image

Threshold $A = 50$, $B = 70$

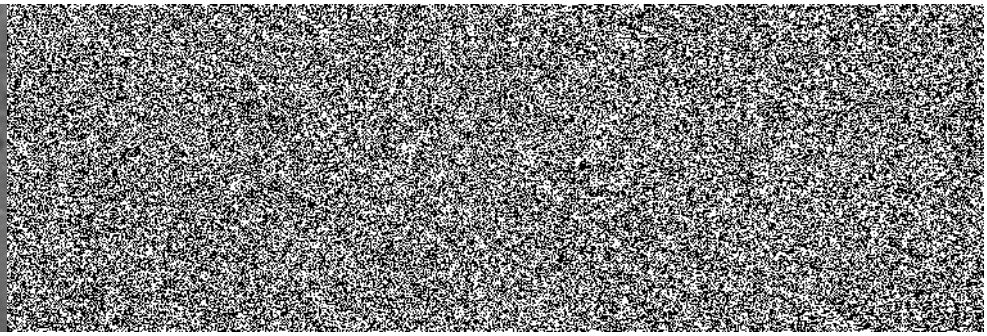


Original Histogram

E. Task 1.5

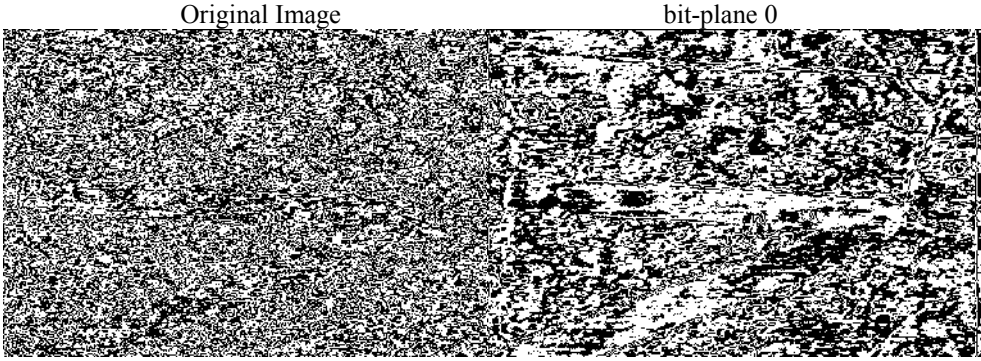


Original Image



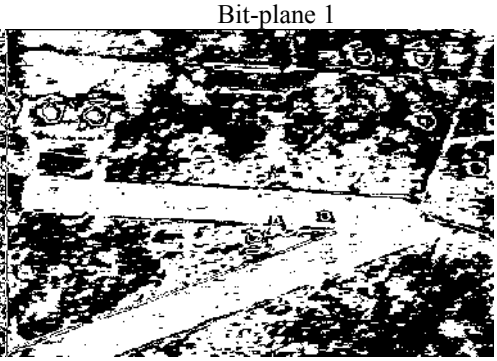
bit-plane 0

Bit-plane 1

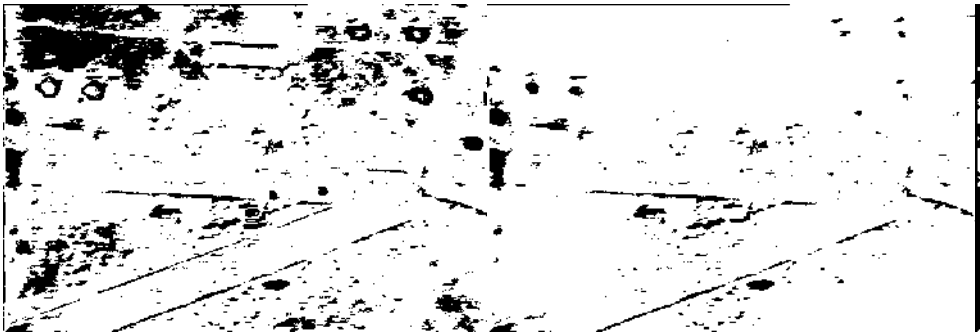


Bit-plane 2

bit-plane 3



Bit-plane 4

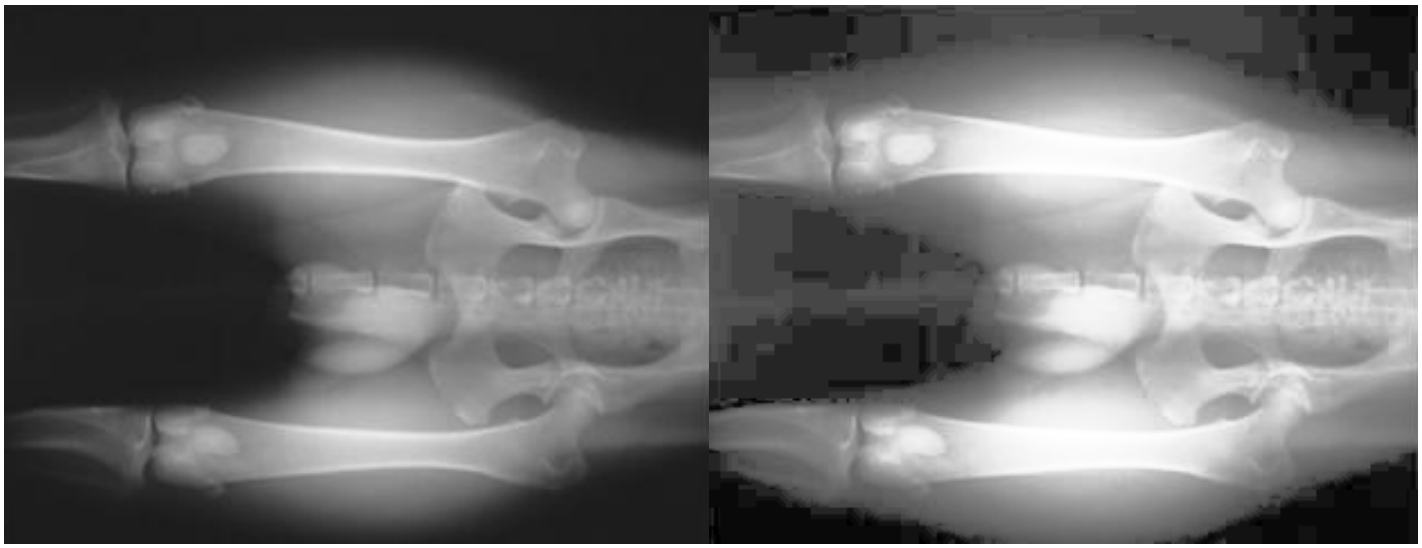


