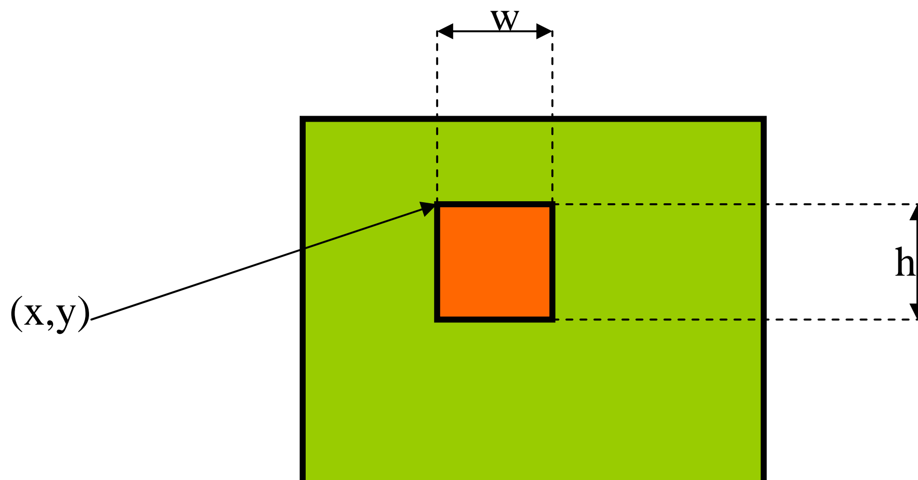
Chapter 3

Image Manipulation and Enhancement

- Image quality can be improved by simple operations on individual pixels.
- These usually include manipulating pixels data such as adding and subtracting, and changing the pixel value, e.g. brightness, color, etc.

3.1 Extracting Regions of Interest

- A region of interest (ROI) is a rectangular area within the image defined by the coordinates of the upper left corner, and its dimensions.



- Java *BufferedImage* class has three useful methods for ROI, two of which are

```
Raster getData(Rectangle rect)
void setData(Raster raster)
```

- The *Rectangle* object specifies the position and dimensions of the ROI. *GetData* returns data for ROI as a *Raster* object.

- Note that the data stored in the raster is independent of the parent image, and subsequent changes to the image does not change the raster.

- Instances of *Raster* are read only, but we can cast the returned object as *WriteableRaster* if we need to modify the pixel values. The modified raster can then be loaded back into its parent image by *setData()* method with raster as its parameter.

- If "in-place" processing of a ROI is required, the *getSubimage()* method of *BufferedImage* is more convenient:

```
BufferedImage  getSubimage(int x,  int y, int w, int h)
```

- Changes made to the su-bimage will affect the parent image. Note that unlike the above Raster method, the coordinates of the sub-image is not that of the original image, but it starts at (0,0).

Example: (a) Find the mean value of an image.

```
public static double meanValue (BufferedImage   image)        {
   Raster  raster = image.getRaster(); //assume grayscale image
   double  sum = 0.0;
   int x, y, w, h;
   h = image.getHeight();
   w = image.getWidth();
   for (y =0;  y < h; ++y)
       for (x = 0;  x < w; ++x)
         sum  += raster.getSample(x, y, 0);
   return ( sum / (h*w));                                        }
```

(b) find the mean value of the pixels in a ROI

```
public static double meanValue (BufferedImage  image,
                                               Rectangle    roi) {
return meanValue(image.getSubimage( roi.x,  roi.y,  roi.width,
                                                   roi.height)  );  }
```

- Note that in (b) we invoke *getSubimage* on the image using the ROI parameters contained in the *Rectangle* object and then pass the image that is returned to the *meanValue()* in part (a) above.

3.2 Arithmetic and Logical Operations

We have several simple arithmetic operations:

Scaling: To enlarge or shrink an image by a factor of m, we have the following code. Note that m > 1 is for enlarging and m < 1 is for shrinking.

```
public static BufferedImage scaling (BufferedImage image, int m)
{
 int  x, y, w, h;

 w = m * image.getWidth();
 h  = m * image.getHeight();
BufferedImage scaledImage = new BufferedImage (w, h,
image.getType());

 for (y = 0; y < h; ++y)
      for (x = 0; x < w; ++x)
         scaledImage.setRGB(x, y, image.getRGB (x/m, y/m));
return   scaledImage;                                                    }
```

- When the image is enlarged (m >1), then each pixel in the input image becomes an m×m block in the enlarged image, and the gray level or color of all pixels in this block become the same as the pixel in the input image.
- When the image is to be shrunk, then we sample every m-th pixel in the input image to get the value of the output pixel.

