3
Pixels

In this chapter we discuss the basis of all digital image processing: *the pixel*. We cover the types of tangible information that pixels can contain and discuss operations both on individual pixels (point transforms) and on distributions of pixels (global transforms).

3.1 What is a pixel?

The word pixel is an abbreviation of 'picture element'. Indexed as an (x, y) or column-row (c, r) location from the origin of the image, it represents the smallest, constituent element in a digital image and contains a numerical value which is the basic unit of information within the image at a given spatial resolution and quantization level. Commonly, pixels contain the colour or intensity response of the image as a small point sample of coloured light from the scene. However, not all images necessarily contain strictly visual information. An image is simply a 2-D signal digitized as a grid of pixels, the values of which may relate to other properties other than colour or light intensity. The information content of pixels can vary considerably depending on the type of image we are processing:

- *Colour/grey-scale images* Commonly encountered images contain information relating to the colour or grey-scale intensity at a given point in the scene or image.

- *Infrared (IR)* The visual spectrum is only a small part of the electromagnetic spectrum. IR offers us the capability of imaging scenes based upon either the IR light reflectance or upon the IR radiation they are emitting. IR radiation is emitted in proportion to heat generated/reflected by an object and, thus, IR imaging is also commonly referred to as thermal imaging. As IR light is invisible to the naked human eye, IR illumination and imaging systems offer a useful method for covert surveillance (as such, IR imaging commonly forms the basis for night-vision systems).

- *Medical imaging* Many medical images contain values that are proportional to the absorption characteristics of tissue with respect to a signal projected through the body. The most common types are computed tomography (CT) and magnetic resonance imaging (MRI). CT images, like conventional X-rays, represent values that are directly proportional to the density of the tissue through which the signal passed. By contrast, magnetic resonance images exhibit greater detail but do not have a direct relationship to

Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab Chris Solomon and Toby Breckon
 ISBN: 978-0-470-84472-4

a single quantifiable property of the tissue. In CT and MRI our 2-D image format is commonly extended to a 3-D volume; the 3-D volume is essentially just a stack of 2-D images.

- *Radar/sonar imaging* A radar or sonar image represents a cross-section of a target in proportion to its distance from the sensor and its associated signal 'reflectivity'. Radar is commonly used in aircraft navigation, although it has also been used on road vehicle projects. Satellite-based radar for weather monitoring is now commonplace, as is the use of sonar on most modern ocean-going vessels. Ground-penetrating radar is being increasingly used for archaeological and forensic science investigations.

- *3-D imaging* Using specific 3-D sensing techniques such as stereo photography or 3-D laser scanning we can capture data from objects in the world around us and represent them in computer systems as 3-D images. 3-D images often correspond to depth maps in which every pixel location contains the distance of the imaged point from the sensor. In this case, we have explicit 3-D information rather than just a projection of 3-D as in conventional 2-D imaging. Depending on the capture technology, we may have only 3-D depth information or both 3-D depth and colour for every pixel location. The image depth map can be re-projected to give a partial view of the captured 3-D object (such data is sometimes called $2^1/_2$-D).

- *Scientific imaging* Many branches of science use a 2-D (or 3-D)-based discrete format for the capture of data and analysis of results. The pixel values may, in fact, correspond to chemical or biological sample densities, acoustic impedance, sonic intensity, etc. Despite the difference in the information content, the data is represented in the same form, i.e. a 2-D image. Digital image processing techniques can thus be applied in many different branches of scientific analysis.

Figure 3.1 shows some examples of images with different types of pixel information. These are just a few examples of both the variety of digital images in use and of the broad scale of application domains for digital image processing. In the colour/grey-scale images we will consider in this book the pixels will usually have integer values within a given quantization range (e.g. 0 to 255, 8-bit images), although for other forms of image information (e.g. medical, 3-D, scientific) floating-point real pixel values are commonplace.

3.2 Operations upon pixels

The most basic type of image-processing operation is a point transform which maps the values at individual points (i.e. pixels) in the input image to corresponding points (pixels) in an output image. In the mathematical sense, this is a one-to-one functional mapping from input to output. The simplest examples of such image transformations are arithmetic or logical operations on images. Each is performed as an operation between two images I_A and I_B or between an image and a constant value C:

$$\begin{aligned} I_{\text{output}} &= I_A + I_B \\ I_{\text{output}} &= I_A + C \end{aligned} \tag{3.1}$$

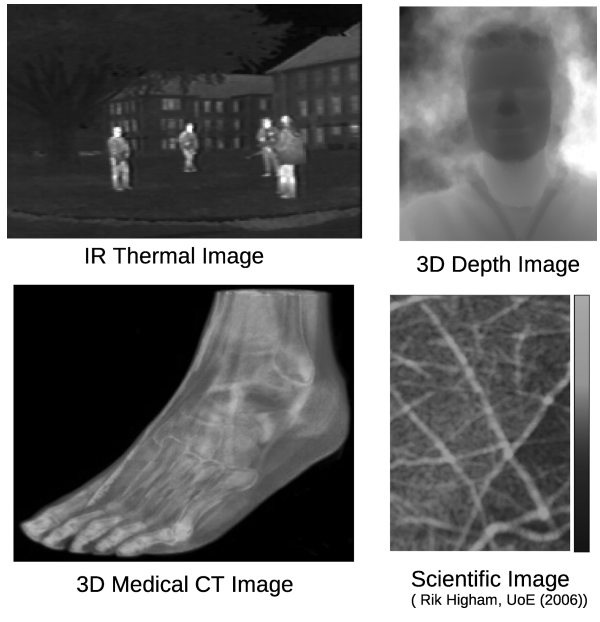

Figure 3.1 Differing types of pixel information

In both cases, the values at an individual pixel location (i, j) in the output image are mapped as follows:

$$\begin{aligned} I_{\text{output}}(i,j) &= I_{\text{A}}(i,j) + I_{\text{B}}(i,j) \\ I_{\text{output}}(i,j) &= I_{\text{A}}(i,j) + C \end{aligned} \tag{3.2}$$

To perform the operation over an entire image of dimension $C \times R$, we simply iterate over all image indices for $(i, j) = \{0 \ldots C-1, 0 \ldots R-1\}$ in the general case (N.B. in Matlab® pixel indices are $\{1 \ldots C, 1 \ldots R\}$).

3.2.1 Arithmetic operations on images

Basic arithmetic operations can be performed quickly and easily on image pixels for a variety of effects and applications.

3.2.1.1 Image addition and subtraction

Adding a value to each image pixel value can be used to achieve the following effects (Figure 3.2):

- *Contrast adjustment* Adding a positive constant value C to each pixel location increases its value and, hence, its brightness.

- *Blending* Adding images together produces a composite image of both input images. This can be used to produce blending effects using weighted addition.

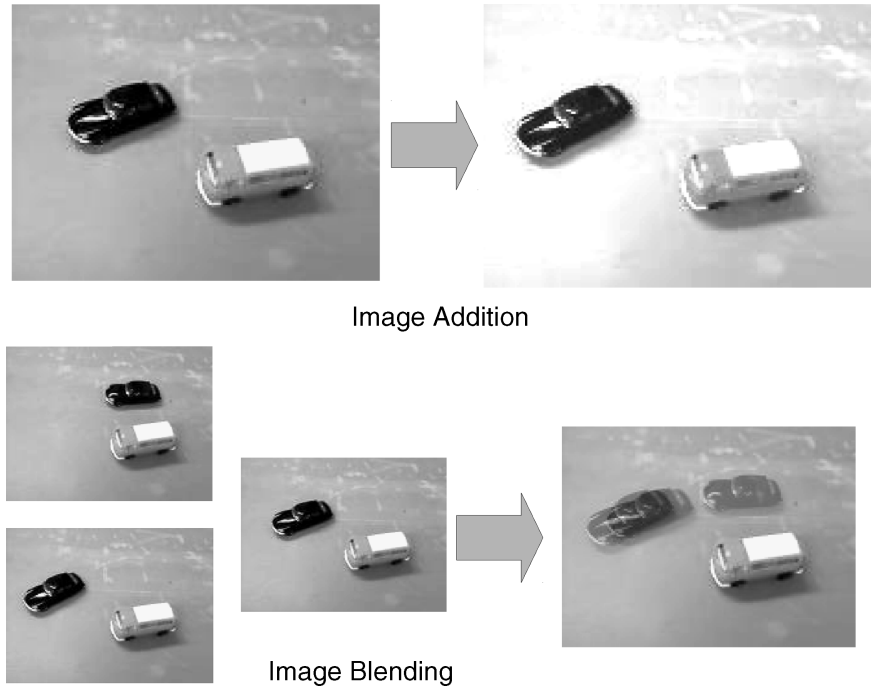

Figure 3.2 Image contrast adjustment and blending using arithmetic addition

Image addition can be carried out in Matlab, as shown in Example 3.1.

Example 3.1

Matlab code	**What is happening?**
A=imread('cameraman.tif');	%Read in image
subplot(1,2,1), imshow(A);	%Display image
B = imadd(A, 100);	%Add 100 to each pixel value in image A
subplot(1,2,2), imshow(B);	%Display result image B

Comments

- Generally, images must be of the same dimension and of the same data type (e.g. 8-bit integer) for addition and subtraction to be possible between them.
- When performing addition operations we must also be aware of integer overflow. An 8-bit integer image can only hold integer values from 0 to 255. The Matlab functions ***imadd, imsubtract*** and ***imabsdiff*** avoid this problem by truncating or rounding overflow values.

Subtracting a constant value from each pixel (like addition) can also be used as a basic form of contrast adjustment. Subtracting one image from another shows us the difference

Figure 3.3 Image differencing using arithmetic subtraction

between images. If we subtract two images in a video sequence then we get a difference image (assuming a static camera) which shows the movement or changes that have occurred between the frames in the scene (e.g. Figure 3.3). This can be used as a basic form of change/movement detection in video sequences. Image subtraction can be carried out in Matlab as shown in Example 3.2.

Example 3.2

Matlab code	**What is happening?**
A=imread('cola1.png');	%Read in 1st image
B=imread('cola2.png');	%Read in 2nd image
subplot(1,3,1), imshow(A);	%Display 1st image
subplot(1,3,2), imshow(B);	%Display 2nd image
Output = imsubtract(A, B);	%Subtract images
subplot(1,3,3), imshow(Output);	%Display result

A useful variation on subtraction is the absolute difference $I_{output} = |I_A - I_B|$ between images (Example 3.3). This avoids the potential problem of integer overflow when the difference becomes negative.

Example 3.3

Matlab code	**What is happening?**
Output = imabsdiff(A, B);	%Subtract images
subplot(1,3,3), imshow(Output);	%Display result

3.2.1.2 Image multiplication and division

Multiplication and division can be used as a simple means of contrast adjustment and extension to addition/subtraction (e.g. reduce contrast to 25% = division by 4; increase contrast by 50% = multiplication by 1.5). This procedure is sometimes referred to as image colour scaling. Similarly, division can be used for image differencing, as dividing an image by another gives a result of 1.0 where the image pixel values are identical and a value not equal to 1.0 where differences occur. However, image differencing using subtraction is

computationally more efficient. Following from the earlier examples, image multiplication and division can be performed in Matlab as shown in Example 3.4.