Bit-plane 5

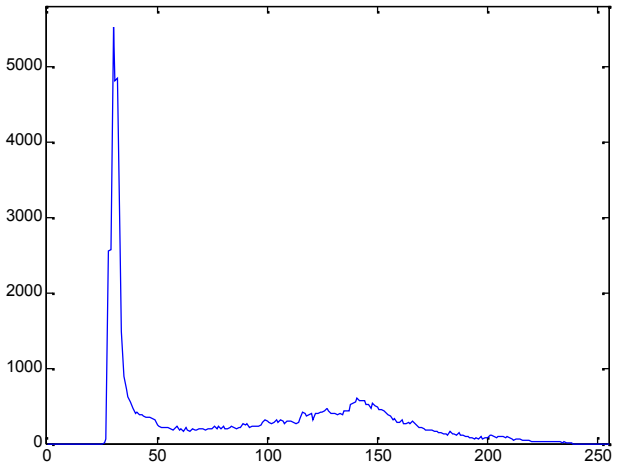
bit-plane 6

Bit-plane 7

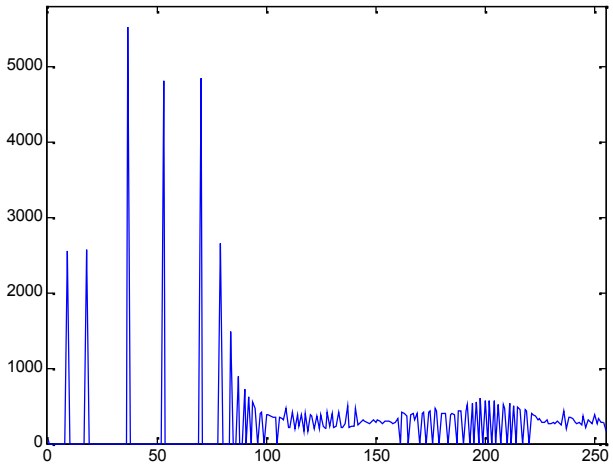
F. Task 2



Original Image | Histogram Equalized Image



Original Histogram

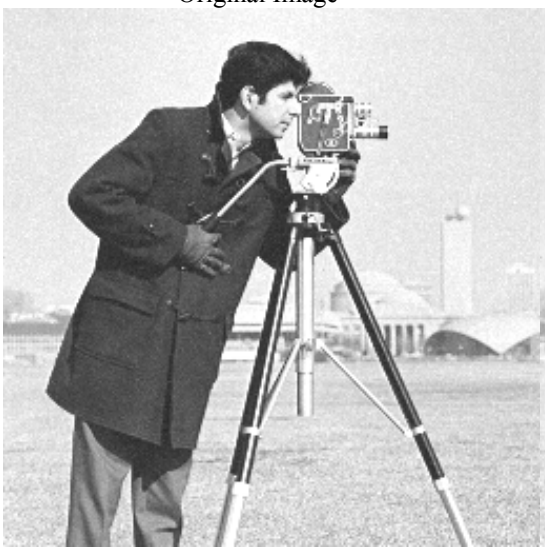


Histogram Equalized Histogram

G. Task 3

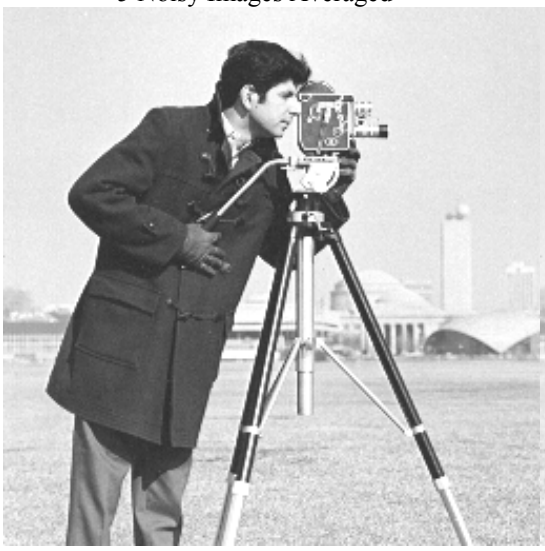
