Unit 2. Point Processes

2.1 Transformations of Gray Levels

Linear Transformations of Image Grayscales. A *linear transformation* of an image is a function that maps each pixel gray level value into another gray level at the same position according to a linear function. The input (argument) is a gray level *f = f(m,n)* at location (m,n) and the output is a new gray level *g = g(m,n)* defined at the same position (m,n). Linear mappings have the form *g(m,n) = T[f(m,n)]* such that

$$T[af(m,n) + bf(j,k)] = ag(m,n) + bg(j,k) \tag{2.1a}$$

However, affine transformations are more useful and they are just a linear transformation followed by a translation, such as the equation of a line, *y = ax + b*, where a is the *slope* and b is the *y-intercept*. These are often called linear and so we will also. For images gray level transformations these take the form

$$g(m,n) = af(m,n) + b \tag{2.1b}$$

Figure 2.1 presents a linear transformation that maps the gray levels of an input image {f(m,n)} into the gray levels of an output image {g(m,n)}. In this case the transformation dilates the input domain from a subinterval of minimum to maximum gray levels for the original image, $[f_{min}, f_{max}]$, onto the full interval $[g_{min}, g_{max}] = [0, 255]$ for the output image. This stretches the contrast to the boundaries of the grayscale.

A linear transformation of the input gray level interval $[f_{min}, f_{max}]$ onto the output interval $[g_{min}, g_{max}]$ has the form of Equation (2.1b) above, where the slope *a* is defined as usual. Thus

$$a = (g_{max} - g_{min})/(f_{max} - f_{min}) \tag{2.2}$$

$$g(m,n) = af(m,n) + b = \left[(g_{max} - g_{min})/(f_{max} - f_{min})\right]f(m,n) + b \tag{2.3}$$

at each location (m,n). When $f = f_{min}$ we desire that $g = g_{min}$, so we can substitute into Equation (2.3) the point (f_{min}, g_{min}) to solve for b via

$$b = g_{min} - \left[(g_{max} - g_{min})/(f_{max} - f_{min})\right]f_{min} = g_{min} - af_{min} \tag{2.4}$$

Upon substituting for b in Equation (2.3) and collecting terms, we obtain

$$g(m,n) = \left[(g_{max} - g_{min})/(f_{max} - f_{min})\right](f(m,n) - f_{min}) + g_{min} \tag{2.5}$$

Figure 2.1. Dilating Linear Transformation.

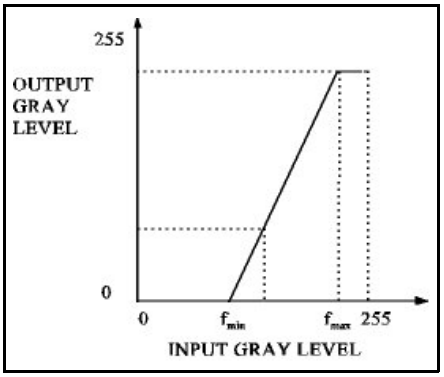
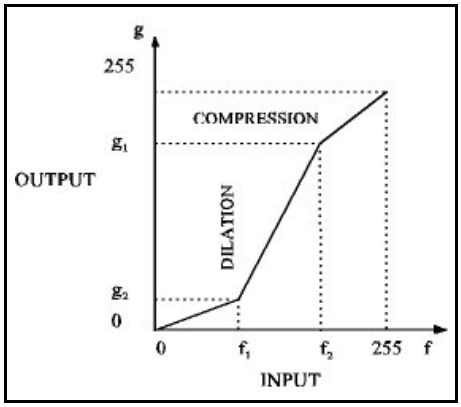


Figure 2.2. Piecewise Linear Transformation.



Equation (2.5) maps f_{min} into g_{min}, f_{max} into g_{max}, and everything else to its proportion p of the way between f_{min} and f_{max} into that proportion p of the way between $g_{min}$and g_{max}. This transformation may be used to map a smaller interval of gray values into a larger one (*contrast stretching*), but also may be used to map a larger interval into a smaller one (*contrast compression*).

A related transformation breaks the range of gray levels into subintervals and uses a linear transformation on each subinterval. Figure 2.2 shows such a *piecewise* linear transformation. The only requirement is that the conditions $0 \le f_{min} \le f_{max} \le 255$ and $0 \le g_{min} \le g_{max} \le 255$ are met. The equation for the linear mapping must be defined appropriately on each subinterval.

The algorithm given below performs linear transformations, where f_{min}, f_{max}, g_{min} and g_{max} are selected beforehand for the appropriate mapping. The input image is $\{f(m,n): 0 \le m \le M, 0 \le n \le N\}$ and the output image is $\{g(m,n): 0 \le m \le M, 0 \le n \le N\}$.

Algorithm 2.1: Linear Transformation

```
input g_max, g_min, f_max, f_min;          //Enter min. and max. values
a = (g_max - g_min)/(f_max - f_min);       //Compute slope for linear transformation
for m = 0 to M-1 do                        //For every row and for every
    for n = 0 to N-1 do                    // column,
    g[m,n] = a*(f[m,n] - f_min) + g_min;   // transform each pixel f[m,n] into g[m,n]
```

Nonlinear Transformations. We discuss three types of nonlinear transformations here, which are *logarithmic, exponential* and *sigmoid* for respectively stretching the darker, lighter and middle gray levels.

1. Logarithmic Transformations: A nonlinear transformation is usually done after a linear transformation has set the contrast and range of gray levels to that desired. It maps small equal intervals into nonequal intervals. Suppose that most of the pixels have values at the lower end of the gray scale and we want to spread them out to see the detail there, but that we don't care about the brighter values in the upper range of grays. Then we want a small input interval at the low end to map to a larger interval at the low end for the output image. We also want 0 to map to 0 and 255 to map to 255. The function

$$g(m,n) = (c)\log_2(f(m,n) + 1) \tag{2.6a}$$

spreads out the lower gray levels. It must map 0 to 0, and $(c)\log_2(1) = 0$ does this. It also must map 255 to 255, so that $255 = (c)\log_2(255 + 1) = (c)\log_2(256) = 8c$. Thus $c = 255/8 = 31.875$, so we have

$$g(m,n) = (31.875)\log_2(f(m,n) + 1) \tag{2.6b}$$

Figure 2.3 shows this type of mapping. For example, 128 maps to $31.875\log_2(128 + 1) \approx (31.875)(7.001) = 223.157$, which is truncated to the integer 223. Thus the gray levels from 0 to 128 are dilated (more strongly at the lower end). We can also use

$$g(m,n) = (c)\log_b(af(m,n) + 1) \tag{2.6c}$$

where $a > 0$ is a constant and $b > 1$ is a logarithm base.

