The image

`monastery.png`

has been taken from `http://www.public-domain-image.com`; more particularly from `http://bit.ly/18V2Z57`: the photographer is Andrew McMillan.

The image

`daisies.png`

has been taken from `www.pixabay.com` as an image in the public domain; the photographer is Marianne Langenbach.

The iconic image

`thylacine.png`

showing the last known living thylacine, or Tasmanian tiger, at the Hobart zoo in 1933, is now in the public domain.

The x-ray image

`xray.png`

has been taken from Wikimedia Commons, as an image in the public domain. It can be found at `http://bit.ly/1Az0Tzk`; the original image has been provided by Diego Grez from Santa Cruz, Chile.

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 colors; 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 that represents something. It may be a picture of a person, people, or animals, an outdoor scene, 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 Section 1.4). It is necessary to realize that these two aspects represent two separate but equally important aspects of image processing. A procedure that satisfies condition (1)—a procedure that 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.

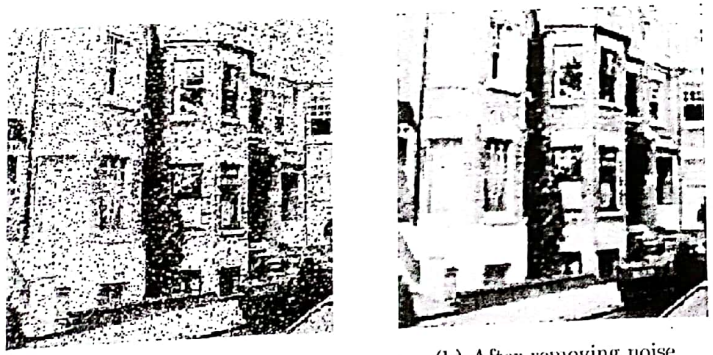
Examples of (1) may include:

- 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

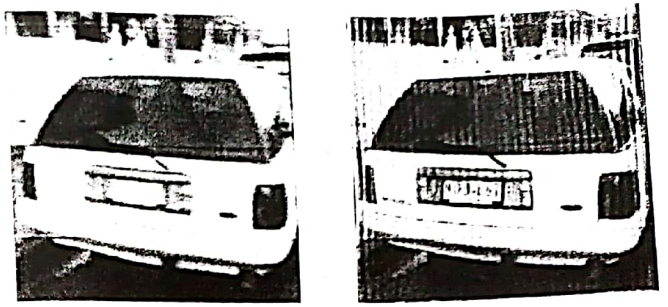
- Removing "noise" from an image; noise being random errors in the image. An example is given in Figure 1.2. Noise is a very common problem in data transmission; all sorts of electronic components may affect data passing through them, and the results may be undesirable. As we shall see in Chapter 8, noise may take many different forms, and each type of noise requires a different method of removal.



(a) The original image (b) After removing noise

FIGURE 1.2: Removing noise from an image

- 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.



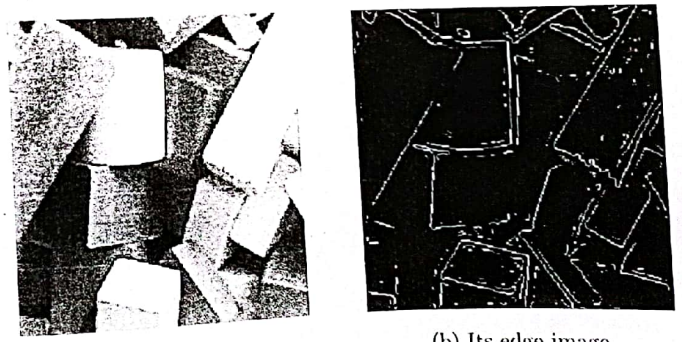
(a) The original image (b) After removing the blur

FIGURE 1.3: Image deblurring

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 previously.

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

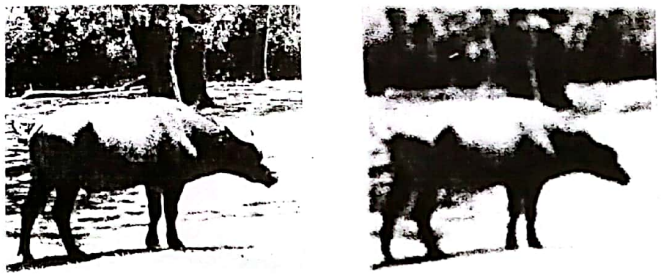


(a) The original image (b) Its edge image

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 color. 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.



(a) The original image (b) Blurring to remove detail

FIGURE 1.5: Blurring an image

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 7. 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 [7].

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*.

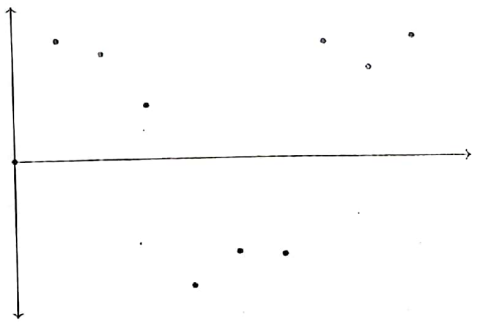


FIGURE 1.6: Sampling a function – undersampling

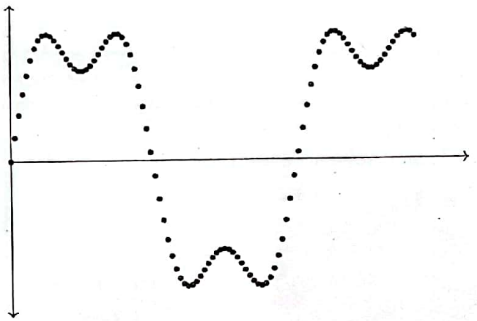
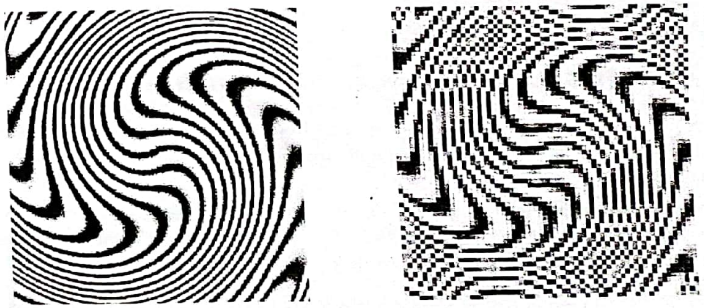


FIGURE 1.7: Sampling a function with more points



Correct sampling; no aliasing An undersampled version with aliasing

FIGURE 1.8: Effects of sampling

The sampling rate will of course affect the final resolution of the image; we discuss this in Chapter 3. 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 that 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.

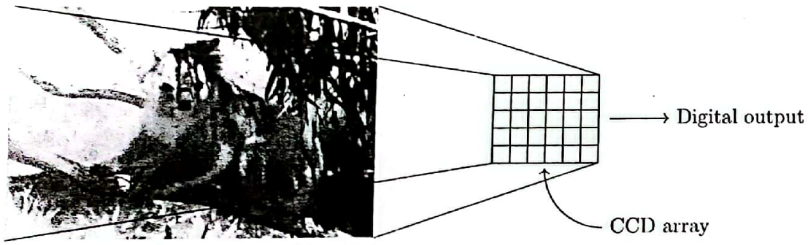


FIGURE 1.9: Capturing an image with a CCD array

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*.

Digital still cameras use a range of devices, from floppy discs and CDs, 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.

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.

FIGURE 1.10: Capturing an image with a CCD scanner

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 may use x-rays or electron beams. As we can see from Figure 1.11,

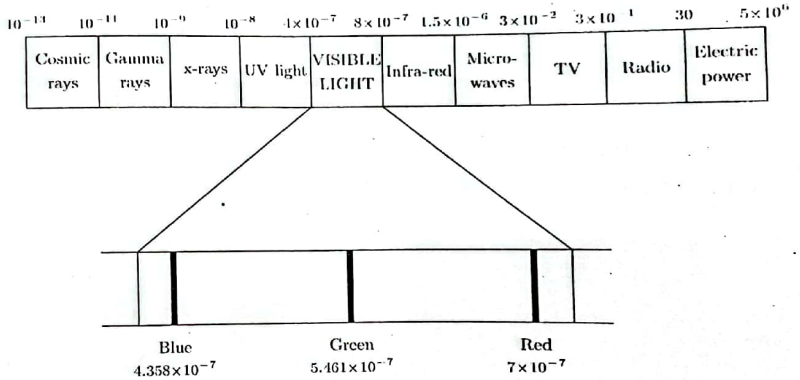


FIGURE 1.11: The electromagnetic spectrum

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 [8] 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 [48].

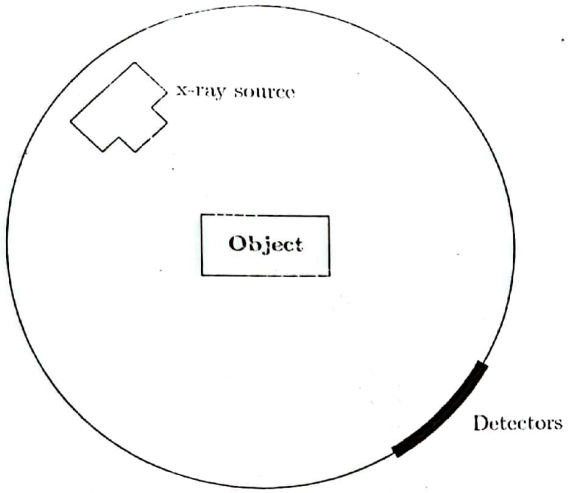


FIGURE 1.12: X-ray tomography

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 gray only), so no color. 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 be vital for the development and implementation of image processing techniques.

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 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 *neighborhood*. A neighborhood can be characterized by its shape in the same way as a matrix: we can speak, for example, of a 3×3 neighborhood or of a 5×7

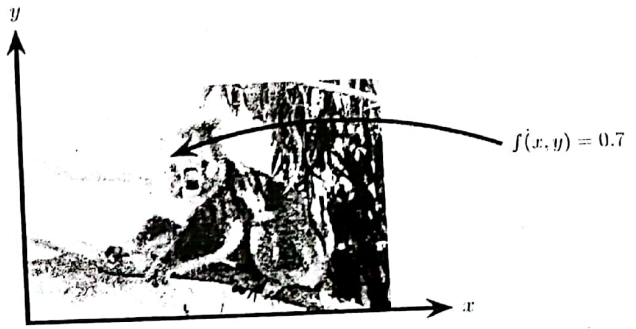


FIGURE 1.13: An image as a function

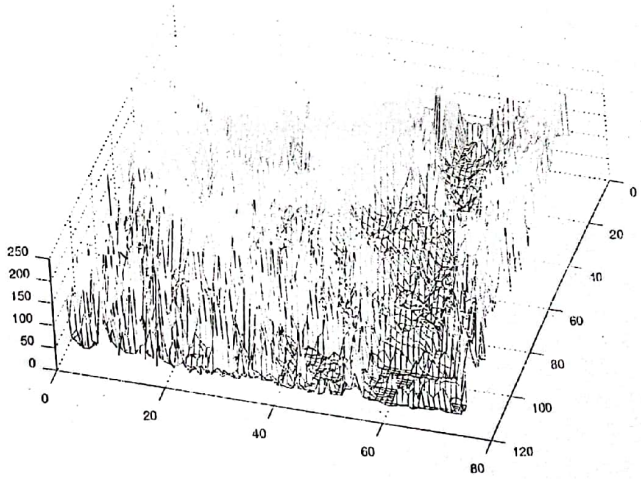


FIGURE 1.14: The image of Figure 1.13 plotted as a function of two variables

neighborhood. Except in very special circumstances, neighborhoods have odd numbers of rows and columns; this ensures that the current pixel is in the center of the neighborhood. An example of a neighborhood is given in Figure 1.15. If a neighborhood has an even number of rows or columns (or both), it may be necessary to specify which pixel in the neighborhood is the "current pixel."

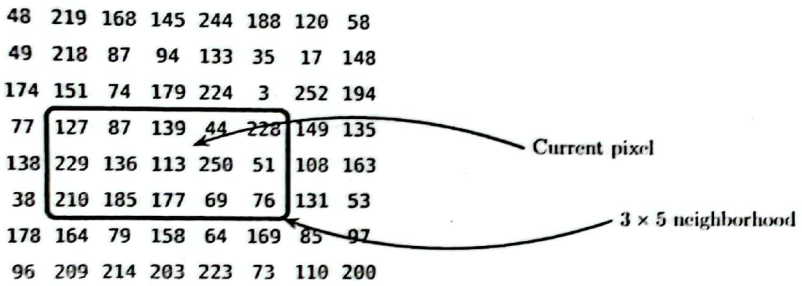


FIGURE 1.15: Pixels, with a neighborhood

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
   - 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 deblurring of speed-camera images

1.6 Image Processing Operations

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. Examples include:

- Sharpening or deblurring 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

Image registration. This involves "matching" distinct images so that they can be compared, or processed together. The initial images must all be joined to share the same coordinate system. In this text, registation as such is not covered, but some tasks that are vital to registration are discussed, for example corner detection.

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 that 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 that 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). In this image, we have only the two colors: white for the edges, and black for the background. See Figure 1.16.

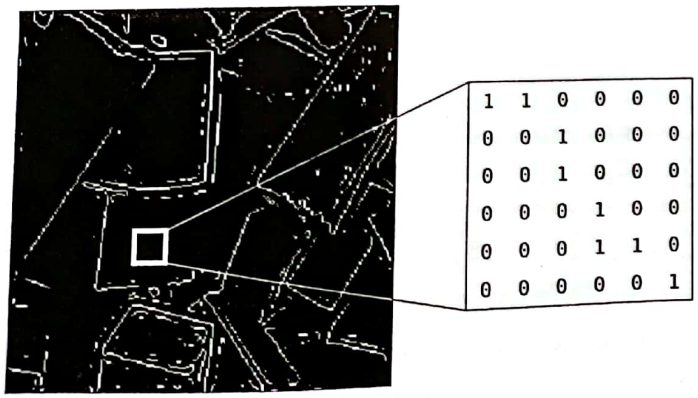


FIGURE 1.16: A binary image

Grayscale. Each pixel is a shade of gray, 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 grayscale 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 gray levels is sufficient for the recognition of most natural objects.

An example is the street scene shown in Figure 1.1, and in Figure 1.17.

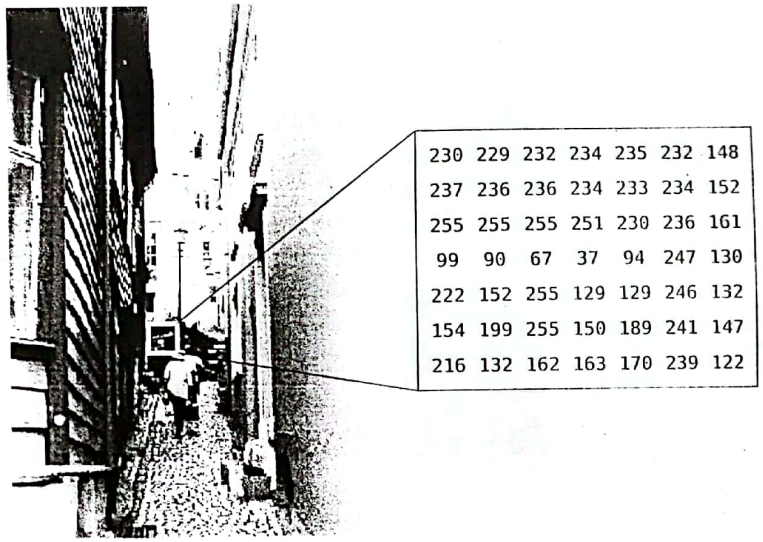


FIGURE 1.17: A grayscale image

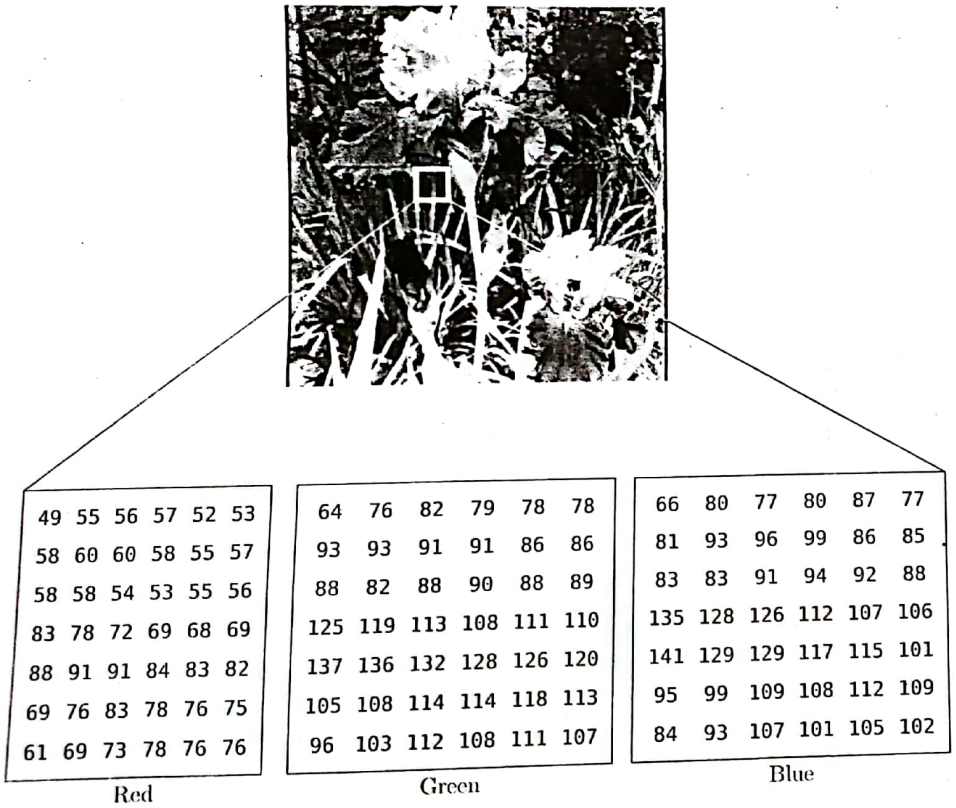
True color, or RGB. Here each pixel has a particular color; that color being described by the amount of red, green, and blue in it. If each of these components has a range 0-255, this gives a total of $255^3 = 16{,}777{,}216$ different possible colors in the image. This is enough colors for any image.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

An example is shown in Figure 1.18.

Indexed. Most color images only have a small subset of the more than sixteen million possible colors. For convenience of storage and file handling, the image has an associated *color map*, or *color palette*, which is simply a list of all the colors used in that image. Each pixel has a value which does not give its color (as for an RGB image), but an *index* to the color in the map.

It is convenient if an image has 256 colors 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 colors or fewer in each image, for precisely this reason.



49	55	56	57	52	53
58	60	60	58	55	57
58	58	54	53	55	56
83	78	72	69	68	69
88	91	91	84	83	82
69	76	83	78	76	75
61	69	73	78	76	76

Red

64	76	82	79	78	78
93	93	91	91	86	86
88	82	88	90	88	89
125	119	113	108	111	110
137	136	132	128	126	120
105	108	114	114	118	113
96	103	112	108	111	107

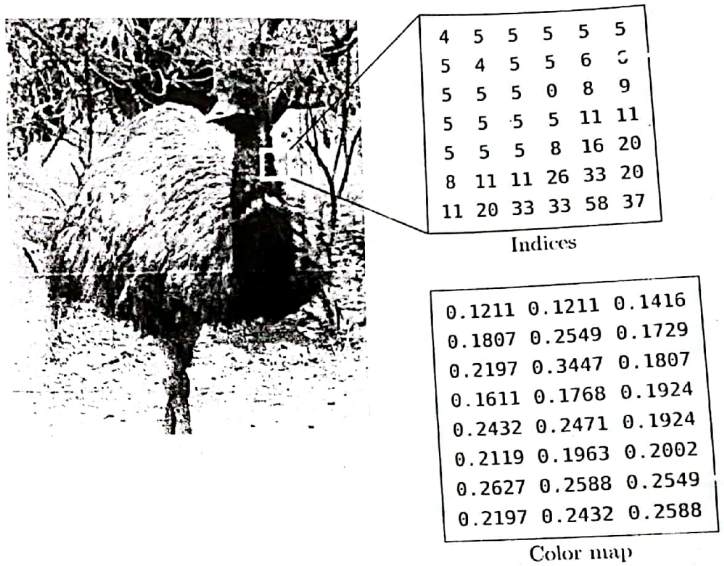
Green

66	80	77	80	87	77
81	93	96	99	86	85
83	83	91	94	92	88
135	128	126	112	107	106
141	129	129	117	115	101
95	99	109	108	112	109
84	93	107	101	105	102

Blue

FIGURE 1.18: **SEE COLOR INSERT** A true color image

Figure 1.19 shows an example. In this image the indices, rather then being the gray values of the pixels, are simply indices into the color map. Without the color map, the image would be very dark and colorless. In the figure, for example, pixels labelled `5` correspond to `0.2627 0.2588 0.2549`, which is a dark grayish color.