Reflection: This operation can be performed in place by reversing the ordering of pixels in the row or column. The reflection of the image f(x, y) about x = w/2 axis, where w is the width of the image, is obtained as follows:

```
public static void xReflection ( bufferedImage image){

 int  w = image.getWidth();
 int  h  = image.getHeight();

 for (int y = 0; y < h; ++y)
```

```
for (int x =0; x < w/2; ++x)                {
   // swap value at (x,y) with its mirror image
   int  value = image.getRGB(x, y);
   image.setRGB ( x, y, image.getRGB(w –x –1 , y));
   image.setRGB(w –x –1 , y, value);  }                 }
```

Rotation: This is a simple operation if the angle of rotation is a multiple of 90 degrees and the rotation is about the image center. Rotation of 90 deg or 270 deg requires creation of a new image with dimensions interchanged relative to the input image (why?). A rotation of 180 deg, can be done in place (without creation of a new image).

Addition and averaging: Like other simple operations, this operation is on pixel by pixel basis. Thus if the sizes of two images are different, then the resulting image will have the width equal to the minimum of the widths of the two images, and the same applies to the height.

- If we add two 8-bit images, the resulting pixel values could range from 0 to 510. If the resulting image is to be 8-bit then we must divide every pixel value by 2, which is an averaging operation.
- To put more emphasis on one image relative to the other we perform "alpha blending", i.e.

$$g(x,y) = \alpha f_1(x,y) + (1-\alpha) f_2(x,y) \tag{3.1}$$

where $\alpha = 0.5$ for simple averaging.

- Averaging can be used for noise reduction, if the scene is static. In this case, if several images are taken, and are then averaged, the image noise is reduced since in most cases noise has zero mean.

Subtraction: Subtracting two b-bit grayscale images can produce values between -2^b *to* $+2^b$. One way to deal with negative values is to rescale the range between 0 and $+2^{b+1}$, or between 0 and

$+2^b$. The main application of the image subtraction is for detection of changes in the scene.

AND/OR: Logical AND/OR operations are useful for masking and making composite images. This involves expressing the value in binary forms and ANDing or ORing with another image or mask. These operations are useful in bit plane slicing, and certain other image manipulation, as we will see in subsequent chapters.

Gray Level Linear Mapping (Contrast Stretching)

Let
f(x, y) be input image
g(x, y) be processed (output) image.

- Changing the pixel values of the input image to produce the output image is called gray level mapping.
-
- Linear mapping

$$g(x, y) = a\ f(x, y) + b \qquad (3.2)$$

a = slope or gain,
b = y-intercept or bias

We often want to map a certain gray level range, say $[f_1, f_2]$ onto a new range $[g_1, g_2]$

$$g(x,y) = g_1 + \frac{g_2 - g_1}{f_2 - f_1}\left(f(x,y) - f_1\right) \qquad (3.3)$$