Example 3.4

Matlab code	**What is happening?**
Output = immultiply(A,1.5);	%Multiply image by 1.5
subplot(1,3,3), imshow(Output);	%Display result
Output = imdivide(A,4);	%Divide image by 4
subplot(1,3,3), imshow(Output);	%Display result

Comments

Multiplying different images together or dividing them by one another is not a common operation in image processing.

For all arithmetic operations between images we must ensure that the resulting pixel values remain within the available integer range of the data type/size available. For instance, an 8-bit image (or three-channel 24-bit colour image) can represent 256 values in each pixel location. An initial pixel value of 25 multiplied by a constant (or secondary image pixel value) of 12 will exceed the 0–255 value range. Integer overflow will occur and the value will ordinarily 'wrap around' to a low value. This is commonly known as saturation in the image space: the value exceeds the representational capacity of the image. A solution is to detect this overflow and avoid it by setting all such values to the maximum value for the image representation (e.g. truncation to 255). This method of handling overflow is implemented in the ***imadd***, ***imsubtract***, ***immultiply*** and ***imdivide*** Matlab functions discussed here. Similarly, we must also be aware of negative pixel values resulting from subtraction and deal with these accordingly; commonly they are set to zero. For three-channel RGB images (or other images with vectors as pixel elements) the arithmetic operation is generally performed separately for each colour channel.

3.2.2 Logical operations on images

We can perform standard logical operations between images such as NOT, OR, XOR and AND. In general, logical operation is performed between each corresponding bit of the image pixel representation (i.e. a bit-wise operator).

- *NOT (inversion)* This inverts the image representation. In the simplest case of a binary image, the (black) background pixels become (white) foreground and vice versa. For grey-scale and colour images, the procedure is to replace each pixel value $I_{\text{input}}(i,j)$ as follows:

$$I_{\text{output}}(i,j) = \text{MAX} - I_{\text{ouput}}(i,j) \tag{3.3}$$

  where MAX is the maximum possible value in the given image representation. Thus, for an 8-bit grey-scale image (or for 8-bit channels within a colour image), MAX = 255.

  In Matlab this can be performed as in Example 3.5.

Example 3.5

Matlab code	What is happening?
A=imread('cameraman.tif');	%Read in image
subplot(1,2,1), imshow(A);	%Display image
B = imcomplement(A);	%Invert the image
subplot(1,2,2), imshow(B);	%Display result image B

- *OR/XOR* Logical OR (and XOR) is useful for processing binary-valued images (0 or 1) to detect objects which have moved between frames. Binary objects are typically produced through application of thresholding to a grey-scale image. Thresholding is discussed in Section 3.2.3.

- *AND* Logical AND is commonly used for detecting differences in images, highlighting target regions with a binary mask or producing bit-planes through an image, as discussed in Section 1.2.1 (Figure 1.3). These operations can be performed in Matlab as in Example 3.6.

Example 3.6

Matlab code	What is happening?
A=imread('toycars1.png');	%Read in 1st image
B=imread('toycars2.png');	%Read in 2nd image
Abw=im2bw(A);	%convert to binary
Bbw=im2bw(B);	%convert to binary
subplot(1,3,1), imshow(Abw);	%Display 1st image
subplot(1,3,2), imshow(Bbw);	%Display 2nd image
Output = xor(Abw, Bbw);	%xor images images
subplot(1,3,3), imshow(Output);	%Display result

Comments

- note that the images are first converted to binary using the Matlab im2bw function (with an automatic threshold - Section 3.2.3).
- note that the resulting images from the im2bw function and the xor logical operation is of Matlab type 'logical'.
- the operators for AND is '&' whilst the operator for OR is '|' and are applied in infix notation form as A & B, A | B.

Combined operators, such as NAND, NOR and NXOR, can also be applied in a similar manner as image-processing operators.

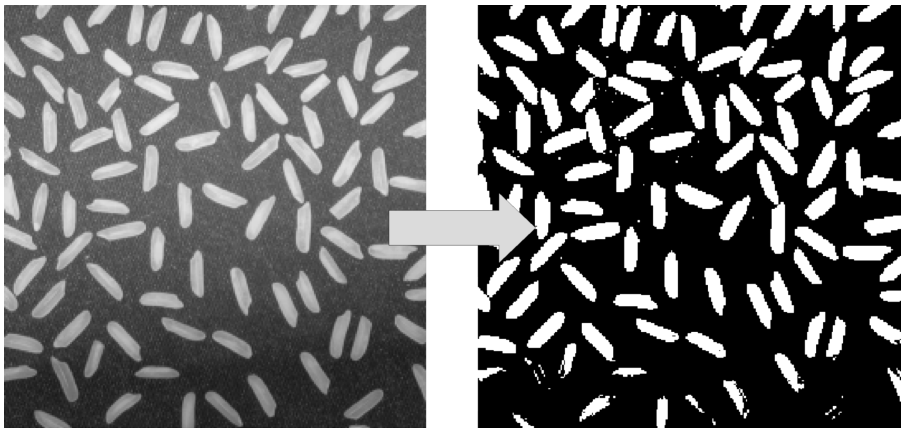

Figure 3.4 Thresholding for object identification

3.2.3 Thresholding

Another basic point transform is thresholding. Thresholding produces a binary image from a grey-scale or colour image by setting pixel values to 1 or 0 depending on whether they are above or below the threshold value. This is commonly used to separate or segment a region or object within the image based upon its pixel values, as shown in Figure 3.4.

In its basic operation, thresholding operates on an image I as follows:

```
for each pixel I(i,j) within the image I
      if I(i,j) > threshold
            I(i,j) = 1
      else
            I(i,j) = 0
      end
end
```

In Matlab, this can be carried out using the function ***im2bw*** and a threshold in the range 0 to 1, as in Example 3.7.

Example 3.7

Matlab code	**What is happening?**
I=imread('trees.tif');	%Read in 1st image
T=im2bw(I, 0.1);	%Perform thresholding
subplot(1,3,1), imshow(I);	%Display original image
subplot(1,3,2), imshow(T);	%Display thresholded image

The ***im2bw*** function automatically converts colour images (such as the input in the example) to grayscale and scales the threshold value supplied (from 0 to 1) according to the given quantization range of the image being processed. An example is shown in Figure 3.5.

For grey-scale images, whose pixels contain a single intensity value, a single threshold must be chosen. For colour images, a separate threshold can be defined for each channel (to

Figure 3.5 Thresholding of a complex image

correspond to a particular colour or to isolate different parts of each channel). In many applications, colour images are converted to grey scale prior to thresholding for simplicity. Common variations on simple thresholding are:

- the use of two thresholds to separate pixel values within a given range;
- the use of multiple thresholds resulting in a labelled image with portions labelled 0 to N;
- retaining the original pixel information for selected values (i.e. above/between thresholds) whilst others are set to black.

Thresholding is the 'work-horse' operator for the separation of image foreground from background. One question that remains is how to select a good threshold. This topic is addressed in Chapter 10 on image segmentation.

3.3 Point-based operations on images

The dynamic range of an image is defined as the difference between the smallest and largest pixel values within the image. We can define certain functional transforms or mappings that alter the effective use of the dynamic range. These transforms are primarily applied to improve the contrast of the image. This improvement is achieved by altering the relationship between the dynamic range of the image and the grey-scale (or colour) values that are used to represent the values.

In general, we will assume an 8-bit (0 to 255) grey-scale range for both input and resulting output images, but these techniques can be generalized to other input ranges and individual channels from colour images.

3.3.1 Logarithmic transform

The dynamic range of an image can be compressed by replacing each pixel value in a given image with its logarithm: $I_{\text{output}}(i,j) = \ln I_{\text{input}}(i,j)$, where $I(i,j)$ is the value of a pixel at a location (i,j) in image I and the function ln() represents the natural logarithm. In practice, as the logarithm is undefined for zero, the following general form of the logarithmic

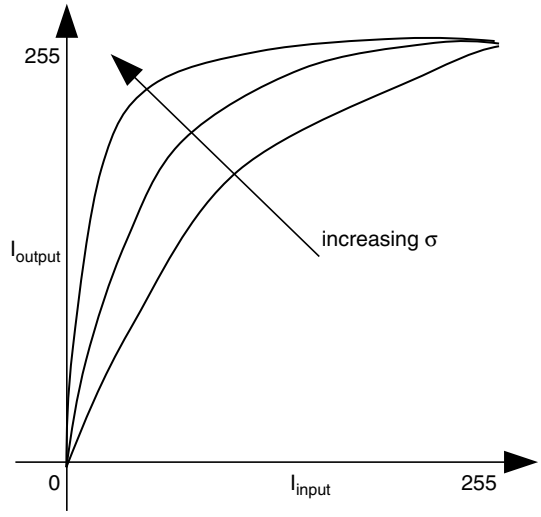

Figure 3.6 The Logarithmic Transform: Varying the parameter σ changes the gradient of the logarithmic function used for input to output

transform is used:

$$I_{\text{output}}(i,j) = c\ln[1 + (e^{\sigma}-1)I_{\text{input}}(i,j)] \tag{3.4}$$

Note that the scaling factor σ controls the input range to the logarithmic function, whilst c scales the output over the image quantization range 0 to 255. The addition of 1 is included to prevent problems where the logarithm is undefined for $I_{\text{input}}(i,j) = 0$. The level of dynamic range compression is effectively controlled by the parameter σ. As shown in Figure 3.6, as the logarithmic function is close to linear near the origin, the compression achieved is smaller for an image containing a low range of input values than one containing a broad range of pixel values.

The scaling constant c can be calculated based on the maximum allowed output value (255 for an 8-bit image) and the maximum value $\max(I_{\text{input}}(i,j))$ present in the input:

$$c = \frac{255}{\log[1 + \max(I_{\text{input}}(i,j))]} \tag{3.5}$$

The effect of the logarithmic transform is to increase the dynamic range of dark regions in an image and decrease the dynamic range in the light regions (e.g. Figure 3.7). Thus, the logarithmic transform maps the lower intensity values or dark regions into a larger number of greyscale values and compresses the higher intensity values or light regions in to a smaller range of greyscale values.

In Figure 3.7 we see the typical effect of being photographed in front of a bright background (left) where the dynamic range of the film or camera aperture is too small to capture the full range of the scene. By applying the logarithmic transform we brighten the foreground of this image by spreading the pixel values over a wider range and revealing more of its detail whilst compressing the background pixel range.

