1
Representation

In this chapter we discuss the representation of images, covering basic notation and information about images together with a discussion of standard image types and image formats. We end with a practical section, introducing Matlab's facilities for reading, writing, querying, converting and displaying images of different image types and formats.

1.1 What is an image?

A digital image can be considered as a discrete representation of data possessing both spatial (layout) and intensity (colour) information. As we shall see in Chapter 5, we can also consider treating an image as a multidimensional signal.

1.1.1 Image layout

The two-dimensional (2-D) discrete, digital image $I(m, n)$ represents the response of some sensor (or simply a value of some interest) at a series of fixed positions ($m = 1, 2, \ldots, M$; $n = 1, 2, \ldots, N$) in 2-D Cartesian coordinates and is derived from the 2-D continuous spatial signal $I(x, y)$ through a sampling process frequently referred to as discretization. Discretization occurs naturally with certain types of imaging sensor (such as CCD cameras) and basically effects a local averaging of the continuous signal over some small (typically square) region in the receiving domain.

The indices m and n respectively designate the rows and columns of the image. The individual picture elements or pixels of the image are thus referred to by their 2-D (m, n) index. Following the Matlab® convention, $I(m, n)$ denotes the response of the pixel located at the mth row and nth column starting from a top-left image origin (see Figure 1.1). In other imaging systems, a column–row convention may be used and the image origin in use may also vary.

Although the images we consider in this book will be discrete, it is often theoretically convenient to treat an image as a continuous spatial signal: $I(x, y)$. In particular, this sometimes allows us to make more natural use of the powerful techniques of integral and differential calculus to understand properties of images and to effectively manipulate and

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

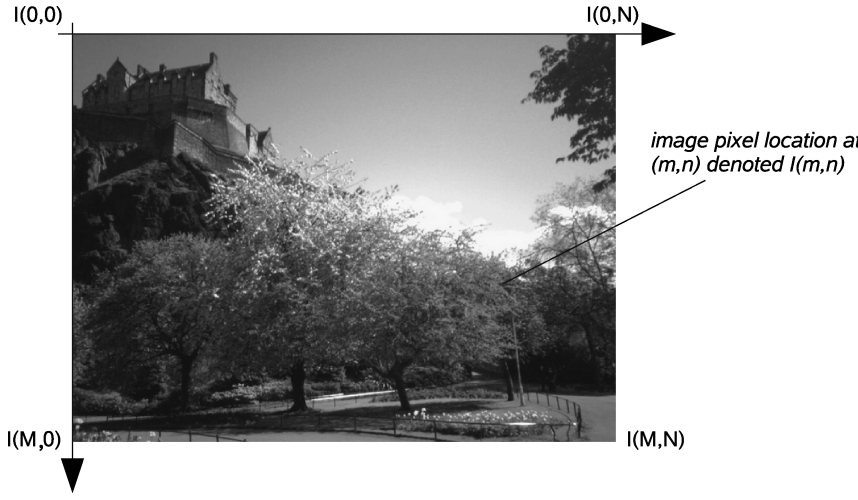

Figure 1.1 The 2-D Cartesian coordinate space of an M x N digital image

process them. Mathematical analysis of discrete images generally leads to a linear algebraic formulation which is better in some instances.

The individual pixel values in most images do actually correspond to some physical response in real 2-D space (e.g. the optical intensity received at the image plane of a camera or the ultrasound intensity at a transceiver). However, we are also free to consider images in abstract spaces where the coordinates correspond to something other than physical space and we may also extend the notion of an image to three or more dimensions. For example, medical imaging applications sometimes consider full three-dimensional (3-D) reconstruction of internal organs and a time sequence of such images (such as a beating heart) can be treated (if we wish) as a single four-dimensional (4-D) image in which three coordinates are spatial and the other corresponds to time. When we consider 3-D imaging we are often discussing spatial volumes represented by the image. In this instance, such 3-D pixels are denoted as voxels (volumetric pixels) representing the smallest spatial location in the 3-D volume as opposed to the conventional 2-D image.

Throughout this book we will usually consider 2-D digital images, but much of our discussion will be relevant to images in higher dimensions.

1.1.2 Image colour

An image contains one or more colour channels that define the intensity or colour at a particular pixel location $I(m, n)$.

In the simplest case, each pixel location only contains a single numerical value representing the signal level at that point in the image. The conversion from this set of numbers to an actual (displayed) image is achieved through a colour map. A colour map assigns a specific shade of colour to each numerical level in the image to give a visual representation of the data. The most common colour map is the greyscale, which assigns all shades of grey from black (zero) to white (maximum) according to the signal level. The

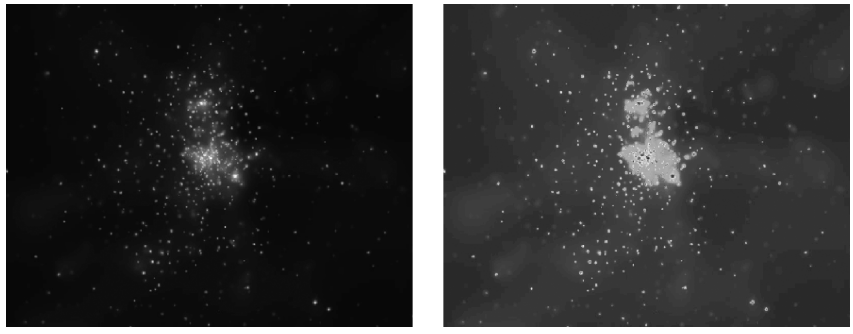

Figure 1.2 Example of grayscale (left) and false colour (right) image display *(See colour plate section for colour version)*

greyscale is particularly well suited to intensity images, namely images which express only the intensity of the signal as a single value at each point in the region.