Original Image

Sample Noisy Image ($\sigma = 20$)



5 Noisy Images Averaged

10 Noisy Images Averaged

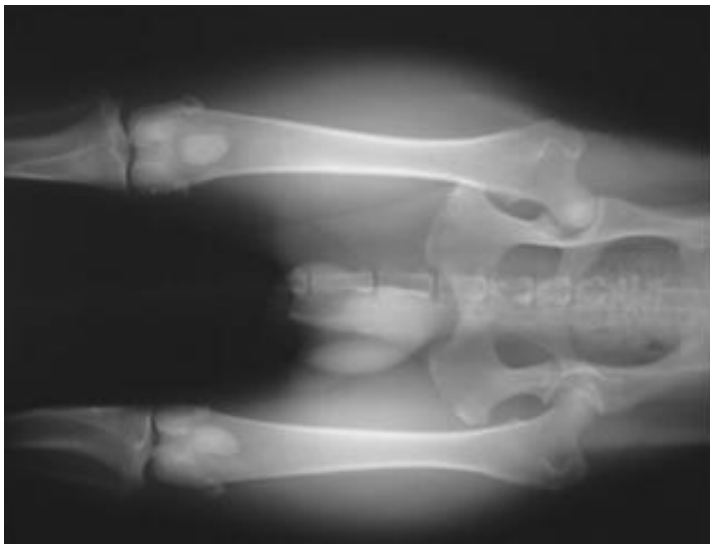


20 Noisy Images Averaged

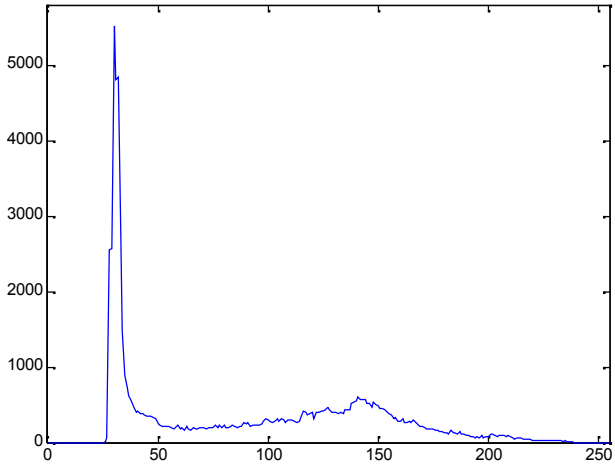
100 Noisy Images Averaged

The more images averaged, the closer the image looks to the original. This result is in agreement with (12).