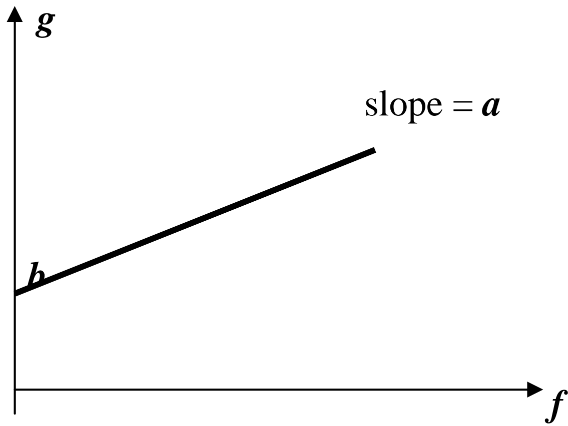


Fig. 3.1 Linear mapping

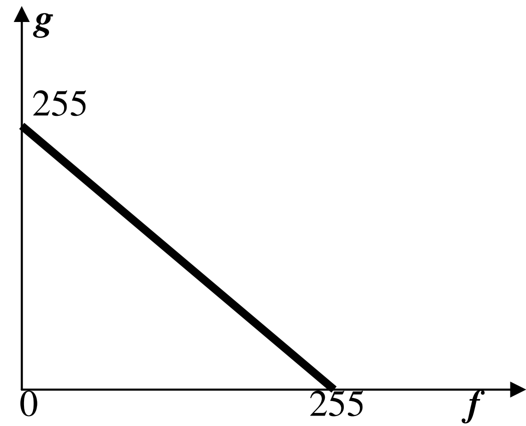


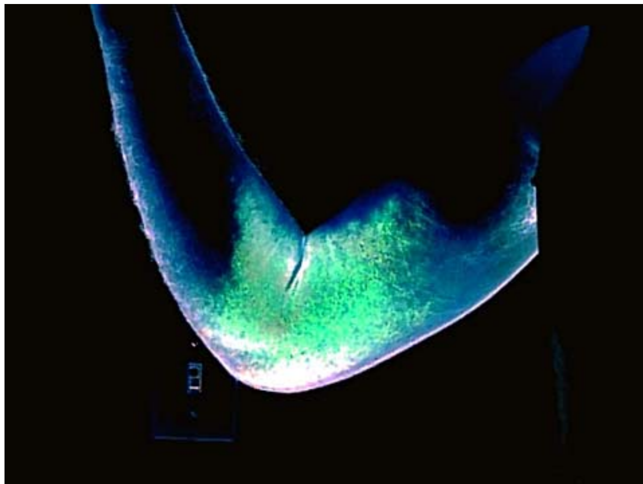
Fig. 3.2 Linear mapping to invert

Special cases of linear mapping:

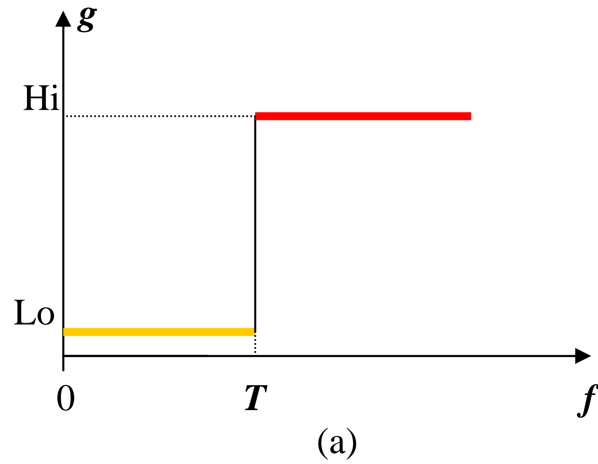
Negatization (inversion):

$\boldsymbol{a} = -1$, and bias of $\boldsymbol{b} = 255$

Inversion of Color Images: Each color band is inverted



Thresholding



(a)

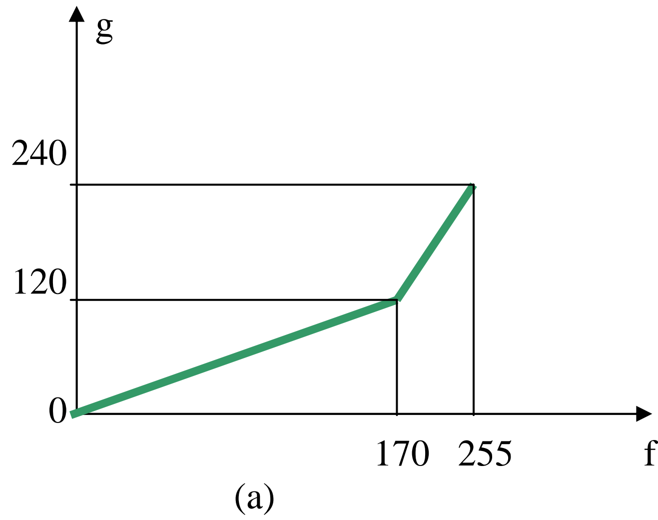
(b)

(c)

Fig 3.4 Thresholding (a) with Lo = 0, Hi = 255 and T = 127 applied to image (b) to produce image (c).

Contrast Stretching

This mapping is piecewise linear. Usually used to improved the contrast, and is usually called contrast stretching.



(a)

(b)

(c)

Fig. 3.5 The mapping (a) is applied to image (b) to produce image (c).

Implementation of linear mapping

This code must be modified if the mapping consists of several line segments.

```
public static BufferedImage linearMapping (BufferedImage image,
                                          float slope, float bias)      {
 // implement Equation (3.2)
 int  w = image.getWidth();
 int  h = image.getHeight();
 BufferedImage mappedImage =
       new BufferedImage ( w, h, BufferedImage.TYPE_BYTE_GRAY);
 WritableRaster input = image.getRaster();
 WritableRaster output = mappedImage.getRaster();
 for (int y = 0;  y < h; ++y)
   for ( int x = 0; x < w; ++x)
    output.setSample(x, y, 0, clamp (slope*input.getSample(x, y, 0) + bias));
  return  mappedImage;                                                   }

public static int Clamp (float  value)                          {
 // return 0 if value <0, or 255 if value >255
 return Math.min(Math.max(Math.round(value), 0), 255);          }
```

Java 2D API provides several built-in image processing operations, one of which is linear mapping. The operation is implemented in the form of a class called RescaleOp whose constructors are:

```
public RescaleOp ( float  gain, float  bias, RenderigHints  hints)

public RescaleOp ( float[] gain,  float[]  bias,  RenderigHints  hints)
```

The second constructor is used for color images where it requires three gains and three biases, one for each color.