4	5	5	5	5	5
5	4	5	5	6	6
5	5	5	0	8	9
5	5	5	5	11	11
5	5	5	8	16	20
8	11	11	26	33	20
11	20	33	33	58	37

Indices

0.1211	0.1211	0.1416
0.1807	0.2549	0.1729
0.2197	0.3447	0.1807
0.1611	0.1768	0.1924
0.2432	0.2471	0.1924
0.2119	0.1963	0.2002
0.2627	0.2588	0.2549
0.2197	0.2432	0.2588

Color map

FIGURE 1.19: **SEE COLOR INSERT** An indexed color image

1.9 Image File Sizes

Image files tend to be large. We shall investigate the amount of information used in different image types 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

$$\begin{aligned} 512 \times 512 \times 1 &= 262{,}144 \\ &= 32{,}768 \text{ bytes} \\ &= 32.768 \text{ Kb} \\ &\approx 0.033 \text{ Mb}. \end{aligned}$$

(Here we use the convention that a kilobyte is 1000 bytes, and a megabyte is one million bytes.)

A grayscale image of the same size requires:

$$\begin{aligned} 512 \times 512 \times 1 &= 262{,}144 \text{ bytes} \\ &= 262.14 \text{ Kb} \\ &\approx 0.262 \text{ Mb.} \end{aligned}$$

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

$$\begin{aligned} 521 \times 512 \times 3 &= 786{,}432 \text{ bytes} \\ &= 786.43 \text{ Kb} \\ &\approx 0.786 \text{ Mb.} \end{aligned}$$

Many images are of course 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 human visual system. Image perception consists of two basic steps:

1. Capturing the image with the eye
2. Recognizing and interpreting the image with the *visual cortex* in the brain

The combination and immense variability of these steps influences the ways in which 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 gray will appear darker if placed on a white background than if it were placed on a black background. That is, we don't perceive gray scales "as they are," but rather as they differ from their surroundings. In Figure 1.20, a gray square is shown on two different backgrounds. Notice how much darker the square appears when it is surrounded by a light gray. However, the two central squares have exactly the same intensity.

2. We may observe non-existent intensities as bars in continuously varying gray 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 gray blob on a dark gray 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.

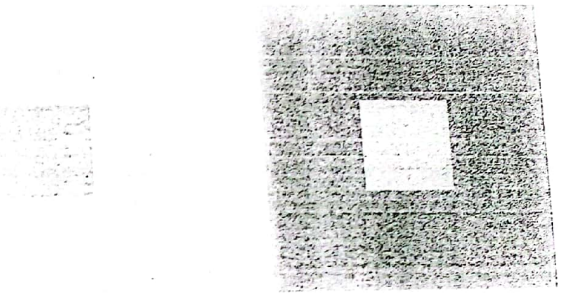


FIGURE 1.20: A gray square on different backgrounds

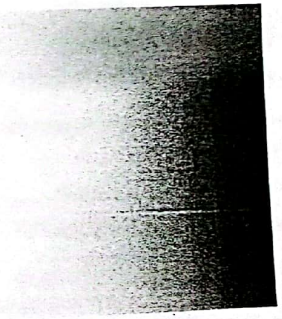


FIGURE 1.21: Continuously varying intensities

Exercises

1. Watch the TV news, and see if you can observe any examples of image processing.
2. If your TV set allows it, turn down the color as far as you can to produce a monochromatic display. How does this affect your viewing? Is there anything that is hard to recognize without color?
3. Look through a collection of old photographs. How can they be enhanced or restored?
4. For each of the following, list five ways in which image processing could be used:
   - Medicine
   - Astronomy
   - Sport
   - Music
   - Agriculture
   - Travel
5. Image processing techniques have become a vital part of the modern movie production process. Next time you watch a film, take note of all the image processing involved.
6. If you have access to a scanner, scan in a photograph, and experiment with all the possible scanner settings.
   (a) What is the smallest sized file you can create which shows all the detail of your photograph?
   (b) What is the smallest sized file you can create in which the major parts of your image are still recognizable?
   (c) How do the color settings affect the output?
7. If you have access to a digital camera, again photograph a fixed scene, using all possible camera settings.
   (a) What is the smallest file you can create?
   (b) How do the light settings affect the output?
8. Suppose you were to scan in a monochromatic photograph, and then print out the result. Then suppose you scanned in the printout, and printed out the result of that, and repeated this a few times. Would you expect any degradation of the image during this process? What aspects of the scanner and printer would minimize degradation?
9. Look up *ultrasonography*. How does it differ from the image acquisition methods discussed in this chapter? What is it used for? If you can, compare an ultrasound image with an x-ray image. How to they differ? In what ways are they similar?
10. If you have access to an image viewing program (other than MATLAB, Octave or Python) on your computer, make a list of the image processing capabilities it offers. Can you find imaging tasks it is unable to do?

Chapter 2

Images Files and File Types

We shall see that matrices can be handled very efficiently in MATLAB, Octave and Python. 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 each system allow us to investigate images and their properties.

2.1 Opening and Viewing Grayscale Images

Suppose you are sitting at your computer and have started your system. You will have a prompt of some sort, and in it you can type:

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

MATLAB/Octave

or from the `io` module of `skimage`

```
In : import skimage.io as io
In : w = io.imread('wombats.png')
```

Python

This takes the gray values of all the pixels in the grayscale image `wombats.png` and puts them all into a matrix `w`. This matrix `w` is now a system variable, and 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. If you are using MATLAB or Octave, end with 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 65,536 elements, we do not really want all its values displayed. Python, however, does not automatically display the results of a computation.
2. The name `wombats.png` is given in quotation marks. Without them, the system would assume that `wombats.png` was the name of a variable, rather than the name of a file.

Now we can display this matrix as a grayscale image. In MATLAB:

```
>> figure,imshow(w),impixelinfo
```

MATLAB

This is really three commands on 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.

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

Pixel info: (c, r) p

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

This is shown in Figure 2.1.

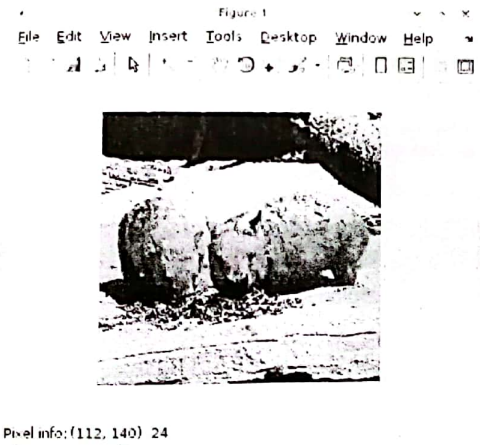


FIGURE 2.1: The wombats image with `impixelinfo`

In Octave, the command is:

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

Octave

and in Python one possible command is:

```
In: io.imshow(w)
```

Python

However, an interactive display is possible using the `ImageViewer` method from the module with the same name:

```
In :  from skimage.viewer import ImageViewer as IV
In :  viewer = IV(w)
In :  viewer.show()
```

Python

This is shown in Figure 2.2.

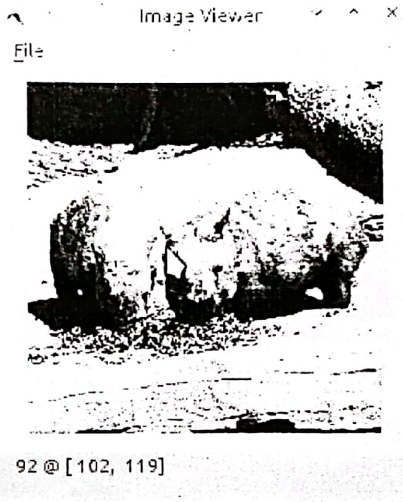


FIGURE 2.2: The wombats image with `ImageViewer`

At present, Octave does not have a built-in method for interactively displaying the pixel indices and value comparable to MATLAB's `impixelinfo` or Python's `ImageViewer`.

If there are no figures open, then in Octave or MATLAB an `imshow` command, or any other command that 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 gray values to a matrix, with the command

```
>> imshow('wombats.png')
```

MATLAB/Octave

or

```
In: io.imshow('wombats.png')
```

Python

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

2.2 RGB Images

As we shall discuss in Chapter 13, we need to define colors in some standard way, usually as a subset of a three-dimensional coordinate system; such a subset is called a *color model*. There are, in fact, a number of different methods for describing color, but for image display and storage, a standard model is RGB, for which we may imagine all the colors sitting inside a "color cube" of side 1 as shown in Figure 2.3. The colors along the black-white diagonal,

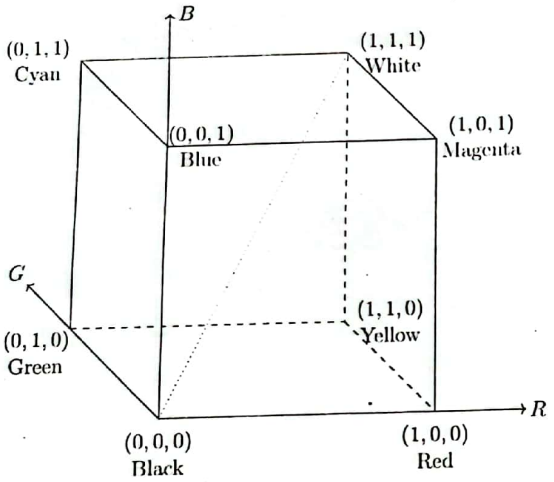


FIGURE 2.3: The color cube for the RGB color model

hown in the diagram as a dotted line, are the points of the space where all the R, G, B alues are equal. They are the different intensities of gray. We may also think of the axes f the color cube as being discretized to integers in the range 0-255.

RGB is the standard for the *display* of colors on computer monitors and TV sets. But it not a very good way of *describing* colors. How, for example, would you define light-brown sing RGB? As we shall see also in Chapter 13, some colors are not realizable with the RGB odel in that they would require negative values of one or two of the RGB components. In neral, 24-bit RGB images are managed in much the same way as grayscale. We can save ie color values to a matrix and view the result:

```
>> b = imread('backyard.png');
>> figure,imshow(b),impixelinfo
```

MATLAB

ote now that the pixel values consist of a list of three values, giving the red, green, and ue components of the color of the given pixel.

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

```
>> size(b)
```

MATLAB/Octave

which returns *three* values: the number of rows, columns, and "pages" of `b`, which is a three-dimensional matrix, also called a *multidimensional array*. Each system can handle arrays of any dimension, and `b` is an example. We can think of `b` as being a stack of three matrices, each of the same size.

In Python:

```
In: b.shape
```

Python

again returns a triplet of values.

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

```
>> b(100,200,2)
ans =

  126
```

MATLAB/Octave

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

```
>> b(100,200,1:3)
ans(:,:,1) =

  114

ans(:,:,2) =

  126

ans(:,:,3) =

   58
```

MATLAB/Octave

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

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

MATLAB/Octave

This can be done similarly in Python:

```
In: print b[99,199,:]
[114 126  58]
```

Python

(Recall that indexing in Python starts at zero, so to obtain the same results as with MATLAB and Octave, the indices need to be one less.)

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

```
>> impixel(b,200,100)
ans =

   114   126    58
```

MATLAB/Octave

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 `impixelinfo` command. This is opposite to the row, column order for matrix indexing. This command also applies to grayscale images:

```
>> impixel(w,100,200)
ans =

   189   189   189
```

MATLAB/Octave

Again, three values are returned, but since `w` is a single two-dimensional matrix, all three values will be the same.

Note that in Octave, the command `impixel` will in fact produce a 3×3 matrix of values; the three columns however will be the same. If we just want the values once, then:

```
impixel(b,200,100)(:,1)'
ans =

   114   126    58
```

Octave

2.3 Indexed Color Images

The command

```
>> figure,imshow('emu.png'),impixelinfo
```

MATLAB/Octave

produces a nice color 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, for example:

Pixel info: (61, 109) ⟨37⟩ [0.58 0.54 0.51]

What is going on here?

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


```
>> em = imread('emu.png');
>> figure,imshow(em),impixelinfo
```

MATLAB/Octave

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

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

```
>> [em,emap] = imread('emu.png');
>> figure,imshow(em,emap),impixelinfo
```

MATLAB/Octave

MATLAB and Octave store the RGB values of an indexed image as values of type `double`, with values between 0 and 1. To obtain RGB values at any value:

```
>> v = em(109,61)

v =

   37
```

MATLAB/Octave

To find the color values at this point, note that the color indexing is done with eight bits, hence the first index value is zero. Thus, index value 37 is in fact the 38th row of the color map:

```
>> >> emap(38,:)

ans =

    0.5801    0.5410    0.5098
```

MATLAB/Octave

Python automatically converts an indexed image to a true-color image as the image file is read:

```
In:  em = io.imread('emu.png')
In:  em.shape
Out: (384, 331, 3)
```

Python

To obtain values at a pixel : :

```
In :  print em[108,60,:]
[148, 138, 130]
In :  np.set_printoptions(precision = 4)
In :  print em[108,60,:].astype(float)/255.0
[ 0.5804 0.5412  0.5098]
```

Python

This last shows how to obtain information comparable to that obtained with MATLAB and Octave.

Information About Your Image

Using MATLAB or Octave, a great deal of information can be obtained with the `imfinfo` function. For example, suppose we take our indexed image `emu.png` from above. The MATLAB and Octave outputs are in fact very slightly different; here is the Octave output:

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

ans =

  scalar structure containing the fields:

    Filename = /home/amca/Images/emu.png
    FileModDate =  2-Jan-2015 15:12:52
    FileSize =  71116
    Format = PNG
    FormatVersion =
    Width =  331
    Height =  384
    BitDepth =  8
    ColorType = indexed
    DelayTime = 0
    DisposalMethod =
    LoopCount = 0
    ByteOrder = undefined
    Gamma = 0
    Chromaticities = [](1x0)
    Comment =
    Quality =  75
    Compression = undefined
    Colormap =
```

Octave

(For saving of space, the colormap, which is given as part of the output, is not listed here.)

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 color type (in this case "indexed").

For comparison, let's look at the output of a true color file (showing only the first few lines of the output), this time using MATLAB:


```
>> imfinfo('backyard.png')

ans =

          Filename: 'backyard.png'
      FileModDate: '02-Jan-2015_05:45:17'
          FileSize: 264103
            Format: 'png'
     FormatVersion: []
             Width: 482
            Height: 643
          BitDepth: 24
         ColorType: 'truecolor'
   FormatSignature: [73 73 42 0]
         ByteOrder: 'little-endian'
    NewSubFileType: 0
     BitsPerSample: [8 8 8]
       Compression: 'JPEG'
```

MATLAB

Now we shall test this function on a binary image, in Octave:

```
>> imfinfo('circles.png')

ans =
  scalar structure containing the fields:

    Filename = /home/amca/Images/circles.png
    FileModDate =  2-Jan-2015 21:37:54
    FileSize =  1268
    Format = PNG
    FormatVersion =
    Width =  256
    Height =  256
    BitDepth =  1
    ColorType = grayscale
```

Octave

In MATLAB and Octave, the value of `ColorType` would be `GrayScale`.

What is going on here? We have a binary image, and yet the color type is given as either "indexed" or "grayscale." The fact is that the `imfinfo` command in both MATLAB nor Octave has no value such as "Binary", "Logical" or "Boolean" for `ColorType`. A binary image is just considered to be a special case of a grayscale image which has only two intensities. However, we can see that `circles.png` is a binary image since the number of bits per pixel is only one.

In Chapter 3 we shall see that a binary image *matrix*, as distinct from a binary image *file*, can have the numerical data type `Logical`.

To obtain image information in Python, we need to invoke one of the libraries that supports the metadata from an image. For TIFF and JPEG images, such data is known as EXIF data, and the Python `exifread` library may be used:

```
In :  import exifread
In :  f = open('backyard.tif')
In :  tags = exifread.process_file(f,details=False)
In :  for x in tags:
...:     print x.ljust(32),":",tags[x]

Image Orientation                : Horizontal (normal)
Image Compression                : JPEG
Image FillOrder                  : 1
Image ImageLength                : 643
Image PhotometricInterpretation  : 2
Image ImageWidth                 : 482
Image DocumentName               : /home/amca/Images/backyard.tif
Image BitsPerSample              : [8, 8, 8]
Image XResolution                : 72
Image YResolution                : 72
Image SamplesPerPixel            : 3
```

Python

Some of the output is not shown here. For images of other types, it is easiest to invoke a system call to something like "ExifTool"[1]:

```
In :  ! exiftool cassowary.png
ExifTool Version Number         : 9.46
File Name                       : cassowary.png
File Size                       : 21 MB
MIME Type                       : image/png
Image Width                     : 4000
Image Height                    : 2248
Bit Depth                       : 8
Color Type                      : RGB
Compression                     : Deflate/Inflate
Background Color                : 255 255 255
Pixels Per Unit X               : 7086
Pixels Per Unit Y               : 7086
Pixel Units                     : Meters
Exif Byte Order                 : Little-endian (Intel, II)
Orientation                     : Horizontal (normal)
X Resolution                    : 180
Y Resolution                    : 180
Resolution Unit                 : inches
```

Python

Again most of the information is not shown.

2.4 Numeric Types and Conversions

Elements in an array representing an image may have a number of different numeric data types; the most common are listed in Table 2.1. MATLAB and Octave use "double"

[1] Available from `http://www.sno.phy.queensu.ca/~phil/exiftool/`

Data type	Description	Range
`logical`	Boolean	0 or 1
`int8`	8-bit integer	−128 — 127
`uint8`	8-bit unsigned integer	0 — 255
`int16`	16-bit integer	−32768 — 32767
`uint16`	16-bit unsigned integer	0 — 65535
`double` or `float`	Double precision real number	Machine specific

TABLE 2.1: Some numeric data types

and Python uses "float." There are others, but those listed will be sufficient for all our work with images. Notice that, strictly speaking, `logical` is not a numeric type. However, we shall see that some image use this particular type. These data types are also functions, and we can convert from one type to another. For example:

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

b =

   23

>> whos a b
  Name      Size            Bytes  Class

  a         1x1                 8  double
  b         1x1                 1  uint8
```

MATLAB/Octave

Similarly in Python:

```
In :  a = 23
In :  b = uint8(a)
In :  %whos int uint8
Variable   Type     Data/Info
-----------------------------
a          int      23
b          uint8    23
```

Python

Note here that `%whos` is in fact a magic function in the IPython shell. If you are using another interface, this function will not be available.

Even though the variables `a` and `b` have the same numeric value, they are of different data types.

A grayscale image may consist of pixels whose values are of data type `uint8`. These images are thus reasonably efficient in terms of storage space, because each pixel requires only one byte.

We can convert images from one image type to another. Table 2.2 lists all of MATLAB's functions for converting between different image types. Note that there is no `gray2rgb`

Function	Use	Format
ind2gray	Indexed to grayscale	y=ind2gray(x,map);
gray2ind	Grayscale to indexed	[y,map]=gray2ind(x);
rgb2gray	RGB to grayscale	y=rgb2gray(x);
rgb2ind	RGB to indexed	[y,map]=rgb2ind;
ind2rgb	Indexed to RGB	y=ind2rgb(x,map);

TABLE 2.2: Converting images in MATLAB and Octave

function. But a gray image can be turned into an RGB image where all the R, G, and B matrices are equal, simply by stacking the grayscale array three deep. This can done using several different methods, and will be investigated in Chapter 13.

In Python, there are the `float` and `uint8` functions, but in the skimage library there are methods from the `util` module; these are listed in Table 2.3.