In certain instances, it can be better to display intensity images using a false-colour map. One of the main motives behind the use of false-colour display rests on the fact that the human visual system is only sensitive to approximately 40 shades of grey in the range from black to white, whereas our sensitivity to colour is much finer. False colour can also serve to accentuate or delineate certain features or structures, making them easier to identify for the human observer. This approach is often taken in medical and astronomical images.

Figure 1.2 shows an astronomical intensity image displayed using both greyscale and a particular false-colour map. In this example the *jet* colour map (as defined in Matlab) has been used to highlight the structure and finer detail of the image to the human viewer using a linear colour scale ranging from dark blue (low intensity values) to dark red (high intensity values). The definition of colour maps, i.e. assigning colours to numerical values, can be done in any way which the user finds meaningful or useful. Although the mapping between the numerical intensity value and the colour or greyscale shade is typically linear, there are situations in which a nonlinear mapping between them is more appropriate. Such nonlinear mappings are discussed in Chapter 4.

In addition to greyscale images where we have a single numerical value at each pixel location, we also have true colour images where the full spectrum of colours can be represented as a triplet vector, typically the (R,G,B) components at each pixel location. Here, the colour is represented as a linear combination of the basis colours or values and the image may be considered as consisting of three 2-D planes. Other representations of colour are also possible and used quite widely, such as the (H,S,V) (hue, saturation and value (or intensity)). In this representation, the intensity V of the colour is decoupled from the chromatic information, which is contained within the H and S components (see Section 1.4.2).

1.2 Resolution and quantization

The size of the 2-D pixel grid together with the data size stored for each individual image pixel determines the spatial resolution and colour quantization of the image.

The representational power (or size) of an image is defined by its resolution. The resolution of an image source (e.g. a camera) can be specified in terms of three quantities:

- *Spatial resolution* The column (C) by row (R) dimensions of the image define the number of pixels used to cover the visual space captured by the image. This relates to the sampling of the image signal and is sometimes referred to as the pixel or digital resolution of the image. It is commonly quoted as $C \times R$ (e.g. 640×480, 800×600, 1024×768, etc.)

- *Temporal resolution* For a continuous capture system such as video, this is the number of images captured in a given time period. It is commonly quoted in frames per second (fps), where each individual image is referred to as a video frame (e.g. commonly broadcast TV operates at 25 fps; 25–30 fps is suitable for most visual surveillance; higher frame-rate cameras are available for specialist science/engineering capture).

- *Bit resolution* This defines the number of possible intensity/colour values that a pixel may have and relates to the *quantization* of the image information. For instance a binary image has just two colours (black or white), a grey-scale image commonly has 256 different grey levels ranging from black to white whilst for a colour image it depends on the colour range in use. The bit resolution is commonly quoted as the number of binary bits required for storage at a given quantization level, e.g. binary is 2 bit, grey-scale is 8 bit and colour (most commonly) is 24 bit. The range of values a pixel may take is often referred to as the *dynamic range* of an image.

It is important to recognize that the bit resolution of an image does not necessarily correspond to the resolution of the originating imaging system. A common feature of many cameras is automatic gain, in which the minimum and maximum responses over the image field are sensed and this range is automatically divided into a convenient number of bits (i.e. digitized into N levels). In such a case, the bit resolution of the image is typically less than that which is, in principle, achievable by the device.

By contrast, the blind, unadjusted conversion of an analog signal into a given number of bits, for instance $2^{16} = 65\ 536$ discrete levels, does not, of course, imply that the true resolution of the imaging device as a whole is actually 16 bits. This is because the overall level of *noise* (i.e. random fluctuation) in the sensor and in the subsequent processing chain may be of a magnitude which easily exceeds a single digital level. The sensitivity of an imaging system is thus fundamentally determined by the noise, and this makes noise a key factor in determining the number of quantization levels used for digitization. There is no point in digitizing an image to a high number of bits if the level of noise present in the image sensor does not warrant it.

1.2.1 Bit-plane splicing

The visual significance of individual pixel bits in an image can be assessed in a subjective but useful manner by the technique of bit-plane splicing.

To illustrate the concept, imagine an 8-bit image which allows integer values from 0 to 255. This can be conceptually divided into eight separate image planes, each corresponding