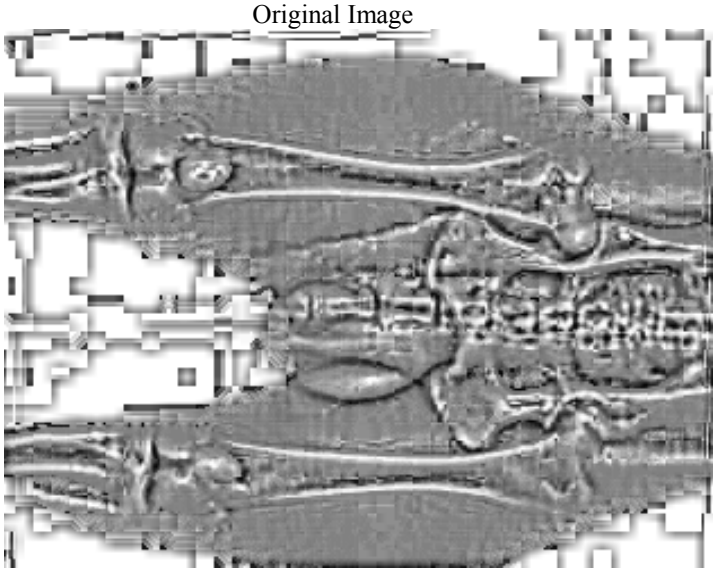
H. Bonus



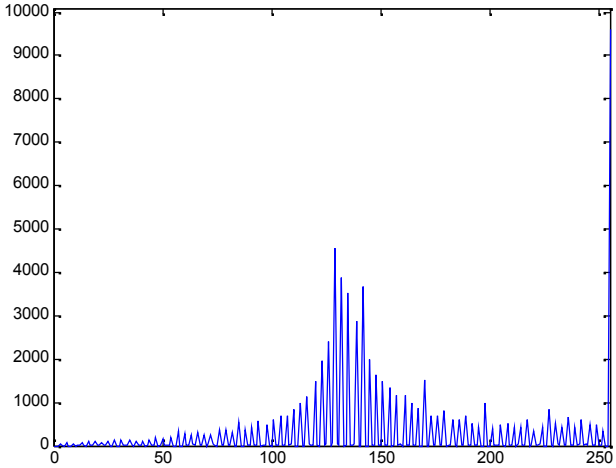
Original Image



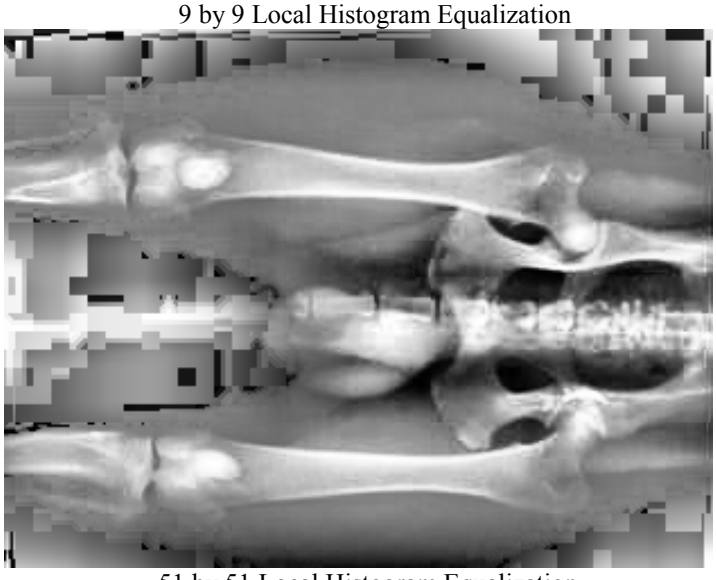
Original Histogram



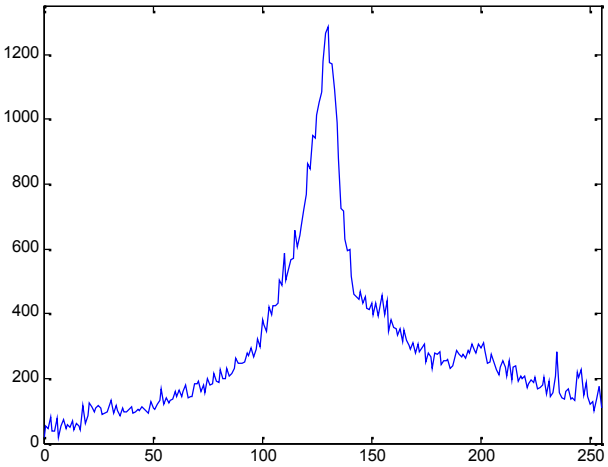
9 by 9 Local Histogram Equalization



9 by 9 Local Histogram Equalized Histogram



51 by 51 Local Histogram Equalization



51 by 51 Local Histogram Equalized Histogram

III. Discussion

The work revealed some of the advantages of viewing and manipulating the histogram of an image. Specifically, letting major humps in the image stretch, either linearly or otherwise such as with histogram equalization, makes an image appear enhanced over the original usually. Contrast enhancing via the log and gamma transformations also taught the importance of having humps on the histogram closer together than farther apart, and these transformations paired with CS or HE would most likely result in the optimal effect. Both of these transformations worked upon the Fourier image to make them more pleasant for viewing, or in the case of gamma below 1, less pleasant to view. Last, the probability principles of expectation and variance of n summed Gaussian random variables whose means were zero became much clearer after developing the proof and experiencing it in practice. Future replicators of this project should be required to operate on a single image with multiple tools (such as log then HE) to see if they benefit each other.

APPENDIX

A. *doSampling.cpp (main)*

Please attach the code in requried format
See project 1 website -1

```
// test sampling()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doSampling inImg outImg s\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 4)
   {
      cout << Usage;
      exit(3);
   }
   else if(atoi(argv[3]) < 1)
   {
      cout << "S cannot be less than one.\n";
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // Create output image with sampling effect.
   const Image OUT_IMG = sampling(IN_IMG, atoi(argv[3]));

   // Output the image
   writeImage(OUT_IMG, argv[2]);

   // Success!
   return(0);
}
```

B. *doQuantization.cpp (main)*

```
// test quantization()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doQuanization inImg outImg q\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 4)
   {
      cout << Usage;
      exit(3);
   }
   else if(atoi(argv[3]) <= 1 || atoi(argv[3]) & (atoi(argv[3]) - 1))
   {
      cout << "Q must be a power of 2 greater than 1.\n";
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // Create output image with quantization effect.
   const Image OUT_IMG = quantization(IN_IMG, atoi(argv[3]));

   // Output the image
   writeImage(OUT_IMG, argv[2]);

   // Success!
   return(0);
}
```