In Matlab, the logarithmic transform can be performed on an image as in Example 3.8. From this example, we can see that increasing the multiplication constant c increases the overall brightness of the image (as we would expect from our earlier discussion of multiplication in Section 3.4). A common variant which achieves a broadly similar result is the simple

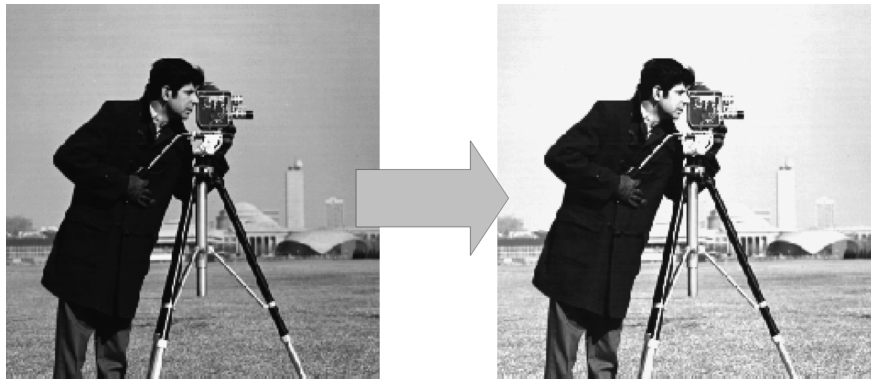

Figure 3.7 Applying the logarithmic transform to a sample image

square-root transform (i.e. mapping output pixel values to the square root of the input) which similarly compresses high-value pixel ranges and spreads out low-value pixel ranges.

Example 3.8

Matlab code	**What is happening?**
I=imread('cameraman.tif');	%Read in image
subplot(2,2,1), imshow(I);	%Display image
Id=im2double(I); Output1=2*log(1 + Id); Output2=3*log(1 + Id); Output3=5*log(1 + Id);	

subplot(2,2,2), imshow(Output1);
subplot(2,2,3), imshow(Output2);
subplot(2,2,4), imshow(Output3);
	%Display result images

3.3.2 Exponential transform

The exponential transform is the inverse of the logarithmic transform. Here, the mapping function is defined by the given base e raised to the power of the input pixel value:

$$I_{\text{output}}(i,j) = \mathrm{e}^{I_{\text{input}}(i,j)} \tag{3.6}$$

where $I(i,j)$ is the value of a pixel at a location (i,j) in image I.

This transform enhances detail in high-value regions of the image (bright) whilst decreasing the dynamic range in low-value regions (dark) – the opposite effect to the logarithmic transform. The choice of base depends on the level of dynamic range compression required. In general, base numbers just above 1 are suitable for photographic image enhancement. Thus, we expand our exponential transform notation to include a variable base and scale to the appropriate output range as before:

$$I_{\text{output}}(i,j) = c[(1+\alpha)^{I_{\text{input}}(i,j)} - 1] \tag{3.7}$$

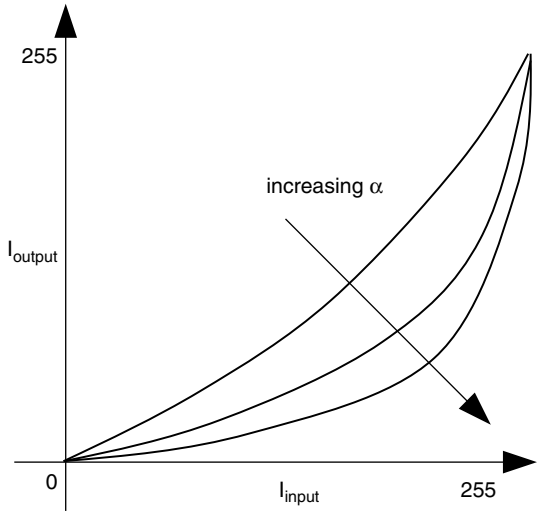

Figure 3.8 The Exponential Transform: Varying the parameter α changes the gradient of the exponential function used for input to output

Here, $(1+\alpha)$ is the base and c is the scaling factor required to ensure the output lies in an appropriate range. As is apparent when $I_{\text{input}}(i,j) = 0$, this results in $I_{\text{output}}(i,j) = c$ unless we add in the -1 to counter this potential offset appearing in the output image. The level of dynamic range compression and expansion is controlled by the portion of the exponential function curve used for the input to output mapping; this is determined by parameter α. As shown in Figure 3.8, as the exponential function is close to linear near the origin, the compression is greater for an image containing a lower range of pixel values than one containing a broader range.

We can see the effect of the exponential transform (and varying the base) in Figure 3.9 and Example 3.9. Here, we see that the contrast of the background in the original image can be improved by applying the exponential transform, but at the expense of contrast in the darker areas of the image. The background is a high-valued area of the image (bright), whilst the darker regions have low pixel values. This effect increases as the base number is increased.

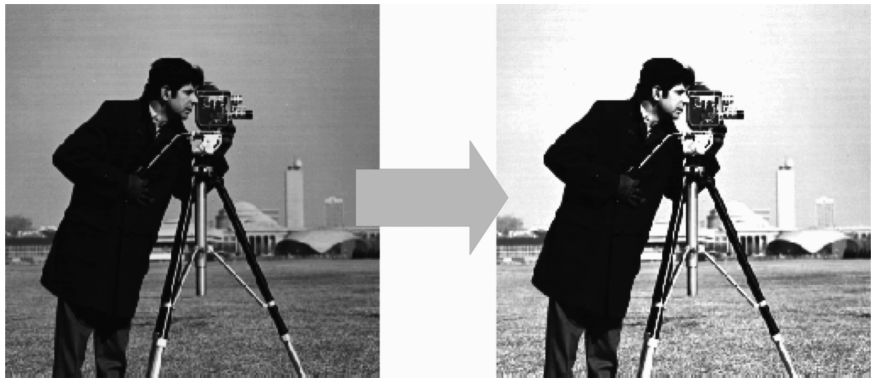

Figure 3.9 Applying the exponential transform to a sample image

Example 3.9

Matlab code	What is happening?
I=imread('cameraman.tif');	%Read in image
subplot(2,2,1), imshow(I);	%Display image
Id=im2double(I); Output1=4*(((1 + 0.3).^(Id)) - 1); Output2=4*(((1 + 0.4).^(Id)) - 1); Output3=4*(((1 + 0.6).^(Id)) - 1);	

subplot(2,2,2), imshow(Output1);
subplot(2,2,3), imshow(Output2);
subplot(2,2,4), imshow(Output3);
	%Display result images

In Matlab, the exponential transform can be performed on an image as in Example 3.9. In this example we keep c constant whilst varying the exponential base parameter α, which in turn varies the effect of the transform on the image (Figure 3.9).

3.3.3 Power-law (gamma) transform

An alternative to both the logarithmic and exponential transforms is the 'raise to a power' or power-law transform in which each input pixel value is raised to a fixed power:

$$I_{\text{output}}(i,j) = c(I_{\text{input}}(i,j))^{\gamma} \tag{3.8}$$

In general, a value of $\gamma > 1$ enhances the contrast of high-value portions of the image at the expense of low-value regions, whilst we see the reverse for $\gamma < 1$. This gives the power-law transform properties similar to both the logarithmic ($\gamma < 1$) and exponential ($\gamma > 1$) transforms. The constant c performs range scaling as before.

In Matlab, this transform can be performed on an image as in Example 3.10. An example of this transform, as generated by this Matlab example, is shown in Figure 3.10.

Example 3.10

Matlab code	What is happening?
I=imread('cameraman.tif');	%Read in image
subplot(2,2,1), imshow(I);	%Display image
Id=im2double(I); Output1=2*(Id.^0.5); Output2=2*(Id.^1.5); Output3=2*(Id.^3.0);	

subplot(2,2,2), imshow(Output1);
subplot(2,2,3), imshow(Output2);
subplot(2,2,4), imshow(Output3);
	%Display result images

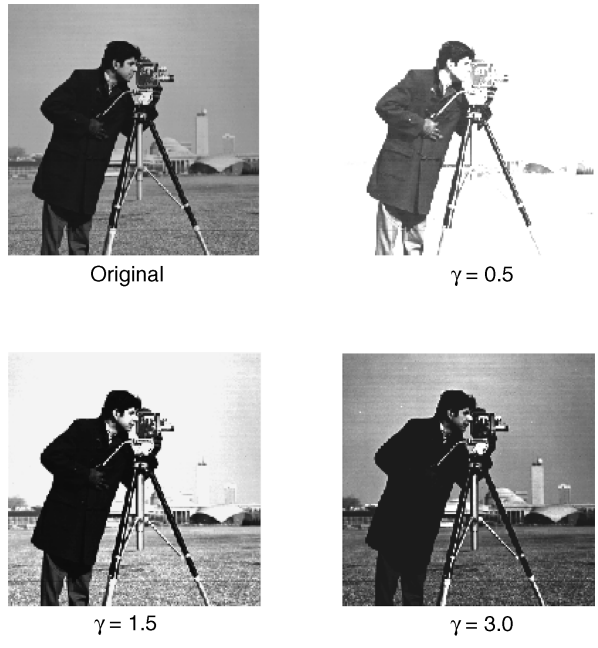

Figure 3.10 Applying the power-law transform to a sample image

3.3.3.1 Application: gamma correction

A common application of the power-law transform is gamma correction. Gamma correction is the term used to describe the correction required for the nonlinear output curve of modern computer displays. When we display a given intensity on a monitor we vary the analogue voltage in proportion to the intensity required. The problem that all monitors share is the nonlinear relationship between input voltage and output intensity. In fact, the monitor output intensity generally varies with the power of input voltage curve, as approximately $\gamma = 2.5$ (this varies with type of monitor, manufacturer, age, etc.). This means that, when you request an output intensity equal to say i, you in fact get an intensity of $i^{2.5}$. In order to counter this problem, we can preprocess image intensities using an inverse power-law transform prior to output to ensure they are displayed correctly. Thus, if the effect of the display gamma can be characterized as $I_{\text{output}} = (I_{\text{input}})^{\gamma}$ where γ is the r value for the output curve of the monitor (lower right), then pre-correcting the input with the inverse power-law transform, i.e. $(I_{\text{input}})^{1/\gamma}$, compensates to produce the correct output as

$$I_{\text{output}} = ((I_{\text{input}})^{1/\gamma})^{\gamma} = I_{\text{input}} \tag{3.9}$$

Modern operating systems have built in gamma correction utilities that allow the user to specify a γ value that will be applied in a power-law transform performed automatically on all graphical output. Generally, images are displayed assuming a γ of 2.2. For a specific monitor, gamma calibration can be employed to determine an accurate γ value.

We can perform gamma correction on an image in Matlab using the ***imadjust*** function to perform the power-law transform, as in Example 3.11. An example of the result is shown in Figure 3.11.