Example – To brighten an image by a factor of 2:

```
RescaleOp rescale = new RescaleOp ( 2.0f, 0, null);
BufferedImage  newImage = rescale.filter(image, null);
```

If we already have an image of correct dimension and type, we can use the second parameter of RescaleOp's filter method, instead of null.

3.4 Nonlinear Mapping

- It is possible to use any (not necessarily linear) single valued mapping.
- A nonlinear mapping has the characteristic that the slope (gain) is variable, i.e. $\frac{\Delta g_1}{\Delta f_1} < \frac{\Delta g_2}{\Delta f_2}$

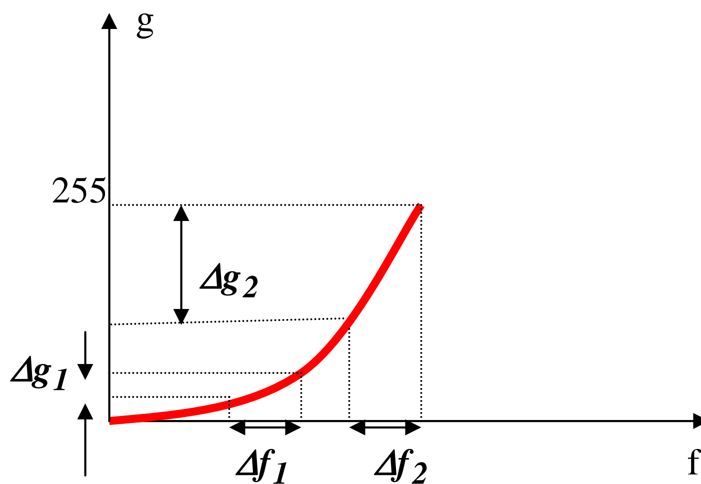
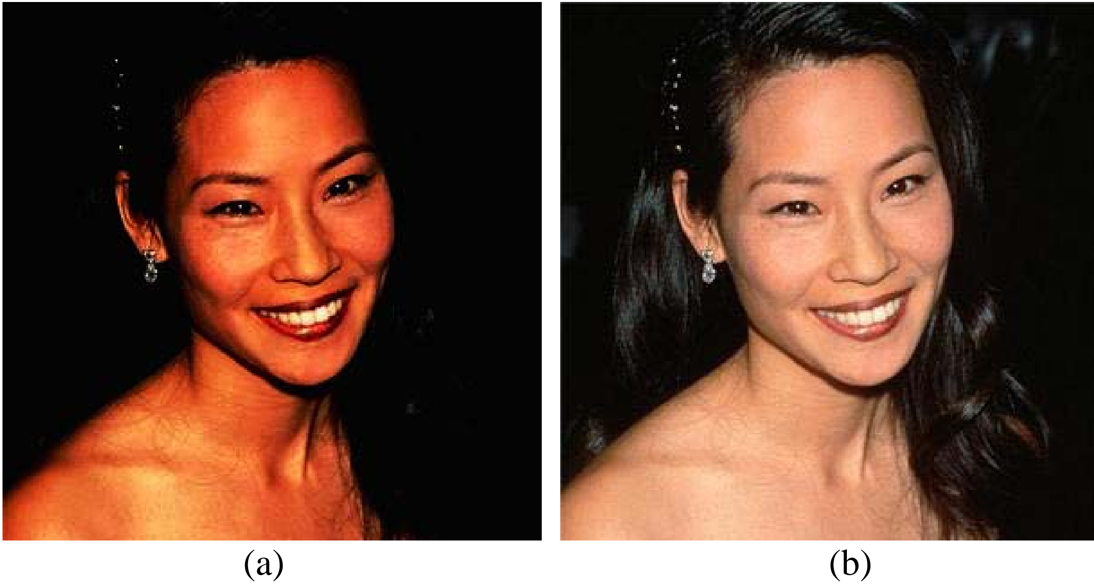


Fig. 3.6 A nonlinear mapping

High red component in Fig. 3.7 has been reduced by a nonlinear transformation $g = a\sqrt{f}$ on the red component.



