

Chapter 1

Introduction

1.1 Images and pictures

As we mentioned in the preface, human beings are predominantly visual creatures: we rely heavily on our vision to make sense of the world around us. We not only look at things to identify and classify them, but we can scan for differences, and obtain an overall rough “feeling” for a scene with a quick glance.

Humans have evolved very precise visual skills: we can identify a face in an instant; we can differentiate colours; we can process a large amount of visual information very quickly.

However, the world is in constant motion: stare at something for long enough and it will change in some way. Even a large solid structure, like a building or a mountain, will change its appearance depending on the time of day (day or night); amount of sunlight (clear or cloudy), or various shadows falling upon it.

We are concerned with single images: snapshots, if you like, of a visual scene. Although image processing can deal with changing scenes, we shall not discuss it in any detail in this text.

For our purposes, an *image* is a single picture which represents something. It may be a picture of a person, of people or animals, or of an outdoor scene, or a microphotograph of an electronic component, or the result of medical imaging. Even if the picture is not immediately recognizable, it will not be just a random blur.

1.2 What is image processing?

Image processing involves changing the nature of an image in order to either

1. improve its pictorial information for human interpretation,
2. render it more suitable for autonomous machine perception.

We shall be concerned with *digital image processing*, which involves using a computer to change the nature of a *digital image* (see below). It is necessary to realize that these two aspects represent two separate but equally important aspects of image processing. A procedure which satisfies condition (1)—a procedure which makes an image “look better”—may be the very worst procedure for satisfying condition (2). Humans like their images to be sharp, clear and detailed; machines prefer their images to be simple and uncluttered.

Examples of (1) may include:

- Enhancing the edges of an image to make it appear sharper; an example is shown in figure 1.1. Note how the second image appears “cleaner”; it is a more pleasant image. Sharpening edges is a vital component of printing: in order for an image to appear “at its best” on the printed page; some sharpening is usually performed.



(a) The original image

(b) Result after “sharpening”

Figure 1.1: Image sharpening

- Removing “noise” from an image; noise being random errors in the image. An example is given in figure 1.2. Noise is a very common problem in data transmission: all sorts of electronic components may affect data passing through them, and the results may be undesirable. As we shall see in chapter 5 noise may take many different forms; each type of noise requiring a different method of removal.
- Removing motion blur from an image. An example is given in figure 1.3. Note that in the deblurred image (b) it is easier to read the numberplate, and to see the spikes on the fence behind the car, as well as other details not at all clear in the original image (a). Motion blur may occur when the shutter speed of the camera is too long for the speed of the object. In photographs of fast moving objects: athletes, vehicles for example, the problem of blur may be considerable.

Examples of (2) may include:

- Obtaining the edges of an image. This may be necessary for the measurement of objects in an image; an example is shown in figures 1.4. Once we have the edges we can measure their spread, and the area contained within them. We can also use edge detection algorithms as a first step in edge enhancement, as we saw above.



(a) The original image

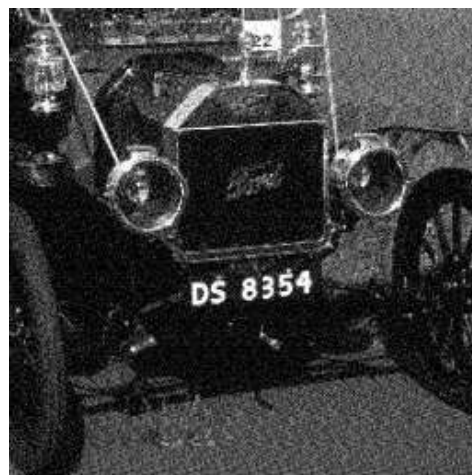


(b) After removing noise

Figure 1.2: Removing noise from an image



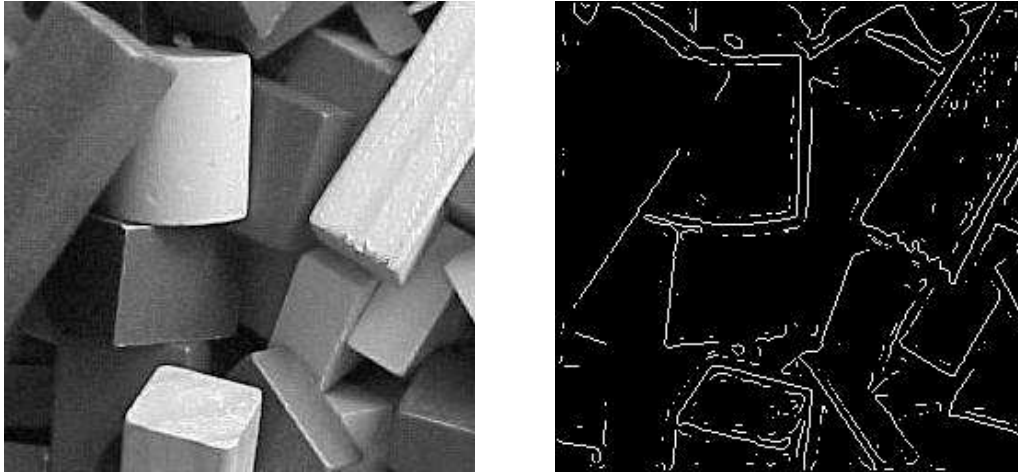
(a) The original image



(b) After removing the blur

Figure 1.3: Image deblurring

From the edge result, we see that it may be necessary to enhance the original image slightly, to make the edges clearer.



(a) The original image

(b) Its edge image

Figure 1.4: Finding edges in an image

- Removing detail from an image. For measurement or counting purposes, we may not be interested in all the detail in an image. For example, a machine inspected items on an assembly line, the only matters of interest may be shape, size or colour. For such cases, we might want to simplify the image. Figure 1.5 shows an example: in image (a) is a picture of an African buffalo, and image (b) shows a blurred version in which extraneous detail (like the logs of wood in the background) have been removed. Notice that in image (b) all the fine detail is gone; what remains is the coarse structure of the image. We could for example, measure the size and shape of the animal without being “distracted” by unnecessary detail.

1.3 Image Acquisition and sampling

Sampling refers to the process of digitizing a continuous function. For example, suppose we take the function

$$y = \sin(x) + \frac{1}{3} \sin(3x).$$

and sample it at ten evenly spaced values of x only. The resulting sample points are shown in figure 1.6. This shows an example of *undersampling*, where the number of points is not sufficient to reconstruct the function. Suppose we sample the function at 100 points, as shown in figure 1.7. We can clearly now reconstruct the function; all its properties can be determined from this sampling. In order to ensure that we have enough sample points, we require that the sampling period is not greater than one-half the finest detail in our function. This is known as the *Nyquist criterion*, and can be formulated more precisely in terms of “frequencies”, which are discussed in chapter 4. The Nyquist criterion can be stated as the *sampling theorem*, which says, in effect, that a continuous function can be reconstructed from its samples provided that the sampling frequency is at least twice the maximum frequency in the function. A formal account of this theorem is provided by Castleman [1].



(a) The original image

(b) Blurring to remove detail

Figure 1.5: Blurring an image

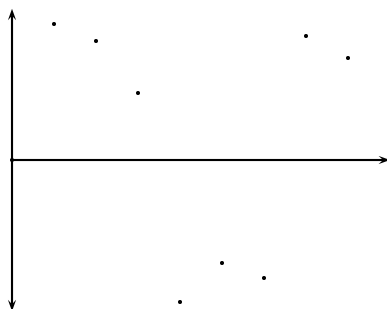


Figure 1.6: Sampling a function—undersampling

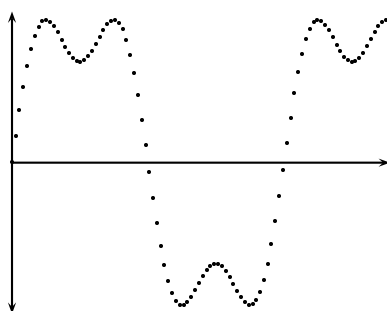
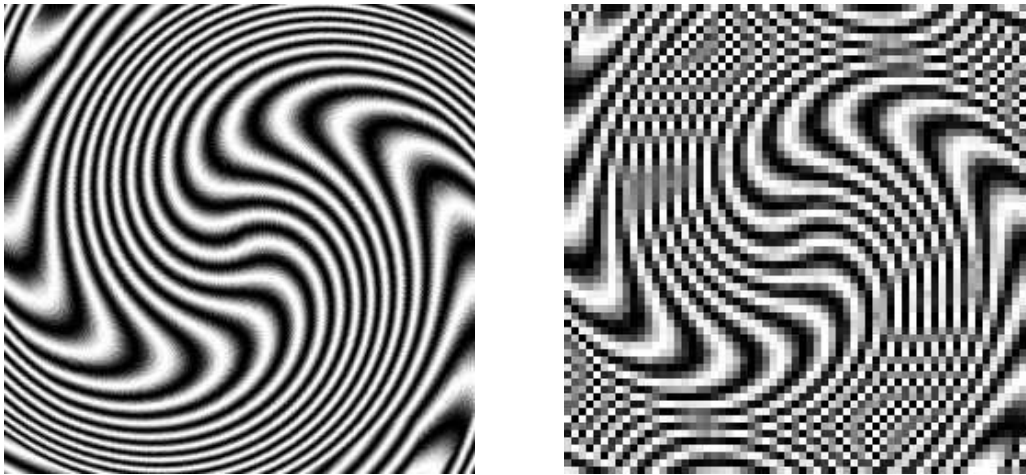


Figure 1.7: Sampling a function with more points

Sampling an image again requires that we consider the Nyquist criterion, when we consider an image as a continuous function of two variables, and we wish to sample it to produce a digital image.

An example is shown in figure 1.8 where an image is shown, and then with an undersampled version. The jagged edges in the undersampled image are examples of *aliasing*. The sampling rate



Correct sampling; no aliasing

An undersampled version with aliasing

Figure 1.8: Effects of sampling

will of course affect the final resolution of the image; we discuss this below. In order to obtain a sampled (digital) image, we may start with a continuous representation of a scene. To view the scene, we record the energy reflected from it; we may use visible light, or some other energy source.

Using light

Light is the predominant energy source for images; simply because it is the energy source which human beings can observe directly. We are all familiar with photographs, which are a pictorial record of a visual scene.

Many digital images are captured using visible light as the energy source; this has the advantage of being safe, cheap, easily detected and readily processed with suitable hardware. Two very popular methods of producing a digital image are with a digital camera or a flat-bed scanner.

CCD camera. Such a camera has, in place of the usual film, an array of *photosites*; these are silicon electronic devices whose voltage output is proportional to the intensity of light falling on them.

For a camera attached to a computer, information from the photosites is then output to a suitable storage medium. Generally this is done on hardware, as being much faster and more efficient than software, using a *frame-grabbing card*. This allows a large number of images to be captured in a very short time—in the order of one ten-thousandth of a second each. The images can then be copied onto a permanent storage device at some later time.

This is shown schematically in figure 1.9.

The output will be an array of values; each representing a sampled point from the original scene. The elements of this array are called *picture elements*, or more simply *pixels*.

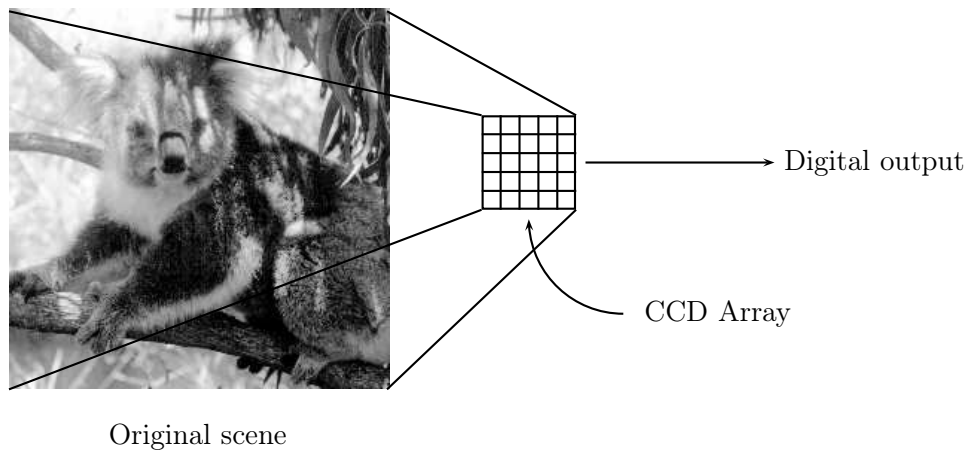


Figure 1.9: Capturing an image with a CCD array

Digital still cameras use a range of devices, from floppy discs and CD's, to various specialized cards and “memory sticks”. The information can then be downloaded from these devices to a computer hard disk.

Flat bed scanner. This works on a principle similar to the CCD camera. Instead of the entire image being captured at once on a large array, a single row of photosites is moved across the image, capturing it row-by-row as it moves. This is shown schematically in figure 1.10.

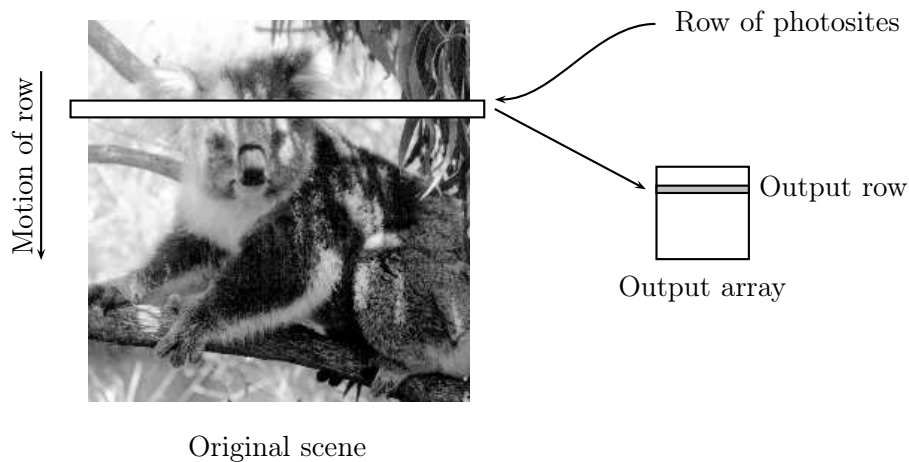


Figure 1.10: Capturing an image with a CCD scanner

Since this is a much slower process than taking a picture with a camera, it is quite reasonable to allow all capture and storage to be processed by suitable software.

Other energy sources

Although light is popular and easy to use, other energy sources may be used to create a digital image. Visible light is part of the *electromagnetic spectrum*: radiation in which the energy takes

the form of waves of varying wavelength. These range from cosmic rays of very short wavelength, to electric power, which has very long wavelength. Figure 1.11 illustrates this. For microscopy, we

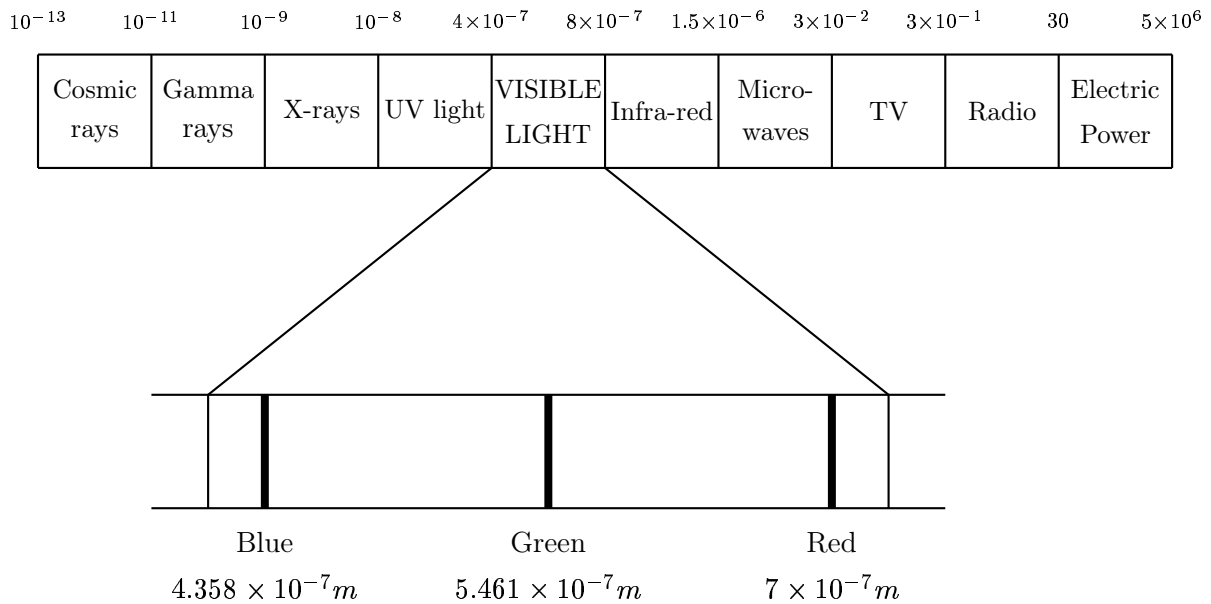


Figure 1.11: The electromagnetic spectrum

may use x-rays, or electron beams. As we can see from figure 1.11, x-rays have a shorter wavelength than visible light, and so can be used to resolve smaller objects than are possible with visible light. See Clark [2] for a good introduction to this. X-rays are of course also useful in determining the structure of objects usually hidden from view: such as bones.

A further method of obtaining images is by the use of *x-ray tomography*, where an object is encircled by an x-ray beam. As the beam is fired through the object, it is detected on the other side of the object, as shown in figure 1.12. As the beam moves around the object, an image of the object can be constructed; such an image is called a *tomogram*. In a CAT (Computed Axial Tomography) scan, the patient lies within a tube around which x-ray beams are fired. This enables a large number of tomographic “slices” to be formed, which can then be joined to produce a three-dimensional image. A good account of such systems (and others) is given by Siedband [13]

1.4 Images and digital images

Suppose we take an image, a photo, say. For the moment, let's make things easy and suppose the photo is monochromatic (that is, shades of grey only), so no colour. We may consider this image as being a two dimensional function, where the function values give the brightness of the image at any given point, as shown in figure 1.13. We may assume that in such an image brightness values can be any real numbers in the range 0.0 (black) to 1.0 (white). The ranges of x and y will clearly depend on the image, but they can take all real values between their minima and maxima.

Such a function can of course be plotted, as shown in figure 1.14. However, such a plot is of limited use to us in terms of image analysis. The concept of an image as a function, however, will

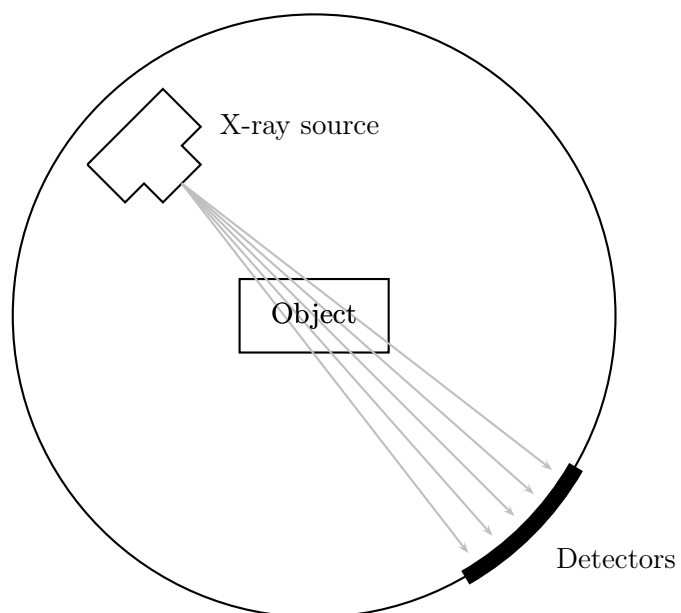


Figure 1.12: X-ray tomography

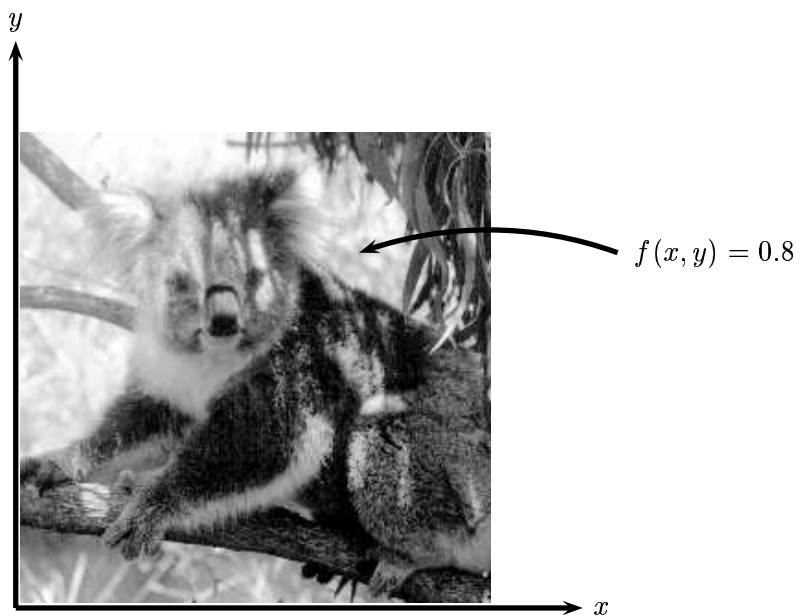


Figure 1.13: An image as a function

be vital for the development and implementation of image processing techniques.

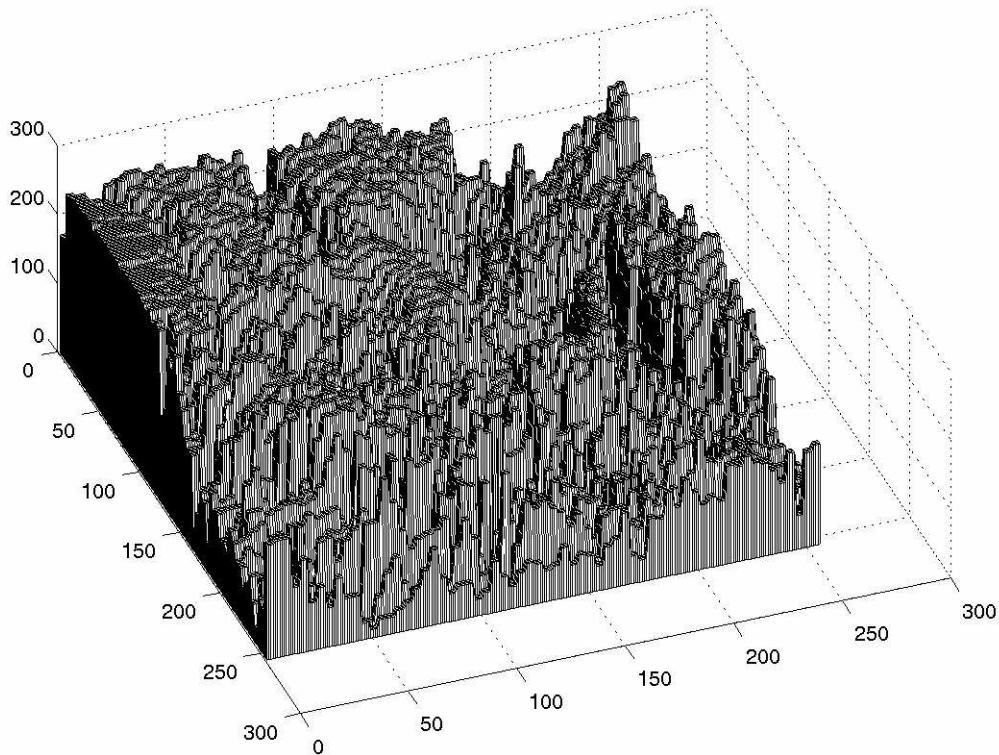


Figure 1.14: The image of figure 1.13 plotted as a function of two variables

A *digital image* differs from a photo in that the x , y , and $f(x, y)$ values are all *discrete*. Usually they take on only integer values, so the image shown in figure 1.13 will have x and y ranging from 1 to 256 each, and the brightness values also ranging from 0 (black) to 255 (white). A digital image, as we have seen above, can be considered as a large array of sampled points from the continuous image, each of which has a particular quantized brightness; these points are the *pixels* which constitute the digital image. The pixels surrounding a given pixel constitute its *neighbourhood*. A neighbourhood can be characterized by its shape in the same way as a matrix: we can speak, for example, of a 3×3 neighbourhood, or of a 5×7 neighbourhood. Except in very special circumstances, neighbourhoods have odd numbers of rows and columns; this ensures that the current pixel is in the centre of the neighbourhood. An example of a neighbourhood is given in figure 1.15. If a neighbourhood has an even number of rows or columns (or both), it may be necessary to specify which pixel in the neighbourhood is the “current pixel”.

1.5 Some applications

Image processing has an enormous range of applications; almost every area of science and technology can make use of image processing methods. Here is a short list just to give some indication of the range of image processing applications.

1. Medicine

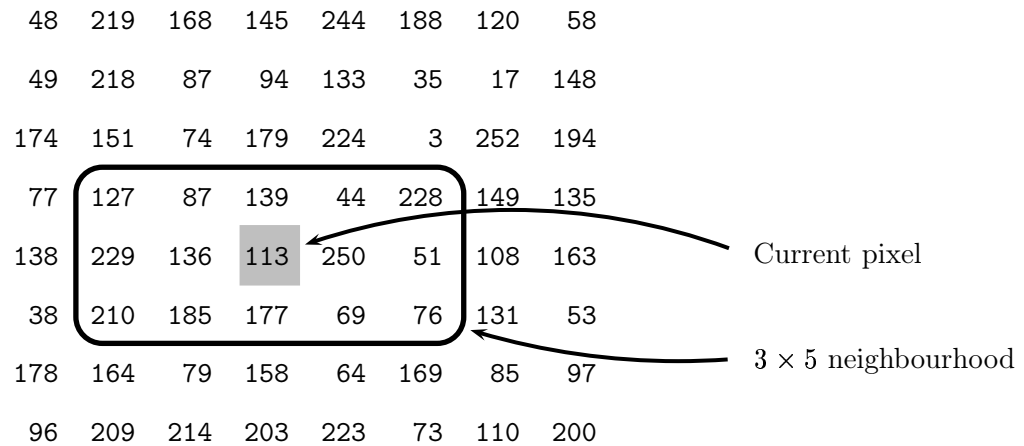


Figure 1.15: Pixels, with a neighbourhood

- Inspection and interpretation of images obtained from X-rays, MRI or CAT scans,
- analysis of cell images, of chromosome karyotypes.

2. Agriculture

- Satellite/aerial views of land, for example to determine how much land is being used for different purposes, or to investigate the suitability of different regions for different crops,
- inspection of fruit and vegetables—distinguishing good and fresh produce from old.

3. Industry

- Automatic inspection of items on a production line,
- inspection of paper samples.

4. Law enforcement

- Fingerprint analysis,
- sharpening or de-blurring of speed-camera images.

1.6 Aspects of image processing

It is convenient to subdivide different image processing algorithms into broad subclasses. There are different algorithms for different tasks and problems, and often we would like to distinguish the nature of the task at hand.

Image enhancement. This refers to processing an image so that the result is more suitable for a particular application. Example include:

- sharpening or de-blurring an out of focus image,
- highlighting edges,
- improving image contrast, or brightening an image,

- removing noise.

Image restoration. This may be considered as reversing the damage done to an image by a known cause, for example:

- removing of blur caused by linear motion,
- removal of optical distortions,
- removing periodic interference.

Image segmentation. This involves subdividing an image into constituent parts, or isolating certain aspects of an image:

- finding lines, circles, or particular shapes in an image,
- in an aerial photograph, identifying cars, trees, buildings, or roads.

These classes are not disjoint; a given algorithm may be used for both image enhancement or for image restoration. However, we should be able to decide what it is that we are trying to do with our image: simply make it look better (enhancement), or removing damage (restoration).

1.7 An image processing task

We will look in some detail at a particular real-world task, and see how the above classes may be used to describe the various stages in performing this task. The job is to obtain, by an automatic process, the postcodes from envelopes. Here is how this may be accomplished:

Acquiring the image. First we need to produce a digital image from a paper envelope. This can be done using either a CCD camera, or a scanner.

Preprocessing. This is the step taken before the “major” image processing task. The problem here is to perform some basic tasks in order to render the resulting image more suitable for the job to follow. In this case it may involve enhancing the contrast, removing noise, or identifying regions likely to contain the postcode.

Segmentation. Here is where we actually “get” the postcode; in other words we extract from the image that part of it which contains just the postcode.

Representation and description. These terms refer to extracting the particular features which allow us to differentiate between objects. Here we will be looking for curves, holes and corners which allow us to distinguish the different digits which constitute a postcode.

Recognition and interpretation. This means assigning labels to objects based on their descriptors (from the previous step), and assigning meanings to those labels. So we identify particular digits, and we interpret a string of four digits at the end of the address as the postcode.

1.8 Types of digital images

We shall consider four basic types of images:

Binary. Each pixel is just black or white. Since there are only two possible values for each pixel, we only need one bit per pixel. Such images can therefore be very efficient in terms of storage. Images for which a binary representation may be suitable include text (printed or handwriting), fingerprints, or architectural plans.

An example was the image shown in figure 1.4(b) above. In this image, we have only the two colours: white for the edges, and black for the background. See figure 1.16 below.

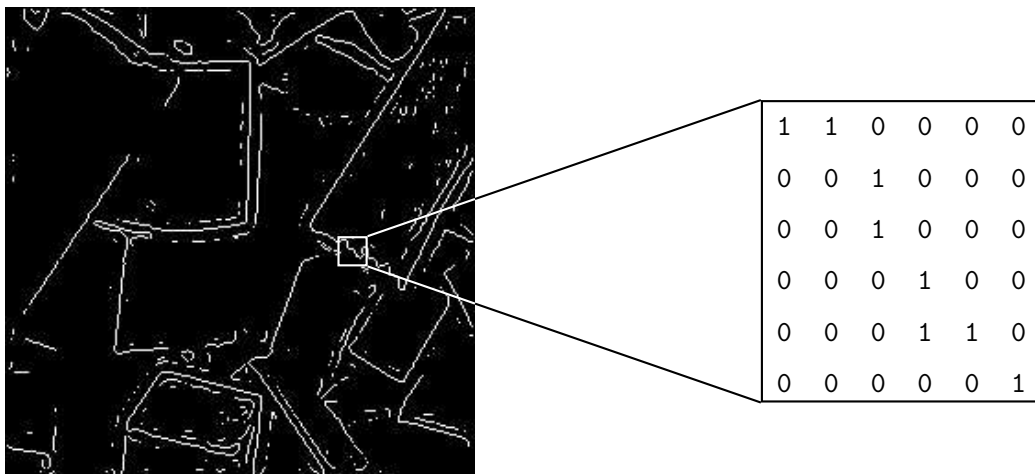


Figure 1.16: A binary image

Greyscale. Each pixel is a shade of grey, normally from 0 (black) to 255 (white). This range means that each pixel can be represented by eight bits, or exactly one byte. This is a very natural range for image file handling. Other greyscale ranges are used, but generally they are a power of 2. Such images arise in medicine (X-rays), images of printed works, and indeed 256 different grey levels is sufficient for the recognition of most natural objects.

An example is the street scene shown in figure 1.1 above, and in figure 1.17 below.

True colour, or RGB. Here each pixel has a particular colour; that colour being described by the amount of red, green and blue in it. If each of these components has a range 0–255, this gives a total of $255^3 = 16,777,216$ different possible colours in the image. This is enough colours for any image. Since the total number of bits required for each pixel is 24, such images are also called *24-bit colour images*.

Such an image may be considered as consisting of a “stack” of three matrices; representing the red, green and blue values for each pixel. This means that for every pixel there correspond three values.

An example is shown in figure 1.18.

Indexed. Most colour images only have a small subset of the more than sixteen million possible colours. For convenience of storage and file handling, the image has an associated *colour map*,



Figure 1.17: A greyscale image

or *colour palette*, which is simply a list of all the colours used in that image. Each pixel has a value which does not give its colour (as for an RGB image), but an *index* to the colour in the map.

It is convenient if an image has 256 colours or less, for then the index values will only require one byte each to store. Some image file formats (for example, CompuServe GIF), allow only 256 colours or fewer in each image, for precisely this reason.

Figure 1.19 shows an example. In this image the indices, rather than being the grey values of the pixels, are simply indices into the colour map. Without the colour map, the image would be very dark and colourless. In the figure, for example, pixels labelled 5 correspond to 0.2627 0.2588 0.2549, which is a dark greyish colour.

1.9 Image File Sizes

Image files tend to be large. We shall investigate the amount of information used in different image type of varying sizes. For example, suppose we consider a 512×512 binary image. The number of bits used in this image (assuming no compression, and neglecting, for the sake of discussion, any header information) is

$$\begin{aligned}
 512 \times 512 \times 1 &= 262,144 \\
 &= 32768 \text{ bytes}
 \end{aligned}$$

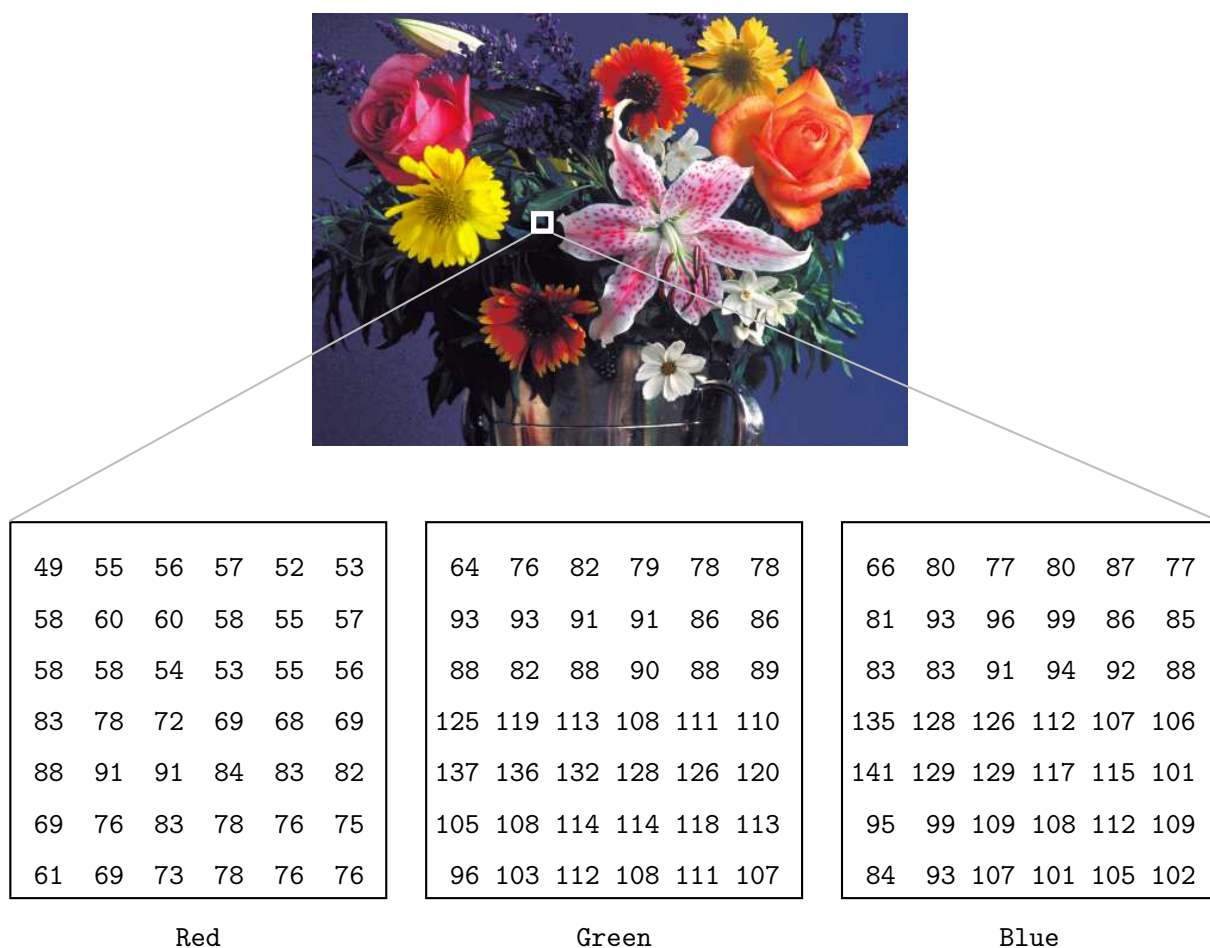


Figure 1.18: A true colour image



Figure 1.19: An indexed colour image

$$= 32.768 \text{ Kb}$$

$$\approx 0.033 \text{ Mb.}$$

(Here we use the convention that a kilobyte is one thousand bytes, and a megabyte is one million bytes.)

A greyscale image of the same size requires:

$$512 \times 512 \times 1 = 262,144 \text{ bytes}$$

$$= 262.14 \text{ Kb}$$

$$\approx 0.262 \text{ Mb.}$$

If we now turn our attention to colour images, each pixel is associated with 3 bytes of colour information. A 512×512 image thus requires

$$512 \times 512 \times 3 = 786,432 \text{ bytes}$$

$$= 786.43 \text{ Kb}$$

$$\approx 0.786 \text{ Mb.}$$

Many images are of course such larger than this; satellite images may be of the order of several thousand pixels in each direction.

1.10 Image perception

Much of image processing is concerned with making an image appear “better” to human beings. We should therefore be aware of the limitations of the the human visual system. Image perception consists of two basic steps:

1. capturing the image with the eye,

2. recognising and interpreting the image with the *visual cortex* in the brain.

The combination and immense variability of these steps influences the ways in we perceive the world around us.

There are a number of things to bear in mind:

1. *Observed intensities* vary as to the background. A single block of grey will appear darker if placed on a white background than if it were placed on a black background. That is, we don't perceive grey scales "as they are", but rather as they differ from their surroundings. In figure 1.20 a grey square is shown on two different backgrounds. Notice how much darker the square appears when it is surrounded by a light grey. However, the two central squares have exactly the same intensity.



Figure 1.20: A grey square on different backgrounds

2. We may observe non-existent intensities as bars in continuously varying grey levels. See for example figure 1.21. This image varies continuously from light to dark as we travel from left to right. However, it is impossible for our eyes not to see a few horizontal edges in this image.
3. Our visual system tends to undershoot or overshoot around the boundary of regions of different intensities. For example, suppose we had a light grey blob on a dark grey background. As our eye travels from the dark background to the light region, the boundary of the region appears lighter than the rest of it. Conversely, going in the other direction, the boundary of the background appears *darker* than the rest of it.

We have seen in the previous chapter that matrices can be handled very efficiently in MATLAB. Images may be considered as matrices whose elements are the pixel values of the image. In this chapter we shall investigate how the matrix capabilities of MATLAB allow us to investigate images and their properties.

1.11 Greyscale images

Suppose you are sitting at your computer and have started MATLAB. You will have a MATLAB *command window* open, and in it the MATLAB prompt



Figure 1.21: Continuously varying intensities

```
>>
```

ready to receive commands. Type in the command

```
>> w=imread('wombats.tif');
```

This takes the grey values of all the pixels in the greyscale image `wombats.tif` and puts them all into a matrix `w`. This matrix `w` is now a MATLAB *variable*, and so we can perform various matrix operations on it. In general the `imread` function reads the pixel values from an image file, and returns a matrix of all the pixel values.

Two things to note about this command:

1. It ends in a *semicolon*; this has the effect of not displaying the results of the command to the screen. As the result of this particular command is a matrix of size 256×256 , or with 65536 elements, we don't really want all its values displayed.
2. The name `wombats.tif` is given in single quote marks. Without them, MATLAB would assume that `wombats.tif` was the name of a variable, rather than the name of a file.

Now we can display this matrix as a greyscale image:

```
>> figure,imshow(w),pixval on
```

This is really three commands on the one line. MATLAB allows many commands to be entered on the same line; using commas to separate the different commands. The three commands we are using here are:

`figure`, which creates a *figure* on the screen. A figure is a window in which a graphics object can be placed. Objects may include images, or various types of graphs.

`imshow(g)`, which displays the matrix `g` as an image.

`pixval on`, which turns on the pixel values in our figure. This is a display of the grey values of the pixels in the image. They appear at the bottom of the figure in the form

$$c \times r = p$$

where c is the column value of the given pixel; r its row value, and p its grey value. Since `wombats.tif` is an 8-bit greyscale image, the pixel values appear as integers in the range 0–255.

This is shown in figure 1.22.

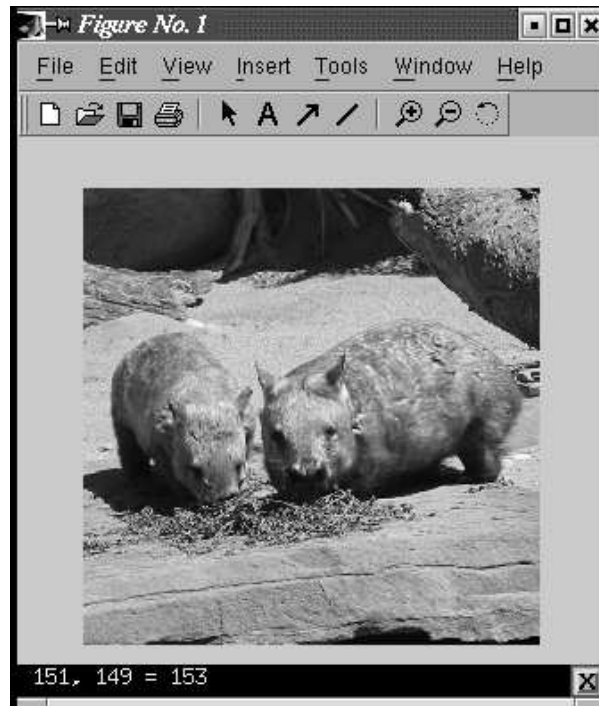


Figure 1.22: The wombats image with `pixval on`

If there are no figures open, then an `imshow` command, or any other command which generates a graphics object, will open a new figure for displaying the object. However, it is good practice to use the `figure` command whenever you wish to create a new figure.

We could display this image directly, without saving its grey values to a matrix, with the command

```
imshow('wombats.tif')
```

However, it is better to use a matrix, seeing as these are handled very efficiently in MATLAB.

1.12 RGB Images

As we shall discuss in chapter 11; we need to define colours in some standard way, usually as a subset of a three dimensional coordinate system; such a subset is called a *colour model*. There are

in fact a number of different methods for describing colour, but for image display and storage a standard model is RGB, for which we may imagine all the colours sitting inside a “colour cube” of side 1 as shown in figure 1.23. The colours along the black-white diagonal, shown in the diagram

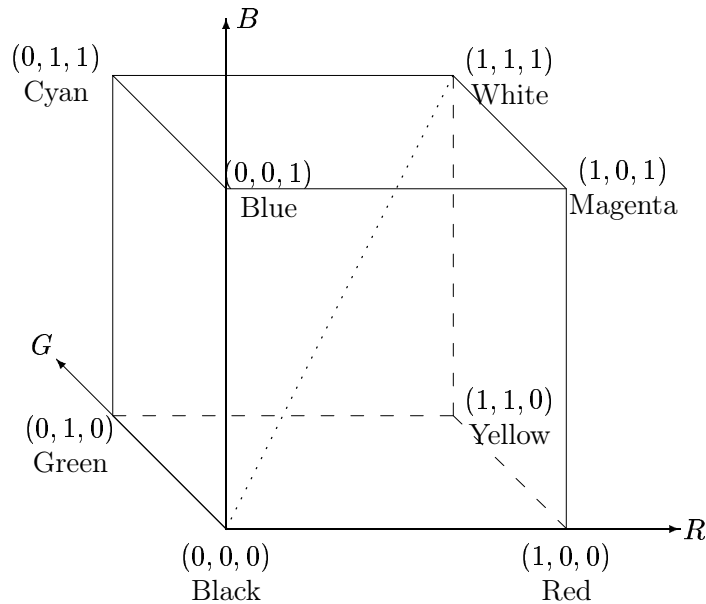


Figure 1.23: The colour cube for the RGB colour model

as a dotted line, are the points of the space where all the R , G , B values are equal. They are the different intensities of grey. We may also think of the axes of the colour cube as being discretized to integers in the range 0–255.

RGB is the standard for the *display* of colours: on computer monitors; on TV sets. But it is not a very good way of *describing* colours. How, for example, would you define light brown using RGB? As we shall see also in chapter 11, there are some colours which are not realizable with the RGB model; in that they would require negative values of one or two of the RGB components. MATLAB handles 24-bit RGB images in much the same way as greyscale. We can save the colour values to a matrix and view the result:

```
>> a=imread('autumn.tif');
>> figure,imshow(a),pival on
```

Note now that the pixel values now consist of a list of three values, giving the red, green and blue components of the colour of the given pixel.

An important difference between this type of image and a greyscale image can be seen by the command

```
>> size(a)
```

which returns *three* values: the number of rows, columns, and “pages” of \mathbf{a} , which is a three-dimensional matrix, also called a *multidimensional array*. MATLAB can handle arrays of any dimension, and \mathbf{a} is an example. We can think of \mathbf{a} as being a stack of three matrices, each of the same size.

To obtain any of the RGB values at a given location, we use similar indexing methods to above. For example

```
>> a(100,200,2)
```

returns the second colour value (green) at the pixel in row 100 and column 200. If we want all the colour values at that point, we can use

```
>> a(100,200,1:3)
```

However, MATLAB allows a convenient shortcut for listing all values along a particular dimension; just using a colon on its own:

```
>> a(100,200,:)
```

A useful function for obtaining RGB values is `impixel`; the command

```
>> impixel(a,200,100)
```

returns the red, green, and blue values of the pixel at column 200, row 100. Notice that the order of indexing is the same as that which is provided by the `pixval` command. This is opposite to the row, column order for matrix indexing. This command also applies to greyscale images:

```
>> impixel(g,100,200)
```

will return three values, but since `g` is a single two-dimensional matrix, all three values will be the same.

1.13 Indexed colour images

The command

```
>> figure,imshow('emu.tif'),pixval on
```

produces a nice colour image of an emu. However, the pixel values, rather than being three integers as they were for the RGB image above, are three fractions between 0 and 1. What is going on here?

If we try saving to a matrix first and then displaying the result:

```
>> em=imread('emu.tif');
>> figure,imshow(em),pixval on
```

we obtain a dark, barely distinguishable image, with single integer grey values, indicating that `em` is being interpreted as a single greyscale image.

In fact the image `emu.tif` is an example of an *indexed image*, consisting of two matrices: a *colour map*, and an *index* to the colour map. Assigning the image to a single matrix picks up only the index; we need to obtain the colour map as well:

```
>> [em,emap]=imread('emu.tif');
>> figure,imshow(em,emap),pixval on
```

MATLAB stores the RGB values of an indexed image as values of type `double`, with values between 0 and 1.

Information about your image

A great deal of information can be obtained with the `imfinfo` function. For example, suppose we take our indexed image `emu.tif` from above.

```
>> imfinfo('emu.tif')

ans =

        Filename: 'emu.tif'
      FileModDate: '26-Nov-2002 14:23:01'
        FileSize: 119804
         Format: 'tif'
  FormatVersion: []
         Width: 331
         Height: 384
        BitDepth: 8
      ColorType: 'indexed'
FormatSignature: [73 73 42 0]
      ByteOrder: 'little-endian'
NewSubfileType: 0
  BitsPerSample: 8
    Compression: 'PackBits'
PhotometricInterpretation: 'RGB Palette'
    StripOffsets: [16x1 double]
SamplesPerPixel: 1
    RowsPerStrip: 24
StripByteCounts: [16x1 double]
    XResolution: 72
    YResolution: 72
ResolutionUnit: 'Inch'
      Colormap: [256x3 double]
PlanarConfiguration: 'Chunky'
      TileWidth: []
      TileLength: []
      TileOffsets: []
      TileByteCounts: []
      Orientation: 1
      FillOrder: 1
GrayResponseUnit: 0.0100
MaxSampleValue: 255
MinSampleValue: 0
      Thresholding: 1
```

Much of this information is not useful to us; but we can see the size of the image in pixels, the size of the file (in bytes), the number of bits per pixel (this is given by `BitDepth`), and the colour type (in this case “indexed”).

For comparison, let's look at the output of a true colour file (showing only the first few lines of the output):

```
>> imfinfo('flowers.tif')

ans =

        Filename: [1x57 char]
    FileModDate: '26-Oct-1996 02:11:09'
        FileSize: 543962
         Format: 'tif'
FormatVersion: []
         Width: 500
         Height: 362
        BitDepth: 24
        ColorType: 'truecolor'
```

Now we shall test this function on a binary image:

```
>> imfinfo('text.tif')

ans =

        Filename: [1x54 char]
    FileModDate: '26-Oct-1996 02:12:23'
        FileSize: 3474
         Format: 'tif'
FormatVersion: []
         Width: 256
         Height: 256
        BitDepth: 1
        ColorType: 'grayscale'
```

What is going on here? We have a binary image, and yet the colour type is given as “grayscale”. The fact is that MATLAB does not distinguish between greyscale and binary images: a binary image is just a special case of a greyscale image which has only two intensities. However, we can see that `text.tif` is a binary image since the number of bits per pixel is only one.

1.14 Data types and conversions

Elements in MATLAB matrices may have a number of different numeric data types; the most common are listed in table 1.1. There are others, but those listed will be sufficient for all our work with images. These data types are also functions, we can convert from one type to another. For example:

```
>> a=23;
>> b=uint8(a);
>> b

b =
```

Data type	Description	Range
<code>int8</code>	8-bit integer	-128 — 127
<code>uint8</code>	8-bit unsigned integer	0 — 255
<code>int16</code>	16-bit integer	-32768 — 32767
<code>uint16</code>	16-bit unsigned integer	0 — 65535
<code>double</code>	Double precision real number	Machine specific

Table 1.1: Data types in MATLAB

```

23

>> whos a b
Name      Size      Bytes  Class

a         1x1         8  double array
b         1x1         1  uint8 array

```

Even though the variables `a` and `b` have the same numeric value, they are of different data types. An important consideration (of which we shall more) is that arithmetic operations are not permitted with the data types `int8`, `int16`, `uint8` and `uint16`.

A greyscale image may consist of pixels whose values are of data type `uint8`. These images are thus reasonably efficient in terms of storage space, since each pixel requires only one byte. However, arithmetic operations are not permitted on this data type; a `uint8` image must be converted to `double` before any arithmetic is attempted.

We can convert images from one image type to another. Table 1.2 lists all of MATLAB's functions for converting between different image types. Note that the `gray2rgb` function, does not create a

Function	Use	Format
<code>ind2gray</code>	Indexed to Greyscale	<code>y=ind2gray(x,map);</code>
<code>gray2ind</code>	Greyscale to indexed	<code>[y,map]=gray2ind(x);</code>
<code>rgb2gray</code>	RGB to greyscale	<code>y=rgb2gray(x);</code>
<code>gray2rgb</code>	Greyscale to RGB	<code>y=gray2rgb(x);</code>
<code>rgb2ind</code>	RGB to indexed	<code>[y,map]=rgb2ind;</code>
<code>ind2rgb</code>	Indexed to RGB	<code>y=ind2rgb(x,map);</code>

Table 1.2: Converting images in MATLAB

colour image, but an image all of whose pixel colours were the same as before. This is done by simply replicating the grey values of each pixel: greys in an RGB image are obtained by equality of the red, green and blue values.

1.15 Basics of image display

An image may be represented as a matrix of the grey values of its pixels. The problem here is to display that matrix on the computer screen. There are many factors which will effect the display;

they include:

1. ambient lighting,
2. the monitor type and settings,
3. the graphics card,
4. monitor resolution.

The same image may appear very different when viewed on a dull CRT monitor or on a bright LCD monitor. The resolution can also affect the display of an image; a higher resolution may result in the image taking up less physical area on the screen, but this may be counteracted by a loss in the colour depth: the monitor may be only to display 24-bit colour at low resolutions. If the monitor is bathed in bright light (sunlight, for example), the display of the image may be compromised. Furthermore, the individual's own visual system will affect the appearance of an image: the same image, viewed by two people, may appear to have different characteristics to each person. For our purpose, we shall assume that the computer set up is as optimal as is possible, and the monitor is able to accurately reproduce the necessary grey values or colours in any image.

A very basic MATLAB function for image display is `image`. This function simply displays a matrix as an image. However, it may not give necessarily very good results. For example:

```
>> c=imread('cameraman.tif');  
>> image(c)
```

will certainly display the cameraman, but possibly in an odd mixture of colours, and with some stretching. The strange colours come from the fact that the `image` command uses the current colour map to assign colours to the matrix elements. The default colour map is called `jet`, and consists of 64 very bright colours, which is inappropriate for the display of a greyscale image.

To display the image properly, we need to add several extra commands to the `image` line.

1. `trueimage`, which displays one matrix element (in this case an image pixel) for each screen pixel. More formally, we may use `trueimage([256 256])` where the vector components give the number of screen pixels vertically and horizontally to use in the display. If the vector is not specified, it defaults to the image size.
2. `axis off` which turns off the axis labelling,
3. `colormap(gray(247))`, which adjusts the image colour map to use shades of grey only. We can find the number of grey levels used by the cameraman image with

```
>> size(unique(c))  
  
ans =  
  
247    1
```

Since the cameraman image thus uses 247 different grey levels, we only need that number of greys in the colour map.

Thus a complete command for viewing this image will be

```
>> image(c),truesize,axis off, colormap(gray(247))
```

We may to adjust the colour map to use less or more colours; however this can have a dramatic effect on the result. The command

```
>> image(c),truesize,axis off, colormap(gray(512))
```

will produce a dark image. This happens because only the first 247 elements of the colour map will be used by the image for display, and these will all be in the first half of the colour map; thus all dark greys. On the other hand,

```
>> image(c),truesize,axis off, colormap(gray(128))
```

will produce a very light image, because any pixel with grey level higher than 128 will simply pick that highest grey value (which is white) from the colour map.

The `image` command works well for an indexed colour image, as long as we remember to use `imread` to pick up the colour map as well:

```
>> [x,map]=imread('cat.tif');  
>> image(x),truesize,axis off,colormap(map)
```

For true colour images, the image data will be read (by `imread`) as a three dimensional array. In such a case, `image` will ignore the current colour map, and assign colours to the display based on the values in the array. So

```
>> t=imread('twins.tif');  
>> image(t),truesize,axis off
```

will produce the correct twins image.

In general the `image` function can be used to display any image or matrix. However, there is a command which is more convenient, and does most of the work of colour mapping for us; we discuss this in the next section.

1.16 The `imshow` function

Greyscale images

We have seen that if `x` is a matrix of type `uint8`, then the command

```
imshow(x)
```

will display `x` as an image. This is reasonable, since the data type `uint8` restricts values to be integers between 0 and 255. However, not all image matrices come so nicely bundled up into this data type, and lots of MATLAB image processing commands produces output matrices which are of type `double`. We have two choices with a matrix of this type:

1. convert to type `uint8` and then display,
2. display the matrix directly.

The second option is possible because `imshow` will display a matrix of type `double` as a greyscale image as long as the matrix elements are between 0 and 1. Suppose we take an image and convert it to type `double`:

```
>> c=imread('caribou.tif');
>> cd=double(c);
>> imshow(c),figure,imshow(cd)
```

The results are shown in figure 1.24.