X-ray images are known to satisfy the intensity function $f(m,n) = f_0\exp[-r(m,n)]$, where r(m,n) is the attenutation of the x-ray signal at (m,n) due to the density and thickness of the material. Therefore, we use logarithmic transformations (to the base e) to enhance the detail on x-ray images.

2. Exponential Transformations: Here we are interested in spreading out the upper gray levels at the expense of the lower gray levels, which must be contracted. Again, we want the end points to map to end points. While a logarithmic function spreads out lower levels disproportionately, an exponential spreads out the upper levels disproportionately. If we use

$$g(m,n) = \exp[af(m,n)] - 1 \tag{2.7a}$$

then 0 maps into exp(0) - 1 = 0. To force 255 to map into 255, we ust have that 255 = exp[a(255)] - 1, so that 256 = exp(255a). Upon taking the natural logarithm of each side, we obtain ln(256) = 255a, or a = ln(256)/255 = 0.0217458. Then this mapping is

$$g(m,n) = \exp[0.021746f(m,n)] - 1 \qquad (2.7b)$$

Figure 2.4 shows an exponential type of transformation. As an example, 128 maps to exp(0.0217458(128)) - 1 = exp(2.78346) - 1 = 16.1749 - 1 = 15.1749, which is truncated to 15. Thus the gray level is strongly contracted on the lower half but stretched on the upper half.

Figure 2.3. Logarithmic Mapping.

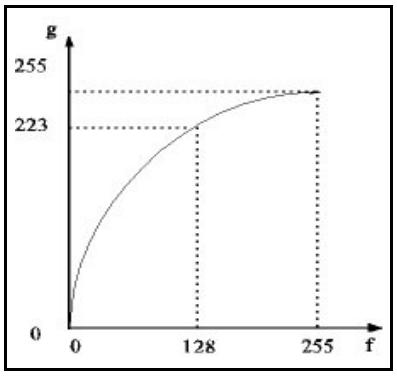
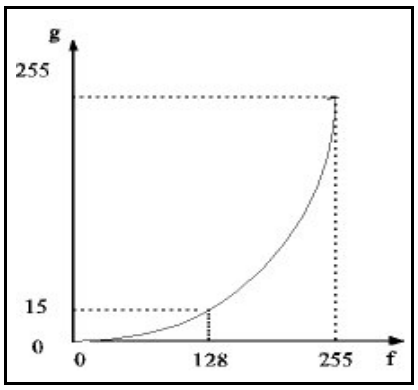


Figure 2.4. Exponential Mapping.

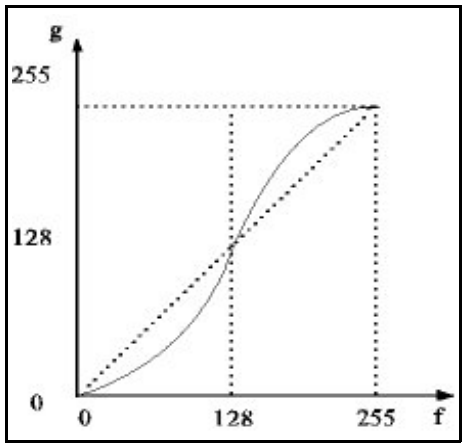


3. Sigmoid Transformations: To stretch the middle gray levels we use a sigmoid function as shown in Figure 2.5, where the gray levels are standardized ($0 \le f(m,n) \le 1$) by dividing the f(m,n) by 255. Thus we must scale the outputs by 255.

$$g(m,n) = 255 / \{1.0 + \exp[-a(f(m,n) - b)]\}, \qquad 0 < f(m,n) \le 1 \qquad (2.8)$$

where $b = 128$ is the usual value and a is the rate that defines the steepness of the curve ($a = 1$ is mild whereas $a = 3.8$ is rather steep).

Figure 2.5. A sigmoid function.



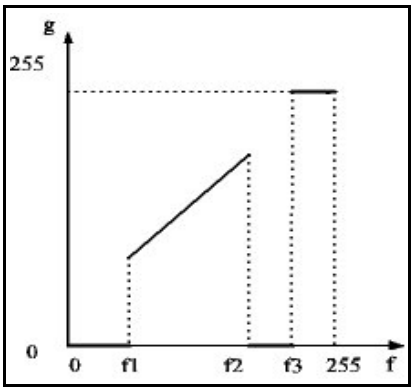
The parameter a is the gain that determines how sharp the curve is (the default value is $a = 1$ but values of 0.5 to 2.0 are allowable). The parameter b centers the location of the maximum stretch and has default $b = 128$, but it could be set lower or higher depending on where the most stretching is desired. This function maps the endpoints to endpoints while stretching the middle gray levels and trading this off by compressing the lowest and highest gray levels.

The algorithm given below uses a predefined point transformation T that is a function on the gray levels to new gray levels. Any parameters that T uses must be input by the user before T can be defined.

Algorithm 2.2 Basic Pixel Transformation:

```
for m = 0 to M-1 do                    //For each row and
    for n= 0 to N-1 do                 // for each column
        g(m,n) = T(f[m,n]);            // apply transformation T to pixel value
```

Figure 2.6. Using threshold mappings.



Thresholding. One or more thresholds can be used to transform an image pointwise. Let f_1, f_2 and f_3 be thresholds. Suppose, for example, that we want to map all gray levels below f_1 into black (0), those between f_1 and f_2 into a larger range and everything betweem f_2 and f_3 into black. Let us map everything above f_3 into white. Figure 2.6 shows a function that performs this transformation. An unlimited variety of maps can be constructed in a similar manner. Note that the gray levels between f_1 and f_2 are dilated, while all other data are severely compressed. The use of thresholds is a powerful tool for exposing certain details in an image.

Segmentation with Thresholds. Sometimes it is desired to convert an image to a set of fewer gray levels so that certain regions will have the same gray level. Such a process is called segmentation because it partitions the image into a number of segments. In the following algorithm it is assumed that we have input the values for the K+1 threshold gray levels $T_0,...T_K$.