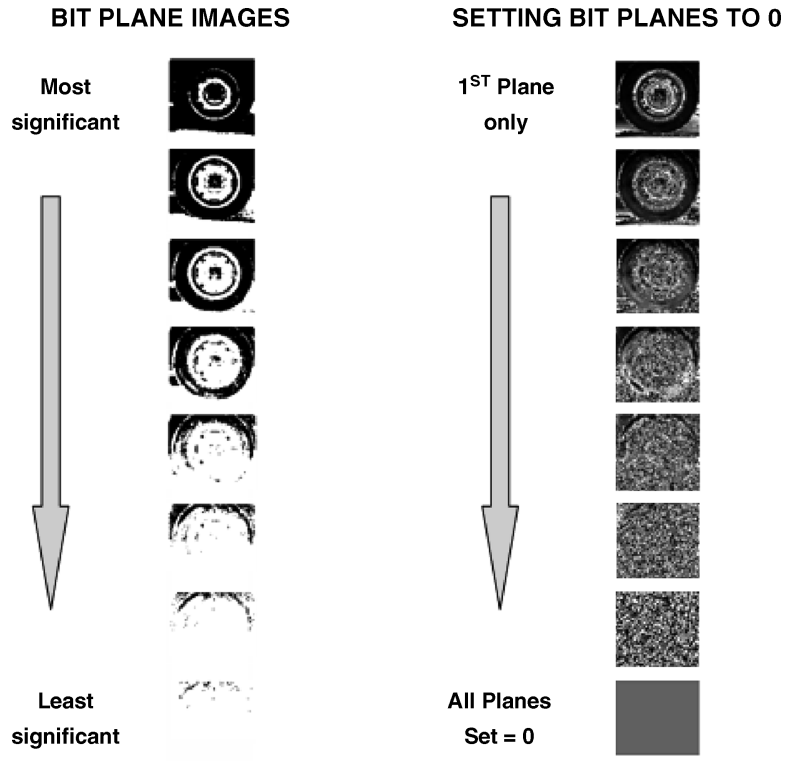

Figure 1.3 An example of bit-plane slicing a grey-scale image

to the values of a given bit across all of the image pixels. The first bit plane comprises the first and most significant bit of information (intensity = 128), the second, the second most significant bit (intensity = 64) and so on. Displaying each of the bit planes in succession, we may discern whether there is any visible structure in them.

In Figure 1.3, we show the bit planes of an 8-bit grey-scale image of a car tyre descending from the most significant bit to the least significant bit. It is apparent that the two or three least significant bits do not encode much useful visual information (it is, in fact, mostly noise). The sequence of images on the right in Figure 1.3 shows the effect on the original image of successively setting the bit planes to zero (from the first and most significant to the least significant). In a similar fashion, we see that these last bits do not appear to encode any visible structure. In this specific case, therefore, we may expect that retaining only the five most significant bits will produce an image which is practically visually identical to the original. Such analysis could lead us to a more efficient method of encoding the image using fewer bits – a method of *image compression*. We will discuss this next as part of our examination of image storage formats.

1.3 Image formats

From a mathematical viewpoint, any meaningful 2-D array of numbers can be considered as an image. In the real world, we need to effectively display images, store them (preferably

Table 1.1 Common image formats and their associated properties

Acronym	Name	Properties
GIF	Graphics interchange format	Limited to only 256 colours (8-bit); lossless compression
JPEG	Joint Photographic Experts Group	In most common use today; lossy compression; lossless variants exist
BMP	Bit map picture	Basic image format; limited (generally) lossless compression; lossy variants exist
PNG	Portable network graphics	New lossless compression format; designed to replace GIF
TIF/TIFF	Tagged image (file) format	Highly flexible, detailed and adaptable format; compressed/uncompressed variants exist

compactly), transmit them over networks and recognize bodies of numerical data as corresponding to images. This has led to the development of standard digital image formats. In simple terms, the image formats comprise a file header (containing information on how exactly the image data is stored) and the actual numeric pixel values themselves. There are a large number of recognized image formats now existing, dating back over more than 30 years of digital image storage. Some of the most common 2-D image formats are listed in Table 1.1. The concepts of lossy and lossless compression are detailed in Section 1.3.2.

As suggested by the properties listed in Table 1.1, different image formats are generally suitable for different applications. GIF images are a very basic image storage format limited to only 256 grey levels or colours, with the latter defined via a colour map in the file header as discussed previously. By contrast, the commonplace JPEG format is capable of storing up to a 24-bit RGB colour image, and up to 36 bits for medical/scientific imaging applications, and is most widely used for consumer-level imaging such as digital cameras. Other common formats encountered include the basic bitmap format (BMP), originating in the development of the Microsoft Windows operating system, and the new PNG format, designed as a more powerful replacement for GIF. TIFF, tagged image file format, represents an overarching and adaptable file format capable of storing a wide range of different image data forms. In general, photographic-type images are better suited towards JPEG or TIF storage, whilst images of limited colour/detail (e.g. logos, line drawings, text) are best suited to GIF or PNG (as per TIFF), as a lossless, full-colour format, is adaptable to the majority of image storage requirements.

1.3.1 Image data types

The choice of image format used can be largely determined by not just the image contents, but also the actual image data type that is required for storage. In addition to the bit resolution of a given image discussed earlier, a number of distinct image types also exist:

- *Binary images* are 2-D arrays that assign one numerical value from the set $\{0, 1\}$ to each pixel in the image. These are sometimes referred to as logical images: black corresponds

to zero (an 'off' or 'background' pixel) and white corresponds to one (an 'on' or 'foreground' pixel). As no other values are permissible, these images can be represented as a simple bit-stream, but in practice they are represented as 8-bit integer images in the common image formats. A fax (or facsimile) image is an example of a binary image.

- *Intensity or grey-scale images* are 2-D arrays that assign one numerical value to each pixel which is representative of the intensity at this point. As discussed previously, the pixel value range is bounded by the bit resolution of the image and such images are stored as N-bit integer images with a given format.

- *RGB or true-colour* images are 3-D arrays that assign three numerical values to each pixel, each value corresponding to the red, green and blue (RGB) image channel component respectively. Conceptually, we may consider them as three distinct, 2-D planes so that they are of dimension C by R by 3, where R is the number of image rows and C the number of image columns. Commonly, such images are stored as sequential integers in successive channel order (e.g. $R_0G_0B_0$, $R_1G_1B_1$, ...) which are then accessed (as in Matlab) by $I(C, R, \text{channel})$ coordinates within the 3-D array. Other colour representations which we will discuss later are similarly stored using the 3-D array concept, which can also be extended (starting numerically from 1 with Matlab arrays) to four or more dimensions to accommodate additional image information, such as an alpha (transparency) channel (as in the case of PNG format images).