(a) The original image

(b) After conversion to type `double`

Figure 1.24: An attempt at data type conversion

However, as you can see, figure 1.24(b) doesn't look much like the original picture at all! This is because for a matrix of type `double`, the `imshow` function expects the values to be between 0 and 1, where 0 is displayed as black, and 1 is displayed as white. A value v with $0 < v < 1$ is displayed as grey scale $[255v]$. Conversely, values greater than 1 will be displayed as 1 (white) and values less than 0 will be displayed as zero (black). In the caribou image, every pixel has value greater than or equal to 1 (in fact the minimum value is 21), so that every pixel will be displayed as white. To display the matrix `cd`, we need to scale it to the range 0—1. This is easily done simply by dividing all values by 255:

```
>> imshow(cd/255)
```

and the result will be the caribou image as shown in figure 1.24(a).

We can vary the display by changing the scaling of the matrix. Results of the commands:

```
>> imshow(cd/512)
>> imshow(cd/128)
```

are shown in figures 1.25.

Dividing by 512 darkens the image, as all matrix values are now between 0 and 0.5, so that the brightest pixel in the image is a mid-grey. Dividing by 128 means that the range is 0—2, and all pixels in the range 1—2 will be displayed as white. Thus the image has an over-exposed, washed-out appearance.

(a) The matrix `cd` divided by 512(b) The matrix `cd` divided by 128

Figure 1.25: Scaling by dividing an image matrix by a scalar

The display of the result of a command whose output is a matrix of type `double` can be greatly affected by a judicious choice of a scaling factor.

We can convert the original image to `double` more properly using the function `im2double`. This applies correct scaling so that the output values are between 0 and 1. So the commands

```
>> cd=im2double(c);
>> imshow(cd)
```

will produce a correct image. It is important to make the distinction between the two functions `double` and `im2double`: `double` changes the data type but does not change the numeric values; `im2double` changes both the numeric data type and the values. The exception of course is if the original image is of type `double`, in which case `im2double` does nothing. Although the command `double` is not of much use for direct image display, it can be very useful for image arithmetic. We have seen examples of this above with scaling.

Corresponding to the functions `double` and `im2double` are the functions `uint8` and `im2uint8`. If we take our image `cd` of type `double`, properly scaled so that all elements are between 0 and 1, we can convert it back to an image of type `uint8` in two ways:

```
>> c2=uint8(255*cd);
>> c3=im2uint8(cd);
```

Use of `im2uint8` is to be preferred; it takes other data types as input, and always returns a correct result.

Binary images

Recall that a binary image will have only two values: 0 and 1. MATLAB does not have a `binary` data type as such, but it does have a `logical` flag, where `uint8` values as 0 and 1 can be interpreted as logical data. The logical flag will be set by the use of relational operations such as `==`, `<` or `>` or any other operations which provide a yes/no answer. For example, suppose we take the caribou matrix and create a new matrix with

```
>> c1=c>120;
```

(we will see more of this type of operation in chapter 2.) If we now check all of our variables with `whos`, the output will include the line:

```
c1      256x256      65536  uint8 array (logical)
```

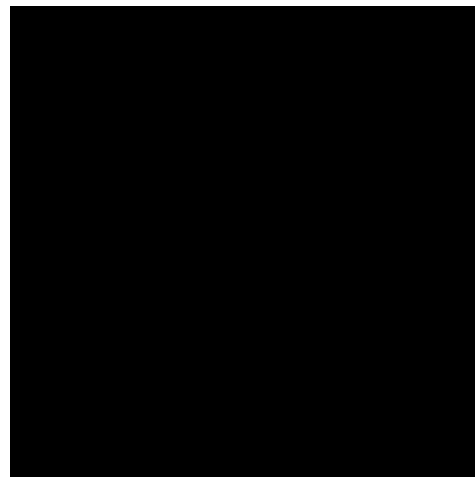
This means that the command

```
>> imshow(c1)
```

will display the matrix as a binary image; the result is shown in figure 1.26.



(a) The caribou image turned binary



(b) After conversion to type `uint8`

Figure 1.26: Making the image binary

Suppose we remove the logical flag from `c1`; this can be done by a simple command:

```
>> c1 = +c1;
```

Now the output of `whos` will include the line:

```
c1      256x256      65536  uint8 array
```

If we now try to display this matrix with `imshow`, we obtain the result shown in figure 1.26(b). A very disappointing image! But this is to be expected; in a matrix of type `uint8`, white is 255, 0 is black, and 1 is a very dark grey—indistinguishable from black.

To get back to a viewable image, we can either turn the logical flag back on, and view the result:

```
>> imshow(logical(c1))
```

or simply convert to type `double`:

```
>> imshow(double(c1))
```

Both these commands will produce the image seen in figure 1.26.

1.17 Bit planes

Greyscale images can be transformed into a sequence of binary images by breaking them up into their *bit-planes*. If we consider the grey value of each pixel of an 8-bit image as an 8-bit binary word, then the 0th bit plane consists of the last bit of each grey value. Since this bit has the least effect in terms of the magnitude of the value, it is called the *least significant bit*, and the plane consisting of those bits the *least significant bit plane*. Similarly the 7th bit plane consists of the first bit in each value. This bit has the greatest effect in terms of the magnitude of the value, so it is called the *most significant bit*, and the plane consisting of those bits the *most significant bit plane*.

If we take a greyscale image, we start by making it a matrix of type `double`; this means we can perform arithmetic on the values.

```
>> c=imread('cameraman.tif');
>> cd=double(c);
```

We now isolate the bit planes by simply dividing the matrix `cd` by successive powers of 2, throwing away the remainder, and seeing if the final bit is 0 or 1. We can do this with the `mod` function.

```
>> c0=mod(cd,2);
>> c1=mod(floor(cd/2),2);
>> c2=mod(floor(cd/4),2);
>> c3=mod(floor(cd/8),2);
>> c4=mod(floor(cd/16),2);
>> c5=mod(floor(cd/32),2);
>> c6=mod(floor(cd/64),2);
>> c7=mod(floor(cd/128),2);
```

These are all shown in figure 1.27. Note that the least significant bit plane, `c0`, is to all intents and purposes a random array, and that as the index value of the bit plane increases, more of the image appears. The most significant bit plane, `c7`, is actually a *threshold* of the image at level 127:

```
>> ct=c>127;
>> all(c7(:)==ct(:))

ans =

     1
```

We shall discuss thresholding in chapter 7.

We can recover and display the original image with

```
>> cc=2*(2*(2*(2*(2*(2*(2*c7+c6)+c5)+c4)+c3)+c2)+c1)+c0;
>> imshow(uint8(cc))
```

1.18 Spatial Resolution

Spatial resolution is the density of pixels over the image: the greater the spatial resolution, the more pixels are used to display the image. We can experiment with spatial resolution with MATLAB's

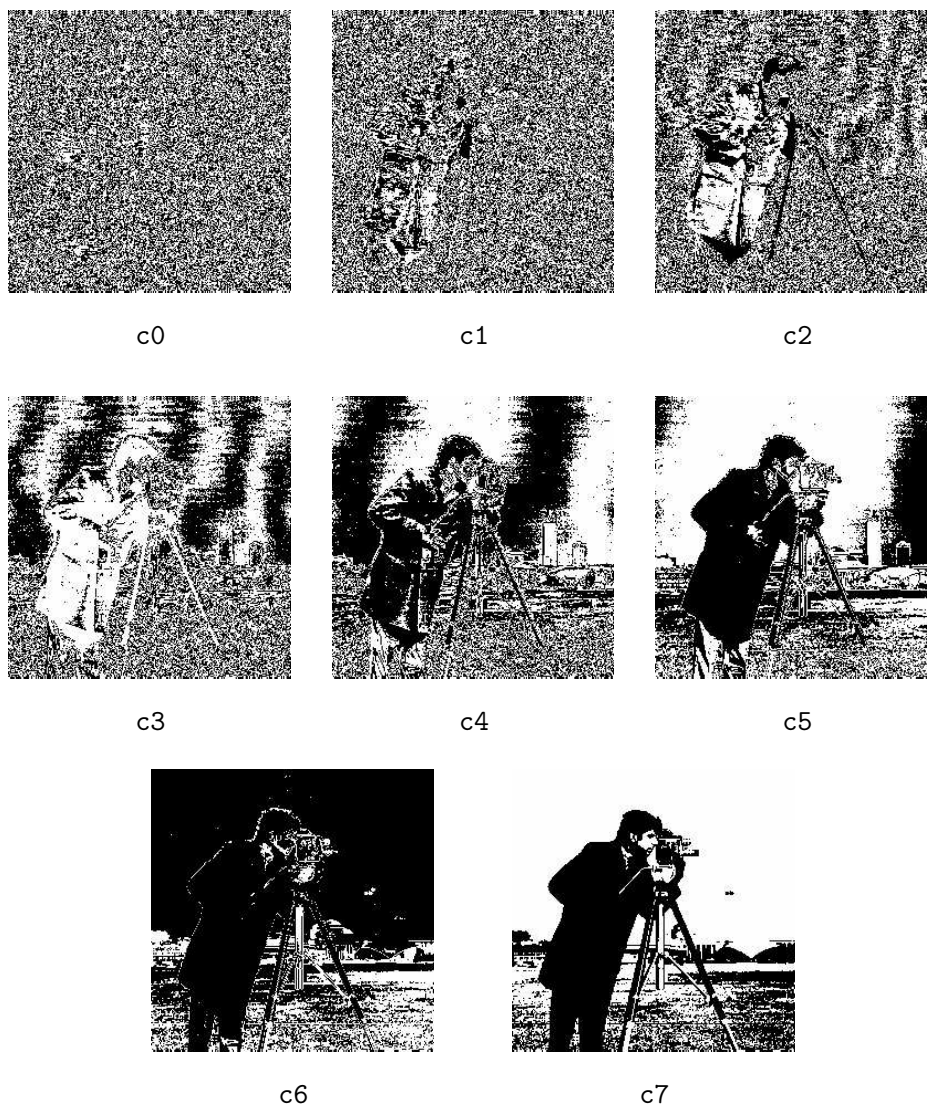


Figure 1.27: The bit planes of an 8-bit greyscale image

`imresize` function. Suppose we have an 256×256 8-bit greyscale image saved to the matrix `x`. Then the command

```
imresize(x,1/2);
```

will halve the size of the image. It does this by taking out every other row and every other column, thus leaving only those matrix elements whose row and column indices are even:

$$\begin{array}{cccccccc} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & \dots & \\ x_{21} & \boxed{x_{22}} & x_{23} & \boxed{x_{24}} & x_{25} & \boxed{x_{26}} & \dots & \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & \dots & \\ x_{41} & \boxed{x_{42}} & x_{43} & \boxed{x_{44}} & x_{45} & \boxed{x_{46}} & \dots & \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & \dots & \\ x_{61} & \boxed{x_{62}} & x_{63} & \boxed{x_{64}} & x_{65} & \boxed{x_{66}} & \dots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \end{array} \rightarrow \text{imresize}(x,1/2) \rightarrow \begin{array}{cccc} x_{22} & x_{24} & x_{26} & \dots \\ x_{42} & x_{44} & x_{46} & \dots \\ x_{62} & x_{64} & x_{66} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

If we apply `imresize` to the result with the parameter 2 rather than 1/2, all the pixels are repeated to produce an image with the same size as the original, but with half the resolution in each direction:

$$\begin{array}{ccc} \boxed{\begin{array}{cc} x_{22} & x_{22} \\ x_{22} & x_{22} \end{array}} & \boxed{\begin{array}{cc} x_{24} & x_{24} \\ x_{24} & x_{24} \end{array}} & \boxed{\begin{array}{cc} x_{26} & x_{26} \\ x_{26} & x_{26} \end{array}} & \dots \\ \boxed{\begin{array}{cc} x_{42} & x_{42} \\ x_{42} & x_{42} \end{array}} & \boxed{\begin{array}{cc} x_{44} & x_{44} \\ x_{44} & x_{44} \end{array}} & \boxed{\begin{array}{cc} x_{46} & x_{46} \\ x_{46} & x_{46} \end{array}} & \dots \\ \boxed{\begin{array}{cc} x_{62} & x_{62} \\ x_{62} & x_{62} \end{array}} & \boxed{\begin{array}{cc} x_{64} & x_{64} \\ x_{64} & x_{64} \end{array}} & \boxed{\begin{array}{cc} x_{66} & x_{66} \\ x_{66} & x_{66} \end{array}} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

The effective resolution of this new image is only 128×128 . We can do all this in one line:

```
x2=imresize(imresize(x,1/2),2);
```

By changing the parameters of `imresize`, we can change the effective resolution of the image to smaller amounts:

Command	Effective resolution
<code>imresize(imresize(x,1/4),4);</code>	64×64
<code>imresize(imresize(x,1/8),8);</code>	32×32
<code>imresize(imresize(x,1/16),16);</code>	16×16
<code>imresize(imresize(x,1/32),32);</code>	8×8

To see the effects of these commands, suppose we apply them to the image `newborn.tif`:

```
x=imread('newborn.tif');
```

The effects of increasing blockiness or *pixelization* become quite pronounced as the resolution decreases; even at 128×128 resolution fine detail, such as the edges of the baby's fingers, are less clear, and at 64×64 all edges are now quite blocky. At 32×32 the image is barely recognizable, and at 16×16 and 8×8 the image becomes unrecognizable.



(a) The original image



(b) at 128×128 resolution

Figure 1.28: Reducing resolution of an image



(a) At 64×64 resolution



(b) At 32×32 resolution

Figure 1.29: Further reducing the resolution of an image

Chapter 2

Point Processing

2.1 Introduction

Any image processing operation transforms the grey values of the pixels. However, image processing operations may be divided into into three classes based on the information required to perform the transformation. From the most complex to the simplest, they are:

1. **Transforms.** A “transform” represents the pixel values in some other, but equivalent form. Transforms allow for some very efficient and powerful algorithms, as we shall see later on. We may consider that in using a transform, the entire image is processed as a single large block. This may be illustrated by the diagram shown in figure 2.1.

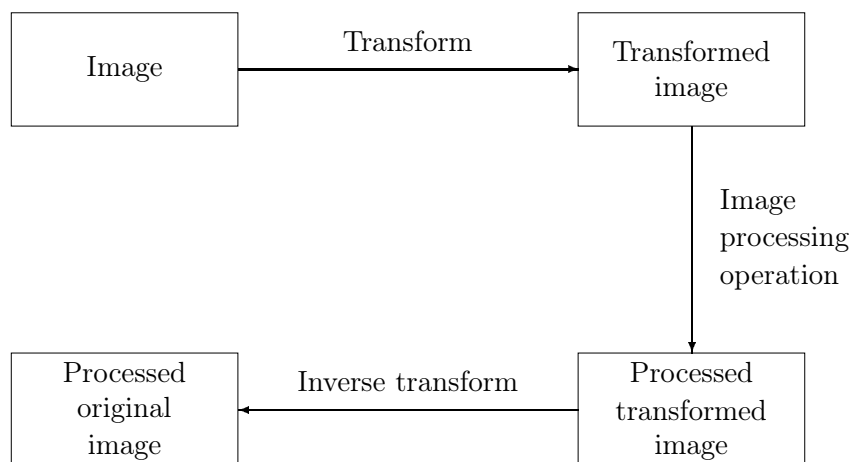


Figure 2.1: Schema for transform processing

2. **Neighbourhood processing.** To change the grey level of a given pixel we need only know the value of the grey levels in a small neighbourhood of pixels around the given pixel.
3. **Point operations.** A pixel’s grey value is changed without any knowledge of its surrounds.

Although point operations are the simplest, they contain some of the most powerful and widely used of all image processing operations. They are especially useful in image *pre-processing*, where an image is required to be modified before the main job is attempted.

2.2 Arithmetic operations

These operations act by applying a simple function

$$y = f(x)$$

to each grey value in the image. Thus $f(x)$ is a function which maps the range $0 \dots 255$ onto itself. Simple functions include adding or subtract a constant value to each pixel:

$$y = x \pm C$$

or multiplying each pixel by a constant:

$$y = Cx.$$

In each case we may have to fiddle the output slightly in order to ensure that the results are integers in the $0 \dots 255$ range. We can do this by first rounding the result (if necessary) to obtain an integer, and then “clipping” the values by setting:

$$y \leftarrow \begin{cases} 255 & \text{if } y > 255, \\ 0 & \text{if } y < 0. \end{cases}$$

We can obtain an understanding of how these operations affect an image by plotting $y = f(x)$. Figure 2.2 shows the result of adding or subtracting 128 from each pixel in the image. Notice that

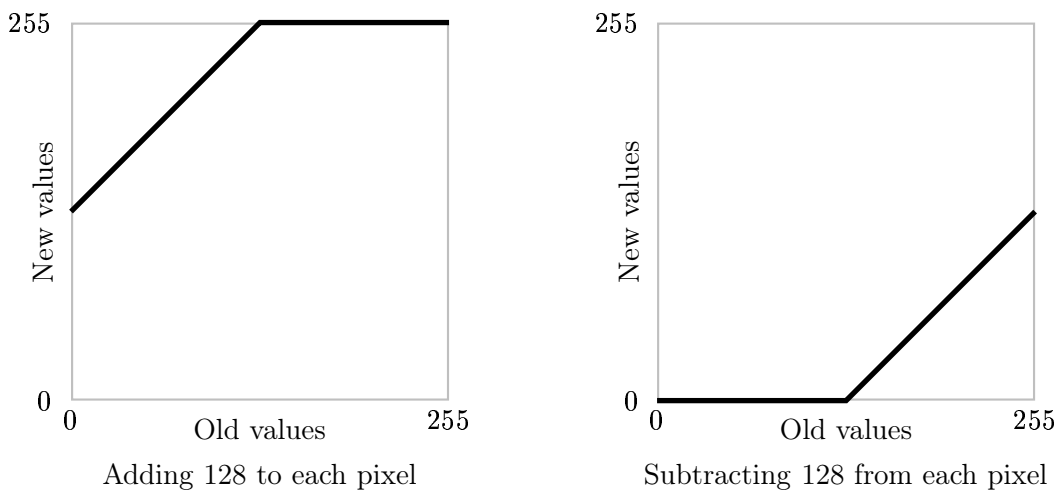


Figure 2.2: Adding and subtracting a constant

when we add 128, all grey values of 127 or greater will be mapped to 255. And when we subtract 128, all grey values of 128 or less will be mapped to 0. By looking at these graphs, we observe that in general adding a constant will lighten an image, and subtracting a constant will darken it.

We can test this on the “blocks” image `blocks.tif`, which we have seen in figure 1.4. We start by reading the image in:

```
>> b=imread('blocks.tif');
>> whos b
  Name      Size      Bytes  Class
  b         256x256   65536  uint8 array
```

The point of the second command was to find the numeric data type of `b`; it is `uint8`. The `uint8` data type is used for data *storage* only; we can't perform arithmetic operations. If we try, we just get an error message:

```
>> b1=b+128
??? Error using ==> +
Function '+' not defined for variables of class 'uint8'.
```

We can get round this in two ways. We can first turn `b` into a matrix of type `double`, add the 128, and then turn back to `uint8` for display:

```
>> b1=uint8(double(b)+128);
```

A second, and more elegant way, is to use the MATLAB function `imadd` which is designed precisely to do this:

```
>> b1=imadd(b,128);
```

Subtraction is similar; we can transform our matrix in and out of `double`, or use the `imsubtract` function:

```
>> b2=imsubtract(b,128);
```

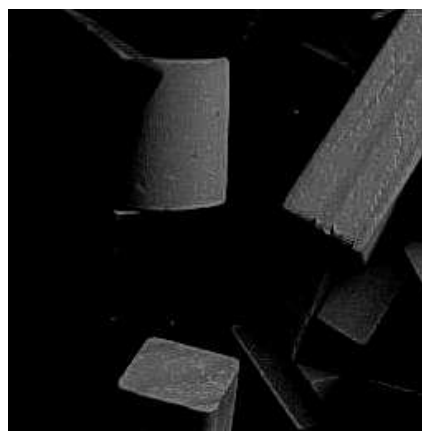
And now we can view them:

```
>> imshow(b1),figure,imshow(b2)
```

and the results are seen in figure 2.3.



b1: Adding 128



b2: Subtracting 128

Figure 2.3: Arithmetic operations on an image: adding or subtracting a constant

We can also perform lightening or darkening of an image by multiplication; figure 2.4 shows some examples of functions which will have these effects. To implement these functions, we use the `immultiply` function. Table 2.1 shows the particular commands required to implement the functions of figure 2.4. All these images can be viewed with `imshow`; they are shown in figure 2.5. Compare the results of darkening `b2` and `b3`. Note that `b3`, although darker than the original, is

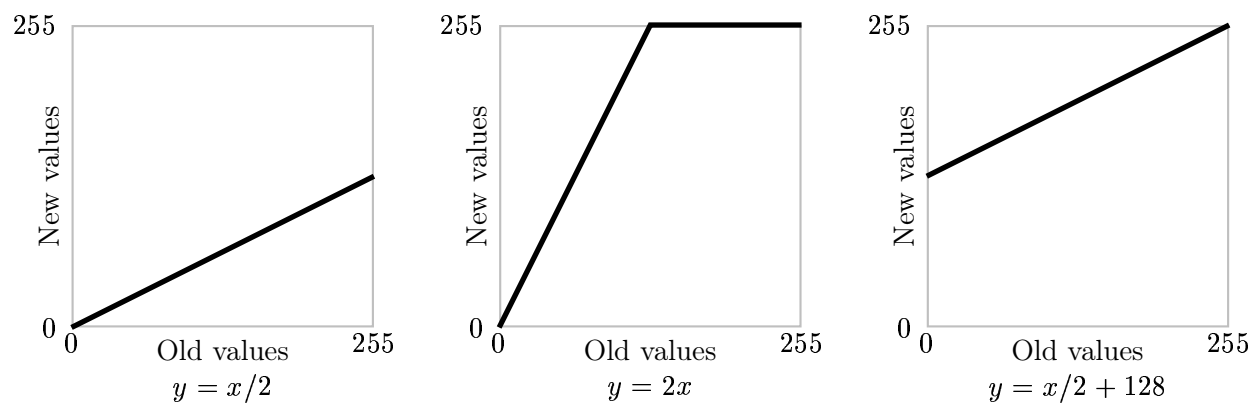


Figure 2.4: Using multiplication and division

```

 $y = x/2$       b3=immultiply(b,0.5); or b3=imdivide(b,2)
 $y = 2x$        b4=immultiply(b,2);
 $y = x/2 + 128$  b5=imadd(immultiply(b,0.5),128); or b5=imadd(imdivide(b,2),128);

```

Table 2.1: Implementing pixel multiplication by MATLAB commands

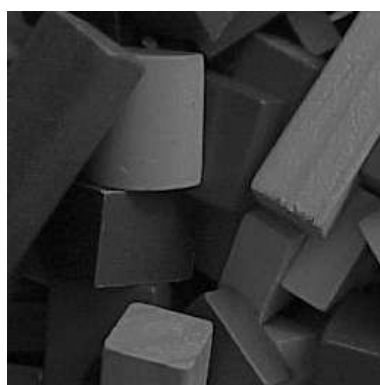
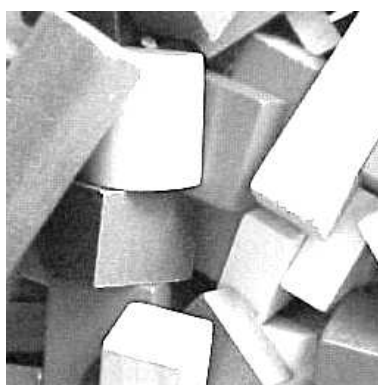
b3: $y = x/2$ b4: $y = 2x$ b5: $y = x/2 + 128$

Figure 2.5: Arithmetic operations on an image: multiplication and division

still quite clear, whereas a lot of information has been lost by the subtraction process, as can be seen in image **b2**. This is because in image **b2** all pixels with grey values 128 or less have become zero.

A similar loss of information has occurred in the images **b1** and **b4**. Note in particular the edges of the light coloured block in the bottom centre; in both **b1** and **b4** the right hand edge has disappeared. However, the edge is quite visible in image **b5**.

Complements

The *complement* of a greyscale image is its photographic negative. If an image matrix **m** is of type `double` and so its grey values are in the range 0.0 to 1.0, we can obtain its negative with the command

```
>> 1-m
```

If the image is binary, we can use

```
>> ~m
```

If the image is of type `uint8`, the best approach is the `imcomplement` function. Figure 2.6 shows the complement function $y = 255 - x$, and the result of the commands

```
>> bc=imcomplement(b);
>> imshow(bc)
```

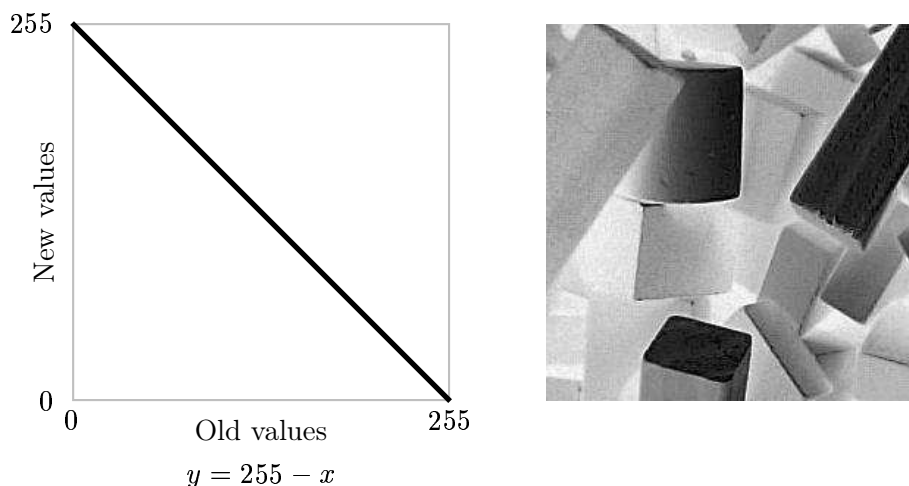


Figure 2.6: Image complementation

Interesting special effects can be obtained by complementing only *part* of the image; for example by taking the complement of pixels of grey value 128 or less, and leaving other pixels untouched. Or we could take the complement of pixels which are 128 or greater, and leave other pixels untouched. Figure 2.7 shows these functions. The effect of these functions is called *solarization*.

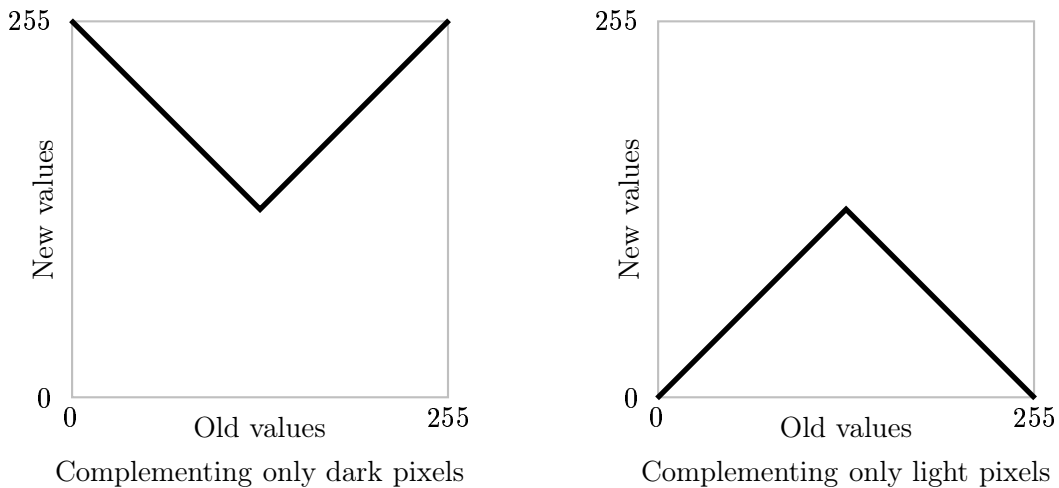


Figure 2.7: Part complementation

2.3 Histograms

Given a greyscale image, its *histogram* consists of the histogram of its grey levels; that is, a graph indicating the number of times each grey level occurs in the image. We can infer a great deal about the appearance of an image from its histogram, as the following examples indicate:

- In a dark image, the grey levels (and hence the histogram) would be clustered at the lower end:
- In a uniformly bright image, the grey levels would be clustered at the upper end:
- In a well contrasted image, the grey levels would be well spread out over much of the range:

We can view the histogram of an image in MATLAB by using the `imhist` function:

```
>> p=imread('pout.tif');
>> imshow(p),figure,imhist(p),axis tight
```

(the `axis tight` command ensures the axes of the histogram are automatically scaled to fit all the values in). The result is shown in figure 2.8. Since the grey values are all clustered together in the centre of the histogram, we would expect the image to be poorly contrasted, as indeed it is.

Given a poorly contrasted image, we would like to enhance its contrast, by spreading out its histogram. There are two ways of doing this.

2.3.1 Histogram stretching (Contrast stretching)

Suppose we have an image with the histogram shown in figure 2.9, associated with a table of the numbers n_i of grey values:

Grey level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n_i	15	0	0	0	0	70	110	45	70	35	0	0	0	0	0	15

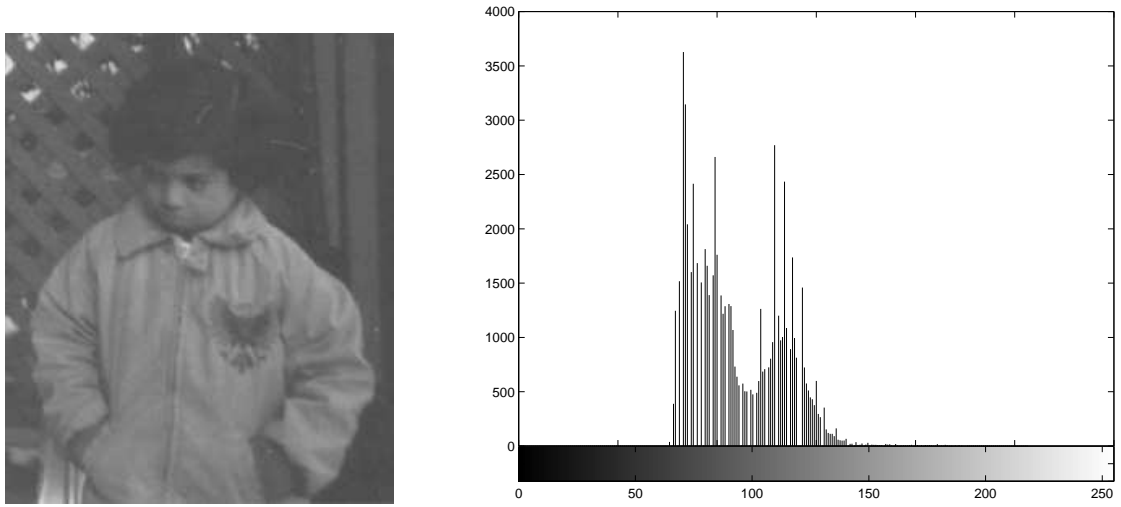
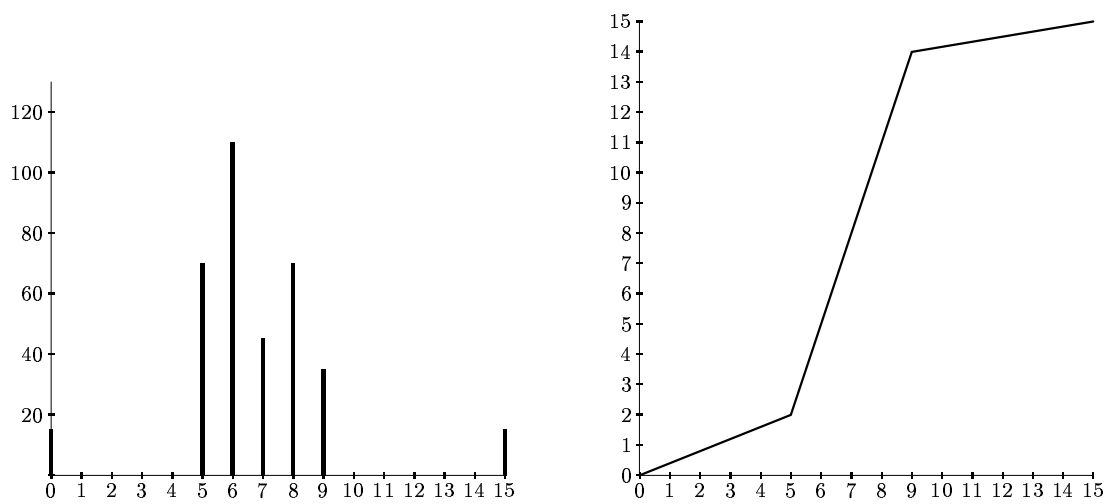
Figure 2.8: The image `pout.tif` and its histogram

Figure 2.9: A histogram of a poorly contrasted image, and a stretching function

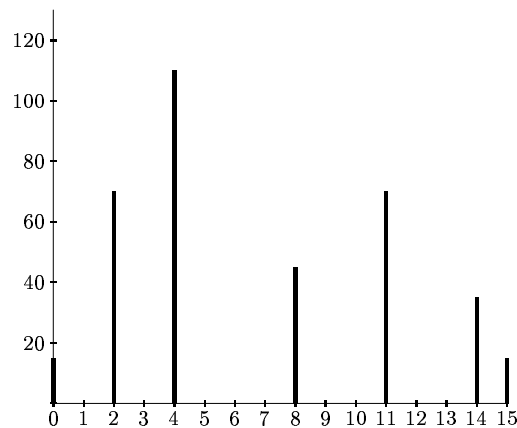
(with $n = 360$, as before.) We can stretch the grey levels in the centre of the range out by applying the piecewise linear function shown at the right in figure 2.9. This function has the effect of stretching the grey levels 5–9 to grey levels 2–14 according to the equation:

$$j = \frac{14 - 2}{9 - 5}(i - 5) + 2$$

where i is the original grey level and j its result after the transformation. Grey levels outside this range are either left alone (as in this case) or transformed according to the linear functions at the ends of the graph above. This yields:

i	5	6	7	8	9
j	2	5	8	11	14

and the corresponding histogram:



which indicates an image with greater contrast than the original.

Use of `imadjust`

To perform histogram stretching in MATLAB the `imadjust` function may be used. In its simplest incarnation, the command

```
imadjust(im, [a,b], [c,d])
```

stretches the image according to the function shown in figure 2.10. Since `imadjust` is designed to

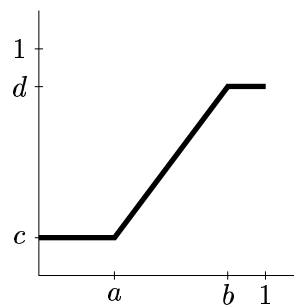


Figure 2.10: The stretching function given by `imadjust`

work equally well on images of type `double`, `uint8` or `uint16` the values of a , b , c and d must be between 0 and 1; the function automatically converts the image (if needed) to be of type `double`.

Note that `imadjust` does not work quite in the same way as shown in figure 2.9. Pixel values less than a are all converted to c , and pixel values greater than b are all converted to d . If either of $[a,b]$ or $[c,d]$ are chosen to be $[0,1]$, the abbreviation `[]` may be used. Thus, for example, the command

```
>> imadjust(im, [], [])
```

does nothing, and the command

```
>> imadjust(im, [], [1,0])
```

inverts the grey values of the image, to produce a result similar to a photographic negative.

The `imadjust` function has one other optional parameter: the *gamma* value, which describes the shape of the function between the coordinates (a, c) and (b, d) . If `gamma` is equal to 1, which is the default, then a linear mapping is used, as shown above in figure 2.10. However, values less than one produce a function which is concave downward, as shown on the left in figure 2.11, and values greater than one produce a figure which is concave upward, as shown on the right in figure 2.11.

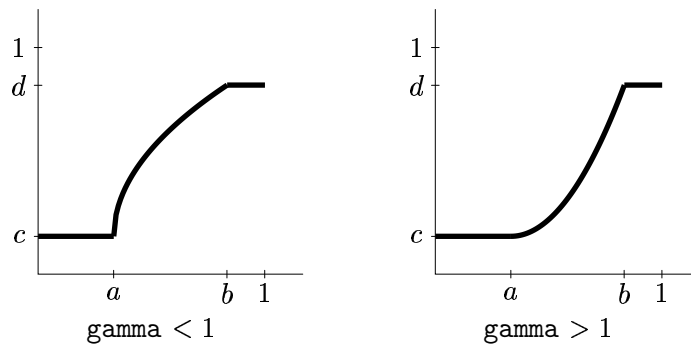


Figure 2.11: The `imadjust` function with `gamma` not equal to 1

The function used is a slight variation on the standard line between two points:

$$y = \left(\frac{x - a}{b - a} \right)^\gamma (d - c) + c.$$

Use of the `gamma` value alone can be enough to substantially change the appearance of the image. For example:

```
>> t=imread('tire.tif');
>> th=imadjust(t, [], [], 0.5);
>> imshow(t), figure, imshow(th)
```

produces the result shown in figure 2.12.

We may view the `imadjust` stretching function with the `plot` function. For example,

```
>> plot(t, th, '.'), axis tight
```

produces the plot shown in figure 2.13. Since `p` and `ph` are matrices which contain the original values and the values after the `imadjust` function, the `plot` function simply plots them, using dots to do it.

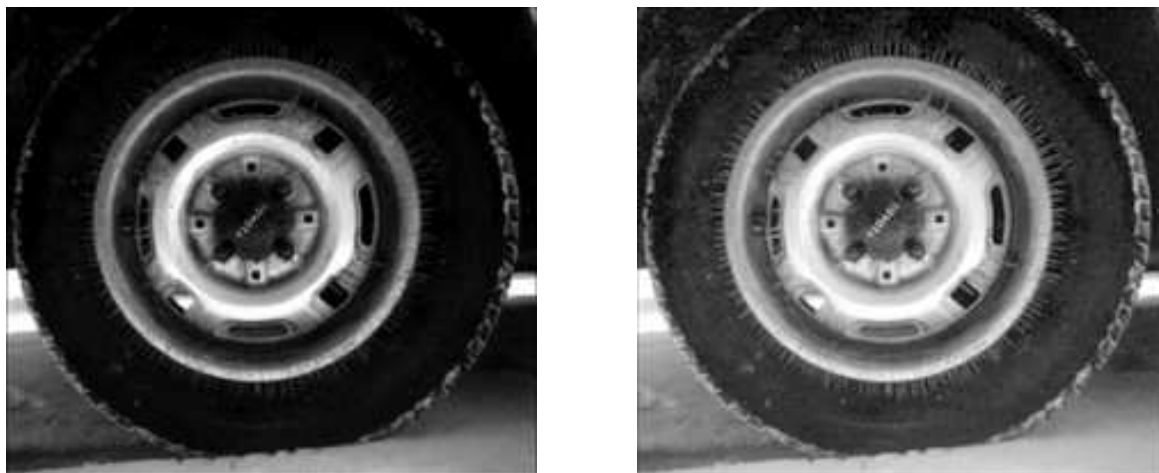


Figure 2.12: The tire image and after adjustment with the gamma value

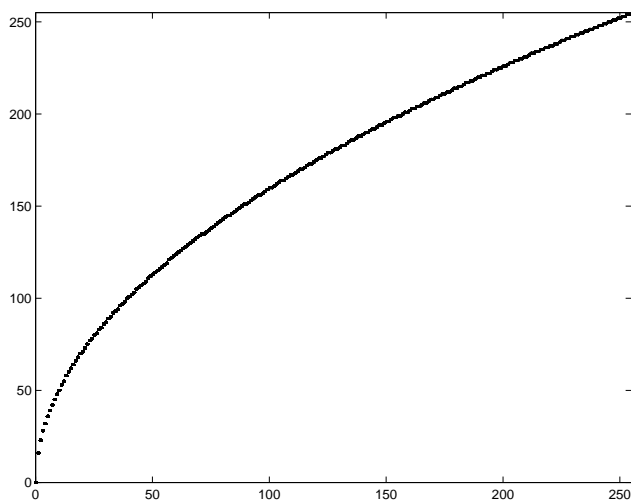


Figure 2.13: The function used in figure 2.12

A piecewise linear stretching function

We can easily write our own function to perform piecewise linear stretching as shown in figure 2.14. To do this, we will make use of the `find` function, to find the pixel values in the image between a_i and a_{i+1} . Since the line between the coordinates (a_i, b_i) and (a_{i+1}, b_{i+1}) has the equation

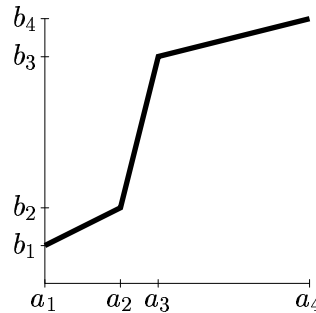


Figure 2.14: A piecewise linear stretching function

$$y = \frac{b_{i+1} - b_i}{a_{i+1} - a_i}(x - a_i) + b_i$$

the heart of our function will be the lines

```
pix=find(im >= a(i) & im < a(i+1));
out(pix)=(im(pix)-a(i))*(b(i+1)-b(i))/(a(i+1)-a(i))+b(i);
```

where `im` is the input image and `out` is the output image. A simple procedure which takes as inputs images of type `uint8` or `double` is shown in figure 2.15. As an example of the use of this function:

```
>> th=histpwl(t,[0 .25 .5 .75 1],[0 .75 .25 .5 1]);
>> imshow(th)
>> figure,plot(t,th,'.'),axis tight
```

produces the figures shown in figure 2.16.

2.3.2 Histogram equalization

The trouble with any of the above methods of histogram stretching is that they require user input. Sometimes a better approach is provided by *histogram equalization*, which is an entirely automatic procedure. The idea is to change the histogram to one which is uniform; that is that every bar on the histogram is of the same height, or in other words that each grey level in the image occurs with the same frequency. In practice this is generally not possible, although as we shall see the result of histogram equalization provides very good results.

Suppose our image has L different grey levels $0, 1, 2, \dots, L - 1$, and that grey level i occurs n_i times in the image. Suppose also that the total number of pixels in the image is n (so that $n_0 + n_1 + n_2 + \dots + n_{L-1} = n$). To transform the grey levels to obtain a better contrasted image, we change grey level i to

$$\left(\frac{n_0 + n_1 + \dots + n_i}{n} \right) (L - 1).$$

and this number is rounded to the nearest integer.

```

function out = histpwl(im,a,b)
%
% HISTPWL(IM,A,B) applies a piecewise linear transformation to the pixel values
% of image IM, where A and B are vectors containing the x and y coordinates
% of the ends of the line segments. IM can be of type UINT8 or DOUBLE,
% and the values in A and B must be between 0 and 1.
%
% For example:
%
%   histpwl(x,[0,1],[1,0])
%
% simply inverts the pixel values.
%
classChanged = 0;
if ~isa(im, 'double'),
    classChanged = 1;
    im = im2double(im);
end

if length(a) ~= length(b)
    error('Vectors A and B must be of equal size');
end

N=length(a);
out=zeros(size(im));

for i=1:N-1
    pix=find(im>=a(i) & im<a(i+1));
    out(pix)=(im(pix)-a(i))*(b(i+1)-b(i))/(a(i+1)-a(i))+b(i);
end

pix=find(im==a(N));
out(pix)=b(N);

if classChanged==1
    out = uint8(255*out);
end

```

Figure 2.15: A MATLAB function for applying a piecewise linear stretching function

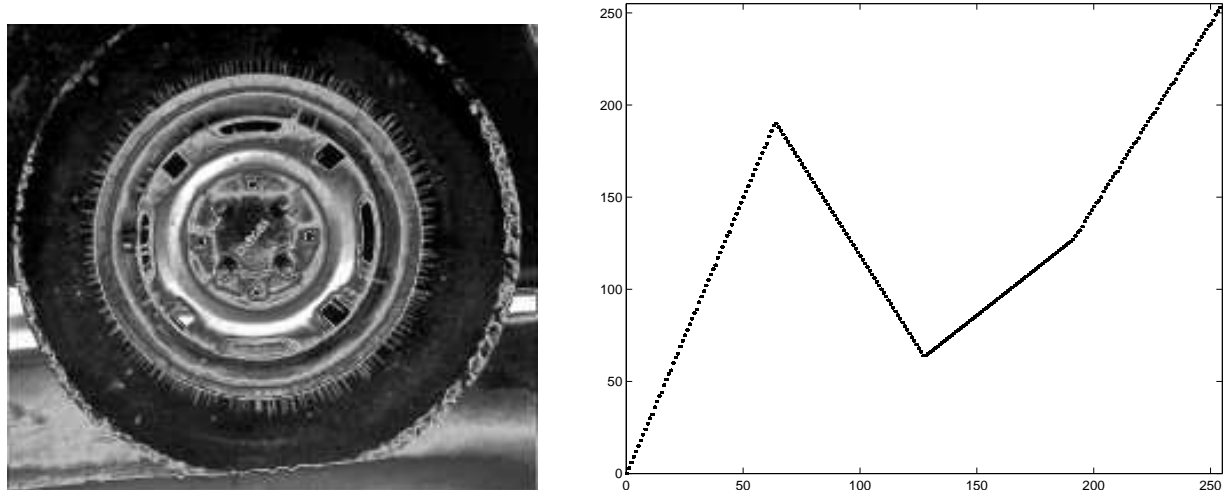


Figure 2.16: The tire image and after adjustment with the γ value

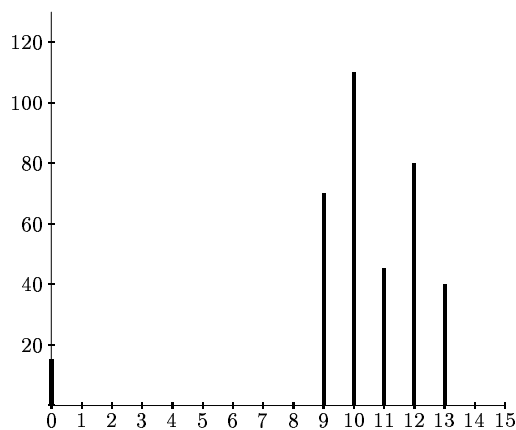


Figure 2.17: Another histogram indicating poor contrast

An example

Suppose a 4-bit greyscale image has the histogram shown in figure 2.17. associated with a table of the numbers n_i of grey values:

Grey level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n_i	15	0	0	0	0	0	0	0	0	70	110	45	80	40	0	0

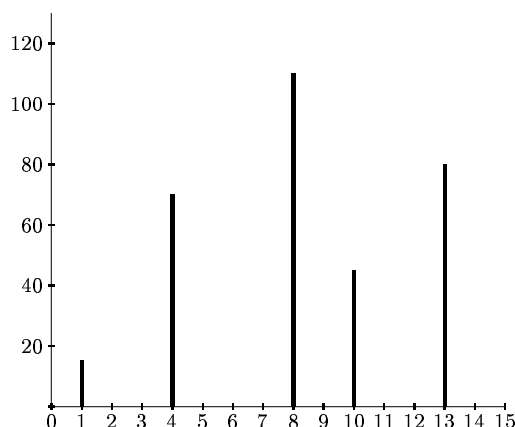


Figure 2.18: The histogram of figure 2.17 after equalization

(with $n = 360$.) We would expect this image to be uniformly bright, with a few dark dots on it. To equalize this histogram, we form running totals of the n_i , and multiply each by $15/360 = 1/24$:

Grey level i	n_i	Σn_i	$(1/24)\Sigma n_i$	Rounded value
0	15	15	0.63	1
1	0	15	0.63	1
2	0	15	0.63	1
3	0	15	0.63	1
4	0	15	0.63	1
5	0	15	0.63	1
6	0	15	0.63	1
7	0	15	0.63	1
8	0	15	0.63	1
9	70	85	3.65	4
10	110	195	8.13	8
11	45	240	10	10
12	80	320	13.33	13
13	40	360	15	15
14	0	360	15	15
15	0	360	15	15

We now have the following transformation of grey values, obtained by reading off the first and last columns in the above table:

Original grey level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Final grey level j	1	1	1	1	1	1	1	1	1	4	8	10	13	15	15	15

and the histogram of the j values is shown in figure 2.18. This is far more spread out than the original histogram, and so the resulting image should exhibit greater contrast.

To apply histogram equalization in MATLAB, use the `histeq` function; for example:

```
>> p=imread('pout.tif');
>> ph=histeq(p);
>> imshow(ph),figure,imhist(ph),axis tight
```

applies histogram equalization to the `pout` image, and produces the resulting histogram. These results are shown in figure 2.19. Notice the far greater spread of the histogram. This corresponds

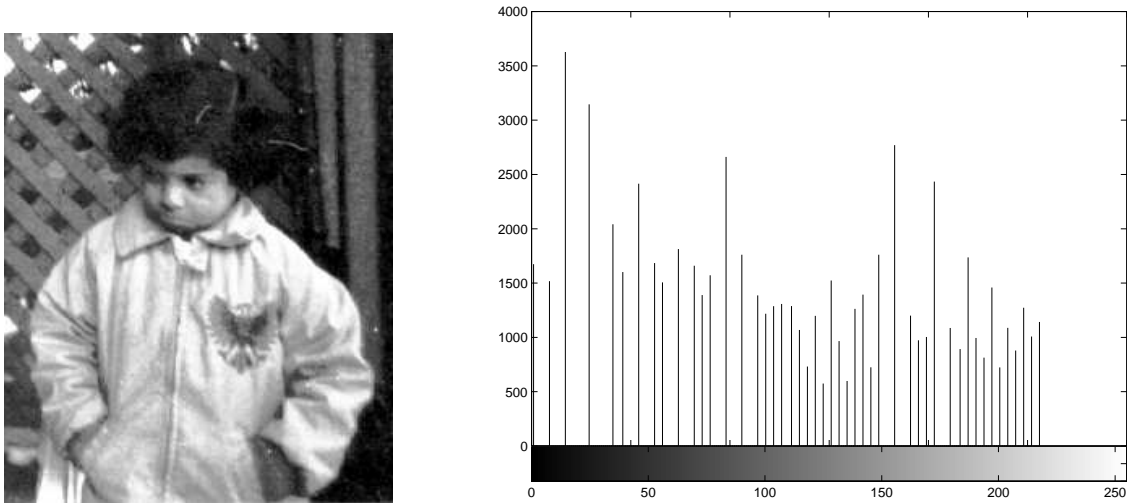


Figure 2.19: The histogram of figure 2.8 after equalization

to the greater increase of contrast in the image.

We give one more example, that of a very dark image. We can obtain a dark image by taking an image and using `imdivide`.

```
>> en=imread('engineer.tif');
>> e=imdivide(en,4);
```

Since the matrix `e` contains only low values it will appear very dark when displayed. We can display this matrix and its histogram with the usual commands:

```
>> imshow(e),figure,imhist(e),axis tight
```

and the results are shown in figure 2.20.

As you see, the very dark image has a corresponding histogram heavily clustered at the lower end of the scale.

But we can apply histogram equalization to this image, and display the results:

```
>> eh=histeq(e);
>> imshow(eh),figure,imhist(eh),axis tight
```

and the results are shown in figure 2.21.

Why it works

Consider the histogram in figure 2.17. To apply histogram stretching, we would need to stretch out the values between grey levels 9 and 13. Thus, we would need to apply a piecewise function similar to that shown in figure 2.9.

Let's consider the cumulative histogram, which is shown in figure 2.22. The dashed line is simply joining the top of the histogram bars. However, it can be interpreted as an appropriate histogram

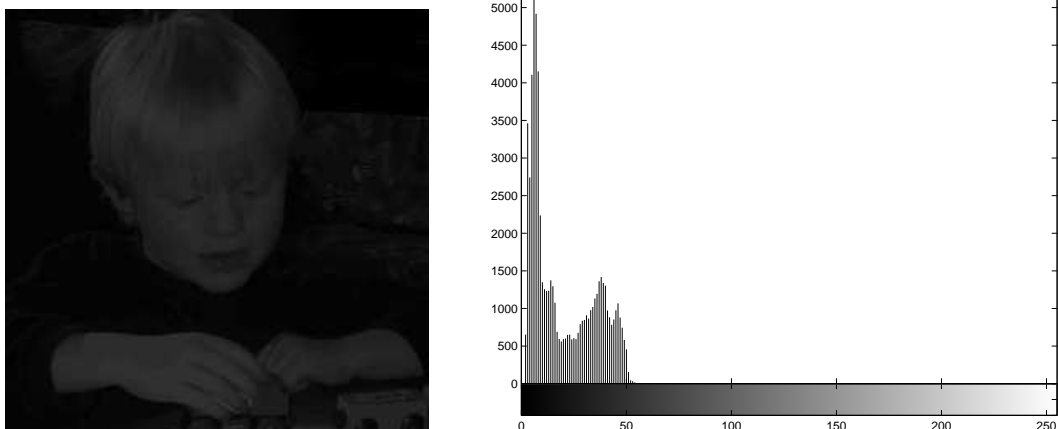


Figure 2.20: The darkened version of `engineer.tif` and its histogram

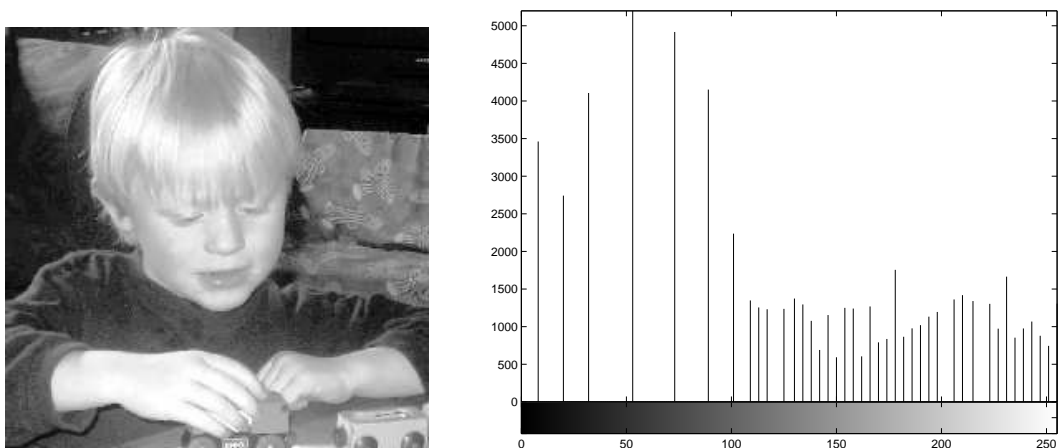


Figure 2.21: The image from 2.20 equalized and its histogram

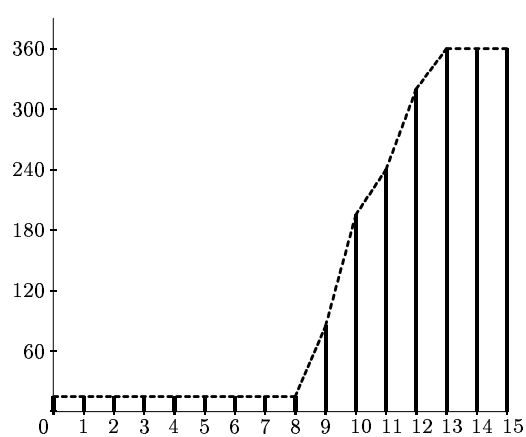


Figure 2.22: The cumulative histogram

stretching function. To do this, we need to scale the y values so that they are between 0 and 15, rather than 0 and 360. But this is precisely the method described in section 2.3.2.