Algorithm 2.3 Basic Segmentation:

```
for m = 0 to M-1 do                          //For each row and
    for n= 0 to N-1 do                       // for each column
        for k=0 to K-1  do                   // and for each threshold range
            if (f(m,n) ≥ Tk) AND (f(m,n) <  Tk+1) then   // if graylevel is in segment range
                g(m,n) = Tk;                 // then put it at segment gray level
```

2.2 Image Histograms

Distribution of Gray Levels. An image as an array of discrete values $\{f(m,n): 0 \le m \le M-1, 0 \le n \le N-1\}$, where for any row m and column n, the *pixel position* (m,n) has *pixel value* f(m,n). Figure 2.7 presents the array of pixel locations in the dimensions x and y. The pixel gray level f = f(m,n) can be considered to be the height in a third dimension above the point (m,n) in the xy-plane. The L integer values that f(m,n) can assume represent gray levels from 0 (black) to L-1 (white). L is most often 256 (which we use henceforth).

Figure 2.8 shows a 3-dimensional graph of an image of the "+" symbol according to our model. The background is black, or 0 level, while the symbol is white, or 255. In this simple case, the proportions of pixels at the gray levels is zero except at gray level 0 and gray level 255. Let P be the number of pixels at gray level 0 and Q be the number of pixels at gray level 255, so that P+Q = MN is the total number of pixels. Thus the distribution of proportions p_k of pixels at the various gray levels k, $0 \le k \le L-1 = 255$ is

$$p_0 = P/(MN); \quad p_k = 0, \quad 1 \le k \le 254; \quad p_{255} = Q/(MN)$$

Figure 2.9 shows a graph of the proportional distribution over the gray levels 0 to 255, where this small image is 16x16 = 256 pixels and we have taken P = 160 and Q = 96. In this case, the proportion of black pixels is 160/256 = 0.625 and the proportion of white pixels is 96/256 = 0.375.

Figure 2.7. An MxN image.

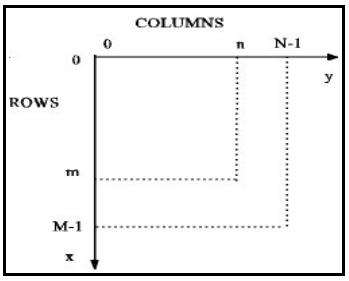


Figure 2.8. A simple image.

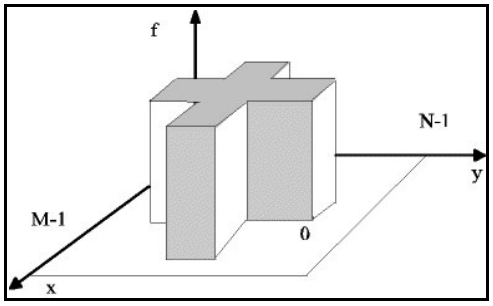
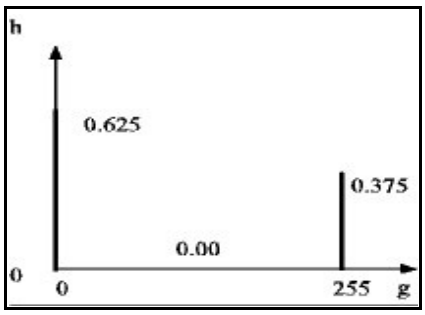


Figure 2.9. A Proportional Distribution.



Now consider the image *lena256.pgm* from Unit 1. The gray levels range from 0 to 255 and the image data size is 256x256 = 65,535 pixels. Each of the gray levels from 0 to 255 can be represented by a byte (8 bits), so the image file contains a short header and the 65,535 pixel bytes. Let us take the *counts* of the number of pixels at each of the gray levels 0,...,255 and denote these counts by c_k, for k = 0,...,255. To obtain the proportions of pixels at each gray level, we divide each gray level count c_k by the total number of pixels to obtain

$$h_k = c_k / 65{,}535, \quad k = 0,...,255 \tag{2.9}$$

We display these 256 proportions as a graph, as in Figure 2.9. Proportions are like probabilities. The probability that any randomly drawn pixel from the image of Figure 2.9 is white is 0.375.

Histograms. In general, an MxN image has MN pixels. We define the *counts* of the number of pixels at gray level k as c_k, k = 0,...,L-1. The proportion $h_k = p_k$ at gray level k is

$$h_k = c_k / (MN), \; k = 0,...,L-1 \tag{2.10a}$$

where $\{h_k\}$ is the *histogram*, or graph of proportions of each gray level k over the image. The proportions of gray levels sum to unity.

$$\Sigma_{(k=1,MN)}\, h_k = (1/(MN)\Sigma_{(k=1,MN)}\, c_k = MN/MN = 1 \tag{2.10b}$$

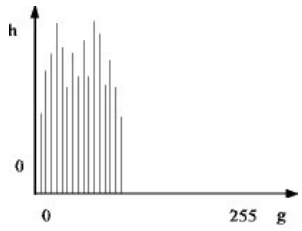
A histogram is a summary that we can view to see some basic characteristics of the image. Sometimes it is convenient to average every two consecutive proportions and display them, so that there are 128 (L/2) rather than 256 proportions. A graph of proportions computed for each set of r consecutive gray levels at a time is called a *histogram* with *bin size* r. With a bin size of 2 there are 128 proportions.

If the nonzero proportions are over a small band of gray levels, then the gray levels are few and close together so that there is little contrast and it will be difficult or impossible to see all features in the image. If the histogram is spread out across all gray levels with approximately equal heights (a rather uniform distribution) then the image will not only have good contrast but will represent all gray levels approximately the same and will expose details that may otherwise be hiding in certain bands of gray levels. Thus we should spread out the pixel distribution more uniformly.

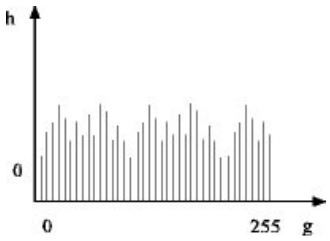
Figure 2.10 shows the approximate histograms for four images. Part (a) has low contrast due to a narrow range of gray levels and is too dark, while the second, Part (b), has good contrast because of a wide range of shades of gray from very dark to very light. Part (c) has low contrast and is too bright. Part (d) is too low in contrast and is devoid of darks and lights, that is, it contains only middle grays.

Figure 2.10. Four Approximate Histograms..

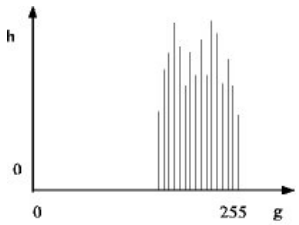
(a) Low Contrast Dark Image



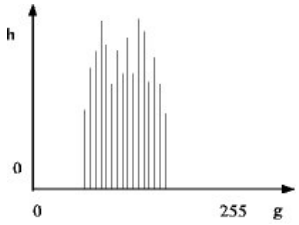
(b) High Contrast Image



(c) Low Contrast Bright Image



(d) Low Contrast Image with Middle Grays



If all pixels of an image were at a single gray level, then the image would be uniformly gray at that shade of gray but it would contain no information. If the pixel values were distributed randomly over all shades of gray, that is, from 0 to 255, then there would also be no information. An arrangement of gray levels that captures a scene of objects contains information about that scene. To the extent that there are too few grays, or there is random error on too many pixels, called *noise*, that information is degraded (there are also other kinds of degradation such as motion of the camera or objects, or an unfocused lens).