Function	Use	Format
img_as_bool	Convert to boolean	y=sk.util.img_as_bool(x)
img_as_float	Convert to 64-bit floating point	y=sk.util.img_as_bool(x)
img_as_bool	Convert to 16-bit integer	y=sk.util.img_as_int(x)
img_as_bool	Convert to 8-bit unsigned integer	y=sk.util.img_as_ubyte(x)
img_as_bool	Convert to 16-bit unsigned integer	y=sk.util.img_as_uint(x)}

TABLE 2.3: Converting images in Python

2.5 Image Files and Formats

We have seen in Section 1.8 that images may be classified into four distinct types: binary, grayscale, colored, and indexed. In this section, we consider some of the different image file formats, their advantages and disadvantages. You can use MATLAB, Octave or Python for image processing very happily without ever really knowing the difference between GIF, TIFF, PNG, and all the other formats. However, some knowledge of the different graphics formats can be extremely useful to be able to make a reasoned decision as to which file type to use and when.

There are a great many different formats for storing image data. Some have been designed to fulfill a particular need (for example, to transmit image data over a network); others have been designed around a particular operations system or environment.

As well as the gray values or color values of the pixels, an image file will contain some *header information*. This will, at the very least, include the size of the image in pixels (height and width); it may also include the color map, compression used, and a description of the image. Each system recognizes many standard formats, and can read image data from them and write image data to them. The examples above have mostly been images with extension `.png`, indicating that they are PNG (Portable Network Graphics) images. This is a particularly general format, as it allows for binary, grayscale, RGB, and indexed color images, as well as allowing different amounts of transparency. PNG is thus a good format for transmitting images between different operating systems and environments. The

PNG standard only allows one image per file. Other formats, for example TIFF, allow for more than one image per file; a particular image from such a multi-image file can be read into MATLAB by using an optional numeric argument to the `imread` function.

Each system can read images from a large variety of file types, and save arrays to images of those types. General image file types, all of which are handled by each system, include:

JPEG Images created using the Joint Photographics Experts Group compression method. We will discuss this more in Chapter 14.

TIFF Tagged Image File Format: a very general format which supports different compression methods, multiple images per file, and binary, grayscale, truecolor and indexed images.

GIF Graphics Interchange Format: This venerable format was designed for data transfer. It is still popular and well supported, but is somewhat restricted in the image types it can handle.

BMP Microsoft Bitmap: This format has become very popular, with its use by Microsoft operating systems.

PNG Portable Network Graphics: This is designed to overcome some of the disadvantages of GIF, and to become a replacement for GIF.

We will discuss some of these briefly below.

A Hexadecimal Dump Function

To explore binary files, we need a simple function that will enable us to list the contents of the file as hexadecimal values. If we try to list the contents of a binary file directly to the screen, we will see masses of garbage. The trouble is that any file printing method will interpret the file's contents as ASCII characters, and this means that most of these will either be unprintable or nonsensical as values.

A simple hexadecimal dump function, called "hexdump," is given at the end of the chapter. It is called with the name of the file, and the number of bytes to be listed:

```
>> hexdump('backyard.tif',64)
000010  4949 2a00 4c01 0400 ffd8 ffdb 0043 0008  II*.L........C..
000020  0606 0706 0508 0707 0709 0908 0a0c 140d  ................
000030  0c0b 0b0c 1912 130f 141d 1a1f 1e1d 1a1c  ................
000040  1c20 242e 2720 222c 231c 1c28 3729 2c30  . $.' ",#..(7),0
```

MATLAB/Octave

and in Python:

```
In :  hexdump('backyard.tif',64)
000000:  4949 2a00 4c01 0400 ffd8 ffdb 0043 0008  |II*.L........C..|
000010:  0606 0706 0508 0707 0709 0908 0a0c 140d  |................|
000020:  0c0b 0b0c 1912 130f 141d 1a1f 1e1d 1a1c  |................|
000030:  1c20 242e 2720 222c 231c 1c28 3729 2c30  |. $.'␣",#..(7),0|
```

Python

The result is a listing of the bytes as hexadecimal characters in three columns: the first gives the hexadecimal value of the index of the first byte in that row, the second column consists of the bytes themselves, and the third column lists those that are representable as ASCII text.

Vector versus Raster Images

We may store image information in two different ways: as a collection of lines or vectors, or as a collection of dots. We refer to the former as *vector* images; the latter as *raster* images. The great advantage of vector images is that they can be magnified to any desired size without losing any sharpness. The disadvantage is that are not very good for the representation of natural scenes, in which lines may be scarce. The standard vector format is Adobe PostScript; this is an international standard for page layout. PostScript is the format of choice for images consisting mostly of lines and mathematically described curves: architectural and industrial plans, font information, and mathematical figures. The reference manual [21] provides all necessary information about PostScript.

The great bulk of image file formats store images as raster information; that is, as a list of the gray or color intensities of each pixel. Images captured by digital means–digital cameras or scanners–will be stored in raster format.

A Simple Raster Format

As well as containing all pixel information, an image file must contain some *header information*; this must include the size of the image, but may also include some documentation, a color map, and compression used. To show the workings of a raster image file, we shall briefly describe the ASCII PGM format. This was designed to be a generic format used for conversion between other formats. Thus, to create conversion routines between, say, 40 different formats, rather than have $40 \times 39 = 1560$ different conversion routines, all we need is the $40 \times 2 = 80$ conversion routines between the formats and PGM.

```
P2
# CREATOR: The GIMP's PNM Filter Version 1.0
256 256
255
 41  53  53  53  53  49  49  53  53  56  56  49  41  46  53  53  53
 53  41  46  56  56  56  53  53  46  53  41  41  53  56  49  39  46
```

FIGURE 2.4: The start of a PGM file

Figure 2.4 shows the beginning of a PGM file. The file begins with P2; this indicates that the file is an ASCII PGM file. The next line gives some information about the file: any line beginning with a hash symbol is treated as a comment line. The next line gives the number of columns and rows, and the following line gives the number of grayscales. Finally we have all the pixel information, starting at the top left of the image, and working across and down. Spaces and carriage returns are delimiters, so the pixel information could be written in one very long line or one very long column.

Note that this format has the advantage of being very easy to write to and to read from; it has the disadvantage of producing very large files. Some space can be saved by using "raw" PGM; the only difference is that the header number is P5, and the pixel values are stored one per byte. There are corresponding formats for binary and colored images (PBM and PPM, respectively); colored images are stored as three matrices; one for each of red, green, and blue; either as ASCII or raw. The format does not support color maps.

Binary, grayscale, or color images using these formats are collectively called PNM images. MATLAB and Octave can read PNM images natively; Python can't, but it is not hard to write a file to read a PNM image to an ndarray.

Microsoft BMP

The Microsoft Windows BMP image format is a fairly simple example of a binary image format, noting that here binary means non-ascii, as opposed to Boolean. Like the PGM format above, it consists of a header followed by the image information. The header is divided into two parts: the first 14 bytes (bytes number 0 to 13), is the "File Header", and the following 40 bytes (bytes 14 to 53), is the "Information Header". The header is arranged as follows:

Bytes	Information	Description
0–1	Signature	"BM" in ASCII = 42 4D in hexadecimal.
2–5	FileSize	The size of the file in bytes.
6–9	Reserved	All zeros.
10–13	DataOffset	File offset to the raster data.
14–17	Size	Size of the information header = 40 bytes.
18–21	Width	Width of the image in pixels.
22–25	Height	Height of the image in pixels.
26–27	Planes	Number of image planes (= 1).
28–29	BitCount	Number of bits per pixel: 1: Binary images; two colors 4: $2^4 = 16$ colors (indexed) 8: $2^8 = 256$ colors (indexed) 16: 16-bit RGB; $2^{16} = 65{,}536$ colors 24: 24-bit RGB; $2^{24} = 17{,}222{,}216$ colors
30–33	Compression	Type of compression used: 0: no compression (most common) 1: 8-bit RLE encoding (rarely used) 2: 4-bit RLE encoding (rarely used)
34–37	ImageSize	Size of the image. If compression is 0, then this value may be 0.
38–41	HorizontalRes	The horizontal resolution in pixels per meter.
42–45	VerticalRes	The vertical resolution in pixels per meter.
46–49	ColorsUsed	The number of colors used in the image. If this value is zero, then the number of colors is the maximum obtainable with the bits per pixel, that is 2^{BitCount}.
50–53	ImportantColors	The number of important colors in the image. If all the colors are important, then this value is set to zero.

After the header comes the Color Table, which is only used if BitCount is less than or equal to 8. The total number of bytes used here is $4 \times$ ColorsUsed. This format uses the Intel "least endian" convention for bytes ordering, where in each word of four bytes the *least* valued byte comes *first*. To see an example of this, consider a simple example:

```
>> hexdump('backyard.bmp',64)
000000  424d 8235 0e00 0000 0000 8a00 0000 7c00  BM.5.........|.
000010  0000 e201 0000 8302 0000 0100 1800 0000  ................
000020  0000 f834 0e00 120b 0000 120b 0000 0000  ...4............
000030  0000 0000 0000 0000 ff00 00ff 0000 ff00  ................
```

MATLAB/Octave

The image width is given by bytes 18-21; they are in the second row:

```
e201 0000
```

To find the actual width; we re-order these bytes back to front:

```
0000 01e2
```

Now we can convert to decimal:

$$(1 \times 16^2) + (14 \times 16^1) + (2 \times 16^0) = 482$$

which is the image width in pixels. We can do the same thing with the image height; bits 22-25:

```
8302 0000
```

Re-ordering and converting to hexadecimal:

$$(2 \times 16^2) + (8 \times 16^1) + (3 \times 16^0) = 16 + 15 = 643.$$

Recall that hexadecimal symbols a, b, c, d, e, f have decimal values 10 to 15, respectively.

GIF and PNG

Compuserve GIF (pronounced "jif") is a venerable image format that was first proposed in the late 1980s as a means for distributing images over networks. Like PGM, it is a raster format, but it has the following properties:

1. Colors are stored using a color map; the GIF specification allows a maximum of 256 colors per image.
2. GIF doesn't allow for binary or grayscale images; except as can be produced with red, green, and blue values.
3. The pixel data is compressed using LZW (Lempel-Ziv-Welch) compression. This works by constructing a "codebook" of the data: the first time a pattern is found, it is placed in the codebook; subsequent times the encoder will output the code for that pattern. LZW compression can be used on any data; until relatively recently, it was a patented algorithm, and legal use required a license from Unisys. LZW is described in Chapter 14.
4. The GIF format allows for multiple images per file; this aspect can be used to create "animated GIFs."

GIF file will contain a header including the image size (in pixels), the color map, the color depth (number of bits per pixel), a flag indicating whether the color map is ordered, the color map size.

The GIF format is greatly used; it has become one of the standard formats supported by the World Wide Web, and by the Java programming language. Full descriptions of the GIF format can be found in [5] or [26].

The PNG (pronounced "ping") format has been more recently designed to replace GIF, and to overcome some of its disadvantages. Specifically, PNG was not to rely on any patented algorithms, and it was to support more image types than GIF. PNG supports grayscale, true-color, and indexed images. Moreover, its compression utility, zlib, *always* results in genuine compression. This is not the case with LZW compression; it can happen that the result of an LZW compression is larger than the original data. PNG also includes support for *alpha channels*, which are ways of associating variable transparencies with an image, and *gamma correction*, which associates different numbers with different computer display systems, to ensure that a given image will appear the same independently of the system.

PNG is described in detail by [38]. PNG is certainly to be preferred to GIF; it is now well supported by every system.

A PNG file consists of what are called "chunks"; each is referenced by a four-letter name. Four of the chunks must be present: IHDR, giving the image dimensions and bit depth, PLTE is the color palette, IDAT contains the image data, IEND marks the end. These four chunks are called "Critical Chunks." Other chunks, known as "Ancillary Chunks," may or may not be present; they include pHYS: the pixel size and aspect ratio; bKGD the background color; sRGB, indicating the use of the standard RGB color space; gAMA, specifying gamma; cHRM, which describes the primary chromaticites and the white point; as well as others.

For example, here are the hexadecimal dumps of the first few bytes of three PNG images, first `cameraman.png`:

```
000000:  8950 4e47 0d0a 1a0a 0000 000d 4948 4452  |.PNG........IHDR|
000010:  0000 0100 0000 0100 0800 0000 0079 19f7  |.............y..|
000020:  ba00 0000 0970 4859 7300 000b 1200 000b  |.....pHYs.......|
000030:  1201 d2dd 7efc 0000 8000 4944 4154 78da  |....~.....IDATx.|
```

Next `iguana.png`:

```
000000:  8950 4e47 0d0a 1a0a 0000 000d 4948 4452  |.PNG........IHDR|
000010:  0000 0267 0000 017d 0800 0000 0064 7c90  |...g...}.....d|.|
000020:  ad00 0000 0262 4b47 4400 ff87 8fcc bf00  |.....bKGD.......|
000030:  0000 0970 4859 7300 000b 1200 000b 1201  |...pHYs.........|
000040:  d2dd 7efc 0000 8000 4944 4154 78da 3cfd  |..~.....IDATx.<.|
```

Finally `backyard.png`:

```
000000:  8950 4e47 0d0a 1a0a 0000 000d 4948 4452  |.PNG........IHDR|
000010:  0000 01e2 0000 0283 0802 0000 0049 cb9e  |.............I..|
000020:  9600 0000 0467 414d 4100 00b1 8f0b fc61  |.....gAMA......a|
000030:  0500 0000 0173 5247 4200 aece 1ce9 0000  |.....sRGB.......|
000040:  0020 6348 524d 0000 7a26 0000 8084 0000  |. cHRM..z&......|
```

Each chunk consists of four bytes giving the length of the chunk's data, four bytes giving its type, then the data of the chunk, and finally four bytes of a checksum.

For example, consider the last file. The IHDR chunk has its initial four bytes of length `0000 000d`, which has decimal value 13, and the four bytes of its name `IHDR`. The 13 bytes of data start with four bytes each for width and height, followed by one byte each for bit depth, color type, compression method, filter method, and interlace method. So, for the example given:

Object	Bytes	Decimal value	Meaning
Width	`0000 01e2`	482	
Height	`0000 0283`	683	
Bit depth	`08`	8	Bits per sample or per palette index
Color type	`02`	2	RGB color space used
Compression	`00`	0	No compression
Interlacing	`00`	0	No interlacing

Note that the "bit depth" refers not to bit depth per pixel, but bit depth (in this case) for each color plane.

As an example of an ancillary chunk, consider pHYS, which is always nine bytes. From the cameraman image we can read:

Object	Bytes	Decimal value	Meaning
Pixels per unit, X axis	`00 000b 12`	2834	
Pixels per unit, Y axis	`00 000b 12`	2834	
Unit specifier	`01`	1	Unit is meter.

If the unit specifier was 0, then the unit is not specified. The cameraman image is thus expected to have 2834 pixels per meter in each direction, or 71.984 pixels per inch.

JPEG

The compression methods used by GIF and PNG are *lossless*: the original information can be recovered completely. The JPEG (Joint Photographics Experts Group) algorithm uses *lossy* compression, in which not all the original data can be recovered. Such methods result in much higher compression rates, and JPEG images are in general much smaller than GIF or PNG images. Compression of JPEG images works by breaking the image into 8×8 blocks, applying the discrete cosine transform (DCT) to each block, and removing small values. JPEG images are best used for the representation of natural scenes, in which case they are to be preferred.

For data with any legal significance, or scientific data, JPEG is less suitable, for the very reason that not all data is preserved. However, the mechanics of the JPEG transform ensures that a JPEG image, when restored from its compression routine, will *look* the same as the original image. The differences are in general too small to be visible to the human eye. JPEG images are thus excellent for display.

We shall investigate the JPEG algorithm in Section 14.5. More detailed accounts can be found in [13] and [37].

A JPEG image then contains the compression data with a small header providing the image size and file identification. We can see a header by using our `hexdump` function applied to the `greentreefrog.jpg` image:

```
000000:  ffd8 ffe0 0010 4a46 4946 0001 0101 00b4  |......JFIF......|
000010:  00b4 0000 ffe1 35fe 4578 6966 0000 4949  |......5.Exif..II|
000020:  2a00 0800 0000 0a00 0e01 0200 2000 0000  |*........... ...|
000030:  8600 0000 0f01 0200 0600 0000 a600 0000  |................|
```


An image file containing JPEG compressed data is usually just called a "JPEG image." But this is not quite correct; such an image should be called a "JFIF image" where JFIF stands for "JPEG File Interchange Format." The JFIF definition allows for the file to contain a *thumbnail* version of the image; this is reflected in the header information:

Bytes	Information	Description
0–1	Start of image marker	Always `ffd8`.
2–3	Application marker	Always `ffe0`.
4–5	Length of segment	
6–10	`JFIF\ 0`	ASCII "JFIF."
11–12	JFIF version	In our example above `01 01`, or version 1.1.
13	Units	Values are 0: Arbitrary units; 1: pixels/in; 2: pixels/cm.
14–15	Horizontal pixel density	
16–17	Vertical pixel density	
18	Thumbnail width	If this is 0, there is no thumbnail.
19	Thumbnail height	If this is 0, there is no thumbnail.

In this image, both horizontal and vertical pixel densities have hexadecimal value `b4`, or decimal value 180. After this would come the thumbnail information (stored as 24-bit RGB values), and further information required for decompressing the image data. See [5] or [26] for further information.

TIFF

The *Tagged Image File Format*, or TIFF, is one of the most comprehensive image formats. It can store multiple images per file. It allows different compression routines (none at all, LZW, JPEG, Huffman, RLE); different byte orderings (little-endian, as used in BMP, or big-endian, in which the bytes retain their order within words); it allows binary, grayscale, truecolor or indexed images; it allows for opacity or transparency.

For that reason, it requires skillful programming to write image reading software that will read all possible TIFF images. But TIFF is an excellent format for data exchange.

The TIFF header is in fact very simple; it consists of just eight bytes:

Bytes	Information	Description
0–1	Byte order	Either `4d4d`: ASCII "MM" for big-endian, or `49 49`: ASCII "II" for little endian.
2–3	TIFF version	Always `002a` or `2a00` (depending on the byte order) = 42.
4–8	Image offset	Pointer to the position in the file of the data for the first image.

We can see this with looking at the `newborn.tif` image:

```
000000:  4949 2a00 e001 0100 327c 5b2d 2319 0e15  |II*.....2|[-#...|
000010:  100e 0d0f 100f 0e10 1111 0e12 1312 1017  |................|
000020:  101d 708e 99a0 aeb5 bbba c2c6 c6cb d3d0  |..p.............|
000030:  d2d1 cadb dede e1e5 e6df e4e9 feeb 0bed  |................|
```

This particular image uses the little-endian byte ordering. The first image in this file (w
is in fact the only image) begins at byte `e001 0100`.

Since this is a little-endian file, we reverse the order of the bytes: `00 01 01 e0`; this works out to 66016.

As well as the previously cited references, Baxes [3] provides a good introduction to TIFF, and the formal specification [1] is also available.

Writing Image Files

In Python, an image matrix may be written to an image file with the `imsave` function; its simplest usage is

```
In :  io.imsave(x,'filename.abc')
```

Python

where `abc` may be any of the image file types recognized by the system: `gif`, `jpg`, `tif`, `bmp`, for example.

MATLAB and Octave have an `imwrite` function:

```
>> imwrite(x,'filename.abc')
```

MATLAB/Octave

An indexed image can be written by including its colormap as well:

```
>> imwrite(x,map,'filename.abc')
```

MATLAB/Octave

If we wish to be doubly sure that the correct image file format is to be used, a string representing the format can be included; in MATLAB (but at the time of writing not in Octave), a list of supported formats can be obtained with the `imformats` function. Octave's image handling makes extensive use of the ImageMagick library;[2] these however support over 100 different formats, as long as your system is set up with the appropriate format handling libraries.