As we have seen, none of the example histograms, after equalization, are uniform. This is a result of the discrete nature of the image. If we were to treat the image as a continuous function $f(x, y)$, and the histogram as the area between different contours (see for example Castleman [1], then we can treat the histogram as a probability density function. But the corresponding cumulative density function will always have a uniform histogram; see for example Hogg and Craig [6].

2.4 Lookup tables

Point operations can be performed very effectively by the use of a *lookup table*, known more simply as an LUT. For operating on images of type `uint8`, such a table consists of a single array of 256 values, each value of which is an integer in the range $0 \dots 255$. Then our operation can be implemented by replacing each pixel value p by the corresponding value t_p in the table.

For example, the LUT corresponding to division by 2 looks like:

Index:	0	1	2	3	4	5	...	250	251	252	253	254	255
LUT:	0	0	1	1	2	2	...	125	125	126	126	127	127

This means, for example, that a pixel with value 4 will be replaced with 2; a pixel with value 253 will be replaced with value 126.

If `T` is a lookup table in MATLAB, and `im` is our image, the the lookup table can be applied by the simple command

```
T(im)
```

For example, suppose we wish to apply the above lookup table to the `blocks` image. We can create the table with

```
>> T=uint8(floor(0:255)/2);
```

apply it to the `blocks` image `b` with

```
>> b2=T(b);
```

The image `b2` is of type `uint8`, and so can be viewed directly with `imshow`.

As another example, suppose we wish to apply an LUT to implement the contrast stretching function shown in figure 2.23. Given the equation used in section 2.3.1, the equations of the three lines used are:

$$y = \frac{64}{96}x,$$

$$y = \frac{192 - 64}{160 - 96}(x - 96) + 64,$$

$$y = \frac{255 - 192}{255 - 160}(x - 160) + 192,$$

and these equations can be written more simply as

$$y = 0.6667x,$$

$$y = 2x - 128,$$

$$y = 0.6632x + 85.8947.$$

Chapter 3

Neighbourhood Processing

3.1 Introduction

We have seen in chapter 2 that an image can be modified by applying a particular function to each pixel value. Neighbourhood processing may be considered as an extension of this, where a function is applied to a neighbourhood of each pixel.

The idea is to move a “mask”: a rectangle (usually with sides of odd length) or other shape over the given image. As we do this, we create a new image whose pixels have grey values calculated from the grey values under the mask, as shown in figure 3.1. The combination of mask and function

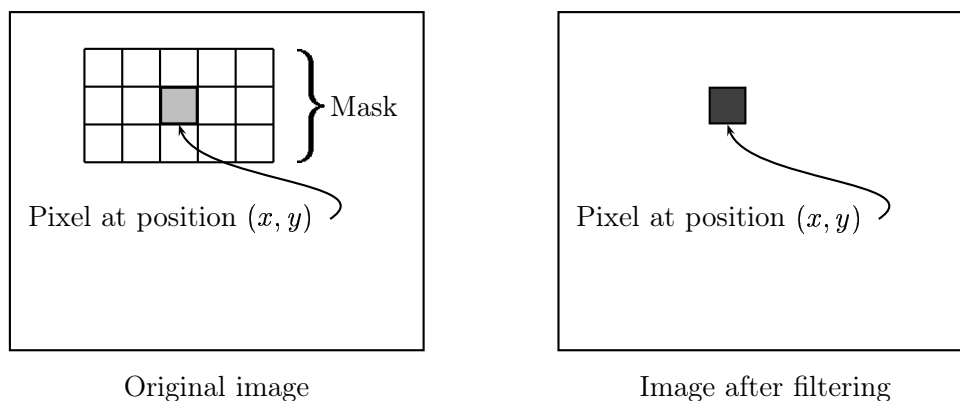


Figure 3.1: Using a spatial mask on an image

is called a *filter*. If the function by which the new grey value is calculated is a linear function of all the grey values in the mask, then the filter is called a *linear filter*.

A linear filter can be implemented by multiplying all elements in the mask by corresponding elements in the neighbourhood, and adding up all these products. Suppose we have a 3×5 mask as illustrated in figure 3.1. Suppose that the mask values are given by:

$m(-1, -2)$	$m(-1, -1)$	$m(-1, 0)$	$m(-1, 1)$	$m(-1, 2)$
$m(0, -2)$	$m(0, -1)$	$m(0, 0)$	$m(0, 1)$	$m(0, 2)$
$m(1, -2)$	$m(1, -1)$	$m(1, 0)$	$m(1, 1)$	$m(1, 2)$

and that corresponding pixel values are

$p(i-1, j-2)$	$p(i-1, j-1)$	$p(i-1, j)$	$p(i-1, j+1)$	$p(i-1, j+2)$
$p(i, j-2)$	$p(i, j-1)$	$p(i, j)$	$p(i, j+1)$	$p(i, j+2)$
$p(i+1, j-2)$	$p(i+1, j-1)$	$p(i+1, j)$	$p(i+1, j+1)$	$p(i+1, j+2)$

We now multiply and add:

$$\sum_{s=-1}^1 \sum_{t=-2}^2 m(s, t)p(i+s, j+t).$$

A diagram illustrating the process for performing spatial filtering is given in figure 3.2.

Spatial filtering thus requires three steps:

1. position the mask over the current pixel,
2. form all products of filter elements with the corresponding elements of the neighbourhood,
3. add up all the products.

This must be repeated for every pixel in the image.

Allied to spatial filtering is spatial *convolution*. The method for performing a convolution is the same as that for filtering, except that the filter must be rotated by 180° before multiplying and adding. Using the $m(i, j)$ and $p(i, j)$ notation as before, the output of a convolution with a 3×5 mask for a single pixel is

$$\sum_{s=-1}^1 \sum_{t=-2}^2 m(-s, -t)p(i+s, j+t).$$

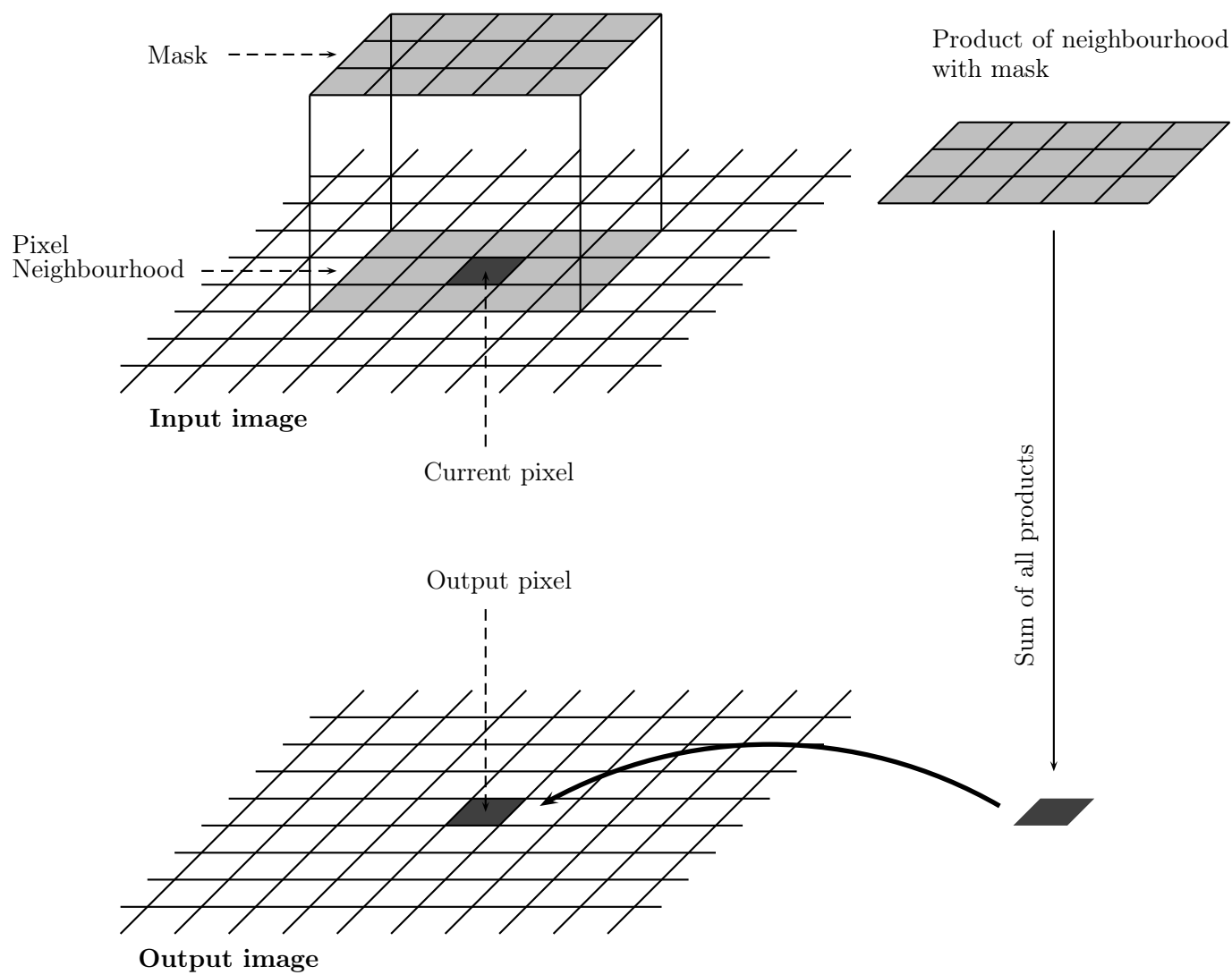


Figure 3.2: Performing spatial filtering

Note the negative signs on the indices of m . The same result can be achieved with

$$\sum_{s=-1}^1 \sum_{t=-2}^2 m(s,t)p(i-s+j-t).$$

Here we have rotated the *image* pixels by 180° ; this does not of course affect the result. The importance of convolution will become apparent when we investigate the Fourier transform, and the convolution theorem. Note also that in practice, most filter masks are rotationally symmetric, so that spatial filtering and spatial convolution will produce the same output.

An example: One important linear filter is to use a 3×3 mask and take the average of all nine values within the mask. This value becomes the grey value of the corresponding pixel in the new image. This operation may be described as follows:

$$\begin{array}{|c|c|c|} \hline & & \\ \hline a & b & c \\ \hline d & e & f \\ \hline g & h & i \\ \hline & & \\ \hline \end{array} \rightarrow \frac{1}{9}(a + b + c + d + e + f + g + h + i)$$

where e is grey value of the current pixel in the original image, and the average is the grey value of the corresponding pixel in the new image.

To apply this to an image, consider the 5×5 “image” obtained by:

```
>> x=uint8(10*magic(5))

x =

    170    240     10     80    150
    230     50     70    140    160
     40     60    130    200    220
    100    120    190    210     30
    110    180    250     20     90
```

We may regard this array as being made of nine overlapping 3×3 neighbourhoods. The output of our working will thus consist only of nine values. We shall see later how to obtain 25 values in the output.

Consider the top left 3×3 neighbourhood of our image x :

170	240	10	80	150
230	50	70	140	160
40	60	130	200	220
100	120	190	210	30
110	180	250	20	90

Now we take the average of all these values:

```
>> mean2(x(1:3,1:3))

ans =

    111.1111
```

which can be rounded to 111. Now we can move to the second neighbourhood:

170	240	10	80	150
230	50	70	140	160
40	60	130	200	220
100	120	190	210	30
110	180	250	20	90

and take its average:

```
>> mean2(x(1:3,2:4))

ans =

    108.8889
```

and this can be rounded either down to 108, or to the nearest integer 109. If we continue in this manner, the following output is obtained:

```
111.1111  108.8889  128.8889
110.0000  130.0000  150.0000
131.1111  151.1111  148.8889
```

This array is the result of filtering \mathbf{x} with the 3×3 averaging filter.

3.2 Notation

It is convenient to describe a linear filter simply in terms of the coefficients of all the grey values of pixels within the mask. This can be written as a matrix.

The averaging filter above, for example, could have its output written as

$$\frac{1}{9}a + \frac{1}{9}b + \frac{1}{9}c + \frac{1}{9}d + \frac{1}{9}e + \frac{1}{9}f + \frac{1}{9}g + \frac{1}{9}h + \frac{1}{9}i$$

and so this filter can be described by the matrix

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

An example: The filter

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

would operate on grey values as

a	b	c		→ a - 2b + c - 2d + 4e - 2d + g - 2h + i
d	e	f		
g	h	i		

Edges of the image

There is an obvious problem in applying a filter—what happens at the edge of the image, where the mask partly falls outside the image? In such a case, as illustrated in figure 3.3 there will be a lack of grey values to use in the filter function.

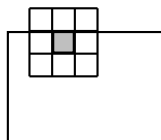


Figure 3.3: A mask at the edge of an image

There are a number of different approaches to dealing with this problem:

Ignore the edges. That is, the mask is only applied to those pixels in the image for which the mask will lie fully within the image. This means all pixels except for the edges, and results in an output image which is smaller than the original. If the mask is very large, a significant amount of information may be lost by this method.

We applied this method in our example above.

“Pad” with zeros. We assume that all necessary values outside the image are zero. This gives us all values to work with, and will return an output image of the same size as the original, but may have the effect of introducing unwanted artifacts (for example, edges) around the image.

3.3 Filtering in MATLAB

The `filter2` function does the job of linear filtering for us; its use is

```
filter2(filter,image,shape)
```

and the result is a matrix of data type `double`. The parameter `shape` is optional, it describes the method for dealing with the edges:

- `filter2(filter,image,'same')` is the default; it produces a matrix of equal size to the original `image` matrix. It uses zero padding:


```
>> a=ones(3,3)/9

a =

    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111

>> filter2(a,x,'same')

ans =

    76.6667    85.5556    65.5556    67.7778    58.8889
    87.7778   111.1111   108.8889   128.8889   105.5556
    66.6667   110.0000   130.0000   150.0000   106.6667
    67.7778   131.1111   151.1111   148.8889    85.5556
    56.6667   105.5556   107.7778    87.7778    38.8889
```

- `filter2(filter,image,'valid')` applies the mask only to “inside” pixels. The result will always be smaller than the original:

```
>> filter2(a,x,'valid')

ans =

   111.1111   108.8889   128.8889
   110.0000   130.0000   150.0000
   131.1111   151.1111   148.8889
```

The result of 'same' above may also be obtained by padding with zeros and using 'valid':

```
>> x2=zeros(7,7);
>> x2(2:6,2:6)=x

x2 =

     0     0     0     0     0     0     0
     0    170    240    10     80    150     0
     0    230     50     70    140    160     0
     0     40     60    130    200    220     0
     0    100    120    190    210     30     0
     0    110    180    250     20     90     0
     0     0     0     0     0     0     0

>> filter2(a,x2,'valid')
```

- `filter2(filter,image,'full')` returns a result *larger* than the original; it does this by padding with zero, and applying the filter at all places on and around the image where the mask intersects the image matrix.

```
>> filter2(a,x,'full')

ans =

    18.8889    45.5556    46.6667    36.6667    26.6667    25.5556    16.6667
    44.4444    76.6667    85.5556    65.5556    67.7778    58.8889    34.4444
    48.8889    87.7778   111.1111   108.8889   128.8889   105.5556   58.8889
    41.1111    66.6667   110.0000   130.0000   150.0000   106.6667   45.5556
    27.7778    67.7778   131.1111   151.1111   148.8889    85.5556   37.7778
    23.3333    56.6667   105.5556   107.7778    87.7778    38.8889   13.3333
    12.2222    32.2222    60.0000    50.0000    40.0000    12.2222   10.0000
```

The `shape` parameter, being optional, can be omitted; in which case the default value is `'same'`.

There is no single “best” approach; the method must be dictated by the problem at hand; by the filter being used, and by the result required.

We can create our filters by hand, or by using the `fspecial` function; this has many options which makes for easy creation of many different filters. We shall use the `average` option, which produces averaging filters of given size; thus

```
>> fspecial('average',[5,7])
```

will return an averaging filter of size 5×7 ; more simply

```
>> fspecial('average',11)
```

will return an averaging filter of size 11×11 . If we leave out the final number or vector, the 3×3 averaging filter is returned.

For example, suppose we apply the 3×3 averaging filter to an image as follows:

```
>> c=imread('cameraman.tif');
>> f1=fspecial('average');
>> cf1=filter2(f1,c);
```

We now have a matrix of data type double. To display this, we can do any of the following:

- transform it to a matrix of type `uint8`, for use with `imshow`,
- divide its values by 255 to obtain a matrix with values in the 0.1–1.0 range, for use with `imshow`,
- use `mat2gray` to scale the result for display. We shall discuss the use of this function later.

Using the second method:

```
>> figure,imshow(c),figure,imshow(cf1/255)
```

will produce the images shown in figures 3.4(a) and 3.4(b).

The averaging filter blurs the image; the edges in particular are less distinct than in the original. The image can be further blurred by using an averaging filter of larger size. This is shown in figure 3.4(c), where a 9×9 averaging filter has been used, and in figure 3.4(d), where a 25×25 averaging filter has been used.



(a) Original image



(b) Average filtering



(c) Using a 9×9 filter



(d) Using a 25×25 filter

Figure 3.4: Average filtering

Notice how the zero padding used at the edges has resulted in a dark border appearing around the image. This is especially noticeable when a large filter is being used. If this is an unwanted artifact of the filtering; if for example it changes the average brightness of the image, then it may

be more appropriate to use the 'valid' shape option.

The resulting image after these filters may appear to be much “worse” than the original. However, applying a blurring filter to reduce detail in an image may be the perfect operation for autonomous machine recognition, or if we are only concentrating on the “gross” aspects of the image: numbers of objects; amount of dark and light areas. In such cases, too much detail may obscure the outcome.

Separable filters

Some filters can be implemented by the successive application of two simpler filters. For example, since

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

the 3×3 averaging filter can be implemented by first applying a 3×1 averaging filter, and then applying a 1×3 averaging filter to the result. The 3×3 averaging filter is thus *separable* into two smaller filters. Separability can result in great time savings. Suppose an $n \times n$ filter is separable into two filters of size $n \times 1$ and $1 \times n$. The application of an $n \times n$ filter requires n^2 multiplications, and $n^2 - 1$ additions for each pixel in the image. But the application of an $n \times 1$ filter only requires n multiplications and $n - 1$ additions. Since this must be done twice, the total number of multiplications and additions are $2n$ and $2n - 2$ respectively. If n is large the savings in efficiency can be dramatic.

All averaging filters are separable; another separable filter is the laplacian

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}.$$

Other examples will be considered below.

3.4 Frequencies; low and high pass filters

It will be convenient to have some standard terminology by which we can discuss the effects a filter will have on an image, and to be able to choose the most appropriate filter for a given image processing task. One important aspect of an image which enables us to do this is the notion of *frequencies*. Roughly speaking, the frequencies of an image are a measure of the amount by which grey values change with distance. This concept will be given a more formal setting in chapter 4. *High frequency components* are characterized by large changes in grey values over small distances; example of high frequency components are edges and noise. *Low frequency components*, on the other hand, are parts of the image characterized by little change in the grey values. These may include backgrounds, skin textures. We then say that a filter is a

high pass filter if it “passes over” the high frequency components, and reduces or eliminates low frequency components,

low pass filter if it “passes over” the low frequency components, and reduces or eliminates high frequency components,

For example, the 3×3 averaging filter is low pass filter, as it tends to blur edges. The filter

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

is a high pass filter.

We note that the sum of the coefficients (that is, the sum of all elements in the matrix), in the high pass filter is zero. This means that in a low frequency part of an image, where the grey values are similar, the result of using this filter is that the corresponding grey values in the new image will be close to zero. To see this, consider a 4×4 block of similar values pixels, and apply the above high pass filter to the central four:

150	152	148	149			
147	152	151	150	→	11	6
152	148	149	151		-13	-5
151	149	150	148			

The resulting values are close to zero, which is the expected result of applying a high pass filter to a low frequency component. We shall see how to deal with negative values below.

High pass filters are of particular value in edge detection and edge enhancement (of which we shall see more in chapter 8). But we can provide a sneak preview, using the cameraman image.

```
>> f=fspecial('laplacian')

f =

    0.1667    0.6667    0.1667
    0.6667   -3.3333    0.6667
    0.1667    0.6667    0.1667

>> cf=filter2(f,c);
>> imshow(cf/100)
>> f1=fspecial('log')

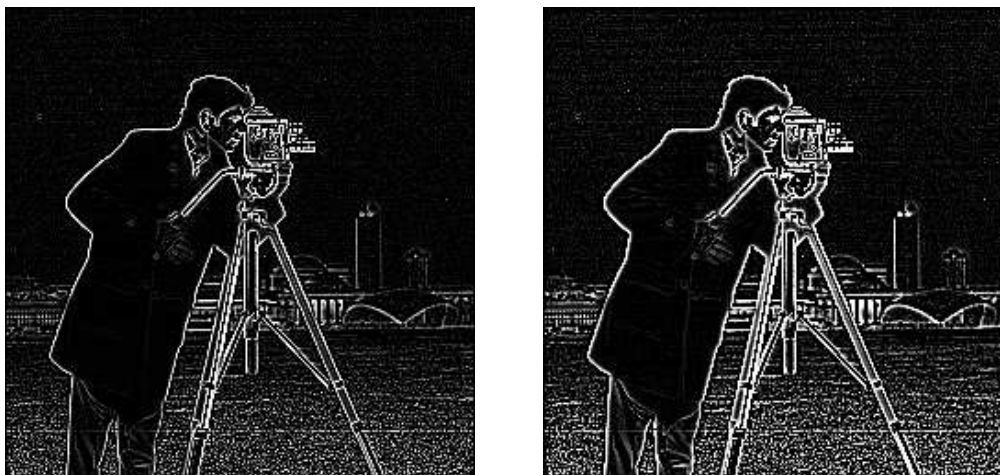
f1 =

    0.0448    0.0468    0.0564    0.0468    0.0448
    0.0468    0.3167    0.7146    0.3167    0.0468
    0.0564    0.7146   -4.9048    0.7146    0.0564
    0.0468    0.3167    0.7146    0.3167    0.0468
    0.0448    0.0468    0.0564    0.0468    0.0448

>> cf1=filter2(f1,c);
>> figure,imshow(cf1/100)
```

The images are shown in figure 3.5. Image (a) is the result of the Laplacian filter; image (b) shows the result of the Laplacian of Gaussian (“log”) filter.

In each case, the sum of all the filter elements is zero.



(a) Laplacian filter

(b) Laplacian of Gaussian (“log”) filtering

Figure 3.5: High pass filtering

Values outside the range 0–255

We have seen that for image display, we would like the grey values of the pixels to lie between 0 and 255. However, the result of applying a linear filter may be values which lie outside this range. We may consider ways of dealing with values outside of this “displayable” range.

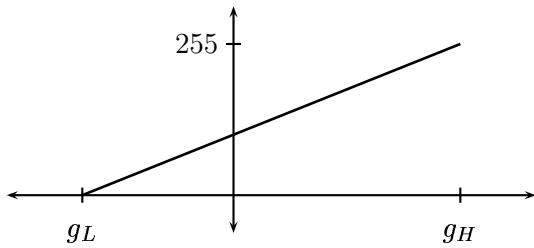
Make negative values positive. This will certainly deal with negative values, but not with values greater than 255. Hence, this can only be used in specific circumstances; for example, when there are only a few negative values, and when these values are themselves close to zero.

Clip values. We apply the following thresholding type operation to the grey values x produced by the filter to obtain a displayable value y :

$$y = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 255 \\ 255 & \text{if } x > 255 \end{cases}$$

This will produce an image with all pixel values in the required range, but is not suitable if there are many grey values outside the 0–255 range; in particular, if the grey values are equally spread over a larger range. In such a case this operation will tend to destroy the results of the filter.

Scaling transformation. Suppose the lowest grey value produced by the filter is g_L and the highest value is g_H . We can transform all values in the range g_L – g_H to the range 0–255 by the linear transformation illustrated below:



Since the gradient of the line is $255/(g_H - g_L)$ we can write the equation of the line as

$$y = 255 \frac{x - g_L}{g_H - g_L}$$

and applying this transformation to all grey levels x produced by the filter will result (after any necessary rounding) in an image which can be displayed.

As an example, let's apply the high pass filter given in section 3.4 to the cameraman image:

```
>> f2=[1 -2 1;-2 4 -2;1 -2 1];
>> cf2=filter2(f2,c);
```

Now the maximum and minimum values of the matrix `cf2` are 593 and -541 respectively. The `mat2gray` function automatically scales the matrix elements to displayable values; for any matrix M , it applies a linear transformation to its elements, with the lowest value mapping to 0.0, and the highest value mapping to 1.0. This means the output of `mat2gray` is always of type `double`. The function also requires that the input type is `double`.

```
>> figure,imshow(mat2gray(cf2));
```

To do this by hand, so to speak, applying the linear transformation above, we can use:

```
>> maxcf2=max(cf2(:));
>> mincf2=min(cf2(:));
>> cf2g=(cf2-mincf2)/(maxcf2-mincf2);
```

The result will be a matrix of type `double`, with entries in the range 0.0–1.0. This can be viewed with `imshow`. We can make it a `uint8` image by multiplying by 255 first. The result can be seen in figure 3.6.

We can generally obtain a better result by dividing the result of the filtering by a constant before displaying it:

```
>> figure,imshow(cf2/60)
```

and this is also shown in figure 3.6.

High pass filters are often used for edge detection. These can be seen quite clearly in the right hand image of figure 3.6.

Using `mat2gray`

Dividing by a constant

Figure 3.6: Using a high pass filter and displaying the result

3.5 Edge sharpening

Spatial filtering can be used to make edges in an image slightly sharper and crisper, which generally results in an image more pleasing to the human eye. The operation is variously called “edge enhancement”, “edge crispening”, or “unsharp masking”. This last term comes from the printing industry.

Unsharp masking

The idea of unsharp masking is to subtract a scaled “unsharp” version of the image from the original. In practice, we can achieve this affect by subtracting a scaled blurred image from the original. The schema for unsharp masking is shown in figure 3.7.

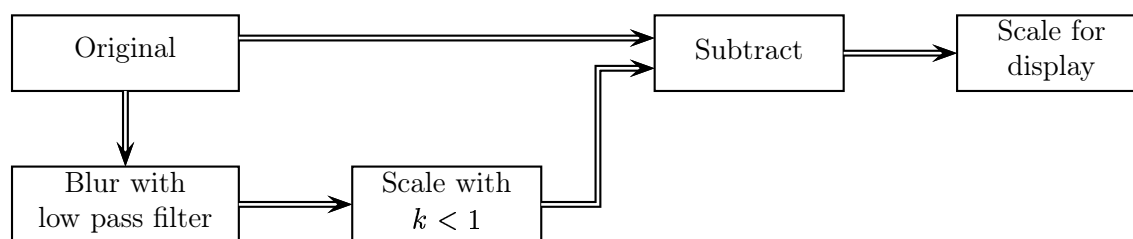


Figure 3.7: Schema for unsharp masking

Suppose an image `x` is of type `uint8`. The unsharp masking can be applied by the following sequence of commands:

```

>> f=fspecial('average');
>> xf=filter2(f,x);
>> xu=double(x)-xf/1.5
>> imshow(xu/70)
  
```


The last command scales the result so that `imshow` displays an appropriate image; the value may need to be adjusted according to the input image. Suppose that `x` is the image shown in figure 3.8(a), then the result of unsharp masking is given in figure 3.8(b). The result appears to be a better image than the original; the edges are crisper and more clearly defined.



(a) Original image



(b) The image after unsharp masking

Figure 3.8: An example of unsharp masking

To see why this works, we may consider the function of grey values as we travel across an edge, as shown in figure 3.9.

As a scaled blur is subtracted from the original, the result is that the edge is enhanced, as shown in graph (c) of figure 3.9.

We can in fact perform the filtering and subtracting operation in one command, using the linearity of the filter, and that the 3×3 filter

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

is the “identity filter”.

Hence unsharp masking can be implemented by a filter of the form

$$f = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{k} \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

where k is a constant chosen to provide the best result. Alternatively, the unsharp masking filter may be defined as

$$f = k \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

so that we are in effect subtracting a blur from a scaled version of the original; the scaling factor may also be split between the identity and blurring filters.

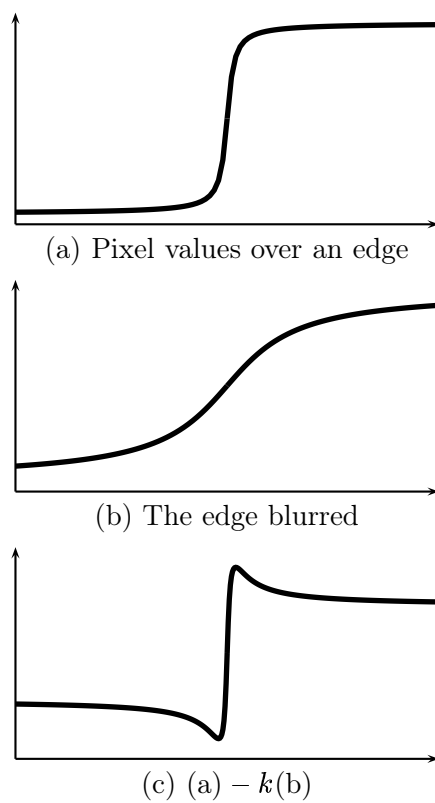


Figure 3.9: Unsharp masking

The unsharp option of `fspecial` produces such filters; the filter created has the form

$$\frac{1}{\alpha + 1} \begin{bmatrix} -\alpha & \alpha - 1 & -\alpha \\ \alpha - 1 & \alpha + 5 & \alpha - 1 \\ -\alpha & \alpha - 1 & -\alpha \end{bmatrix}$$

where α is an optional parameter which defaults to 0.2. If $\alpha = 0.5$ the filter is

$$\frac{1}{3} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 11 & -1 \\ -1 & -1 & -1 \end{bmatrix} = 4 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - 3 \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Figure 3.10 was created using the MATLAB commands

```
>> p=imread('pelicans.tif');
>> u=fspecial('unsharp',0.5);
>> pu=filter2(u,p);
>> imshow(p),figure,imshow(pu/255)
```

Figure 3.10(b), appears much sharper and “cleaner” than the original. Notice in particular the rocks and trees in the background, and the ripples on the water.



(a) The original



(b) After unsharp masking

Figure 3.10: Edge enhancement with unsharp masking

Although we have used averaging filters above, we can in fact use any low pass filter for unsharp masking.

High boost filtering

Allied to unsharp masking filters are the *high boost* filters, which are obtained by

$$\text{high boost} = A(\text{original}) - (\text{low pass}).$$

where A is an “amplification factor”. If $A = 1$, then the high boost filter becomes an ordinary high pass filter. If we take as the low pass filter the 3×3 averaging filter, then a high boost filter will have the form

$$\frac{1}{9} \begin{bmatrix} -1 & -1 & -1 \\ -1 & z & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

where $z > 8$. If we put $z = 11$, we obtain a filtering very similar to the unsharp filter above, except for a scaling factor. Thus the commands:

```
>> f=[-1 -1 -1;-1 11 -1;-1 -1 -1]/9;
>> xf=filter2(x,f);
>> imshow(xf/80)
```

will produce an image similar to that in figure 3.8. The value 80 was obtained by trial and error to produce an image with similar intensity to the original.

We can also write the high boost formula above as

$$\begin{aligned} \text{high boost} &= A(\text{original}) - (\text{low pass}) \\ &= A(\text{original}) - ((\text{original}) - (\text{high pass})) \\ &= (A - 1)(\text{original}) + (\text{high pass}). \end{aligned}$$

Best results for high boost filtering are obtained if we multiply the equation by a factor w so that the filter values sum to 1; this requires

$$wA - w = 1$$

or

$$w = \frac{1}{A - 1}.$$

So a general unsharp masking formula is

$$\frac{A}{A - 1}(\text{original}) - \frac{1}{A - 1}(\text{low pass}).$$

Another version of this formula is

$$\frac{A}{2A - 1}(\text{original}) - \frac{1 - A}{2A - 1}(\text{low pass})$$

where for best results A is taken so that

$$\frac{3}{5} \leq A \leq \frac{5}{6}.$$

If we take $A = 3/5$, the formula becomes

$$\frac{3/5}{2(3/5) - 1}(\text{original}) - \frac{1 - (3/5)}{2(3/5) - 1}(\text{low pass}) = 3(\text{original}) - 2(\text{low pass})$$

If we take $A = 5/6$ we obtain

$$\frac{5}{4}(\text{original}) - \frac{1}{4}(\text{low pass})$$

Using the identity and averaging filters, we can obtain high boost filters by:

```
>> id=[0 0 0;0 1 0;0 0 0];
>> f=fspecial('average');
>> hb1=3*id-2*f

hb1 =

   -0.2222   -0.2222   -0.2222
   -0.2222    2.7778   -0.2222
   -0.2222   -0.2222   -0.2222

>> hb2=1.25*id-0.25*f

hb2 =

   -0.0278   -0.0278   -0.0278
   -0.0278    1.2222   -0.0278
   -0.0278   -0.0278   -0.0278
```

If each of the filters `hb1` and `hb2` are applied to an image with `filter2`, the result will have enhanced edges. The images in figure 3.11 show these results; figure 3.11(a) was obtained with

```
>> x1=filter2(hb1,x);
>> imshow(x1/255)
```

and figure 3.11(b) similarly.



(a) High boost filtering with `hb1`



(b) High boost filtering with `hb2`

Figure 3.11: High boost filtering

Of the two filters, `hb1` appears to produce the best result; `hb2` produces an image not very much crisper than the original.

3.6 Non-linear filters

Linear filters, as we have seen in the previous sections, are easy to describe, and can be applied very quickly and efficiently by MATLAB.

A *non-linear filter* is obtained by a non-linear function of the greyscale values in the mask. Simple examples are the *maximum filter*, which has as its output the maximum value under the mask, and the corresponding *minimum filter*, which has as its output the minimum value under the mask.

Both the maximum and minimum filters are examples of *rank-order filters*. In such a filter, the elements under the mask are ordered, and a particular value returned as output. So if the values are given in increasing order, the minimum filter is a rank-order filter for which the *first* element is returned, and the maximum filter is a rank-order filter for which the *last* element is returned.

For implementing a general non-linear filter in MATLAB, the function to use is `nlfilter`, which applies a filter to an image according to a pre-defined function. If the function is not already defined, we have to create an m-file which defines it.

Here are some examples; first to implement a maximum filter over a 3×3 neighbourhood:

```
>> cmax=nlfilter(c,[3,3],'max(x(:))');
```

The `nlfilter` function requires three arguments: the image matrix, the size of the filter, and the function to be applied. The function must be a matrix function which returns a scalar value. The result of this operation is shown in figure 3.12(a).

A corresponding implementation of the minimum filter is:

```
>> cmin=nlfilter(c,[3,3],'min(x(:))');
```

and the result is shown in figure 3.12(b).



(a) Using a maximum filter



(b) Using a minimum filter

Figure 3.12: Using non-linear filters

Note that in each case the image has lost some sharpness, and has been brightened by the maximum filter, and darkened by the minimum filter. The `nlfilter` function is very slow; in

general there is little call for non-linear filters except for a few which are defined by their own commands. We shall investigate these in later chapters.

Non-linear filtering using `nlfilter` can be very slow. A faster alternative is to use the `colfilt` function, which rearranges the image into columns first. For example, to apply the maximum filter to the cameraman image, we can use

```
>> cmax=colfilt(c,[3,3],'sliding',@max);
```

The parameter `sliding` indicates that overlapping neighbourhoods are being used (which of course is the case with filtering). This particular operation is almost instantaneous, as compared with the use of `nlfilter`.

To implement the maximum and minimum filters as rank-order filters, we may use the MATLAB function `ordfilt2`. This requires three inputs: the image, the index value of the ordered results to choose as output, and the definition of the mask. So to apply the maximum filter on a 3×3 mask, we use

```
>> cmax=ordfilt2(c,9,ones(3,3));
```

and the minimum filter can be applied with

```
>> cmin=ordfilt2(c,1,ones(3,3));
```

A very important rank-order filter is the *median filter*, which takes the *central* value of the ordered list. We could apply the median filter with

```
>> cmed=ordfilt2(c,5,ones(3,3));
```

However, the median filter has its own command, `medfilt2`, which we discuss in more detail in chapter 5.

Other non-linear filters are the *geometric mean filter*, which is defined as

$$\left(\prod_{(i,j) \in M} x(i,j) \right)^{(1/|M|)}$$

where M is the filter mask, and $|M|$ its size; and the *alpha-trimmed mean filter*, which first orders the values under the mask, trims off elements at either end of the ordered list, and takes the mean of the remainder. So, for example, if we have a 3×3 mask, and we order the elements as

$$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_9$$

and trim off two elements at either end, the result of the filter will be

$$(x_3 + x_4 + x_5 + x_6 + x_7)/5.$$

Both of these filters have uses for image restoration; again see chapters 5 and 6.

Exercises

1. The array below represents a small greyscale image. Compute the images that result when the image is convolved with each of the masks (a) to (h) shown. At the edge of the image use a restricted mask. (In other words, pad the image with zeroes.)

Chapter 4

The Fourier Transform

4.1 Introduction

The Fourier Transform is of fundamental importance to image processing. It allows us to perform tasks which would be impossible to perform any other way; its efficiency allows us to perform other tasks more quickly. The Fourier Transform provides, among other things, a powerful alternative to linear spatial filtering; it is more efficient to use the Fourier transform than a spatial filter for a large filter. The Fourier Transform also allows us to isolate and process particular image “frequencies”, and so perform low-pass and high-pass filtering with a great degree of precision.

Before we discuss the Fourier transform of images, we shall investigate the one-dimensional Fourier transform, and a few of its properties.

4.2 Background

Our starting place is the observation that a periodic function may be written as the sum of sines and cosines of varying amplitudes and frequencies. For example, in figure 4.1 we plot a function, and its decomposition into sine functions.

Some functions will require only a finite number of functions in their decomposition; others will require an infinite number. For example, a “square wave”, such as is shown in figure 4.2, has the decomposition

$$f(x) = \sin x + \frac{1}{3} \sin 3x + \frac{1}{5} \sin 5x + \frac{1}{7} \sin 7x + \frac{1}{9} \sin 9x + \dots \quad (4.1)$$

In figure 4.2 we take the first four terms only to provide the approximation. The more terms of the series we take, the closer the sum will approach the original function.

4.3 The one-dimensional discrete Fourier transform

When we deal with a *discrete* function, as we shall do for images, the situation from the previous section changes slightly. Since we only have to obtain a finite number of values, we only need a finite number of functions to do it.

Consider for example the discrete sequence

$$1, \quad 1, \quad 1, \quad 1, \quad -1, \quad -1, \quad -1, \quad -1$$

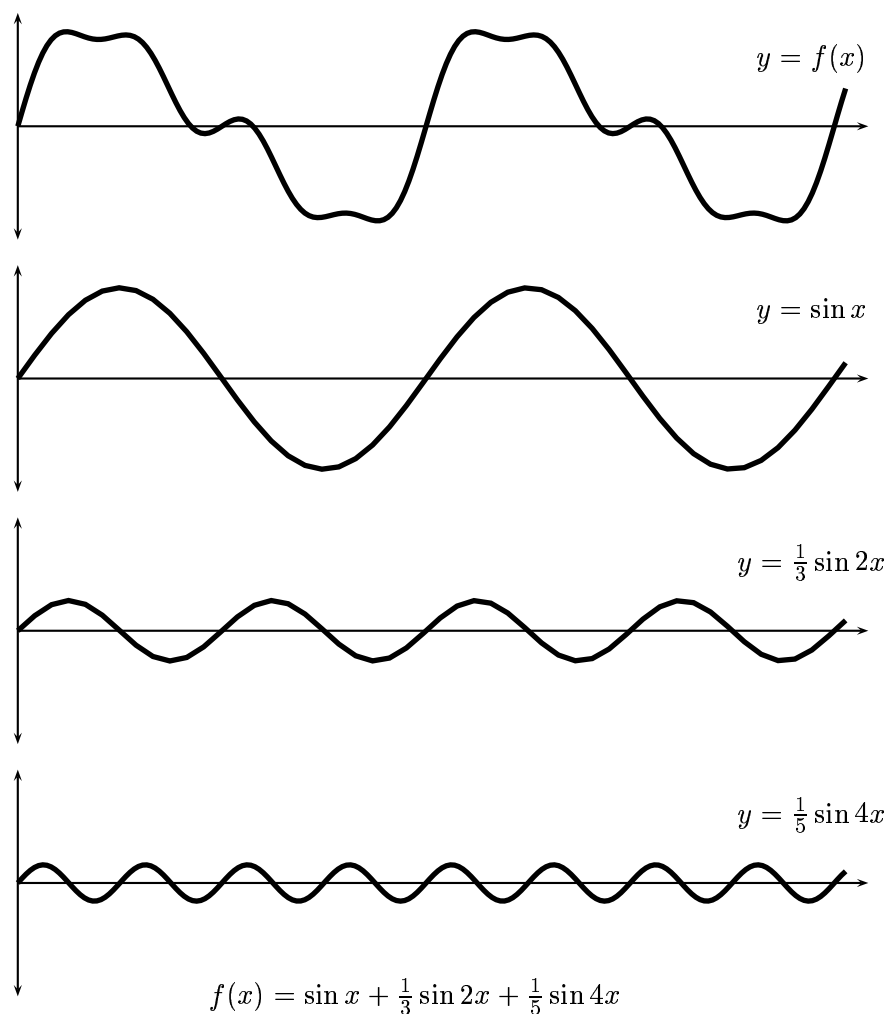


Figure 4.1: A function and its trigonometric decomposition

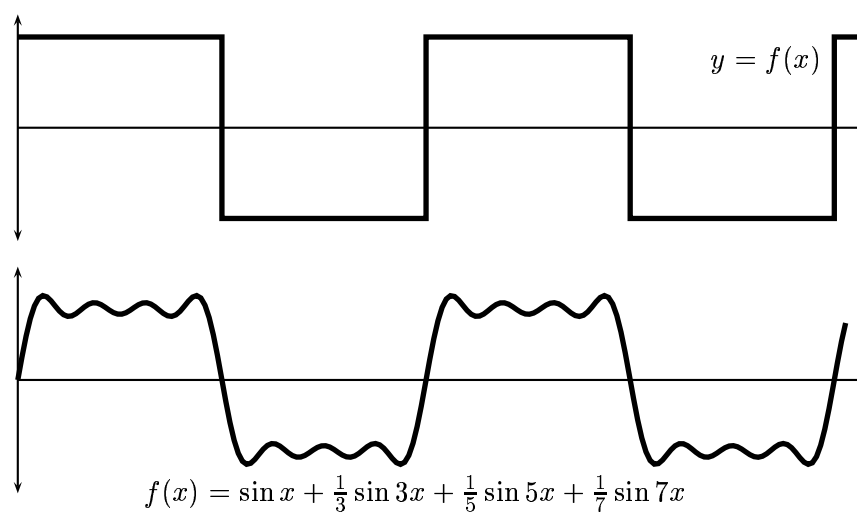


Figure 4.2: A square wave and its trigonometric approximation

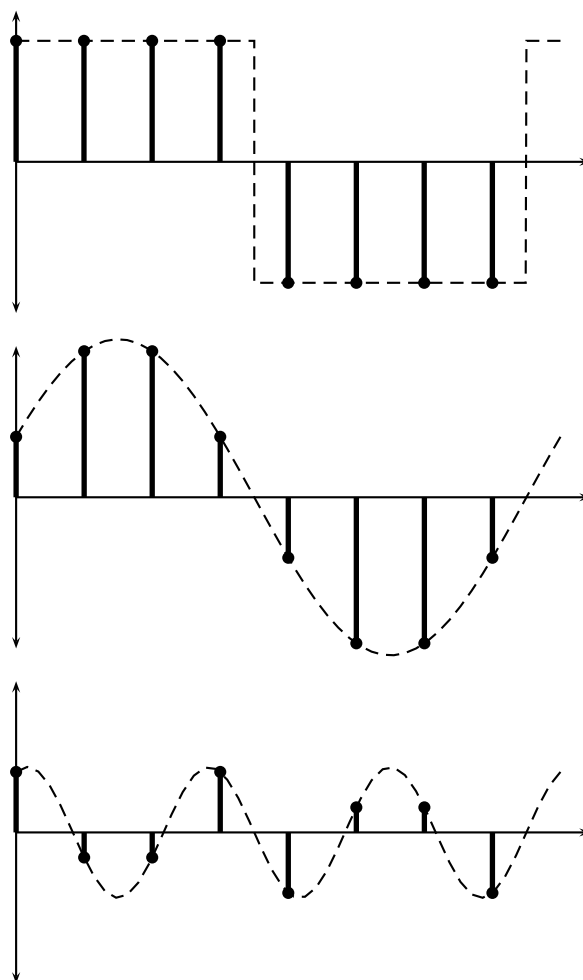


Figure 4.3: Expressing a discrete function as the sum of sines

which we may take as a discrete approximation to the square wave of figure 4.2. This can be expressed as the sum of only *two* sine functions; this is shown in figure 4.3.

The Fourier transform allows us to obtain those individual sine waves which compose a given function or sequence. Since we shall be concerned with discrete sequences, and of course images, we shall investigate only the *discrete Fourier transform*, abbreviated DFT.

4.3.1 Definition of the one dimensional DFT

Suppose

$$\mathbf{f} = [f_0, f_1, f_2, \dots, f_{N-1}]$$

is a sequence of length N . We define its *discrete Fourier transform* to be the sequence

$$\mathbf{F} = [F_0, F_1, F_2, \dots, F_{N-1}]$$

where

$$F_u = \frac{1}{N} \sum_{x=0}^{N-1} \exp \left[-2\pi i \frac{xu}{N} \right] f_x. \quad (4.2)$$

The formula for the inverse DFT is very similar to the forward transform:

$$x_u = \sum_{x=0}^{N-1} \exp \left[2\pi i \frac{xu}{N} \right] F_u. \quad (4.3)$$

If you compare this equation with equation 4.2 you will see that there are really only two differences:

1. there is no scaling factor $1/N$,
2. the sign inside the exponential function has been changed to positive.

The Fast Fourier Transform. One of the many aspects which make the DFT so attractive for image processing is the existence of very fast algorithms to compute it. There are a number of extremely fast and efficient algorithms for computing a DFT; such an algorithm is called a *fast Fourier transform*, or FFT. The use of an FFT vastly reduces the time needed to compute a DFT.

One FFT method works recursively by dividing the original vector into two halves, computing the FFT of each half, and then putting the results together. This means that the FFT is most efficient when the vector length is a power of 2.

Table 4.1 shows that advantage gained by using the FFT algorithm as opposed to the direct arithmetic definition of equations 4.6 and 4.7 by comparing the number of multiplications required for each method. For a vector of length 2^n , the direct method takes $(2^n)^2 = 2^{2n}$ multiplications; the FFT only $n2^n$. The saving in time is thus of an order of $2^n/n$. Clearly the advantage of using an FFT algorithm becomes greater as the size of the vector increases.

Because of this computational advantage, any implementation of the DFT will use an FFT algorithm.

2^n	Direct arithmetic	FFT	Increase in speed
4	16	8	2.0
8	84	24	2.67
16	256	64	4.0
32	1024	160	6.4
64	4096	384	10.67
128	16384	896	18.3
256	65536	2048	32.0
512	262144	4608	56.9
1024	1048576	10240	102.4

Table 4.1: Comparison of FFT and direct arithmetic

4.4 The two-dimensional DFT

In two dimensions, the DFT takes a matrix as input, and returns another matrix, of the same size, as output. If the original matrix values are $f(x, y)$, where x and y are the indices, then the output matrix values are $F(u, v)$. We call the matrix F the *Fourier transform of f* and write

$$F = \mathcal{F}(f).$$

Then the original matrix f is the *inverse Fourier transform of F* , and we write

$$f = \mathcal{F}^{-1}(F).$$

We have seen that a (one-dimensional) function can be written as a sum of sines and cosines. Given that an image may be considered as a two-dimensional function $f(x, y)$, it seems reasonable to assume that f can be expressed as sums of “corrugation” functions which have the general form

$$z = a \sin(bx + cy).$$

A sample such function is shown in figure 4.4. And this is in fact exactly what the two-dimensional

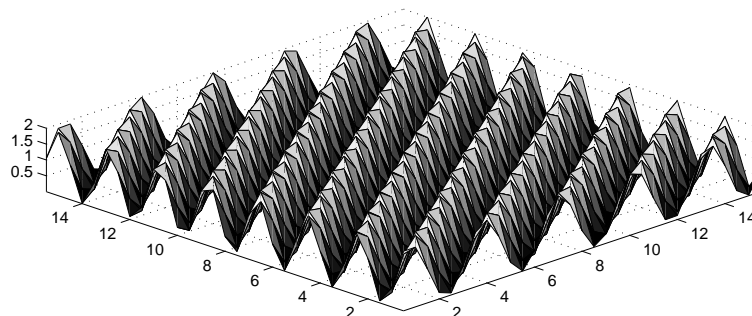


Figure 4.4: A “corrugation” function

Fourier transform does: it rewrites the original matrix in terms of sums of corrugations.

The definition of the two-dimensional discrete Fourier transform is very similar to that for one dimension. The forward and inverse transforms for an $M \times N$ matrix, where for notational

convenience we assume that the x indices are from 0 to $M - 1$ and the y indices are from 0 to $N - 1$, are:

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \exp \left[-2\pi i \left(\frac{xu}{M} + \frac{yv}{N} \right) \right]. \quad (4.4)$$

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) \exp \left[2\pi i \left(\frac{xu}{M} + \frac{yv}{N} \right) \right]. \quad (4.5)$$

These are horrendous looking formulas, but if we spend a bit of time pulling them apart, we shall see that they aren't as bad as they look.

Before we do this, we note that the formulas given in equations 4.4 and 4.5 are not used by all authors. The main change is the position of the scaling factor $1/MN$. Some people put it in front of the sums in the forward formula. Others put a factor of $1/\sqrt{MN}$ in front of both sums. The point is the sums by themselves would produce a result (after both forward and inverse transforms) which is too large by a factor of MN . So somewhere in the forward-inverse formulas a corresponding $1/MN$ must exist; it doesn't really matter where.

4.4.1 Some properties of the two dimensional Fourier transform

All the properties of the one-dimensional DFT transfer into two dimensions. But there are some further properties not previously mentioned, which are of particular use for image processing.

Similarity. First notice that the forward and inverse transforms are very similar, with the exception of the scale factor $1/MN$ in the inverse transform, and the negative sign in the exponent of the forward transform. This means that the same algorithm, only very slightly adjusted, can be used for both the forward and inverse transforms.

The DFT as a spatial filter. Note that the values

$$\exp \left[\pm 2\pi i \left(\frac{xu}{M} + \frac{yv}{N} \right) \right]$$

are independent of the values f or F . This means that they can be calculated in advance, and only then put into the formulas above. It also means that every value $F(u, v)$ is obtained by multiplying every value of $f(x, y)$ by a fixed value, and adding up all the results. But this is precisely what a linear spatial filter does: it multiplies all elements under a mask with fixed values, and adds them all up. Thus we can consider the DFT as a linear spatial filter which is as big as the image. To deal with the problem of edges, we assume that the image is tiled in all directions, so that the mask always has image values to use.

Separability. Notice that the Fourier transform "filter elements" can be expressed as products:

$$\exp \left[2\pi i \left(\frac{xu}{M} + \frac{yv}{N} \right) \right] = \exp \left[2\pi i \frac{xu}{M} \right] \exp \left[2\pi i \frac{yv}{N} \right].$$

The first product value

$$\exp \left[2\pi i \frac{xu}{M} \right]$$

depends only on x and u , and is independent of y and v . Conversely, the second product value

$$\exp \left[2\pi i \frac{yv}{N} \right]$$

depends only on y and v , and is independent of x and u . This means that we can break down our formulas above to simpler formulas that work on single rows or columns:

$$F(u) = \sum_{x=0}^{M-1} f(x) \exp \left[-2\pi i \frac{xu}{M} \right], \quad (4.6)$$

$$f(x) = \frac{1}{M} \sum_{u=0}^{M-1} F(u) \exp \left[2\pi i \frac{xu}{M} \right]. \quad (4.7)$$

If we replace x and u with y and v we obtain the corresponding formulas for the DFT of matrix columns. These formulas define the *one-dimensional DFT* of a vector, or simply the DFT.

The 2-D DFT can be calculated by using this property of “separability”; to obtain the 2-D DFT of a matrix, we first calculate the DFT of all the rows, and then calculate the DFT of all the columns of the result, as shown in figure 4.5. Since a product is independent of the order, we can equally well calculate a 2-D DFT by calculating the DFT of all the columns first, then calculating the DFT of all the rows of the result.

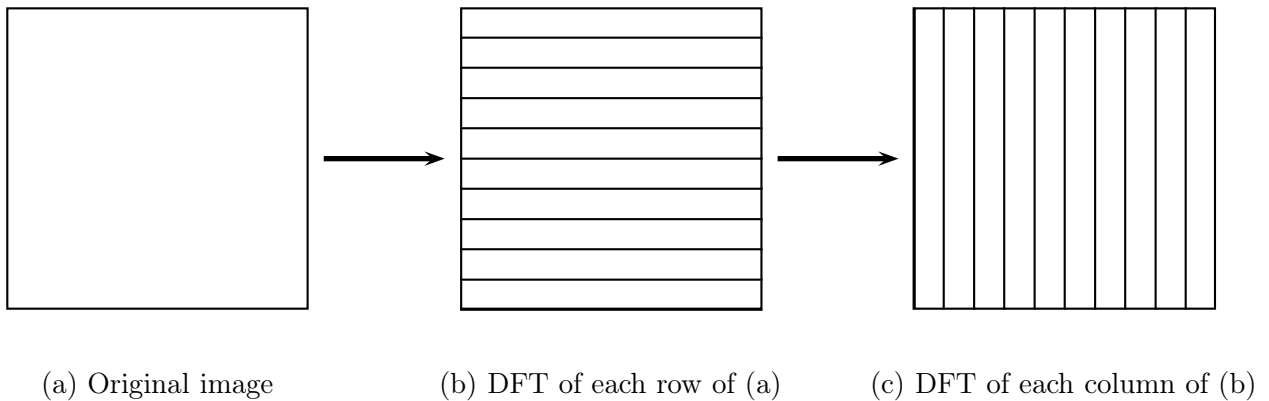


Figure 4.5: Calculating a 2D DFT

Linearity An important property of the DFT is its linearity; the DFT of a sum is equal to the sum of the individual DFT’s, and the same goes for scalar multiplication:

$$\begin{aligned} \mathcal{F}(f + g) &= \mathcal{F}(f) + \mathcal{F}(g) \\ \mathcal{F}(kf) &= k\mathcal{F}(f) \end{aligned}$$

where k is a scalar, and f and g are matrices. This follows directly from the definition given in equation 4.4.

This property is of great use in dealing with image degradation such as noise which can be modelled as a sum:

$$d = f + n$$

where f is the original image; n is the noise, and d is the degraded image. Since

$$\mathcal{F}(d) = \mathcal{F}(f) + \mathcal{F}(n)$$

we may be able to remove or reduce n by modifying the transform. As we shall see, some noise appears on the DFT in a way which makes it particularly easy to remove.