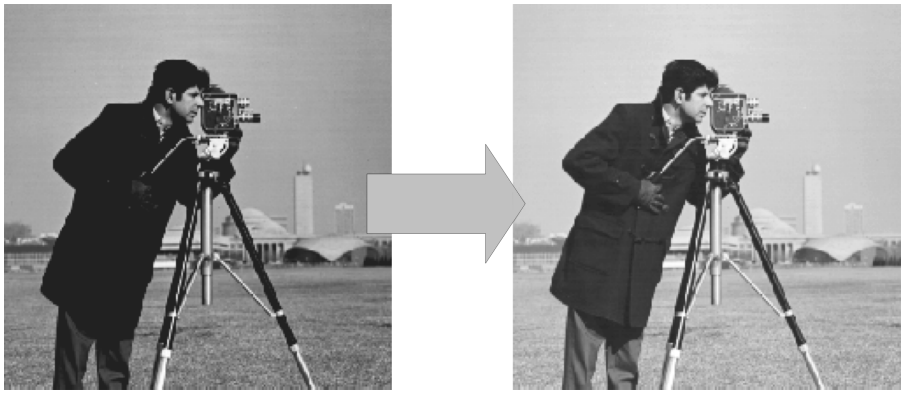

Figure 3.11 Gamma correction on a sample image

Example 3.11

Matlab code	**What is happening?**
A=imread('cameraman.tif');	%Read in image
subplot(1,2,1), imshow(A);	%Display image
B=imadjust(A,[0 1],[0 1],1./3);	%Map input grey values of image A in range 0–1 to %an output range of 0–1 with gamma factor of 1/3 %(i.e. r = 3).
	%Type ≫ *doc imadjust* for details of possible syntaxes
subplot(1,2,2), imshow(B);	%Display result.

3.4 Pixel distributions: histograms

An image histogram is a plot of the relative frequency of occurrence of each of the permitted pixel values in the image against the values themselves. If we normalize such a frequency plot, so that the total sum of all frequency entries over the permissible range is one, we may treat the image histogram as a discrete probability density function which defines the likelihood of a given pixel value occurring within the image. Visual inspection of an image histogram can reveal the basic contrast that is present in the image and any potential differences in the colour distribution of the image foreground and background scene components.

For a simple grey-scale image, the histogram can be constructed by simply counting the number of times each grey-scale value (0–255) occurs within the image. Each 'bin' within the histogram is incremented each time its value is encountered thus an image histogram can easily be constructed as follows -

```
initialize all histogram array entries to 0
for each pixel I(i,j) within the image I
     histogram(I(i,j)) = histogram(I(i,j)) + 1
end
```

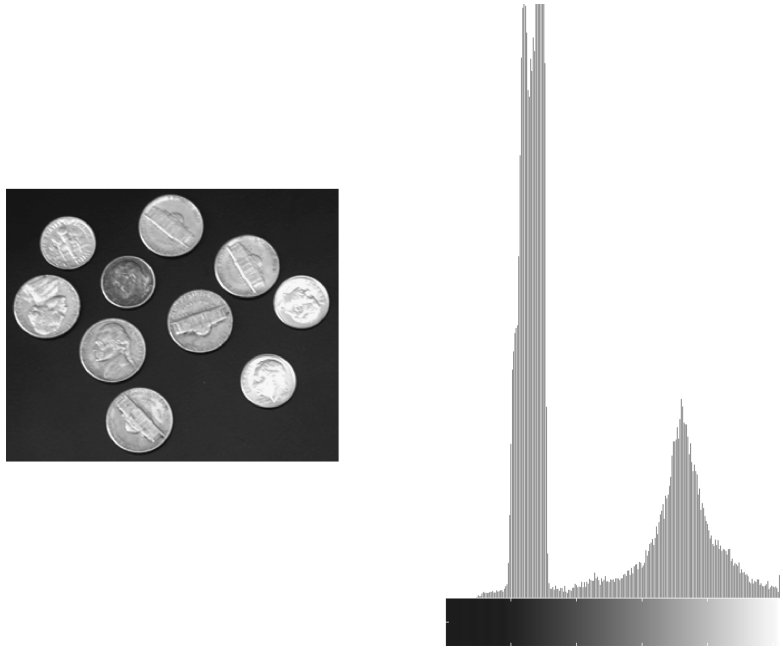

Figure 3.12 Sample image and corresponding image histogram

In Matlab we can calculate and display an image histogram as in Example 3.12. An example histogram is shown in Figure 3.12, where we see a histogram plot with two distinctive peaks: a high peak in the lower range of pixel values corresponds to the background intensity distribution of the image and a lower peak in the higher range of pixel values (bright pixels) corresponds to the foreground objects (coins).

Following on from Example 3.12, we can also individually query and address the histogram 'bin' values to display the pixel count associated with a selected 'bin' within the first peak; see Example 3.13.

Example 3.12

Matlab code	**What is happening?**
I=imread('coins.png');	%Read in image
subplot(1,2,1), imshow(I);	%Display image
subplot(1,2,2), imhist(I);	%Display histogram

Example 3.13

Matlab code	**What is happening?**
I=imread('coins.png');	%Read in image
[counts,bins]=imhist(I);	%Get histogram bin values
counts(60)	%Query 60th bin value

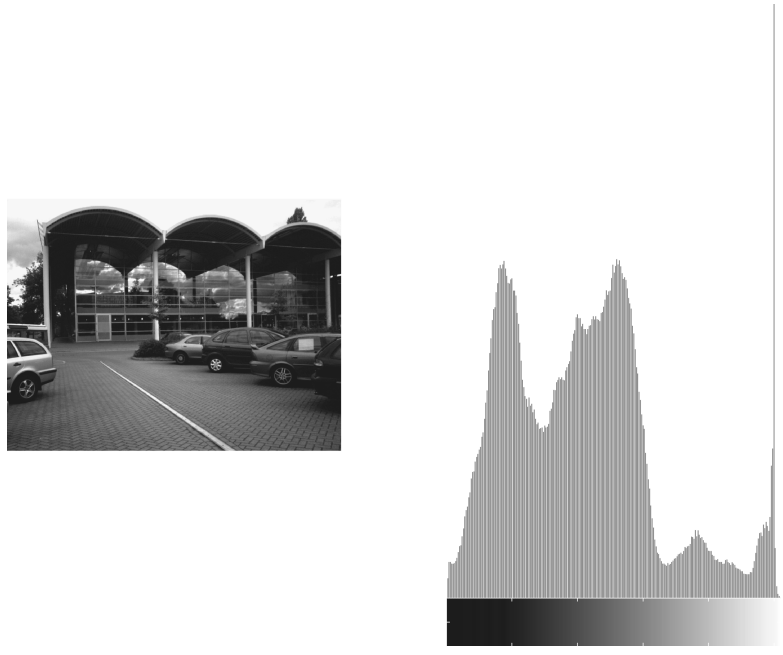

Figure 3.13 Sample image and corresponding image histogram

3.4.1 Histograms for threshold selection

In the example shown in Figure 3.12, we essentially see the image histogram plotted as a bar graph. The *x*-axis represents the range of values within the image (0–255 for 8-bit grayscale) and the *y*-axis shows the number of times each value actually occurs within the particular image. By selecting a threshold value between the two histogram peaks we can successfully separate the background/foreground of the image using the thresholding approach of Section 3.2.3 (a threshold of ~120 is suitable for Figure 3.12). Generally, scenes with clear bimodal distributions work best for thresholding. For more complex scenes, such as Figure 3.13, a more complex histogram occurs and simple foreground/background thresholding cannot be applied. In such cases, we must resort to more sophisticated image segmentation techniques (see Chapter 10).

In Matlab, we can use the image histogram as the basis for calculation of an automatic threshold value. The function ***graythresh*** in Example 3.14 exploits the Otsu method, which chooses that threshold value which minimizes the interclass statistical variance of the thresholded black and white pixels.

Example 3.14

Matlab code	**What is happening?**
I=imread('coins.png');	%Read in image
level=graythresh(I);	%Get OTSU threshold
It=im2bw(I, level);	%Threshold image
imshow(It);	%Display it

3.4.2 Adaptive thresholding

Adaptive thresholding is designed to overcome the limitations of conventional, global thresholding by using a different threshold at each pixel location in the image. This local threshold is generally determined by the values of the pixels in the neighbourhood of the pixel. Thus, adaptive thresholding works from the assumption that illumination may differ over the image but can be assumed to be roughly uniform in a sufficiently small, local neighbourhood.

The local threshold value t in adaptive thresholding can be based on several statistics. Typically, the threshold is chosen to be either $t = \text{mean} + C$, $t = \text{median} + C$ or $\text{floor}((\max - \min)/2) + C$ of the local $N \times N$ pixel neighbourhood surrounding each pixel. The choice of N is important and must be large enough to cover sufficient background and foreground pixels at each point but not so large as to let global illumination deficiencies affect the threshold. When insufficient foreground pixels are present, then the chosen threshold is often poor (i.e. the mean/median or average maximum/minimum difference of the neighbourhood varies largely with image noise). For this reason, we introduce the constant offset C into the overall threshold to set the threshold above the general image noise variance in uniform pixel areas.

In Example 3.15 we carry out adaptive thresholding in Matlab using the threshold $t = \text{mean} + C$ for a neighbourhood of $N = 15$ and $C = 20$. The result of this example is shown in Figure 3.14, where we see the almost perfect foreground separation of the rice grains

Example 3.15

Matlab code	**What is happening?**
I=imread('rice.png');	%Read in image
Im=imfilter(I,fspecial('average',[15 15]),'replicate');	%Create mean image
It=I-(Im + 20);	%Subtract mean image (+ constant C=20)
Ibw=im2bw(It,0);	%Threshold result at 0 (keep + ve results only)
subplot(1,2,1), imshow(I);	%Display image
subplot(1,2,2), imshow(Ibw);	%Display result

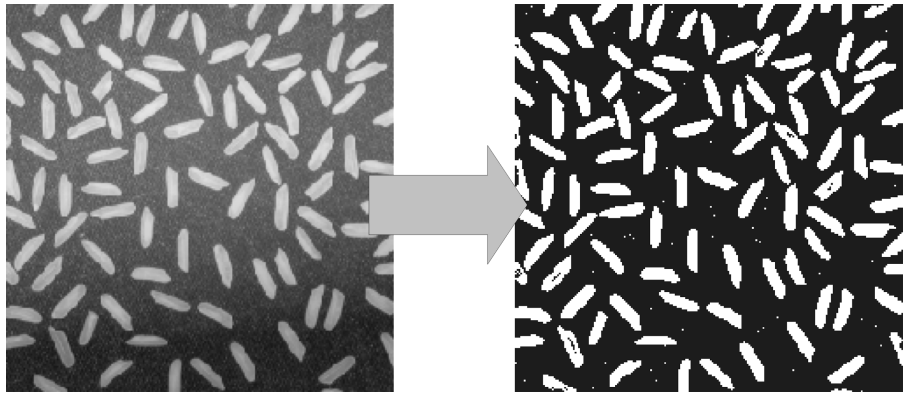

Figure 3.14 Adaptive thresholding applied to sample image

despite the presence of an illumination gradient over the image (contrast this to the bottom of Figure 3.4 using global thresholding). Further refinement of this result could be performed using the morphological operations of opening and closing discussed in Chapter 8.