- *Floating-point* images differ from the other image types we have discussed. By definition, they do not store integer colour values. Instead, they store a floating-point number which, within a given range defined by the floating-point precision of the image bit-resolution, represents the intensity. They may (commonly) represent a measurement value other than simple intensity or colour as part of a scientific or medical image. Floating point images are commonly stored in the TIFF image format or a more specialized, domain-specific format (e.g. medical DICOM). Although the use of floating-point images is increasing through the use of high dynamic range and stereo photography, file formats supporting their storage currently remain limited.

Figure 1.4 shows an example of the different image data types we discuss with an example of a suitable image format used for storage. Although the majority of images we will encounter in this text will be of integer data types, Matlab, as a general matrix-based data analysis tool, can of course be used to process floating-point image data.

1.3.2 Image compression

The other main consideration in choosing an image storage format is compression. Whilst compressing an image can mean it takes up less disk storage and can be transferred over a network in less time, several compression techniques in use exploit what is known as *lossy compression*. Lossy compression operates by removing redundant information from the image.

As the example of bit-plane slicing in Section 1.2.1 (Figure 1.3) shows, it is possible to remove some information from an image without any apparent change in its visual

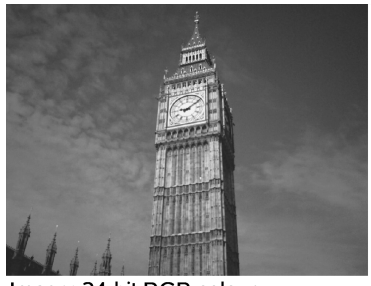

Image: 24-bit RGB colour
Pixel Data Type: 3 x integer (0→255)
Image Format: JPEG

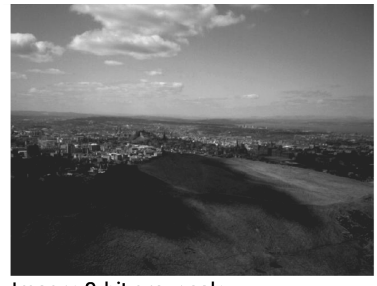

Image: 8-bit grayscale
Pixel Data Type: integer (0→255)
Image Format: GIF

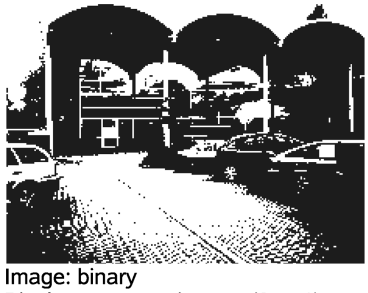

Image: binary
Pixel Data Type: integer (0 or 1)
Image Format: PNG

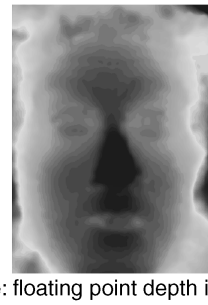

Image: floating point depth image Pixel Data Type: floating point values Image Format: TIFF (Copyright: Tim Lukins, UoE)

Figure 1.4 Examples of different image types and their associated storage formats

appearance. Essentially, if such information is visually redundant then its transmission is unnecessary for appreciation of the image. The form of the information that can be removed is essentially twofold. It may be in terms of fine image detail (as in the bit-slicing example) or it may be through a reduction in the number of colours/grey levels in a way that is not detectable by the human eye.

Some of the image formats we have presented, store the data in such a compressed form (Table 1.1). Storage of an image in one of the compressed formats employs various algorithmic procedures to reduce the raw image data to an equivalent image which appears identical (or at least nearly) but requires less storage. It is important to distinguish between compression which allows the original image to be reconstructed perfectly from the reduced data without any loss of image information (*lossless compression*) and so-called lossy compression techniques which reduce the storage volume (sometimes dramatically) at the expense of some loss of detail in the original image as shown in Figure 1.5, the lossless and lossy compression techniques used in common image formats can significantly reduce the amount of image information that needs to be stored, but in the case of lossy compression this can lead to a significant reduction in image quality.

Lossy compression is also commonly used in video storage due to the even larger volume of source data associated with a large sequence of image frames. This loss of information, itself a form of noise introduced into the image as *compression artefacts*, can limit the effectiveness of later image enhancement and analysis.