The convolution theorem. This result provides one of the most powerful advantages of using the DFT. Suppose we wish to convolve an image M with a spatial filter S . Our method has been place S over each pixel of M in turn, calculate the product of all corresponding grey values of M and elements of S , and add the results. The result is called the *digital convolution* of M and S , and is denoted

$$M * S.$$

This method of convolution can be very slow, especially if S is large. The *convolution theorem* states that the result $M * S$ can be obtained by the following sequence of steps:

1. Pad S with zeroes so that is the same size as M ; denote this padded result by S' .
2. Form the DFT's of both M and S , to obtain $\mathcal{F}(M)$ and $\mathcal{F}(S')$.
3. Form the element-by-element product of these two transforms:

$$\mathcal{F}(M) \cdot \mathcal{F}(S').$$

4. Take the inverse transform of the result:

$$\mathcal{F}^{-1}(\mathcal{F}(M) \cdot \mathcal{F}(S')).$$

Put simply, the convolution theorem states:

$$M * S = \mathcal{F}^{-1}(\mathcal{F}(M) \cdot \mathcal{F}(S'))$$

or equivalently that

$$\mathcal{F}(M * S) = \mathcal{F}(M) \cdot \mathcal{F}(S').$$

Although this might seem like an unnecessarily clumsy and roundabout way of computing something so simple as a convolution, it can have enormous speed advantages if S is large.

For example, suppose we wish to convolve a 512×512 image with a 32×32 filter. To do this directly would require $32^2 = 1024$ multiplications for each pixel, of which there are $512 \times 512 = 262144$. Thus there will be a total of $1024 \times 262144 = 268,435,456$ multiplications needed. Now look at applying the DFT (using an FFT algorithm). Each row requires 4608 multiplications by table 4.1; there are 512 rows, so a total of $4608 \times 512 = 2359296$ multiplications; the same must be done again for the columns. Thus to obtain the DFT of the image requires 4718592 multiplications. We need the same amount to obtain the DFT of the filter, and for the inverse DFT. We also require 512×512 multiplications to perform the product of the two transforms.

Thus the total number of multiplications needed to perform convolution using the DFT is

$$4718592 \times 3 + 262144 = 14,417,920$$

which is an enormous saving compared to the direct method.

The DC coefficient. The value $F(0,0)$ of the DFT is called the *DC coefficient*. If we put $u = v = 0$ in the definition given in equation 4.4 then

$$F(0,0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp(0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y).$$

That is, this term is equal to the sum of *all* terms in the original matrix.

Shifting. For purposes of display, it is convenient to have the DC coefficient in the centre of the matrix. This will happen if all elements $f(x,y)$ in the matrix are multiplied by $(-1)^{x+y}$ before the transform. Figure 4.6 demonstrates how the matrix is shifted by this method. In each diagram the DC coefficient is the top left hand element of submatrix *A*, and is shown as a black square.

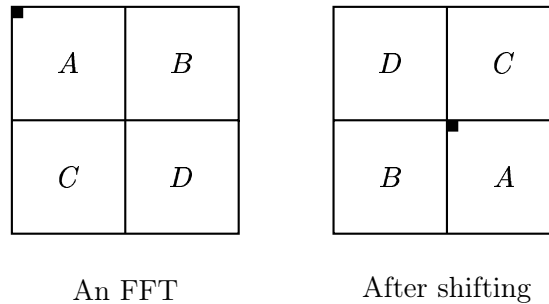


Figure 4.6: Shifting a DFT

Conjugate symmetry An analysis of the Fourier transform definition leads to a symmetry property; if we make the substitutions $u = -u$ and $v = -v$ in equation 4.4 then

$$\mathcal{F}(u, v) = \mathcal{F}^*(-u + pM, -v + qN)$$

for any integers p and q . This means that half of the transform is a mirror image of the conjugate of the other half. We can think of the top and bottom halves, or the left and right halves, being mirror images of the conjugates of each other.

Figure 4.7 demonstrates this symmetry in a shifted DFT. As with figure 4.6, the black square shows the position of the DC coefficient. The symmetry means that its information is given in just half of a transform, and the other half is redundant.

Displaying transforms. Having obtained the Fourier transform $F(u,v)$ of an image $f(x,y)$, we would like to see what it looks like. As the elements $F(u,v)$ are complex numbers, we can't view them directly, but we can view their magnitude $|F(u,v)|$. Since these will be numbers of type `double`, generally with large range, we have two approaches

1. find the maximum value m of $|F(u,v)|$ (this will be the DC coefficient), and use `imshow` to view $|F(u,v)|/m$,
2. use `mat2gray` to view $|F(u,v)|$ directly.

	a		a^*
b^*	B^*	d^*	A^*
	c		c^*
b	A	d	B

Figure 4.7: Conjugate symmetry in the DFT

One trouble is that the DC coefficient is generally very much larger than all other values. This has the effect of showing a transform as a single white dot surrounded by black. One way of stretching out the values is to take the logarithm of $|F(u, v)|$ and to display

$$\log(1 + |F(u, v)|).$$

The display of the magnitude of a Fourier transform is called the *spectrum* of the transform. We shall see some examples later on.

4.5 Fourier transforms in MATLAB

The relevant MATLAB functions for us are:

- `fft` which takes the DFT of a vector,
- `ifft` which takes the inverse DFT of a vector,
- `fft2` which takes the DFT of a matrix,
- `ifft2` which takes the inverse DFT of a matrix,
- `fftshift` which shifts a transform as shown in figure 4.6.

of which we have seen the first two above.

Before attacking a few images, let's take the Fourier transform of a few small matrices to get more of an idea what the DFT "does".

Example 1. Suppose we take a constant matrix $f(x, y) = 1$. Going back to the idea of a sum of corrugations, then *no* corrugations are required to form a constant. Thus we would hope that the DFT consists of a DC coefficient and zeroes everywhere else. We will use the `ones` function, which produces an $n \times n$ matrix consisting of 1's, where n is an input to the function.

```
>> a=ones(8);
>> fft2(a)
```

The result is indeed as we expected:

```
ans =
    64     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

Note that the DC coefficient is indeed the sum of all the matrix values.

Example 2. Now we'll take a matrix consisting of a single corrugation:

```
>> a = [100 200; 100 200];
>> a = repmat(a,4,4)

ans =
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200
    100    200    100    200    100    200    100    200

>> af = fft2(a)

ans =

   9600     0     0     0  -3200     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

What we have here is really the sum of two matrices: a constant matrix each element of which is 150, and a corrugation which alternates -50 and 50 from left to right. The constant matrix alone would produce (as in example 1), a DC coefficient alone of value $64 \times 150 = 9600$; the corrugation a single value. By linearity, the DFT will consist of just the two values.

Example 3. We will take here a single step edge:

```
>> a = [zeros(8,4) ones(8,4)]
a =
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
    0    0    0    0    1    1    1    1
```

Now we shall perform the Fourier transform with a shift, to place the DC coefficient in the centre, and since it contains some complex values, for simplicity we shall just show the rounded absolute values:

```
>> af=fftshift(fft2(a));
>> round(abs(af))

ans =
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    9    0   21   32   21    0    9
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0
```

The DC coefficient is of course the sum of all values of **a**; the other values may be considered to be the coefficients of the necessary sine functions required to form an edge, as given in equation 4.1. The mirroring of values about the DC coefficient is a consequence of the symmetry of the DFT.

4.6 Fourier transforms of images

We shall create a few simple images, and see what the Fourier transform produces.

Example 1. We shall produce a simple image consisting of a single edge:

```
>> a=[zeros(256,128) ones(256,128)];
```

This is displayed on the left in figure 4.9. Now we shall take its DFT, and shift it:

```
>> af=fftshift(fft2(a));
```

Now we'll view its spectrum; we have the choice of two commands:

```
1. af1=log(1+abs(af));
   imshow(af1/af1(129,129))
```

This works because after shifting, the DC coefficient is at position $x = 129$, $y = 129$. We stretch the transform using `log`, and divide the result by the middle value to obtain matrix of type `double` with values in the range 0.0–1.0. This can then be viewed directly with `imshow`.

```
2. imshow(mat2gray(log(1+abs(af))))
```

The `mat2gray` function automatically scales a matrix for display as an image, as we have seen in chapter 3

It is in fact convenient to write a small function for viewing transforms. One such is shown in figure 4.8. Then for example

```
function fftshow(f,type)

% Usage:  FFTSHOW(F,TYPE)
%
% Displays the fft matrix F using imshow, where TYPE must be one of
% 'abs' or 'log'.  If TYPE='abs', then then abs(f) is displayed; if
% TYPE='log' then log(1+abs(f)) is displayed.  If TYPE is omitted, then
% 'log' is chosen as a default.
%
% Example:
%   c=imread('cameraman.tif');
%   cf=fftshift(fft2(c));
%   fftshow(cf,'abs')
%

if nargin<2,
    type='log';
end

if (type=='log')
    fl = log(1+abs(f));
    fm = max(fl(:));
    imshow(im2uint8(fl/fm))
elseif (type=='abs')
    fa=abs(f);
    fm=max(fa(:));
    imshow(fa/fm)
else
    error('TYPE must be abs or log.');
```

Figure 4.8: A function to display a Fourier transform

```
>> fftshow(af,'log')
```

will show the logarithm of the absolute values of the transform, and

```
>> fftshow(af,'abs')
```

will show the absolute values of the transform without any scaling.

The result is shown on the right in figure 4.9. We observe immediately that the result is similar

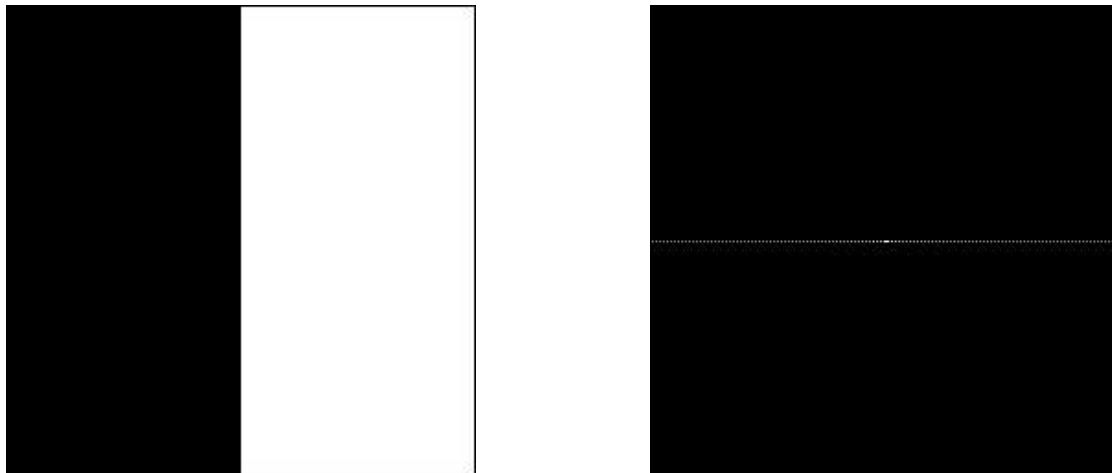


Figure 4.9: A single edge and its DFT

(although larger) to example 3 in the previous section.

Example 2. Now we'll create a box, and then its Fourier transform:

```
>> a=zeros(256,256);
>> a(78:178,78:178)=1;
>> imshow(a)
>> af=fftshift(fft2(a));
>> figure,fftshow(af,'abs')
```

The box is shown on the left in figure 4.10, and its Fourier transform is shown on the right.

Example 3. Now we shall look at a box rotated 45° .

```
>> [x,y]=meshgrid(1:256,1:256);
>> b=(x+y<329)&(x+y>182)&(x-y>-67)&(x-y<73);
>> imshow(b)
>> bf=fftshift(fft2(b));
>> figure,fftshow(bf)
```

The results are shown in figure 4.11. Note that the transform of the rotated box is the rotated transform of the original box.

Example 4. We will create a small circle, and then transform it:

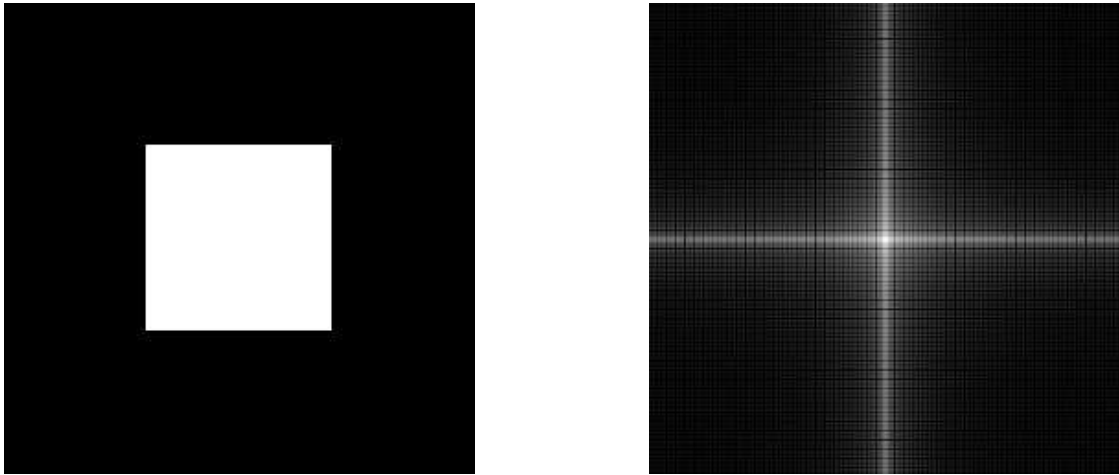


Figure 4.10: A box and its DFT

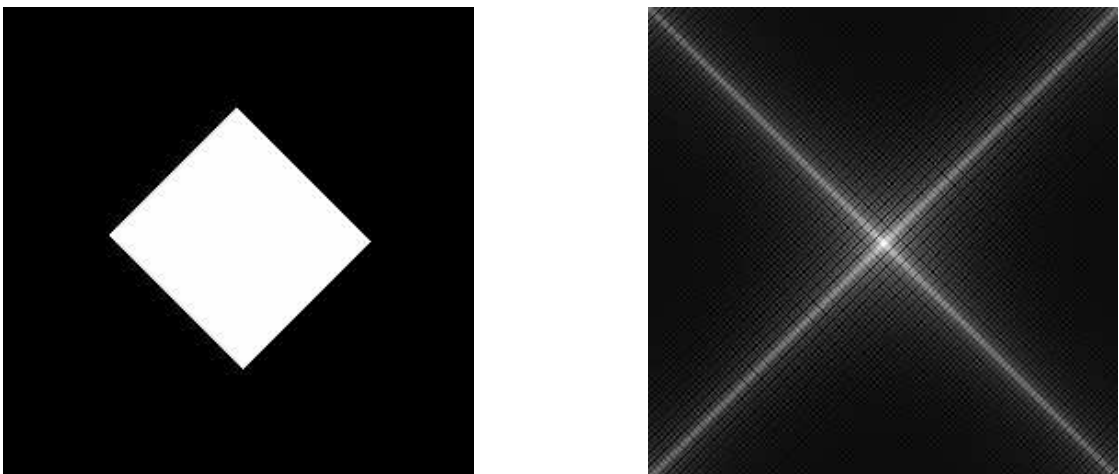


Figure 4.11: A rotated box and its DFT

```
>> [x,y]=meshgrid(-128:217,-128:127);
>> z=sqrt(x.^2+y.^2);
>> c=(z<15);
```

The result is shown on the left in figure 4.12. Now we will create its Fourier transform and display it:

```
>> cf=fft2shift(fft2(z));
>> fftshow(cf,'log')
```

and this is shown on the right in figure 4.12. Note the “ringing” in the Fourier transform. This is an

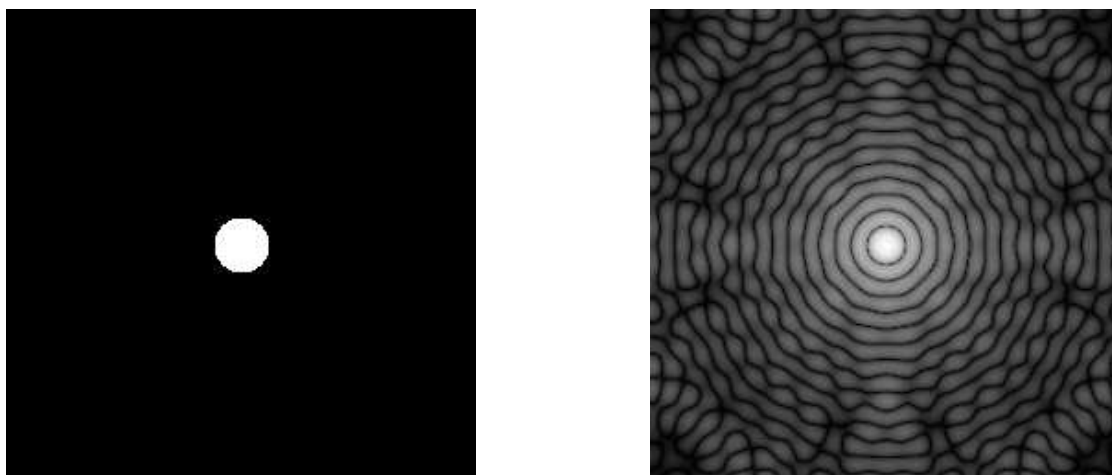


Figure 4.12: A circle and its DFT

artifact associated with the sharp cutoff of the circle. As we have seen from both the edge and box images in the previous examples, an edge appears in the transform as a line of values at right angles to the edge. We may consider the values on the line as being the coefficients of the appropriate corrugation functions which sum to the edge. With the circle, we have lines of values radiating out from the circle; these values appear as circles in the transform.

A circle with a gentle cutoff, so that its edge appears blurred, will have a transform with no ringing. Such a circle can be made with the command (given z above):

```
b=1./(1+(z./15).^2);
```

This image appears as a blurred circle, and its transform is very similar—check them out!

4.7 Filtering in the frequency domain

We have seen in section 4.4 that one of the reasons for the use of the Fourier transform in image processing is due to the convolution theorem: a spatial convolution can be performed by element-wise multiplication of the Fourier transform by a suitable “filter matrix”. In this section we shall explore some filtering by this method.

4.7.1 Ideal filtering

Low pass filtering

Suppose we have a Fourier transform matrix F , shifted so that the DC coefficient is in the centre. Since the low frequency components are towards the centre, we can perform low pass filtering by multiplying the transform by a matrix in such a way that centre values are maintained, and values away from the centre are either removed or minimized. One way to do this is to multiply by an *ideal low-pass matrix*, which is a binary matrix m defined by:

$$m(x, y) = \begin{cases} 1 & \text{if } (x, y) \text{ is closer to the center than some value } D, \\ 0 & \text{if } (x, y) \text{ is further from the center than } D. \end{cases}$$

The circle c displayed in figure 4.12 is just such a matrix, with $D = 15$. Then the inverse Fourier transform of the element-wise product of F and m is the result we require:

$$\mathcal{F}^{-1}(F \cdot m).$$

Let's see what happens if we apply this filter to an image. First we obtain an image and its DFT.

```
>> cm=imread('cameraman.tif');
>> cf=fftshift(fft2(cm));
>> figure,fftshow(cf,'log')
```

The cameraman image and its DFT are shown in figure 4.13. Now we can perform a low pass filter



Figure 4.13: The “cameraman” image and its DFT

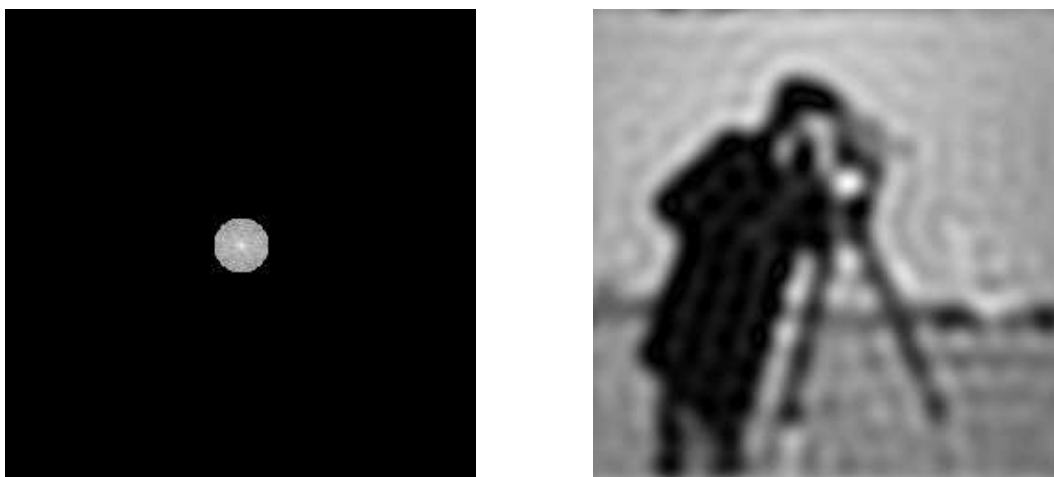
by multiplying the transform matrix by the circle matrix (recall that “dot asterisk” is the MATLAB syntax for element-wise multiplication of two matrices):

```
>> cfl=cf.*c;
>> figure,fftshow(cfl,'log')
```

and this is shown in figure 4.14(a). Now we can take the inverse transform and display the result:


```
>> cfli=ifft2(cfl);
>> figure,fftshow(cfli,'abs')
```

and this is shown in figure 4.14(b). Note that even though `cfli` is supposedly a matrix of real numbers, we are still using `fftshow` to display it. This is because the `fft2` and `ifft2` functions, being numeric, will not produce mathematically perfect results, but rather very close numeric approximations. So using `fftshow` with the `'abs'` option rounds out any errors obtained during the transform and its inverse. Note the “ringing” about the edges in this image. This is a direct result



(a) Ideal filtering on the DFT

(b) After inversion

Figure 4.14: Applying ideal low pass filtering

of the sharp cutoff of the circle. The ringing as shown in figure 4.12 is transferred to the image.

We would expect that the smaller the circle, the more blurred the image, and the larger the circle; the less blurred. Figure 4.15 demonstrates this, using cutoffs of 5 and 30. Notice that ringing is still present, and clearly visible in figure 4.15(b).

High pass filtering

Just as we can perform low pass filtering by keeping the centre values of the DFT and eliminating the others, so high pass filtering can be performed by the opposite: eliminating centre values and keeping the others. This can be done with a minor modification of the preceding method of low pass filtering. First we create the circle:

```
>> [x,y]=meshgrid(-128:127,-128:127);
>> z=sqrt(x.^2+y.^2);
>> c=(z>15);
```

and then multiply it by the DFT of the image:

```
>> cfh=cf.*c;
>> figure,fftshow(cfh,'log')
```

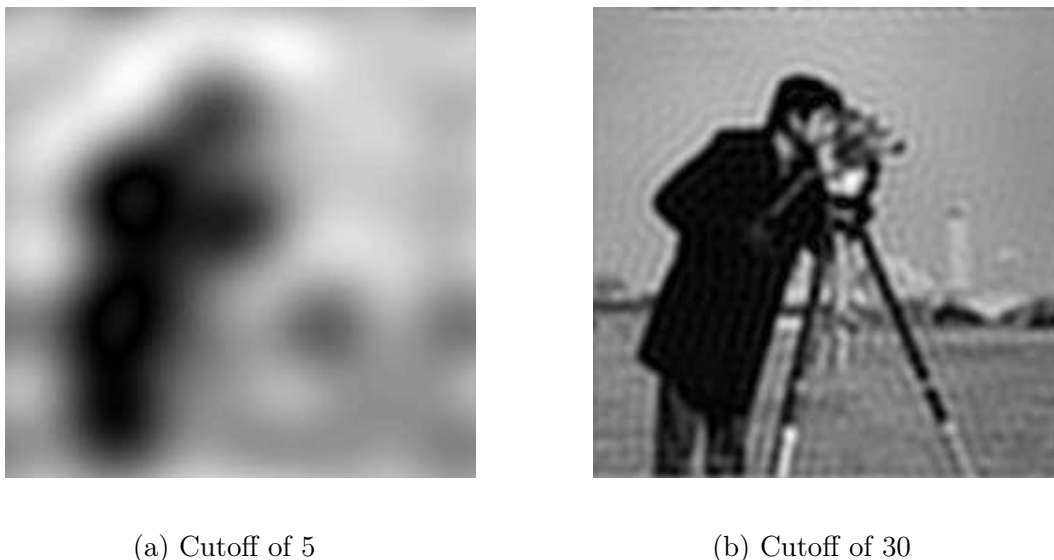


Figure 4.15: Ideal low pass filtering with different cutoffs

This is shown in figure 4.16(a). The inverse DFT can be easily produced and displayed:

```
>> cfhi=ifft2(cf);
>> figure,fftshow(cfhi,'abs')
```

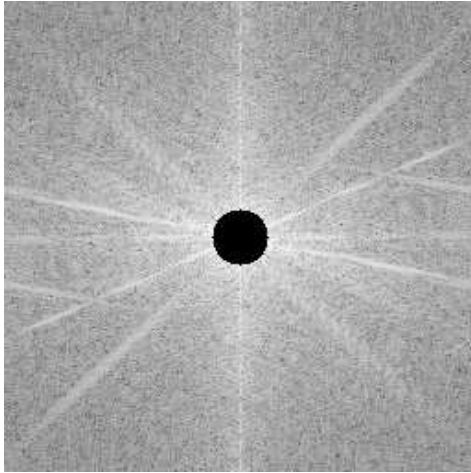
and this is shown in figure 4.16(b). As with low pass filtering, the size of the circle influences the information available to the inverse DFT, and hence the final result. Figure 4.17 shows some results of ideal high pass filtering with different cutoffs. If the cutoff is large, then more information is removed from the transform, leaving only the highest frequencies. This can be observed in figure 4.17(c) and (d); only the edges of the image remain. If we have small cutoff, such as in figure 4.17(a), we are only removing a small amount of the transform. We would thus expect that only the lowest frequencies of the image would be removed. And this is indeed true, as seen in figure 4.17(b); there is some greyscale detail in the final image, but large areas of low frequency are close to zero.

4.7.2 Butterworth filtering

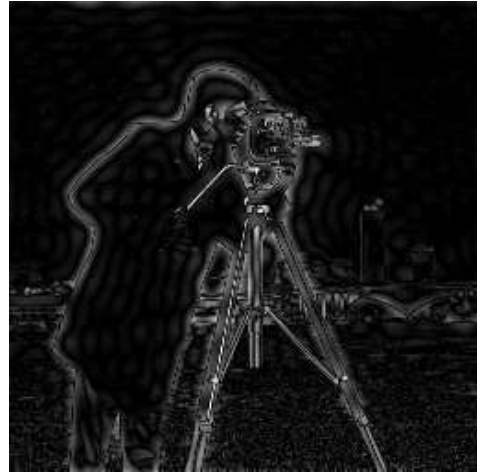
Ideal filtering simply cuts off the Fourier transform at some distance from the centre. This is very easy to implement, as we have seen, but has the disadvantage of introducing unwanted artifacts: ringing, into the result. One way of avoiding this is to use as a filter matrix a circle with a less sharp cutoff. A popular choice is to use *Butterworth filters*.

Before we describe these filters, we shall look again at the ideal filters. As these are radially symmetric about the centre of the transform, they can be simply described in terms of their cross sections. That is, we can describe the filter as a function of the distance x from the centre. For an ideal low pass filter, this function can be expressed as

$$f(x) = \begin{cases} 1 & \text{if } x < D, \\ 0 & \text{if } x \geq D \end{cases}$$



(a) The DFT after high pass filtering



(b) The resulting image

Figure 4.16: Applying an ideal high pass filter to an image

where D is the cutoff radius. Then the ideal high pass filters can be described similarly:

$$f(x) = \begin{cases} 1 & \text{if } x > D, \\ 0 & \text{if } x \leq D \end{cases}$$

These functions are illustrated in figure 4.18. Butterworth filter functions are based on the following functions for low pass filters:

$$f(x) = \frac{1}{1 + (x/D)^{2n}}$$

and for high pass filters:

$$f(x) = \frac{1}{1 + (D/x)^{2n}}$$

where in each case the parameter n is called the *order* of the filter. The size of n dictates the sharpness of the cutoff. These functions are illustrated in figures 4.19 and 4.20.

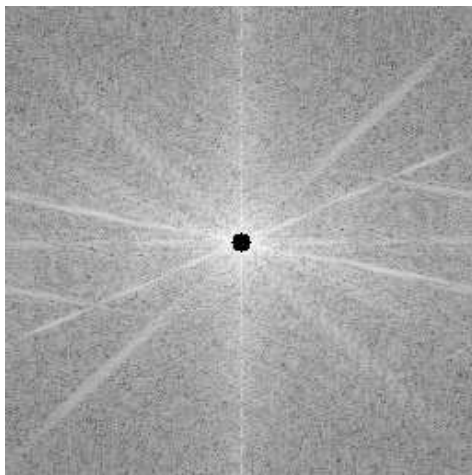
It is easy to implement these in MATLAB; here are the commands to produce a Butterworth low pass filter of size 256×256 with $D = 15$ and order $n = 2$:

```
>> [x,y]=meshgrid(-128:217,-128:127);
>> bl=1./(1+((x.^2+y.^2)/15).^2);
```

Since a Butterworth high pass filter can be obtained by subtracting a low pass filter from 1, we can write general MATLAB functions to generate Butterworth filters of general sizes. These are shown in figures 4.21 and 4.22.

So to apply a Butterworth low pass filter to the DFT of the cameraman image:

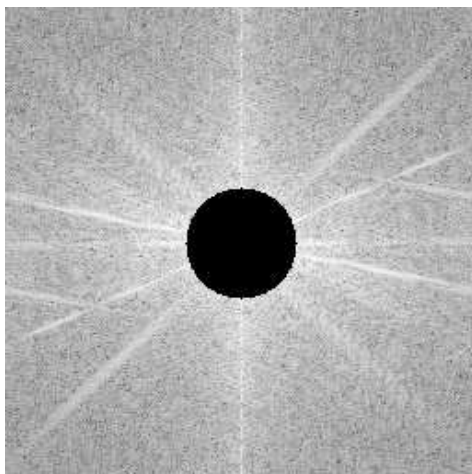
```
>> bl=lbutter(c,15,1);
>> cfbl=cf.*bl;
>> figure,fftshow(cfbl,'log')
```



(a) Cutoff of 5



(b) The resulting image



(a) Cutoff of 30



(b) The resulting image

Figure 4.17: Ideal high pass filtering with different cutoffs

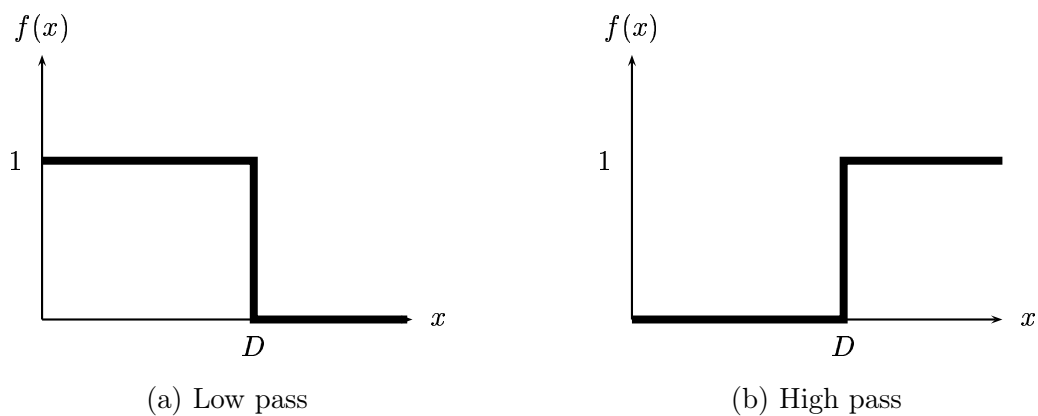
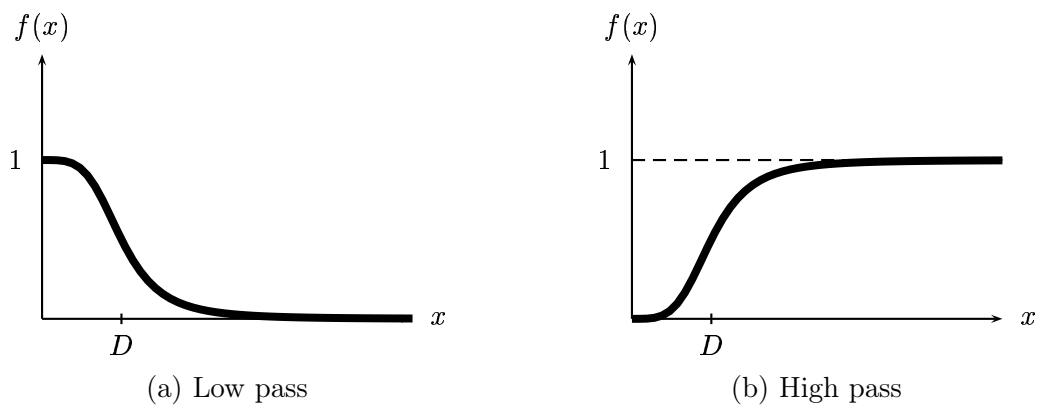
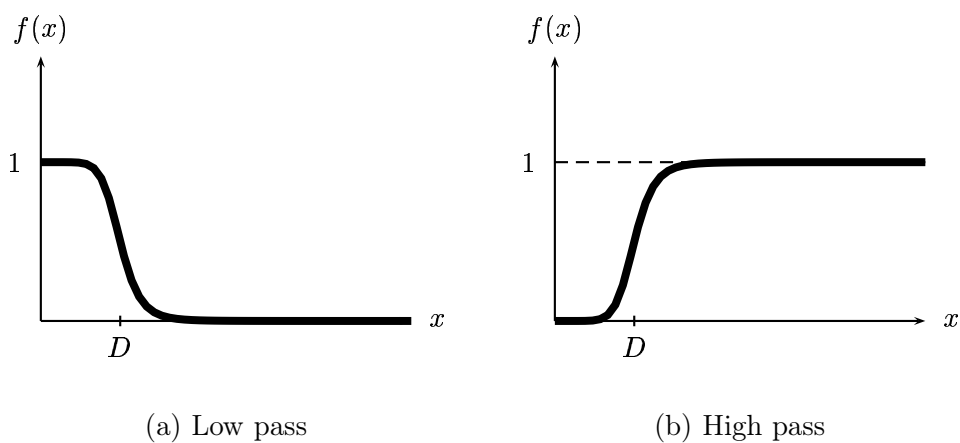


Figure 4.18: Ideal filter functions

Figure 4.19: Butterworth filter functions with $n = 2$ Figure 4.20: Butterworth filter functions with $n = 4$

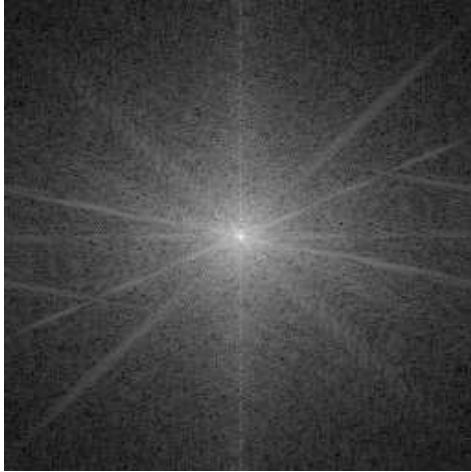
```
function out=lbutter(im,d,n)
% LBUTTER(IM,D,N) creates a low-pass Butterworth filter
% of the same size as image IM, with cutoff D, and order N
%
% Use:
%   x=imread('cameraman.tif');
%   l=lbutter(x,25,2);
%
height=size(im,1);
width=size(im,2);
[x,y]=meshgrid(-floor(width/2):floor((width-1)/2),-floor(height/2): ...
               floor((height-1)/2));
out=1./(1+(sqrt(2)-1)*((x.^2+y.^2)/d^2).^n);
```

Figure 4.21: A function to generate a low pass Butterworth filter

```
function out=hbutter(im,d,n)
% HBUTTER(IM,D,N) creates a high-pass Butterworth filter
% of the same size as image IM, with cutoff D, and order N
%
% Use:
%   x=imread('cameraman.tif');
%   l=hbutter(x,25,2);
%
out=1-lbutter(im,d,n);
```

Figure 4.22: A function to generate a high pass Butterworth filter

and this is shown in figure 4.23(a). Note that there is no sharp cutoff as seen in figure 4.14; also that the outer parts of the transform are not equal to zero, although they are dimmed considerably. Performing the inverse transform and displaying it as we have done previously produces figure 4.23(b). This is certainly a blurred image, but the ringing seen in figure 4.14 is completely absent. Compare



(a) The DFT after Butterworth low pass filtering



(b) The resulting image

Figure 4.23: Butterworth low pass filtering

the transform after multiplying with a Butterworth filter (figure 4.23(a)) with the original transform (in figure 4.13). The Butterworth filter does cause an attenuation of values away from the centre, even if they don't become suddenly zero, as with the ideal low pass filter in figure 4.14.

We can apply a Butterworth high pass filter similarly, first by creating the filter and applying it to the image transform:

```
>> bh=hbutter(cm,15,1);
>> cfbh=cf.*bh;
>> figure,fftshow(cfbh,'log')
```

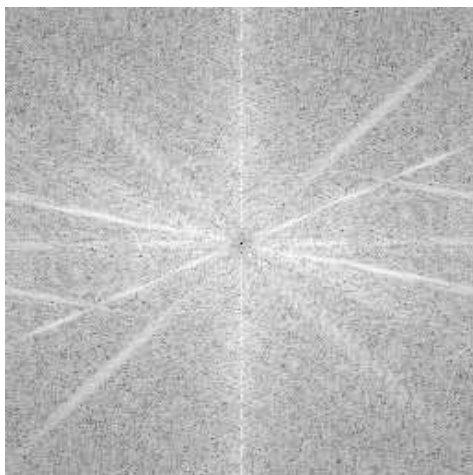
and then inverting and displaying the result:

```
>> cfbhi=ifft2(cfbh);
>> figure,fftshow(cfbhi,'abs')
```

The images are shown in figure 4.24

4.7.3 Gaussian filtering

We have met Gaussian filters in chapter 3, and we saw that they could be used for low pass filtering. However, we can also use Gaussian filters in the frequency domain. As with ideal and Butterworth filters, the implementation is very simple: create a Gaussian filter, multiply it by the image transform, and invert the result. Since Gaussian filters have the very nice mathematical property that a Fourier transform of a Gaussian is a Gaussian, we should get exactly the same results as when using a linear Gaussian spatial filter.



(a) The DFT after Butterworth high pass filtering



(b) The resulting image

Figure 4.24: Butterworth high pass filtering

Gaussian filters may be considered to be the most “smooth” of all the filters we have discussed so far, with ideal filters the least smooth, and Butterworth filters in the middle.

We can create Gaussian filters using the `fspecial` function, and apply them to our transform.

```
>> g1=mat2gray(fspecial('gaussian',256,10));
>> cg1=cf.*g1;
>> fftshow(cg1,'log')
>> g2=mat2gray(fspecial('gaussian',256,30));
>> cg2=cf.*g2;
>> figure,fftshow(cg2,'log')
```

Note the use of the `mat2gray` function. The `fspecial` function on its own produces a low pass Gaussian filter with a very small maximum:

```
>> g=fspecial('gaussian',256,10);
>> format long, max(g(:)), format

ans =

    0.00158757552679
```

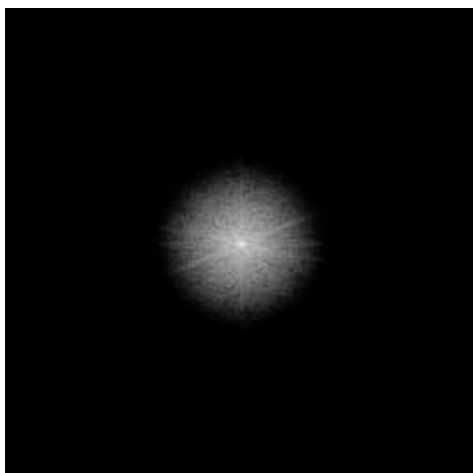
The reason is that `fspecial` adjusts its output to keep the volume under the Gaussian function always 1. This means that a wider function, with a large standard deviation, will have a low maximum. So we need to scale the result so that the central value will be 1; and `mat2gray` does that automatically.

The transforms are shown in figure 4.25(a) and (c). In each case, the final parameter of the `fspecial` function is the standard deviation; it controls the width of the filter. Clearly, the larger the standard deviation, the wider the function, and so the greater amount of the transform is preserved.

The results of the transform on the original image can be produced using the usual sequence of commands:


```
>> cgi1=ifft2(cg1);  
>> cgi2=ifft2(cg2);  
>> fftshow(cgi1,'abs');  
>> fftshow(cgi2,'abs');
```

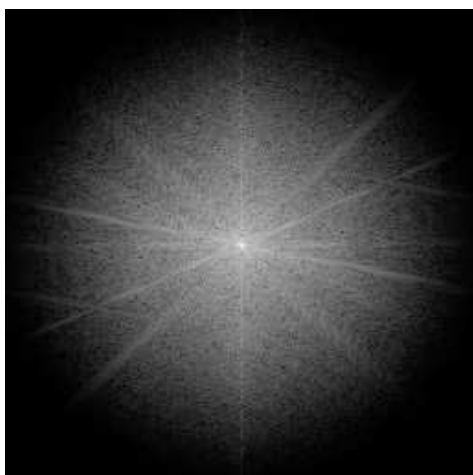
and the results are shown in figure 4.25(b) and (d)



(a) $\sigma = 10$



(b) Resulting image



(c) $\sigma = 30$



(d) Resulting image

Figure 4.25: Applying a Gaussian low pass filter in the frequency domain

We can apply a high pass Gaussian filter easily; we create a high pass filter by subtracting a low pass filter from 1.

```
>> h1=1-g1;  
>> h2=1-g2;
```

```

>> ch1=cf.*h1;
>> ch2=cf.*h2;
>> chi1=ifft2(ch1);
>> chi1=ifft2(ch1);
>> chi2=ifft2(ch2);
>> fftshow(chi1,'abs')
>> figure,fftshow(chi2,'abs')

```

and the images are shown in figure 4.26. As with ideal and Butterworth filters, the wider the high



(a) Using $\sigma = 10$

(b) Using $\sigma = 30$

Figure 4.26: Applying a Gaussian high pass filter in the frequency domain

pass filter, the more of the transform we are reducing, and the less of the original image will appear in the result.

Exercises

- By hand, compute the DFT of each of the following sequences:
 - $[2, 3, 4, 5]$
 - $[2, -3, 4, -5]$
 - $[-9, -8, -7, -6]$
 - $[-9, 8, -7, 6]$
 Compare your answers with those given by MATLAB's `fft` function.
- For each of the transforms you computed in the previous question, compute the inverse transform by hand.
- By hand, verify the convolution theorem for each of the following pairs of sequences:
 - $[2, 4, 6, 8]$ and $[-1, 2, -3, 4]$
 - $[4, 5, 6, 7]$ and $[3, 1, 5, -1]$
- Using MATLAB, verify the convolution theorem for the following pairs of sequences:
 - $[2, -3, 5, 6, -2, -1, 3, 7]$ and $[-1, 5, 6, 4, -3, -5, 1, 2]$
 - $[7, 6, 5, 4, -4, -5, -6, -7]$ and $[2, 2, -5, -5, 6, 6, -7, -7]$

Chapter 5

Image Restoration (1)

5.1 Introduction

Image *restoration* concerns the removal or reduction of degradations which have occurred during the acquisition of the image. Such degradations may include *noise*, which are errors in the pixel values, or optical effects such as out of focus blurring, or blurring due to camera motion. We shall see that some restoration techniques can be performed very successfully using neighbourhood operations, while others require the use of frequency domain processes. Image restoration remains one of the most important areas of image processing, but in this chapter the emphasis will be on the techniques for dealing with restoration, rather than with the degradations themselves, or the properties of electronic equipment which give rise to image degradation.

A model of image degradation

In the spatial domain, we might have an image $f(x, y)$, and a spatial filter $h(x, y)$ for which convolution with the image results in some form of degradation. For example, if $h(x, y)$ consists of a single line of ones, the result of the convolution will be a motion blur in the direction of the line. Thus we may write

$$g(x, y) = f(x, y) * h(x, y)$$

for the degraded image, where the symbol $*$ represents spatial filtering. However, this is not all. We must consider noise, which can be modelled as an additive function to the convolution. Thus if $n(x, y)$ represents random errors which may occur, we have as our degraded image:

$$g(x, y) = f(x, y) * h(x, y) + n(x, y).$$

We can perform the same operations in the frequency domain, where convolution is replaced by multiplication, and addition remains as addition, because of the linearity of the Fourier transform. Thus

$$G(i, j) = F(i, j)H(i, j) + N(i, j)$$

represents a general image degradation, where of course F , H and N are the Fourier transforms of f , h and n respectively.

If we knew the values of H and N we could recover F by writing the above equation as

$$F(i, j) = (G(i, j) - N(i, j))/H(i, j).$$

However, as we shall see, this approach may not be practical. Even though we may have some statistical information about the noise, we will not know the value of $n(x, y)$ or $N(i, j)$ for all, or even any, values. As well, dividing by $H(i, j)$ will cause difficulties if there are values which are close to, or equal to, zero.

5.2 Noise

We may define *noise* to be any degradation in the image signal, caused by external disturbance. If an image is being sent electronically from one place to another, via satellite or wireless transmission, or through networked cable, we may expect errors to occur in the image signal. These errors will appear on the image output in different ways depending on the type of disturbance in the signal. Usually we know what type of errors to expect, and hence the type of noise on the image; hence we can choose the most appropriate method for reducing the effects. Cleaning an image corrupted by noise is thus an important area of *image restoration*.

In this chapter we will investigate some of the standard noise forms, and the different methods of eliminating or reducing their effects on the image.

We will look at four different noise types, and how they appear on an image.

Salt and pepper noise

Also called *impulse noise*, *shot noise*, or *binary noise*. This degradation can be caused by sharp, sudden disturbances in the image signal; its appearance is randomly scattered white or black (or both) pixels over the image.

To demonstrate its appearance, we will first generate a grey-scale image, starting with a colour image:

```
>> tw=imread('twins.tif');
>> t=rgb2gray(tw);
```

To add noise, we use the MATLAB function `imnoise`, which takes a number of different parameters. To add salt and pepper noise:

```
>> t_sp=imnoise(t,'salt & pepper');
```

The amount of noise added defaults to 10%; to add more or less noise we include an optional parameter, being a value between 0 and 1 indicating the fraction of pixels to be corrupted. Thus, for example

```
>> imnoise(t,'salt & pepper',0.2);
```

would produce an image with 20% of its pixels corrupted by salt and pepper noise.

The twins image is shown in figure 5.1(a) and the image with noise is shown in figure 5.1(b).

Gaussian noise

Gaussian noise is an idealized form of *white noise*, which is caused by random fluctuations in the signal. We can observe white noise by watching a television which is slightly mistuned to a particular channel. Gaussian noise is white noise which is normally distributed. If the image is represented as I , and the Gaussian noise by N , then we can model a noisy image by simply adding the two:

$$I + N.$$

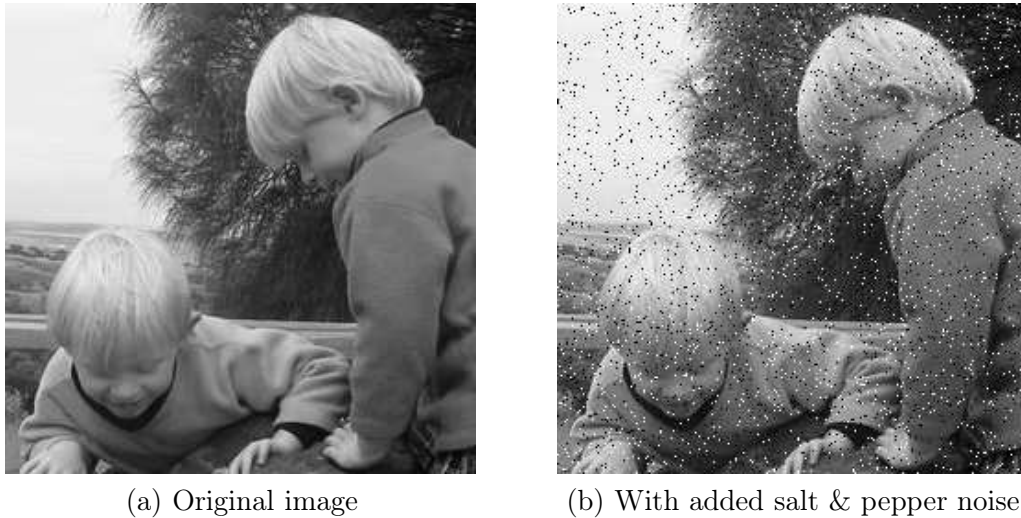


Figure 5.1: Noise on an image

Here we may assume that I is a matrix whose elements are the pixel values of our image, and N is a matrix whose elements are normally distributed. It can be shown that this is an appropriate model for noise. The effect can again be demonstrated by the `imnoise` function:

```
>> t_ga=imnoise(t,'gaussian');
```

As with salt and pepper noise, the “`gaussian`” parameter also can take optional values, giving the mean and variance of the noise. The default values are 0 and 0.01, and the result is shown in figure 5.2(a).

Speckle noise

Whereas Gaussian noise can be modelled by random values *added* to an image; *speckle noise* (or more simply just *speckle*) can be modelled by random values *multiplied* by pixel values, hence it is also called *multiplicative noise*. Speckle noise is a major problem in some radar applications. As above, `imnoise` can do speckle:

```
>> t_spk=imnoise(t,'speckle');
```

and the result is shown in figure 5.2(b). In MATLAB, speckle noise is implemented as

$$I(1 + N)$$

where I is the image matrix, and N consists of normally distributed values with mean 0. An optional parameter gives the variance of N ; its default value is 0.04.

Although Gaussian noise and speckle noise appear superficially similar, they are formed by two totally different methods, and, as we shall see, require different approaches for their removal.

Periodic noise

If the image signal is subject to a periodic, rather than a random disturbance, we might obtain an image corrupted by *periodic noise*. The effect is of bars over the image. The function `imnoise`

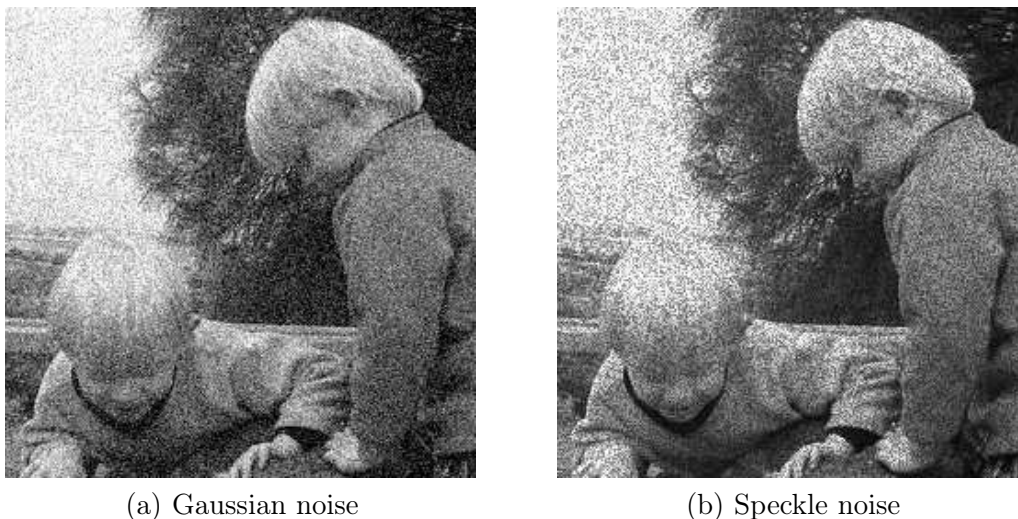


Figure 5.2: The twins image corrupted by Gaussian and speckle noise

does not have a periodic option, but it is quite easy to create our own, by adding a periodic matrix (using a trigonometric function), to our image:

```
>> s=size(t);
>> [x,y]=meshgrid(1:s(1),1:s(2));
>> p=sin(x/3+y/5)+1;
>> t_pn=(im2double(t)+p/2)/2;
```

and the resulting image is shown in figure 5.3.

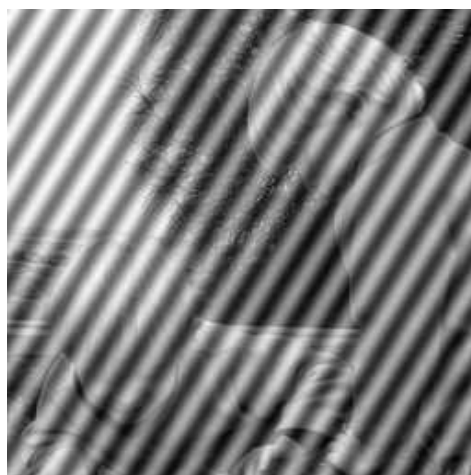


Figure 5.3: The twins image corrupted by periodic noise

Salt and pepper noise, Gaussian noise and speckle noise can all be cleaned by using spatial filtering techniques. Periodic noise, however, requires the use of frequency domain filtering. This is

because whereas the other forms of noise can be modelled as *local* degradations, periodic noise is a *global* effect.

5.3 Cleaning salt and pepper noise

Low pass filtering

Given that pixels corrupted by salt and pepper noise are high frequency components of an image, we should expect a low-pass filter should reduce them. So we might try filtering with an average filter:

```
>> a3=fspecial('average');  
>> t_sp_a3=filter2(a3,t_sp);
```

and the result is shown in figure 5.4(a). Notice, however, that the noise is not so much removed as “smeared” over the image; the result is not noticeably “better” than the noisy image. The effect is even more pronounced if we use a larger averaging filter:

```
>> a7=fspecial('average',[7,7]);  
>> t_sp_a7=filter2(a7,t_sp);
```

and the result is shown in figure 5.4(b).



(a) 3×3 averaging



(b) 7×7 averaging

Figure 5.4: Attempting to clean salt & pepper noise with average filtering

Median filtering

Median filtering seems almost tailor-made for removal of salt and pepper noise. Recall that the *median* of a set is the middle value when they are sorted. If there are an even number of values, the median is the mean of the middle two. A median filter is an example of a non-linear spatial filter;

using a 3×3 mask, the output value is the median of the values in the mask. For example:

50	65	52	→	50	52	57	58	60	61	63	65	255	→	60	
63	255	58		50	52	57	58	61	63	65	255	255		255	60
61	60	57		50	52	57	58	61	63	65	255	255		255	60

The operation of obtaining the median means that very large or very small values—noisy values—will end up at the top or bottom of the sorted list. Thus the median will in general replace a noisy value with one closer to its surroundings.

In MATLAB, median filtering is implemented by the `medfilt2` function:

```
>> t_sp_m3=medfilt2(t_sp);
```

and the result is shown in figure 5.5. The result is a vast improvement on using averaging filters. As



Figure 5.5: Cleaning salt and pepper noise with a median filter

with most functions, `medfilt2` takes an optional parameter; in this case a 2 element vector giving the size of the mask to be used.