A Histogram Algorithm. We present an efficient algorithm here for computing the histogram of an image. The algorithm is described in a high level pseudo language that is easily translated into C or C++.

Algorithm 2.4: Computation of Histogram

```
for k = 0 to 255 do                    //Initialize all counts
   c[k] = 0;                           //c[k] = count of pixels at gray level k
for m = 0 to M-1 do                    //For each row and for each
   for n = 0 to N-1 do                 //column in the image
      c[f[m,n]] = c[f[m,n]] + 1;       //increment count at gray level f(m,n)
for k = 0 to 255 do                    //Proportionalize each gray level count
   h[k] = c[k]/(M*N);                  //M*N = total pixel count, h[k] is proportion
```

Average Image Gray Level. Each image has an average gray level μ and a variance σ^2 computed from

$$\mu = \Sigma_{(k=0,255)} (k)(h_k) \tag{2.11}$$

$$\sigma^2 = \Sigma_{(k=0,255)} (k-\mu)^2(h_k) \tag{2.12}$$

Without *apriori* information we would expect that μ is approximately L/2 = 256/2 = 128. However, many important details may be in the lower range (or in the upper range) of gray levels. In that case we want more pixels to be distributed over the gray level range of interest. It is sometimes useful to use the average μ over the entire image, or over a portion of interest, in a process that changes μ to a desired value. The gray level variation can also be increased to increase contrast.

Using a Tool for Histograms. It is almost always useful to look at the histogram of an image to see what may need to be done to enhance it. Here we use the tool *Matlab*.

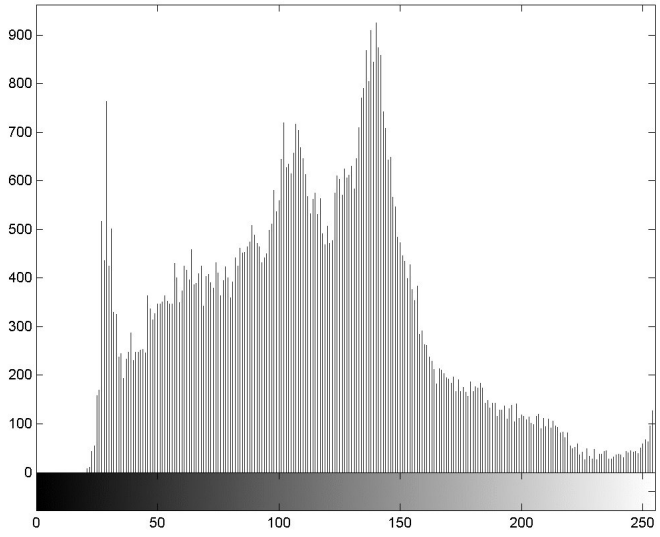
1. *Matlab* Histograms: Run Matlab, click inside of the Command Window to bring that window into focus and then enter the following commands.

```
>> Im1 = imread('building.tif');        //load image into memory at Im1
>> imshow(Im1);                         //show image on screen
>> figure, imhist(Im1);                 //make and show histogram as new figure
```

Figure 2.11 shows the original *building.tif* image and Figure 2.12 shows its histogram. The histogram graph can be saved in the same way as any image is saved. To exit an image display, click on *File* and then on *Exit* in the top bar of the image frame. To exit *Matlab*, click on *File* and then *Exit* on the top menu bar of the main window.

Figure 11. Original *building*.

Figure 12. Histogram of original *building*.



2.3 Histogram Equalization

Cumulative Distributions. A histogram $\{h_k\}$ for an image may have its nonzero proportions predominately in the lower, upper or middle part of the grayscale. Ideally, the image grays should cover the range [0,L-1] and not have too many or too few counts in any gray levels. A transformation that spreads out the gray levels used and also changes the proportions to be more uniform is called *histogram equalization*. Figure 2.10b shows an approximately equalized histogram.

Consider the distribution of gray levels in Figure 2.11 (shown as a continuous function for convenience rather than as a discrete one). The area under the curve is unity as it represents the total proportions over all gray levels designated here by f. This is the same as a *probability density function* (pdf) $h_F(f)$ for the random variable F that assumes grayscale values f. The total probability is

$$\int_0^{256} h_F(f)df = 1 \qquad (2.13)$$

For the purposes of this discussion, f is a continuous grayscale variable with normalized domain of [0,1]. The accumulated probability at any grayscale value f (summed over a dummy grayscale variable *r*) is

$$g = H_F(f) = \int_0^f h_F(r)dr \tag{2.14}$$

The function $H_F(f)$ that accumulates area up to each point f is called the *cumulative distribution function* (cdf) for the pdf $h_F(f)$. Figure 2.12 shows the cdf $H_F(f)$ for the pdf of Figure 2.11. Cdf's are monotonic and an assumption of strict monotonicity implies that they are one-to-one and have values between 0 and 1. We now use this function $H_F(f)$ as a nonlinear transformation function on the gray levels. The objective of such a transformation is to obtain transformed gray levels $g = H_F(f)$ that are uniformly distributed across the grayscale, that is, the pdf $h_G(g)$ is a constant over all g.

We now show that the transformation $g = H_F(f)$ can be used as a transformation that maps the gray levels f to gray levels g that are uniformly distributed. The subscripts on the pdf's indicate what random variables they describe, F or G. From probability theory for the transformation of random variables F → G, we have that

$$g = H_F(f) \tag{2.15}$$

$$h_G(g) = [d/dg]H_F(f) = h_F(f)[df/dg] \tag{2.16}$$

The inverse transformation exists as an inverse function whenever $H_F(f)$ is strictly increasing and one-to-one and onto [0,1].

$$H_F^{-1}(g) = H_F^{-1}(H_F(f)) = f \tag{2.17}$$

Figure 2.14. A Density Function.