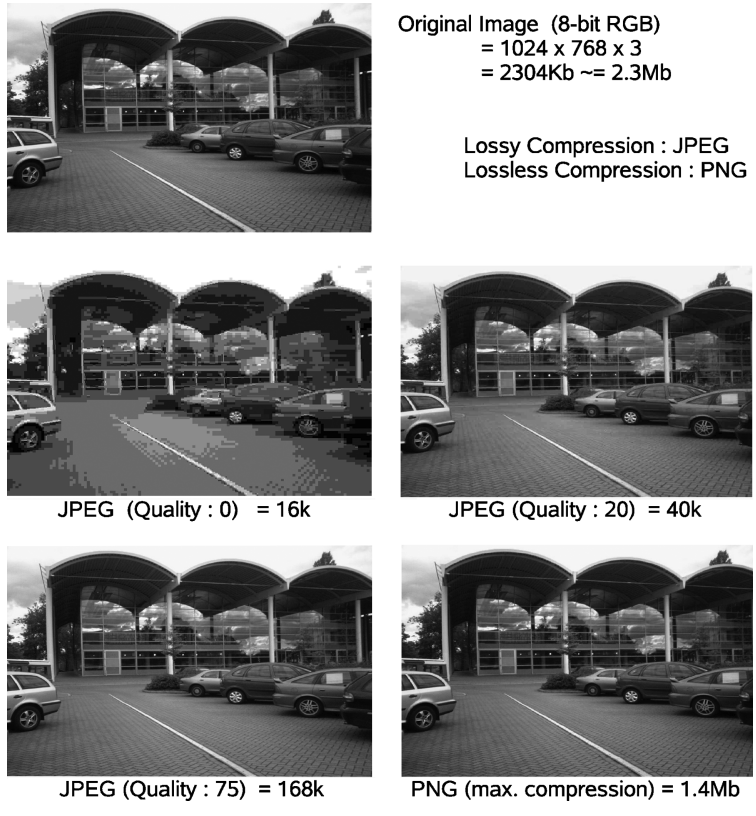

Figure 1.5 Example image compressed using lossless and varying levels of lossy compression *(See colour plate section for colour version)*

In terms of practical image processing in Matlab, it should be noted that an image written to file from Matlab in a lossy compression format (e.g. JPEG) will not be stored as the exact Matlab image representation it started as. Image pixel values will be altered in the image output process as a result of the lossy compression. This is not the case if a lossless compression technique is employed.

An interesting Matlab exercise is posed for the reader in Exercise 1.4 to illustrate this difference between storage in JPEG and PNG file formats.

1.4 Colour spaces

As was briefly mentioned in our earlier discussion of image types, the representation of colours in an image is achieved using a combination of one or more colour channels that are combined to form the colour used in the image. The representation we use to store the colours, specifying the number and nature of the colour channels, is generally known as the colour space.

Considered as a mathematical entity, an image is really only a spatially organized set of numbers with each pixel location addressed as $I(C, R)$. Grey-scale (intensity) or binary images are 2-D arrays that assign one numerical value to each pixel which is representative of

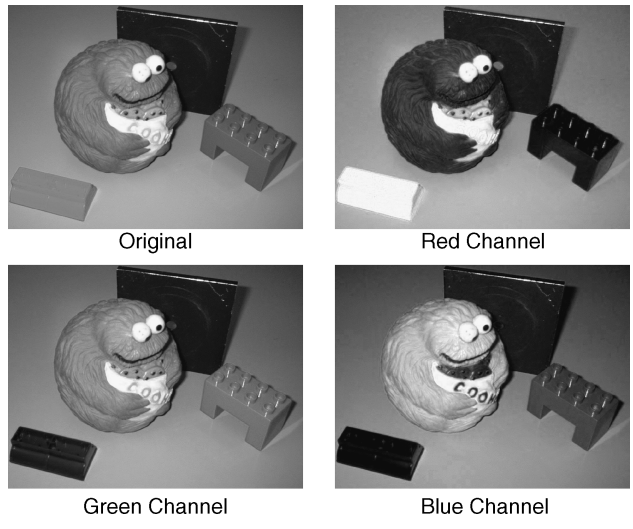

Figure 1.6 Colour RGB image separated into its red (R), green (G) and blue (B) colour channels *(See colour plate section for colour version)*

the intensity at that point. They use a single-channel colour space that is either limited to a 2-bit (binary) or intensity (grey-scale) colour space. By contrast, RGB or true-colour images are 3-D arrays that assign three numerical values to each pixel, each value corresponding to the red, green and blue component respectively.

1.4.1 RGB

RGB (or true colour) images are 3-D arrays that we may consider conceptually as three distinct 2-D planes, one corresponding to each of the three red (R), green (G) and blue (B) colour channels. RGB is the most common colour space used for digital image representation as it conveniently corresponds to the three primary colours which are mixed for display on a monitor or similar device.

We can easily separate and view the red, green and blue components of a true-colour image, as shown in Figure 1.6. It is important to note that the colours typically present in a real image are nearly always a blend of colour components from all three channels. A common misconception is that, for example, items that are perceived as blue will only appear in the blue channel and so forth. Whilst items perceived as blue will certainly appear brightest in the blue channel (i.e. they will contain more blue light than the other colours) they will also have milder components of red and green.

If we consider all the colours that can be represented within the RGB representation, then we appreciate that the RGB colour space is essentially a 3-D colour space (cube) with axes R, G and B (Figure 1.7). Each axis has the same range $0 \rightarrow 1$ (this is scaled to 0–255 for the common 1 byte per colour channel, 24-bit image representation). The colour black occupies the origin of the cube (position $(0, 0, 0)$), corresponding to the absence of all three colours; white occupies the opposite corner (position $(1, 1, 1)$), indicating the maximum amount of all three colours. All other colours in the spectrum lie within this cube.

The RGB colour space is based upon the portion of the electromagnetic spectrum visible to humans (i.e. the continuous range of wavelengths in the approximate range

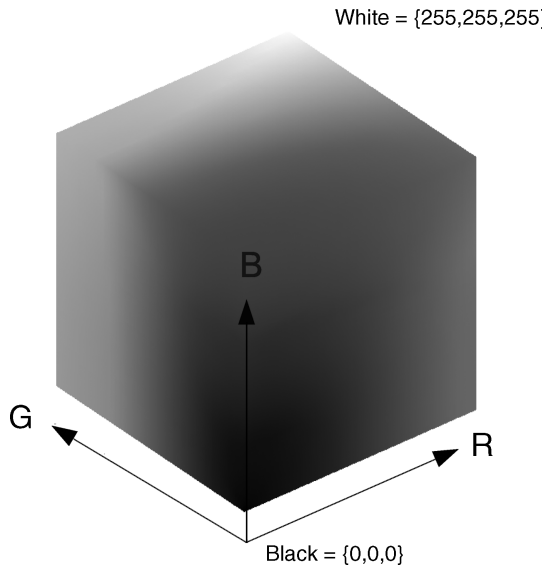

Figure 1.7 An illustration of RGB colour space as a 3-D cube *(See colour plate section for colour version)*

400–700 nm). The human eye has three different types of colour receptor over which it has limited (and nonuniform) absorbency for each of the red, green and blue wavelengths. This is why, as we will see later, the colour to grey-scale transform uses a nonlinear combination of the RGB channels.