Example 3.15 can also be adapted to use the threshold $t = \text{median} + C$ by replacing the line of Matlab code that constructs the mean image with one that constructs the median image; see Example 3.16. Adjustments to the parameters N and C are also required to account for the differences in the threshold approach used.

Example 3.16

Matlab code	**What is happening?**
Im=medfilt2(I,[N N]);	%Create median image

3.4.3 Contrast stretching

Image histograms are also used for contrast stretching (also known as normalization) which operates by stretching the range of pixel intensities of the input image to occupy a larger dynamic range in the output image.

In order to perform stretching we must first know the upper and lower pixel value limits over which the image is to be normalized, denoted a and b respectively. These are generally the upper and lower limits of the pixel quantization range in use (i.e. for an 8-bit image, $a = 255$ and $b = 0$ – see Section 1.2). In its simplest form, the first part of the contrast stretch operation then scans the input image to determine the maximum and minimum pixel values currently present, denoted c and d respectively. Based on these four values (a, b, c and d), the image pixel range is stretched according to the following formula:

$$I_{\text{output}}(i,j) = (I_{\text{input}}(i,j) - c)\left(\frac{a-b}{c-d}\right) + a \tag{3.10}$$

for each pixel location denoted (i, j) in the input and output images. Contrast stretching is thus a further example of a point transform, as discussed earlier in this chapter.

In reality, this method of choosing c and d is very naive, as even a single outlier pixel (i.e. one that is unrepresentative of the general pixel range present within the image – e.g. salt and pepper noise) will drastically affect the overall enhancement result. We can improve upon this method by ensuring that c and d are truly representative of the image content and robust to such statistical outliers. Two such methods are presented here.

- *Method 1* Compute the histogram of the input image and select c and d as the 5th and 95th percentile points of the cumulative distribution (i.e. 5% of the image pixels will be less than c and 5% greater than d).

- *Method 2* Compute the histogram of the input image and find the most frequently occurring intensity value within the image. Let us assume that this peak value has a bin count of N. Select as a cut-off some fraction of N (e.g. 5%). Move away from the peak in either direction (left towards 0 one way, right towards 255) until the last values greater than the cut-off are reached. These values are c and d (see the histogram in Figure 3.12).

Figure 3.15 Contrast stretch applied to sample image

Method 2 is somewhat less robust to complex, multi-peak histograms or histograms that do not follow typical intensity gradient distributions.

In Example 3.17, we carry out contrast stretching in Matlab using method 1. The result of Example 3.17 is shown in Figure 3.15, where we see that the contrast is significantly improved (albeit slightly saturated). We can display the histograms before and after contrast stretching for this example using Example 3.18. These histogram distributions are shown in Figure 3.16. We can clearly see the restricted dynamic range of the original image and how the contrast-stretched histogram corresponds to a horizontally scaled version of the original.

Example 3.17

Matlab code	**What is happening?**
I=imread('pout.tif');	%Read in image
Ics=imadjust(I,stretchlim(I, [0.05 0.95]),[]);	%Stretch contrast using method 1
subplot(1,2,1), imshow(I);	%Display image
subplot(1,2,2), imshow(Ics);	%Display result

Comments

Here we use the ***stretchlim()*** function to identify the *c* and *d* pixel values at the 5th and 95th percentile points of the (normalized) histogram distribution of the image. The ***imadjust()*** function is then used to map the this range to the (default) maximum quantization range of the image.

Example 3.18

Matlab code	**What is happening?**
subplot(1,2,3), imhist(I);	%Display input histogram
subplot(1,2,4), imhist(Ics);	%Display output histogram

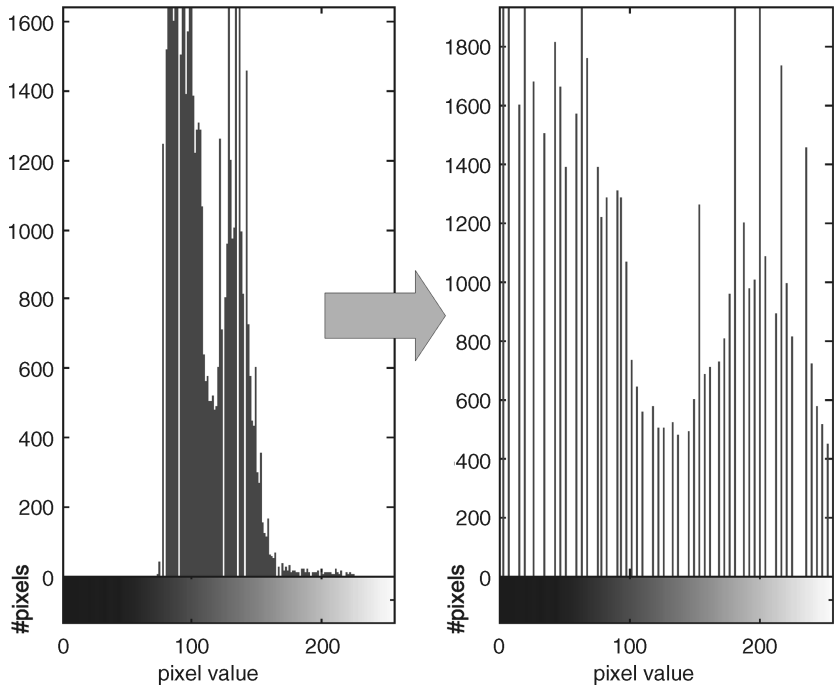

Figure 3.16 Before and after histogram distributions for Figure 3.15

As is evident in this example (Figure 3.15), image enhancement operations such as this are highly subjective in their evaluation, and parameter adjustments (i.e. values c and d) can subsequently make a significant difference to the resulting output.

3.4.4 Histogram equalization

The second contrast enhancement operation based on the manipulation of the image histogram is histogram equalization. This is one of the most commonly used image enhancement techniques.

3.4.4.1 Histogram equalization theory

Initially, we will assume a grey-scale input image, denoted $I_{\text{input}}(x)$. If the variable x is continuous and normalized to lie within the range $[0, 1]$, then this allows us to consider the normalized image histogram as a probability density function (PDF) $p_x(x)$, which defines the likelihood of given grey-scale values occurring within the vicinity of x. Similarly, we can denote the resulting grey-scale output image after histogram equalisation as $I_{\text{output}}(y)$ with corresponding PDF $p_y(y)$.

The essence of the histogram equalization problem is that we seek some transformation function $y = f(x)$ that maps between the input and the output grey-scale image values and which will transform the input PDF $p_x(x)$ to produce the desired output PDF $p_y(y)$. A

standard result from elementary probability theory states that:

$$p_y(y) = p_x(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right| \tag{3.11}$$

which implies that the desired output PDF depends only on the known input PDF and the transformation function $y = f(x)$. Consider, then, the following transformation function, which calculates the area under the input probability density curve (i.e. integral) between 0 and an upper limit x:

$$y(x) = \int_0^x p_x(x')\,\mathrm{d}x' \tag{3.12}$$

This is recognizable as the cumulative distribution function (CDF) of the random variable x. Differentiating this formula, applying Leibniz's rule[1] and substituting into our previous statement we obtain the following:

$$p_y(y) = p_x(x)\left|\frac{1}{p_x(x)}\right| \tag{3.13}$$

Finally, because $p_x(x)$ is a probability density and guaranteed to be positive ($0 \leq p_x(x) \leq 1$), we can thus obtain:

$$p_y(y) = \frac{p_x(x)}{p_x(x)} = 1 \qquad 0 \leq y \leq 1 \tag{3.14}$$

The output probability density $p_y(y)$ is thus constant, indicating that all output intensity values y are equally probable and, thus, the histogram of output intensity values is equalized.

The principle of histogram equalization for the continuous case, therefore, is encapsulated by this mapping function $y = f(x)$ between the input intensity x and the output intensity y, which, as shown, is given by the cumulative integral of the input PDF (the cumulative distribution in statistics). However, we must note that the validity of the density transformation result given above depends on the satisfaction of two conditions. The function $y(x)$ must be (a) single valued and (b) monotonically increasing, i.e. $0 \leq y(x) \leq 1$ for $0 \leq x \leq 1$. Condition (a) first ensures that the inverse transformation $x^{-1}(y)$ exists and that the ascending pixel value order from black to white (dark to light, 0 to 255) is preserved. Note that, if the function were not monotonically increasing, the inverse transformation would not be single valued, resulting in a situation in which some output intensities map to more than one input intensity. The second requirement simply ensures that the output range maps the input.

3.4.4.2 Histogram equalization theory: discrete case

Intensity values in real digital images can occupy only a finite and discrete number of levels. We assume that there are S possible discrete intensity values denoted x_k, where $k = \{0, 1, 2, \ldots, S-1\}$, and the input probability density for level x_k is $p_x(x_k)$. In this

[1] Leibniz's rule: the derivative of a definite integral with respect to its upper limit is equal to the integrand evaluated at the upper limit.

discrete case, the required general mapping $y = f(x)$ can be defined specifically for x_k as the following summation:

$$y(x_k) = \sum_{j=0}^{k} p_x(x_k) \tag{3.15}$$

This is essentially the cumulative histogram for the input image x, where the kth entry $y(x_k)$ is the sum of all the histogram bin entries up to and including k. Assuming, as is normally the case, that the permissible intensity values in the input and output image can be denoted simply by discrete integer values k, where $k = \{0, 1, 2, \ldots, S-1\}$, this can be written simply as:

$$y_k = \sum_{j=0}^{k} p_x(j) = \frac{1}{N} \sum_{j=0}^{k} n_j \tag{3.16}$$

where the population of the kth level is given by n_k and the total number of pixels in the image is N.

This definition makes the corresponding, computational procedure of histogram equalization a simple procedure to implement in practice. Unlike its continuous counterpart, however, the discrete transformation for $y = f(x)$ cannot in general produce a perfectly equalized (uniform) output histogram (i.e. in which all intensity values are strictly equally probable). In practice, it forms a good approximation to the ideal, driven and adapted to the cumulative histogram of the input image, that spreads the intensity values more evenly over the defined quantization range of the image.

3.4.4.3 Histogram equalization in practice

The two main attractions of histogram equalization are that it is a fully automatic procedure and is computationally simple to perform. The intensity transformation $y = f(x)$ we have defined in this section depends only on the readily available histogram of the input image.