For example, given the matrix `c` representing the cameraman image, it can be written to a PNG image with

```
In :  io.imsave(c,'cameraman.png')
```

Python

or with

```
>> imwrite(c,'cameraman.png')
```

MATLAB/Octave

2.6 Programs

Here are the programs for the hexadecimal dump. First in MATLAB/Octave:

[2] http://www.imagemagick.org/


```
function hexdump(filenm, n)
% hexdump(filenm, n)
% Print the first n bytes of a file in he        nd ASCII.
fid = fopen(filenm, 'r');
if (fid<0) disp(['Error opening ',filenm]       eturn; end;
nread = 0;
while (nread < n)
    width = 16;
    [A,count] = fread(fid, width, 'uchar')
    nread = nread + count;
    if (nread>n) count = count - (nread-n);      = A(1:count); end;
    hexstring = repmat(' ',1,width*2);
    hexstring(1:2*count) = sprintf('%02x',A
    hexdisp = repmat(' ',1,40);
    for i = 1:idivide(count,2)
        hexdisp(5*i-4:5*i-1) = hexstring(4*i     *1);
    end
    ascstring = repmat('.',1, count);
    idx = find(double(A)>=32 & double(A)<=126
    ascstring(idx) = char(A(idx));
    fprintf('%s:  %s |%s|\n', num2str(dec2hex     ad-16,6)), hexdisp,
        ascstring);
end;
fclose(fid);
```

MATLAB/Octave

Now Python:

```
def hexdump(file,num):
    from math import ceil
    f = open(file)
    h = f.read(num)
    hl = [hex(ord(c))[2:].zfill(2) for c in h] #     e bytes as 2
        character hex strings
    hl2 = hl +(-len(hl)%16)*['  ']  # fill up to      iple of width 16
    asc = ['.']*num
    for i in range(num):            # creates ascii     es (if printable)
        of bytes
        ii = int(hl[i],16)
        if ii>=32 and ii<=126:
            asc[i] = chr(ii)
    asc = asc + (-len(hl)%16)*[' '] # fill up to a m     le of width 16
    for n in range(int(ceil(num/16.0))):
        print hex(n*16)[2:].zfill(6)+':  ',
        print ' '.join([hl2[16*n+2*i]+hl2[16*n+2*i+1]     i in range(8)]),
        print ' |'+''.join([asc[16*n+i] for i in range     )+'|'
```

Python

Exercises

1. Dig around in your system and see if you can find what image formats are supported.

2. If you are using MATLAB or Octave, read in an RGB image and save it as an indexed image.

3. If you are using Python, several images are distributed with the `skimage` library: enter `from skimage import data` at the prompt, and then open up some of the images. This can be done, for example, with

```
In :  x = data.clock()
In :  io.imshow(x)
```

Python

   List the data images and their types (binary, grayscale, color).

4. If you are using MATLAB, there are a lot of sample images distributed with the Image Processing Toolbox, in the `imdata` subdirectory. Use your file browser to enter that directory and check out the images.

   For each of the images you choose:

   (a) Determine its type (binary, grayscale, true color or indexed color)

   (b) Determine its size (in pixels)

   (c) Give a brief description of the picture (what it looks like; what it seems to be a picture of)

5. Pick a grayscale image, say `cameraman.png` or `wombats.png`. Using the `imwrite` function, write it to files of type JPEG, PNG, and BMP.

   What are the sizes of those files?

6. Repeat the above question with

   (a) A binary image,

   (b) An indexed color image,

   (c) A true color image.

7. The following shows the hexadecimal dump of a PNG file:

```
000000  8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
000010  0000 012c 0000 00f6 0800 0000 0049 c4e5  ...,.........I..
000020  5400 0000 0774 494d 4507 d209 1314 1f0c  T....tIME.......
000030  035d c49d 0000 0027 7445 5874 436f 7079  .]....'tEXtCopy
```

   Determine the height and width of this image (in pixels), and whether it is a grayscale or color image.

8. Repeat the previous question with this BMP image:

```
000000  424d 3603 0000 0000 0000 3600 0000 2800  BM6.......6...(.
000010  0000 1000 0000 1000 0000 0100 1800 0000  ................
000020  0000 0003 0000 c40e 0000 c40e 0000 0000  ................
000030  0000 0000 0000 e1f5 ffe1 f5ff e1f5 ffe1  ................
```

Chapter 3

Image Display

3.1 Introduction

We have touched briefly in Chapter 2 on image display. In this chapter, we investigate this matter in more detail. We look more deeply at the use of the image display functions in our systems, and show how spatial resolution and quantization can affect the display and appearance of an image. In particular, we look at image quality, and how that may be affected by various image attributes. Quality is of course a highly subjective matter: no two people will agree precisely as to the quality of different images. However, for human vision in general, images are preferred to be sharp and detailed. This is a consequence of two properties of an image: its spatial resolution, and its quantization.

An image may be represented as a matrix of the gray values of its pixels. The problem here is to display that matrix on the computer screen. There are many factors that 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 color depth: the monitor may only be able to display 24-bit color 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 setup is as optimal as possible, and the monitor is able to accurately reproduce the necessary gray values or colors in any image.

3.2 The imshow Function

Grayscale Images

MATLAB and Octave can display a grayscale image with `imshow` if the image matrix `x` is of type `uint8`, or of type `double` with all values in the range $0.0 - 1.0$. They can also display images of type `uint16`, but for Octave this requires that the underlying ImageMagick library is compiled to handle 16-bit images. Then

```
imshow(x)
```

will display `x` as an image.

This means that any image processing that produces an output of type `double` must either be scaled to the appropriate range or converted to type `uint8` before display.

If scaling is not done, then `imshow` will treat all values greater than 1 as white, and all values lower than 0 as black. Suppose we take an image and convert it to type `double`:

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

MATLAB/Octave

The results are shown in Figure 3.1.

(a) The original image (b) After conversion to type `double`

FIGURE 3.1: An attempt at data type conversion

As you can see, Figure 3.1(b) doesn't look much like the original picture at all! This is because any original values that were greater than 1 are now being displayed as white. 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)
```

MATLAB/Octave

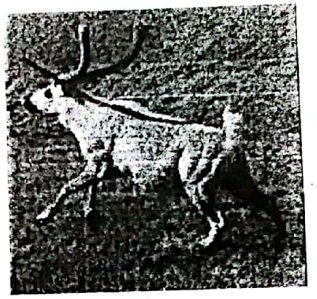
and the result will be the caribou image as shown in Figure 3.1(a).

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

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

MATLAB/Octave

are shown in Figure 3.2.



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

FIGURE 3.2: Scaling by dividing an image matrix by a scalar

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-gray. 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.

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)
```

MATLAB/Octave

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);
```

MATLAB/Octave

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

Python is less prescriptive about values in an image. The `io.imshow` method will automatically scale the image for display. So, for example:

```
In :  c = io.imread('caribou.png')
In :  cd1 = c.astype(float)
In :  cd2 = util.img_as_float(c)
```

Python

all produce images that are equally displayable with `io.imshow`. Note that the data types and values are all different:

```
In :  c.dtype, cd1.dtype, cd2.dtype
Out: (dtype('uint8'), dtype('float64'), dtype('float64'))
```

Python

To see the numeric values, just check the minima and maxima of each array:

```
In : c.min(), c.max()
Out: (21, 254)
In : cd1.min(), cd1.max()
Out: (21.0, 254.0)
In : cd2.min(), cd2.max()
Out: (0.082352941176470587, 0.99607843137254903)
```

Python

This means that multiplying and dividing an image by a fixed value will not affect the display, as the result will be scaled. In order to obtain the effects of darkening and lightening, the default scaling, which uses the maximum and minimum values of the image matrix, can be adjusted by the use of two parameters: `vmin`, which gives the minimum gray level to be viewed, and `vmax`, which gives the maximum gray level.

For example, to obtain a dark caribou image:

```
In :  io.imshow(cd1/2,vmin=0,vmax=255)
```

Python

and to obtain a light, washed out image:

```
In :  io.imshow(cd1*2,vmin=0,vmax=255)
```

Python

Without the `vmin` and `vmax` values given, the image would be automatically scaled to look like the original.

Note that MATLAB/Octave and Python differ in their handling of arithmetic on uint8 numbers. For example:

```
>> a = uint8([0 80;160 240]);
>> (a-10)*2
ans =

   0   80
 160  255
```

MATLAB/Octave

MATLAB and Octave *clip* the output, so that values greater than 255 are set equal to 255, and values less than 0 are set equal to zero. But in Python the result is different:

```
In :  a = array([[0,80],[160,240]]).astype(uint8)
In :  (a-10)*2
Out:
array([[236, 140],
       [ 44, 204]], dtype=uint8)
```

Python

Python works *modulo* 256: numbers outside the range 0–255 are divided by 256 and the remainder given.

Binary images

Recall that a binary image will have only two values: 0 and 1. MATLAB and Octave do not have a `binary` data type as such, but they do 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 that provide a yes/no answer. For example, suppose we take the caribou matrix and create a new matrix with

```
>> cl = c>120;
```

MATLAB/Octave

(we will see more of this type of operation in Chapter 4.) If we now check all of our variables with `whos`, the output will include the line:

```
>> whos c cl
  Name        Size              Bytes  Class     Attributes

  c         256x256             65536  uint8
  cl        256x256             65536  logical
```

MATLAB

The Octave output is slightly different:

```
  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       c           256x256                  65536  uint8
       cl          256x256                  65536  logical
```

Octave

This means that the command

```
>> imshow(cl)
```

MATLAB/Octave

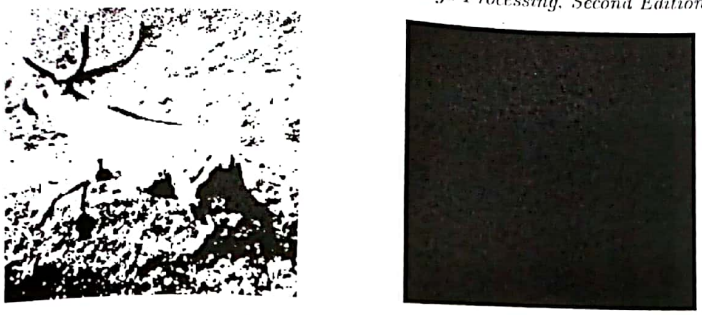
will display the matrix as a binary image; the result is shown in Figure 3.3.

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

```
>> clu = uint8(cl);
```

MATLAB/Octave

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



(a) The caribou image turned binary (b) After conversion to type `uint8`

FIGURE 3.3: Making the image binary

```
clu      256x256        65536  uint8
```

MATLAB/Octave

If we now try to display this matrix with `imshow`, we obtain the result shown in Figure 3.3(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 gray which is indistinguishable from black.

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

```
>> imshow(logical(clu))
```

MATLAB/Octave

or simply convert to type `double`:

```
>> imshow(double(clu))
```

MATLAB/Octave

Both these commands will produce the image seen in Figure 3.3.

Python Python does have a boolean type:

```
In :  cl = c>120
In :  cl.dtype
Out:  dtype('bool')
```

Python

and the elements of the matrix `cl` are either `True` or `False`. This is still quite displayable with `io.imshow`. This matrix can be turned into a numeric matrix with any of:

```
In :  sk.util.img_as_ubyte(cl)
In :  uint8(cl*255)
In :  float16(cl*1)
```

Python

A numeric matrix `x` can be made boolean by

```
In :  x.astype(bool)
In :  util.img_as_bool(x)
```

Python

3.3 Bit Planes

Grayscale images can be transformed into a sequence of binary images by breaking them up into their *bit-planes*. If we consider the gray 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 gray 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 have a grayscale image of type `uint8`:

```
>> c = imread('cameraman.png');
```

MATLAB/Octave

then its bit planes can be accessed by the `bitget` function. In general, the function call

```
>> bitget(x,n)
```

MATLAB/Octave

will isolate the n-th rightmost bit from every element in `x`

All bitplanes can be simultaneously displayed using `subplot`, which places graphic objects such as images or plots on a rectangular grid in a single figure, but in order to minimize the distance between the images we can use the `position` parameter, starting off by creating the x and y positions for each subplot:

```
>> posx = [0 1 2 0 1 2 0 1]/3
>> posy = [2 2 2 1 1 1 0 0]/3
```

MATLAB/Octave

Then they can be plotted as:

```
>> for i = 1:8
>    subplot("position",[posx(i),posy(i),0.3,0.3]),
>    imshow(logical(bitget(c,i))),axis image
>  end
```

MATLAB/Octave

The result is shown in Figure 3.4. Note that the least significant bit plane, at top left, 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, which is at the bottom center, is actually a *threshold* of the image at level 128:

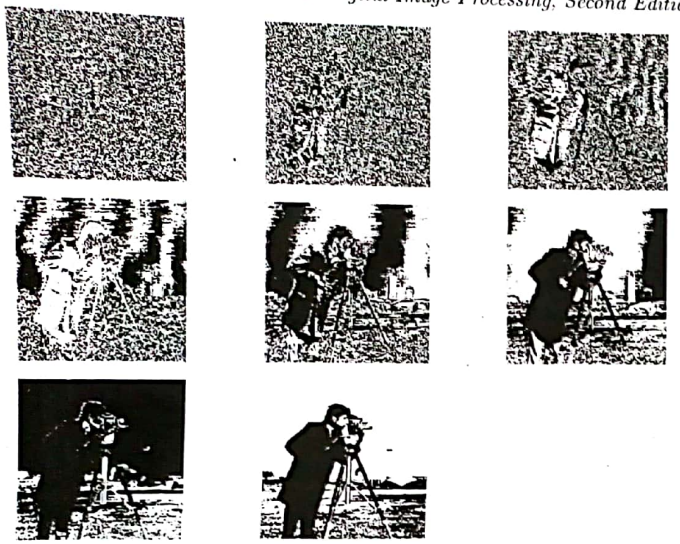


FIGURE 3.4: The bit planes of an 8-bit grayscale image

```
>> ct = c>127;
>> b8 = logical(bitget(c,8));
>> isequal(ct,b8)

ans =

    1
```

MATLAB/Octave

We shall discuss thresholding in Chapter 9.

We can recover and display the original image by multiplying each bitplane by an appropriate power of two and adding them; each scaled bitplane is added to an empty array which is initiated with the useful `zeros` function:

```
>> y = zeros(size(c));
>> for i = 1:8
>    y = y + 2^(i-1)*double(bitget(c,i));
>>   end
```

MATLAB/Octave

This new image `y` is the cameraman image:

```
>> isequal(c,uint8(y))
ans = 1
```

MATLAB/Octave

Python does not have a `bitget` function, however the equivalent result can be obtained by `bitshifting`: taking the binary string corresponding to a grayscale, and shifting it successively to the right, so the rightmost bits successively "drop off." Python uses the "»" notation for bitshifting:

```
In :  n = uint8(175)
In :  for i in range(8):
...:      print bin(n>>i)
...:
0b10101111
0b1010111
0b101011
0b10101
0b1010
0b101
0b10
0b1
```

Python

The rightmost bits can be obtained by using the modulo operator:

```
In :  for i in range(8):
...:      print (n>>i)%2
...:
1
1
1
1
0
1
0
1
```

Python

So the following generates and displays all bit planes using the `pyplot` module of the `matplotlib` library:

```
In :  import matplotlib.pyplot as plt
In :  c = io.imread('cameraman.png')
In :  bps = [(c>>i)%2 for i in range(8)]
In :  for i in range(8):
...:      plt.subplot(3,3,i+1)
...:      io.imshow(bps[i])
...:      plt.axis('off')
```

Python

and the bit planes can be reassembled with

```
In :  z = sum(bps[i]*2**i for i in range(8))
In :  (c==z).all()
Out:  True
```

Python

3.4 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 `imresize` function. Suppose we have a 256×256 8-bit grayscale 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:

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

If we apply `imresize` to the result with the parameter `2` and the method "`nearest`," 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:

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

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,'nearest');
```

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

Command	Effective resolution
`imresize(imresize(x,1/4),4,'nearest');`	64×64
`imresize(imresize(x,1/8),8,'nearest');`	32×32
`imresize(imresize(x,1/16),16,'nearest');`	16×16
`imresize(imresize(x,1/32),32,'nearest');`	8×8

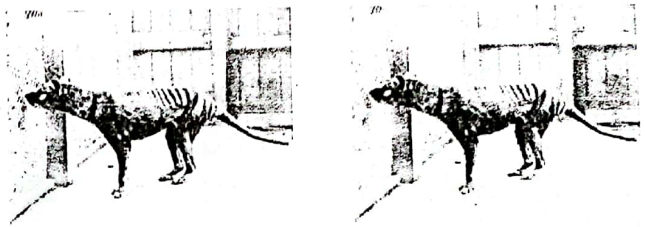
To see the effects of these commands, suppose we apply them to the image `thylacine.png`:[1]

[1] This is a famous image showing the last known *thylacine*, or Tasmanian Tiger, a now extinct carnivorous marsupial, in the Hobart Zoo in 1933.


```
>> x = imread('thylacine.png;
>> for i = 1:4
>     subplot(2,2,i),imshow(imresize(imresize(x,1/(2^(i+1))),2^(i+1),'
    nearest'))
>   end
```

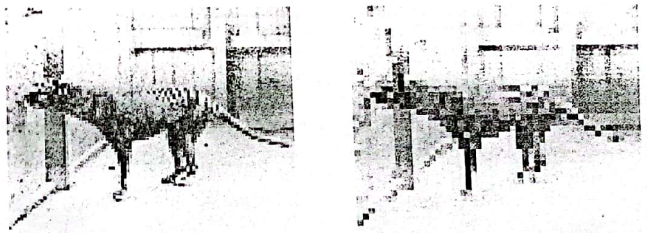
MATLAB/Octave

Since this image has size 320×400, the effective resolutions will be these dimensions divided by powers of two.



(a) The original image (b) at 160×200 resolution

FIGURE 3.5: Reducing resolution of an image



(a) At 80×100 resolution (b) At 40×50 resolution

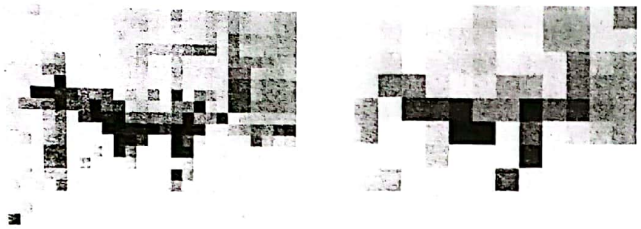
FIGURE 3.6: Further reducing the resolution of an image

The effects of increasing blockiness or *pixelization* become quite pronounced as the resolution decreases; even at 160×200 resolution fine detail, such as the wire mesh of the enclosure are less clear, and at 80×100 all edges are now quite blocky. At 40×50, the image is barely recognizable, and at 20×25 and 10×12 the image becomes unrecognizable.

Python again is similar to MATLAB and Octave. To change the spatial resolution, use the `rescale` method from the `skimage.transform` module:

```
In : import skimage.transform as tr
In : x4 = tr.rescale(tr.rescale(x,0.25),4,order=0)
```

Python



(a) At 20 × 25 resolution (b) At 10 × 12 resolution

FIGURE 3.7: Reducing the resolution of an image even more

The "`order`" parameter indicates the method of interpolation used; 0 corresponds to nearest neighbor interpolation.

3.5 Quantization and Dithering

Quantization refers to the number of grayscales used to represent the image. As we have seen, most images will have 256 grayscales, which is more than enough for the needs of human vision. However, there are circumstances in which it may be more practical to represent the image with fewer grayscales. One simple way to do this is by *uniform quantization*: to represent an image with only n grayscales, we divide the range of grayscales into n equal (or nearly equal) ranges, and map the ranges to the values 0 to $n-1$. For example, if $n = 4$, we map grayscales to output values as follows:

Original values	Output value
0–63	0
64–127	1
128–191	2
192–255	3

and the values 0, 1, 2, 3 may need to be scaled for display. This mapping can be shown graphically, as in Figure 3.8.

To perform such a mapping in MATLAB, we can perform the following operations, supposing `x` to be a matrix of type `uint8`:

```
>> q = (f/64)*64
```

MATLAB/Octave

Since `f` is of type `uint8`, dividing by 64 will also produce values of type `uint8`; so any fractional parts will be removed. For example:

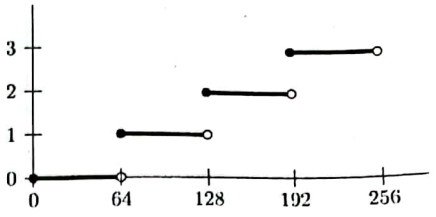


FIGURE 3.8: A mapping for uniform quantization

```
>> y = uint8(reshape(0:16:16*15,4,4))
y =

   0  64 128 192
  16  80 144 208
  32  96 160 224
  48 112 176 240
>> z = y/64
z =

   0   1   2   3
   0   1   2   3
   1   2   3   4
   1   2   3   4
>> z*64
ans =

   0  64 128 192
   0  64 128 192
  64 128 192 255
  64 128 192 255
```

MATLAB/Octave

The original array has now been quantized to only five different values. Given a `uint8` array `x`, and a value `v`, then in general the command

```
>> (x/v)*v
```

MATLAB/Octave

will produce a maximum of $256/v+1$ different possible grayscales. So with $v = 64$ as above, we would expect $256/64 + 1 = 5$ possible output grayscales.