C. *doLogtran.cpp (main)*

```
// test logtran()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doLogtran inImg outImg c\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 4)
```

```
   {
      cout << Usage;
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // Create output image with logtran effect.
   const Image OUT_IMG = logtran(IN_IMG, atof(argv[3]));

   // Output the image
   writeImage(OUT_IMG, argv[2]);

   // Success!
   return(0);
}
```

D. doPowerlaw.cpp (main)

```
// test powerlaw()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doPowerlaw inImg outImg c gamma\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 5)
   {
      cout << Usage;
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // Create output image with powerlaw effect.
   const Image OUT_IMG = powerlaw(IN_IMG, atof(argv[3]), atof(argv[4]));

   // Output the image
   writeImage(OUT_IMG, argv[2]);

   // Success!
   return(0);
}
```

E. doCs.cpp (main)

```
// test cs()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doCs inImg outImg m1 b1 s1 m2 b2 s2 m3 b3\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 11)
   {
      cout << Usage;
      exit(3);
   }
   if(atoi(argv[5]) < 0 || atoi(argv[5]) > atoi(argv[8]) || atoi(argv[8]) > 255)
   {
      cout << "s1, s2, or s3 are problematic.\n";
      exit(3);
   }


   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // outputs that will be used to construct the final output
   const Image IMG1 = cs(IN_IMG, atof(argv[3]), atof(argv[4]));
   const Image IMG2 = cs(IN_IMG, atof(argv[6]), atof(argv[7]));
   const Image IMG3 = cs(IN_IMG, atof(argv[9]), atof(argv[10]));

   // initialize output
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // piece together the image.
   for (int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            if(IN_IMG(row, col, chan) <= atoi(argv[5]))
               outImg(row, col, chan) = IMG1(row, col, chan);
            else if(IN_IMG(row, col, chan) <= atoi(argv[8]))
               outImg(row, col, chan) = IMG2(row, col, chan);
            else if(IN_IMG(row, col, chan) > atoi(argv[8])) //i.e. s2 != 255 && r > s2
```

```
                outImg(row, col, chan) = IMG3(row, col, chan);

    // Output the image
    writeImage(outImg, argv[2]);

    // save before and after histograms
    saveHist(IN_IMG, 0, "input.dat");
    saveHist(outImg, 0, "output.dat");

    // Success!
    return(0);
}
```

F. doThreshold.cpp (main)

```
// test threshold()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doThreshold inImg outImg a b\n"

int main(int argc, char **argv)
{
    // Check for proper number of arguments.
    if(argc < 5)
    {
        cout << Usage;
        exit(3);
    }

    // Read in image specified by user.
    const Image IN_IMG = readImage(argv[1]);

    // Create output image with threshold effect.
    const Image OUT_IMG = threshold(IN_IMG, atof(argv[3]), atof(argv[4]));

    // Output the image
    writeImage(OUT_IMG, argv[2]);

    // save histograms
    saveHist(IN_IMG, 0, "input.dat");

    // Success!
    return(0);
}
```

G. doBitplane.cpp (main)

```
// test bitplane()
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doBitplane inImg outImgBaseName\n"

int main(int argc, char **argv)
{
    // Check for proper number of arguments.
    if(argc < 3)
    {
        cout << Usage;
        exit(3);
    }

    // Read in image specified by user.
    const Image IN_IMG = readImage(argv[1]);

    // determine size of base output name
    const size_t OUTPUT_SIZE = strlen(argv[2]);

    // allocate and initialize output name char array
    char* outputName = new char[OUTPUT_SIZE + 1];
    strcpy(outputName, argv[2]);
    strcpy(outputName + OUTPUT_SIZE - 4, ".pgm");

    // perform 8 bitplane slices and save them under basename{slice#}
    for(int plane = 0; plane < 8; ++plane)
    {
        // Create output image with bitplane effect.
        const Image OUT_IMG = bitplane(IN_IMG, plane);

        // alter name
        char temp[2];
        itoa(plane, temp, 10);
        strcpy(outputName + OUTPUT_SIZE - 5, temp);
        outputName[OUTPUT_SIZE - 4] = '.';

        // Output the image
        writeImage(OUT_IMG, outputName);
    }

    delete [] outputName;
```

```
   // Success!
   return(0);
}
```

H. doHistEq.cpp (main)

```
// test histeq()
// also saves histogram of input and output under
// i.dat for input image and o.dat for output image.

#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doHisteq inImg outImg\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 3)
   {
      cout << Usage;
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // Create output image with histeq effect.
   const Image OUT_IMG = histeq(IN_IMG);

   // Output the image
   writeImage(OUT_IMG, argv[2]);

   // save input image and output image's histograms for matlab
   saveHist(IN_IMG, 0, "input.dat");
   saveHist(OUT_IMG, 0, "output.dat");

   // Success!
   return(0);
}
```