Histogram modelling provides a means of modifying the dynamic range of an image such that its histogram distribution conforms to a given shape. In histogram equalization, we employ a monotonic, nonlinear mapping such that the pixels of the input image are mapped to an output image with a uniform histogram distribution. As shown in Section 3.4.4.2, this required mapping can be defined as the cumulative histogram $C(i)$ such that each entry in is the sum of the frequency of occurrence for each grey level up to and including the current histogram bin entry i. By its very nature $C()$ is a single-valued, monotonically increasing function. Its gradient will be in proportion to the current equalization of the image (e.g. a constant gradient of 1 will be a perfectly equalized image, as the increase at each step is constant).

In the idealized case, the resulting equalized image will contain an equal number of pixels each having the same grey level. For L possible grey levels within an image that has N pixels, this equates to the jth entry of the cumulative histogram $C(j)$ having the value jN/L in this idealized case (i.e. j times the equalized value). We can thus find a mapping between input

pixel intensity values i and output pixel intensity values j as follows:

$$C(i) = j\frac{N}{L} \tag{3.17}$$

from which rearrangement gives:

$$j = \frac{L}{N}C(i) \tag{3.18}$$

which represents a mapping from a pixel of intensity value i in the input image to an intensity value of j in the output image via the cumulative histogram $C()$ of the input image.

Notably, the maximum value of j from the above is L, but the range of grey-scale values is strictly $j = \{0 \ldots (L-1)\}$ for a given image. In practice this is rectified by adding a -1 to the equation, thus also requiring a check to ensure a value of $j = -1$ is not returned:

$$j = \max\left(0, \frac{L}{N}C(i) - 1\right) \tag{3.19}$$

which, in terms of our familiar image-based notation, for 2-D pixel locations $i = (c, r)$ and $j = (c, r)$ transforms to

$$I_{\text{output}}(c, r) = \max\left(0, \frac{L}{N}C(I_{\text{input}}(c, r) - 1\right) \tag{3.20}$$

where $C()$ is the cumulative histogram for the input image I_{input}, N is the number of pixels in the input image ($C \times R$) and L is the number of possible grey levels for the images (i.e. quantization limit, Section 1.2). This effectively provides a compact look-up table for mapping pixel values in I_{input} to I_{output}. As an automatic procedure, this mapping, which constitutes histogram equalization, can be readily performed as in Example 3.19. The output result of Example 3.19 is shown in Figure 3.17, which can be compared with the input dynamic range of the image and corresponding input image histogram of Figures 3.15 (left) and 3.16 (left) respectively. Here (Figure 3.17), we can see the equalization effect on the dynamic range of the image and the corresponding equalization of the histogram

Example 3.19

Matlab code	**What is happening?**
I=imread('pout.tif');	%Read in image
Ieq=histeq(I);	
subplot(2,2,1), imshow(I);	%Display image
subplot(2,2,2), imshow(Ieq);	%Display result
subplot(2,2,3), imhist(I);	%Display histogram of image
subplot(2,2,4), imhist(Ieq);	%Display histogram of result

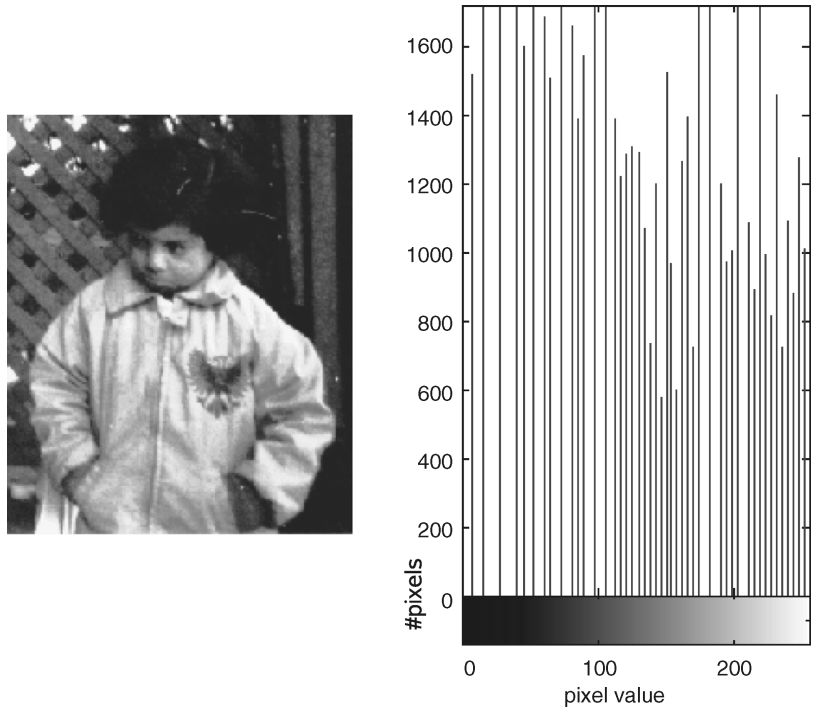

Figure 3.17 Histogram equalization applied to sample image

distribution over the full quantization range of the image (in comparison with the inputs of Figures 3.15 (left) and 3.16 (left)). Notably, the resulting histogram of Figure 3.17 does not contain all equal histogram entries, as discussed in Sections 3.4.4.1 and 3.4.4.2 – *this is a common misconception of histogram equalization*.

3.4.5 Histogram matching

Despite its attractive features, histogram equalization is certainly no panacea. There are many instances in which equalization produces quite undesirable effects. A closely related technique known as histogram matching (also known as *histogram specification*) is sometimes more appropriate and can be employed both as a means for improving visual contrast and for regularizing an image prior to subsequent processing or analysis.

The idea underpinning histogram matching is very simple. Given the original (input) image I_{input} and its corresponding histogram $p_x(x)$, we seek to effect a transformation $f(x)$ on the input intensity values such that the transformed (output) image I_{output} has a desired (target) histogram $p_z(z)$. Often, the target histogram will be taken from a model or 'typical' image of a similar kind.

3.4.5.1 Histogram matching theory

For the moment, we will gloss over the question of how to obtain the target histogram and simply assume it to be known. Our matching task, then, can be formulated as follows.

Let the input intensity levels be denoted by the variable $x(0 \leq x \leq 1)$ and the PDF (i.e. the continuous, normalized version of the histogram) of the input image be denoted by $p_x(x)$. Similarly, let the output (matched) intensity levels be denoted by the variable $z(0 \leq z \leq 1)$ and the corresponding PDF of the input image be denoted by $p_z(z)$. Our histogram-matching task can then be formulated as the derivation of a mapping $f(x)$ between the input intensities x and the output intensities z such that the mapped output intensity values have the desired PDF $p_z(z)$. We can approach this problem as follows.

The cumulative distribution (CDF) $C_x(x)$ of the input image is, by definition, given by the integral of the input PDF:

$$C_x(x) = \int_0^x p_{x'}(x')\,\mathrm{d}x' \tag{3.21}$$

Similarly, the CDF of the output image $C_z(z)$ is given by the integral of the output PDF:

$$C_z(z) = \int_0^z p_{z'}(z')\,\mathrm{d}z' \tag{3.22}$$

A key point in our reasoning is the recognition that both of these transforms are invertible. Specifically, an arbitrary PDF $f(x)$ is related to its CDF $p(X)$ by the relation

$$p(x) = \left.\frac{\mathrm{d}P}{\mathrm{d}X}\right|_{X=x} \tag{3.23}$$

Thus, knowledge of a CDF uniquely determines the PDF; and if the CDF is known, then the PDF is also known and calculable through this explicit differential. It follows that if we can define a mapping $f(x)$ between input intensities (x) and output intensities (z) such that the input and output CDFs are identical, we thereby guarantee that the corresponding PDFs will be the same. Accordingly, we demand that the CDFs defined previously, $C_x(x)$ and $C_z(z)$, be equal for the mapping $z = f(x)$:

$$C_z[f(x)] = C_x(x) \tag{3.24}$$

from which it follows that the required mapping $f()$ is

$$f(x) = C_z^{-1}[C_x(x)] \tag{3.25}$$

The definition of this mapping is fundamental to the ability to map input $C_x(x)$ to $C_z(z)$ and, hence, input PDF $p_x(x)$ to output PDF $p_z(z)$.

3.4.5.2 Histogram matching theory: discrete case

In general, the inverse mapping C^{-1} defined previously in Section 3.4.5.1 is not an analytic function and we must resort to numerical techniques to approximate the required mapping.

To effect a discrete approximation of the matching of two arbitrary image histograms, we proceed by first calculating the discrete CDF $C_x(k)$ of the input image:

$$C_x(k) = \sum_{j=0}^{k} p_x(j) = \frac{1}{N} \sum_{j=0}^{k} x_j \tag{3.26}$$

where x_j is the population of the jth intensity level in the input image, N is the total number of pixels in that image and $k = \{0, 1, 2, \ldots, L-1\}$, where L is the number of possible grey levels for the image.

Second, in a similar manner, we calculate the discrete CDF $C_z(l)$ of the output image:

$$C_z(l) = \sum_{j=0}^{l} p_z(j) = \frac{1}{N} \sum_{j=0}^{l} z_j \tag{3.27}$$

where z_j is the population of the jth intensity level in the output image, N is the total number of pixels and $l = \{0, 1, 2, \ldots, L-1\}$, where L is the number of possible grey levels for the image as before. Finally, we effect the discrete version of the earlier transform $f(x) = C_z^{-1}[C_x(x)]$ and must find the mapping defined by C_z^{-1} when the input to it is the CDF of the input image, $C_x()$. Like the earlier example on histogram equalization (Section 3.4.4), this mapping effectively defines a look-up table between the input and output image pixel values which can be readily and efficiently computed.

3.4.5.3 Histogram matching in practice

Histogram matching extends the principle of histogram equalization by generalizing the form of the target histogram. It is an automatic enhancement technique in which the required transformation is derived from a (user-) specified target histogram distribution.

In practice, the target histogram distribution t will be extracted from an existing reference image or will correspond to a specified mathematical function with the correct properties. In Matlab, histogram matching can be achieved as in Example 3.20. In this example, we specify a linearly ramped PDF as the target distribution t and we can see the resulting output together with the modified image histograms in Figure 3.18.