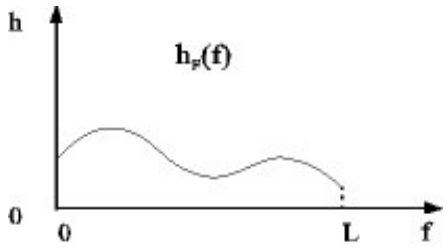
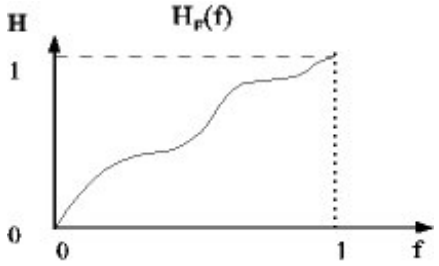


Figure 2.15. A Cumulative Distribution.



Upon differentiating g with respect to f we obtain

$$dg/df = dH_F(f)/df = h_F(f) \tag{2.18}$$

By the one-to-one property and Equation (2.18) where $h_F(f) \neq 0$, we have

$$df/dg = 1/(dg/df) = 1/h_F(f) \tag{2.19}$$

Upon substituting Equation (2.19) into Equation (2.16), we obtain

$$h_G(g) = h_F(f)[1/h_F(f)] = 1, \quad 0 \le f \le 1 \qquad (1/h_F(f) \neq 0) \tag{2.20}$$

so that the pdf $h_G(g)$ (or histogram) of g is the constant 1 on the normalized gray levels [0,1]. Thus we have proved the following theorem: if the cdf of a histogram $\{h_k\}$ for an image is used as a nonlinear transformation on the gray levels f of that image, then the resulting transformed image is uniformly distributed. The implementation is done via $f \rightarrow g = H_F(f)$, which is the cdf of the random variable F.

While this is true in the continuous case, it is only approximate in the discrete case. However, it does tend to spread out the gray levels and approximately equalize the proportions at the various gray levels. We have shown that $g = H_F(f)$ is a grayscale transformation $f \rightarrow g$ on [0,1] to [0,1] such that g has a uniform distribution. Because g satisfies $0 \leq g \leq 1$, we need to multiply it by 255 to obtain the scaled gray levels, so that $0 \leq (255g) \leq 255$.

The Equalizing Transformation. Let $\{f_k: k = 1,...,L-1\}$ be the set of gray levels with histogram proportions $\{h_k: k = 0,...,L-1\}$. Then the gray level transformation is given by

$$g_k = \Sigma_{(j=0,k)}\, h_j = H_F(k) \quad \text{(for each gray level k)} \tag{2.21}$$

where the summation replaces the integral of Equation (2.13). An algorithm for histogram equalization is given below for the case when the histogram has already been computed.

Algorithm 2.4: Histogram Equalization:

```
sum ← 0.0;                          //Initialize sum to zero
for k = 0 to 255 do                 //For each gray level
   sum ← sum + h[k];                //sum histogram proportions
   H[k] ← sum;                      //Collect cumulative values
for m = 0 to M-1 do                 /For each row and each column pixel
   for n = 0 to N-1 do              //position, compute new gray level g
     g[m,n] ← 255*H[f[m,n]];        //Compute and scale g, 0 ≤ g ≤ 1
```

The transformed gray levels g[m,n] are approximately uniformly distributed across the grayscale (and are more uniform when the cdf is more one-to-one).

Histogram Equalization Using Software Tools. It is easy to use the histogram function of the tool we have available. We use *Matlab*. Histogram does not always make the image look better, but often does.

1. Histogram Equalization with *Matlab*: The function *imhist*(Im) makes and displays a histogram for the image represented by the variable *Im1* that has been read into memory with *imread*('building.tif'), for example. The *Matlab* function (Im2 =) *histeq*(Im1) makes a histogram of the image in memory at Im1 but does not show it on the screen so we must use *imshow*(Im2) to display it. The following command script shows the steps to equalize and show the new histogram of *building.tif*.

```
>> Im1 = imread('building.tif');    //load image into memory at Im1
>> imshow(Im1);                     //show image on screen (Fig. 16)
>> figure, imhist(Im1);             //make and show histogram of image (Fig. 17)
>> Im2 = histeq(Im1);               //equal. hist. of Im1, put new image at Im2
>> figure, imshow(Im2);             //show new image as separate figure (Fig. 18)
>> figure, imhist(Im2);             //make/show histogram of new image (Fig. 19)
```

Figure 16. Original building.

Figure 17. Histogram of Original building.

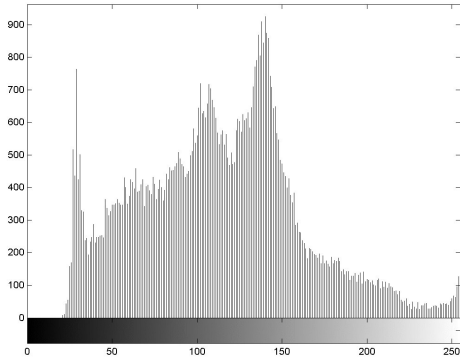
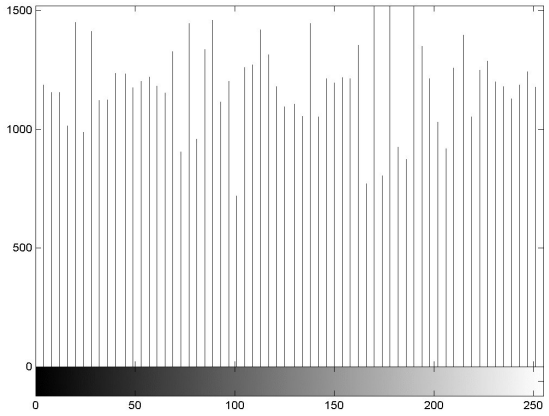


Figure 18. Image with equalized histogram.

Figure 19. Equalized histogram.



Figures 16 and 17 show respectively the original image and its histogram, while Figures 18 and 19 show respectively the image and histogram after histogram equalization.

Multi-Level Histogram Equalization. Histogram equalization often yields too many dark pixels and too many bright pixels. To better equalize an image we partition it into two sets of gray levels according to a threshold T, below which 50% of the pixels fall (and 50% above). Let D and B be the sets of dark and bright pixels, respectively. The percentages of D and B do not necessarily need to be equal, but without prior knowledge, it is a good value for T, which can be found by counting up to MN/2 and finding the closest gray level T so that approximately 50% of the pixels are below T. D and B are found by

$$D = \{p(m,n): p(m,n) \le T\}, \quad B = \{p(m,n): p(m,n) > T\}$$

Now we equalize the histogram of only those pixels in the interval [0, T] and then equalize the histogram of B on the interval [T+1, 255], writing both sets of pixels to their locations in the output image. The results can be striking, depending upon the image and its original gray level distribution.