In digital image processing we use a simplified RGB colour model (based on the CIE colour standard of 1931) that is optimized and standardized towards graphical displays. However, the primary problem with RGB is that it is perceptually nonlinear. By this we mean that moving in a given direction in the RGB colour cube (Figure 1.7) does not necessarily produce a colour that is perceptually consistent with the change in each of the channels. For example, starting at white and subtracting the blue component produces yellow; similarly, starting at red and adding the blue component produces pink. For this reason, RGB space is inherently difficult for humans to work with and reason about because it is not related to the natural way we perceive colours. As an alternative we may use perceptual colour representations such as HSV.

1.4.1.1 RGB to grey-scale image conversion

We can convert from an RGB colour space to a grey-scale image using a simple transform. Grey-scale conversion is the initial step in many image analysis algorithms, as it essentially simplifies (i.e. reduces) the amount of information in the image. Although a grey-scale image contains less information than a colour image, the majority of important, feature-related information is maintained, such as edges, regions, blobs, junctions and so on. Feature detection and processing algorithms then typically operate on the converted grey-scale version of the image. As we can see from Figure 1.8, it is still possible to distinguish between the red and green apples in grey-scale.

An RGB colour image, I_{colour}, is converted to grey scale, $I_{\text{grey-scale}}$, using the following transformation:

$$I_{\text{grey-scale}}(n,m) = \alpha I_{\text{colour}}(n,m,r) + \beta I_{\text{colour}}(n,m,g) + \gamma I_{\text{colour}}(n,m,b) \qquad (1.1)$$

Figure 1.8 An example of RGB colour image (left) to grey-scale image (right) conversion *(See colour plate section for colour version)*

where (n, m) indexes an individual pixel within the grey-scale image and (n, m, c) the individual channel at pixel location (n, m) in the colour image for channel c in the red r, blue b and green g image channels. As is apparent from Equation (1.1), the grey-scale image is essentially a weighted sum of the red, green and blue colour channels. The weighting coefficients (α, β and γ) are set in proportion to the perceptual response of the human eye to each of the red, green and blue colour channels and a standardized weighting ensures uniformity (NTSC television standard, $\alpha = 0.2989, \beta = 0.5870$ and $\gamma = 0.1140$). The human eye is naturally more sensitive to red and green light; hence, these colours are given higher weightings to ensure that the relative intensity balance in the resulting grey-scale image is similar to that of the RGB colour image. An example of performing a grey-scale conversion in Matlab is given in Example 1.6.

RGB to grey-scale conversion is a noninvertible image transform: the true colour information that is lost in the conversion cannot be readily recovered.

1.4.2 Perceptual colour space

Perceptual colour space is an alternative way of representing true colour images in a manner that is more natural to the human perception and understanding of colour than the RGB representation. Many alternative colour representations exist, but here we concentrate on the Hue, Saturation and Value (HSV) colour space popular in image analysis applications.

Changes within this colour space follow a perceptually acceptable colour gradient. From an image analysis perspective, it allows the separation of colour from lighting to a greater degree. An RGB image can be transformed into an HSV colour space representation as shown in Figure 1.9.

Each of these three parameters can be interpreted as follows:

- H (hue) is the dominant wavelength of the colour, e.g. red, blue, green
- S (saturation) is the 'purity' of colour (in the sense of the amount of white light mixed with it)
- V (value) is the brightness of the colour (also known as luminance).

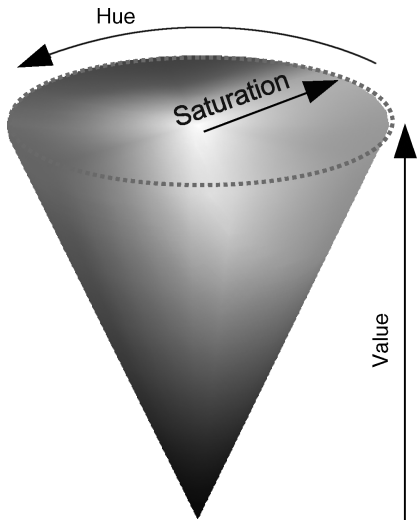

Figure 1.9 HSV colour space as a 3-D cone *(See colour plate section for colour version)*

The HSV representation of a 2-D image is also as a 3-D array comprising three channels (h, s, v) and each pixel location within the image, $I(n, m)$, contains an (h, s, v) triplet that can be transformed back into RGB for true-colour display. In the Matlab HSV implementation each of h, s and v are bounded within the range $0 \rightarrow 1$. For example, a blue hue (top of cone, Figure 1.9) may have a value of $h = 0.9$, a saturation of $s = 0.5$ and a value $v = 1$ making it a vibrant, bright sky-blue.