Python acts similarly:

```
In :  y = uint8(16*np.reshape(range(16),(4,4)))
In :  y
Out:
array([[  0,  16,  32,  48],
       [ 64,  80,  96, 112],
       [128, 144, 160, 176],
       [192, 208, 224, 240]], dtype=uint8)

In: z = y/64; z
Out:
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]], dtype=uint8)

In : z*64
Out:
array([[  0,   0,   0,   0],
       [ 64,  64,  64,  64],
       [128, 128, 128, 128],
       [192, 192, 192, 192]], dtype=uint8)
```

Python

Note that dividing gives a different result in MATLAB/Octave and Python. In the former, the result is the closest integer (by rounding) to the value of $n/64$; in Python, the result is the floor.

If we wish to precisely specify the number of output grayscales, then we can use the `grayslice` function. Given an image matrix `x` and an integer `n`, the MATLAB command `grayslice(x,n)` produces a matrix whose values have been reduced to the values $0, 1, \ldots, n-1$. So, for example

```
>> x4 = grayslice(x,4);
```

MATLAB

will produce a `uint8` version of our image with values 0, 1, 2 and 3. Note that Octave works slightly differently from MATLAB here; Octave requires that the input image be of type `double`:

```
>> x4 = grayslice(im2double(x),4);
```

Octave

We cannot view this directly, as it will appear completely black: the four values are too close to zero to be distinguishable. We need to treat this matrix as the indices to a color map, and the color map we shall use is `gray(4)`, which produces a color map of four evenly spaced gray values between 0 (black) and 1.0 (white). In general, given an image `x` and a number of grayscales n, the commands

```
>> y = grayslice(x,n);
>> imshow(x,gray(n))
```

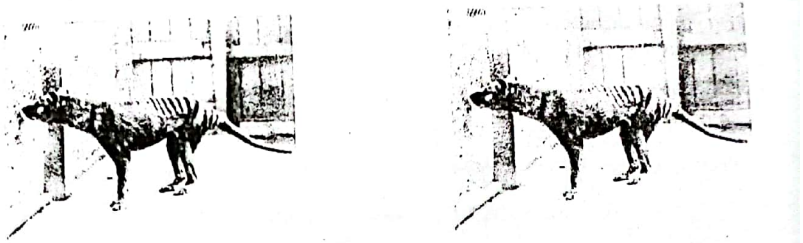
MATLAB/Octave

will display the quantized image. Note that if you are using Octave, you will need to ensure that `x` is of type `double`. Quantized images can be displayed in one go using `subplot`:


```
>> qs = [256,64,32,16,4,2]
>> px = [1/6,1/2,1/6,1/2,1/6,1/2]
>> py = [2/3,2/3,1/3,1/3,0,0]
>> for i = 1:6
>    subplot("position",[px[i],py[i],1/3,1/3]),imshow(grayslice(x,qs[i]),
    gray(qs[i]))
>    end
```

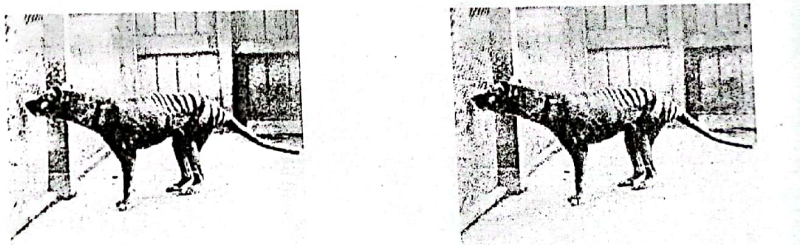
MATLAB/Octave

If we apply these commands to the thylacine image, we obtain the results shown in Figures 3.9 to 3.12.



(a) The image quantized to 128 grayscales (b) The image quantized to 64 grayscales

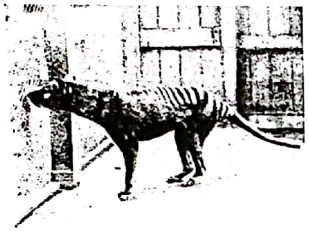
FIGURE 3.9: Quantization (1)



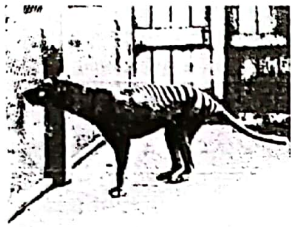
(a) The image quantized to 32 grayscales (b) The image quantized to 16 grayscale

FIGURE 3.10: Quantization (2)

One immediate consequence of uniform quantization is that of "false contours," most
ticeable with fewer grayscales. For example, in Figure 3.11, we can see on the ground
rear fence that the gray texture is no longer smooth; there are observable discontinui
between different gray values. We may expect that if fewer grayscales are used, and
jumps between consecutive grayscales becomes larger, that such false contours will oc



(a) The image quantized to 8 grayscales



(b) The image quantized to 4 grayscales

FIGURE 3.11: Quantization (3)

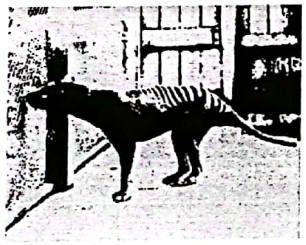


FIGURE 3.12: The image quantized to 2 grayscales

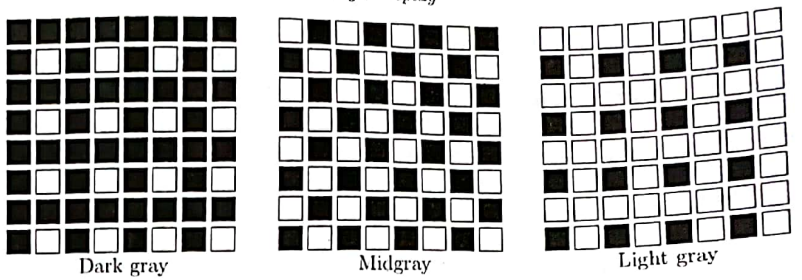


FIGURE 3.13: Patterns for dithering output

Dithering

Dithering, in general terms, refers to the process of reducing the number of colors in an image. For the moment, we shall be concerned only with grayscale dithering. Dithering is necessary sometimes for display, if the image must be displayed on equipment with a limited number of colors, or for printing. Newsprint, in particular, only has two scales of gray: black and white. Representing an image with only two tones is also known as *halftoning*.

One method of dealing with such false contours involves adding random values to the image before quantization. Equivalently, for quantization to 2 grayscales, we may compare the image to a random matrix r. The trick is to devise a suitable matrix so that grayscales are represented evenly in the result. For example, an area containing mid-way gray level (around 127) would have a checkerboard pattern; a darker area will have a pattern containing more black than white, and a light area will have a pattern containing more white than black. Figure 3.13 illustrates this. One standard matrix is

$$D = \begin{pmatrix} 0 & 128 \\ 192 & 64 \end{pmatrix}$$

which is repeated until it is as big as the image matrix, when the two are compared. Suppose $d(i,j)$ is the matrix obtained by replicating D. Thus, an output pixel $p(i,j)$ is defined by

$$p(i,j) = \begin{cases} 1 & \text{if } x(i,j) > d(i,j) \\ 0 & \text{if } x(i,j) \leq d(i,j) \end{cases}$$

This approach to quantization is called *dithering*, and the matrix D is an example of a *dither matrix*. Another dither matrix is given by

$$D_2 = \begin{pmatrix} 0 & 128 & 32 & 160 \\ 192 & 64 & 224 & 96 \\ 48 & 176 & 16 & 144 \\ 240 & 112 & 208 & 80 \end{pmatrix}$$

We can apply the matrices to our thylacine image matrix x by using the following commands in MATLAB/Octave:

```
>> D = [0 128;192 64]
>> r = repmat(D,160,200);
>> x2 = x>r; imshow(x2)
>> D2 = [0 128 32 160;192 64 224 96;48 176 16 144;240 112 208 80];
>> r2 = repmat(D2,80,100);
>> x4 = x>r2; imshow(x4)
```

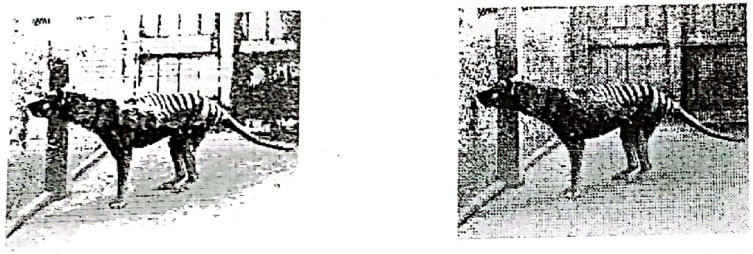
MATLAB/Octave

or in Python:

```
In :  D = array([[0,128],[192,64]])
In :  r = np.tile(D,(160,200));
In :  x2 = (x>r).astype(uint8)
In :  io.imshow(x2)
In :  D2 = array([[0, 128, 32, 160],[192, 64, 224, 96],[48, 176, 16, 144],\
...:  [240, 112, 208, 80]])
In :  r2 = np.tile(D2,80,100);
In :  x4 = (x>r2).astype(uint8)
In :  io.imshow(x2)
```

Python

The results are shown in Figure 3.14. The dithered images are an improvement on the uniformly quantized image shown in Figure 3.12. General dither matrices are provided by Hawley [15].



(a) The thylacine image dithered using D (b) The thylacine image dithered using D_2

FIGURE 3.14: Examples of dithering

Dithering can be easily extended to more than two output gray values. Suppose, for example, we wish to quantize to four output levels 0, 1, 2, and 3. Since $255/3 = 85$, we first quantize by dividing the gray value $x(i,j)$ by 85:

$$q(i,j) = \lfloor x(i,j)/85 \rfloor.$$

This will produce only the values 0, 1, and 2, except for when $x(i,j) = 255$. Suppose now that our replicated dither matrix $d(i,j)$ is scaled so that its values are in the range $0 \ldots 85$. The final value $p(i,j)$ is then defined by

$$p(i,j) = q(i,j) + \begin{cases} 1 & \text{if } x(i,j) - 85q(i,j) > d(i,j) \\ 0 & \text{if } x(i,j) - 85q(i,j) \leq d(i,j) \end{cases}$$

This can be easily implemented in a few commands, modifying `D` slightly from above and using the `repmat` function which simply tiles an array as many times as given:


```
>> D = [0 56;84 28]
>> r = repmat(D,128,128);
>> x = double(x);
>> q = floor(x/85);
>> x4 = q+(x-85*q>r);
>> imshow(uint8(85*x4))
```

MATLAB/Octave

The commands in Python are similar:

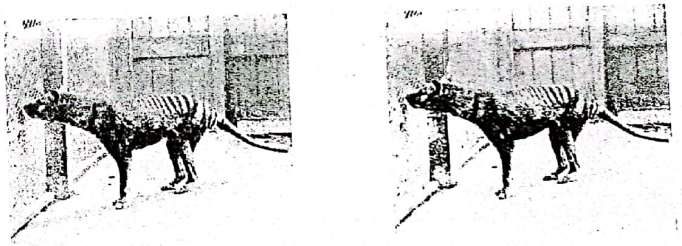
```
In :  D = array([[0, 56],[84, 28]])
In :  r = np.tile(D,(128,128))
In :  x = x.astype(float64)
In :  q = floor(x/85)
In :  x4 = q+(x-85*q>r)
In :  io.imshow(uint8(x4*85))
```

Python

and the result is shown in Figure 3.15(a). We can dither to eight gray levels by using $255/7 = 37$ (we round the result up to ensure that our output values stay within range) instead of 85 above; our starting dither matrix will be

$$D = \begin{pmatrix} 0 & 24 \\ 36 & 12 \end{pmatrix}$$

and the result is shown in Figure 3.15(b). Note how much better these images look than the corresponding uniformly quantized images in Figures 3.11(b) and 3.11(a). The eight-level result, in particular, is almost indistinguishable from the original, in spite of the quantization.



(a) Dithering to 4 output grayscales (b) Dithering to 8 output grayscales

FIGURE 3.15: Dithering to more than two grayscales

Error Diffusion

A different approach to quantization from dithering is that of *error diffusion*. The image is quantized at two levels, but for each pixel we take into account the *error* between its gray value and its quantized value. Since we are quantizing to gray values 0 and 255, pixels close to these values will have little error. However, pixels close to the center of the range: 128 will have a large error. The idea is to spread this error over neighboring pixels. A popula

method, developed by Floyd and Steinberg, works by moving through the image pixel by pixel, starting at the top left, and working across each row in turn. For each pixel $p(i,j)$ in the image we perform the following sequence of steps:

1. Perform the quantization.
2. Calculate the quantization error. This is defined as:

$$E = \begin{cases} p(i,j) & \text{if } p(i,j) < 128 \\ p(i,j) - 255 & \text{if } p(i,j) >= 128 \end{cases}$$

3. Spread this error E over pixels to the right and below according to this table:

	$+\frac{7}{16}E$	
$+\frac{3}{16}E$	$+\frac{5}{16}E$	$+\frac{1}{16}E$

There are several points to note about this algorithm:

- The error is spread to pixels *before* quantization is performed on them. Thus, the error diffusion will affect the quantization level of those pixels.
- Once a pixel has been quantized, its value will never be affected. This is because the error diffusion only affects pixels to the right and below, and we are working from the left and above.
- To implement this algorithm, we need to embed the image in a larger array of zeros, so that the indices do not go outside the bounds of our array.

The `dither` function, when applied to a grayscale image, actually implements Floyd-Steinberg error diffusion. However, it is instructive to write a simple MATLAB function to implement it ourselves; one possibility is given at the end of the chapter.

The result of applying this function to the thylacine image is shown in Figure 3.16. Note that the result is very pleasing; so much so that it is hard to believe that every pixel in the image is either back or white, and so that we have a binary image.

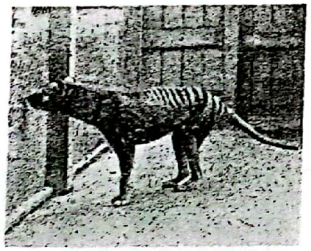


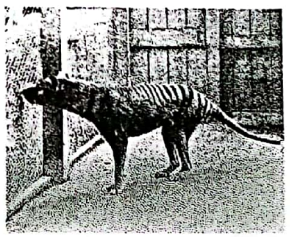
FIGURE 3.16: The thylacine image after Floyd-Steinberg error diffusion

			$+\frac{7}{48}E$	$+\frac{5}{48}E$			$+\frac{8}{42}E$	$+\frac{4}{42}E$	
$+\frac{3}{48}E$	$+\frac{5}{48}E$	$+\frac{7}{48}E$	$+\frac{5}{48}E$	$+\frac{3}{48}E$	$+\frac{2}{42}E$	$+\frac{4}{42}E$	$+\frac{8}{42}E$	$+\frac{4}{42}E$	$+\frac{2}{42}E$
$+\frac{1}{48}E$	$+\frac{3}{48}E$	$+\frac{5}{48}E$	$+\frac{3}{48}E$	$+\frac{1}{48}E$	$+\frac{1}{12}E$	$+\frac{2}{42}E$	$+\frac{4}{42}E$	$+\frac{2}{42}E$	$+\frac{1}{42}E$

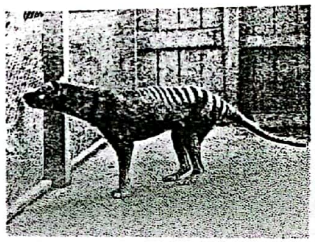
FIGURE 3.17: Different error diffusion schemes

Other error diffusion schemes are possible; two such schemes are Jarvis-Judice-Ninke and Stucki, which have error diffusion schemes shown in Figure 3.17.

They can be applied by modifying the Floyd-Steinberg function given at the end of the chapter. The results of applying these two error diffusion methods are shown in Figure 3.18.



(a) Result of Jarvis-Judice-Ninke error diffusion



(b) Result of Stucki error diffusion

FIGURE 3.18: Using other error-diffusion schemes

A good introduction to error diffusion (and dithering) is given by Schumacher [44].

3.6 Programs

Here are programs for error diffusion, first Floyd-Steinberg (with arbitrary levels), in MATLAB/Octave:

```
function y = fs(x,k)
height = size(x,1);
width = size(x,2);
ed = [0 0 0 7 0;0 3 5 1 0;0 0 0 0 0]/16;
y = uint8(zeros(height,width));
z = zeros(height+4,width+4);
z(3:height+2,3:width+2) = x;
for i = 3:height+2,
  for j = 3:width+2,
    quant = floor(255/(k-1))*floor(z(i,j)*k/256);
    y(i-2,j-2) = quant;
    e = z(i,j)-quant;
    z(i:i+2,j-2:j+2) = z(i:i+2,j-2:j+2)+e*ed;
  endfor
endfor
endfunction
```

MATLAB/Octave

and now Jarvis-Judice-Ninke with two levels:

```
function out = jjn(im)
height = size(im,1);
width = size(im,2);
out = zeros(size(im));
ed = [0 0 0 7 5;3 5 7 5 3;1 3 5 3 1]/48;
z = zeros(size(im)+4);
z(3:height+2,3:width+2) = double(im);
for i = 3:height+2,
  for j = 3:width+2,
    quant = 255*(z(i,j)>=128);
    out(i-2,j-2) = quant;
    e = z(i,j)-quant;
    z(i:i+2,j-2:j+2) = z(i:i+2,j-2:j+2)+e*ed;
  endfor
endfor
out = im2uint8(out);
endfunction
```

MATLAB/Octave

Here is Floyd-Steinberg in Python:

```
def fs(im,k):  #FS error diffusion at k levels
    rs,cs = im.shape
    ed = array([[0,0,7],[3,5,1]])/16.0
    z = zeros((rs+2,cs+2))
    z[1:rs+1,1:cs+1] = float64(im)
    for i in range(1,rs+1):
        for j in range(1,cs+1):
            old = z[i,j]
            new = (old//(255//k))*(255//(k-1))
            z[i,j] = new
            E = old - new
            z[i:i+2,j-1:j+2] = z[i:i+2,j-1:j+2]+E*ed;
    return uint8(z[1:rs+1,1:cs+1])
```

Python

and Jarvis-Judice-Ninke:

```
def jjn(im): #JJN error diffusion at two levels
    rs,cs = im.shape
    ed = array([[0,0,0,7,5],[3,5,7,5,3],[1,3,5,3,1]])/48.0
    z = zeros((rs+4,cs+4))
    z[2:rs+2,2:cs+2] = float64(im)
    for i in range(2,rs+2):
        for j in range(2,cs+2):
            old = z[i,j]
            new = (old//128)*255
            z[i,j] = new
            E = old - new
            z[i:i+3,j-2:j+3] = z[i:i+3,j-2:j+3]+E*ed;
    return uint8(z[2:rs+2,2:cs+2]>0)
```

Python

Exercises

1. Open the grayscale image `cameraman.png` and view it. What data type is it?
2. Enter the following commands (if you are using MATLAB or Octave):

```
>> [em,map] = imread('emu.png');
>> e = ind2gray(em,map);
```

MATLAB/Octave

and if you are using Python:

```
In :  import skimage.color as co
In :  em = io.imread('emu.png')
In :  e = co.rgb2gray(em)
```

Python

These will produce a grayscale image of type `double`. View this image.

3. Enter the command

```
>> e2 = im2uint8(e);
```

MATLAB/Octave

or

```
In :  import skimage.util as ut
In :  e2 = ut.img_as_ubyte(e)
```

Python

and view the output.

What does the function `im2uint8/img_as_ubyte` do? What affect does it have on

(a) The appearance of the image?

(b) The elements of the image matrix?

4. What happens if you apply that function to the cameraman image?

5. Experiment with reducing spatial resolution of the following images:

   (a) `cameraman.png`

   (b) The grayscale emu image

   (c) `blocks.png`

   (d) `buffalo.png`

   In each case, note the point at which the image becomes unrecognizable.

6. Experiment with reducing the quantization levels of the images in the previous question. Note the point at which the image becomes seriously degraded. Is this the same for all images, or can some images stand lower levels of quantization than others?

   Check your hypothesis with some other grayscale images.

7. Look at a grayscale photograph in a newspaper with a magnifying glass. Describe the colors you see.

8. Show that the 2×2 dither matrix D provides appropriate results on areas of unchanging gray. Find the results of $D > G$ when G is a 2×2 matrix of values (a) 50, (b) 100, (c) 150, (d) 200.

9. What are the necessary properties of D to obtain the appropriate patterns for the different input gray levels?

10. How do quantization levels effect the result of dithering? Use `gray2ind` to display a grayscale image with fewer grayscales, and apply dithering to the result.

11. Apply each of Floyd-Steinberg, Jarvis-Judice-Ninke, and Stucki error diffusion to the images in Question 5. Which of the images looks best? Which error-diffusion method seems to produce the best results?