Fig. 2.20. The original image.

Fig. 2.21. Hist. equal.

Fig, 2.22. Dual hist. equal.

Figures 2.20, 2.21 and 2.22 respectively show the original, the histogram equalized and the dual level histogram equalization images. In this case the original histogram was fairly well balanced and the histogram equalization made it too dark (histogram equalization does not always improve an image). Here the dual level histogram equalization gave a better balance with more darks, lights and middle grays than did the regular histogram equalization.

2.4 Statistical Techniques

Statistics-based Linear Transformations. Let μ be the average pixel gray level over an image $\{f(m,n)\}$ and let σ^2 be the variance. If the overall image is too dark or too light, we may choose a desired mean μ_d. If the contrast is too low or too high, we may choose a desired variance σ_d^2, where either $\sigma_d > \sigma$ or $\sigma_d < \sigma$. We compute the *slope* (*gain*) for a linear transformation $g = af + b$ via

$$a = \sigma_d/\sigma \tag{2.22}$$

We can now compute the *vertical axis-intercept*, or *bias*, *b* and find the expected value E[b] via

$$b = g - af = g - (\sigma_d/\sigma)f$$

$$E[b] = E[g - (\sigma_d/\sigma)f] = E[g] - (\sigma_d/\sigma)E[f]$$

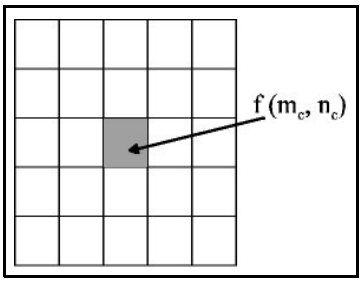
$$b = \mu_d - a\mu \tag{2.23}$$

where μ is the actual mean. The following linear transformation is called the *statistics-based* linear map and can easily be worked into Algorithm 2.2 where $T(f(m,n)) = g(m,n)$ is defined by

$$g(m,n) = af + b = [\sigma_d/\sigma]f(m,n) + (\mu_d - [\sigma_d/\sigma]\mu) \tag{2.24}$$

Statistical Differencing. For the next method, we need the concept of a neighborhood of a pixel. This is actually an area process rather than a point process, but is strongly related to the point process above. For our purposes here, a pxq *neighborhood* of a pixel is a rectangular array of pixels of p rows and q columns, where p and q are odd integers and p_c is the center pixel. Thus we could write it as a (2r+1)x(2s+1) neighborhood of the center pixel. Figure 2.17 shows a 5x5-neighborhood of a pixel at position (m,n).

Figure 2.23. 5x5 Nbhds.



Statistical differencing is a local adjustment method that tends to produce a similar contrast throughout the image and the degree of contrast is user selectable. It uses statistics over a large neighborhood, say 11x11 to 31x31, to adjust the center pixel. The statistics μ and σ are computed over the neighborhood pixels and applied to the center pixel value $f(m_c,n_c)$ via

$$g(m_c,n_c) = \mu + ß[f(m_c,n_c) - \mu] \tag{2.25}$$

where $ß = \sigma_d/\sigma$ and σ_d is the desired parameter. We select the variance to satisfy either of $\sigma_d > \sigma$ or $\sigma_d < \sigma$ to respectively stretch (dilate) or compress (contract) the output grayscale.

Adapting Gray Levels with Statistical Methods. The user control of this method can be improved by use of a desired mean parameter μ_d per

$$g(m_c,n_c) = \alpha\mu_d + (1-\alpha)\mu + ß[f(m_c,n_c) - \mu] \tag{2.26}$$

where $0 < \alpha < 1$. This gives the user the ability to adjust the average levels μ on the neighborhoods up or down. Another adjustment is needed to prevent ß from being too large when σ is too close to zero over a neighborhood of similar values. We therefore use $ß_0$ in place of ß, where

$$ß_0 = r\sigma_d/(\sigma_d+r\sigma) \tag{2.27}$$

When $\sigma = 0$, $ß_0 = r$. By choosing r such that $r > 1$ or $r < 1$ (to move $g(m_c,n_c)$ farther from, or closer to $f(m_c,n_c)$), we may dilate or contract the difference $\delta = f(m,n) - \mu$. The computation required is much greater than for the statistics-based global adjustment because the neighborhood statistics must be computed for

every pixel in the image. When a pixel is near the boundary, we use only those neighborhood and mask entries that intersect the image. Algorithm 2.1 may be modified to use Equation (2.25) per

$$g(m_c,n_c) = \alpha\mu_d + (1-\alpha)\mu + [r\sigma_d/(\sigma_d+r\sigma)][f(m_c,n_c) - \mu] \quad (2.28)$$

where the μ and σ are computed over a large neighborhood. Some rather dramatic enhancement effects can be achieved by the use of statistical differencing in the form of Equation (2.27).

2.5 Exercises

2.1 Write the function for a linear transformation that maps [0, 255] into [64, 128]. What will this transformation do to the histograms of Figure 2.10 (approximately)? What are the slopes and vertical axis-intercept? Write the function for a piecewise linear transformation that maps [64, 128] into [0, 255] also.

2.2 Make a general statement of the effects any piecewise linear transformation will have on the histogram? What about the proportions on subintervals - do they change? Explain. Can a logarithmic type of transformation be approximated with a piecewise linear transformation? Explain with graph drawings.

2.3 Suppose that an image had a histogram that showed a high proportion of pixels in the middle gray range, and that here is where it is suspected that a lot of detail lies. Design a nonlinear transformation that compresses the lowest and highest gray levels and stretches the middle grays. Give the equation for this transformation.

2.4 How can the transformation of Exercise 2.1 be implemented in *Matlab*? What about the transformation in Exercise 2.3.

2.5 Design a linear transformation of gray levels that stretches out the darkest and lightest shades of gray but compresses the gray levels between 80 and 180. Show the equations for T.

2.6 Graph the linear transformation with slope of -1 that maps 0 to 255 and 255 to 0. What does this transformation do to an image? Implement this using *Matlab*.

2.7 Find a way to use *Matlab* to make an image into a negative image via the mapping f = 1.0 - f (recall Matlab standardizes gray level to values between 0 and 1). Use the help menu for functions for image processing and look for subtraction of images under the *Image Processing Toolbox* item).

2.8 Write a high level (pseudo-code - see the histogram algorithm for a model) algorithm to map an image via a sigmoid function.

2.9 Write a pseudo-code algorithm to read a PGM P5 image file and write a new P5 image file that is the negative of the original image that has the middle gray levels stretched.