By examining the individual colour channels of images in the HSV space, we can see that image objects are more consistently contained in the resulting hue field than in the channels of the RGB representation, despite the presence of varying lighting conditions over the scene (Figure 1.10). As a result, HSV space is commonly used for colour-based

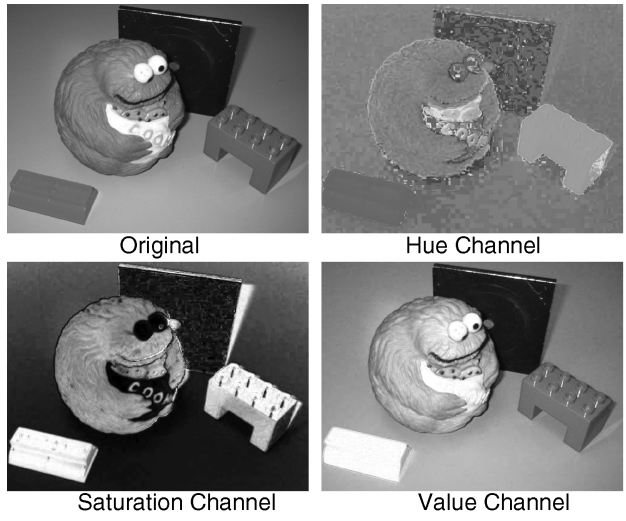

Figure 1.10 Image transformed and displayed in HSV colour space *(See colour plate section for colour version)*

image segmentation using a technique known as colour slicing. A portion of the hue colour wheel (a slice of the cone, Figure 1.9) is isolated as the colour range of interest, allowing objects within that colour range to be identified within the image. This ease of colour selection in HSV colour space also results in its widespread use as the preferred method of colour selection in computer graphical interfaces and as a method of adding false colour to images (Section 1.1.2).

Details of RGB to HSV image conversion in Matlab are given in Exercise 1.6.

1.5 Images in Matlab

Having introduced the basics of image representation, we now turn to the practical aspect of this book to investigate the initial stages of image manipulation using Matlab. These are presented as a number of worked examples and further exercises for the reader.

1.5.1 Reading, writing and querying images

Reading and writing images is accomplished very simply via the ***imread*** and ***imwrite*** functions. These functions support all of the most common image formats and create/export the appropriate 2-D/3-D image arrays within the Matlab environment. The function ***imfinfo*** can be used to query an image and establish all its important properties, including its type, format, size and bit depth.

Example 1.1

Matlab code	**What is happening?**
imfinfo('cameraman.tif')	%Query the cameraman image that %is available with Matlab %imfinfo provides information %ColorType is gray-scale, width is 256 . . . etc.
I1=imread('cameraman.tif');	%Read in the TIF format cameraman image
imwrite(I1,'cameraman.jpg','jpg');	%Write the resulting array I1 to %disk as a JPEG image
imfinfo('cameraman.jpg')	%Query the resulting disk image %Note changes in storage size, etc.

Comments

- Matlab functions: ***imread***, ***imwrite*** and ***iminfo***.
- Note the change in file size when the image is stored as a JPEG image. This is due to the (lossy) compression used by the JPEG image format.

1.5.2 Basic display of images

Matlab provides two basic functions for image display: ***imshow*** and ***imagesc***. Whilst ***imshow*** requires that the 2-D array specified for display conforms to an image data type (e.g. intensity/colour images with value range 0–1 or 0–255), ***imagesc*** accepts input arrays of any Matlab storage type (uint 8, uint 16 or double) and any numerical range. This latter function then scales the input range of the data and displays it using the current/default colour map. We can additionally control this display feature using the ***colormap*** function.

Example 1.2

Matlab code	**What is happening?**
A=imread('cameraman.tif');	%Read in intensity image
imshow(A);	%First display image using imshow
imagesc(A);	%Next display image using imagesc
axis image;	%Correct aspect ratio of displayed image
axis off;	%Turn off the axis labelling
colormap(gray);	%Display intensity image in grey scale

Comments

- Matlab functions: ***imshow***, ***imagesc*** and ***colormap***.
- Note additional steps required when using ***imagesc*** to display conventional images.

In order to show the difference between the two functions we now attempt the display of unconstrained image data.

Example 1.3

Matlab code	**What is happening?**
B=rand(256).*1000;	%Generate random image array in range 0–1000
imshow(B);	%Poor contrast results using imshow because data %exceeds expected range
imagesc(B); axis image; axis off; colormap(gray); colorbar;	%imagesc automatically scales colourmap to data %range
imshow(B,[0 1000]);	%But if we specify range of data explicitly then %imshow also displays correct image contrast

Comments

- Note the automatic display scaling of ***imagesc***.

If we wish to display multiple images together, this is best achieved by the ***subplot*** function. This function creates a mosaic of axes into which multiple images or plots can be displayed.

Example 1.4

Matlab code	What is happening?
B=imread('cell.tif');	%Read in 8-bit intensity image of cell
C=imread('spine.tif');	%Read in 8-bit intensity image of spine
D=imread('onion.png');	%Read in 8-bit colour image
subplot(3,1,1); imagesc(B); axis image;	%Creates a 3×1 mosaic of plots
axis off; colormap(gray);	%and display first image
subplot(3,1,2); imagesc(C); axis image; axis off; colormap(jet);	%Display second image %Set colourmap to jet (false colour)
subplot(3,1,3); imshow(D);	%Display third (colour) image

Comments

- Note the specification of different colour maps using ***imagesc*** and the combined display using both ***imagesc*** and ***imshow***.