Example 3.20

Matlab code	**What is happening?**
I=imread('pout.tif');	
pz=0:255;	%Define ramp-like pdf as desired output histogram
Im=histeq(I, pz);	%Supply desired histogram to perform matching
subplot(2,3,1), imshow(I);	%Display image
subplot(2,3,2), imshow(Im);	%Display result
subplot(2,3,3), plot(pz);	%Display distribution t
subplot(2,3,4), imhist(I);	%Display histogram of image
subplot(2,3,5), imhist(Im);	%Display histogram of result

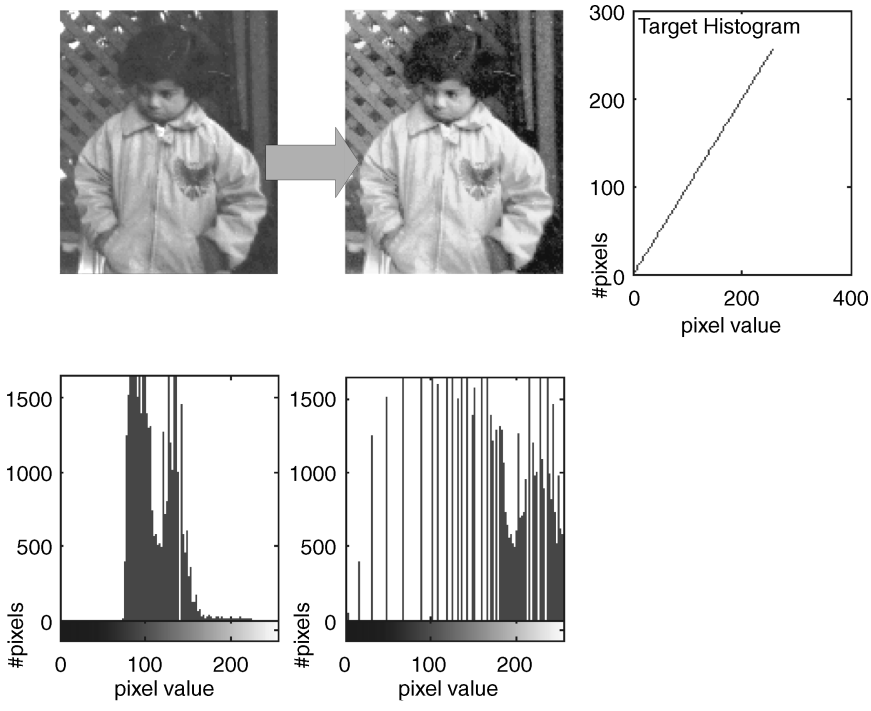

Figure 3.18 Histogram matching applied to sample image

It is evident from Figure 3.18 that the desired histogram of the output image (in this case chosen arbitrarily) does not produce an output with particularly desirable characteristics above that of the histogram equalization shown in Figure 3.17. In general, choice of an appropriate target histogram presupposes knowledge of what structures in the input image are present and need enhancing. One situation in which the choice of output histogram may be essentially fixed is when the image under consideration is an example of a given class (and in which the intensity distribution should thus be constrained to a specific form) but has been affected by extraneous factors. One example of this is a sequence of faces in frontal pose. The histogram of a given face may be affected by illumination intensity and by shadowing effects. A face captured under model/ideal conditions could supply the desired histogram to correct these illumination and shadowing effects prior to subsequent use of the transformed input image for facial recognition.

3.4.6 Adaptive histogram equalization

Sometimes the overall histogram of an image may have a wide distribution whilst the histogram of local regions is highly skewed towards one end of the grey spectrum. In such cases, it is often desirable to enhance the contrast of these local regions, but global histogram equalization is ineffective. This can be achieved by adaptive histogram equalization. The term adaptive implies that different regions of the image are processed differently (i.e. different look-up tables are applied) depending on local properties.

There are several variations on adaptive histogram equalization, but perhaps the simplest and most commonplace is the so-called sliding window (or tile-based) approach. In this method, the underlying image is broken down into relatively small contiguous 'tiles' or local $N \times M$ neighbourhood regions (e.g. 16×16 pixels). Each tile or inner window is surrounded by a larger, outer window which is used to calculate the appropriate histogram equalization look-up table for the inner window.

This is generally effective in increasing local contrast, but 'block artefacts' can occur as a result of processing each such region in isolation and artefacts at the boundaries between the inner windows can tend to produce the impression of an image consisting of a number of slightly incongruous blocks. The artefacts can generally be reduced by increasing the size of the outer window relative to the inner window.

An alternative method for adaptive histogram equalization (attributed to Pizer) has been applied successfully, the steps of which are as follows:

- A regular grid of points is superimposed over the image. The spacing of the grid points is a variable in this approach, but is generally a few tens of pixels.

- For each and every grid point, a rectangular window with twice the grid spacing is determined. A given window thus has a 50% overlap with its immediate neighbours to north, south, east and west.

- A histogram-equalized look-up table is calculated for each such window. Owing to the 50% overlap between the windows, every pixel within the image lies within four adjacent, rectangular windows or four neighbourhoods.

- The transformed value of a given image pixel is calculated as a weighted combination of the output values from the four neighbourhood look-up tables using the following bilinear formula:

$$I = (1-a)(1-b)I_1 + a(1-b)I_2 + (1-a)bI_3 + abI_4 \tag{3.28}$$

  for distance weights $0 \leq a, b \leq 1$ and histogram-equalized look-up table values I_1, I_2, I_3 and I_4 from each of the four adjacent neighbourhoods. This methodology is illustrated further in Figure 3.19. Note how this formula gives appropriate importance to the neighbourhoods to which the point of calculation I most fully belongs; e.g. a pixel located precisely on a grid point derives its equalized value from its surrounding neighbourhood only, whereas a point which is equidistant from its four nearest grid points is an equally weighted combination of all four surrounding neighbourhood values. It is important to remember the overlapping nature of the neighbourhoods when considering Figure 3.19.

A final extension to this method is the contrast-limited adaptive histogram equalization approach. In general terms, calculating local region histograms and equalizing the intensity values within those regions has a tendency to increase the contrast too much and amplify noise. Some regions within an image are inherently smooth with little real contrast and blindly equalizing the intensity histogram in the given region can have undesirable results.

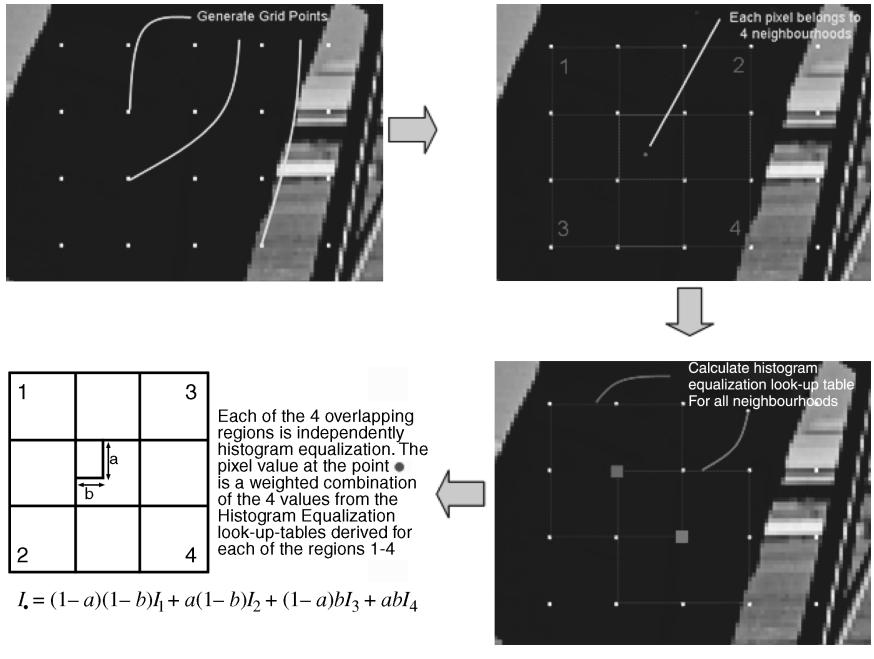

Figure 3.19 Adaptive histogram equalization multi-region calculation approach

The main idea behind the use of contrast limiting is to place a limit l, $0 \leq l \leq 1$, on the overall (normalized) increase in contrast applied to any given region.

This extended method of contrast-limited adaptive histogram equalization is available in Matlab as in Example 3.21. The result of Example 3.21 is shown in Figure 3.20, where we see the differing effects of the specified contrast limit l and target distribution t on the resulting adaptively equalized image. Adaptive histogram equalization can sometimes effect significant improvements in local image contrast. However, owing to the fundamentally different nature of one image from the next, the ability to vary the number of tiles into which the image is decomposed and the specific form of the target probability distribution, there is little general theory to guide us. Obtaining an image with the desired contrast characteristics is thus, to a considerable degree, an art in which these parameters are varied on an experimental basis.

Example 3.21

Matlab code	**What is happening?**
I=imread('pout.tif');	%Read in image
I1=adapthisteq(I,'clipLimit',0.02,'Distribution','rayleigh');	
I2=adapthisteq(I,'clipLimit',0.02,'Distribution','exponential');	
I3=adapthisteq(I,'clipLimit',0.08,'Distribution','uniform');	
subplot(2,2,1), imshow(I); subplot(2,2,2), imshow(I2);	%Display orig. + output
subplot(2,2,3), imshow(I2); subplot(2,2,4), imshow(I3);	%Display outputs

Comments

- Here we use the ***adapthisteq()*** function to perform this operation with a contrast limit l (parameter 'clipLimit'=0.02/0.08) set accordingly. In addition, the Matlab implementation of adaptive histogram equalization also allows the user to specify a target distribution t for use with every region in the image (in the style of histogram matching, Section 3.4.5). Standard equalization is performed with the specification of (the default) *uniform* distribution.
- By default, the Matlab implementation uses an 8×8 division of the image into windows/neighbourhoods for processing. This, like the other parameters in this example, can be specified as named ('parameter name', value) pairs as inputs to the function. Please refer to the Matlab documentation on ***adapthisteq()*** for further details (*doc adapthisteq* at the Matlab command prompt).

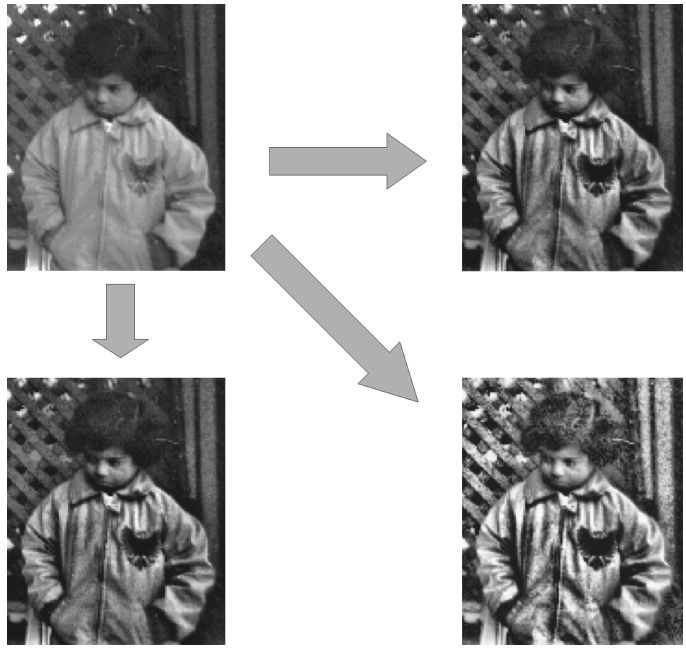

Figure 3.20 Adaptive histogram equalization applied to a sample image

3.4.7 Histogram operations on colour images

Until this point, we have only considered the application of our contrast manipulation operations on single-channel, grey-scale images. Attempting to improve the contrast of colour images is a slightly more complex issue than for grey-scale intensity images. At first glance, it is tempting to consider application of histogram equalization or matching independently to each of the three channels (R,G,B) of the true colour image. However, the RGB values of a pixel determine both its intensity and its chromaticity (i.e. the subjective impression of colour). Transformation of the RGB values of the pixels to improve contrast will, therefore, in general, alter the chromatic (i.e. the colour hue) content of the image.