2.10 Compile the program of Appendix 2.A and build in the sigmoid nonlinear transformation. Run this on the image *building.pgm* (P5) and write the results to a new P5 image file. Study the results and compare with the original. Now change the parameters *a* and *b* one at a time in a nonnegligible manner. Describe the effects of increasing/decreasing these.

Appendix 2.A - C Programs for Point Processes

PGM (P5) Image Files. A P2 file saves binary characters in ASCII, that is, the pixel value 234 would be saved with 3 bytes respectively for the digits 2, 3 and 4 (00000010, 00000011 and 00000100). We will read and write in binary integers where 234 is encoded as the single byte 11101010, which is the P5 format for PGM. When we look at the file with an editor, the numbers are interpreted as characters. Thus the following is what *building.pgm* looks like from the top.

P5
 # CREATOR: XV Version 3.10a Rev: 12/29/94 (PNG patch 1.2)
320 240 255
mklmmll)##)mjc\TaimnljjlmgaZVadehjigge_ZYdedegfffedb[S\bcddefffibXR_efc_SE@AFRj,Œ...

The dots at the end of the fourth line indicate that the data continues to the end of the file. This image is 320x240 (columns by rows), which gives 76,800 pixels of data in addition to the small header. We list a simple C program that can be compiled with MS Visual C++ (console mode), Borland C++ Builder, or under Gnu C or C++ in Linux or standard C or C++ in Solaris or UNIX.

The algorithm given below uses the main() function to call various functions to perform the parts needed to read a PGM image file and implement a pointwise transformation on an image and write the results to an output image PGM file. The main function displays a heading on the screen (console, or command line, mode), opens the input and output files with the user-given file names, reads the input file header and writes it to the output file, applies the transformation and then, if the operating system is *Linux, Solaris, Iris*, or any other type *UNIX* with *XView*, makes a system call to *XView* to display the before and after images..

The C Code for Point Processing.

```
// DIP1 - POINT PROCESSES Digital Image Processing Program
//          Linux Version -- Gnu C or C++ Compiler (gcc)
//            UNIX Version with C or C++ Compiler (cc)
//   This can be compiled with MS Visual C++, Console Mode, but
//          certain lines indicated below must be taken out
//=========================================================
//       This program reads an input PGM P5 image, implements a
//   user supplied point transformation and writes the result to a PGM
//   P5 type image file.
//=========================================================
//                          Version: 29June2002
//=========================================================
//       The PGM file may be of size up to N = 4096 columns and M
//   rows where M can be up to 4096 or even greater.
//---------------------------------------------------------------------------------
//       The user is asked to type in the input image file name and a name
// for  the output image file, which will also be in PGM P5 format.
//
//---------------------P R O G R A M Modular Layout-------------------------
//  main();
//     heading();                  display program heading
//     openfiles();                opens input & output image files
//     readhdr();                  reads input image file header
//     getrow();                   reads rows from input image file
//     applytransformation();      does transformation on input image
//     writefile();                writes processed row to output image file
//     closefiles();               closes input and processed output image files
//     display();                  displays before/after  images in XV (not used here)
//---------------------------------------------------------------------------------
```

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
//-------------------------------------------------------------------------------------
void    heading(void);
void    openfiles(void);
void    readhdr(void);
void    applytransformation(void);
void    display(void);              // take out for MS Visual C++
//-------------------------------------------------------------------------------------
int     fin[4096];                  //input pixels in a row
int     gout[4096];                 //output pixels in a row
int     MRows, NCols;               //indices for rows and cols
int     rowcount;                   //count of row being processed
FILE  *infptr, *outfptr;                    //input/output file pointers
char    infile[40];                 //name of input file
char    static  outfile[40];        //names of output image file
char    key;                        //key to select instructions
char    display_File1[] = "Original_Image";    //take this out for MS Windows
char    display_File2[] = "Processed_Image";  //take this out for MS Windows
int     first_time;
//-------------------------------------------------------------------------------------

//-------------------------------------MAIN--------------------------------------------
//-------------------------------------------------------------------------------------
main()
{ void closefiles(void);
  void readhdr(void);
  char   changekey, stopkey;      //key to process another image or stop
//------------------------(Put Heading on Screen)-----------------------------------
  do
  { heading();
    first_time = 1;
    do
    {  //-----------------------------(Open I/O Image Files)-----------------------------
       openfiles();
       //----------------------------(Read Input File Header)--------------------------
       readhdr();

     //-----------------------(Do Transformations on Row Pixels)--------------------
      applytransformation();
      //----------------------(Display Image with XV Program)----------------------
      display();                        //take out for MS Visual C++
      //----------------------(Select to Change this Image or Not)--------------------
      do
      { printf("\n Process this image again with new parameters (y/n): ?");
        scanf("%1s",&changekey);
        if ((changekey=='y')||(changekey=='Y')) first_time = 1;
      } while ((changekey!='y') && (changekey!='Y') &&
              (changekey!='n') && (changekey!='N'));
    } while ((changekey=='y') || (changekey=='Y'));
    //--------------------------(Close Any Open Image Files)-------------------------
    closefiles();
```

```
   //-------------------------(Select Stop or Process an Image)----------------
   do
   { printf("\n Enter <s> to stop or <i> to process another image ");
     scanf("%1s",&stopkey);
     if ((stopkey=='i')||(stopkey=='I')) first_time = 1;
   } while ((stopkey != 's') && (stopkey != 'S') &&
           (stopkey != 'i') && (stopkey != 'I'));
   //------------------------------------------------------------------------------------
  } while ((stopkey != 's') && (stopkey != 'S'));
  printf("\n Bye! Bye!\n");
  return;
} //end main()
//-----------------------------------------------------------------------------------------
//---------------------------------HEADING-------------------------------------------
//-----------------------------------------------------------------------------------------
void heading()
{ int      i;
  for (i=0;i<16;i++) printf("                                    +\n");
  printf("       DIP1 - POINT TRANSFORMATION of DIGITAL IMAGE\n");
  printf("                          by Prof. Carl G. Looney\n");
  printf("          Computer Science and Engineering Department/171\n");
  printf("                         UNIVERSITY OF NEVADA\n");
  printf("                              Reno, NV 89557\n");
  printf("                            looney@cs.unr.edu\n");
  printf("                             Updated: 22Jun2002\n");
  for (i=0;i<4;i++)  printf("                                  +\n");
  do
  {  printf("\n Enter <c> to continue of <x> to exit: ");
    scanf("%1s",&key); printf("\n");
  } while ((key != 'c') && (key != 'C') && (key != 'x') && (key != 'X'));
 if ((key == 'x') || (key == 'X')) exit();
 return;
}//end heading()
//-------------------------------------------------------------------------------------------
//---------------------------------------OPENFILES-----------------------------------
//-------------------------------------------------------------------------------------------
void openfiles(void)
{
  if (first_time == 1)
  { printf("\n                OPEN an image file\n");
    printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
    printf(" Enter name of *.pgm INPUT image file: ? ");
    scanf("%s",&infile); printf("\n");
    printf(" Enter name of *.pgm OUTPUT image file: ? ");
    scanf("%s",&outfile); printf("\n");
  }
  if ((infptr = fopen(infile, "r")) == NULL)
  { printf(" Can NOT open input image file: <%s>\n",infile);
    printf(" Exiting program..... "); exit(1);
  }
  else  printf(" Input file <%s> opened sucessfully\n\n",infile);
  if ((outfptr = fopen(outfile,"w")) == NULL)
  { printf(" Can NOT open output image file <%s>\n\n",outfile);
    printf(" Exiting program....."); exit(1);
  }
```