(a) (b)

Fig 3.7 Reducing the red component through a nonlinear mapping

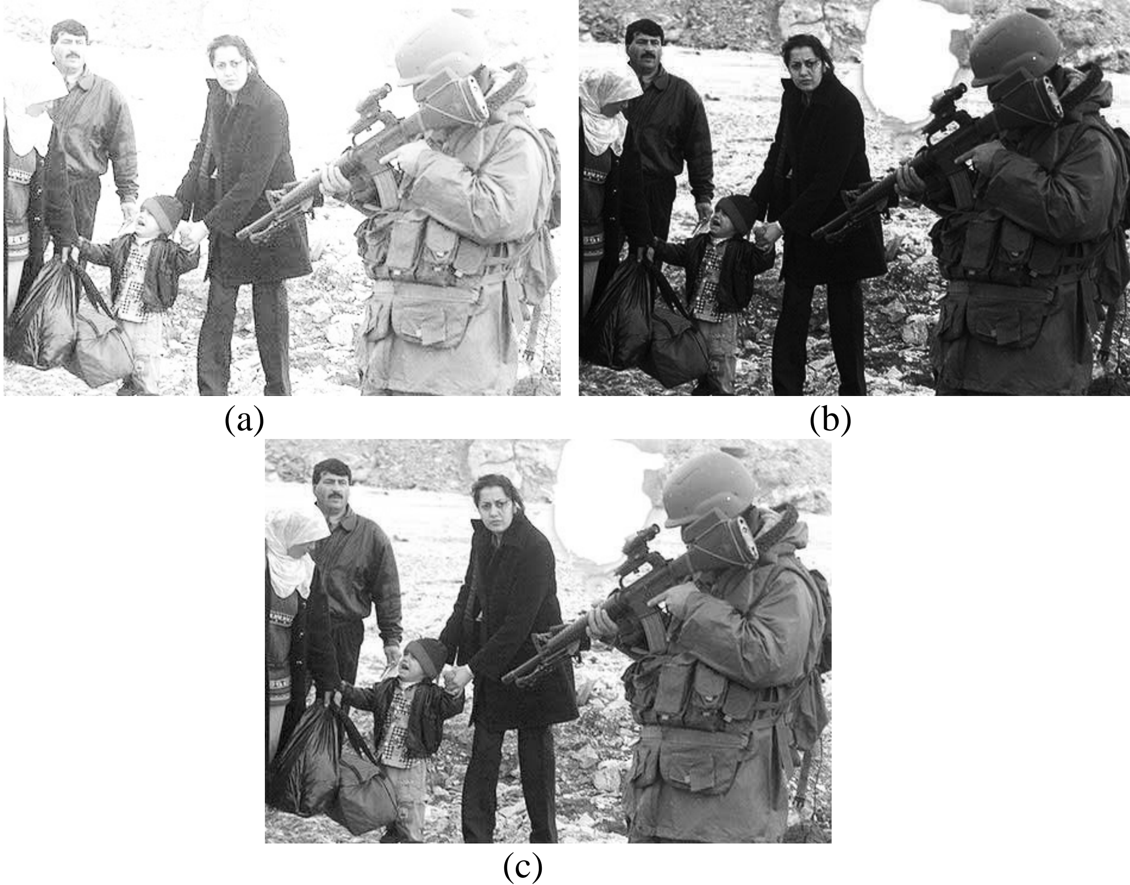
Example: Suppose that for an 8-bit gray level image we use the quadratic mapping $g = a\,f^2$ where a is a constant to be determined. Since we must have $g = 255$ for f $= 255$, then

$$g = \frac{1}{255} f^2$$

Thus a pixel value of say $f = 100$ in the input image will map into a pixel value of $g = 39$. On the other hand if we wish to have a square root mapping then we must have

$$g = \sqrt{255\ f}$$

The quadratic mapping is gnerally used to reduce contrast, whereas the square root mapping mapping usually increases the constrast. In Fig. 3.7 square root mapping is applied to the low contrast image (a) to get the right constrast iamge (c) whereas quadratic mapping is applied to (b) to get (c).



(a) (b) (c)

Fig. 3.7 showing a low constract image (a), a high contrast image (b) and right constract image (c).

Implementations of mapping

Approach One

The algorithm below shows one of two possible algorithms to implement the square root mapping, assuming a square image where w = h = n.

$\mathbf{a = \sqrt{255}}$

for (y = 0; y < n; ++y)
    for (x = 0; x < n; ++x)
        $\mathbf{g(x,y) = a\sqrt{f(x,y)}}$;

The above algorithm requires n^2 square root calculations that can be costly.