If we corrupt more pixels with noise:

```
>> t_sp2=imnoise(t,'salt & pepper',0.2);
```

then `medfilt2` still does a remarkably good job, as shown in figure 5.6. To remove noise completely, we can either try a second application of the 3×3 median filter, the result of which is shown in figure 5.7(a) or try a 5×5 median filter on the original noisy image:

```
>> t_sp2_m5=medfilt2(t_sp2,[5,5]);
```

the result of which is shown in figure 5.7(b).

Rank-order filtering

Median filtering is a special case of a more general process called *rank-order filtering*. Rather than take the median of a set, we order the set and take the n -th value, for some predetermined value of n .



(a) 20% salt & pepper noise



(b) After median filtering

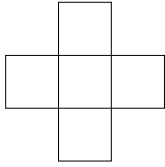
Figure 5.6: Using a 3×3 median filter on more noise

(a) Using medfilt2 twice

(b) using a 5×5 median filter

Figure 5.7: Cleaning 20% salt & pepper noise with median filtering

Thus median filtering using a 3×3 mask is equivalent to rank-order filtering with $n = 5$. Similarly, median filtering using a 5×5 mask is equivalent to rank-order filtering with $n = 13$. MATLAB implements rank-order filtering with the `ordfilt2` function; in fact the procedure for `medfilt2` is really just a wrapper for a procedure which calls `ordfilt2`. There is only one reason for using rank-order filtering instead of median filtering, and that is that it allows us to choose the median of non-rectangular masks. For example, if we decided to use as a mask a 3×3 cross shape:



then the median would be the *third* of these values when sorted. The command to do this is

```
>> ordfilt2(t_sp,3,[0 1 0;1 1 1;0 1 0]);
```

In general, the second argument of `ordfilt2` gives the value of the ordered set to take, and the third element gives the *domain*; the non-zero values of which specify the mask. If we wish to use a cross with size and width 5 (so containing nine elements), we can use:

```
>> ordfilt2(t_sp,5,[0 0 1 0 0;0 0 1 0 0;1 1 1 1 1;0 0 1 0 0;0 0 1 0 0])
```

An outlier method

Applying the median filter can in general be a slow operation: each pixel requires the sorting of at least nine values¹. To overcome this difficulty, Pratt [8] has proposed the use of cleaning salt and pepper noise by treating noisy pixels as *outliers*; that is, pixels whose grey values are significantly different from those of their neighbours. This leads to the following approach for noise cleaning:

1. Choose a threshold value D .
2. For a given pixel, compare its value p with the mean m of the values of its eight neighbours.
3. If $|p - m| > D$, then classify the pixel as noisy, otherwise not.
4. If the pixel is noisy, replace its value with m ; otherwise leave its value unchanged.

There is no MATLAB function for doing this, but it is very easy to write one. First, we can calculate the average of a pixel's eight neighbours by convolving with the linear filter

$$\frac{1}{8} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.125 & 0.125 & 0.125 \\ 0.125 & 0 & 0.125 \\ 0.125 & 0.125 & 0.125 \end{bmatrix}$$

We can then produce a matrix r consisting of 1's at only those places where the difference of the original and the filter are greater than D ; that is, where pixels are noisy. Then $1 - r$ will consist of ones at only those places where pixels are not noisy. Multiplying r by the filter replaces noisy values with averages; multiplying $1 - r$ with original values gives the rest of the output.

¹In fact, this is not the case with MATLAB, which uses a highly optimized method. Nonetheless, we introduce a different method to show that there are other ways of cleaning salt and pepper noise.

A MATLAB function for implementing this is shown in figure 5.8. An immediate problem with the outlier method is that it is not completely automatic—the threshold D must be chosen. An appropriate way to use the outlier method is to apply it with several different thresholds, and choose the value which provides the best results. Suppose we attempt to use the outlier method to clean

```
function res=outlier(im,d)
% OUTLIER(IMAGE,D) removes salt and pepper noise using an outlier method.
% This is done by using the following algorithm:
%
% For each pixel in the image, if the difference between its grey value
% and the average of its eight neighbours is greater than D, it is
% classified as noisy, and its grey value is changed to that of the
% average of its neighbours.
%
% IMAGE can be of type UINT8 or DOUBLE; the output is of type
% UINT8. The threshold value D must be chosen to be between 0 and 1.

f=[0.125 0.125 0.125; 0.125 0 0.125; 0.125 0.125 0.125];
imd=im2double(im);
imf=filter2(f,imd);
r=abs(imd-imf)-d>0;
res=im2uint8(r.*imf+(1-r).*imd);
```

Figure 5.8: A MATLAB function for cleaning salt and pepper noise using an outlier method

the noise from figure 5.1(b); that is, the twins image with 10% salt and pepper noise. Choosing $D = 0.2$ gives the image in figure 5.9(a). This is not as good a result as using a median filter: the affect of the noise has been lessened, but there are still noise “artifacts” over the image. In this case we have chosen a threshold which is too small. If we choose $D = 0.4$, we obtain the image in figure 5.9(b), which still has some noise artifacts, although in different places. We can see that a lower values of D tends to remove noise from dark areas, and a higher value of D tends to remove noise from light areas. A mid-way value, round about $D = 0.3$ does in fact produce an acceptable result, although not quite as good as median filtering.

Clearly using an appropriate value of D is essential for cleaning salt and pepper noise by this method. If D is too small, then too many “non-noisy” pixels will be classified as noisy, and their values changed to the average of their neighbours. This will result in a blurring effect, similar to that obtained by using an averaging filter. If D is chosen to be too large, then not enough noisy pixels will be classified as noisy, and there will be little change in the output.

The outlier method is not particularly suitable for cleaning large amounts of noise; for such situations the median filter is to be preferred. The outlier method may thus be considered as a “quick and dirty” method for cleaning salt and pepper noise when the median filter proves too slow.

5.4 Cleaning Gaussian noise

Image averaging

It may sometimes happen that instead of just *one* image corrupted with Gaussian noise, we have many different copies of it. An example is satellite imaging; if a satellite passes over the same spot many times, we will obtain many different images of the same place. Another example is in

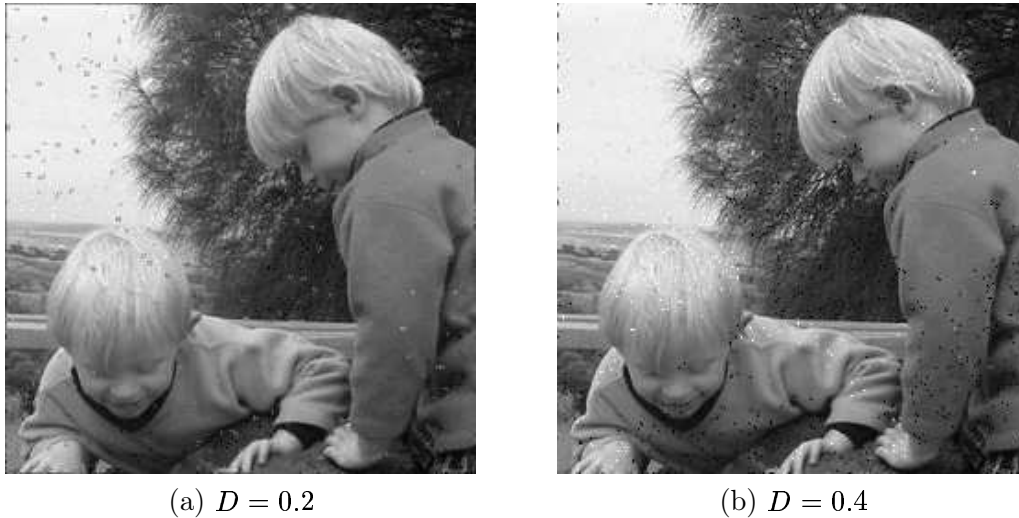


Figure 5.9: Applying the outlier method to 10% salt and pepper noise

microscopy: we might take many different images of the same object. In such a case a very simple approach to cleaning Gaussian noise is to simply take the average—the mean—of all the images. T

To see why this works, suppose we have 100 copies of our image, each with noise; then the i -th noisy image will be:

$$M + N_i$$

where M is the matrix of original values, and N_i is a matrix of normally distributed random values with mean 0. We can find the mean M' of these images by the usual add and divide method:

$$\begin{aligned} M' &= \frac{1}{100} \sum_{i=1}^{100} (M + N_i) \\ &= \frac{1}{100} \sum_{i=1}^{100} M + \frac{1}{100} \sum_{i=1}^{100} N_i \\ &= M + \frac{1}{100} \sum_{i=1}^{100} N_i \end{aligned}$$

Since N_i is normally distributed with mean 0, it can be readily shown that the mean of all the N_i 's will be close to zero—the greater the number of N_i 's; the closer to zero. Thus

$$M' \approx M$$

and the approximation is closer for larger number of images $M + N_i$.

We can demonstrate this with the twins image. We first need to create different versions with Gaussian noise, and then take the average of them. We shall create 10 versions. One way is to create an empty three-dimensional array of depth 10, and fill each “level” with a noisy image:

```
>> s=size(t);
>> t_ga10=zeros(s(1),s(2),10);
>> for i=1:10 t_ga10(:,:,i)=imnoise(t,'gaussian'); end
```

Note here that the “gaussian” option of `imnoise` calls the random number generator `randn`, which creates normally distributed random numbers. Each time `randn` is called, it creates a *different* sequence of numbers. So we may be sure that all levels in our three-dimensional array do indeed contain different images. Now we can take the average:

```
>> t_ga10_av=mean(t_ga10,3);
```

The optional parameter 3 here indicates that we are taking the mean along the *third* dimension of our array. The result is shown in figure 5.10(a). This is not quite clear, but is a vast improvement on the noisy image of figure 5.2(a). An even better result is obtained by taking the average of 100 images; this can be done by replacing 10 with 100 in the commands above, and the result is shown in figure 5.10(b). Note that this method only works if the Gaussian noise has mean 0.

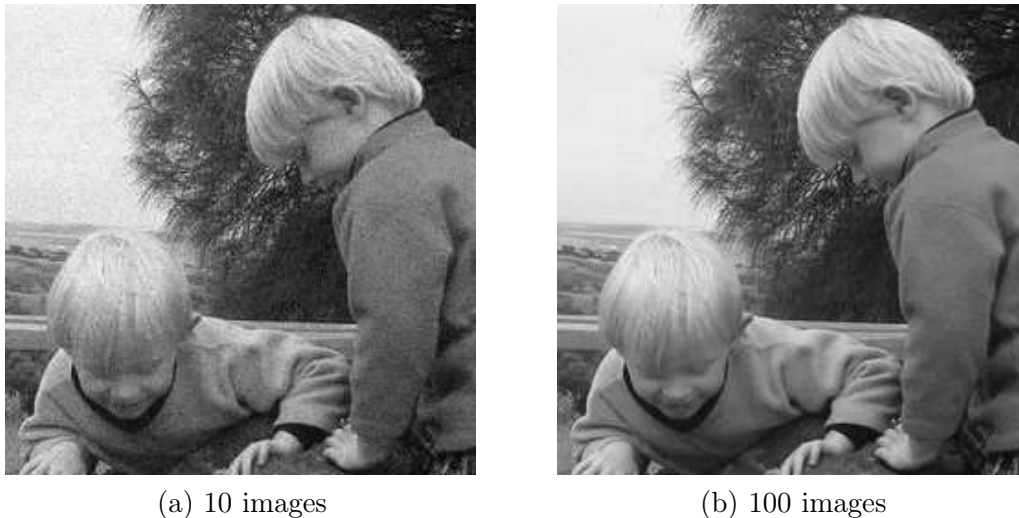


Figure 5.10: Image averaging to remove Gaussian noise

Average filtering

If the Gaussian noise has mean 0, then we would expect that an average filter would average the noise to 0. The larger the size of the filter mask, the closer to zero. Unfortunately, averaging tends to blur an image, as we have seen in chapter 3. However, if we are prepared to trade off blurring for noise reduction, then we can reduce noise significantly by this method.

Suppose we take the 3×3 and 5×5 averaging filters, and apply them to the noisy image `t_ga`.

```
>> a3=fspecial('average');
>> a5=fspecial('average',[5,5]);
>> tg3=filter2(a3,t_ga);
>> tg5=filter2(a5,t_ga);
```

The results are shown in figure 5.11. The results are not really particularly pleasing; although there has been some noise reduction, the “smeary” nature of the resulting images is unattractive.

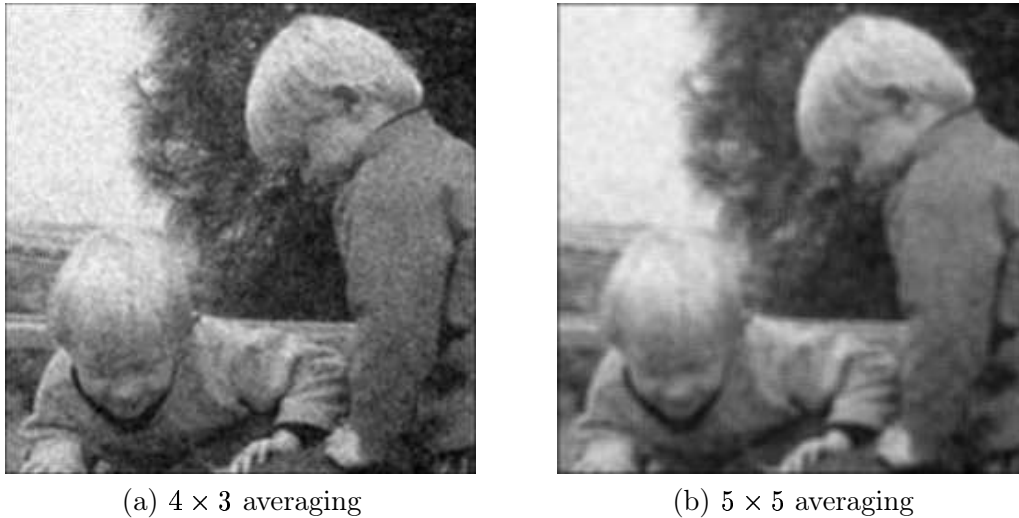


Figure 5.11: Using averaging filtering to remove Gaussian noise

Adaptive filtering

Adaptive filters are a class of filters which change their characteristics according to the values of the greyscales under the mask; they may act more like median filters, or more like average filters, depending on their position within the image. Such a filter can be used to clean Gaussian noise by using local statistical properties of the values under the mask.

One such filter is the *minimum mean-square error filter*; this is a non-linear spatial filter; and as with all spatial filters, is implemented by applying a function to the grey values under the mask.

Since we are dealing with additive noise, our noisy image M' can be written as

$$M' = M + N$$

where M is the original correct image, and N is the noise; which we assume to be normally distributed with mean 0. However, within our mask, the mean may not be zero; suppose the mean is m_f , and the variance in the mask is σ_f^2 . Suppose also that the variance of the noise over the entire image is known to be σ_g^2 . Then the output value can be calculated as

$$m_f + \frac{\sigma_f^2}{\sigma_f^2 + \sigma_g^2}(g - m_f)$$

where g is the current value of the pixel in the noisy image. Note that if the local variance σ_f^2 is high, then the fraction will be close to 1, and the output close to the original image value g . This is appropriate, as high variance implies high detail such as edges, which should be preserved. Conversely, if the local variance is low, such as in a background area of the image, the fraction is close to zero, and the value returned is close to the mean value m_f . See Lim [7] for details.

Another version of this filter [15] has output defined by

$$g - \frac{\sigma_g^2}{\sigma_f^2}(g - m_f)$$

and again the filter returns a value close to either g or m_f depending on whether the local variance is high or low.

In practice, m_f can be calculated by simply taking the mean of all grey values under the mask, and σ_f^2 by calculating the variance of all grey values under the mask. The value σ_g^2 may not necessarily be known, so a slight variant of the first filter may be used:

$$m_f + \frac{\max\{0, \sigma_f^2 - n\}}{\max\{\sigma_f^2, n\}}(g - m_f)$$

where n is the computed noise variance, and is calculated by taking the mean of all values of σ_f^2 over the entire image. This particular filter is implemented in MATLAB with the function `wiener2`. The name reflects the fact that this filter attempts to minimize the square of the difference between the input and output images; such filters are in general known as *Wiener filters*. However, Wiener filters are more usually applied in the frequency domain; see section 6.3 below.

Suppose we take the noisy image shown in figure 5.2(a), and attempt to clean this image with adaptive filtering. We will use the `wiener2` function, which can take an optional parameter indicating the size of the mask to be used. The default size is 3×3 . We shall create four images:

```
>> t1=wiener2(t_ga);
>> t2=wiener2(t_ga,[5,5]);
>> t3=wiener2(t_ga,[7,7]);
>> t4=wiener2(t_ga,[9,9]);
```

and these are shown in figure 5.12. Being a low pass filter, adaptive filtering does tend to blur edges and high frequency components of the image. But it does a far better job than using a low pass blurring filter.

We can achieve very good results for noise where the variance is not as high as that in our current image.

```
>> t2=imnoise(t,'gaussian',0,0.005);
>> imshow(t2)
>> t2w=wiener2(t2,[7,7]);
>> figure,imshow(t2w)
```

The image and its appearance after adaptive filtering as shown in figure 5.13. The result is a great improvement over the original noisy image. Notice in each case that there may be some blurring of the background, but the edges are preserved well, as predicted by our analysis of the adaptive filter formulas above.

Exercises

1. The arrays below represent small greyscale images. Compute the 4×4 image that would result in each case if the middle 16 pixels were transformed using a 3×3 median filter:

8	17	4	10	15	12
10	12	15	7	3	10
15	10	50	5	3	12
4	8	11	4	1	8
16	7	4	3	0	7
16	24	19	3	20	10

1	1	2	5	3	1
3	20	5	6	4	6
4	6	4	20	2	2
4	3	3	5	1	5
6	5	20	2	20	2
6	3	1	4	1	2

7	8	11	12	13	9
8	14	0	9	7	10
11	23	10	14	1	8
14	7	11	8	9	11
13	13	18	10	7	12
9	11	14	12	13	10

(a) 3×3 filtering(b) 5×5 filtering(a) 7×7 filtering(b) 9×9 filtering

Figure 5.12: Examples of adaptive filtering to remove Gaussian noise

Chapter 6

Image Restoration (2)

6.1 Removal of periodic noise

Periodic noise may occur if the imaging equipment (the acquisition or networking hardware) is subject to electronic disturbance of a repeating nature, such as may be caused by an electric motor. We can easily create periodic noise by overlaying an image with a trigonometric function:

```
>> [x,y]=meshgrid(1:256,1:256);  
>> p=1+sin(x+y/1.5);  
>> tp=(double(t)/128+p)/4;
```

where `cm` is the cameraman image from previous sections. The second line simply creates a sine function, and adjusts its output to be in the range 0–2. The last line first adjusts the cameraman image to be in the same range; adds the sine function to it, and divides by 4 to produce a matrix of type `double` with all elements in the range 0.0–1.0. This can be viewed directly with `imshow`, and it is shown in figure 6.1(a). We can produce its shifted DFT and this is shown in figure 6.1(b). The

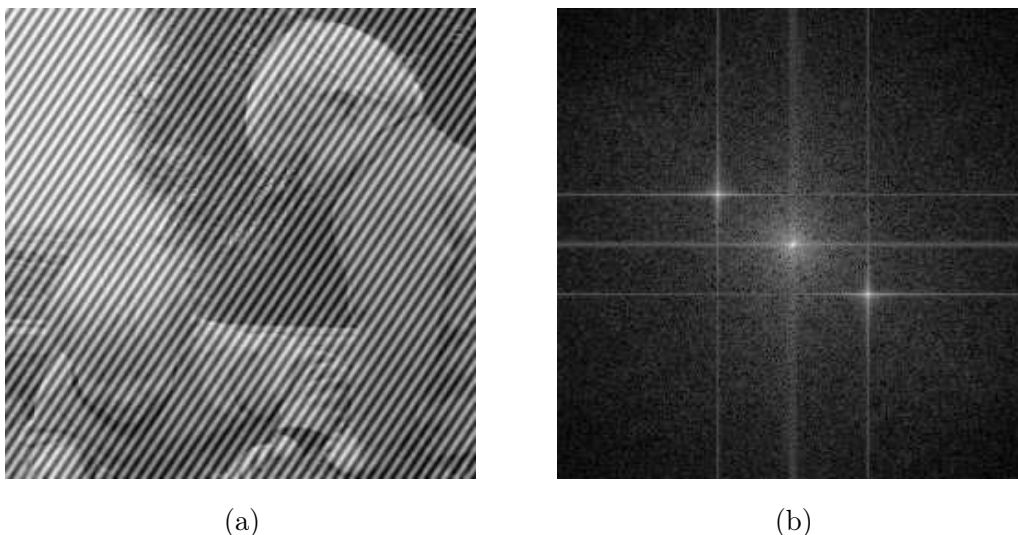


Figure 6.1: The twins image (a) with periodic noise, and (b) its transform

extra two “spikes” away from the centre correspond to the noise just added. In general the tighter

the period of the noise, the further from the centre the two spikes will be. This is because a small period corresponds to a high frequency (large change over a small distance), and is therefore further away from the centre of the shifted transform.

We will now remove these extra spikes, and invert the result. If we put `pixval` on and move around the image, we find that the spikes have row, column values of (156, 170) and (102, 88). These have the same distance from the centre: 49.0918. We can check this by

```
>> z=sqrt((x-129).^2+(y-129).^2);
>> z(156,170)
>> z(102,88)
```

There are two methods we can use to eliminate the spikes, we shall look at both of them.

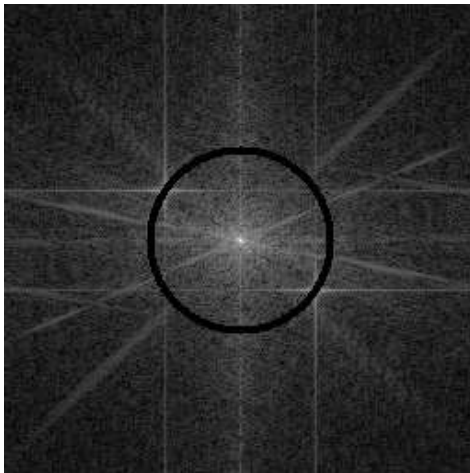
Band reject filtering. We create a filter consisting of ones with a ring of zeroes; the zeroes lying at a radius of 49 from the centre:

```
>> br=(z < 47 | z > 51);
```

where `z` is the matrix consisting of distances from the origin. This particular ring will have a thickness large enough to cover the spikes. Then as before, we multiply this by the transform:

```
>> tbr=tf.*br;
```

and this is shown in figure 6.2(a). The result is that the spikes have been blocked out by this filter. Taking the inverse transform produces the image shown in figure 6.2(b). Note that not all the noise



(a) A band-reject filter



(b) After inversion

Figure 6.2: Removing periodic noise with a band-reject filter

has gone, but a significant amount has, especially in the centre of the image.

Notch filtering. With a notch filter, we simply make the rows and columns of the spikes zero:

```
>> tf(156,:) = 0;
>> tf(102,:) = 0;
>> tf(:,170) = 0;
>> tf(:,88) = 0;
```

and the result is shown in figure 6.3(a). The image after inversion is shown in figure 6.3(b). As

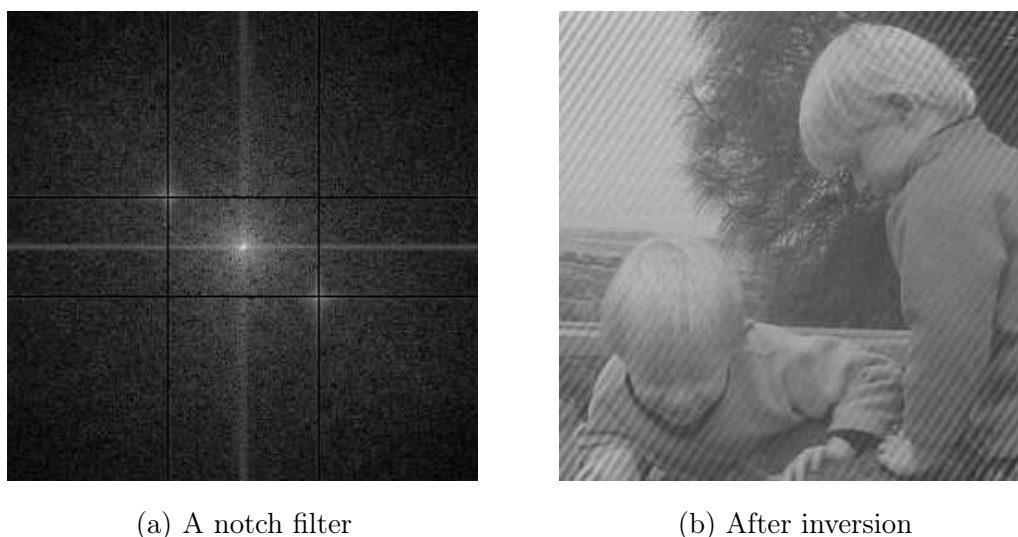


Figure 6.3: Removing periodic noise with a notch filter

before, much of the noise in the centre has been removed. Making more rows and columns of the transform zero would result in a larger reduction of noise.

6.2 Inverse filtering

We have seen that we can perform filtering in the Fourier domain by multiplying the DFT of an image by the DFT of a filter: this is a direct use of the convolution theorem. We thus have

$$Y(i, j) = X(i, j)F(i, j)$$

where X is the DFT of the image; F is the DFT of the filter, and Y is the DFT of the result. If we are given Y and F , then we should be able to recover the (DFT of the) original image X simply by dividing by F :

$$X(i, j) = \frac{Y(i, j)}{F(i, j)}. \quad (6.1)$$

Suppose, for example we take the wombats image `wombats.tif`, and blur it using a low-pass Butterworth filter:

```
>> w=imread('wombats.tif');
>> wf=fftshift(fft2(w));
>> b=lbutter(w,15,2);
>> wb=wf.*b;
```

```
>> wba=abs(fft2(wb));
>> wba=uint8(255*mat2gray(wba));
>> imshow(wba)
```

The result is shown on the left in figure 6.4. We can attempt to recover the original image by dividing by the filter:

```
>> w1=fftshift(fft2(wba))./b;
>> w1a=abs(fft2(w1));
>> imshow(mat2gray(w1a))
```

and the result is shown on the right in figure 6.4. This is no improvement! The trouble is that some

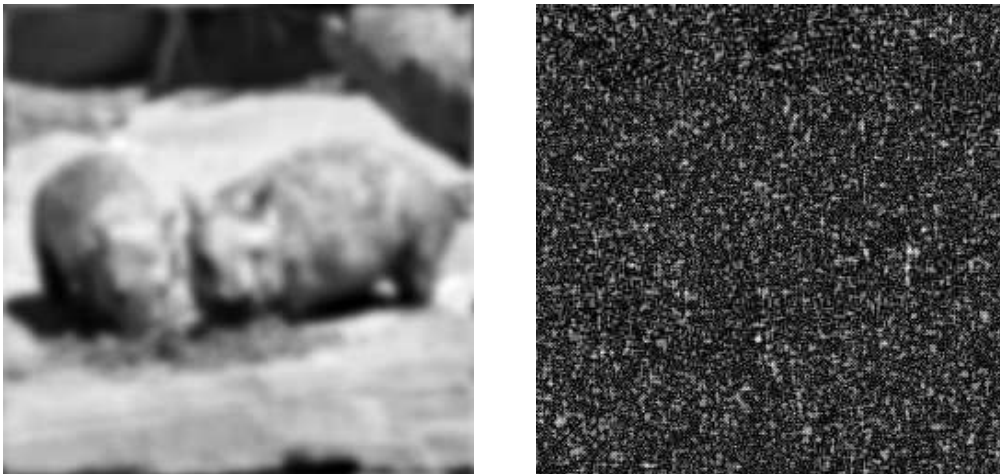


Figure 6.4: An attempt at inverse filtering

elements of the Butterworth matrix are very small, so dividing produces very large values which dominate the output. We can deal with this problem in two ways:

1. Apply a low pass filter L to the division:

$$X(i, j) = \frac{Y(i, j)}{F(i, j)} L(i, j).$$

This should eliminate very low (or zero) values.

2. “Constrained division”: choose a threshold value d , and if $|F(i, j)| < d$, we don’t perform a division, but just keep our original value. Thus:

$$X(i, j) = \begin{cases} \frac{Y(i, j)}{F(i, j)} & \text{if } |F(i, j)| \geq d, \\ Y(i, j) & \text{if } |F(i, j)| < d. \end{cases}$$

We can apply the first method by multiplying a Butterworth low pass filter to the matrix `c1` above:

```
>> wbf=fftshift(fft2(wba));  
>> w1=(wbf./b).*lbutter(w,40,10);  
>> w1a=abs(iff2(w1));  
>> imshow(mat2gray(w1a))
```

Figure 6.5 shows the results obtained by using a different cutoff radius of the Butterworth filter each time: (a) uses 40 (as in the MATLAB commands just given); (b) uses 60; (c) uses 80, and (d) uses 100. It seems that using a low pass filter with a cutoff round about 60 will yield the best results.



(a)



(b)



(c)



(d)

Figure 6.5: Inverse filtering using low pass filtering to eliminate zeros

After we use larger cutoffs, the result degenerates.

We can try the second method; to implement it we simply make all values of the filter which are too small equal to 1:

```
>> d=0.01;
```

```

>> b=lbutter(w,15,2);b(find(b<d))=1;
>> w1=fftshift(fft2(wba))./b;
>> w1a=abs(ifft2(w1));
>> imshow(mat2gray(w1a))

```

Figure 6.6 shows the results obtained by using a different cutoff radius of the Butterworth filter each time: (a) uses $d = 0.01$ (as in the MATLAB commands just given); (b) uses $d = 0.005$; (c) uses $d = 0.002$, and (d) uses $d = 0.001$. It seems that using a threshold d in the range $0.002 \leq d \leq 0.005$



(a)



(b)



(c)



(d)

Figure 6.6: Inverse filtering using constrained division

produces reasonable results.

Motion deblurring

We can consider the removal of blur caused by motion to be a special case of inverse filtering. Suppose we take an image and blur it by a small amount.

```
>> bc=imread('board.tif');
>> bg=im2uint8(rgb2gray(bc));
>> b=bg(100:355,50:305);
>> imshow(b)
```

These commands simply take the colour image of a circuit board (the image `board.tif`), makes a greyscale version of data type `uint8`, and picks out a square subimage. The result is shown as figure 6.7(a). To blur it, we can use the `blur` parameter of the `fspecial` function.

```
>> m=fspecial('motion',7,0);
>> bm=imfilter(b,m);
>> imshow(bm)
```

and the result is shown as figure 6.7(b). The result of the blur has effectively obliterated the text

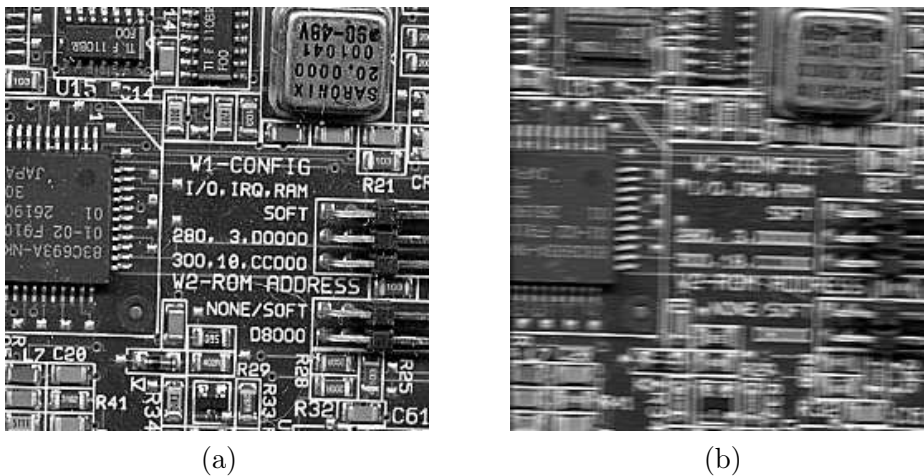


Figure 6.7: The result of motion blur

on the image.

To deblur the image, we need to divide its transform by the transform corresponding to the blur filter. This means that we first must create a matrix corresponding to the transform of the blur:

```
>> m2=zeros(256,256);
>> m2(1,1:7)=m;
>> mf=fft2(m2);
```

Now we can attempt to divide by this transform.

```
>> bmi=ifft2(fft2(bm)./mf);
>> fftshow(bmi,'abs')
```

and the result is shown in figure 6.8(a). As with inverse filtering, the result is not particularly good, because the values close to zero in the matrix `mf` have tended to dominate the result. As above, we can constrain the division by only dividing by values which are above a certain threshold.

```
>> d=0.02;
```

```
>> mf=fft2(m2);mf(find(abs(mf)<d))=1;
>> bmi=ifft2(fft2(bm)./mf);
>> imshow(mat2gray(abs(bmi))*2)
```

where the last multiplication by 2 just brightens the result, which is shown in figure 6.8(b). The

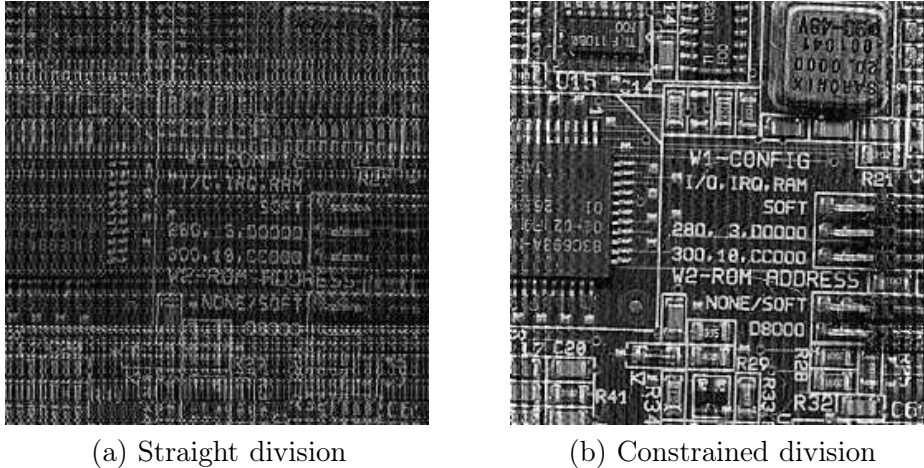


Figure 6.8: Attempts at removing motion blur

writing, especially in the centre of the image, is now quite legible.

6.3 Wiener filtering

As we have seen from the previous section, inverse filtering does not necessarily produce particularly pleasing results. The situation is even worse if the original image has been corrupted by noise. Here we would have an image X filtered with a filter F and corrupted by noise N . If the noise is additive (for example, Gaussian noise), then the linearity of the Fourier transform gives us

$$Y(i, j) = X(i, j)F(i, j) + N(i, j)$$

and so

$$X(i, j) = \frac{Y(i, j) - N(i, j)}{F(i, j)}$$

as we have seen in the introduction to this chapter. So not only do we have the problem of dividing by the filter, we have the problem of dealing with noise. In such a situation the presence of noise can have a catastrophic effect on the inverse filtering: the noise can completely dominate the output, making direct inverse filtering impossible.

To introduce Wiener filtering, we shall discuss a more general question: given a degraded image M' of some original image M and a restored version R , what measure can we use to say whether our restoration has done a good job? Clearly we would like R to be as close as possible to the “correct” image M . One way of measuring the closeness of R to M is by adding the squares of all differences:

$$\sum (m_{i,j} - r_{i,j})^2$$

where the sum is taken over all pixels of R and M (which we assume to be of the same size). This sum can be taken as a measure of the closeness of R to M . If we can minimize this value, we may be sure that our procedure has done as good a job as possible. Filters which operate on this principle of *least squares* are called *Wiener filters*. We can obtain X by

$$X(i, j) \approx \left[\frac{1}{F(i, j)} \frac{|F(i, j)|^2}{|F(i, j)|^2 + K} \right] Y(i, j) \quad (6.2)$$

where K is a constant [4]. This constant can be used to approximate the amount of noise: if the variance σ^2 of the noise is known, then $K = 2\sigma^2$ can be used. Otherwise, K can be chosen interactively (in other words, by trial and error) to yield the best result. Note that if $K = 0$, then equation 6.2 reduces to equation 6.1.

We can easily implement equation 6.2:

```
>> K=0.01;
>> wbf=fftshift(fft2(wba));
>> w1=wbf.*(abs(b).^2./(abs(b).^2+K)./b); % This is the equation
>> w1a=abs(ifft2(w1));
>> imshow(mat2gray(w1a))
```

The result is shown in figure 6.9(a). Images (b), (c) and (d) in this figure show the results with $K = 0.001$, $K = 0.0001$ and $K = 0.00001$ respectively. Thus as K becomes very small, noise starts to dominate the image.

Exercises

1. Add the sine waves to the engineer face using the same commands as for the cameraman:

```
>> [x,y]=meshgrid(1:256,1:256);
>> s=1+sin(x+y/1.5);
>> ep=(double(en)/128+s)/4;
```

Now attempt to remove the noise using band-reject filtering or notch filtering. Which one gives the best result?

2. For each of the following sine commands:

- (a) $s=1+\sin(x/3+y/5)$;
- (b) $s=1+\sin(x/5+y/1.5)$;
- (c) $s=1+\sin(x/6+y/6)$;

add the sine wave to the image as shown in the previous question, and attempt to remove the resulting periodic noise using band-reject filtering or notch filtering.

Which of the three is easiest to “clean up”?

3. Apply a 5×5 blurring filter to the cameraman image with `imfilter`. Attempt to deblur the result using inverse filtering with constrained division. Which threshold gives the best results?
4. Repeat the previous question using a 7×7 blurring filter.



(a)



(b)



(c)



(d)

Figure 6.9: Wiener filtering

Chapter 7

Image Segmentation (1)

7.1 Introduction

Segmentation refers to the operation of partitioning an image into component parts, or into separate objects. In this chapter, we shall investigate two very important topics: thresholding, and edge detection.

7.2 Thresholding

7.2.1 Single thresholding

A greyscale image is turned into a binary (black and white) image by first choosing a grey level T in the original image, and then turning every pixel black or white according to whether its grey value is greater than or less than T :

A pixel becomes $\begin{cases} \text{white if its grey level is } > T, \\ \text{black if its grey level is } \leq T. \end{cases}$

Thresholding is a vital part of image *segmentation*, where we wish to isolate objects from the background. It is also an important component of robot vision.

Thresholding can be done very simply in MATLAB. Suppose we have an 8 bit image, stored as the variable X . Then the command

```
X>T
```

will perform the thresholding. We can view the result with `imshow`. For example, the commands

```
>> r=imread('rice.tif');  
>> imshow(r),figure,imshow(r>110)
```

will produce the images shown in figure 7.1. The resulting image can then be further processed to find the number, or average size of the grains.

To see how this works, recall that in MATLAB, an operation on a single number, when applied to a matrix, is interpreted as being applied simultaneously to all elements of the matrix. The command $X>T$ will thus return 1 (for true) for all those pixels for which the grey values are greater than T , and 0 (for false) for all those pixels for which the grey values are less than or equal to T . We thus end up with a matrix of 0's and 1's, which can be viewed as a binary image.

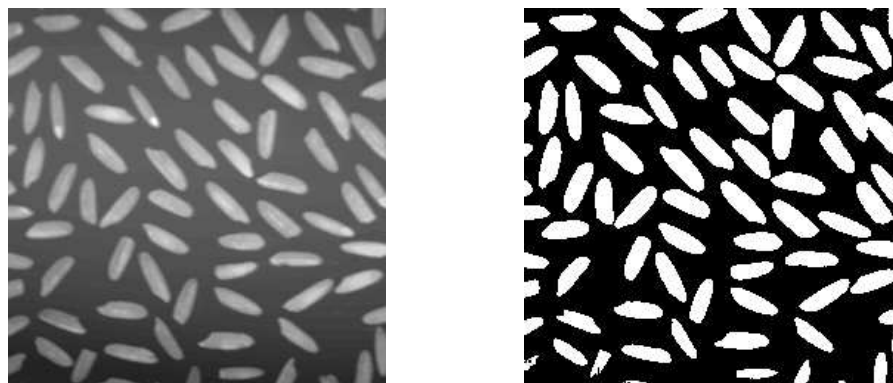


Figure 7.1: Thresholded image of rice grains

The rice image shown above has light grains on a dark background; an image with dark objects over a light background may be treated the same::

```
>> b=imread('bacteria.tif');  
>> imshow(b),figure,imshow(b>100)
```

will produce the images shown in figure 7.2.

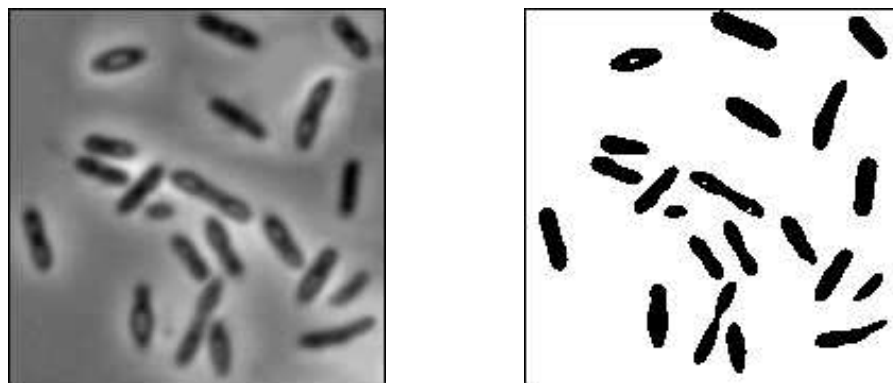


Figure 7.2: Thresholded image of bacteria

As well as the above method, MATLAB has the `im2bw` function, which thresholds an image of *any data type*, using the general syntax

```
im2bw(image,level)
```

where `level` is a value between 0 and 1 (inclusive), indicating the fraction of grey values to be turned white. This command will work on greyscale, coloured and indexed images of data type `uint8`, `uint16` or `double`. For example, the thresholded rice and bacteria images above could be obtained using

```
>> im2bw(r,0.43);
>> im2bw(b,0.39);
```

The `im2bw` function automatically scales the value `level` to a grey value appropriate to the image type, and then performs a thresholding by our first method.

As well as isolating objects from the background, thresholding provides a very simple way of showing hidden aspects of an image. For example, the image `paper.tif` appears all white, as nearly all the grey values are very high. However, thresholding at a high level produces an image of far greater interest. We can use the commands

```
>> p=imread('paper1.tif');
>> imshow(p),figure,imshow(p>241)
```

to provide the images shown in figure 7.3.

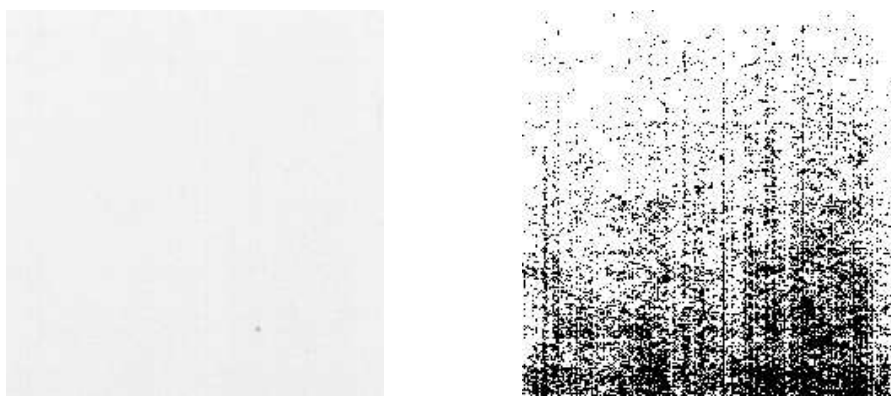


Figure 7.3: The paper image and result after thresholding

7.2.2 Double thresholding

Here we choose two values T_1 and T_2 and apply a thresholding operation as:

A pixel becomes $\begin{cases} \text{white if its grey level is between } T_1 \text{ and } T_2, \\ \text{black if its grey level is otherwise.} \end{cases}$

We can implement this by a simple variation on the above method:

```
X>T1 & X<T2
```

Since the ampersand acts as a logical “and”, the result will only produce a one where both inequalities are satisfied. Consider the following sequence of commands, which start by producing an 8-bit grey version of the indexed image `spine.tif`:

```
>> [x,map]=imread('spine.tif');
>> s=uint8(256*ind2gray(x,map));
>> imshow(s),figure,imshow(s>115 & s<125)
```

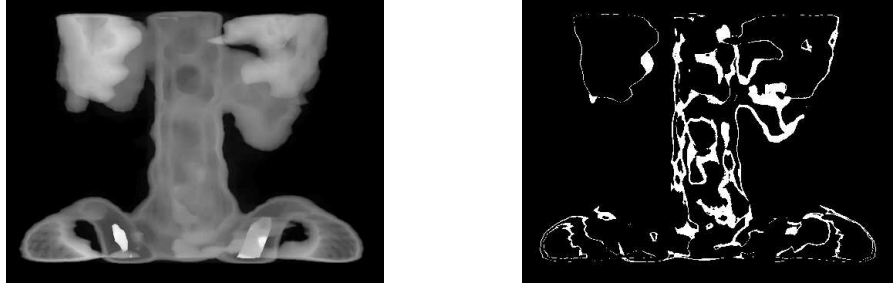


Figure 7.4: The image `spine.tif` and the result after double thresholding

The output is shown in figure 7.4. Note how double thresholding brings out subtle features of the spine which single thresholding would be unable to do. We can obtain similar results using `im2bw`:

```
imshow(im2bw(x,map,0.45)&~{im2bw(x,map,0.5)})
```

but this is somewhat slower because of all the extra computation involved when dealing with an indexed image.

7.3 Applications of thresholding

We have seen that thresholding can be useful in the following situations:

1. When we want to remove unnecessary detail from an image, to concentrate on essentials. Examples of this were given in the rice and bacteria images: by removing all grey level information, the rice and bacteria were reduced to binary blobs. But this information may be all we need to investigate sizes, shapes, or numbers of blobs.
2. To bring out hidden detail. This was illustrated with paper and spine images. In both, the detail was obscured because of the similarity of the grey levels involved.

But thresholding can be vital for other purposes. We list a few more:

3. When we want to remove a varying background from text or a drawing. We can simulate a varying background by taking the image `text.tif` and placing it on a random background. This can be easily implemented with some simple MATLAB commands:

```
>> r=rand(256)*128+127;
>> t=imread('text.tif');
>> tr=uint8(r.*double(not(t)));
>> imshow(tr)
```

The first command simply uses the `rand` function (which produces matrices of uniformly generated random numbers between 0.0 and 1.0), and scales the result so the random numbers are between 127 and 255. We then read in the text image, which shows white text on a dark background.

The third command does several things at once: `not(t)` reverses the text image so as to have black text on a white background; `double` changes the numeric type so that the matrix can be used with arithmetic operations; finally the result is multiplied into the random matrix, and the whole thing converted to `uint8` for display. The result is shown on the left in figure 7.5.

If we threshold this image and display the result with

```
>> imshow(tr>100)
```

the result is shown on the right in figure 7.5, and the background has been completely removed.

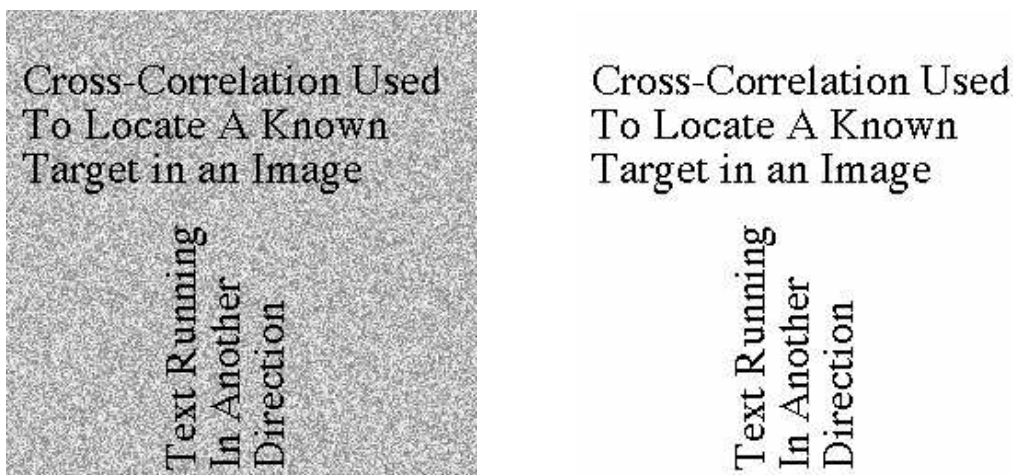


Figure 7.5: Text on a varying background, and thresholding

7.4 Adaptive thresholding

Sometimes it is not possible to obtain a single threshold value which will isolate an object completely. This may happen if both the object and its background vary. For example, suppose we take the circles image and adjust it so that both the circles and the background vary in brightness across the image.

```
>> c=imread('circles.tif');
>> x=ones(256,1)*[1:256];
>> c2=double(c).*(x/2+50)+(1-double(c)).*x/2;
>> c3=uint8(255*mat2gray(c2));
```

Figure 7.6 shows an attempt at thresholding, using `graythresh`.

```
>> t=graythresh(c3)

t =

    0.4196
```

```
>> ct=im2bw(c3,t);
```

As you see, the result is not particularly good; not all of the object has been isolated from its background. Even if different thresholds are used, the results are similar. Figure 7.7 illustrates the

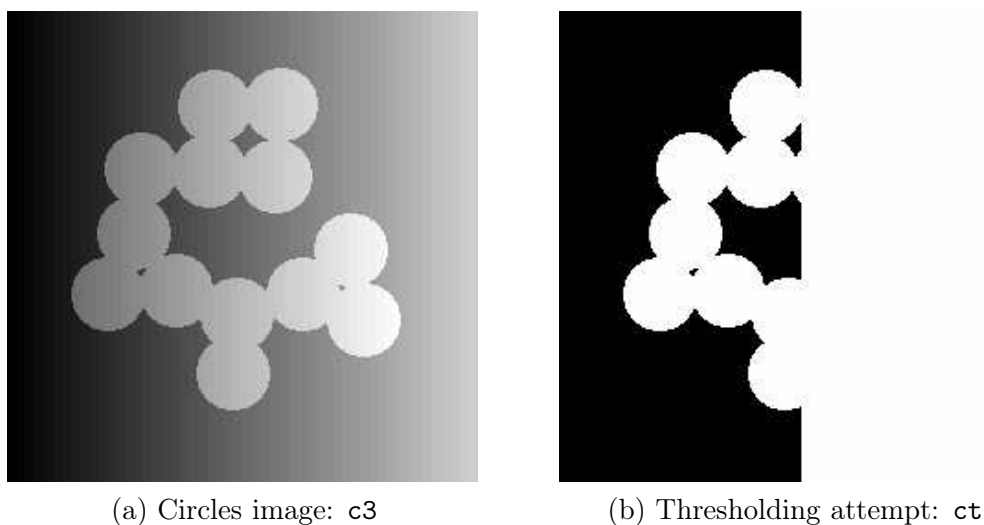


Figure 7.6: An attempt at thresholding

reason why a single threshold cannot work. In this figure the image is being shown as a function; the threshold is shown on the right as a horizontal plane. It can be seen that no position of the

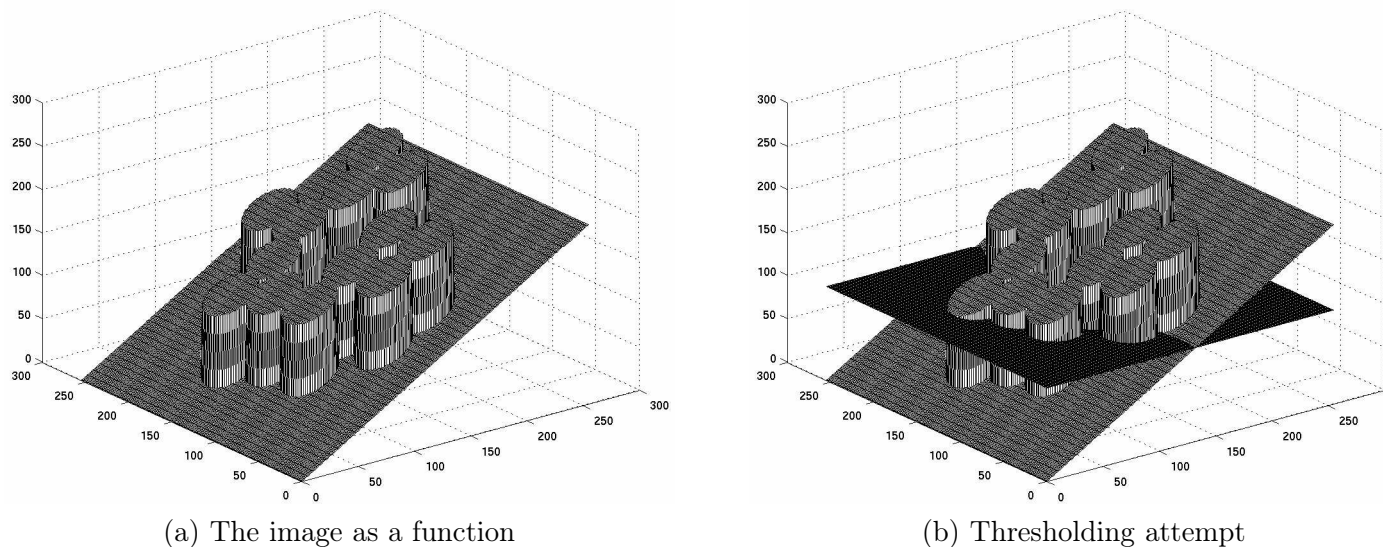


Figure 7.7: An attempt at thresholding—functional version

plane can cut off the circles from the background.

What can be done in a situation like this is to cut the image into small pieces, and apply thresholding to each piece individually. Since in this particular example the brightness changes from left to right, we shall cut up the image into four pieces:

```
>> p1=c3(:,1:64);
```



```
>> p2=c3(:,65:128);
>> p3=c3(:,129:192);
>> p4=c3(:,193:256);
```

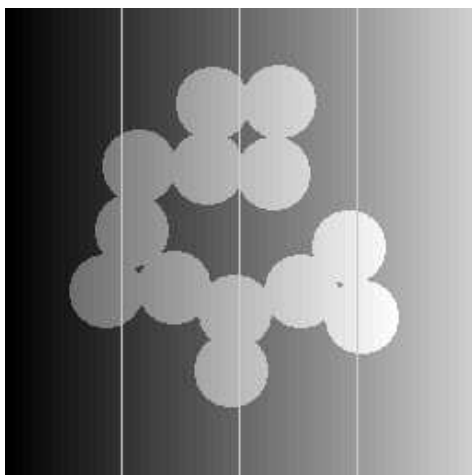
Figure 7.8(a) shows how the image is sliced up. Now we can threshold each piece:

```
>> g1=im2bw(p1,graythresh(p1));
>> g2=im2bw(p2,graythresh(p2));
>> g3=im2bw(p3,graythresh(p3));
>> g4=im2bw(p4,graythresh(p4));
```

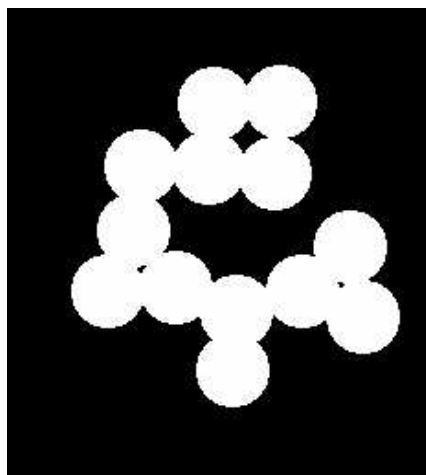
and now display them as a single image:

```
>> imshow([g1 g2 g3 g4])
```

and the result is shown in figure 7.8(b). The above commands can be done much more simply by



(a) Cutting up the image



(b) Thresholding each part separately

Figure 7.8: Adaptive thresholding

using the command `blkproc`, which applies a particular function to each block of the image. We can define our function with

```
>> fun=inline('im2bw(x,graythresh(x))');
```

Notice that this is the same as the commands used above to create `g1`, `g2`, `g3` and `g4` above, except that now `x` is used to represent a general input variable.