```
  else printf(" Output file <%s> is opened sucessfully\n\n",outfile);
  return;
}//end openfiles()

//-------------------------------------------------------------------------------------
//----------------------APPLYTRANSFORMATION-----------------------------

//---------------------------Put your function here---------------------------------
void applytransformation()
{
  void    getrow(void);
  void    writefile(void);
  void    closefiles(void);
  rowcount = 0;
  do
  { //-----------------------[Read an Image Row to Process]-------------------------
    getrow();
    //-------------------[Transform  Row with Transform Map]---------------------
    for (col=0;col<NCols;NCols++)
    { gvalue = fin[col];
      if ((gvalue >= 0)    && (gvalue < 50))   gout[col] =   25;
      if ((gvalue >= 50)  && (gvalue < 100)) gout[col] =   75;
      if ((gvalue >=100)  && (gvalue < 150)) gout[col] = 125;
      if ((gvalue >=150)  && (gvalue < 200)) gout[col] = 175;
      if ((gvalue >= 200) && (gvalue < 256)) gout[col] = 205;
    }
    //--------------------[Write Processed Row to Output File]------------------
    writefile();
    rowcount++;
  } while (rowcount < MRows);
  closefiles();
  return;
}//end applytransformation()

//-------------------------------------------------------------------------------------
//-----------------------------------READHDR-------------------------------------
//-------------------------------------------------------------------------------------
void readhdr()
{
  int     i, k, Maxgrays;
  char    c, c1, buffer[128];
  //---------------------------[Read PGM File Header]--------------------------
  printf("\n\n File <%s> Header Bytes:\n",infile);
  printf("    ------------------------------------------------------------------------\n");
  k = 0;                                //k is line number of header lines
  do
  { i = 0;
    do
    { c = fgetc(infptr);                //read characters to end of header line
      buffer[i] = c; i++;               //put characters into a memory buffer
    } while (c != '\n');                //until a newline is encountered
    if (k == 0)                         //if line is the first line
    { c1 = buffer[1];                   //get second character read
      if (c1 == '5')  printf("\n File is:  <P%c>\n",c1); //print out 'P5' if so
    }
```

```
    else
    { printf(" Image NOT in P5 format!! Quitting.....\n\n");
      exit(0);
    }
    buffer[i] = '\0'; k++;              //end buffer with NULL (‘\0')
    fprintf(outfptr,"%s",buffer);  //write buffer to output file
    printf("%s",buffer);               //write buffer to screen
  } while (k < 2);                     // for first two lines - next read third header line
   fscanf(infptr,"%d %d %d",&NCols, &MRows, &Maxgrays); c = fgetc(infptr);
   fprintf(outfptr,"%d %d", NCols, Mrows); //write no. cols, rows to output file
   fprintf(outfptr,"%c %d %c",'\n', Maxgrays, '\n'); //write max. gray level to output file
   printf(" %d ",Ncols);                          //write no. cols, rows, max gray value to screen
   printf(" %d              <-----(Width & Height)\n", MRows);
   printf(" %d                 <-----(Max. Gray Level)\n\n",Maxgrays);
}//end readhdr()

//-------------------------------------------------------------------------------------------
//---------------------------------------GETROW----------------------------------------
//-------------------------------------------------------------------------------------------
void getrow()
{ int     row, col;
   unsigned char    item;                    //pixel s are read as chars
  for (col=0;col<NCols;col++)               //now read in new row
   { item = fgetc(infptr);
      fin[col] = (int) item;
   }
    printf(" . ");
}
}//end getrow()

//-------------------------------------------------------------------------------------------
//--------------------------------WRITEFILE----------------------------------------
//-------------------------------------------------------------------------------------------
void writefile()
{ int       col;
   int       pixchar;
   //----------------[write processed row to output file]------------------
   for (col=0;col<NCols;col++)
   { pixchar = gout[col];
      fprintf(outfptr,"%c", (char) pixchar);
   }
} //end writefile()

//-----------------------------------------------------------------------------
//---------------------------CLOSEFILES----------------------------------
//-----------------------------------------------------------------------------
void closefiles()
{ //----------------------------(Close Files)---------------------------------
  fclose(infptr);
  fclose(outfptr);
  return;
} //end closefiles()
```

```
//-------------------------------------------------------------------------------
//-----------DISPLAY BEFORE/AFTER IMAGES IN UNIX-----------
//--------------------(take out for MS Visual C++)-------------------------
void display()
{ char    buffer1[100], buffer2[100];
//----------------Part 1: Copy and Display Original Image-----------------
 //copy original image to display_File1 for displaying on first time
  if (first_time == 1)
  { sprintf(buffer1," %s %s %s "," cp ", infile, display_File1);
    system(buffer1);
    system("sleep 2");
    //-------------------(Display Original Image at (y,x)----------------------
    sprintf(buffer2," %s %s %s "," xv -geometry +2-2 ", display_File1,"&");
    system(buffer2);
  }
  //-------------------------------------------------------------------------------
  //------------Part 2: Copy and Display Processed Image-----------------
  sprintf(buffer1," %s %s %s "," cp ", outfile, display_File2);
  system(buffer1);
  system("sleep 2");
  //display processed image on first time this function is called
  if (first_time == 1)
  { sprintf(buffer2," %s %s %s "," xv -poll -geometry +200-200 ",
              display_File2," & ");
    system(buffer2);
    //-------------------------Turn first_time off-------------------------------
    first_time = 0;
  }
  printf("\n Move images to desired position on screen!\n");
  return;
} //end display()
//-------------------------------------------------------------------------------
```