1.5.3 Accessing pixel values

Matlab also contains a built-in interactive image viewer which can be launched using the ***imview*** function. Its purpose is slightly different from the other two: it is a graphical, image viewer which is intended primarily for the inspection of images and sub-regions within them.

Example 1.5

Matlab code	What is happening?
B=imread('cell.tif');	%Read in 8-bit intensity image of cell
imview(B);	%Examine grey-scale image in interactive viewer
D=imread('onion.png');	%Read in 8-bit colour image.
imview(B);	%Examine RGB image in interactive viewer
B(25,50) B(25,50)=255; imshow(B);	%Print pixel value at location (25,50) %Set pixel value at (25,50) to white %View resulting changes in image
D(25,50,:) D(25,50, 1)	%Print RGB pixel value at location (25,50) %Print only the red value at (25,50)

```
D(25,50,:)=(255, 255, 255);     %Set pixel value to RGB white
imshow(D);                      %View resulting changes in image
```

Comments

- Matlab functions: ***imview***.
- Note how we can access individual pixel values within the image and change their value.

1.5.4 Converting image types

Matlab also contains built in functions for converting different image types. Here, we examine conversion to grey scale and the display of individual RGB colour channels from an image.

Example 1.6

Matlab code	What is happening?
D=imread('onion.png');	%Read in 8-bit RGB colour image
Dgray=rgb2gray(D);	%Convert it to a grey-scale image
subplot(2,1,1); imshow(D); axis image; subplot(2,1,2); imshow(Dgray);	%Display both side by side

Comments

- Matlab functions: ***rgb2gray***.
- Note how the resulting grayscale image array is 2-D while the originating colour image array was 3-D.

Example 1.7

Matlab code	What is happening?
D=imread('onion.png');	%Read in 8-bit RGB colour image.
Dred=D(:,:,1); Dgreen=D(:,:,2); Dblue=D(:,:,3);	%Extract red channel (first channel) %Extract green channel (second channel) %Extract blue channel (third channel)
subplot(2,2,1); imshow(D); axis image;	%Display all in 2×2 plot

subplot(2,2,2); imshow(Dred); title('red');
subplot(2,2,3); imshow(Dgreen); title('green');
subplot(2,2,4); imshow(Dblue); title('blue');
	%Display and label

Comments

- Note how we can access individual channels of an RGB image and extract them as separate images in their own right.

Exercises

The following exercises are designed to reinforce and develop the concepts and Matlab examples introduced in this chapter

Matlab functions: ***imabsdiff***, ***rgb2hsv***.

Exercise 1.1 Using the examples presented for displaying an image in Matlab together with those for accessing pixel locations, investigate adding and subtracting a scalar value from an individual location, i.e. $I(i,j) = I(i,j) + 25$ or $I(i,j) = I(i,j) - 25$. Start by using the grey-scale 'cell.tif' example image and pixel location $(100, 20)$. What is the effect on the grey-scale colour of adding and subtracting?

Expand your technique to RGB colour images by adding and subtracting to all three of the colour channels in a suitable example image. Also try just adding to one of the individual colour channels whilst leaving the others unchanged. What is the effect on the pixel colour of each of these operations?

Exercise 1.2 Based on your answer to Exercise 1.1, use the ***for*** construct in Matlab (see *help for* at the Matlab command prompt) to loop over all the pixels in the image and brighten or darken the image.

You will need to ensure that your program does not try to create a pixel value that is larger or smaller than the pixel can hold. For instance, an 8-bit image can only hold the values 0–255 at each pixel location and similarly for each colour channel for a 24-bit RGB colour image.

Exercise 1.3 Using the grey-scale 'cell.tif' example image, investigate using different false colour maps to display the image. The Matlab function ***colormap*** can take a range of values to specify different false colour maps: enter *help graph3d* and look under the *Color maps* heading to get a full list. What different aspects and details of the image can be seen using these false colourings in place of the conventional grey-scale display?

False colour maps can also be specified numerically as parameters to the ***colormap*** command: enter *help colormap* for further details.

Exercise 1.4 Load an example image into Matlab and using the functions introduced in Example 1.1 save it once as a JPEG format file (e.g. sample.jpg) and once as a PNG format image (e.g. sample.png). Next, reload the images from both of these saved files as new images in Matlab, 'Ijpg' and 'Ipng'.

We may expect these two images to be exactly the same, as they started out as the same image and were just saved in different image file formats. If we compare them by subtracting one from the other and taking the absolute difference at each pixel location we can check whether this assumption is correct.

Use the ***imabsdiff*** Matlab command to create a difference image between 'Ijpg' and 'Ipng'. Display the resulting image using ***imagesc***.

The difference between these two images is not all zeros as one may expect, but a noise pattern related to the difference in the images introduced by saving in a lossy compression format (i.e. JPEG) and a lossless compression format (i.e. PNG). The difference we see is due

to the image information removed in the JPEG version of the file which is not apparent to us when we look at the image. Interestingly, if we view the difference image with ***imshow*** all we see is a black image because the differences are so small they have very low (i.e. dark) pixel values. The automatic scaling and false colour mapping of ***imagesc*** allows us to visualize these low pixel values.

Exercise 1.5 Implement a program to perform the bit-slicing technique described in Section 1.2.1 and extract/display the resulting plane images (Figure 1.3) as separate Matlab images.

You may wish to consider displaying a mosaic of several different bit-planes from an image using the ***subplot*** function.

Exercise 1.6 Using the Matlab ***rgb2hsv*** function, write a program to display the individual hue, saturation and value channels of a given RGB colour image. You may wish to refer to Example 1.6 on the display of individual red, green and blue channels.

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