The function can then be applied it to the image `t3` with

```
>> t4=blkproc(t3,[256,64],fun);
```

What this command means is that we apply our function `fun` to each distinct 256×64 block of our image.

Chapter 8

Image Segmentation (2)

8.1 Edge detection

Edges contain some of the most useful information in an image. We may use edges to measure the size of objects in an image; to isolate particular objects from their background; to recognize or classify objects. There are a large number of edge-finding algorithms in existence, and we shall look at some of the more straightforward of them. The general MATLAB command for finding edges is

```
edge(image, 'method', parameters...)
```

where the parameters available depend on the method used. In this chapter, we shall show how to create edge images using basic filtering methods, and discuss the MATLAB edge function.

An *edge* may be loosely defined as a local discontinuity in the pixel values which exceeds a given threshold. More informally, an edge is an observable difference in pixel values. For example, consider the two 4 blocks of pixels shown in figure 8.1.

51	52	53	59
54	52	53	62
50	52	53	68
55	52	53	55

50	53	155	160
51	53	160	170
52	53	167	190
51	53	162	155

Figure 8.1: Blocks of pixels

In the right hand block, there is a clear difference between the grey values in the second and third columns, and for these values the differences exceed 100. This would be easily discernible in an image—the human eye can pick out grey differences of this magnitude with relative ease. Our aim is to develop methods which will enable us to pick out the edges of an image.

8.2 Derivatives and edges

8.2.1 Fundamental definitions

Consider the image in figure 8.2, and suppose we plot the gray values as we traverse the image from left to right. Two types of edges are illustrated here: a *ramp edge*, where the grey values change slowly, and a *step edge*, or an *ideal edge*, where the grey values change suddenly.

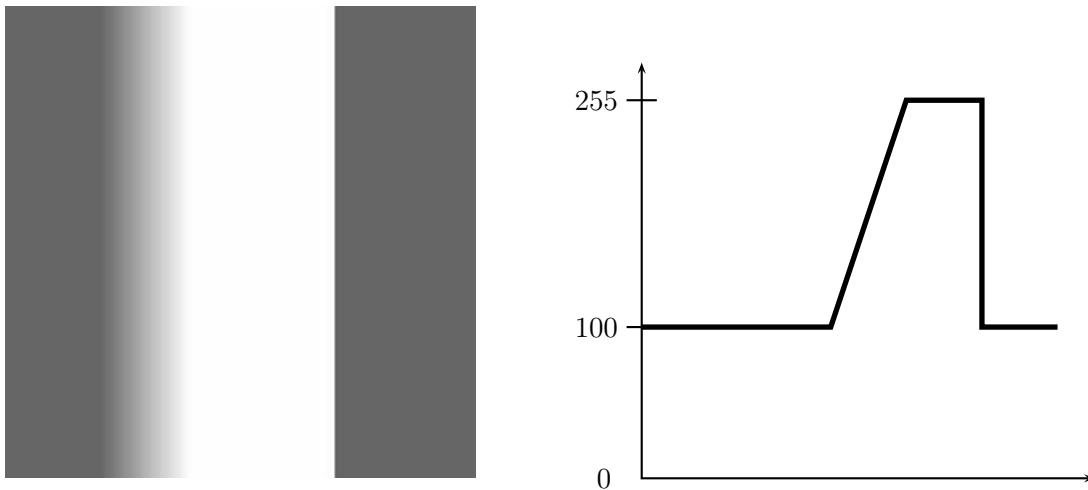


Figure 8.2: Edges and their profiles

Suppose the function which provides the profile in figure 8.2 is $f(x)$; then its derivative $f'(x)$ can be plotted; this is shown in figure 8.3. The derivative, as expected, returns zero for all constant

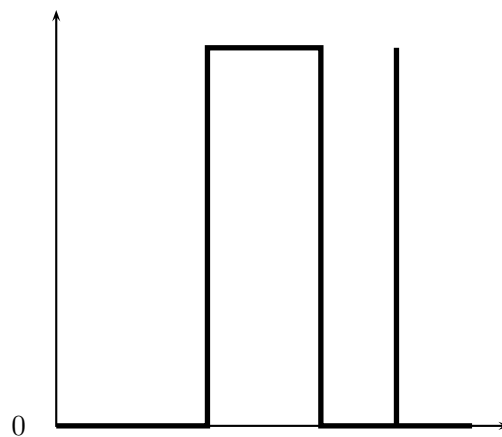


Figure 8.3: The derivative of the edge profile

sections of the profile, and is non zero (in this example) only in those parts of the image in which difference occur.

Many edge finding operators are based on differentiation; to apply the continuous derivative to a discrete image, first recall the definition of the derivative:

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

Since in an image, the smallest possible value of h is 1, being the difference between the index values of two adjacent pixels, a discrete version of the derivative expression is

$$f(x+1) - f(x).$$

Other expressions for the derivative are

$$\lim_{h \rightarrow 0} \frac{f(x) - f(x-h)}{h}, \quad \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

with discrete counterparts

$$f(x) - f(x - 1), \quad (f(x + 1) - f(x - 1))/2.$$

For an image, with two dimensions, we use partial derivatives; an important expression is the *gradient*, which is the vector defined by

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}$$

which for a function $f(x, y)$ points in the direction of its greatest increase. The direction of that increase is given by

$$\tan^{-1} \left(\frac{\partial f / \partial y}{\partial f / \partial x} \right)$$

and its magnitude by

$$\sqrt{\left(\frac{\partial f}{\partial x} \right)^2 + \left(\frac{\partial f}{\partial y} \right)^2}.$$

Most edge detection methods are concerned with finding the magnitude of the gradient, and then applying a threshold to the result.

8.2.2 Some edge detection filters

Using the expression $f(x + 1) - f(x - 1)$ for the derivative, leaving the scaling factor out, produces horizontal and vertical filters:

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

These filters will find vertical and horizontal edges in an image and produce a reasonably bright result. However, the edges in the result can be a bit “jerky”; this can be overcome by smoothing the result in the opposite direction; by using the filters

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Both filters can be applied at once, using the combined filter:

$$P_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

This filter, and its companion for finding horizontal edges:

$$P_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

are the *Prewitt* filters for edge detection.

If p_x and p_y are the grey values produced by applying P_x and P_y to an image, then the magnitude of the gradient is obtained with

$$\sqrt{p_x^2 + p_y^2}.$$

In practice, however, it is more convenient to use either of

$$\max\{|p_x|, |p_y|\}$$

or

$$|p_x| + |p_y|.$$

For example, let us take the image of the integrated circuit shown in figure 8.4, which can be read into MATLAB with

```
>> ic=imread('ic.tif');
```

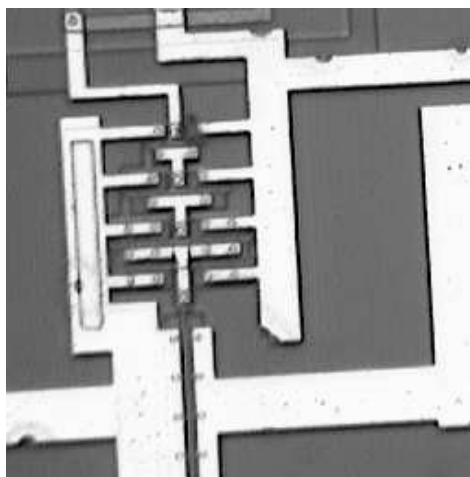


Figure 8.4: An integrated circuit

Applying each of P_x and P_y individually provides the results shown in figure 8.5. Figure 8.5(a) was produced with the following MATLAB commands:

```
>> px=[-1 0 1;-1 0 1;-1 0 1];
>> icx=filter2(px,ic);
>> figure,imshow(icx/255)
```

and figure 8.5(b) with

```
>> py=px';
>> icy=filter2(py,ic);
>> figure,imshow(icy/255)
```

Note that the filter P_x highlights vertical edges, and P_y horizontal edges. We can create a figure containing all the edges with:

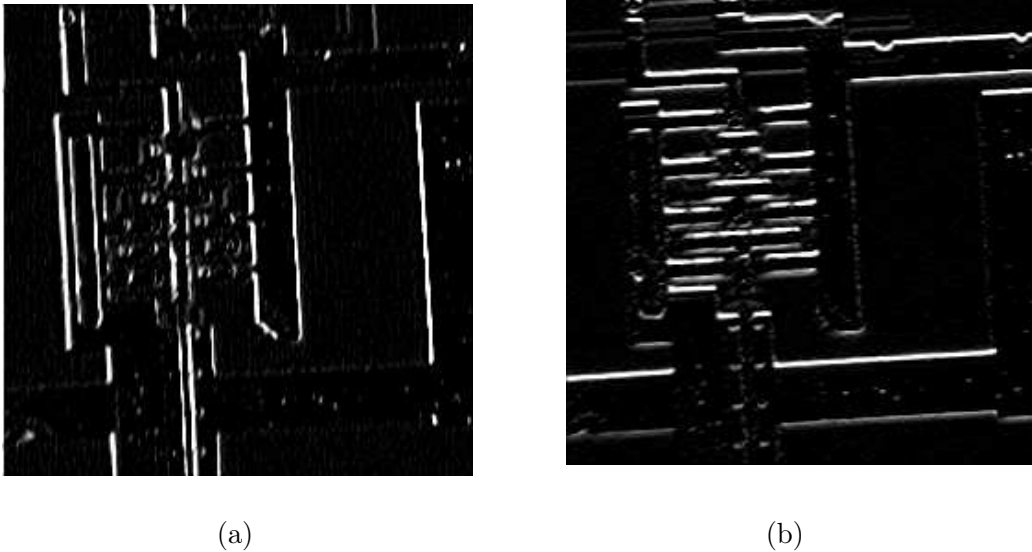


Figure 8.5: The circuit after filtering with the Prewitt filters

```
>> edge_p=sqrt(icx.^2+icy.^2);
>> figure,imshow(edge_p/255)
```

and the result is shown in figure 8.6(a). This is a grey-scale image; a binary image containing edges only can be produced by thresholding. Figure 8.6(b) shows the result after the command

```
>> edge_t=im2bw(edge_p/255,0.3);
```

We can obtain edges by the Prewitt filters directly by using the command

```
>> edge_p=edge(ic,'prewitt');
```

and the `edge` function takes care of all the filtering, and of choosing a suitable threshold level; see its help text for more information. The result is shown in figure 8.7. Note that figures 8.6(b) and 8.7 seem different to each other. This is because the `edge` function does some extra processing over and above taking the square root of the sum of the squares of the filters.

Slightly different edge finding filters are the *Roberts cross-gradient filters*:

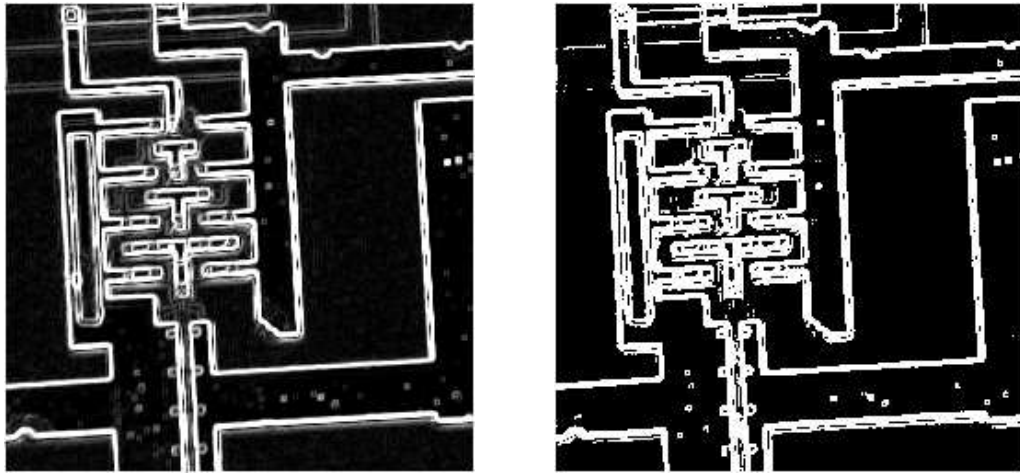
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and the *Sobel filters*:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

The *Sobel filters* are similar to the Prewitt filters, in that they apply a smoothing filter in the opposite direction to the central difference filter. In the Sobel filters, the smoothing takes the form

$$\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$



(a)

(b)

Figure 8.6: All the edges of the circuit

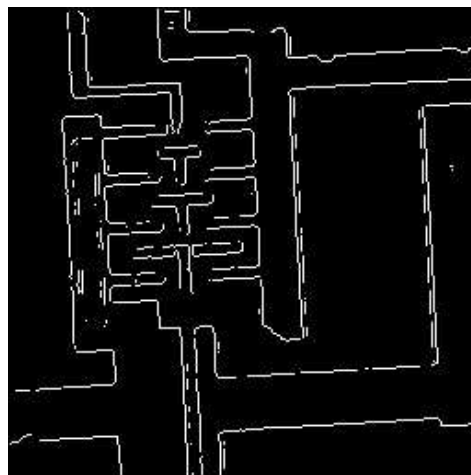


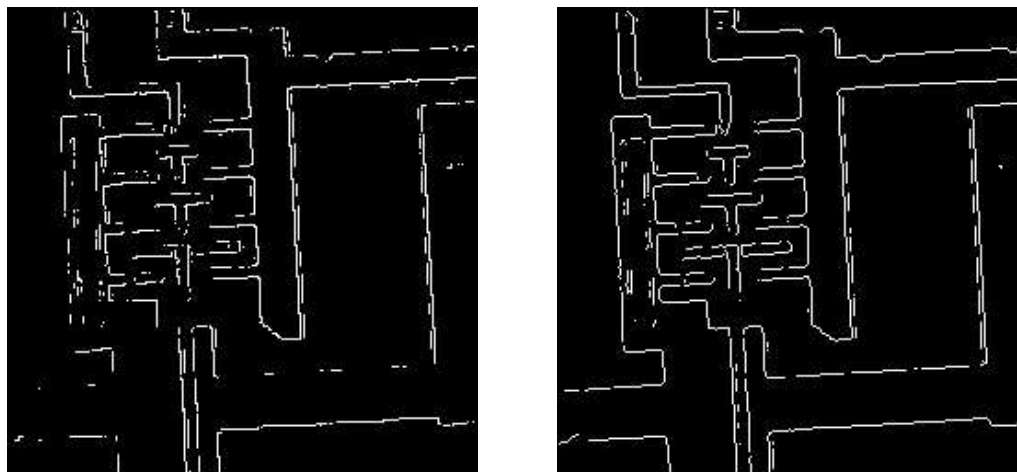
Figure 8.7: The prewitt option of edge

which gives slightly more prominence to the central pixel. Figure 8.8 shows the respective results of the MATLAB commands

```
>> edge_r=edge(ic,'roberts');
>> figure,imshow(edge_r)
```

and

```
>> edge_s=edge(ic,'sobel');
>> figure,imshow(edge_s)
```



(a) Roberts edge detection

(b) Sobel edge detection

Figure 8.8: Results of the Roberts and Sobel filters

The appearance of each of these can be changed by specifying a threshold level.

Of the three filters, the Sobel filters are probably the best; they provide good edges, and they perform reasonably well in the presence of noise.

8.3 Second derivatives

8.3.1 The Laplacian

Another class of edge-detection method is obtained by considering the second derivatives.

The sum of second derivatives in both directions is called the *laplacian*; it is written as

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

and it can be implemented by the filter

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

This is known as a *discrete Laplacian*. The laplacian has the advantage over first derivative methods in that it is an *isotropic filter* [11]; this means it is invariant under rotation. That is, if the laplacian is applied to an image, and the image then rotated, the same result would be obtained if the image were rotated first, and the laplacian applied second. This would appear to make this class of filters ideal for edge detection. However, a major problem with all second derivative filters is that they are very sensitive to noise.

To see how the second derivative affects an edge, take the derivative of the pixel values as plotted in figure 8.2; the results are shown schematically in figure 8.9.

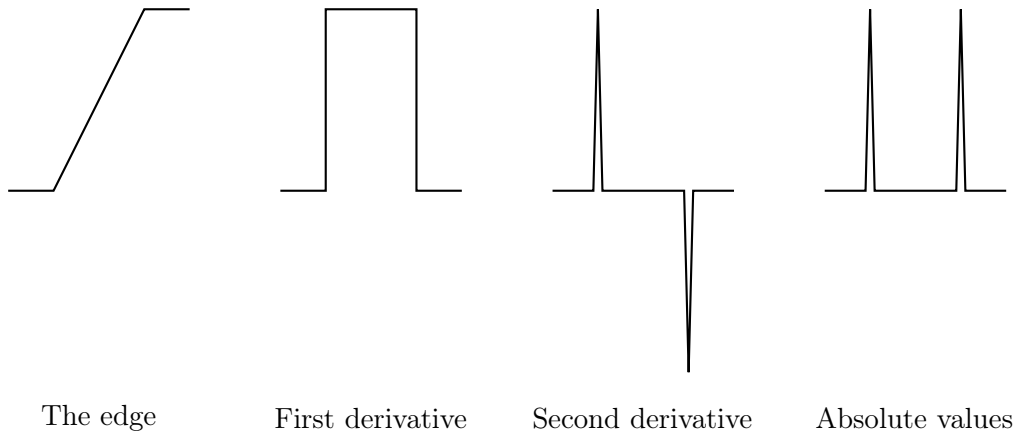


Figure 8.9: Second derivatives of an edge function

The Laplacian (after taking an absolute value, or squaring) gives double edges. To see an example, suppose we enter the MATLAB commands:

```
>> l=fspecial('laplacian',0);
>> ic_l=filter2(l,ic);
>> figure,imshow(mat2gray(ic_l))
```

the result of which is shown in figure 8.10.

Although the result is adequate, it is very messy when compared to the results of the Prewitt and Sobel methods discussed earlier. Other Laplacian masks can be used; some are:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{bmatrix}.$$

In MATLAB, Laplacians of all sorts can be generated using the `fspecial` function, in the form

```
fspecial('laplacian',ALPHA)
```

which produces the Laplacian

$$\frac{1}{\alpha+1} \begin{bmatrix} \alpha & 1-\alpha & \alpha \\ 1-\alpha & -4 & 1-\alpha \\ \alpha & 1-\alpha & \alpha \end{bmatrix}.$$

If the parameter `ALPHA` (which is optional) is omitted, it is assumed to be 0.2. The value 0 gives the Laplacian developed earlier.

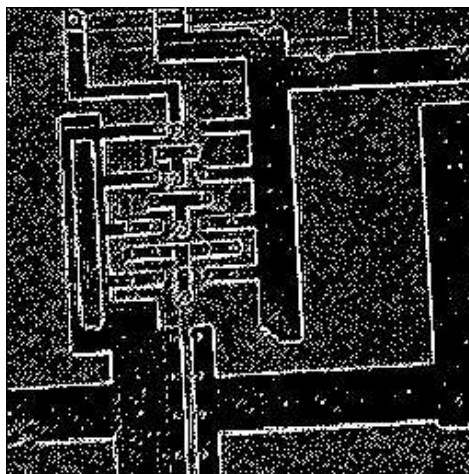


Figure 8.10: Result after filtering with a discrete laplacian

8.3.2 Zero crossings

A more appropriate use for the Laplacian is to find the *position* of edges by locating *zero crossings*. From figure 8.9, the position of the edge is given by the place where the value of the filter takes on a zero value. In general, these are places where the result of the filter changes sign. For example, consider the the simple image given in figure 8.11(a), and the result after filtering with a Laplacian mask in figure 8.11(b).

50	50	50	50	50	50	50	50	50	50
50	50	50	50	50	50	50	50	50	50
50	50	200	200	200	200	200	200	50	50
50	50	200	200	200	200	200	200	50	50
50	50	200	200	200	200	200	200	50	50
50	50	200	200	200	200	200	200	50	50
50	50	50	50	200	200	200	200	50	50
50	50	50	50	200	200	200	200	50	50
50	50	50	50	50	50	50	50	50	50
50	50	50	50	50	50	50	50	50	50

(a) A simple image

-100	-50	-50	-50	-50	-50	-50	-50	-50	-100
-50	0	150	150	150	150	150	150	0	-50
-50	150	-300	-150	-150	-150	-150	-300	150	-50
-50	150	-150	0	0	0	0	-150	150	-50
-50	150	-150	0	0	0	0	-150	150	-50
-50	150	-300	-150	0	0	0	-150	150	-50
-50	0	150	300	-150	0	0	-150	150	-50
-50	0	0	150	-300	-150	-150	-300	150	-50
-50	0	0	0	150	150	150	150	0	-50
-100	-50	-50	-50	-50	-50	-50	-50	-50	-100

(b) After laplace filtering

Figure 8.11: Locating zero crossings in an image

We define the *zero crossings* in such a filtered image to be pixels which satisfy either of the following:

1. they have a negative grey value and are next to (by four-adjacency) a pixel whose grey value is positive,

2. they have a value of zero, and are between negative and positive valued pixels.

To give an indication of the way zero-crossings work, look at the edge plots and their second differences in figure 8.12.

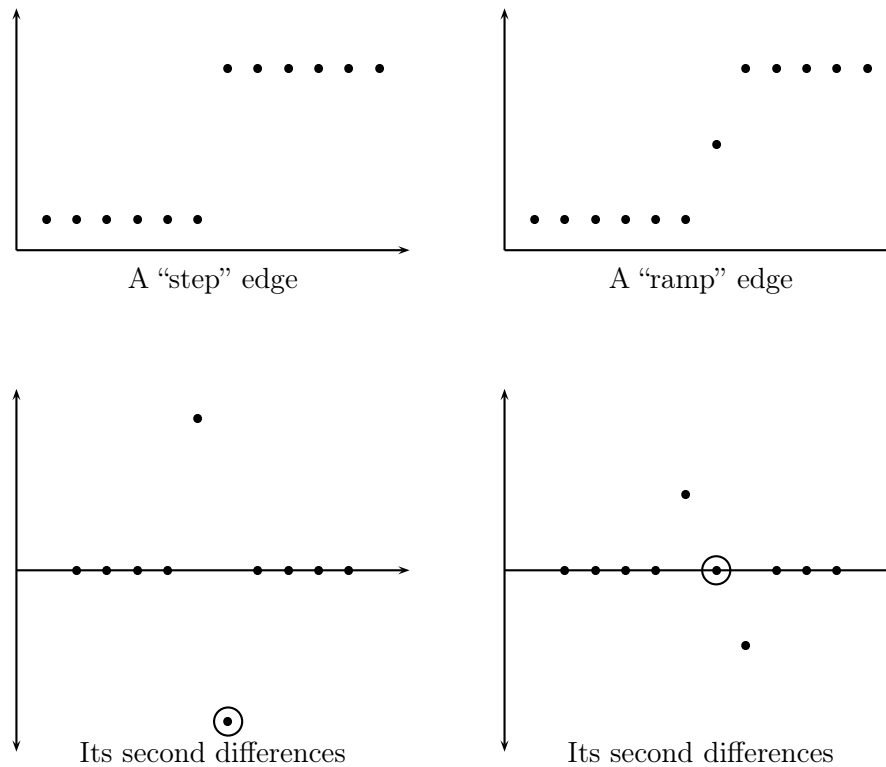


Figure 8.12: Edges and second differences

In each case the zero-crossing is circled. The important point is to note that across any edge there can be only *one* zero-crossing. Thus an image formed from zero-crossings has the potential to be very neat.

In figure 8.11(b) the zero crossings are shaded. We now have a further method of edge detection: take the zero-crossings after a laplace filtering. This is implemented in MATLAB with the `zerocross` option of `edge`, which takes the zero crossings after filtering with a given filter:

```
>> l=fspecial('laplace',0);
>> icz=edge(ic,'zerocross',l);
>> imshow(icz)
```

The result is shown in figure 8.13(a). This is not in fact a very good result—far too many grey level changes have been interpreted as edges by this method. To eliminate them, we may first smooth the image with a Gaussian filter. This leads to the following sequence of steps for edge detection; the *Marr-Hildreth* method:

1. smooth the image with a Gaussian filter,
2. convolve the result with a laplacian,
3. find the zero crossings.

This method was designed to provide an edge detection method to be as close as possible to biological vision. The first two steps can be combined into one, to produce a “Laplacian of Gaussian” or “LoG” filter. These filters can be created with the `fspecial` function. If no extra parameters are provided to the `zerocross` edge option, then the filter is chosen to be the LoG filter found by

```
>> fspecial('log',13,2)
```

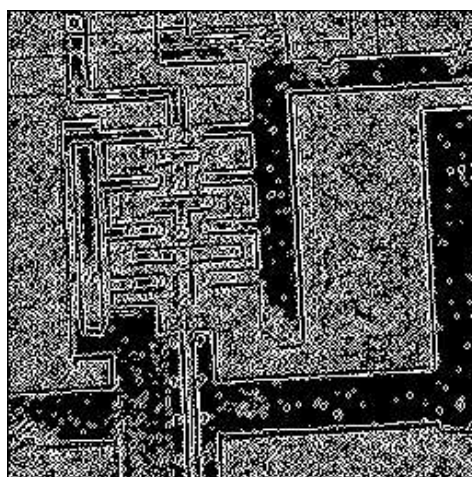
This means that the following command:

```
>> edge(ic,'log');
```

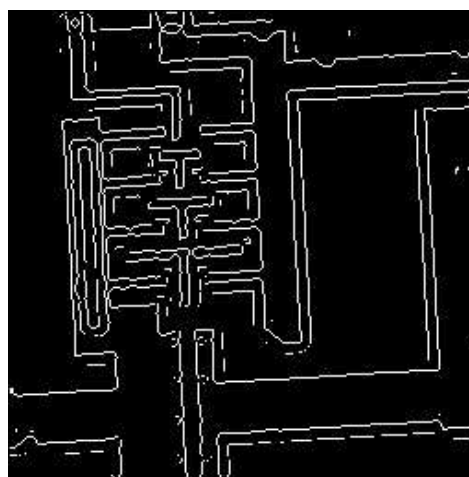
produces exactly the same result as the commands:

```
>> log=fspecial('log',13,2);  
>> edge(ic,'zerocross',log);
```

In fact the `LoG` and `zerocross` options implement the same edge finding method; the difference being that the `zerocross` option allows you to specify your own filter. The result after applying an LoG filter and finding its zero crossings is given in figure 8.13(b).



(a) Zeros crossings



(b) Using an LoG filter first

Figure 8.13: Edge detection using zero crossings

8.4 The Hough transform

If the edge points found by the above edge detection methods are sparse, the resulting edge image may consist of individual points, rather than straight lines or curves. Thus in order to establish a boundary between the regions, it might be necessary to fit a line to those points. This can be a time consuming and computationally inefficient process, especially if there are many such edge points. One way of finding such boundary lines is by use of the “Hough transform”.

The *Hough transform*¹ is designed to find lines in images, but it can be easily varied to find other shapes. The idea is simple. Suppose (x, y) is a point in the image (which we shall assume to be binary). We can write $y = ax + b$, and consider all pairs (a, b) which satisfy this equation, and plot them into an “accumulator array”. The (a, b) array is the “transform array”.

For example, take $(x, y) = (1, 1)$. Since the equation relating a and b is

$$1 = a \cdot 1 + b$$

we can write

$$b = -a + 1.$$

Thus the line $b = -a + 1$ consists of all pairs of points relating to the single point $(1, 1)$. This is shown in figure 8.14.

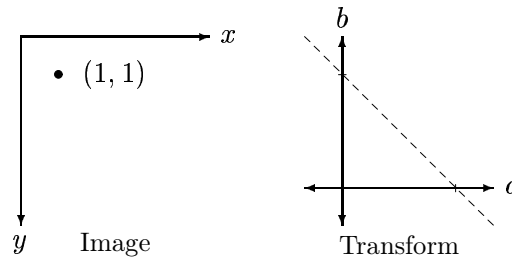


Figure 8.14: A point in an image and its corresponding line in the transform

Each point in the image is mapped onto a line in the transform. The points in the transform corresponding to the greatest number of intersections correspond to the strongest line in the image.

For example, suppose we consider an image with five points: $(1, 0)$, $(1, 1)$, $(2, 1)$, $(4, 1)$ and $(3, 2)$. Each of these points corresponds to a line as follows:

$$\begin{aligned} (1, 0) &\rightarrow b = -a \\ (1, 1) &\rightarrow b = -a + 1 \\ (2, 1) &\rightarrow b = -2a + 1 \\ (4, 1) &\rightarrow b = -4a + 1 \\ (3, 2) &\rightarrow b = -3a + 2. \end{aligned}$$

Each of these lines appears in the transform as shown in figure 8.15.

The dots in the transform indicate places where there are maximum intersections of lines: at each dot three lines intersect. The coordinates of these dots are $(a, b) = (1, 0)$ and $(a, b) = (1, -1)$. These values correspond to the lines

$$y = 1 \cdot x + 0$$

and

$$y = 1 \cdot x + (-1)$$

or $y = x$ and $y = x - 1$. These lines are shown on the image in figure 8.16.

¹“Hough” is pronounced “Huff”.

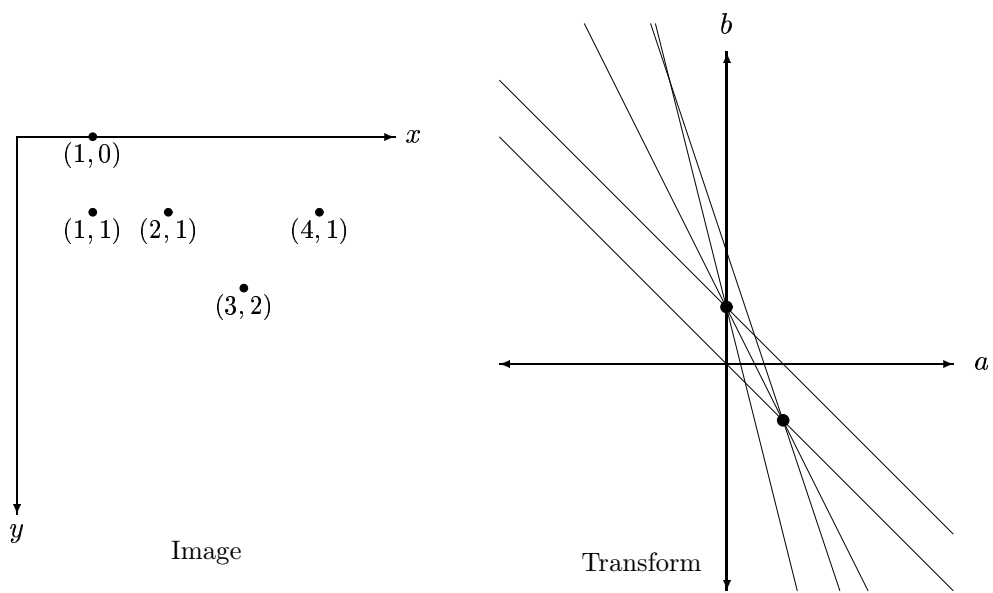


Figure 8.15: An image and its corresponding lines in the transform

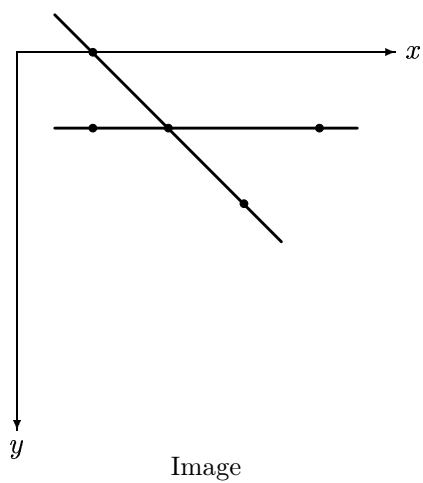


Figure 8.16: Lines found by the Hough transform

These are indeed the “strongest” lines in the image in that they contain the greatest number of points.

There is a problem with this implementation of the Hough transform, and that is that it can’t find vertical lines: we can’t express a vertical line in the form $y = mx + c$, as m represents the gradient, and a vertical line has infinite gradient. We need another parameterization of lines.

Consider a general line, as shown in figure 8.17. Clearly any line can be described in terms of

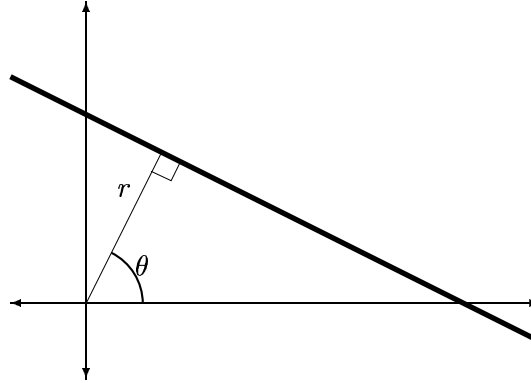


Figure 8.17: A line and its parameters

the two parameters r and θ : r is the perpendicular distance from the line to the origin, and θ is the angle of the line’s perpendicular to the x -axis. In this parameterization, vertical lines are simply those which have $\theta = 0$. If we allow r to have negative values, we can restrict θ to the range

$$-90 < \theta \leq 90.$$

Given this parameterization, we need to be able to find the equation of the line. First note that the point (p, q) where the perpendicular to the line meets the line is $(p, q) = (r \cos \theta, r \sin \theta)$. Also note that the gradient of the perpendicular is $\tan \theta = \sin \theta / \cos \theta$. Now let (x, y) be any point on then line. The gradient of the line is

$$\begin{aligned} \frac{\text{rise}}{\text{run}} &= \frac{y - q}{x - p} \\ &= \frac{y - r \sin \theta}{x - r \cos \theta}. \end{aligned}$$

But since the gradient of the line’s perpendicular is $\tan \theta$, the gradient of the line itself must be

$$-\frac{1}{\tan \theta} = -\frac{\cos \theta}{\sin \theta}.$$

Putting these two expressions for the gradient together produces:

$$\frac{y - r \sin \theta}{x - r \cos \theta} = -\frac{\cos \theta}{\sin \theta}.$$

If we now multiply out these fractions we obtain:

$$y \sin \theta - r \sin^2 \theta = -x \cos \theta + r \cos^2 \theta$$

and this equation can be rewritten as

$$\begin{aligned} y \sin \theta + x \cos \theta &= r \sin^2 \theta + r \cos^2 \theta \\ &= r(\sin^2 \theta + \cos^2 \theta) \\ &= r. \end{aligned}$$

We finally have the required equation for the line as:

$$x \cos \theta + y \sin \theta = r.$$

The Hough transform can then be implemented as follows: we start by choosing a discrete set of values of r and θ to use. For each pixel (x, y) in the image, we compute

$$x \cos \theta + y \sin \theta$$

for each value of θ , and place the result in the appropriate position in the (r, θ) array. At the end, the values of (r, θ) with the highest values in the array will correspond to strongest lines in the image.

An example will clarify this: consider the image shown in figure 8.18.

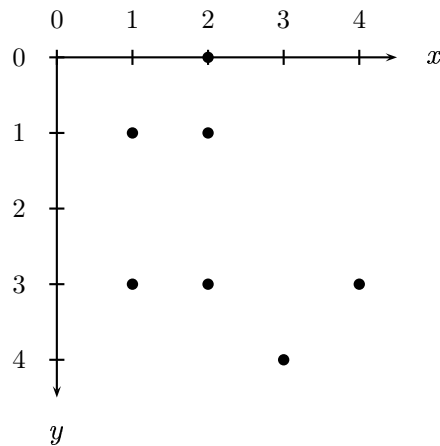


Figure 8.18: A small image

We shall discretize θ to use only the values

$$-45^\circ, \quad 0^\circ, \quad 45^\circ, \quad 90^\circ.$$

We can start by making a table containing all values $x \cos \theta + y \sin \theta$ for each point, and for each value of θ :

(x, y)	-45°	0°	45°	90°
(2, 0)	1.4	2	1.4	0
(1, 1)	0	1	1.4	1
(2, 1)	0.7	2	2.1	1
(1, 3)	-1.4	1	2.8	3
(2, 3)	-0.7	2	3.5	3
(4, 3)	0.7	4	4.9	3
(3, 4)	-0.7	3	4.9	4

The accumulator array contains the number of times each value of (r, θ) appears in the above table:

	-1.4	-0.7	0	0.7	1	1.4	2	2.1	2.8	3	3.5	4	4.9
-45°	1	2	1	2		1							
0°					2		3			1		1	
45°						2		1	1		1		2
90°			1		2					3		2	

In practice this array will be very large, and can be displayed as an image. In this example the two equal largest values occur at $(r, \theta) = (2, 0^\circ)$ and $(r, \theta) = (3, 90^\circ)$. The lines then are

$$x \cos 0 + y \sin 0 = 2$$

or $x = 2$, and

$$x \cos 90 + y \sin 90 = 3$$

or $y = 3$. These lines are shown in figure 8.19

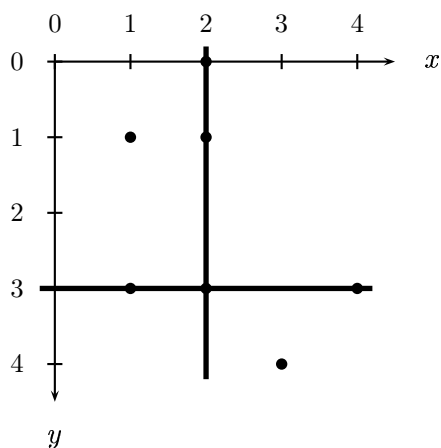


Figure 8.19: Lines found by the Hough transform

Exercises

1. Enter the following matrix into MATLAB:

```

201  195  203  203  199  200  204  190  198  203
201  204  209  197  210  202  205  195  202  199
205  198   46   60   53   37   50   51  194  205
208  203   54   50   51   50   55   48  193  194
200  193   50   56   42   53   55   49  196  211
200  198  203   49   51   60   51  205  207  198
205  196  202   53   52   34   46  202  199  193
199  202  194   47   51   55   48  191  190  197
194  206  198  212  195  196  204  204  199  200
201  189  203  200  191  196  207  203  193  204

```

Chapter 9

Mathematical morphology (1)

9.1 Introduction

Morphology, or *morphology* for short, is a branch of image processing which is particularly useful for analyzing shapes in images. We shall develop basic morphological tools for investigation of binary images, and then show how to extend these tools to greyscale images. MATLAB has many tools for binary morphology in the image processing toolbox; most of which can be used for greyscale morphology as well.

9.2 Basic ideas

The theory of mathematical morphology can be developed in many different ways. We shall adopt one standard method which uses operations on sets of points. A very solid and detailed account can be found in Haralick and Shapiro [5].

Translation

Suppose that A is a set of pixels in a binary image, and $w = (x, y)$ is a particular coordinate point. Then A_w is the set A “translated” in direction (x, y) . That is

$$A_x = \{(a, b) + (x, y) : (a, b) \in A\}.$$

For example, in figure 9.1, A is the cross shaped set, and $w = (2, 2)$. The set A has been shifted in the x and y directions by the values given in w . Note that here we are using matrix coordinates, rather than Cartesian coordinates, so that the origin is at the top left, x goes down and y goes across.

Reflection

If A is set of pixels, then its *reflection*, denoted \hat{A} , is obtained by reflecting A in the origin:

$$\hat{A} = \{(-x, -y) : (x, y) \in A\}.$$

For examples, in figure 9.2, the open and closed circles form sets which are reflections of each other.

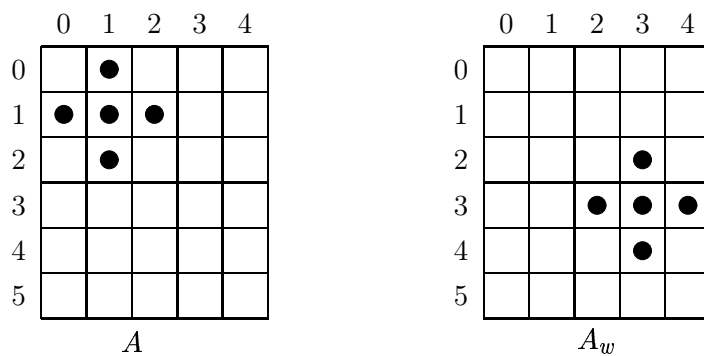


Figure 9.1: Translation

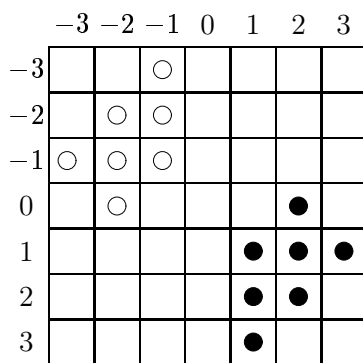


Figure 9.2: Reflection

9.3 Dilation and erosion

These are the basic operations of morphology, in the sense that all other operations are built from a combination of these two.

9.3.1 Dilation

Suppose A and B are sets of pixels. Then the *dilation of A by B* , denoted $A \oplus B$, is defined as

$$A \oplus B = \bigcup_{x \in B} A_x.$$

What this means is that for every point $x \in B$, we translate A by those coordinates. Then we take the union of all these translations.

An equivalent definition is that

$$A \oplus B = \{(x, y) + (u, v) : (x, y) \in A, (u, v) \in B\}.$$

From this last definition, dilation is shown to be commutative; that

$$A \oplus B = B \oplus A.$$

An example of a dilation is given in figure 9.3. In the translation diagrams, the grey squares show the original position of the object. Note that $A_{(0,0)}$ is of course just A itself. In this example, we have

$$B = \{(0, 0), (1, 1), (-1, 1), (1, -1), (-1, -1)\}$$

and those these are the coordinates by which we translate A .

In general, $A \oplus B$ can be obtained by replacing every point (x, y) in A with a copy of B , placing the $(0, 0)$ point of B at (x, y) . Equivalently, we can replace every point (u, v) of B with a copy of A .

Dilation is also known as *Minkowski addition*; see Haralick and Shapiro [5] for more information.

As you see in figure 9.3, dilation has the effect of increasing the size of an object. However, it is not necessarily true that the original object A will lie within its dilation $A \oplus B$. Depending on the coordinates of B , $A \oplus B$ may end up quite a long way from A . Figure 9.4 gives an example of this: A is the same as in figure 9.3; B has the same shape but a different position. In this figure, we have

$$B = \{(7, 3), (6, 2), (6, 4), (8, 2), (8, 4)\}$$

so that

$$A \oplus B = A_{(7,3)} \cup A_{(6,2)} \cup A_{(6,4)} \cup A_{(8,2)} \cup A_{(8,4)}.$$

For dilation, we generally assume that A is the image being processed, and B is a small set of pixels. In this case B is referred to as a *structuring element* or as a *kernel*.

Dilation in MATLAB is performed with the command

```
>> imdilate(image, kernel)
```

To see an example of dilation, consider the commands:

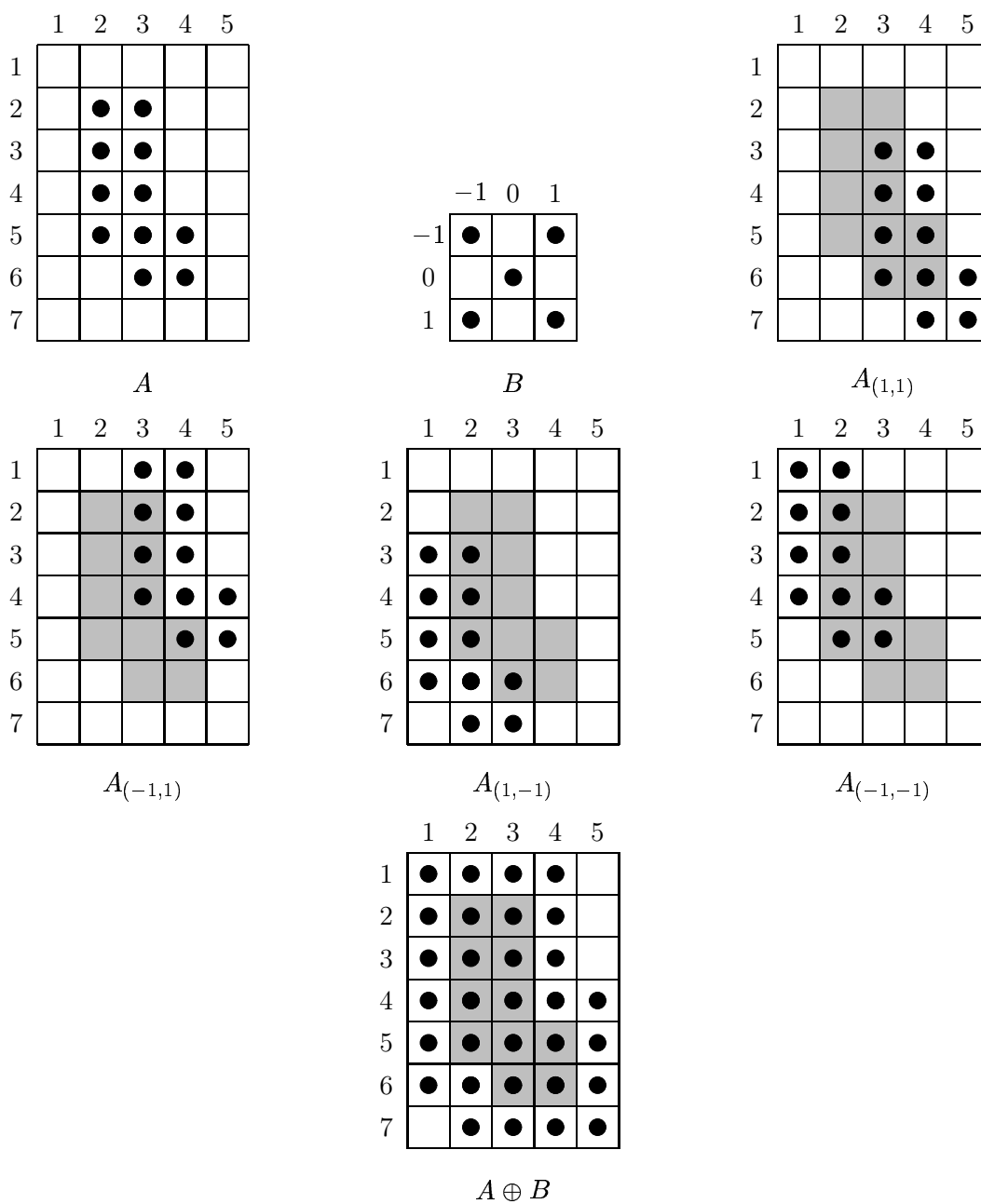
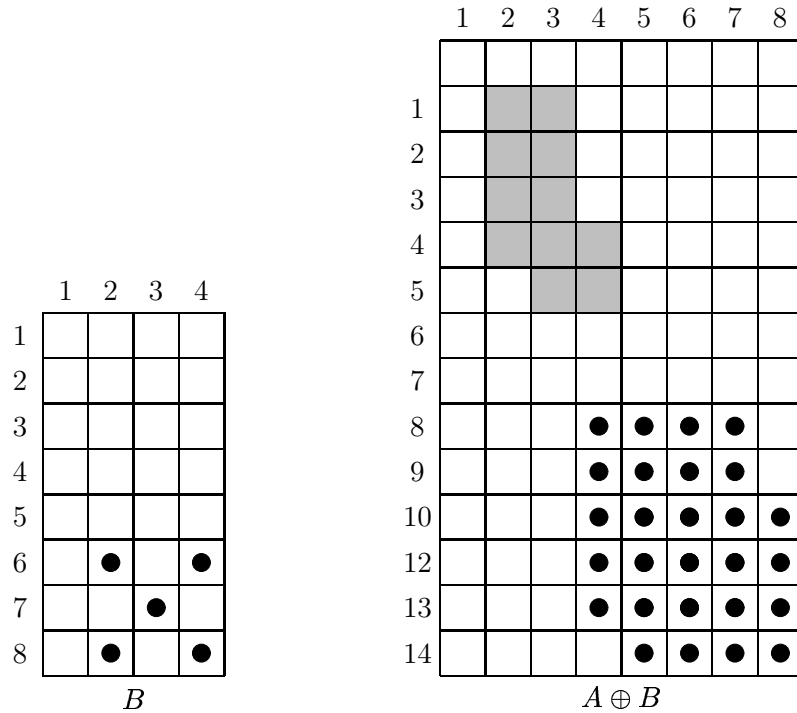


Figure 9.3: Dilation

Figure 9.4: A dilation for which $A \not\subseteq A \oplus B$

```
>> t=imread('text.tif');
>> sq=ones(3,3);
>> td=imdilate(t,sq);
>> subplot(1,2,1),imshow(t)
>> subplot(1,2,2),imshow(td)
```

The result is shown in figure 9.5. Notice how the image has been “thickened”. This is really what dilation does; hence its name.

9.3.2 Erosion

Given sets A and B , the *erosion of A by B* , written $A \ominus B$, is defined as:

$$A \ominus B = \{w : B_w \subseteq A\}.$$

In other words the erosion of A by B consists of all points $w = (x, y)$ for which B_w is in A . To perform an erosion, we can move B over A , and find all the places it will fit, and for each such place mark down the corresponding $(0, 0)$ point of B . The set of all such points will form the erosion.

An example of erosion is given in figures 9.6.

Note that in the example, the erosion $A \ominus B$ was a subset of A . This is not necessarily the case; it depends on the position of the origin in B . If B contains the origin (as it did in figure 9.6), then the erosion will be a subset of the original object.

Figure 9.7 shows an example where B does not contain the origin. In this figure, the open circles in the right hand figure form the erosion.



Figure 9.5: Dilation of a binary image

Note that in figure 9.7, the *shape* of the erosion is the same as that in figure 9.6; however its *position* is different. Since the origin of B in figure 9.7 is translated by $(-4, -3)$ from its position in figure 9.6, we can assume that the erosion will be translated by the same amount. And if we compare figures 9.6 and 9.7, we can see that the second erosion has indeed been shifted by $(-4, -3)$ from the first.

For erosion, as for dilation, we generally assume that A is the image being processed, and B is a small set of pixels: the structuring element or kernel.

Erosion is related to *Minkowski subtraction*: the Minkowski subtraction of B from A is defined as

$$A - B = \bigcap_{b \in B} A_b.$$

Erosion in MATLAB is performed with the command

```
>> imerode(image, kernel)
```

We shall give an example; using a different binary image:

```
>> c=imread('circbw.tif');
>> ce=imerode(c,sq);
>> subplot(1,2,1),imshow(c)
>> subplot(1,2,2),imshow(ce)
```

The result is shown in figure 9.8. Notice how the image has been “thinned”. This is the expected result of an erosion; hence its name. If we kept on eroding the image, we would end up with a completely black result.

Relationship between erosion and dilation

It can be shown that erosion and dilation are “inverses” of each other; more precisely, the complement of an erosion is equal to the dilation of the complement. Thus:

$$\overline{A \ominus B} = \overline{A} \oplus \hat{B}.$$

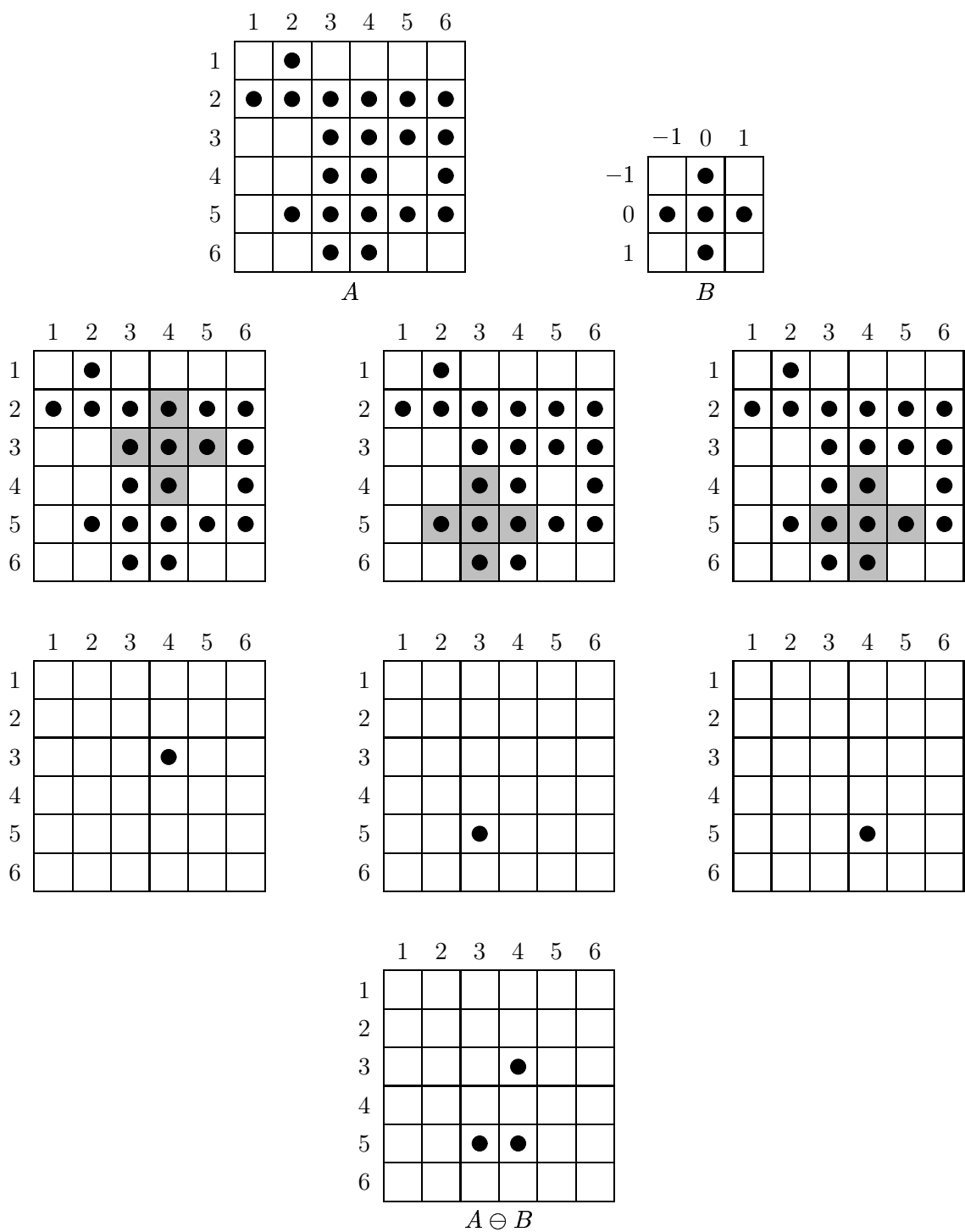


Figure 9.6: Erosion with a cross-shaped structuring element

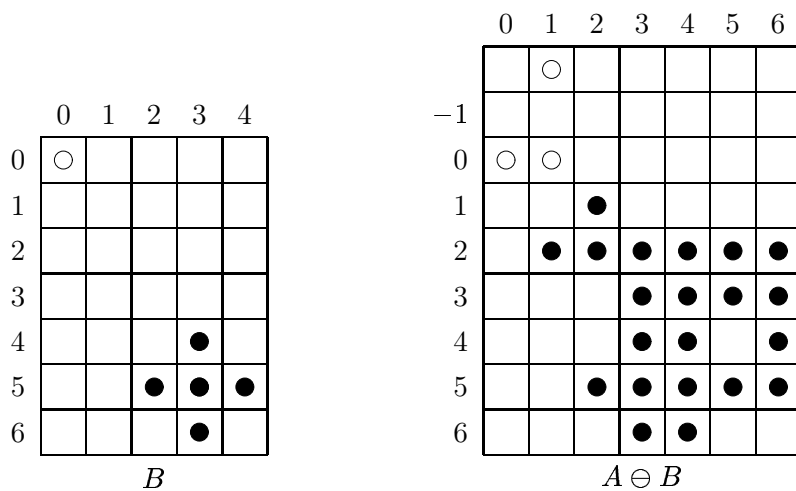


Figure 9.7: Erosion with a structuring element not containing the origin

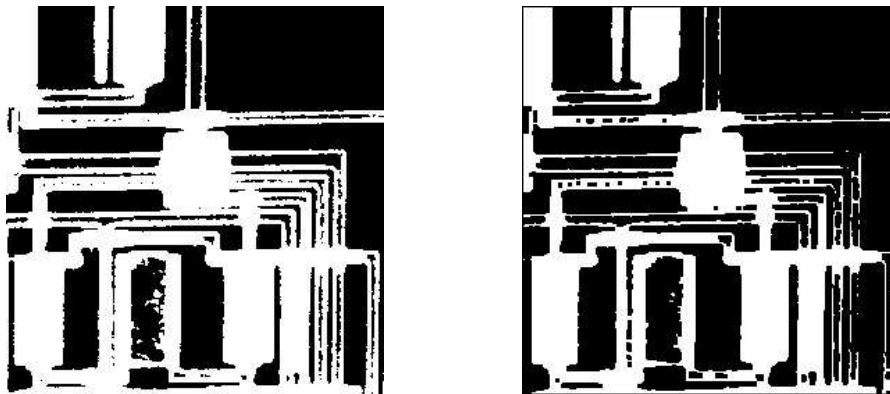


Figure 9.8: Erosion of a binary image

A proof of this can be found in Haralick and Shapiro [5].