I. doLocalHistEq.cpp (main)

```
// test localHistEq()
// also saves histogram of input and output under
// i.dat for input image and o.dat for output image.

#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "doLocalHistEq inImg outImg windowRow windowCol\n"

int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 5)
   {
      cout << Usage;
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // Create output image with histeq effect.
   const Image OUT_IMG = localHistEq(IN_IMG, atoi(argv[3]), atoi(argv[4]));

   // Output the image
   writeImage(OUT_IMG, argv[2]);

   // save input image and output image's histograms for matlab
   saveHist(IN_IMG, 0, "input.dat");
   saveHist(OUT_IMG, 0, "output.dat");

   // Success!
   return(0);
}
```

J. doGaussNoise.cpp (main)

```
// test addGaussNoise()
#include "Dip.h"
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

#define Usage "doGaussNoise inImg outImgBaseName trials SD\n"
```

```
int main(int argc, char **argv)
{
   // Check for proper number of arguments.
   if(argc < 4)
   {
      cout << Usage;
      exit(3);
   }

   // Read in image specified by user.
   const Image IN_IMG = readImage(argv[1]);

   // initialize output image to all zeros
   Image averagedImage(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType()); // output, will contain average
   for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
         for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
               averagedImage(row, col, chan) = 0.0;

   // allocate outputname string for noisy images
   const int NAME_SIZE = strlen(argv[2]);
   char* outputNoiseName = new char[NAME_SIZE + 8]; // trials can be no larger than 9999
   strcpy(outputNoiseName, argv[2]);
   strcpy(outputNoiseName + NAME_SIZE - 4, "NOISY");

   // seed random number generator
   srand(time(NULL));

   for(int trial = 1; trial < atoi(argv[3]) + 1; ++trial)
   {
      // number the name
      outputNoiseName[NAME_SIZE + 1] = '\0';
      char trialNum[5];
      itoa(trial, trialNum, 10);
      strcat(outputNoiseName, trialNum);
      strcat(outputNoiseName, ".pgm");

      // add noise to input image
      const Image NOISY_IMG = addGaussNoise(IN_IMG, atoi(argv[4]));

      // add noise to averaged image
      for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
         for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
               averagedImage(row, col, chan) += NOISY_IMG(row, col, chan);

      //output noisy image under baseNameNoise + trial#
      writeImage(NOISY_IMG, outputNoiseName);
   }

   // average the summed image
   averagedImage = averagedImage/atoi(argv[3]);

   // Output the image
   writeImage(averagedImage, argv[2]);

   // Success!
   return(0);
}
```

K. Dip.h

```
/*****************************************************************
 * Dip.h - header file of the Image processing library
 *
 * Author: Hairong Qi, hqi@utk.edu, ECE, University of Tennessee
 * Author: Ali Ghezawi, aghezawi@utk.edu (all except cs())
 *
 * Created: 01/22/06
 *
 * Modification:
 * 9/5/11 added gaussian noise function and point-based image enhancement
 * 8/22/11 added oil, swirl, edge, pixelMatrixMult
 *****************************************************************/

#ifndef DIP_H
#define DIP_H

#include "Image.h"

#define PI 3.1415926

/*
 * Gaussian Noise Functions
 */
// Returns a single sample of a Gaussian random variable with mean 0
float gaussianNoise(const float&);

// returns input image with noise from gaussianNoise added to it.
Image addGaussNoise(const Image&, const float&);    // input image

/*
 * point-based image enhancement processing
 * note: all can work for RGB pictures, but the
 * algorithms are designed for graycale.
 */
```

```
// samples an image by S
Image sampling(const Image&,  // input image
   const int&);              // S

// quantizes an image's levels
Image quantization(const Image&,  // input image
   const int&);                  // # of levels, > 1

// performs log transformation on each pixel
Image logtran(const Image&,       // input image
   const float&);                // scaling factor

// performs power law (gamma) transformation on each pixel
Image powerlaw(const Image&,  // input image
   const float&,              // scaling factor
   const float&);             // gamma

// performs contrast stretching
Image cs(const Image &,
   const float&,                  // slope
   const float&);                 // intercept

// performs threshold effect
Image threshold(const Image&,      // input image
   const float&,                   // bottom threshold
   const float&);                  // top threshold

// performs bitplane slicing
Image bitplane(const Image&,     // input image
   const int&);                  // BIT to slice

// performs histogram equalization
Image histeq(const Image& ); // input image

// saves histogram of image to a file
void saveHist(const Image&,   // input image to take hist of
   const int&,                // what channel to operate in
   char*);                    // filename to save hist data to

// performs localized histogram equalization. Each pixel
// has histeq() done to it using a ROW by COL box around
// that pixel. ROW & COL must be odd & > 1
Image localHistEq(const Image&,     // input image
   const int&,                      // ROW
   const int&);                     // COL

// Swirls an image about its center.
Image swirl(const Image&,            // Input image
   const float&);                   // swirl coefficient

// outputs an image with edges highlighted.
Image edge(const Image& IN_IMG);     // Input image

// Makes an image look as if it were painted with oil paints.
Image oil(const Image&,              // Input image
   const int&);                     // arg2 = (s-1)/2 of search sqr

// sum(sum(MASK*neighborhoodOfPixel)). Mask nbyn, n%2 !=0
float pixelMatrixMult(const Image&,  // Input image
   const int&,                      // ROW of pixel
   const int&,                      // COL of pixel
   const int&,                      // CHANNEL being worked in
   const float*,                    // ->MASK (flattened into 1d)
   const size_t&);                  // size of one side of the MASK.
#endif
```