Approach Two

Use a look-up table (LUT) as follows

```
a = √255
//create an array (table) with space for 256 gray level values
for (i = 0; i < 256; ++i)
   table[i] = a√i ;
for (y = 0; y < n; ++y)
    for (x = 0; x < n; ++x)
       g(x,y) = table[f(x,y)];
```

Note that in the look-up table (LUT) implementation, we require only 256 square root evaluations, and n^2 assignments. This takes considerably less computing time than n^2 square root calculations required by the first algorithm. In general LUT is preferable if the number of gray levels is $L << n^2$ (why?)

Java2D API provides a *LookupOp* class that implements the *BufferedImageOp* interface. *LookupOp* objects are constructed as follows:

public LookupOp (LookupTable table, RenderingHints hints)

LookupTable is an abstract base class used to represent LUT's. *LookupOp* objects require an instance of one of *LookupTable*'s subclasses *ByteLookupTable* or *ShortLookupTable* from *java.awt.image* package. We pass null for the *RenderingHints* parameter.

The following code fragment shows how we can invert an image using a lookup table.

```
byte[]  table = new byte[256];
for (int i = 0; i < 256; ++i)
   table[i] = (byte)(255-i);  //1
ByteLookupTable  invertTable = new BytelookupTable(0, table);  //2
LookupOp  invertOp = newLookupOp (invertTable, null); //3
BufferedImage  invertedImage = invertOp.filter(image, null);//4
```

- Note that similar code can be used to threshold an image, apply non-linear mapping, etc. In each case:
    1- set up an array of bytes containing the LUT data,
    2- create an instance of *ByteLookupTable* from these data,
    3- create an instance of *LookupOp* using the *ByteLookupTable*
    4- invoke the filter method of *LookupOp* to process the image.
- We can simplify the application of *LookupOp* to an image if we develop a class that carries out this common set of tasks. The class can then be extended by subclasses that generate the LUT entries appropriate to particular mapping function, e.g. linear, square root, etc.

3.5 Image Histogram

The histogram of an image is the distribution of gray levels in that image.

Histogram array H[*l*], *l* = 0, 1, 2, …, L where H[*l*] represents the number of pixels that have gray level equal to *l*. For an 8-bit image there are 256 gray levels and L = 255.   Fig 3.8 shows a typical histogram, representing an image that has many dark pixels and a few bright pixels.

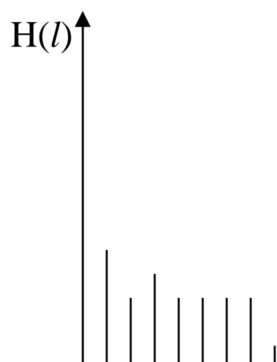


Fig. 3.8 Histogram of an image

0 1 2 … L l

- The histogram of an image provides a useful indication of the relative importance of different gray levels. Using a histogram, it is possible to determine if the brightness or contrast adjustment is necessary. For example, a low contrast image has few bright and few dark pixels.

We can normalize a histogram by dividing the values $H[l]$ by the total number of pixels in the image per gray level , i.e. by n^2/L if the image is square. The normalized histogram is

$$h[l] = \frac{1}{n^2} H[l] \qquad (3.4)$$

The normalized histogram $h[l]$ gives us the probability that a randomly chosen pixel has the gray level l.

- Gray level mapping operations affect the histogram. Multiplication of gray levels by a constant gain a will spread out the histogram if $a > 1$, or compresses it if $a < 1$.

The algorithm for determining the histogram of an image is:

```
//Create an array histogram with L+1 elements
for (i = 0; i <= L, i++)
histogram[i] = 0;
for( y = 0; y < h; y++)
   for(x = 0; x < w; x++)
```

```
histogram[f(x,y)] = histogram[f(x,y)] + 1;
```

It is to be noted that two completely different images can have exactly the same histogram.

Java2D API does not provide a histogram class. However it is relatively easy to develop one. The following methods are useful for this purpose.

```
Histogram(BufferedImage img)  //constructs histogram from the image.
Object  clone  //returns a copy of this histogram
int  getNumBands()  // returns 1 for grayscale and 3 for color images
int getNumPixels(int value)  //returns the number of pixels of specified
                             // gray level value
```

3.6 Histogram Equalization

An image that has a histogram with the property

$$\mathbf{H[i]} = \frac{n^2}{L}, \quad i = 0, 1, 2, \ldots, L \tag{3.5}$$

or for the normalized case

$$h[i] = \frac{1}{L}, \quad i = 0, 1, 2, \ldots, L \tag{3.6}$$

is called a perfectly equalized histogram. In this image there are equal number of pixels of different gray levels. Such an image has good contrast (why?). A low contrast image has most pixels in a certain gray level range, and to improve the contrast one must spread out the gray levels such that the resulting histogram is approximately equalized.