It can be similarly shown that the same relationship holds if erosion and dilation are interchanged; that

$$\overline{A \oplus B} = \overline{A} \ominus \hat{B}.$$

We can demonstrate the truth of these using MATLAB commands; all we need to know is that the complement of a binary image

`b`

is obtained using

```
>> ~b
```

and that given two images `a` and `b`; their equality is determined with

```
>> all(a(:)==b(:))
```

To demonstrate the equality

$$\overline{A \ominus B} = \overline{A} \oplus \hat{B}.$$

pick a binary image, say the text image, and a structuring element. Then the left hand side of this equation is produced with

```
>> lhs=~imerode(t,sq);
```

and the right hand side with

```
>> rhs=imdilate(~t,sq);
```

Finally, the command

```
>> all(lhs(:)==rhs(:))
```

should return 1, for true.

9.3.3 An application: boundary detection

If A is an image, and B a small structuring element consisting of point symmetrically places about the origin, then we can define the boundary of A by any of the following methods:

- (i) $A - (A \ominus B)$ “internal boundary”
- (ii) $(A \oplus B) - A$ “external boundary”
- (iii) $(A \oplus B) - (A \ominus B)$ “morphological gradient”

In each definition the minus refers to set difference. For some examples, see figure 9.9. Note that the internal boundary consists of those pixels in A which are at its edge; the external boundary consists of pixels outside A which are just next to it, and that the morphological gradient is a combination of both the internal and external boundaries.

To see some examples, choose the image `rice.tif`, and threshold it to obtain a binary image:

```
>> rice=imread('rice.tif');
>> r=rice>110;
```

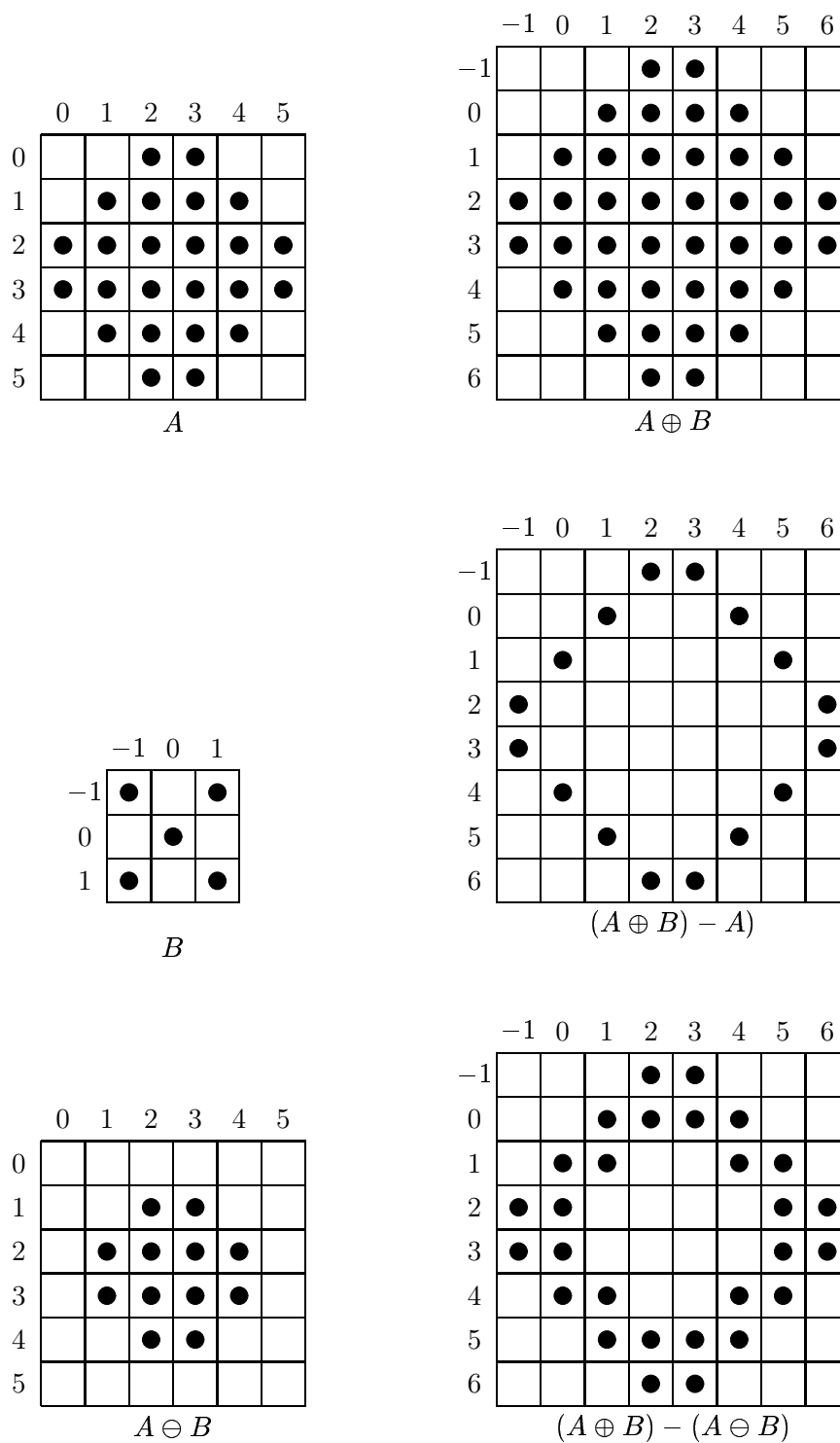


Figure 9.9: Boundaries

Then the internal boundary is obtained with:

```
>> re=imerode(r,sq);
>> r_int=r&~re;
>> subplot(1,2,1),imshow(r)
>> subplot(1,2,2),imshow(r_int)
```

The result is shown in figure 9.10.

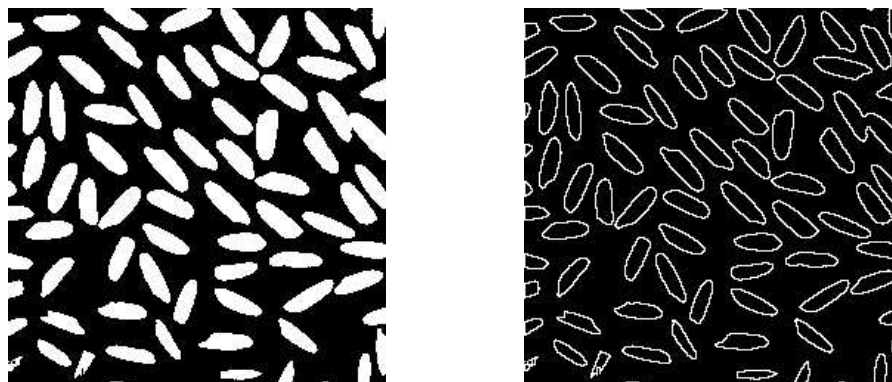


Figure 9.10: “Internal boundary” of a binary image

The external boundary and morphological gradients can be obtained similarly:

```
>> rd=imdilate(r,sq);
>> r_ext=rd&~r;
>> r_grad=rd&~re;
>> subplot(1,2,1),imshow(r_ext)
>> subplot(1,2,2),imshow(r_grad)
```

The results are shown in figure 9.11.

Note that the external boundaries are larger than the internal boundaries. This is because the internal boundaries show the outer edge of the image components; whereas the external boundaries show the pixels just outside the components. The morphological gradient is thicker than either, and is in fact the union of both.

Exercises

1. For each of the following images A and structuring elements B :

$A =$

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 0 1 1 0
0 1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 1 1 0
0 1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 1 1 0
```

Chapter 10

Mathematical morphology (2)

10.1 Opening and closing

These operations may be considered as “second level” operations; in that they build on the basic operations of dilation and erosion. They are also, as we shall see, better behaved mathematically.

10.1.1 Opening

Given A and a structuring element B , the *opening of A by B* , denoted $A \circ B$, is defined as:

$$A \circ B = (A \ominus B) \oplus B.$$

So an opening consists of an erosion followed by a dilation. An equivalent definition is

$$A \circ B = \cup \{B_w : B_w \subseteq A\}.$$

That is, $A \circ B$ is the union of all translations of B which fit inside A . Note the difference with erosion: the erosion consists only of the $(0,0)$ point of B for those translations which fit inside A ; the opening consists of all of B . An example of opening is given in figure 10.1.

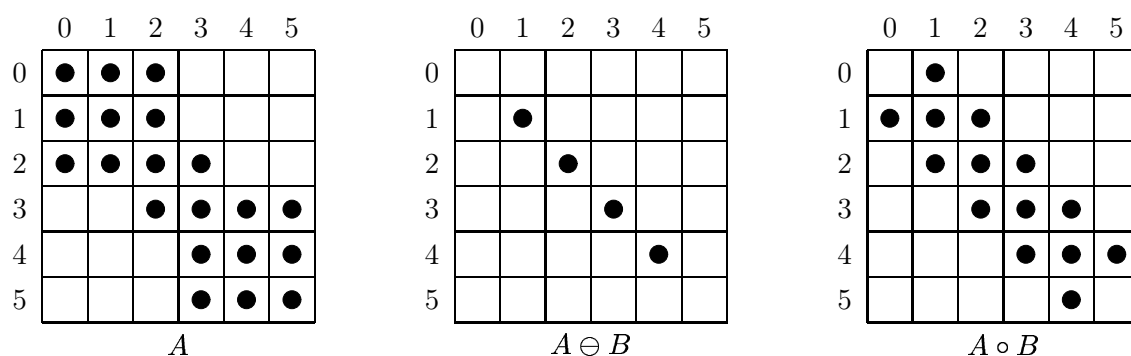


Figure 10.1: Opening

The opening operation satisfies the following properties:

1. $(A \circ B) \subseteq A$. Note that this is not the case with erosion; as we have seen, an erosion may not necessarily be a subset.

2. $(A \circ B) \circ B = A \circ B$. That is, an opening can never be done more than once. This property is called *idempotence*. Again, this is not the case with erosion; you can keep on applying a sequence of erosions to an image until nothing is left.
3. If $A \subseteq C$, then $(A \circ B) \subseteq (C \circ B)$.
4. Opening tends to “smooth” an image, to break narrow joins, and to remove thin protrusions.

10.1.2 Closing

Analogous to opening we can define *closing*, which may be considered as a dilation followed by an erosion, and is denoted $A \bullet B$:

$$A \bullet B = (A \oplus B) \ominus B.$$

Another definition of closing is that $x \in A \bullet B$ if *all* translations B_w which contain x have non-empty intersections with A . An example of closing is given in figure 10.2. The closing operation satisfies

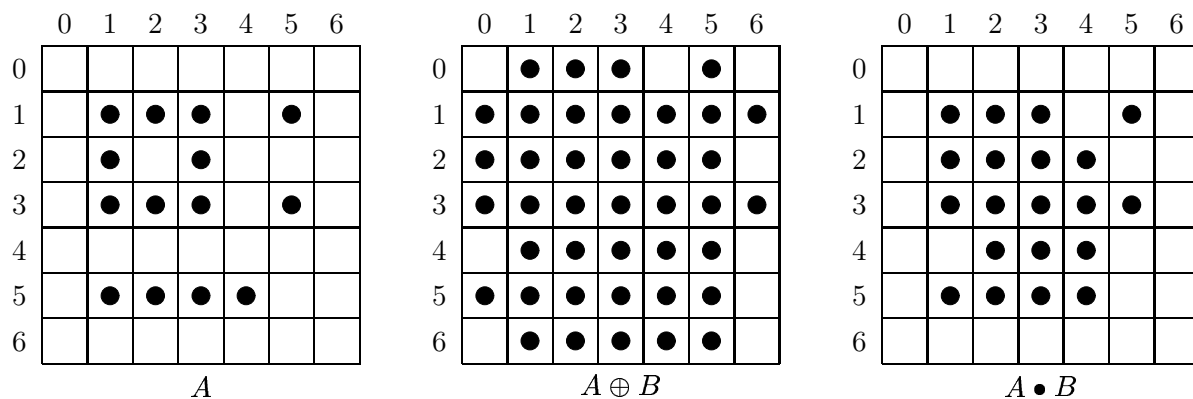


Figure 10.2: Closing

the following properties:

1. $A \subseteq (A \bullet B)$.
2. $(A \bullet B) \bullet B = A \bullet B$; that is, closing, like opening, is idempotent.
3. If $A \subseteq C$, then $(A \bullet B) \subseteq (C \bullet B)$.
4. Closing tends also to smooth an image, but it fuses narrow breaks and thin gulfs, and eliminates small holes.

Opening and closing are implemented by the `imopen` and `imclose` functions respectively. We can see the effects on a simple image using the square and cross structuring elements.

```
>> cr=[0 1 0;1 1 1;0 1 0];
>> test=zeros(10,10);test(2:6,2:4)=1;test(3:5,6:9)=1;test(8:9,4:8)=1;test(4,5)=1
test =
```

```

0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0
0 1 1 1 0 1 1 1 1 0
0 1 1 1 1 1 1 1 1 0
0 1 1 1 0 1 1 1 1 0
0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 0 0
0 0 0 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0

```

```
>> imopen(test,sq)
```

```
ans =
```

```

0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0
0 1 1 1 0 1 1 1 1 0
0 1 1 1 0 1 1 1 1 0
0 1 1 1 0 1 1 1 1 0
0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

```

```
>> imopen(test,cr)
```

```
ans =
```

```

0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 1 1 1 0 1 1 1 0 0
0 1 1 1 1 1 1 1 1 0
0 1 1 1 0 1 1 1 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

```

Note that in each case the image has been separated into distinct components, and the lower part has been removed completely.

```
>> imclose(test,sq)
```

```
ans =
```

```

1  1  1  1  0  0  0  0  0  0
1  1  1  1  0  0  0  0  0  0
1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  0  0

>> imclose(test,cr)

ans =

0  0  1  0  0  0  0  0  0  0
0  1  1  1  0  0  0  0  0  0
1  1  1  1  1  1  1  1  1  0
1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  0
0  1  1  1  1  1  1  1  0  0
0  0  1  1  1  1  1  0  0  0
0  0  0  1  1  1  1  1  0  0
0  0  0  1  1  1  1  1  0  0
0  0  0  0  1  1  1  0  0  0

```

With closing, the image is now fully “joined up”. We can obtain a joining-up effect with the text image, using a diagonal structuring element.

```

>> diag=[0 0 1;0 1 0;1 0 0]

diag =

0  0  1
0  1  0
1  0  0

>> tc=imclose(t,diag);
>> imshow(tc)

```

The result is shown in figure 10.3.

An application: noise removal

Suppose A is a binary image corrupted by impulse noise—some of the black pixels are white, and some of the white pixels are black. An example is given in figure 10.4. Then $A \ominus B$ will remove the single black pixels, but will enlarge the holes. We can fill the holes by dilating twice:

$$((A \ominus B) \oplus B) \oplus B.$$

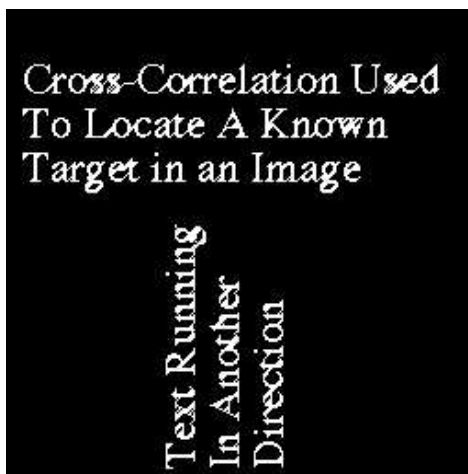


Figure 10.3: An example of closing

The first dilation returns the holes to their original size; the second dilation removes them. But this will enlarge the objects in the image. To reduce them to their correct size, perform a final erosion:

$$(((A \ominus B) \oplus B) \oplus B) \ominus B.$$

The inner two operations constitute an opening; the outer two operations a closing. Thus this noise removal method is in fact an opening followed by a closing:

$$(A \circ B) \bullet B).$$

This is called *morphological filtering*.

Suppose we take an image and apply 10% shot noise to it:

```
>> c=imread('circles.tif');
>> x=rand(size(c));
>> d1=find(x<=0.05);
>> d2=find(x>=0.95);
>> c(d1)=0;
>> c(d2)=1;
>> imshow(c)
```

The result is shown as figure 10.4(a). The filtering process can be implemented with

```
>> cf1=imclose(imopen(c,sq),sq);
>> figure,imshow(cf1)
>> cf2=imclose(imopen(c,cr),cr);
>> figure,imshow(cf2)
```

and the results are shown as figures 10.4(b) and (c). The results are rather “blocky”; although less so with the cross structuring element.

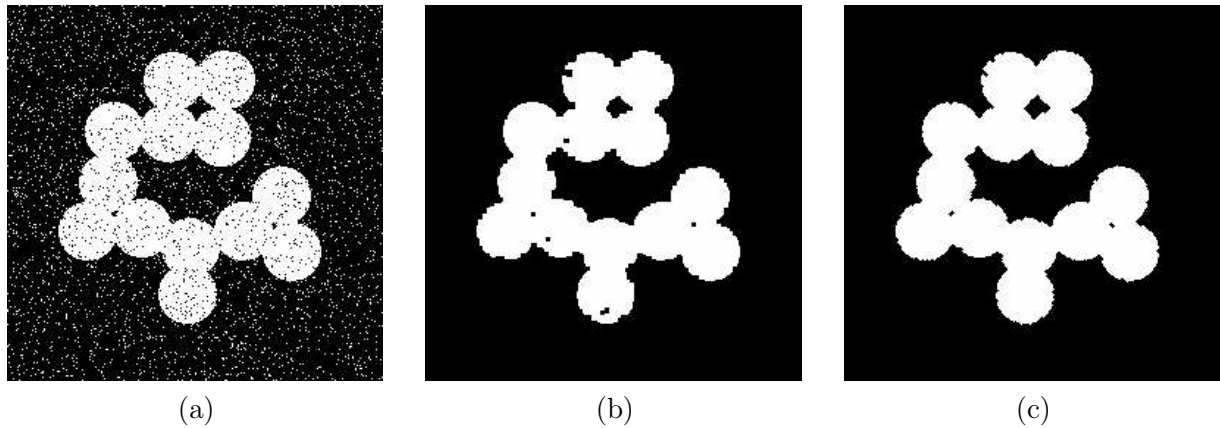


Figure 10.4: A noisy binary image and results after morphological filtering with different structuring elements.

Relationship between opening and closing

Opening and closing share a relationship very similar to that of erosion and dilation: the complement of an opening is equal to the closing of a complement, and complement of a closing is equal to the opening of a complement. Specifically:

$$\overline{A \bullet B} = \overline{A} \circ \hat{B}$$

and

$$\overline{A \circ B} = \overline{A} \bullet \hat{B}.$$

Again see Haralick and Shapiro [5] for a formal proof.

10.2 The hit-or-miss transform

This is a powerful method for finding shapes in images. As with all other morphological algorithms, it can be defined entirely in terms of dilation and erosion; in this case, erosion only.

Suppose we wish to locate 3×3 square shapes, such as is in the centre of the image A in figure 10.5.

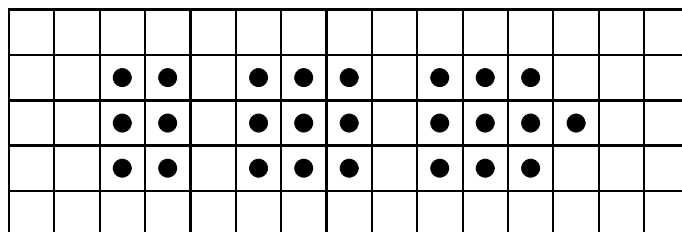


Figure 10.5: An image A containing a shape to be found

If we performed an erosion $A \ominus B$ with B being the square structuring element, we would obtain the result given in figure 10.6.

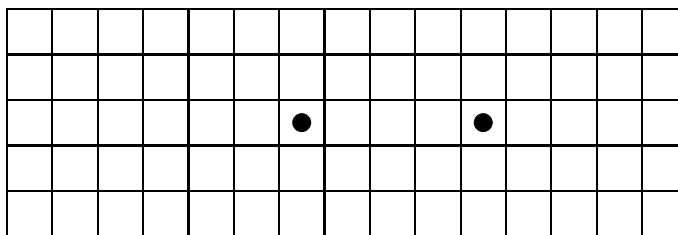


Figure 10.6: The erosion $A \ominus B$

The result contains two pixels, as there are exactly two places in A where B will fit. Now suppose we also erode the complement of A with a structuring element C which fits exactly around the 3×3 square; \bar{A} and C are shown in figure 10.7. (We assume that $(0, 0)$ is at the centre of C .)

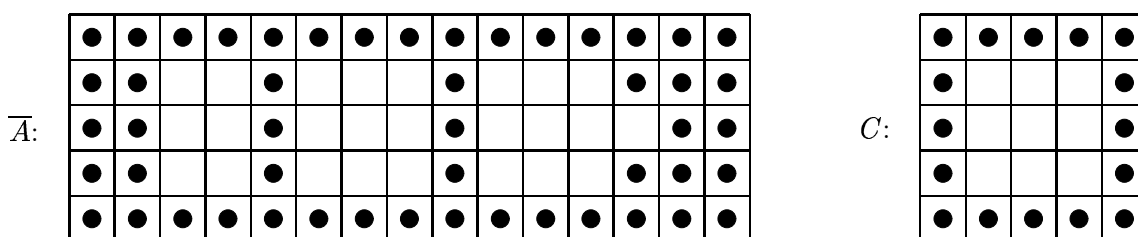


Figure 10.7: The complement and the second structuring element

If we now perform the erosion $\bar{A} \ominus C$ we would obtain the result shown in figure 10.8.

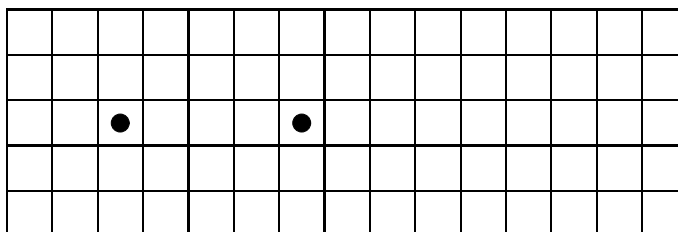


Figure 10.8: The erosion $\bar{A} \ominus C$

The intersection of the two erosion operations would produce just one pixel at the position of the centre of the 3×3 square in A , which is just what we want. If A had contained more than one square, the final result would have been single pixels at the positions of the centres of each. This combination of erosions forms the hit-or-miss transform.

In general, if we are looking for a particular shape in an image, we design two structuring elements: B_1 which is the same shape, and B_2 which fits around the shape. We then write $B = (B_1, B_2)$ and

$$A \circledast B = (A \ominus B_1) \cap (\bar{A} \ominus B_2)$$

for the hit-or-miss transform.

As an example, we shall attempt to find the hyphen in “Cross-Correlation” in the text image shown in figure 9.5. This is in fact a line of pixels of length six. We thus can create our two structuring elements as:

```
>> b1=ones(1,6);
>> b2=[1 1 1 1 1 1 1;1 0 0 0 0 0 0 1; 1 1 1 1 1 1 1 1];
>> tb1=erode(t,b1);
>> tb2=erode(~t,b2);
>> hit_or_miss=tb1&tb2;
>> [x,y]=find(hit_or_miss==1)
```

and this returns a coordinate of (41, 76), which is right in the middle of the hyphen. Note that the command

```
>> tb1=erode(t,b1);
```

is not sufficient, as there are quite a few lines of length six in this image. We can see this by viewing the image `tb1`, which is given in figure 10.9.

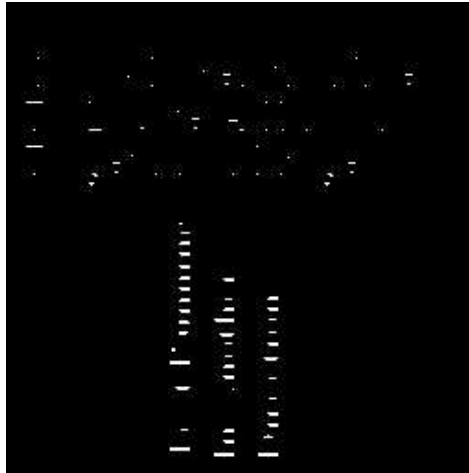


Figure 10.9: Text eroded by a hyphen-shaped structuring element

10.3 Some morphological algorithms

In this section we shall investigate some simple algorithms which use some of the morphological techniques we have discussed in previous sections.

10.3.1 Region filling

Suppose in an image we have a region bounded by an 8-connected boundary, as shown in figure 10.10. Given a pixel p within the region, we wish to fill up the entire region. To do this, we start with p , and dilate as many times as necessary with the cross-shaped structuring element B (as used in

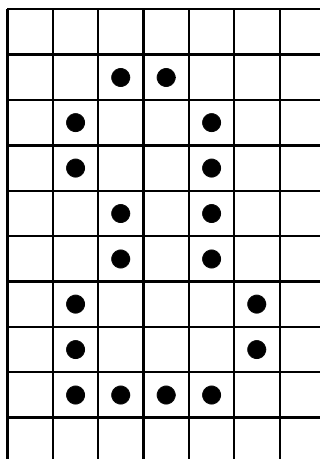


Figure 10.10: An 8-connected boundary of a region to be filled

figure 9.6), each time taking an intersection with \bar{A} before continuing. We thus create a sequence of sets:

$$\{p\} = X_0, X_1, X_2, \dots, X_k = X_{k+1}$$

for which

$$X_n = (X_{n-1} \oplus B) \cap \bar{A}.$$

Finally $X_k \cup A$ is the filled region. Figure 10.11 shows how this is done.

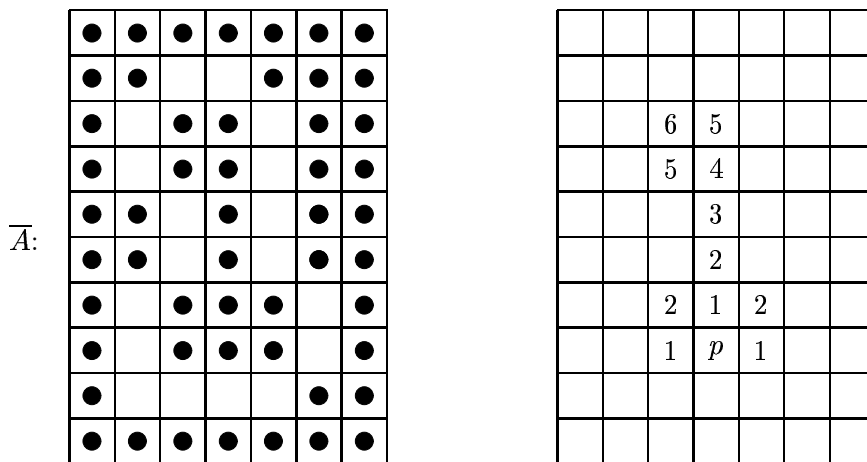


Figure 10.11: The process of filling a region

In the right hand grid, we have

$$X_0 = \{p\}, X_1 = \{p, 1\}, X_2 = \{p, 1, 2\}, \dots$$

Note that the use of the cross-shaped structuring element means that we never cross the boundary.

10.3.2 Connected components

We use a very similar algorithm to fill a connected component; we use the cross-shaped structuring element for 4-connected components, and the square structuring element for 8-connected components. Starting with a pixel p , we fill up the rest of the component by creating a sequence of sets

$$X_0 = \{p\}, X_1, X_2, \dots$$

such that

$$X_n = (X_{n-1} \oplus B) \cap A$$

until $X_k = X_{k-1}$. Figure 10.12 shows an example.

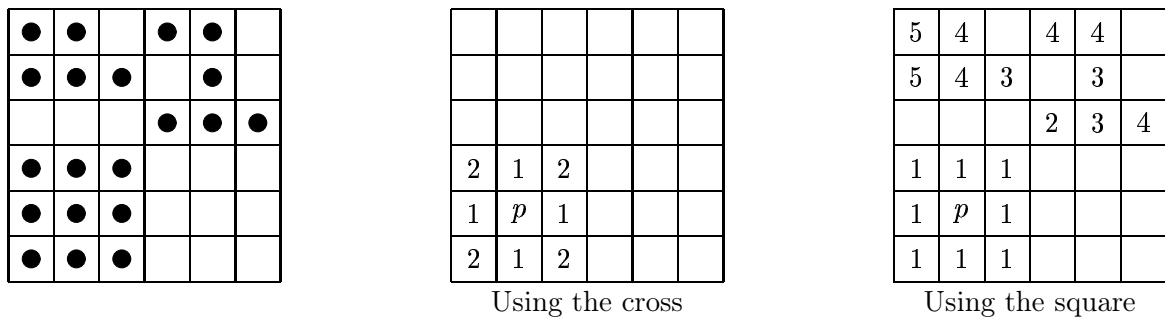


Figure 10.12: Filling connected components

In each case we are starting in the centre of the square in the lower left. As this square is itself a 4-connected component, the cross structuring element cannot go beyond it.

Both of these algorithms can be very easily implemented by MATLAB functions. To implement region filling, we keep track of two images: `current` and `previous`, and stop when there is no difference between them. We start with `previous` being the single point p in the region, and `current` the dilation $(p \oplus B) \cap \bar{A}$. At the next step we set

$$\begin{aligned} \text{previous} &\leftarrow \text{current}, \\ \text{current} &\leftarrow (\text{current} \oplus B) \cap \bar{A}. \end{aligned}$$

Given B , we can implement the last step in MATLAB by

```
imdilate(current,B)&~A.
```

The function is shown in figure 10.13 We can use this to fill a particular region delineated by a boundary.

```
>> n=imread('nicework.tif');
>> imshow(n),pixval on
>> nb=n&~imerode(n,sq);
>> figure,imshow(nb)
>> nf=regfill(nb,[74,52],sq);
>> figure,imshow(nf)
```

```

function out=regfill(im,pos,kernel)
% REGFILL(IM,POS,KERNEL) performs region filling of binary image IMAGE,
% with kernel KERNEL, starting at point with coordinates given by POS.
%
% Example:
%         n=imread('nicework.tif');
%         nb=n&~imerode(n,ones(3,3));
%         nr=regfill(nb,[74,52],ones(3,3));
%
current=zeros(size(im));
last=zeros(size(im));
last(pos(1),pos(2))=1;
current=imdilate(last,kernel)&~im;
while any(current(:)~=last(:)),
    last=current;
    current=imdilate(last,kernel)&~im;
end;
out=current;

```

Figure 10.13: A simple program for filling regions

The results are shown in figure 10.14. Image (a) is the original; (b) the boundary, and (c) the result of a region fill. Figure (d) shows a variation on the region filling, we just include all boundaries. This was obtained with

```
>> figure,imshow(nf|nb)
```

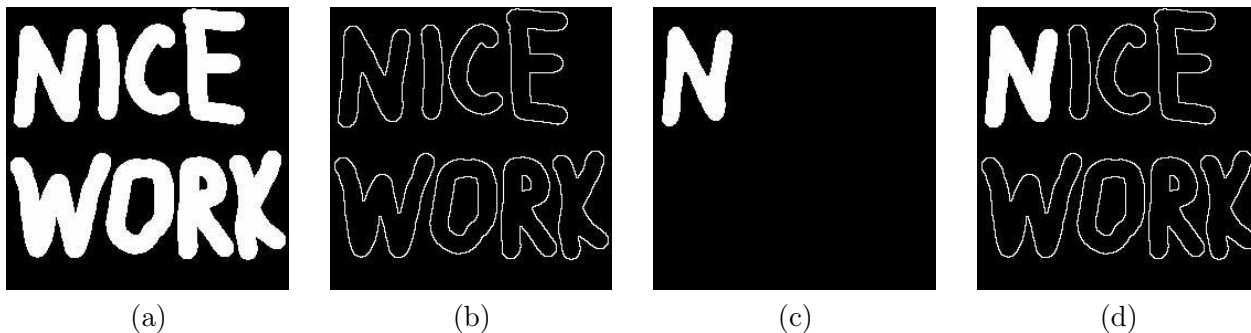


Figure 10.14: Region filling

The function for connected components is almost exactly the same as that for region filling, except that whereas for region filling we took an intersection with the *complement* of our image, for connected components we take the intersection with the image itself. Thus we need only change one line, and the resulting function is shown in 10.15 We can experiment with this function with the “nice work” image. We shall use the square structuring element, and also a larger structuring element of size 11×11 .

```

>> sq2=ones(11,11);
>> nc=components(n,[57,97],sq);

```

```

function out=components(im,pos,kernel)
% COMPONENTS(IM,POS,KERNEL) produces the connected component of binary image
% IMAGE which includes the point with coordinates given by POS, using
% kernel KERNEL.
%
% Example:
%         n=imread('nicework.tif');
%         nc=components(nb,[74,52],ones(3,3));
%
current=zeros(size(im));
last=zeros(size(im));
last(pos(1),pos(2))=1;
current=imdilate(last,kernel)&im;
while any(current(:)~=last(:)),
    last=current;
    current=imdilate(last,kernel)&im;
end;
out=current;

```

Figure 10.15: A simple program for connected components

```

>> imshow(nc)
>> nc2=components(n,[57,97],sq2);
>> figure,imshow(nc2)

```

and the results are shown in figure 10.16. Image (a) uses the 3×3 square; image (b) uses the 11×11 square.

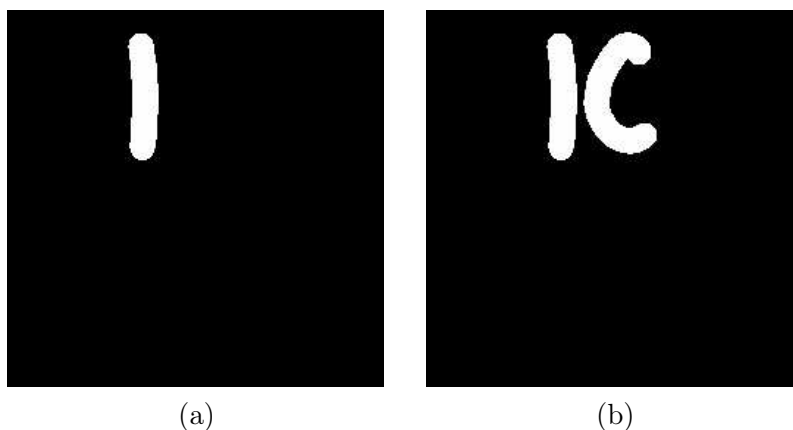


Figure 10.16: Connected components

10.3.3 Skeletonization

Recall that the *skeleton* of an object can be defined by the “medial axis transform”; we may imagine fires burning in along all edges of the object. The places where the lines of fire meet form the skeleton. The skeleton may be produced by morphological methods.

Consider the table of operations as shown in table 10.1.

Erosions	Openings	Set differences
A	$A \circ B$	$A - (A \circ B)$
$A \ominus B$	$(A \ominus B) \circ B$	$(A \ominus B) - ((A \ominus B) \circ B)$
$A \ominus 2B$	$(A \ominus 2B) \circ B$	$(A \ominus 2B) - ((A \ominus 2B) \circ B)$
$A \ominus 3B$	$(A \ominus 3B) \circ B$	$(A \ominus 3B) - ((A \ominus 3B) \circ B)$
\vdots	\vdots	\vdots
$A \ominus kB$	$(A \ominus kB) \circ B$	$(A \ominus kB) - ((A \ominus kB) \circ B)$

Table 10.1: Operations used to construct the skeleton

Here we use the convention that a sequence of k erosions using the same structuring element B is denoted $A \ominus kB$. We continue the table until $(A \ominus kB) \circ B$ is empty. The skeleton is then obtained by taking the unions of all the set differences. An example is given in figure 10.17, using the cross structuring element.

Since $(A \ominus 2B) \circ B$ is empty, we stop here. The skeleton is the union of all the sets in the third column; it is shown in figure 10.18. This method of skeletonization is called *Lantuéjoul's method*; for details see Serra [12].

This algorithm again can be implemented very easily; a function to do so is shown in figure 10.19. We shall experiment with the nice work image.

```
>> nk=imskel(n,sq);
>> imshow(nk)
>> nk2=imskel(n,cr);
>> figure,imshow(nk2)
```

The result is shown in figure 10.20. Image (a) is the result using the square structuring element; Image (b) is the result using the cross structuring element.

Exercises

1. Read in the image `circlesm.tif`.
 - (a) Erode with squares of increasing size until the image starts to split into disconnected components.
 - (b) Using `pixval on`, find the coordinates of a pixel in one of the components.
 - (c) Use the `components` function to isolate that particular component.
2. (a) With your disconnected image from the previous question, compute its boundary.
 - (b) Again with `pixval on`, find a pixel inside one of the boundaries.
 - (c) Use the `regfill` function to fill that region.
 - (d) Display the image as a boundary with one of the regions filled in.

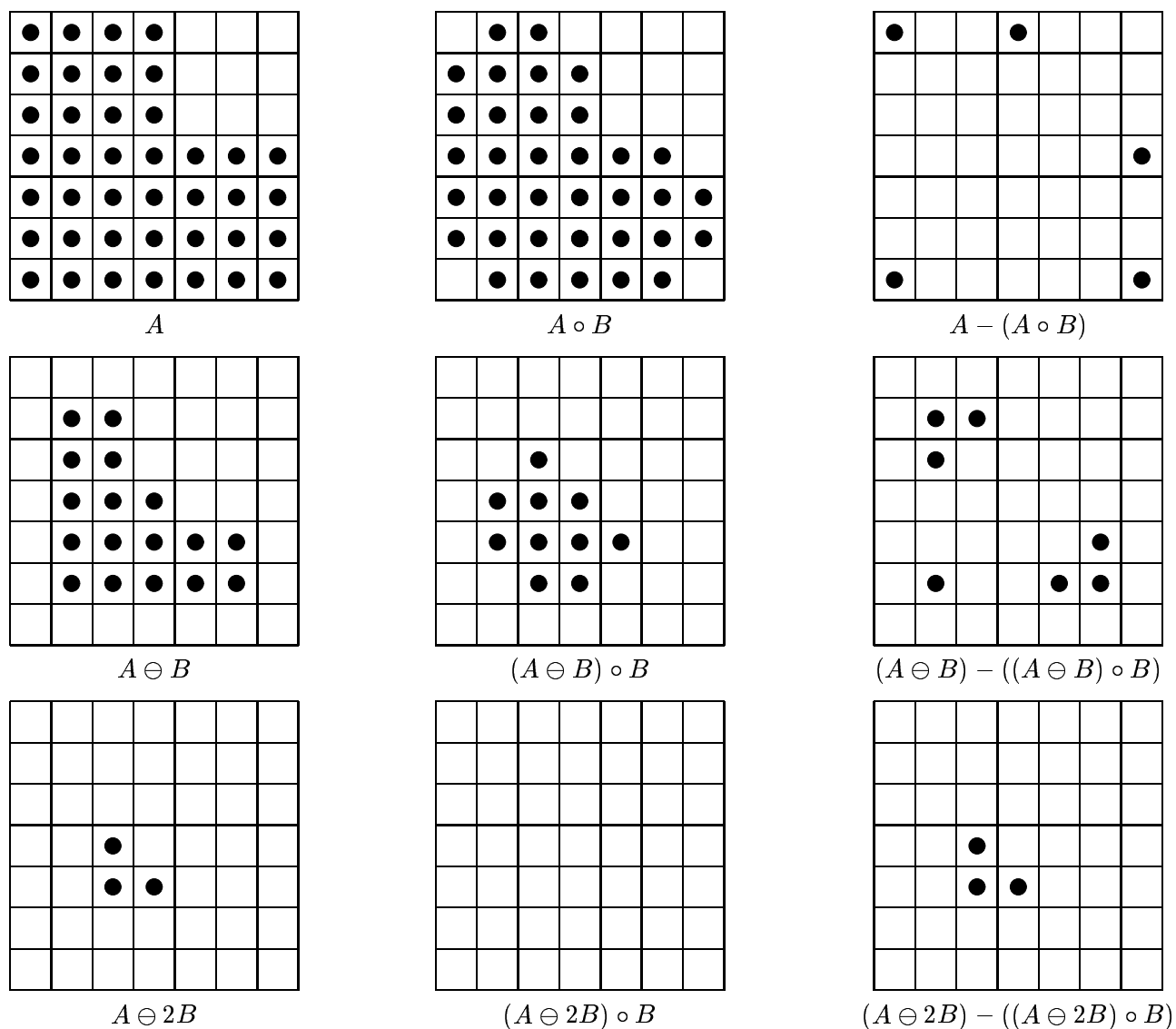


Figure 10.17: Skeletonization

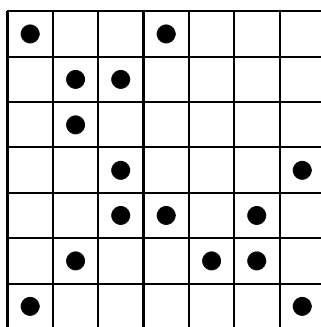


Figure 10.18: The final skeleton

```
function skel = imskel(image,str)
% IMSKEL(IMAGE,STR) - Calculates the skeleton of binary image IMAGE using
% structuring element STR. This function uses Lantejoul's algorithm.
%
skel=zeros(size(image));
e=image;
while (any(e(:))),
    o=imopen(e,str);
    skel=skel | (e&~o);
    e=imerode(e,str);
end
```

Figure 10.19: A simple program for computing skeletons

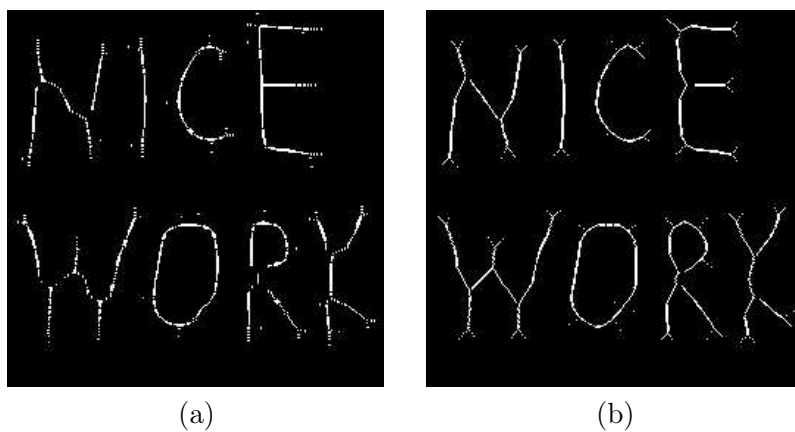


Figure 10.20: Skeletonization of a binary image

3. Using the 3×3 square structuring element, compute the skeletons of
 - (a) a 7 square,
 - (b) a 5×9 rectangle,
 - (c) an L shaped figure formed from an 8×8 square with a 3×3 square taken from a corner,
 - (d) an H shaped figure formed from a 15×15 square with 5×5 squares taken from the centres of the top and bottom,
 - (e) a cross formed from an 11×11 square with 3×3 squares taken from each corner.

In each case check your answer with MATLAB

4. Repeat the above question but use the cross structuring element.
5. For the images listed in question 4, obtain their skeletons by both the `bwmorph` function, and by using the function given in figure 10.19. Which seems to provide the best result?
6. Use the hit-or-miss transform with appropriate structuring elements to find the dot on the “i” in the word “in” in the image `text.tif`.

Chapter 11

Colour processing

For human beings, colour provides one of the most important descriptors of the world around us. The human visual system is particularly attuned to two things: edges, and colour. We have mentioned that the human visual system is not particularly good at recognizing subtle changes in grey values. In this section we shall investigate colour briefly, and then some methods of processing colour images

11.1 What is colour?

Colour study consists of

1. the physical properties of light which give rise to colour,
2. the nature of the human eye and the ways in which it detects colour,
3. the nature of the human vision centre in the brain, and the ways in which messages from the eye are perceived as colour.

Physical aspects of colour

As we have seen in chapter 1, visible light is part of the electromagnetic spectrum. The values for the wavelengths of blue, green and red were set in 1931 by the CIE (Commission Internationale d'Eclairage), an organization responsible for colour standards.

Perceptual aspects of colour

The human visual system tends to perceive colour as being made up of varying amounts of red, green and blue. That is, human vision is particularly sensitive to these colours; this is a function of the cone cells in the retina of the eye. These values are called the *primary colours*. If we add together any two primary colours we obtain the *secondary colours*:

$$\begin{aligned}\text{magenta (purple)} &= \text{red} + \text{blue}, \\ \text{cyan} &= \text{green} + \text{blue}, \\ \text{yellow} &= \text{red} + \text{green}.\end{aligned}$$

The amounts of red, green, and blue which make up a given colour can be determined by a *colour matching experiment*. In such an experiment, people are asked to match a given colour (a *colour*

source) with different amounts of the additive primaries red, green and blue. Such an experiment was performed in 1931 by the CIE, and the results are shown in figure 11.1. Note that for some

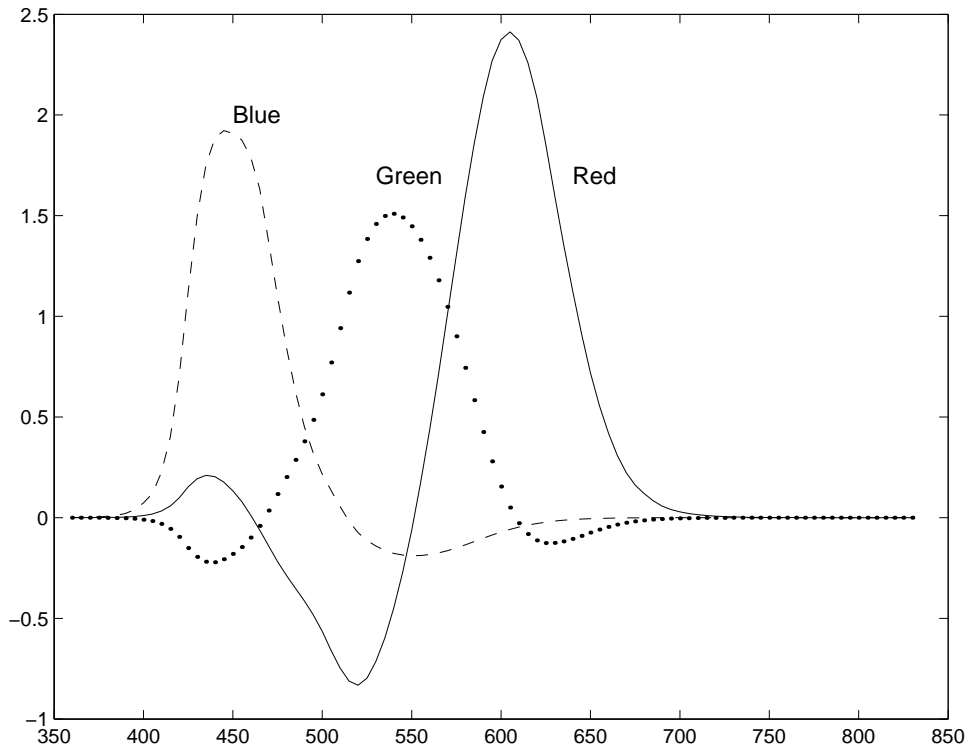


Figure 11.1: RGB colour matching functions (CIE, 1931)

wavelengths, various of the red, green or blue values are *negative*. This is a physical impossibility, but it can be interpreted by adding the primary beam to the colour source, to maintain a colour match.

To remove negative values from colour information, the CIE introduced the XYZ colour model. The values of X , Y and Z can be obtained from the corresponding R , G and B values by a linear transformation:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.431 & 0.342 & 0.178 \\ 0.222 & 0.707 & 0.071 \\ 0.020 & 0.130 & 0.939 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

The inverse transformation is easily obtained by inverting the matrix:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.063 & -1.393 & -0.476 \\ -0.969 & 1.876 & 0.042 \\ 0.068 & -0.229 & 1.069 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}.$$

The XYZ colour matching functions corresponding to the R , G , B curves of figure 11.1 are shown in figure 11.2. The matrices given are not fixed; other matrices can be defined according to the definition of the colour white. Different definitions of white will lead to different transformation matrices.

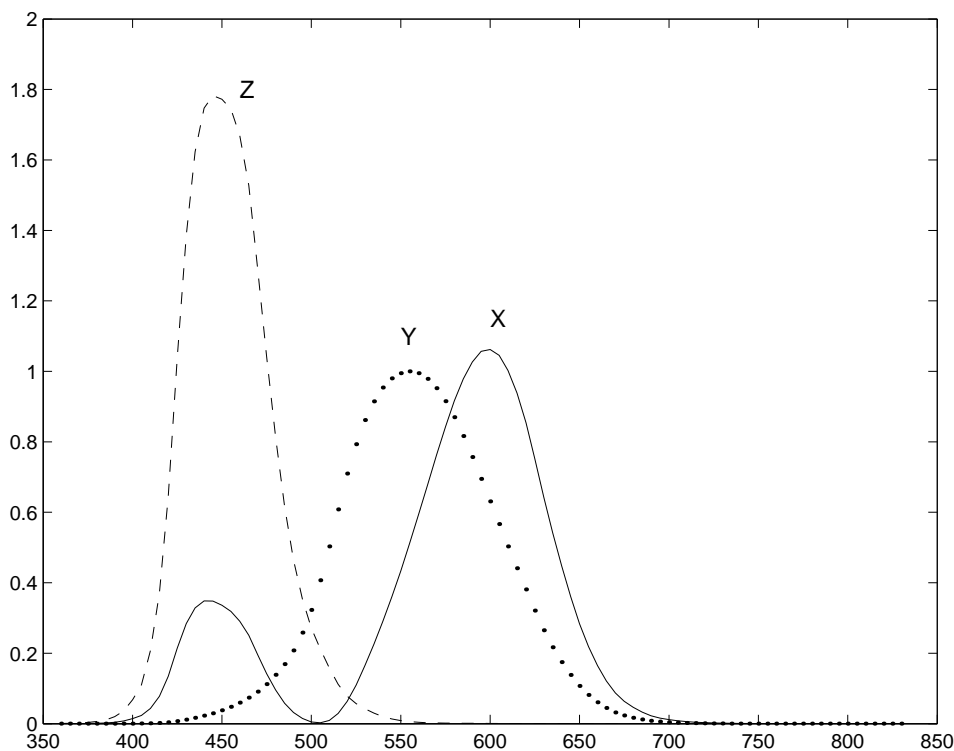


Figure 11.2: XYZ colour matching functions (CIE, 1931)

The CIE required that the Y component corresponded with *luminance*, or perceived brightness of the colour. That is why the row corresponding to Y in the first matrix (that is, the second row) sums to 1, and also why the Y curve in figure 11.2 is symmetric about the middle of the visible spectrum.

In general, the values of X , Y and Z needed to form any particular colour are called the *tristimulus values*. Values corresponding to particular colours can be obtained from published tables. In order to discuss colour independent of brightness, the tristimulus values can be normalized by dividing by $X + Y + Z$:

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

$$z = \frac{Z}{X + Y + Z}$$

and so $x + y + z = 1$. Thus a colour can be specified by x and y alone, called the *chromaticity coordinates*. Given x , y , and Y , we can obtain the tristimulus values X and Z by working through the above equations backwards:

$$X = \frac{x}{y}Y$$

$$Z = \frac{1-x-y}{y}Y.$$

We can plot a chromaticity diagram, using the `ciexyz31.txt`¹ file of XYZ values:

```
>> wxyz=load('ciexyz31.txt');
>> xyz=wxyz(:,2:4)';
>> xy=xyz'./sum(xyz)*[1 1 1];
>> x=xy(:,1)';
>> y=xy(:,2)';
>> figure,plot([x x(1)],[y y(1)]),xlabel('x'),ylabel('y'),axis square
```

Here the matrix `xyz` consists of the second, third and fourth columns of the data, and `plot` is a function which draws a polygon with vertices taken from the `x` and `y` vectors. The extra `x(1)` and `y(1)` ensures that the polygon joins up. The result is shown in figure 11.3. The values of x and y

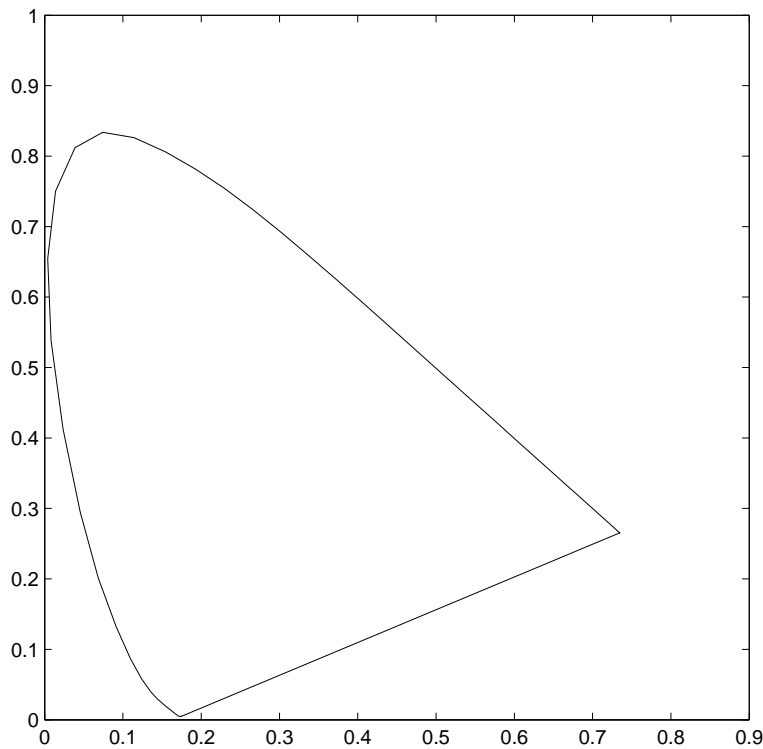


Figure 11.3: A chromaticity diagram

which lie within the horseshoe shape in figure 11.3 represent values which correspond to physically realizable colours. A good account of the XYZ model and associated colour theory can be found in Foley et. al [3].

¹This file can be obtained from the *Colour & Vision Research Laboratories* web page <http://www.cvr1.org>.

11.2 Colour models

A *colour model* is a method for specifying colours in some standard way. It generally consists of a three dimensional coordinate system and a subspace of that system in which each colour is represented by a single point. We shall investigate three systems.

11.2.1 RGB

In this model, each colour is represented as three values R , G and B , indicating the amounts of red, green and blue which make up the colour. This model is used for displays on computer screens; a monitor has three independent electron “guns” for the red, green and blue component of each colour. We have met this model in chapter 1.

Note also from figure 11.1 that some colours require negative values of R , G or B . These colours are not realizable on a computer monitor or TV set, on which only positive values are possible. The colours corresponding to positive values form the *RGB gamut*; in general a colour “gamut” consists of all the colours realizable with a particular colour model. We can plot the RGB gamut on a chromaticity diagram, using the xy coordinates obtained above. To define the gamut, we shall create a $100 \times 100 \times 3$ array, and to each point (i, j) in the array, associate an XYZ triple defined by $(i/100, j/100, 1 - i/100 - j/100)$. We can then compute the corresponding RGB triple, and if any of the RGB values are negative, make the output value white. This is easily done with the simple function shown in figure 11.4.