L. pointProcessping.cpp

```
/*************************************************************
 * pointProcessing.cpp: 8 point processing functions
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/5/11
 *
 *************************************************************/

#include "Dip.h"
#include <iostream>
#include <cmath>
#include <fstream>

/*
 * Sampling by s where s >= 1
 * @param IN_IMG input image
 * @param S the sampling amount
 * @return sampled image
 */
Image sampling(const Image& IN_IMG, const int& S)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform sampling
   for (int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
```

```
            outImg(row, col, chan) = IN_IMG(row - row%S, col - col%S, chan);

   return outImg;
}

/*
 * Quantization by Q
 * @param IN_IMG input image
 * @param Q number of levels in output, q > 1
 * @return quantized image
 */
Image quantization(const Image& IN_IMG, const int& Q)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform quantization
   for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = std::floor(255*float(IN_IMG(row, col, chan) - int(IN_IMG(row, col, chan))%(256/Q))/(255-255%(256/Q)) + .5);

   return outImg;
}

/*
 * Log transformations. s = c*log(1 + |r|) where
 * s: enhanced pixel intensity
 * r: original pixel intensity
 * c: scaling constant
 * @param IN_IMG input image
 * @param c scaling constant
 * @return log transformed image.
 */
Image logtran(const Image& IN_IMG, const float& C)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform log transformation
   for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = std::floor(C*log(1 + std::abs(IN_IMG(row, col, chan)) + .5));

   return outImg;
}

/*
 * Power-law (gamma) transformation s = c*r^(g), g > 0
 * s: enhanced pixel intensity
 * r: original pixel intensity
 * g: gamma. < 1 => brighter, > 1 => darker, else same
 * c: scaling constant
 * @param IN_IMG input image
 * @param C scaling constant
 * @param GAMMA gamma constant (g)
 * @return power-law (gamma) transformed image.
 */
Image powerlaw(const Image& IN_IMG, const float& C, const float& GAMMA)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform power-law (gamma) transformation
   for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = std::floor(C*std::pow(IN_IMG(row, col, chan), GAMMA) + .5);

   return outImg;
}

/*
 * Contrast stretching. s = m*r + b where
 * s: enhanced pixel intensity
 * r: original pixel intensity
 * @param IN_IMG Input image
 * @param m Slope
 * @param b Intercept
 * @return Contrast stretched image.
 */
Image cs(const Image& IN_IMG, const float& m, const float& b)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform contrast stretching
   for (int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row,col,chan) = std::floor(m*IN_IMG(row, col, chan) + b + .5);

   return outImg;
}

/*
 * gray-level slicing. s = 255 if A <= r <= B else 0
 * s: enhanced pixel intensity
 * r: original pixel intensity
 * A: lower threshold
```