The concept of cumulative histogram plays an important role in achieving equalization, and is defined as

$$H_c[i] = \sum_{l=0}^{i} h[l] \quad ; \quad i = 0, 1, 2, \ldots, L \tag{3.7}$$

A perfectly equalized histogram image has a flat histogram (Fig 3.9 a), and has a linearly increasing cumulative histogram. This is verified by substituting (3.6) into (3.7) to get

$$H_c[i] = \sum_{l=0}^{i} 1/L = i/L \qquad i = 0, 1, 2, \ldots, L \qquad (3.8)$$

(3.8)

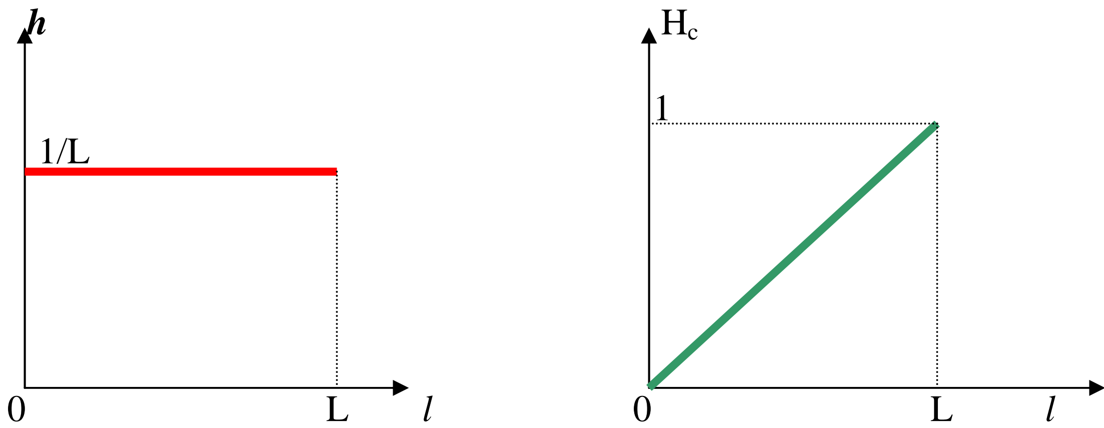


Fig. 3. 9 Normalized histogram, and cumulative histogram (b)

It can be shown that the mapping needed to achieve the equalization is

$$g = H_c[i] = \sum_{l=0}^{i} h[l] \qquad (3.9)$$

The meaning of (3.9) is that the gray level i $(i = 0, 1, 2, \ldots L)$ in the input image $f(x, y)$ must map into the gray level $\sum_{l=0}^{i} h[l]$ in the output image $g(x,y)$, where $h[l]$ is the normalized histogram of the input image.

- Note that (3.9) achieves perfect equalization if the gray levels values are continuous, i.e. if there are infinite number of gray levels, $L \to \infty$.

- In most cases, however, L is finite and small (e.g. L = 255), and therefore the histogram of the equalized image is not perfect.

Algorithm implements histogram equalization:

Compute normalization factor $a = \frac{L}{n^2}$ or $a = \frac{L}{w * h}$ for non square images

Calculate histogram using the above algorithm

```
Hc[0] = a*histogram[0]   //histogram[ ] is the same as H[ ]
for (l = 1; l <= L; l ++)
    Hc[l] = Hc[l -1] + a * histogram[i];
for (y = 0; y < h; y++)
   for (x= 1; x < w; x++)
        g(x,y) = Hc[f(x,y)];
```

(a) (b)

- Note that the histogram of the original image has a high concentration of bright and medium values with few darker pixels. The equalized image has a more uniform distribution of the gray levels.
- Histogram equalization is widely used in image processing, and it can improve the quality of an image as shown in Fig. 3.10.
- However, this operation is not always beneficial since the improvement in contrast is optimal statistically rather than perceptually (why?). In images with narrow histograms and relatively few gray levels, excessive increase in contrast due to histogram equalization can have the adverse effect of reducing the perceived image quality, especially if noise is present in the image.

3.7 Color Images and Contrast Enhancement

- To find the histogram of a color image, we must provide a Histogram class that gives us three separate components, one for each of R, G, B. The histogram array is now histogram[r][g][b], and can be implemented by the following algorithm.