    Can you isolate what aspects of an image will render it most suitable for error-diffusion?

Chapter 4

Point Processing

4.1 Introduction

Any image processing operation transforms the values of the pixels. However, image processing operations may be divided 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 4.1.

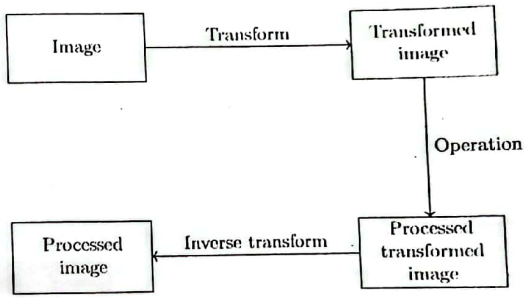


FIGURE 4.1: Schema for transform processing

2. **Neighborhood processing.** To change the gray level of a given pixel, we need only know the value of the gray levels in a small neighborhood of pixels around the given pixel.

3. **Point operations.** A pixel's gray 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 *preprocessing*, where an image is required to be modified before the main job is attempted.

4.2 Arithmetic Operations

These operations act by applying a simple function

$$y = f(x)$$

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

$$y = x \pm C$$

or multiplying each pixel by a constant:

$$y = Cx.$$

In each case, the output will need to be adjusted in order to ensure that the results are integers in the $0 \ldots 255$ range. We have seen that MATLAB and Octave work by "clipping" the values by setting:

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

and Python works by dividing an overflow by 256 and returning the remainder. We can see this by considering a list of `uint8` values in each system, first in MATLAB/Octave

```
>> x = uint8(0:32:255)
ans =
   0   32   64   96  128  160  192  224

>> x + 100
ans =
  100  132  164  196  228  255  255  255

>> x - 100
ans =
   0    0    0    0   28   60   92  124
```

MATLAB/Octave

and then in Python with `arange`, which produces a list of values as an array:

```
In : x = uint8(array(arange(0,255,32)));x
Out: array([  0,  32,  64,  96, 128, 160, 192, 224], dtype=uint8)

In : x + 100
Out: array([100, 132, 164, 196, 228,   4,  36,  68], dtype=uint8)

In : x - 100
Out: array([156, 188, 220, 252,  28,  60,  92, 124], dtype=uint8)
```

Python

We can obtain an understanding of how these operations affect an image by plotting $y = f(x)$. Figure 4.2 shows the result of adding or subtracting 64 from each pixel in the image. Notice that when we add 128, all gray values of 127 or greater will be mapped to 255. And

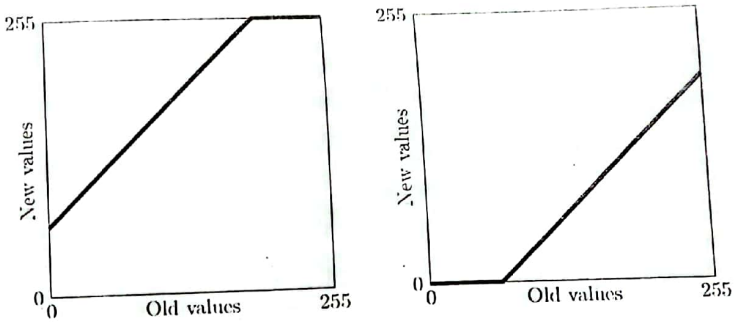


FIGURE 4.2: Adding and subtracting a constant in MATLAB and Octave

when we subtract 128, all gray 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.

Figure 4.3 shows the results in Python.

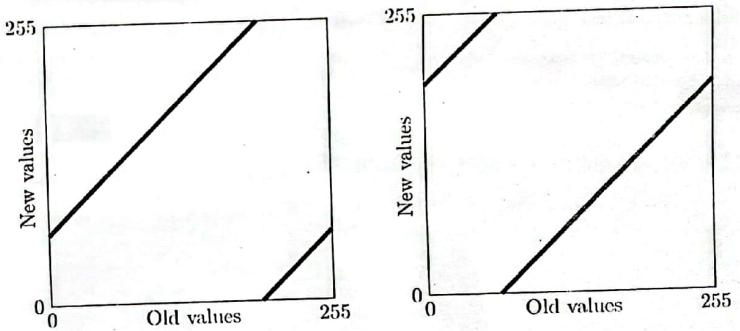


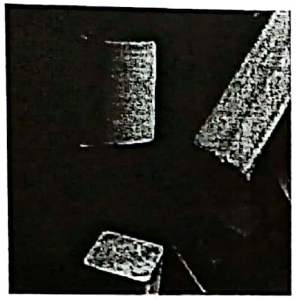
FIGURE 4.3: Adding and subtracting a constant in Python

We can test this on the "blocks" image `blocks.png`, which we can see in Figure 1.4. We start by reading the image in:

```
>> b = imread('blocks.png');
>> class(b)
ans =
    uint8
```

MATLAB/Octave

The point of the second command `class` is to find the numeric data type of `b`; it is `uint8`. This tells us that adding and subtracting constants will result in automatic clipping:



Adding 128 Subtracting 128

FIGURE 4.4: Arithmetic operations on an image: adding or subtracting a constant

```
>> imshow(b+128)
>> figure, imshow(b-128)
```

MATLAB/Octave

and the results are seen in Figure 4.4. Because of Python's arithmetic, the results of

```
In : b = io.imread('blocks.png');
In : io.imshow(b+128)
```

Python

will not be a brightened image: it is shown in Figure 4.5.

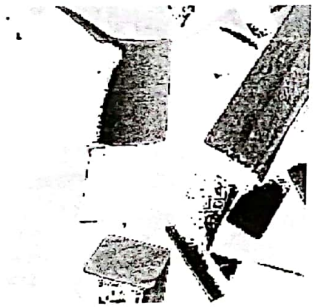


FIGURE 4.5: Adding 128 to an image in Python

In order to obtain the same results as given by MATLAB and Octave, first we need to use floating point arithmetic, and clip the output with `np.clip`:

```
In :  bf = float64(b)
In :  b1 = uint8(np.clip(bf+128,0,255))
In :  b2 = uint8(np.clip(bf-128,0,255))
```

Python

Then `b1` and `b2` will be the same as the images shown in Figure 4.4.

We can also perform lightening or darkening of an image by multiplication; Figure 4.6 shows some examples of functions that will have these effects. Again, these functions assume clipping. All these images can be viewed with `imshow`; they are shown in Figure 4.7.

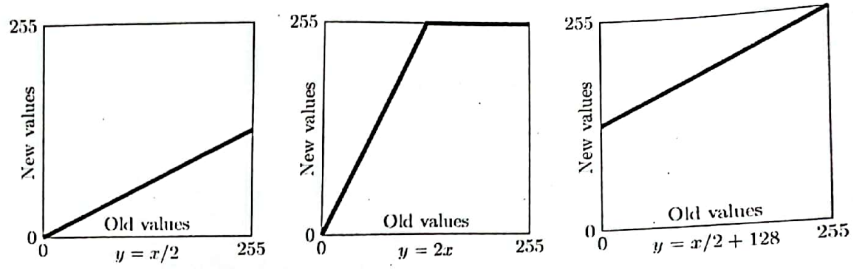
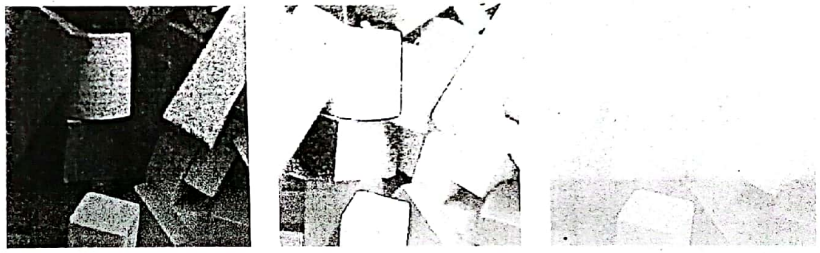


FIGURE 4.6: Using multiplication and division



`imshow(b/2)` `imshow(b*2)` `b/2+128`

FIGURE 4.7: Arithmetic operations on an image: multiplication and division

Compare the results of darkening with division by two and subtraction by 128. Note that the division result, although darker than the original, is still quite clear, whereas a lot of information was lost by the subtraction process. This is because in image `b2` all pixels with gray values 128 or less have become zero.

A similar loss of information has occurred by adding 128, and so a better result is obtained by `b/2+128`.

Complements

The *complement* of a grayscale image is its photographic negative. If an image matrix `m` is of type `uint8` and so its gray values are in the range 0 to 255, we can obtain its negative with the command

```
>> 255 - m
```

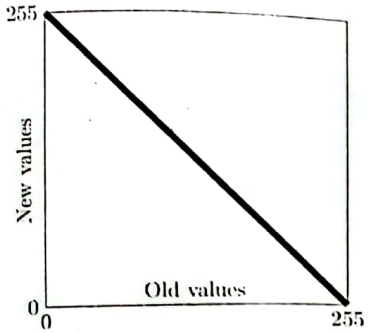
MATLAB/Octave

(and the same in Python). If the image is binary, we can use

```
>> ~m
```

MATLAB/Octave

which also works for boolean arrays in Python.



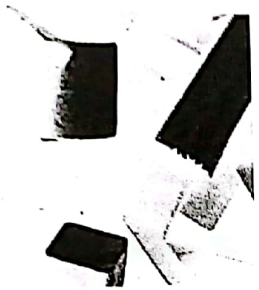
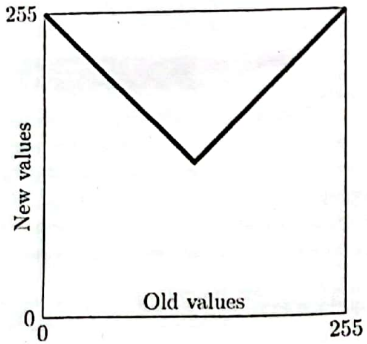


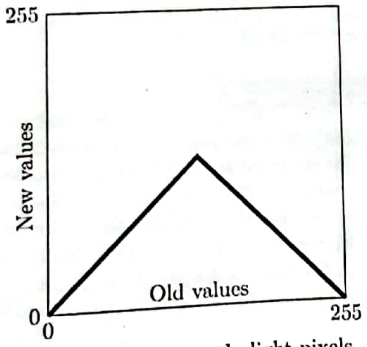
FIGURE 4.8: Image complementation: `255-b`

Interesting special effects can be obtained by complementing only *part* of the image; for example by taking the complement of pixels of gray value 128 or less, and leaving other pixels untouched. Or, we could take the complement of pixels that are 128 or greater and leave other pixels untouched. Figure 4.9 shows these functions.

The effect of these functions is called *solarization*, and will be discussed further in Chapter 16.



Complementing only dark pixels



Complementing only light pixels

FIGURE 4.9: Part complementation

4.3 Histograms

Given a grayscale image, its *histogram* consists of the histogram of its gray levels; that is, a graph indicating the number of times each gray 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 gray levels (and hence the histogram) would be clustered at the lower end.
- In a uniformly bright image, the gray levels would be clustered at the upper end.
- In a well-contrasted image, the gray 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:

```
>> c = imread('chickens.png');
>> imshow(c),figure,imhist(c),axis tight
```

MATLAB/Octave

(the `axis tight` command ensures the axes of the histogram are automatically scaled to fit all the values in). In Python, the commands are:

```
In :  c = io.imread('chickens.png')
In :  io.imshow(c)
In :  f = figure(); f.show(plt.hist(c.flatten(),bins=256))
```

Python

The result is shown in Figure 4.10. Since most of the gray values are all clustered

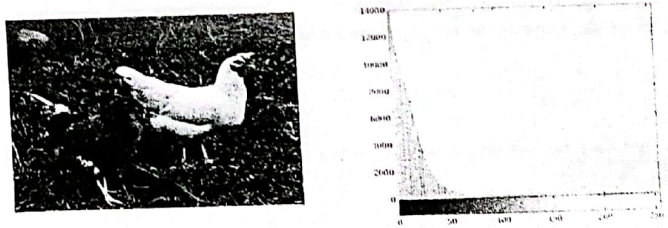


FIGURE 4.10: The image `chickens.png` and its histogram

together at the left of the histogram, we would expect the image to be dark and poo contrasted, as indeed it is.

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

Histogram Stretching (Contrast Stretching)

Suppose we have an image with the histogram shown in Figure 4.11, associated wi

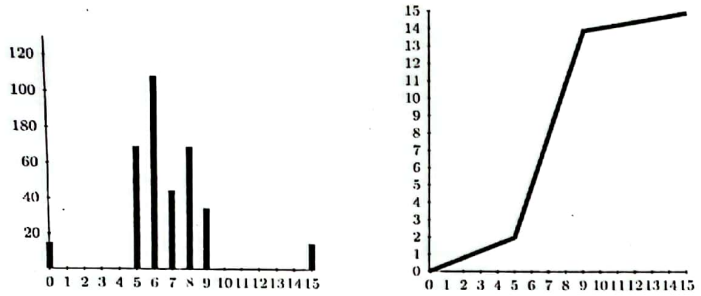


FIGURE 4.11: A histogram of a poorly contrasted image and a stretching function

table of the numbers n_i of gray values:

Gray 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

(with $n = 360$, as before.) We can stretch the gray levels in the center of the range out by applying the piecewise linear function shown at the right in Figure 4.11. This function has the effect of stretching the gray levels 5–9 to gray levels 2–14 according to the equation:

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

where i is the original gray level and j its result after the transformation. Gray 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 given in Figure 4.12:

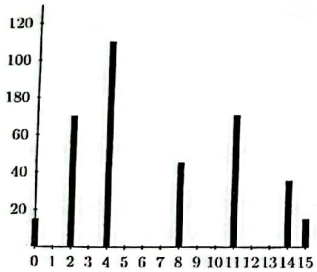


FIGURE 4.12: Histogram after stretching

which indicates an image with greater contrast than the original.

MATLAB/Octave: Use of `imadjust`.

To perform histogram stretching in MATLAB or Octave 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 4.13. Since `imadjust` is

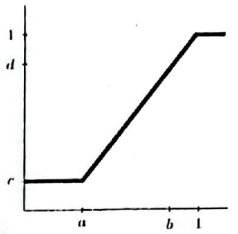


FIGURE 4.13: The stretching function given by `imadjust`

designed to 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 4.11. 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,[],[])
```

MATLAB/Octave

does nothing, and the command

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

MATLAB/Octave

inverts the gray 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 in Figure 4.13. However, values less than one produce a function that is concave downward, as shown on the left in Figure 4.14, and values greater than one produce a figure that is concave upward, as shown on the right in Figure 4.14.

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, with the chickens image:

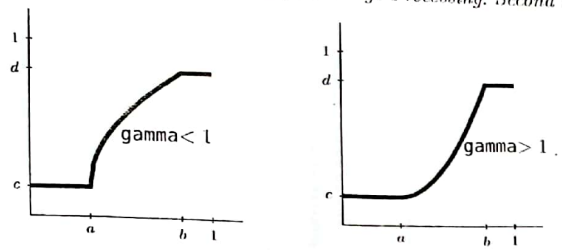


FIGURE 4.14: The `imadjust` function with `gamma` not equal to 1

```
>> ca1 = imadjust(t,[],[],0.5);
>> ca2 = imadjust(t,[],[],0.25);
>> imshow(ca1),figure,imshow(ca2)
```

MATLAB/Octave

produces the result shown in Figure 4.15.

FIGURE 4.15: The chickens image with different adjustments with the `gamma` value

Both results show background details that are hard to determine in the original image. Finding the correct gamma value for a particular image will require some trial and error.

The `imadjust` stretching function can be viewed with the `plot` function. For example,

```
>> plot(c,ca1,'.'),axis tight
```

MATLAB/Octave

produces the plot shown in Figure 4.16. Since `p` and `ph` are matrices that contain the original values and the values after the `imadjust` function, the `plot` function simply plots them, using dots to do it.

Adjustment in Python

Adjustment methods are held in the `exposure` module of `skimage`. Adjustment with gamma can be achieved with:

```
In :  import skimage.exposure as ex
In :  c = io.imread('chickens.png')
In :  ca1 = ex.adjust_gamma(c,0.5)
```

Python

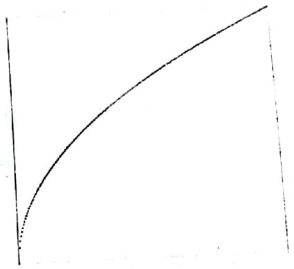


FIGURE 4.16: The function used in Figure 4.15

A Piecewise Linear Stretching Function

We can easily write our own function to perform piecewise linear stretching as shown in Figure 4.17. This is easily implemented by first specifying the points (a_i, b_i) involved and

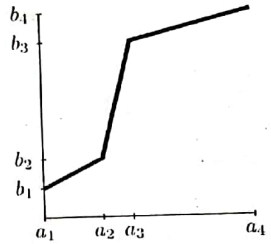


FIGURE 4.17: A piecewise linear stretching function

then using the one-dimensional interpolation function `interp1` to join them. In this case, we might have:

```
>> a = [0 100 150 255]
>> b = [40 100 220 255]
>> lin = interp1(a,b,0:255,'linear');
```

MATLAB/Octave

Then it can be applied to the image with

```
>> c1 = uint8(lin(c+1));
```

MATLAB/Octave

Python also provides easy interpolation:

```
In :  a = array([0, 100, 150, 255])
In :  b = array([40, 100, 220, 255])
In :  lin = np.interp(range(256),a,b)
```

Python

Then it can be applied to the image with

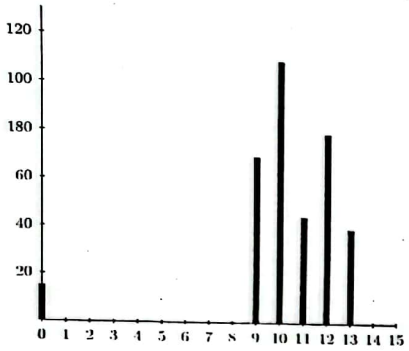


FIGURE 4.18: Another histogram indicating poor contrast

```
>> cl = uint8(lin[c])
```

Python

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 that is uniform so that every bar on the histogram is of the same height, or in other words each gray 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 gray levels $0, 1, 2, \ldots L-1$, and that gray level i occurs n_i times in the image. Suppose also that the total number of pixels in the image is n so that $n_0 + n_1 + n_2 + \cdots + n_{L-1} = n$. To transform the gray levels to obtain a better contrasted image, we change gray level i to

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

and this number is rounded to the nearest integer.

An Example

Suppose a 4-bit grayscale image has the histogram shown in Figure 4.18 associated with a table of the numbers n_i of gray values:

Gray 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

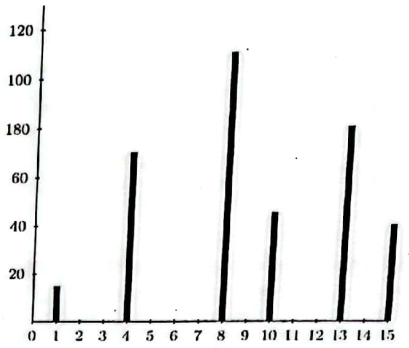


FIGURE 4.19: The histogram of Figure 4.18 after equalization

(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 $15/360 = 1/24$:

Gray 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 gray values, obtained by reading off the first and last columns in the above table:

Original gray level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Final gray 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 4.19. 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 or Octave, use the `histeq` function; for example:

```
>> c = imread('chickens.png');
>> ch = histeq(c);
>> imshow(ch),figure,imhist(ch),axis tight
```

MATLAB/Octave

Python supplies the `equalize_hist` method in the `exposure` module:

```
In :  c = io.imread('chickens.png')
In :  import sk.exposure as ex
In :  ch = ex.equalize_hist(c)
In :  f = figure(); f.show(plt.hist(ch.flatten(),bins=256))
```

Python

applies histogram equalization to the chickens image, and produces the resulting histogram. These results are shown in Figure 4.20. Notice the far greater spread of the histogram. This

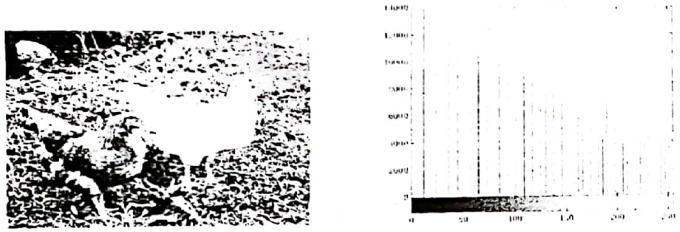


FIGURE 4.20: The histogram of Figure 4.10 after equalization

corresponds to the greater increase of contrast in the image.

We give one more example, that of a very dark image. For example, consider an underexposed image of a sunset:

```
>> s = imread('sunset.png');
>> imshow(s)
```

MATLAB/Octave

It can be seen both from the image and the histogram shown in Figure 4.21 that the matrix e contains mainly low values, and even without seeing the image first we can infer from its histogram that it will appear very dark when displayed. We can display this matrix and its histogram with the usual commands:

```
>> sh = imhist(s);
>> imhow(sh),figure,imhist(sh),axis auto
```

MATLAB/Octave

or in Python as

```
In :  s = io.imread('sunset.png')
In :  io.imshow(s)
In :  f = figure(); f.show(plt.hist(s.flatten(),bins=256))
```

Python

and improve the contrast of the image with histogram equalization, as well as displaying the resulting histogram. The results are shown in the top row of Figure 4.21.

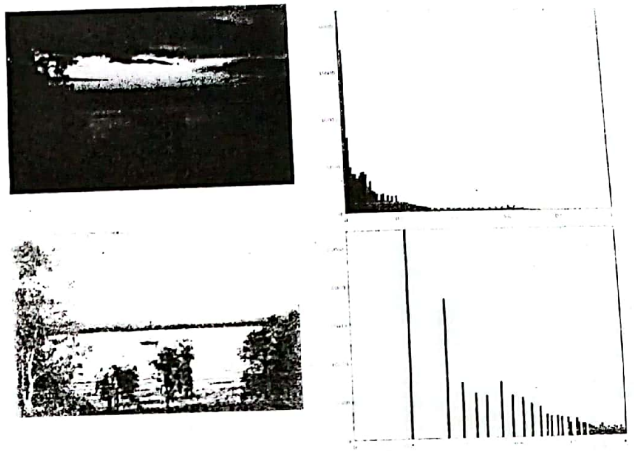


FIGURE 4.21: The sunset image and its histogram, with equalization

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:

```
>> sh = histeq(s);
>> imshow(sh),figure,imhist(sh),axis tight
```

MATLAB/Octave

or

```
In :  sh = ex.equalize_hist(s)
In :  f = figure(); f.show(plt.hist(sh.flatten(),bins=256))
```

Python

and the results are shown in the bottom row of Figure 4.21. Note that many details that are obscured in the original image are now quite clear: the trunks of the foreground trees, the ripples on the water, the clouds at the very top.

Why It Works

Consider the histogram in Figure 4.18. To apply histogram stretching, we would need to stretch out the values between gray levels 9 and 13. Thus, we would need to apply a piecewise function similar to that shown in Figure 4.11.

Let's consider the cumulative histogram, which is shown in Figure 4.22. The dashed line is simply joining the top of the histogram bars. However, it can be interpreted as an appropriate 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 4.3.

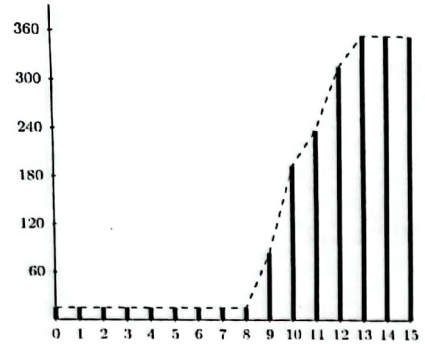


FIGURE 4.22: The cumulative histogram

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 [7]), 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 [17].

4.4 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...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, and `im` is an image, then the lookup table can be applied by the simple command

```
T(im) or T[im]
```

depending on whether we are working in MATLAB/Octave or Python.

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);
```

MATLAB/Octave

or

```
In : T = uint8(arange(256))/2
```

Python

apply it to the blocks image `b` with

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

MATLAB/Octave

or

```
In : b2 = T[b]
```

Python

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

The piecewise stretching function discussed above was in fact an example of the use of a lookup table.

Exercises

Image Arithmetic

1. Describe lookup tables for
   (a) Multiplication by 2
   (b) Image complements
2. Enter the following command on the blocks image `b`:

   ```
   b2 = (b/64)*64
   ```

   Comment on the result. Why is the result not equivalent to the original image?
3. Replace the value 64 in the previous question with 32 and 16.

Histograms

4. Write informal code to calculate a histogram $h[f]$ of the gray values of an image $f[row][col]$.
5. The following table gives the number of pixels at each of the gray levels 0–7 in an image with those gray values only:

0	1	2	3	4	5	6	7
3244	3899	4559	2573	1428	530	101	50

Draw the histogram corresponding to these gray levels, and then perform a histogram equalization and draw the resulting histogram.

6. The following tables give the number of pixels at each of the gray levels 0-15 in an image with those gray values only. In each case, draw the histogram corresponding to these gray levels, and then perform a histogram equalization and draw the resulting histogram.

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	40	60	75	80	75	65	55	50	45	40	35	30	25	20	30

(b)

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

7. The following small image has gray values in the range 0 to 19. Compute the gray level histogram and the mapping that will equalize this histogram. Produce an 8×8 grid containing the gray values for the new histogram-equalized image.

```
12   6   5  13  14  14  16  15
11  10   8   5   8  11  14  14
 9   8   3   4   7  12  18  19
10   7   4   2  10  12  13  17
16   9  13  13  16  19  19  17
12  10  14  15  18  18  16  14
11   8  10  12  14  13  14  15
 8   6   3   7   9  11  12  12
```

8. Is the histogram equalization operation idempotent? That is, is performing histogram equalization *twice* the same as doing it just once?

9. Apply histogram equalization to the indices of the image `emu.png`.

10. Create a dark image with

```
>> c = imread('cameraman.png');
>> [x,map] = gray2ind(c);
```

MATLAB/Octave

The matrix `x`, when viewed, will appear as a very dark version of the cameraman image. Apply histogram equalization to it and compare the result with the original image.

11. Using either `c` and `ch` or `s` and `sh` from Section 4.3, enter the command

```
>> figure,plot(c,ch,'.'),grid on
```

MATLAB/Octave

or

```
In :  plt.plot(c,ch,'.'),plt.grid('on'),plt.axis('image')
```

Python

What are you seeing here?

12. Experiment with some other grayscale images.

13. Using LUTs, and following the example given in Section 4.4, write a simpler function for performing piecewise stretching than the function described in Section 4.3.

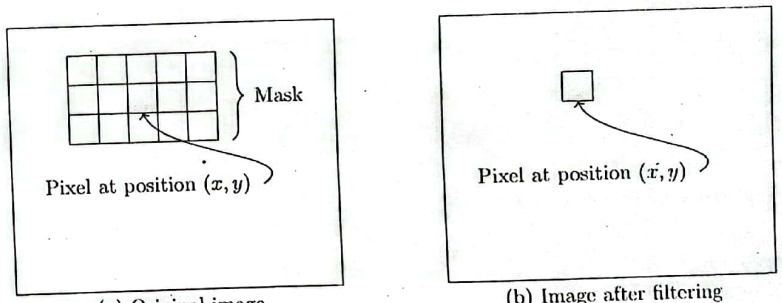
Chapter 5

Neighborhood Processing

5.1 Introduction

We have seen in Chapter 4 that an image can be modified by applying a particular function to each pixel value. Neighborhood processing may be considered an extension of this, where a function is applied to a neighborhood 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 gray values calculated from the gray values under the mask, as shown in Figure 5.1. The



(a) Original image (b) Image after filtering

FIGURE 5.1: Using a spatial mask on an image

combination of mask and function is called a *filter*. If the function by which the new gray value is calculated is a linear function of all the gray 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 neighborhood, and adding up all these products. Suppose we have a 3×5 mask as illustrated in Figure 5.1. Suppose that the mask values are given by:

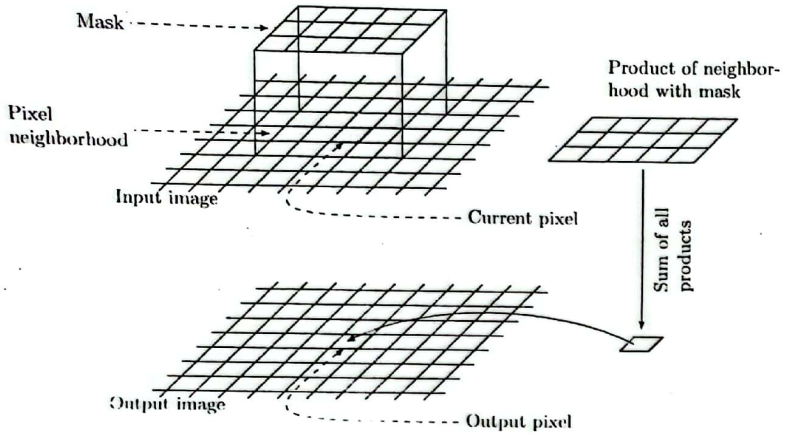


FIGURE 5.2: Performing linear spatial filtering

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

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 5.2.

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 neighborhood
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

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

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. Filtering as described above, where the filter is *not* rotated by 180°, is also called *correlation*.

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 gray value of the corresponding pixel in the new image. This operation may be described as follows:

a	b	c
d	e	f
g	h	i

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

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

To apply this to an image, consider the 5×5 "image" obtained by multiplying a magic square by 10. In MATLAB/Octave:

```
>> 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
```

MATLAB/Octave

and in Python:

```
In:  x = 10*uint8(array([[17,24,1,8,15],[23,5,7,14,16],\
...: [4,6,13,20,22],[10,12,19,21,3],[11,18,25,2,9]]))
In: x
Out:
array([[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]], dtype=uint8)
```

Python

We may regard this array as being made of nine overlapping 3×3 neighborhoods. 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 neighborhood 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 in MATLAB/Octave as

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

ans =

  111.1111
```

MATLAB/Octave

or in Python as

```
In : x[0:3,0:3].mean()
Out: 111.11111111111111
```

Python

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

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 in MATLAB/Octave:


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

ans =

  108.8889
```

MATLAB/Octave

and in Python:

```
In : x[0:3,1:4].mean()
Out: 108.88888888888889
```

Python

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 `x` with the 3×3 averaging filter.

5.2 Notation

It is convenient to describe a linear filter simply in terms of the coefficients of all the gray 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 gray values as

a	b	c
d	e	f
g	h	i

$\longrightarrow a-2b+c-2d+4e-2f+g-2h+i$

Borders of the Image

There is an obvious problem in applying a filter—what happens at the border of the image, where the mask partly falls outside the image? In such a case, as illustrated in Figure 5.3 there will be a lack of gray values to use in the filter function.

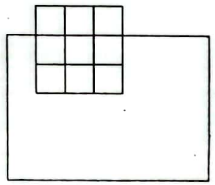


FIGURE 5.3: A mask at the edge of an image

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 those along the borders, and results in an output image that 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, dark borders) around the image.

Repeat the image. This means tiling the image in all directions, so that the mask always lies over image values. Since values might thus change across edges, this method may introduce unwanted artifacts in the result.

Reflect the image. This is similar to repeating, except that the image is reflected across all of its borders. This will ensure that there are no extra sudden changes introduced at the borders.

The last two methods are illustrated in Figure 5.4.

We can also see this by looking at the image which increases from top left to bottom right:

```
 20   40   60   80  100
 40   60   80  100  120
 60   80  100  120  140
 80  100  120  140  160
100  120  140  160  180
```

The results of filtering with a 3×3 averaging filter with zero padding, reflection, and repetition are shown in Figure 5.5.

Note that with repetition the upper left and bottom right values are "wrong"; this is because they have taken as extra values pixels that are very different. With zero-padding the upper left value has the correct size, but the bottom right values are far smaller than they should be, as they have been convolved with zeros. With reflection the values are all of correct size.

FIGURE 5.4: Repeating an image for filtering at its borders

17	33	46	59	44
33	60	80	100	73
46	80	100	120	86
60	100	120	140	100
44	73	86	99	71

Zero-padding

86	73	93	113	99
73	60	80	100	86
93	80	100	120	106
113	100	120	140	126
99	86	106	126	112

Repetition

32	46	66	86	99
46	60	80	100	113
66	80	100	120	133
86	100	120	140	153
99	113	133	153	166

Reflection

FIGURE 5.5: The result of filtering with different modes

5.3 Filtering in MATLAB and Octave

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

```
imfilter(image,filter,...)
```

and the result is a matrix of the same data type as the original image. There are other parameters for controlling the behavior of the filtering. Pixels at the boundary of the image can be managed by padding with zeros; this being the default method, but there are several other options:

Extra parameter	Implements
`'symmetric'`	Filtering with reflection
`'circular'`	Filtering with tiling repetition
`'replication'`	Filtering by repeating the border elements.
`'full'`	Padding with zero, and applying the filter at all places on and around the image where the mask intersects the image matrix.

The last option returns a result that is *larger* than the original:

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

ans =

   19   46   47   37   27   26   17
   44   77   86   66   68   59   34
   49   88  111  109  129  106   59
   41   67  110  130  150  107   46
   28   68  131  151  149   86   38
   23   57  106  108   88   39   13
   12   32   60   50   40   12   10
```

MATLAB/Octave

The central 5×5 values of the last operation are the same values as provided by the command `imfilter(x,a)` and with no extra parameters.

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])
```

MATLAB/Octave

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

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

MATLAB/Octave

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:

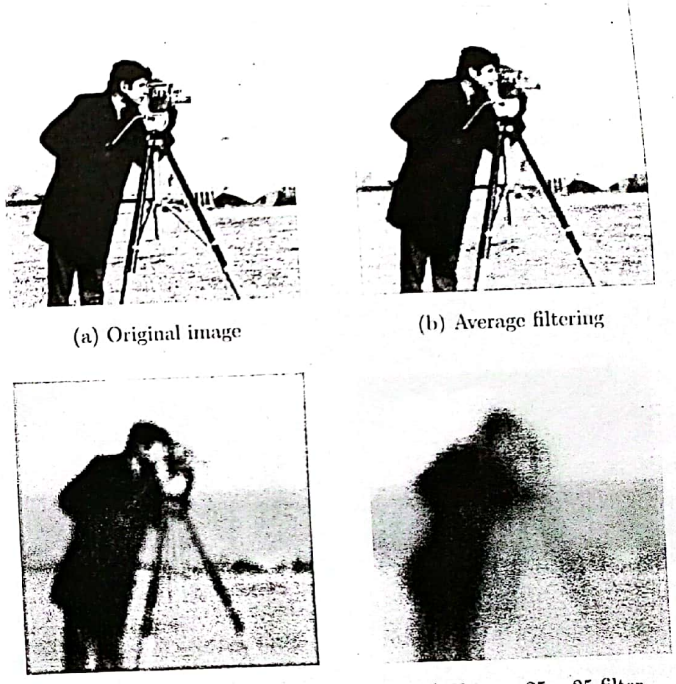
```
>> c = imread('cameraman.png');
>> f1 = fspecial('average');
>> cf1 = imfilter(c,f1);
```

MATLAB/Octave

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 5.6(c), where a 9×9 averaging filter has been used, and in Figure 5.6(d), where a 25×25 averaging filter has been used.

Notice how the default zero padding used at the edges has resulted in a dark border appearing around the image (c). This would be especially noticeable when a large filter is being used. Any of the above options can be used instead; image (d) was created with the `symmetric` 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 the perfect operation for autonomous machine recognition, or if we are only concentrating on the "gross" aspects of the image: numbers of objects or amount of dark and light areas. In such cases, too much detail may obscure the outcome.



(a) Original image (b) Average filtering

(c) Using a 9×9 filter (d) Using a 25×25 filter

FIGURE 5.6: Average filtering

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

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

We will also consider other examples.

5.4 Filtering in Python

There is no Python equivalent to the `fspecial` and `imfilter` commands, but there are a number of commands for applying either a generic filter or a specific filter.

Linear filtering can be performed by using `convolve` or `correlate` from the `ndimage` module of the library, or `generic_filter`.

For example, we may apply 3×3 averaging to the 5×5 magic square array:

```
In:  import scipy.ndimage as ndi
In:  x = uint8(array([[17,24,1,8,15],[23,5,7,14,16],[4,6,13,20,22],\
     [10,12,19,21,3],[11,18,25,2,9]])*10)
In:  a = ones((3,3))/9
In:  ndi.convolve(x,a,mode='constant')
Out:
array([[ 76,  85,  65,  67,  58],
       [ 87, 111, 108, 128, 105],
       [ 66, 109, 130, 150, 106],
       [ 67, 131, 151, 148,  85],
       [ 56, 105, 107,  87,  38]], dtype=uint8)
```

Python

Note that this result is the same as that obtained with MATLAB's `imfilter(x,a)` command, and Python also automatically converts the result to the same data type as the input; here as an unsigned 8-bit array. The `mode` parameter, here set to `'constant'`, tells the function that the image is to be padded with constant values, the default value of which is zero, although other values can be specified with the `cval` parameter.

So to obtain, for example, the image in Figure 5.6(c), you could use the following Python commands:

```
In:  c = io.imread('cameraman.png')
In:  cf = ndi.convolve(c,ones((9,9))/81,mode='constant')
In:  io.imshow(cfs)
```

Python

Alternately, you could use the built in `uniform_filter` function:

```
In:  cf = ndi.uniform_filter(c,[9,9],mode='constant')
```

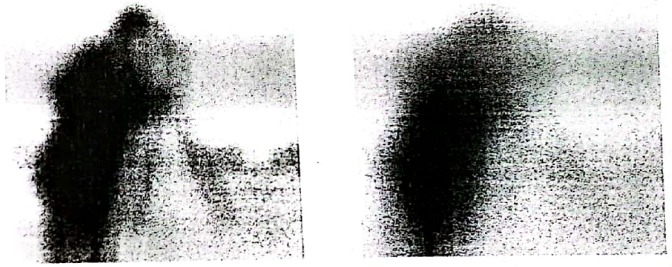
Python

Python differs from MATLAB and Octave here in that the default behavior of spatial convolution at the edges of the image is to reflect the image in all its edges. This will ameliorate the dark borders seen in Figure 5.6. So with leaving the `mode` parameter in its default setting, the commands

```
In:  cf = ndi.uniform_filter(c,25)
In:  cf2 = ndi.uniform_filter(c,50)
```

Python

will produce the images shown in Figure 5.7.



(a) Using a 25×25 filter (b) Using a 50×50 filter

FIGURE 5.7: Average filtering in Python

5.5 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 gray values change with distance. This concept will be

given a more formal setting in Chapter 7. *High frequency components* are characterized by large changes in gray values over small distances; examples 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 gray values. These may include backgrounds or 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 a 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 e elements in the matrix), in the high pass filter is zero. This means that in a low frequency part of an image, where the gray values are similar, the result of using this filter is that the corresponding gray 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
152	148	149	151
151	149	150	148

$\longrightarrow$

11	6
−13	−5

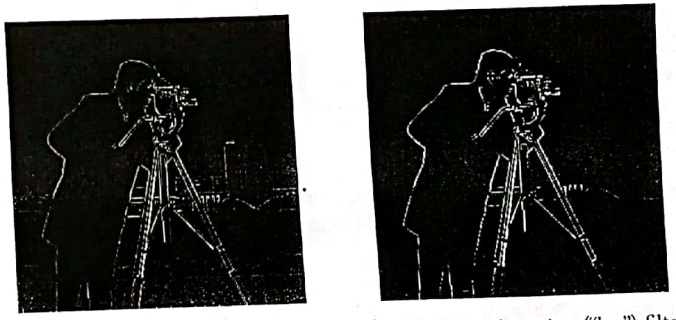
The resulting values are close to zero, which is the expected result of applying a high pass filter to a low frequency component. We shall see how to deal with negative values later.