use L instead of 255

```
 * B: higher threshold
 * @param IN_IMG Input image
 * @param A lower threshold
 * @param B upper threshold
 * @return threshold image
 */
Image threshold(const Image& IN_IMG, const float& A, const float& B)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform threshold
   for (int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = (IN_IMG(row, col, chan) >= A && IN_IMG(row, col, chan) <= B) ? 255 : 0;

   return outImg;
}

/*
 * Bitplane slicing. s = nth bit of r times 255
 * s: enhanced pixel intensity
 * r: original pixel intensity
 * n: the nth bit starting at n = 0
 * @param IN_IMG Input image
 * @param BIT is the bit to extract
 * @return returns a bitplane slice of input
 */
Image bitplane(const Image& IN_IMG, const int& BIT)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // perform bitplane slice
   for (int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for (int row = 0; row < IN_IMG.getRow(); ++row)
         for (int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = 255*((int(IN_IMG(row, col, chan))>>BIT)%2);

   return outImg;
}

/*
 * Histogram equalization. s = (L-1)*CDF_r(r)
 * s: enhanced pixel intensity
 * r: original pixel intensity
 * CDF_r: cumulative distribution function of original pixels
 * L: number of colors possible in channel (256)
 * @param IN_IMG input image
 * @return equalized image.
 */
Image histeq(const Image& IN_IMG)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());
   float cumulativeCount[256];

   for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
   {
      for(size_t color = 0; color < 256; ++color)
         cumulativeCount[color] = 0.0;

      //cumulativeCount holds frequency count
      for(int row = 0; row < IN_IMG.getRow(); ++row)
         for(int col = 0; col < IN_IMG.getCol(); ++col)
            ++cumulativeCount[size_t(IN_IMG(row, col, chan))];

      //makes cumulativeCount actually a cumulative count of occurrances now
      for(size_t color = 1; color < 256; ++color)
         cumulativeCount[color] += cumulativeCount[color - 1];//cumulative frequency count

      //normalize cumulative count and apply T(r), using it.
      for(int row = 0; row < IN_IMG.getRow(); ++row)
         for(int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = std::floor(255*cumulativeCount[size_t(IN_IMG(row, col, chan))]/IN_IMG.getRow()/IN_IMG.getCol() + .5);
   }

   return outImg;
}

/*
 * Saves historygram in .dat file
 * @param IN_IMG input image
 * @param CHAN is the channel to work in of IN_IMG
 * @param fileName pointer to char array containing file name to save to
 * @return file with normalized frequencies in it
 */
void saveHist(const Image& IN_IMG, const int& CHAN, char* fileName)
{
   float hist[256];

   // figure out the historgram quantities
   for(size_t color = 0; color < 256; ++color)
         hist[color] = 0.0;

   for(int row = 0; row < IN_IMG.getRow(); ++row)
      for(int col = 0; col < IN_IMG.getCol(); ++col)
         ++hist[size_t(IN_IMG(row, col, CHAN))];
```

```
   // save answer to file, camma delimited
   std::ofstream file(fileName);

   if(file.is_open())
   {
      for(size_t color = 0; color < 255; ++color)
         file << hist[color] << ",";
      file << hist[255] << std::endl;
      file.close();
   }
   else
      std::cout << "Unable to open file\n";
}

/*
 * Performs localized histogram equalization. It uses a
 * ROW by COL rectangle to define the neighborhood. Invalid regions are ignored.
 * @param IN_IMG input image
 * @param ROW row amount for neighborhood
 * @param col of neighborhood
 * @return local histogram equalized image
 * row & col must be odd & >= 3
 */
Image localHistEq(const Image& IN_IMG, const int& ROW, const int& COL)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   // calculate row/col to the left/above central pixel
   const int ROW_LEFT = (ROW - 1)/2; // ROW_LEFT = ROW_RIGHT
   const int COL_UP = (COL - 1)/2;   //COL_UP = COL_DOWN

   for(int row = 0; row < IN_IMG.getRow(); ++row)
      for(int col = 0; col < IN_IMG.getCol(); ++col)
      {
         // define window starts and finishes
         const int WIN_ROW_START = row - ROW_LEFT < 0 ? 0 : row - ROW_LEFT;
         const int WIN_COL_START = col - COL_UP < 0 ? 0 : col - COL_UP;
         const int WIN_ROW_END = row + ROW_LEFT + 1 > IN_IMG.getRow() ? IN_IMG.getRow() : row + ROW_LEFT + 1;
         const int WIN_COL_END = col + COL_UP + 1> IN_IMG.getCol() ? IN_IMG.getCol() : col + COL_UP + 1;
         const float AREA = (WIN_ROW_END - WIN_ROW_START)*(WIN_COL_END - WIN_COL_START);

         // do histeq for neighborhood
         for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
         {
            float cumulativeCount[256];
            for(size_t color = 0; color < 256; ++color)
               cumulativeCount[color] = 0.0;

            //cumulativeCount holds frequency count
            for(int winRow = WIN_ROW_START; winRow < WIN_ROW_END; ++winRow)
               for(int winCol = WIN_COL_START; winCol < WIN_COL_END; ++winCol)
                        ++cumulativeCount[size_t(IN_IMG(winRow, winCol, chan))];

            //makes cumulativeCount actually a cumulative count of occurrances now
            for(size_t color = 1; color < 256; ++color)
               cumulativeCount[color] += cumulativeCount[color - 1];//cumulative frequency count

            outImg(row, col, chan) = std::floor(255*cumulativeCount[size_t(IN_IMG(row, col, chan))]/AREA + .5);
         }

      }

   return outImg;
}
```

M. addNoise.cpp

```
/*****************************************************************
 * addNoise.cpp: has Gaussian Noise generator
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/5/11
 *
 *****************************************************************/

#include "Dip.h"
#include <iostream>
#include <cmath>
#include <time.h>

using namespace std;

/*
 * generates a random gauss number with mean = 0 and standard D = SD
 * @return gaussian random variable
 */
float gaussianNoise(const float& SD)
{
   float a = SD*sqrt(-2*log(rand()/float(RAND_MAX))*cos(2*PI*rand()/float(RAND_MAX)));
   while(a - a)
      a = SD*sqrt(-2*log(rand()/float(RAND_MAX))*cos(2*PI*rand()/float(RAND_MAX)));

   return a;
   //return SD*sqrt(-2*log(rand()/float(RAND_MAX))*cos(2*PI*rand()/float(RAND_MAX)));
}
```

```
/*
 * Adds noise from GaussNoise to an image.
 * @param IN_IMG: input image
 * @return gaussian-noise corrupted image
 */
Image addGaussNoise(const Image& IN_IMG, const float& SD)
{
   Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getType());

   for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
      for(int row = 0; row < IN_IMG.getRow(); ++row)
         for(int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col, chan) = IN_IMG(row, col, chan) + gaussianNoise(SD);

   return outImg;
}
```