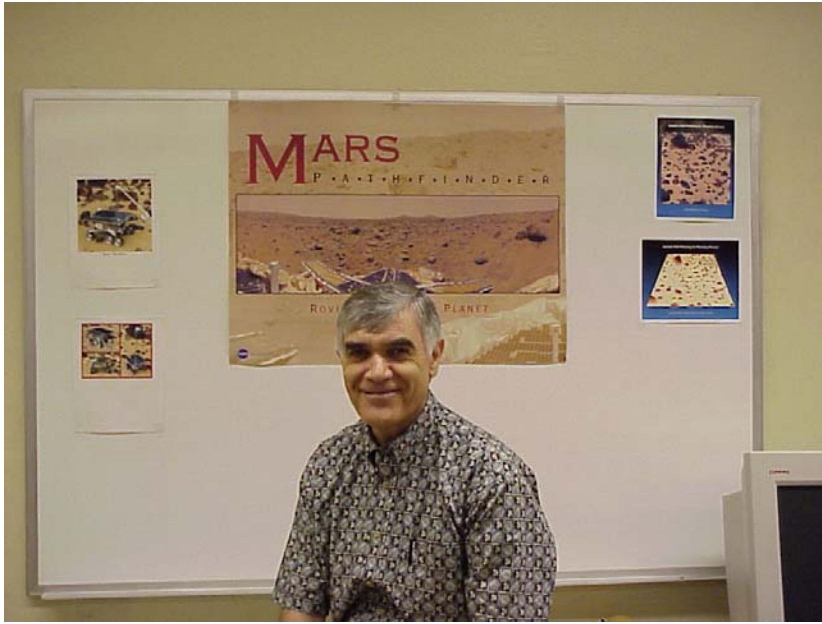
```
Create a 3D array of histogram of dimension (L+1)×(L+1) ×(L+1)
for (r = 0; r <= L, r++)
   for (g = 0; g <= L, g++)
     for (b = 0; b <= L, b++)
        histogram[r][g][b] = 0;
for (y = 0; y< h, y++)
  for (x = 0; x< h, x++)
     find red component of f(x,y)
     find green component of f(x,y)
     find blue component of f(x,y)
     increment histogram[r][g][b] by 1;
```

- A histogram for color images can be plotted either as three 1D-curves (one for each color) on the same graph, as three 2-Ddimensional surfaces, or as one 3-D volume. The 3-D volume representation is used in the above algorithm.

- If L = 255, then an array of dimension $256 \times 256 \times 256$ is needed giving over 16 million values. If a 32 bit integer is used for pixel counts, then the total storage requirement for a single histogram will be 64 MB ! This type of histogram will be very sparsely populated, and we should use a data structure for the histogram (e.g. a hash table) that will use the sparseness property.

- If we want to enhance the image, say by histogram equalization on each of the three components, separately, then the intensity distribution of each component is changed in a different way with the result that both contrast and color are changed.

- The above problem arises because each component of the RGB model contains both color and intensity information. If we wish to manipulate color and intensity separately, we must use the HSI model, where intensity (I), hue (H) and saturation (S) can be adjusted independently.



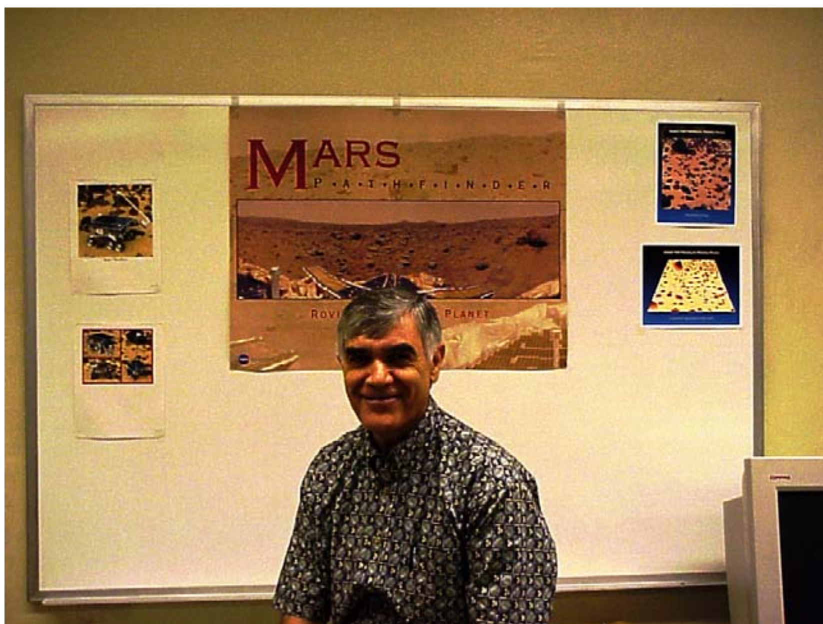


Fig. 3.10 (a)Original image, (b) equalized image.