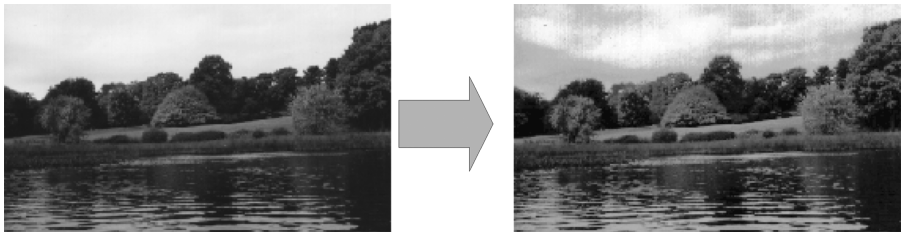

Figure 3.21 Adaptive histogram equalization applied to a sample colour image *(See colour plate section for colour version)*

The solution is first to transform the colour image to an alternative, perceptual colour model, such as HSV, in which the luminance component (intensity V) is decoupled from the chromatic (H and S) components which are responsible for the subjective impression of colour. In order to perform such histogram operations on colour images, we thus (a) transform the RGB component image to the HSV representation (hue, saturation, variance), (b) apply the histogram operation to the intensity component and finally (c) convert the result back to the RGB colour space as required. The HSV colour model space is not unique in this sense and there are actually several colour models (e.g. L^*a^*b) that we could use to achieve this. HSV is used as our example following on from its introduction in Chapter 1.

Using such a colour space conversion histogram operations can be applied in Matlab as in Example 3.22. The result of Example 3.22 is shown in Figure 3.21, where we see the application of histogram equalization (Section 3.4.4.3, Example 3.19) to an RGB colour image using an intermediate HSV colour space representation. Note that the subjective impression of the colour (chromaticity) has been maintained, yet the effect of the histogram equalization of the intensity component is evident in the result.

Example 3.22

Matlab code	**What is happening?**
I=imread('autumn.tif ');	%Read in image
Ihsv=rgb2hsv(I);	%Convert original to HSV image, I2
V=histeq(Ihsv(:,:,3));	%Histogram equalise V (3rd) channel of I2
Ihsv(:,:,3)=V;	%Copy equalized V plane into (3rd) channel I2
Iout=hsv2rgb(Ihsv);	%Convert I2 back to RGB form
subplot(1,2,1), imshow(I);	
subplot(1,2,2), imshow(Iout);	

Comments

- The functions ***rgb2hsv*** *and* ***hsv2rgb*** *are used to convert between the RGB and HSV colour spaces.*
- Individual colour channels of an image *I* can be referenced as 'I(:,:,n)' in Matlab syntax, where n is the numerical channel number in range $n = \{1,2,3\} = \{R,G,B\} = \{H,S,V\}$ and similarly for other colour space and image representations.

Exercises

The following exercises are designed to reinforce and develop the concepts and Matlab examples introduced in this chapter. Additional information on all of the Matlab functions represented in this chapter and throughout these exercises is available in Matlab from the function help browser (use *doc <function name>* at the Matlab command prompt, where *<function name>* is the function required)

Matlab functions: ***imresize***, ***size***, ***whos***, ***imadd***, ***imsubtract***, ***immultiply***, ***imagesc***, ***imcomplement***, ***imabsdiff***, ***implay***, ***uint8***, ***horzcat***, ***tic***, ***toc***, ***rgb2ycbcr***.

Exercise 3.1 Using the examples presented on arithmetic operations (Examples 3.1–3.4) we can investigate the use of these operators in a variety of ways. First, load the Matlab example images 'rice.png' and 'cameraman.tif'. Investigate the combined use of the Matlab ***imresize()*** and ***size()*** functions to resize one of the images to be the same size as the other. The Matlab command ***whos*** used in the syntax '***whos*** v' will display information about the size and type of a given image or other variable *v*.

Try adding both of these images together using the standard Matlab addition operator '+' and displaying the result. What happens? Now add them together by using the ***imadd()*** function but adding a third parameter to the function of 'uint16' or 'double' with quotes that forces an output in a 16-bit or floating-point double-precision data type. You will need to display the output using the ***imagesc()*** function. How do these two addition operator results differ and why do they differ?

Repeat this exercise using the standard Matlab subtraction operator '−' and also by using the ***imsubtract()*** function.

Exercise 3.2 Building upon the logical inversion example presented in Example 3.6, investigate the application of the ***imcomplement()*** function to different image types and different applications. First load the example image 'peppers.png' and apply this operator. What effect do you see and to what aspect of traditional photography does the resulting image colouring relate?

Image inversion is also of particular use in medical imaging for highlighting different aspects of a given image. Apply the operator to the example images 'mir.tif', 'spine.tif' and cell image 'AT3_1m4_09.tif'. What resulting effects do you see in the transformed images which may be beneficial to a human viewer?

Exercise 3.3 Use the sequence of cell images ('AT3_1m4_01.tif', 'AT3_1m4_02.tif, ... 'AT3_1m4_09.tif', 'AT3_1m4_10.tif') provided in combination with the Matlab ***imabsdiff()*** function and a Matlab ***for*** loop construct to display an animation of the differences between images in the sequence.

You may wish to use an additional enhancement approach to improve the dynamic range of difference images that result from the ***imabsdiff()*** function. What is result of this differencing operation? How could this be useful?

Hint. You may wish to set up an array containing each of the image file names 01 to 10. The animation effect can be achieved by updating the same figure for each set of differences (e.g. between the kth and $(k-1)$th images in the sequence) or by investigating the Matlab ***implay()*** function to play back an array of images.

Exercise 3.4 Using a combination of the Matlab ***immultiply()*** and ***imadd()*** functions implement a Matlab function (or sequence of commands) for blending two images A and B into a single image with corresponding blending weights w_A and w_B such that output image C is

$$C = w_A A + w_B B \tag{3.29}$$

Experiment with different example images and also with blending information from a sequence of images (e.g. cell images from Exercise 3.3). How could such a technique be used in a real application?

Exercise 3.5 Using the thresholding technique demonstrated in Example 3.7 and with reference to the histogram display functions of Example 3.12, manually select and apply a threshold to the example image 'pillsetc.png'. Compare your result with the adaptive threshold approaches shown in Examples 3.15 and 3.16. Which is better? (Also, which is easier?)

Repeat this procedure to isolate the foreground items in example images 'tape.png', 'coins.png' and 'eight.tif'. Note that images require to be transformed to grey scale (see Section 1.4.1.1) prior to thresholding.

Exercise 3.6 Looking back to Examples 3.15 and 3.16, investigate the effects of varying the constant offset parameter C when applying it to example images 'cameraman.tif' and 'coins.png'. How do the results for these two images differ?

Implement the third method of adaptive thresholding from Section 3.4.2 using the threshold $t = \text{floor}((\max - \min)/2) + C$ method. Compare this approach against the examples already provided for thresholding the previous two images and other available example images.

Exercise 3.7 Read in the example images 'cameraman.tif' and 'circles.png' and convert the variable resulting from the latter into the unsigned 8-bit type of the former using the Matlab casting function ***uint8()***. Concatenate these two images into a single image (using the Matlab function ***horzcat()***) and display it. Why can you not see the circles in the resulting image? (Try also using the Matlab ***imagesc*** function).

Using the logarithmic transform (Example 3.8), adjust the dynamic range of this concatenated image so that both the outline of the cameraman's jacket and the outline of the circles are just visible (parameter $C > 10$ will be required). By contrast, investigate the use of both histogram equalization and adaptive histogram equalization on this concatenated image. Which approach gives the best results for overall image clarity and why is this approach better for this task?

Exercise 3.8 Consider the Matlab example image 'mandi.tif', where we can see varying lighting across both the background and foreground. Where is information not clearly visible within this image? What are the properties of these regions in terms of pixel values?

Consider the corresponding properties of the logarithmic and exponential transforms (Sections 3.3.1 and 3.3.2) and associated Examples 3.8 and 3.9. Experiment with both of

these transforms and determine a suitable choice (with parameters) for enhancing this image. Note that this is large image example and processing may take a few seconds (use the Matlab functions ***tic*** and ***toc*** to time the operation).

Contrast the results obtained using these two transforms to applying histogram equalization contrast stretch (Section 3.4.4) or adaptive histogram equalization (Section 3.4.6) to this image.

Exercise 3.9 Based on the grey-scale gamma correction presented in Example 3.11, apply gamma correction to a colour image using the example 'autumn.tif'. Why can we (and why would we) apply this technique directly to the RGB image representation and not an alternative HSV representation as per the colour image histogram processing of Example 3.22?

Exercise 3.10 Based on Example 3.22, where we apply histogram equalization to a colour image, apply contrast stretching (Section 3.4.3, Example 3.17) to the colour example image 'westconcordaerial.png' using the same approach. Experiment with different parameter values to find an optimum for the visualization of this image. Compare the application of this approach with histogram equalization and adaptive histogram equalization on the same image. Which produces the best result? Which approach is more 'tunable' and which is more automatic in nature?

Repeat this for the Matlab image example 'peppers.png'. Do the results differ significantly for this example?

Exercise 3.11 Using the various contrast enhancement and dynamic range manipulation approaches presented throughout this chapter, investigate and make a selection of transforms to improve the contrast in the example image 'AT3_1m4_01.tif'. Once you have achieved a suitable contrast for this example image, extract the corresponding histogram distribution of the image (Example 3.13) and apply it to the other images in the sequence ('AT3_1m4_02.tif', ... 'AT3_1m4_09.tif', 'AT3_1m4_10.tif') using histogram matching (Example 3.20). Are the contrast settings determined for the initial example image suitable for all of the others? What if the images where not all captured under the same conditions?

Here, we see the use of histogram matching as a method for automatically setting the dynamic range adjustment on a series of images based on the initial determination of suitable settings for one example. Where else could this be applied?

Exercise 3.12 In contrast to the approach shown for colour histogram equalization shown in Example 3.22, perform histogram equalization on each of the (R,G,B) channels of an example colour image. Compare the results with that obtained using the approach of Example 3.22. What is the difference in the resulting images?

Repeat this for the operations of contrast stretching and adaptive histogram equalization. Investigate also using the YCbCr colour space (Matlab function ***rgb2ybcr()***) for performing colour histogram operations on images. Which channel do you need to use? What else do you need to consider?

For further examples and exercises see http://www.fundipbook.com