```
function res=gamut()

global cg;
x2r=[3.063 -1.393 -0.476;-0.969 1.876 0.042;0.068 -0.229 1.069];
cg=zeros(100,100,3);
for i=1:100,
    for j=1:100,
        cg(i,j,:)=x2r*[j/100 i/100 1-i/100-j/100]';
        if min(cg(i,j,:))<0,
            cg(i,j,:)= [1 1 1];
        end;
    end;
end;
res=cg;
```

Figure 11.4: Computing the RGB gamut

We can then display the gamut inside the chromaticity figure by

```
>> imshow(cG),line([x' x(1)],[y' y(1)]),axis square,axis xy,axis on
```

and the result is shown in figure 11.5.

11.2.2 HSV

HSV stands for Hue, Saturation, Value. These terms have the following meanings:

Hue: The “true colour” attribute (red, green, blue, orange, yellow, and so on).

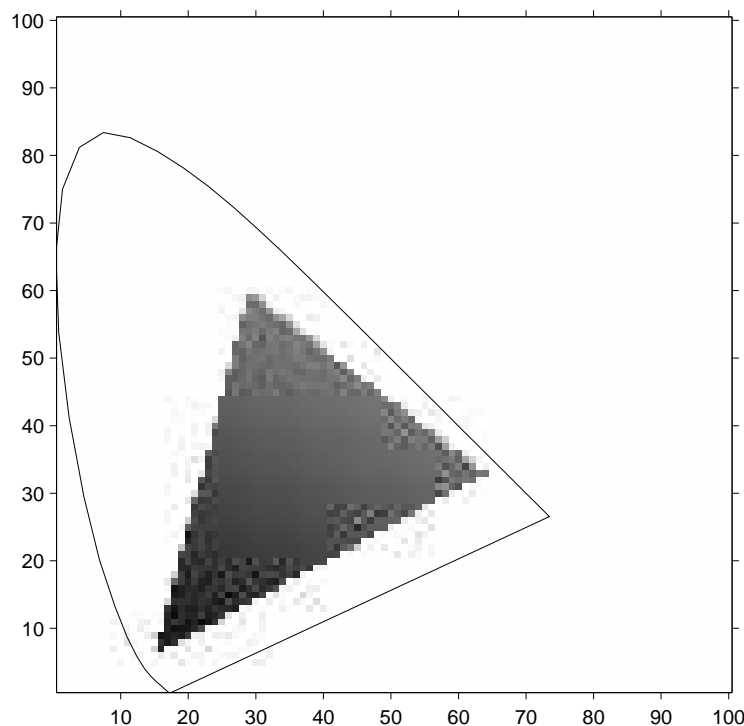


Figure 11.5: The RGB gamut

Saturation: The amount by which the colour has been diluted with white. The more white in the colour, the lower the saturation. So a deep red has high saturation, and a light red (a pinkish colour) has low saturation.

Value: The degree of brightness: a well lit colour has high intensity; a dark colour has low intensity.

This is a more intuitive method of describing colours, and as the intensity is independent of the colour information, this is a very useful model for image processing. We can visualize this model as a cone, as shown in figure 11.6.

Any point on the surface represents a purely saturated colour. The saturation is thus given as the relative distance to the surface from the central axis of the structure. Hue is defined to be the angle measurement from a pre-determined axis, say red.

11.2.3 Conversion between RGB and HSV

Suppose a colour is specified by its RGB values. If all the three values are equal, then the colour will be a grey scale; that is, an intensity of white. Such a colour, containing just white, will thus have a saturation of zero. Conversely, if the RGB values are very different, we would expect the resulting colour to have a high saturation. In particular, if one or two of the RGB values are zero, the saturation will be one, the highest possible value.

Hue is defined as the fraction around the circle starting from red, which thus has a hue of zero. Reading around the circle in figure 11.6 produces the following hues:

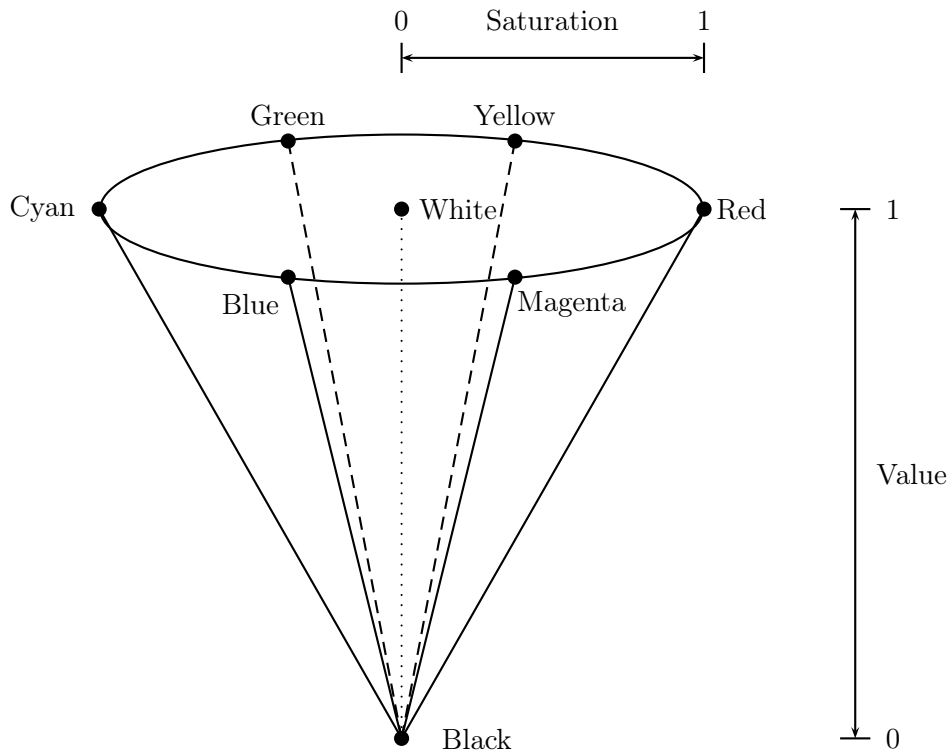


Figure 11.6: The colour space HSV as a cone

Colour	Hue
Red	0
Yellow	0.1667
Green	0.3333
Cyan	0.5
Blue	0.6667
Magenta	0.8333

Suppose we are given three R , G , B values, which we suppose to be between 0 and 1. So if they are between 0 and 255, we first divide each value by 255. We then define:

$$\begin{aligned} V &= \max\{R, G, B\} \\ \delta &= V - \min\{R, G, B\} \\ S &= \frac{\delta}{V} \end{aligned}$$

To obtain a value for Hue, we consider several cases:

1. if $R = V$ then $H = \frac{1}{6} \frac{G - B}{\delta}$,
2. if $G = V$ then $H = \frac{1}{6} \left(2 + \frac{B - R}{\delta} \right)$,

$$3. \text{ if } B = V \text{ then } H = \frac{1}{6} \left(4 + \frac{R - G}{\delta} \right).$$

If H ends up with a negative value, we add 1. In the particular case $(R, G, B) = (0, 0, 0)$, for which both $V = \delta = 0$, we define $(H, S, V) = (0, 0, 0)$.

For example, suppose $(R, G, B) = (0.2, 0.4, 0.6)$. We have

$$V = \max\{0.2, 0.4, 0.6\} = 0.6$$

$$\delta = V - \min\{0.2, 0.4, 0.6\} = 0.6 - 0.2 = 0.4$$

$$S = \frac{0.4}{0.6} = 0.6667$$

Since $B = G$ we have

$$H = \frac{1}{6} \left(4 + \frac{0.2 - 0.4}{0.4} \right) = 0.5833.$$

Conversion in this direction is implemented by the `rgb2hsv` function. This is of course designed to be used on $m \times n \times 3$ arrays, but let's just experiment with our previous example:

```
>> rgb2hsv([0.2 0.4 0.6])

ans =

    0.5833    0.6667    0.6000
```

and these are indeed the H , S and V values we have just calculated.

To go the other way, we start by defining:

$$H' = \lfloor 6H \rfloor$$

$$F = 6H - H'$$

$$P = V(1 - S)$$

$$Q = V(1 - SF)$$

$$T = V(1 - S(1 - F))$$

Since H' is an integer between 0 and 5, we have six cases to consider:

H'	R	G	B
0	V	T	P
1	Q	V	P
2	P	V	T
3	P	Q	V
4	T	P	V
5	V	P	Q

Let's take the HSV values we computed above. We have:

$$H' = \lfloor 6(0.5833) \rfloor = 3$$

$$F = 6(0.5833) - 3 = 0.5$$

$$P = 0.6(1 - 0.6667) = 0.2$$

$$Q = 0.6(1 - (0.6667)(0.5)) = 0.4$$

$$T = 0.6(1 - 0.6667(1 - 0.5)) = 0.4$$

Since $H' = 3$ we have

$$(R, G, B) = (P, Q, V) = (0.2, 0.4, 0.6).$$

Conversion from HSV to RGB is implemented by the `hsv2rgb` function.

11.2.4 YIQ

This colour space is used for TV/video in America and other countries where NTSC is the video standard (Australia uses PAL). In this scheme Y is the “luminance” (this corresponds roughly with intensity), and I and Q carry the colour information. The conversion between RGB is straightforward:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

and

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

The two conversion matrices are of course inverses of each other. Note the difference between Y and V:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ V &= \max\{R, G, B\}. \end{aligned}$$

This reflects the fact that the human visual system assigns more intensity to the green component of an image than to the red and blue components. We note here that other transformations [4] $RGB \leftrightarrow HSV$ have

$$V = 0.333R + 0.333G + 0.333B$$

where the intensity is a simple average of the primary values. Note also that the Y of YIQ is different to the Y of XYZ, with the similarity that both represent luminance.

Since YIQ is a linear transformation of RGB, we can picture YIQ to be a parallelepiped (a rectangular box which has been skewed in each direction) for which the Y axis lies along the central $(0, 0, 0)$ to $(1, 1, 1)$ line of RGB. Figure 11.7 shows this.

That the conversions are linear, and hence easy to do, makes this a good choice for colour image processing. Conversion between RGB and YIQ are implemented with the MATLAB functions `rgb2ntsc` and `ntsc2rgb`.

11.3 Colour images in MATLAB

Since a colour image requires three separate items of information for each pixel, a (true) colour image of size $m \times n$ is represented in MATLAB by an array of size $m \times n \times 3$: a three dimensional array. We can think of such an array as a single entity consisting of three separate matrices aligned vertically. Figure 11.8 shows a diagram illustrating this idea. Suppose we read in an RGB image:

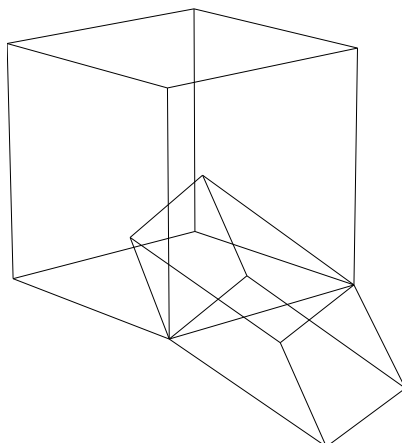


Figure 11.7: The RGB cube and its YIQ transformation

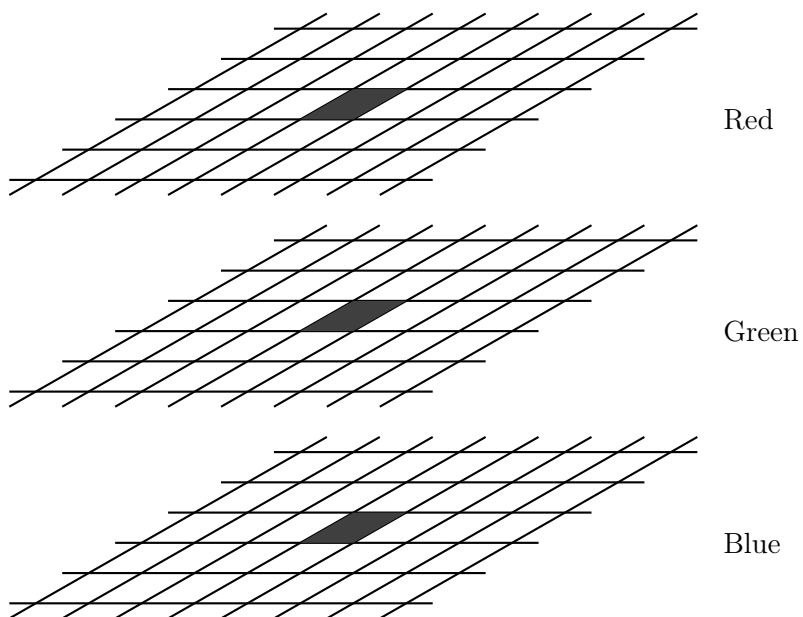


Figure 11.8: A three dimensional array for an RGB image

```
>> x=imread('lily.tif');
>> size(x)

ans =

    186    230     3
```

We can isolate each colour component by the colon operator:

```
x(:,:,1)  The first, or red component
x(:,:,2)  The second, or green component
x(:,:,3)  The third, or blue component
```

These can all be viewed with `imshow`:

```
>> imshow(x)
>> figure,imshow(x(:,:,1))
>> figure,imshow(x(:,:,1))
>> figure,imshow(x(:,:,2))
```

These are all shown in figure 11.9. Notice how the colours with particular hues show up with high



A colour image

Red component

Green component

Blue component

Figure 11.9: An RGB colour image and its components

intensities in their respective components. For the rose in the top right, and the flower in the bottom left, both of which are predominantly red, the red component shows a very high intensity for these two flowers. The green and blue components show much lower intensities. Similarly the green leaves—at the top left and bottom right—show up with higher intensity in the green component than the other two.

We can convert to YIQ or HSV and view the components again:

```
>> xh=rgb2hsv(x);
>> imshow(xh(:,:,1))
>> figure,imshow(xh(:,:,2))
>> figure,imshow(xh(:,:,3))
```

and these are shown in figure 11.10. We can do precisely the same thing for the YIQ colour space:

```
>> xn=rgb2ntsc(x);
>> imshow(xn(:,:,1))
```



Figure 11.10: The HSV components

```
>> figure,imshow(xn(:,:,2))
>> figure,imshow(xn(:,:,3))
```

and these are shown in figure 11.11. Notice that the Y component of YIQ gives a better greyscale



Figure 11.11: The YIQ components

version of the image than the value of HSV. The top right rose, in particular, is quite washed out in figure 11.10 (Value), but shows better contrast in figure 11.11 (Y).

We shall see below how to put three matrices, obtained by operations on the separate components, back into a single three dimensional array for display.

11.4 Pseudocolouring

This means assigning colours to a grey-scale image in order to make certain aspects of the image more amenable for visual interpretation—for example, for medical images. There are different methods of pseudocolouring.

11.4.1 Intensity slicing

In this method, we break up the image into various grey level ranges. We simply assign a different colour to each range. For example:

grey level:	0–63	64–127	128–191	192–255
colour:	blue	magenta	green	red

We can consider this as a mapping, as shown in figure 11.12.

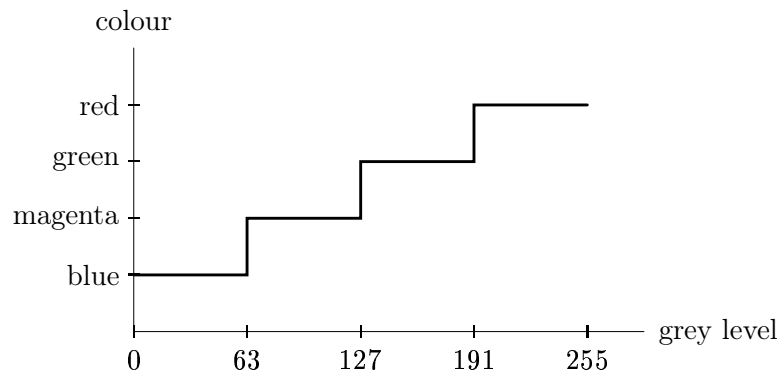
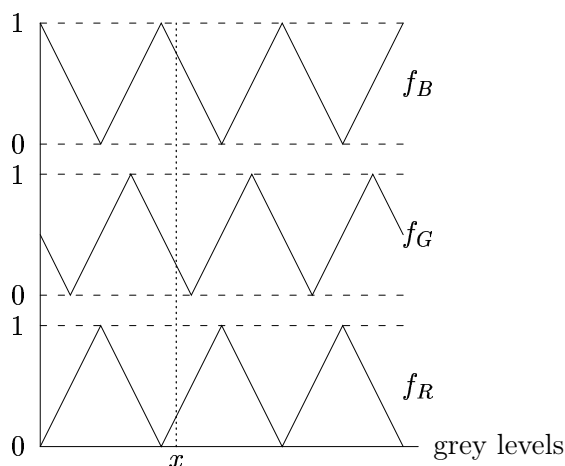


Figure 11.12: Intensity slicing as a mapping

11.4.2 Grey—Colour transformations

We have three functions $f_R(x)$, $f_G(x)$, $f_B(x)$ which assign red, green and blue values to each grey level x . These values (with appropriate scaling, if necessary) are then used for display. Using an appropriate set of functions can enhance a grey-scale image with impressive results.



The grey level x in the diagram is mapped onto red, green and blue values of 0.375, 0.125 and 0.75 respectively.

In MATLAB, a simple way to view an image with added colour is to use `imshow` with an extra `colormap` parameter. For example, consider the image `blocks.tif`. We can add a colour map with the `colormap` function; there are several existing colour maps to choose from. Figure 11.13 shows the children's blocks image (from figure 1.4) after colour transformations. We created the colour image (a) with:

```
>> b=imread('blocks.tif');
>> imshow(b,colormap(jet(256)))
```

However, a bad choice of colour map can ruin an image. Image (b) in figure 11.13 is an example of this, where we apply the `vga` colour map. Since this only has 16 rows, we need to reduce the number of greyscale in the image to 16. This is done with the `grayscale` function:

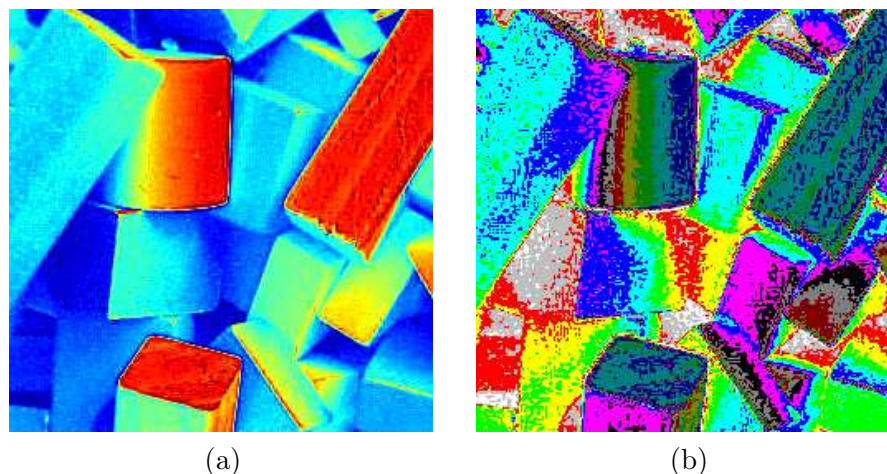


Figure 11.13: Applying a colour map to a greyscale image

```
>> b16=grayscale(b,16);
>> figure,imshow(b16,colormap(vga))
```

The result, although undeniably colourful, is not really an improvement on the original image. The available colour maps are listed in the help file for `graph3d`:

```
hsv      - Hue-saturation-value color map.
hot      - Black-red-yellow-white color map.
gray     - Linear gray-scale color map.
bone     - Gray-scale with tinge of blue color map.
copper   - Linear copper-tone color map.
pink     - Pastel shades of pink color map.
white    - All white color map.
flag     - Alternating red, white, blue, and black color map.
lines    - Color map with the line colors.
colorcube - Enhanced color-cube color map.
vga      - Windows colormap for 16 colors.
jet      - Variant of HSV.
prism    - Prism color map.
cool     - Shades of cyan and magenta color map.
autumn   - Shades of red and yellow color map.
spring   - Shades of magenta and yellow color map.
winter   - Shades of blue and green color map.
summer   - Shades of green and yellow color map.
```

There are help files for each of these colour maps, so that

```
>> help hsv
```

will provide some information on the `hsv` colour map.

We can easily create our own colour map: it must be a matrix with 3 columns, and each row consists of RGB values between 0.0 and 1.0. Suppose we wish to create a blue, magenta, green, red colour map as shown in figure 11.12. Using the RGB values:

Colour	Red	Green	blue
Blue	0	0	1
Magenta	1	0	1
Green	0	1	0
Red	1	0	0

we can create our colour map with:

```
>> mycolourmap=[0 0 1;1 0 1;0 1 0;1 0 0];
```

Before we apply it to the blocks image, we need to scale the image down so that there are only the four greyscales 0, 1, 2 and 3:

```
>> b4=grayscale(b,4);
>> imshow(b4,mycolourmap)
```

and the result is shown in figure 11.14.

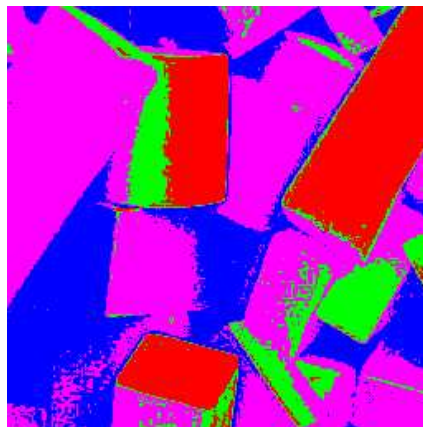


Figure 11.14: An image coloured with a “handmade” colour map

11.5 Processing of colour images

There are two methods we can use:

1. we can process each R, G, B matrix separately,
2. we can transform the colour space to one in which the intensity is separated from the colour, and process the intensity component only.

Schemas for these are given in figures 11.15 and 11.16.

We shall consider a number of different image processing tasks, and apply either of the above schema to colour images.

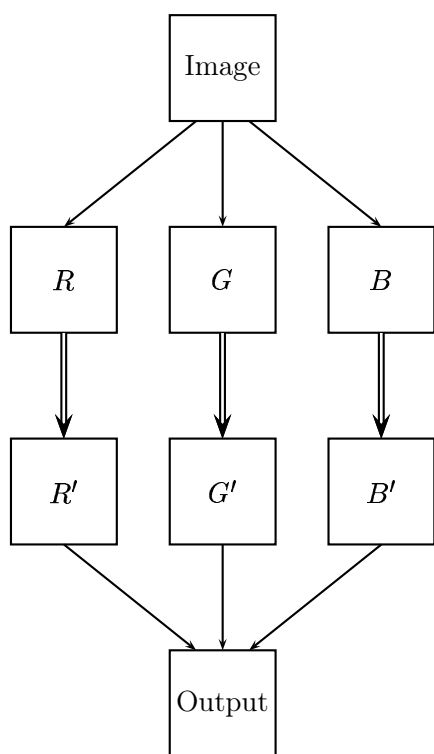


Figure 11.15: RGB processing

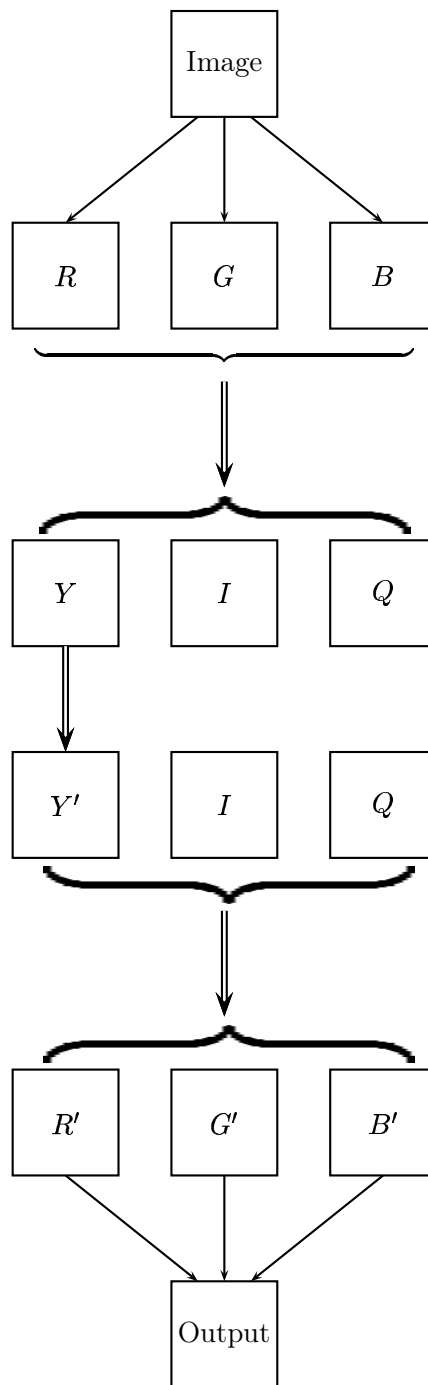


Figure 11.16: Intensity processing

Contrast enhancement

This is best done by processing the intensity component. Suppose we start with the image `cat.tif`, which is an indexed colour image, and convert it to a truecolour (RGB) image.

```
>> [x,map]=imread('cat.tif');
>> c=ind2rgb(x,map);
```

Now we have to convert from RGB to YIQ, so as to be able to isolate the intensity component:

```
>> cn=rgb2ntsc(c);
```

Now we apply histogram equalization to the intensity component, and convert back to RGB for display:

```
>> cn(:,:,1)=histeq(cn(:,:,1));
>> c2=ntsc2rgb(cn);
>> imshow(c2)
```

The result is shown in figure 11.17. Whether this is an improvement is debatable, but it has had its contrast enhanced.

But suppose we try to apply histogram equalization to each of the RGB components:

```
>> cr=histeq(c(:,:,1));
>> cg=histeq(c(:,:,2));
>> cb=histeq(c(:,:,3));
```

Now we have to put them all back into a single 3 dimensional array for use with `imshow`. The `cat` function is what we want:

```
>> c3=cat(3,cr,cg,cb);
>> imshow(c3)
```

The first variable to `cat` is the dimension along which we want our arrays to be joined. The result is shown for comparison in figure 11.17. This is not acceptable, as some strange colours have been introduced; the cat's fur has developed a sort of purplish tint, and the grass colour is somewhat washed out.

Spatial filtering

It very much depends on the filter as to which schema we use. For a low pass filter, say a blurring filter, we can apply the filter to each RGB component:

```
>> a15=fspecial('average',15);
>> cr=filter2(a15,c(:,:,1));
>> cg=filter2(a15,c(:,:,2));
>> cb=filter2(a15,c(:,:,3));
>> blur=cat(3,cr,cg,cb);
>> imshow(blur)
```

and the result is shown in figure 11.18. We could also obtain a similar effect by applying the filter to the intensity component only. But for a high pass filter, for example an unsharp masking filter, we are better off working with the intensity component only:



Intensity processing



Using each RGB component

Figure 11.17: Histogram equalization of a colour image

```
>> cn=rgb2ntsc(c);
>> a=fspecial('unsharp');
>> cn(:,:,1)=filter2(a,cn(:,:,1));
>> cu=ntsc2rgb(cn);
>> imshow(cu)
```

and the result is shown in figure 11.18. In general, we will obtain reasonable results using the



Low pass filtering



High pass filtering

Figure 11.18: Spatial filtering of a colour image

intensity component only. Although we can sometimes apply a filter to each of the RGB components, as we did for the blurring example above, we cannot be guaranteed a good result. The problem is that any filter will change the values of the pixels, and this may introduce unwanted colours.

Noise reduction

As we did in chapters 5 and 6, we shall use the image `twins.tif`: but now in full colour!

```
>> tw=imread('twins.tif');
```

Now we can add noise, and look at the noisy image, and its RGB components:

```
>> tn=imnoise(tw,'salt & pepper');
>> imshow(tn)
>> figure,imshow(tn(:,:,1))
>> figure,imshow(tn(:,:,2))
>> figure,imshow(tn(:,:,3))
```

These are all shown in figure 11.19. It would appear that we should apply median filtering to each

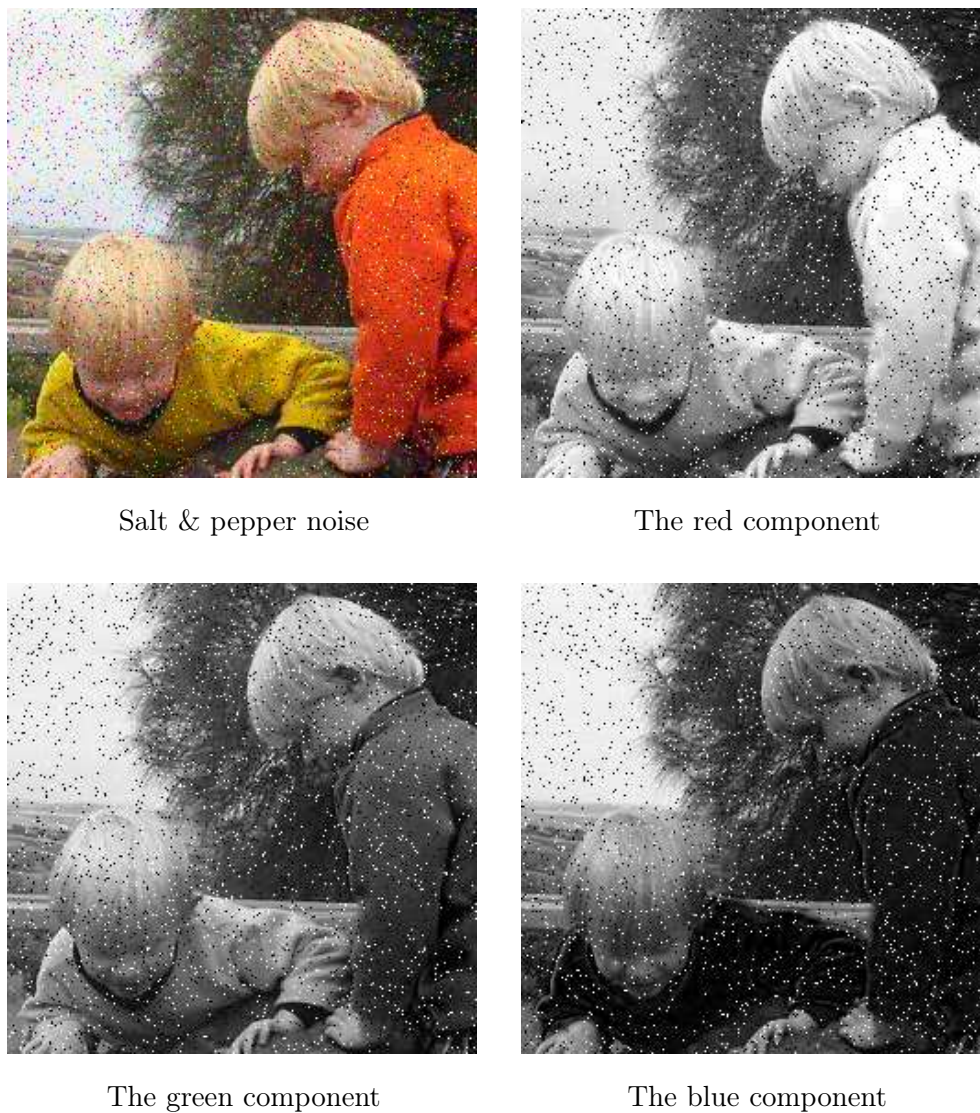


Figure 11.19: Noise on a colour image

of the RGB components. This is easily done:

```
>> trm=medfilt2(tn(:,:,1));
>> tgm=medfilt2(tn(:,:,2));
>> tbn=medfilt2(tn(:,:,3));
```

```
>> tm=cat(3, trm, tgm, tbn);
>> imshow(tm)
```

and the result is shown in figure 11.20. We can't in this instance apply the median filter to the intensity component only, because the conversion from RGB to YIQ spreads the noise across all the YIQ components. If we remove the noise from Y only:

```
>> tnn=rgb2ntsc(tm);
>> tnn(:,:,1)=medfilt2(tnn(:,:,1));
>> tm2=ntsc2rgb(tnn);
>> imshow(tm2)
```

then the noise has been slightly diminished as shown in figure 11.20, but it is still there. If the noise



Denoising each RGB component

Denoising Y only

Figure 11.20: Attempts at denoising a colour image

applies to only one of the RGB components, then it would be appropriate to apply a denoising technique to this component only.

Also note that the method of noise removal must depend on the generation of noise. In the above example we tacitly assumed that the noise was generated after the image had been acquired and stored as RGB components. But as noise can arise anywhere in the image acquisition process, it is quite reasonable to assume that noise might affect only the brightness of the image. In such a case denoising the Y component of YIQ will produce the best results.

Edge detection

An edge image will be a binary image containing the edges of the input. We can go about obtaining an edge image in two ways:

1. we can take the intensity component only, and apply the `edge` function to it,
2. we can apply the `edge` function to each of the RGB components, and join the results.

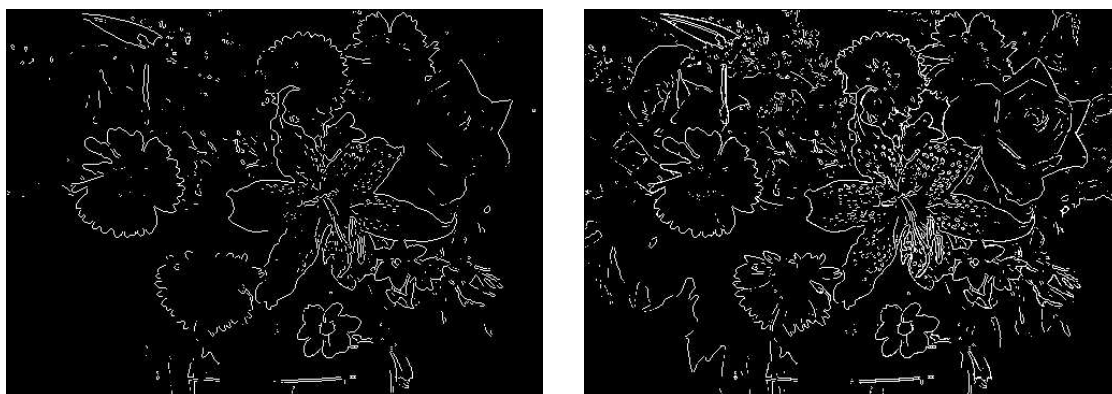
To implement the first method, we start with the `rgb2gray` function:


```
>> fg=rgb2gray(f);
>> fe1=edge(fg);
>> imshow(fe1)
```

Recall that `edge` with no parameters implements Sobel edge detection. The result is shown in figure 11.21. For the second method, we can join the results with the logical “or”:

```
>> f1=edge(f(:,:,1));
>> f2=edge(f(:,:,2));
>> f3=edge(f(:,:,3));
>> fe2=f1 | f2 | f3;
>> figure,imshow(fe2)
```

and this is also shown in figure 11.21. The edge image `fe2` is a much more complete edge image.



`fe1`: Edges after `rgb2gray`

`fe2`: Edges of each RGB component

Figure 11.21: The edges of a colour image

Notice that the rose now has most of its edges, where in image `fe1` only a few were shown. Also note that there are the edges of some leaves in the bottom left of `fe2` which are completely missing from `fe1`. The success of these methods will also depend on the parameters of the edge function chosen; for example the threshold value used. In the examples shown, the `edge` function has been used with its default threshold.

Exercises

1. By hand, determine the saturation and intensity components of the following image, where the RGB values are as given:

(0, 1, 1)	(1, 2, 3)	(7, 7, 7)	(5, 1, 2)	(1, 1, 7)
(2, 1, 2)	(1, 7, 7)	(2, 0, 2)	(3, 3, 2)	(5, 5, 0)
(4, 4, 4)	(4, 6, 7)	(4, 5, 6)	(1, 5, 7)	(3, 6, 7)
(3, 0, 3)	(5, 2, 2)	(1, 1, 1)	(6, 6, 0)	(2, 2, 2)
(1, 2, 1)	(0, 4, 4)	(3, 1, 6)	(3, 3, 3)	(2, 4, 6)

Chapter 12

Image coding and compression

12.1 Lossless and lossy compression

We have seen that image files can be very large. It is thus important for reasons both of storage and file transfer to make these file sizes smaller, if possible. In section 1.9 we touched briefly on the topic of compression; in this section we investigate some standard compression methods. It will be necessary to distinguish between two different classes of compression methods: *lossless compression*, where all the information is retained, and *lossy compression* where some information is lost.

Lossless compression is preferred for images of legal, scientific or political significance, where loss of data, even of apparent insignificance, could have considerable consequences. Unfortunately this style tends not to lead to high compression ratios. However, lossless compression is used as part of many standard image formats.

12.2 Huffman coding

The idea of Huffman coding is simple. Rather than using a fixed length code (8 bits) to represent the grey values in an image, we use a variable length code, with smaller length codes corresponding to more probable grey values.

A small example will make this clear. Suppose we have a 2-bit greyscale image with only four grey levels: 0, 1, 2, 3, with the probabilities 0.2, 0.4, 0.3 and 0.1 respectively. That is, 20% of pixels in the image have grey value 0; 40% have grey value 1, and so on. The following table shows fixed length and variable length codes for this image:

Grey value	Probability	Fixed code	Variable code
0	0.2	00	000
1	0.4	01	1
2	0.3	10	01
3	0.1	11	001

Now consider how this image has been compressed. Each grey value has its own unique identifying code. The average number of bits per pixel can be easily calculated as the expected value (in a probabilistic sense):

$$(0.2 \times 3) + (0.4 \times 1) + (0.3 \times 2) + (0.1 \times 3) = 1.9.$$

Notice that the longest codewords are associated with the lowest probabilities. This average is indeed smaller than 2.

This can be made more precise by the notion of *entropy*, which is a measure of the amount of information. Specifically, the entropy H of an image is the theoretical minimum number of bits per pixel required to encode the image with no loss of information. It is defined by

$$H = - \sum_{i=0}^{L-1} p_i \log_2(p_i)$$

where the index i is taken over all greyscales of the image, and p_i is the probability of grey level i occurring in the image. Very good accounts of the basics of information theory and entropy are given by Roman [10] and Welsh [16]. In the example given above,

$$H = - \left(0.2 \log_2(0.2) + 0.4 \log_2(0.4) + 0.3 \log_2(0.3) + 0.1 \log_2(0.1) \right) = 1.8464.$$

This means that no matter what coding scheme is used, it will never use less than 1.8464 bits per pixel. On this basis, the Huffman coding scheme given above, giving an average number of bits per pixel much closer to this theoretical minimum than 2, provides a very good result.

To obtain the Huffman code for a given image we proceed as follows:

1. Determine the probabilities of each grey value in the image.
2. Form a binary tree by adding probabilities two at a time, always taking the two lowest available values.
3. Now assign 0 and 1 arbitrarily to each branch of the tree from its apex.
4. Read the codes from the top down.

To see how this works, consider the example of a 3-bit greyscale image (so the grey values are 0–7) with the following probabilities:

Grey value	0	1	2	3	4	5	6	7
Probability	0.19	0.25	0.21	0.16	0.08	0.06	0.03	0.02

For these probabilities, the entropy can be calculated to be 2.6508. We can now combine probabilities two at a time as shown in figure 12.1.

Note that if we have a choice of probabilities we choose arbitrarily. The second stage consists of arbitrarily assigning 0's and 1's to each branch of the tree just obtained. This is shown in figure 12.2.

To obtain the codes for each grey value, start at the 1 on the top right, and work back towards the grey value in question, listing the numbers passed on the way. This produces:

Grey value	Huffman code
0	00
1	10
2	01
3	110
4	1110
5	11110
6	111110
7	111111

As above, we can evaluate the average number of bits per pixel as an expected value:

$$(0.19 \times 2) + (0.25 \times 2) + (0.21 \times 2) + (0.16 \times 3) + (0.08 \times 4) + (0.06 \times 5) + (0.03 \times 6) + (0.02 \times 6) = 2.7$$

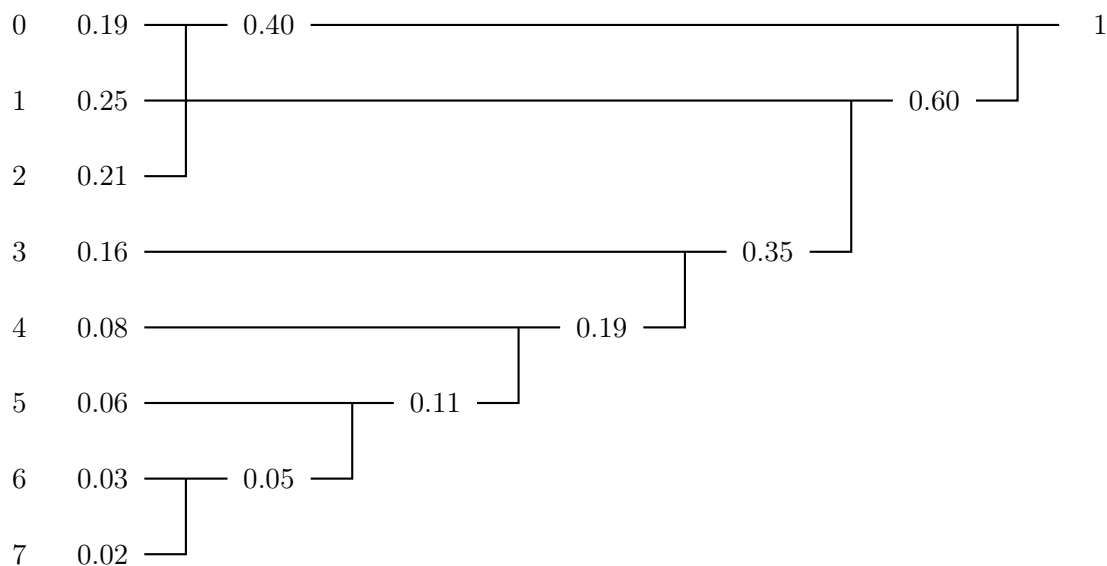


Figure 12.1: Forming the Huffman code tree

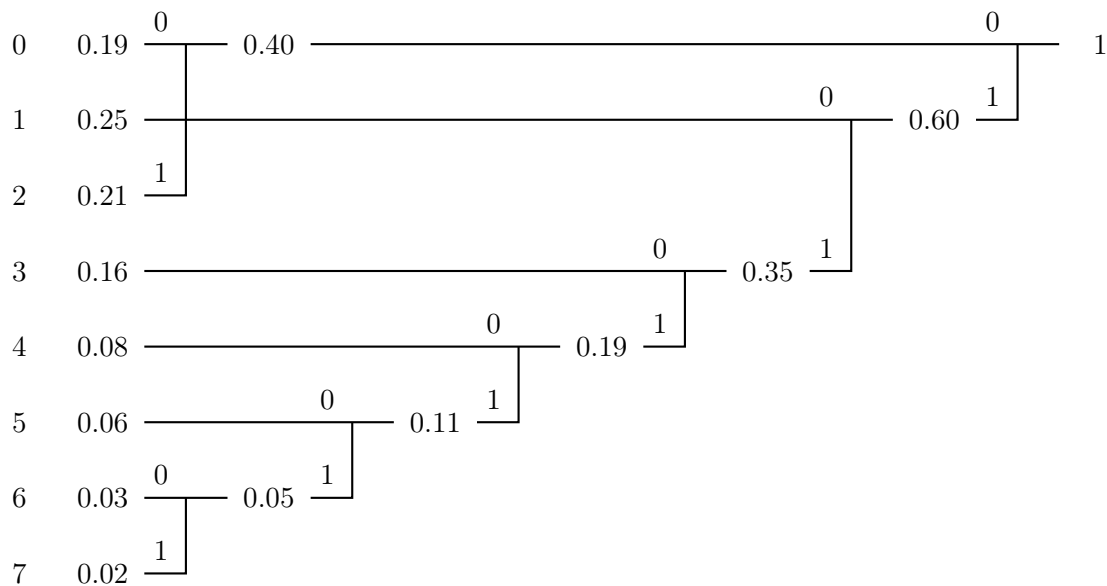


Figure 12.2: Assigning 0's and 1's to the branches

which is a significant improvement over 3 bits per pixel, and very close to the theoretical minimum of 2.6508 given by the entropy.

Huffman codes are *uniquely decodable*, in that a string can be decoded in only one way. For example, consider the string

1 1 0 1 1 1 0 0 0 0 1 0 0 1 1 1 1 1 0

to be decoded with the Huffman code generated above. There is no code word 1, or 11, so we may take the first three bits 110 as being the code for grey value 3. Notice also that no other code word begins with this string. For the next few bits, 1110 is a code word; no other begins with this string, and no other smaller string is a codeword. So we can decode this string as grey level 4. Continuing in this way we obtain:

$\underbrace{1\ 1\ 0}_3$ $\underbrace{1\ 1\ 1\ 0}_4$ $\underbrace{0\ 0}_0$ $\underbrace{0\ 0}_0$ $\underbrace{1\ 0}_1$ $\underbrace{0\ 1}_2$ $\underbrace{1\ 1\ 1\ 1\ 0}_5$

as the decoding for this string.

For more information about Huffman coding, and its limitations and generalizations, see [4, 9].

12.3 Run length encoding

Run length encoding (RLE) is based on a simple idea: to encode strings of zeros and ones by the number of repetitions in each string. RLE has become a standard in facsimile transmission. For a binary image, there are many different implementations of RLE; one method is to encode each line separately, starting with the number of 0's. So the following binary image:

```
0 1 1 0 0 0
0 0 1 1 1 0
1 1 1 0 0 1
0 1 1 1 1 0
0 0 0 1 1 1
1 0 0 0 1 1
```

would be encoded as

(123)(231)(0321)(141)(33)(0132)

Another method [14] is to encode each row as a list of pairs of numbers; the first number in each pair given the starting position of a run of 1's, and the second number its length. So the above binary image would have the encoding

(22)(33)(1361)(24)(43)(1152)

Greyscale images can be encoded by breaking them up into their *bit planes*; these were discussed in chapter 1.

To give a simple example, consider the following 4-bit image and its binary representation:

10	7	8	9		1010	0111	1000	1001
11	8	7	6		1011	1000	0111	0110
9	7	5	4	→	1001	0111	0101	0100
10	11	2	1		1010	1011	0010	0001

We may break it into bit planes as shown:

0 1 0 1	1 1 0 0	0 1 0 0	1 0 1 1
1 0 1 0	1 0 1 1	0 0 1 1	1 1 0 0
1 1 1 0	0 1 0 0	0 1 1 1	1 0 0 0
0 1 0 1	1 1 1 0	0 0 0 0	1 1 0 0
0th plane	1st plane	2nd plane	3rd plane

and then each plane can be encoded separately using our chosen implementation of RLE.

However, there is a problem with bit planes, and that is that small changes of grey value may cause significant changes in bits. For example, the change from value 7 to 8 causes the change of all four bits, since we are changing the binary strings 0111 to 1000. The problem is of course exacerbated for 8-bit images. For RLE to be effective, we should hope that long runs of very similar grey values would result in very good compression rates for the code. But this may not be the case. A 4-bit image consisting of randomly distributed 7's and 8's would thus result in uncorrelated bit planes, and little effective compression.

To overcome this difficulty, we may encode the grey values with their binary *Gray codes*. A Gray code is an ordering of all binary strings of a given length so that there is only one bit change between a string and the next. So a 4-bit Gray code is:

15	1 0 0 0
14	1 0 0 1
13	1 0 1 1
12	1 0 1 0
11	1 1 1 0
10	1 1 1 1
9	1 1 1 0
8	1 1 0 0
7	0 1 0 0
6	0 1 0 1
5	0 1 1 1
4	0 1 1 0
3	0 0 1 0
2	0 0 1 1
1	0 0 0 1
0	0 0 0 0

See [9] for discussion and detail. To see the advantages, consider the following 4-bit image with its binary and Gray code encodings:

8 8 7 8	1000 1000 0111 1000	1100 1100 0100 1100
8 7 8 7	1000 0111 1000 0111	1100 0100 1100 0100
7 7 8 7	0111 0111 1000 0111	0100 0100 1100 0100
7 8 7 7	0111 1000 0111 0111	0100 1100 0100 0100

where the first binary array is the standard binary encoding, and the second array the Gray codes. The binary bit planes are:

0 0 1 0	0 0 1 0	0 0 1 0	1 1 0 1
0 1 0 1	0 1 0 1	0 1 0 1	1 0 1 0
1 1 0 1	1 1 0 1	1 1 0 1	0 0 1 0
1 0 1 1	1 0 1 1	1 0 1 1	0 1 0 0
0th plane	1st plane	2nd plane	3rd plane

and the bit planes corresponding to the Gray codes are:

0 0 0 0	0 0 0 0	1 1 1 1	1 1 0 1
0 0 0 0	0 0 0 0	1 1 1 1	1 0 1 0
0 0 0 0	0 0 0 0	1 1 1 1	0 0 1 0
0 0 0 0	0 0 0 0	1 1 1 1	0 1 0 0
0th plane	1st plane	2nd plane	3rd plane

Notice that the Gray code planes are highly correlated except for one bit plane, whereas all the binary bit planes are uncorrelated.

Run length encoding in MATLAB

We can experiment with run length encoding by writing a simple function to implement it. To make it easy on ourselves, we shall just stick with single binary images. Our output will be a single vector, giving the numbers of zeros and ones, alternating, through our image row by row. We start by putting our image into a single row. For a binary image `im`, this can be done with the two commands

```
L=prod(size(im));
im=reshape(im',1,L);
```

To find the number of beginning zeros, we obtain the position of the first 1, thus:

```
min(find(im==1))
```

We append one less than this result to our output vector. It may well be that there are no further ones, in which case we have reached the end of the file, and we stop by appending the current length of our image to the output vector. We now change to looking for the place of the next zero; we can use the `min(find)` command again, but we first reduce our image by the zeroes we have already found.

The following table shows how we can implement run length encoding:

Image	Looking for	Place	RLE output
			<code>[]</code>
<code>[0 0 1 1 1 0 0 0 1]</code>	1	3	<code>[2]</code>
<code> [1 1 1 0 0 0 1]</code>	0	4	<code>[2 3]</code>
<code> [0 0 0 1]</code>	1	4	<code>[2 3 3]</code>
<code> [1]</code>	0	<i>Not found</i>	<code>[2 3 3 1]</code>

One more example:

Image	Looking for	Place	RLE output
			[]
[1 1 1 0 0 0 0 1 1]	1	1	[0]
[1 1 1 0 0 0 0 1 1]	0	4	[0 3]
[0 0 0 0 1 1]	1	4	[0 3 4]
[1 1]	0	<i>Not found</i>	[0 3 4 2]

Notice that in this second example, since the length of our initial run of zeros was found to be zero, we do not reduce the length of our image in the next step.

Figure 12.3 shows the implementation of this algorithm in Matlab.

```
function out=rle(image)
%
% RLE(IMAGE) produces a vector containing the run-length encoding of
% IMAGE, which should be a binary image. The image is set out as a long
% row, and the code contains the number of zeros, followed by the number
% of ones, alternating.
%
% Example:
%
%   rle([1 1 1 0 0;0 0 1 1 1;1 1 0 0 0])
%
%   ans =
%
%       0     3     4     5     3
%
L=prod(size(image));
im=reshape(image',1,L);
x=1;
out=[];
while L ~= 0,
    temp=min(find(im == x));
    if isempty(temp),
        out=[out L];
        break
    end;
    out=[out temp-1];
    x=1-x;
    im=im(temp:L);
    L=L-temp+1;
end;
```

Figure 12.3: A MATLAB function for obtaining the run length code of a binary image

Now we can test this on a few images:

```
>> c=imread('circles.tif');
>> cr=rle(c);
>> whos c cr
Name      Size      Bytes  Class
c         256x256   65536  uint8 array (logical)
```



```
cr          1x693          5544  double array
```

We can reduce the size of the output by storing it using the data type `uint16`: unsigned 16-bit integers.

```
>> cr=uint16(cr);
>> whos cr
Name      Size      Bytes  Class
cr        1x693      1386  uint16 array
```

Even if the original circles image was stored as one bit per pixel, or eight pixels per byte, we would have a total of

$$\frac{65536}{8} = 8192$$

bytes: still more than the run length code. So in this example, run length encoding provides a reasonable amount of compression.

```
>> t=imread('text.tif');
>> tr=rle(t);
>> whos t tr
Name      Size      Bytes  Class
t         256x256    65536  uint8 array (logical)
tr        1x2923     23384  double array
```

Again better compression can be obtained by changing the data type.

```
>> tr=uint16(tr);
>> whos tr
Name      Size      Bytes  Class
tr        1x2923     5846  uint16 array
```

Although this is not as good as for the previous image, it is still better than the minimum of 8192 bytes for the original image.

Exercises

1. Construct a Huffman code for each of the probability tables given:

grey scale		0	1	2	3	4	5	6	7
probability	(a)	.07	.11	.08	.04	.5	.05	.06	.09
	(b)	.13	.12	.13	.13	.12	.12	.12	.13
	(c)	.09	.13	.15	.1	.14	.12	.11	.16

In each case determine the average bits/pixel given by your code.

2. From your results of the previous question, what do think are the conditions of the probability distribution which give rise to a high compression rate using Huffman coding?