High pass filters are of particular value in edge detection and edge enhancement (of which we shall see more in Chapter 9). 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=imfilter(c,f,'symmetric');
>> 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=imfilter(c,f1,'symmetric');
```

MATLAB/Octave

The images are shown in Figure 5.8. Image (a) is the result of the Laplacian filter; image (b) shows the result of the Laplacian of Gaussian ("log") filter. We discuss Gaussian filters in Section 5.6.



(a) Laplacian filter (b) Laplacian of Gaussian ("log") filtering

FIGURE 5.8: High pass filtering

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

Note that both MATLAB and Octave in fact do more than merely apply the Laplacia or LoG filters to the image; the results in Figure 5.8 have been "cleaned up" from the ra filtering. We can see this by applying the Laplacian filter in Python:

```
In: f = array([[1,4,1],[4,-20,4],[1,4,1]])
In: cf = ndi.convolve(c,f)
In: io.imshow(cf)
```

Pytho

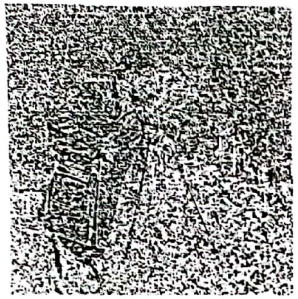


FIGURE 5.9: Laplacian filtering without any extra processing

of which the result is shown in Figure 5.9. We shall see in Chapter 9 how Python can be used to obtain results similar to those in Figure 5.8.

Values Outside the Range 0–255

We have seen that for image display, we would like the gray values of the pixels to lie between 0 and 255. However, the result of applying a linear filter may be values that 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 gray 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 gray values outside the 0–255 range; in particular, if the gray 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 gray value produced by the filter if 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 gray levels x produced by the filter will result (after any necessary rounding) in an image that can be displayed.

As an example, let's apply the high pass filter given in Section 5.5 to the cameraman image:

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

MATLAB/Octave

We have to convert the image to type "double" before the filtering, or else the result will be an automatically scaled image of type `uint8`. 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));
```

MATLAB/Octave

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-mncf2);
```

MATLAB/Octave

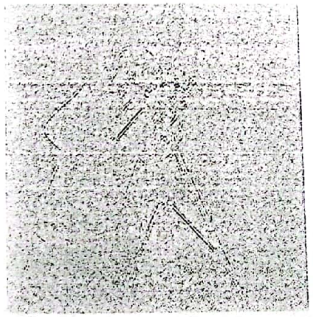
The result will be a matrix of type `double`, with entries in the range 0.0–1.0. This can be be viewed with `imshow`. We can make it a `uint8` image by multiplying by 255 first. The result can be seen in Figure 5.10.

Sometimes a better result can be obtained by dividing an output of type `double` by a constant before displaying it:

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

MATLAB/Octave

and this is also shown in Figure 5.10.



Using `mat2gray`

Dividing by a constant

FIGURE 5.10: Using a high pass filter and displaying the result

High pass filters are often used for edge detection. These can be seen quite clearly in the right-hand image of Figure 5.10.

In Python, as we have seen, the `convolve` operation returns a `uint8` array as output if the image is of type `uint8`. To apply a linear transformation, we need to start with an output of type `float32`:

```
In:  cf2 = ndi.convolve(float32(c),f,mode='constant')
In:  maxcf2 = cf2.max()
In:  mincf2 = cf1.min()
In:  cf2f = (cf2-mincf2)/(maxcf2-mincf2)
In:  io.imshow(cf2f)
```

Python

Note that Python's `imshow` commands automatically scale the image to full brightness range. To switch off that behavior the parameters `vmax` and `vmin` will give the "true" range, as opposed to the displayed range:

```
In:  io.imshow(cf2/60,vmax=1.0,vmin=0.0)
```

Python

These last two `imshow` commands will produce the same images as shown in Figure 5.10.

5.6 Gaussian Filters

We have seen some examples of linear filters so far: the averaging filter and a high pass filter. The `fspecial` function can produce many different filters for use with the `imfilter` function; we shall look at a particularly important filter here.

Gaussian filters are a class of low pass filters, all based on the Gaussian probability distribution function

$$f(x) = e^{-\frac{x^2}{2\sigma^2}}$$

where σ is the standard deviation: a large value of σ produces to a flatter curve, and a small value leads to a "pointier" curve. Figure 5.11 shows examples of such one-dimensional Gaussians. Gaussian filters are important for a number of reasons:

1. They are mathematically very "well behaved"; in particular the Fourier transform (see Chapter 7) of a Gaussian filter is another Gaussian.
2. They are rotationally symmetric, and so are very good starting points for some edge detection algorithms (see Chapter 9),
3. They are *separable*; in that a Gaussian filter may be applied by first applying a one-dimension Gaussian in the x direction, followed by another in the y direction. This can lead to very fast implementations.
4. The convolution of two Gaussians is another Gaussian.

A two-dimensional Gaussian function is given by

$$f(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Large value of σ Small value of σ

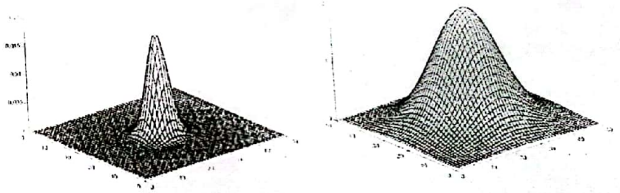
FIGURE 5.11: One-dimensional Gaussians

The command `fspecial('gaussian')` produces a discrete version of this function. We can draw pictures of this with the `surf` function, and to ensure a nice smooth result, we shall create a large filter (size 50×50) with different standard deviations.

```
>> a = 50; s = 3;
>> g = fspecial('gaussian',[a a],s);
>> surf(1:a,1:a,g)
>> s = 9;
>> g2 = fspecial('gaussian',[a a],s);
>> figure,surf(1:a,1:a,g2)
```

MATLAB/Octave

The surfaces are shown in Figure 5.12.



$\sigma = 3$ $\sigma = 9$

FIGURE 5.12: Two-dimensional Gaussians

Gaussian filters have a blurring effect which looks very similar to that produced by neighborhood averaging. Let's experiment with the cameraman image, and some different Gaussian filters.

```
>> g1 = fspecial('gaussian',[5,5]);
>> g1 = fspecial('gaussian',[5,5],2);
>> g1 = fspecial('gaussian',[11,11],1);
>> g1 = fspecial('gaussian',[11,11],5);
```

MATLAB/Octave

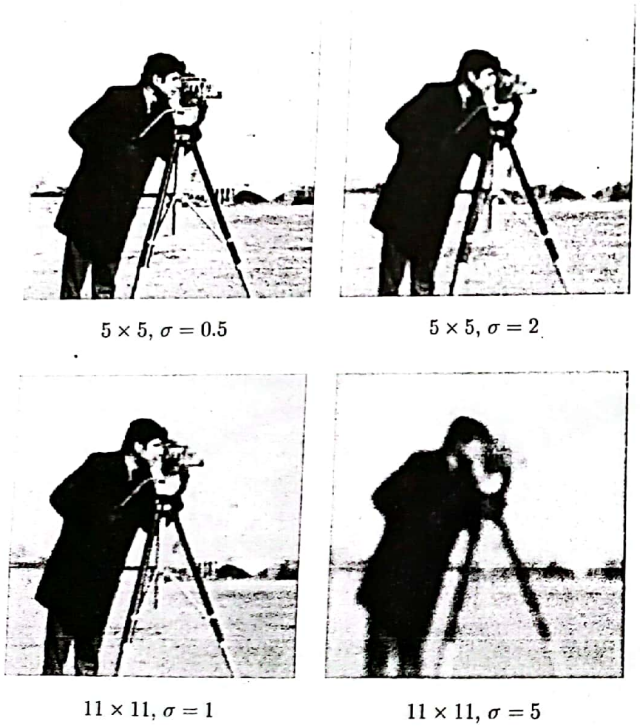
The final parameter is the standard deviation; which if not given, defaults to 0.5. The second parameter (which is also optional), gives the size of the filter; the default is 3×3. If the filter is to be square, as in all the previous examples, we can just give a single number in each case.

Now we can apply the filter to the cameraman image matrix `c` and view the result.

```
>> imshow(imfilter(c,g1))
>> figure,imshow(imfilter(c,g2))
>> figure,imshow(imfilter(c,g3,'symmetric'))
>> figure,imshow(imfilter(c,g4,'symmetric'))
```

MATLAB/Octave

As for the averaging filters earlier, the `symmetric` option is used to prevent the dark borders which would arise from low pass filtering using zero padding. The results are shown in Figure 5.13. Thus, to obtain a spread out blurring effect, we need a large standard deviation.



5×5, $\sigma = 0.5$ 5×5, $\sigma = 2$

11×11, $\sigma = 1$ 11×11, $\sigma = 5$

FIGURE 5.13: Effects of different Gaussian filters on an image

In fact, if we let the standard deviation grow large without bound, we obtain the averaging filters as limiting values. For example:


```
>> fspecial('gaussian',3,100)

ans =

    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
```

MATLAB/Octave

and we have the 3×3 averaging filter.

Although the results of Gaussian blurring and averaging look similar, the Gaussian filter has some elegant mathematical properties that make it particularly suitable for blurring.

In Python, the command `gaussian_filter` can be used, which takes as parameters the standard variation, and also a `truncate` parameter, which gives the number of standard deviations at which the filter should be cut off. This means that the images in Figure 5.13 can be obtained with:

```
In:  cg1 = ndi.gaussian_filter(c,0.5,truncate=4.5)
In:  cg2 = ndi.gaussian_filter(c,2,truncate=1)
In:  cg3 = ndi.gaussian_filter(c,1,truncate=5)
In:  cg4 = ndi.gaussian_filter(c,5,truncate=1)
```

Python

As with the uniform filter, the default behavior is to reflect the image in its edges, which is the same result as the `symmetric` option used in MATLAB/Octave. To see the similarity between MATLAB/Octave and Python, we can apply a Gaussian filter in Python to an image consisting of all zeros except for a central one. This will produce the same output as MATLAB's `fspecial` function:

```
>> fspecial('gaussian',[5,5],1)
ans =

   0.0029690   0.0133062   0.0219382   0.0133062   0.0029690
   0.0133062   0.0596343   0.0983203   0.0596343   0.0133062
   0.0219382   0.0983203   0.1621028   0.0983203   0.0219382
   0.0133062   0.0596343   0.0983203   0.0596343   0.0133062
   0.0029690   0.0133062   0.0219382   0.0133062   0.0029690
```

MATLAB/Octave

and the Python equivalent:

```
In:  x = ones((5,5)); x(2,2)=1
In:  ndi.gaussian_filter(x,1,truncate=2).round(7)
Out:
array([[ 0.002969 ,  0.0133062,  0.0219382,  0.0133062,  0.002969 ],
       [ 0.0133062,  0.0596343,  0.0983203,  0.0596343,  0.0133062],
       [ 0.0219382,  0.0983203,  0.1621028,  0.0983203,  0.0219382],
       [ 0.0133062,  0.0596343,  0.0983203,  0.0596343,  0.0133062],
       [ 0.002969 ,  0.0133062,  0.0219382,  0.0133062,  0.002969 ]])
```

Python

Other filters will be discussed in future chapters; also check the documentation for `fspecial` for other filters.

5.7 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 5.14.

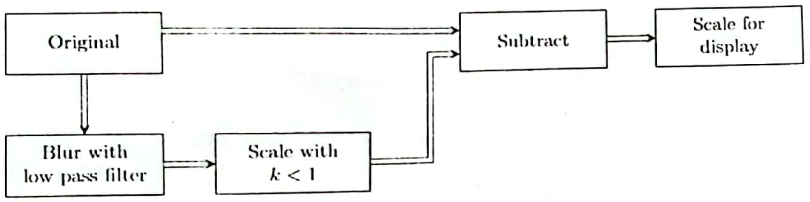


FIGURE 5.14: Schema for unsharp masking

In fact, all these steps can be built into the one filter. Starting with the filter

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

which is an "identity" filter (in that applying it to an image leaves the image unchanged), all the steps in Figure 5.14 could be implemented by the filter

$$u = s(id - \frac{1}{k}a)$$

where s is the scaling factor. To ensure the output image has the same levels of brightness as the original, we need to ensure that the sum of all the elements of u is 1, that is, that

$$s(1 - \frac{1}{k}) = 1$$

or that

$$s = \frac{k}{k-1}$$

Suppose for example we let $k = 1.5$. Then $s = 3$ and so the filter is

$$3\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - 2\left(\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}\right) = \frac{1}{9}\begin{bmatrix} -2 & -2 & -2 \\ -2 & 25 & -2 \\ -2 & -2 & -2 \end{bmatrix}$$

An alternative derivation is to write the equation relating s and k as

$$s - \frac{s}{k} = 1.$$

If $s/k = t$, say, then $s = t + 1$. So if $s/k = 2/3$, then $s = 5/3$ and so the filter is

$$\frac{5}{3}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{2}{3}\left(\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}\right) = \frac{1}{27}\begin{bmatrix} -2 & -2 & -2 \\ -2 & 43 & 2 \\ -2 & -2 & 2 \end{bmatrix}$$

For example, consider an image of an iconic Australian animal:

```
>> k = imread('koala.png');
```

MATLAB/Octave

The first filter above can be constructed as

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

   -0.22222   -0.22222   -0.22222
   -0.22222    2.77778   -0.22222
   -0.22222   -0.22222   -0.22222
```

MATLAB/Octave

and applied to the image in the usual way:

```
>> ku = imfilter(k,u);
>> imshow(ku)
```

MATLAB/Octave

The image k is shown in Figure 5.15(a), then the result of unsharp masking is given Figure 5.15(b). The result appears to be a better image than the original; the edges a crisper and more clearly defined.

(a) Original image (b) The image after unsharp masking

FIGURE 5.15: An example of unsharp masking

The same effect can be seen in Python, where we convolve with the image convert floats, so as not to lose any information in the arithmetic:

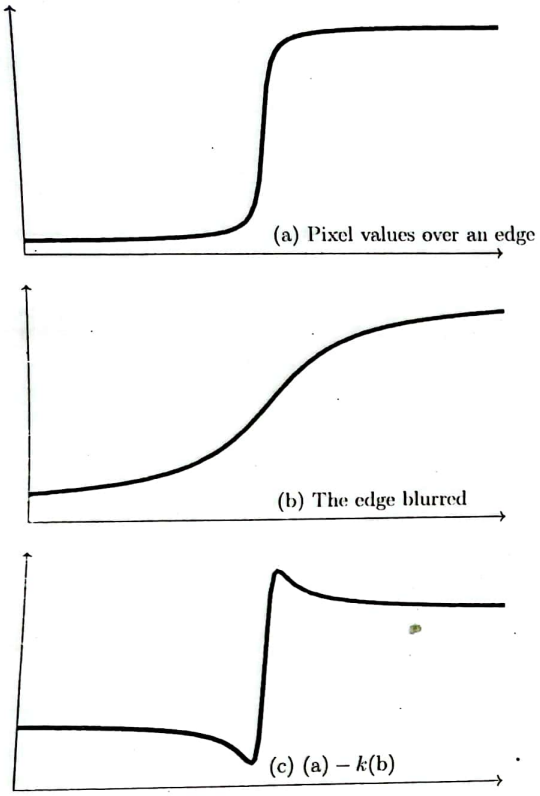


FIGURE 5.16: Unsharp masking

```
In:  k = io.imread('koala.png')
In:  u = array([[-2,-2,-2],[-2,25,-2],[-2,-2,-2]])/9.0
In:  ku = ndi.convolve(k.astype(float),u)
In:  io.imshow(ku/255,vmax=1.0,vmin=0.0)
```

Python

To see why this works, we may consider the function of gray values as we travel across an edge, as shown in Figure 5.16.

As a scaled blur is subtracted from the original, the result is that the edge is enhanced, as shown in graph (c) of Figure 5.16.

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.

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 5.17 was created using the MATLAB commands

```
>> p = imread('pelicans.png');
>> u = fspecial('unsharp',0.5);
>> pu = imfilter(p,u);
>> imshow(p),figure,imshow(pu)
```

MATLAB/Octave

Figure 5.17(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.

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

Exactly the same affect can be produced in Python; starting by creating an `unsharp` function to produce the filter:

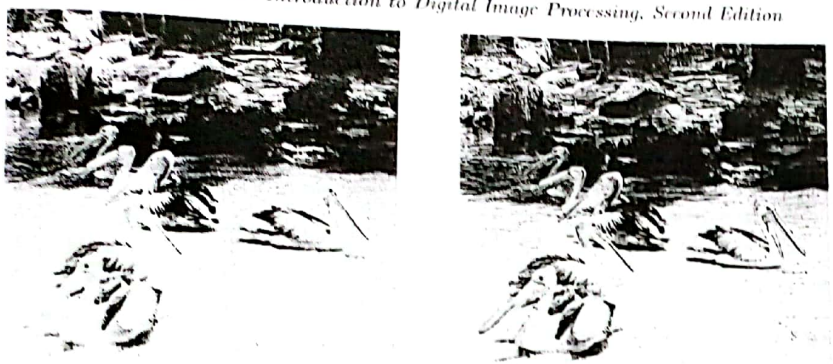
```
In :  def unsharp(alpha=0.2):
...:      A1 = array([[-1,1,-1],[1,1,1],[-1,1,-1]])
...:      A2 = array([[0,-1,0],[-1,5,-1],[0,-1,0]])
...:      return (alpha*A1+A2)/(alpha+1)
```

Python

This can be applied with `convolve`:

```
In:  p = io.imread('pelicans.png')
In:  u = unsharp(0.5)
In:  pu = ndi.convolve(p.astype(float),u)
In:  io.imshow(pu/255,vmax=1.0,vmin=0.0)
```

Python



(a) The original (b) After unsharp masking

FIGURE 5.17: Edge enhancement with 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 = imfilter(f,x);
>> imshow(xf/80)
```

MATLAB/Octave

will produce an image similar to that in Figure 5.15. 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:

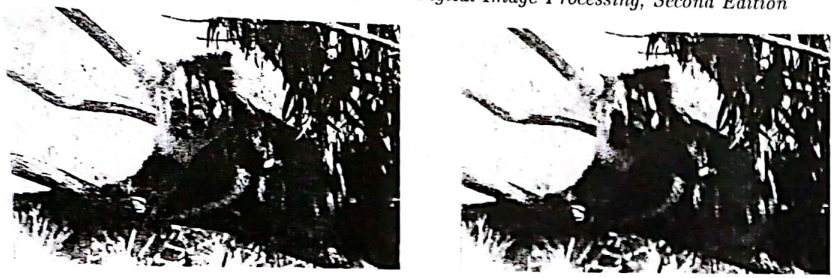
```
>> id=[0 0 0;0 1 0;0 0 0];
>> f=fspecial('average');
>> hb1=3*id-2*f

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
```

MATLAB/Octave

If each of the filters `hb1` and `hb2` are applied to an image with `imfilter`, the result will have enhanced edges. The images in Figure 5.18 show these results; Figure 5.18(a) was obtained with

```
>> k1 = imfilter(k,hb1);
>> imshow(k1)
```

MATLAB/Octave



(a) High boost filtering with `hb1` (b) High boost filtering with `hb2`

FIGURE 5.18: High boost filtering

and Figure 5.18(b) similarly.

With Python, these images could be obtained easily as:

```
In:  k = im2fl(io.imread('koala.png'))
In:  kf = ndi.uniform_filter(k,3)
In:  hb1 = 3*k - 2*kf
In:  hb2 = 1.25*k - 0.25*kf
In:  subplot(121),io.imshow(hb1,vmax=1.0,vmin=0.0)
In:  subplot(122),io.imshow(hb2,vmax=1.0,vmin=0.0)
```

Python

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

5.8 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.

A *non-linear filter* is obtained by a non-linear function of the grayscale 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 or Octave, 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 that defines it.

Here are some examples; first to implement a maximum filter over a 3×3 neighborhood (note that the commands are slightly different for MATLAB and for Octave):


```
>> cmax = nlfilter(c,[3,3],'max(x(:))');      # This is MATLAB
>> cmax = nlfilter(c,[3,3],@(x) max(x(:)));   # This is Octave
```

MATLAB/Octave

Python has a different syntax, made easier for maxima and minima in that the `max` function is applied to the entire array, rather than just its columns:

```
In:  cmax = ndi.generic_filter(c,max,[3,3])
```

Python

The `nlfilter` function and the `generic_filter` function each require three arguments: the image matrix, the size of the filter, and the function to be applied. The function must be a matrix function that returns a scalar value. The result of these operations is shown in Figure 5.19(a). Replacing `max` with `min` in the above commands implements a minimum filter, and the result is shown in Figure 5.19(b).

(a) Using a maximum filter (b) Using a minimum filter

FIGURE 5.19: 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 that are defined by their own commands. We shall investigate these in later chapters.

However, if a non-linear filter is needed in MATLAB or Octave, 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);
```

MATLAB/Octave

The parameter `sliding` indicates that overlapping neighborhoods 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/Octave 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));
```

MATLAB/Octave

and the minimum filter can be applied with

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

MATLAB/Octave

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));
```

MATLAB/Octave

However, the median filter has its own command, `medfilt2`, which we discuss in more detail in Chapter 8.

Python has maximum and minimum filters in the `scipy.ndimage` module and also in the `skimage.filter.rank` module. As an example of each:

```
In:  cmin = ndi.minimum_filter(c, size=(3,3))
In:  cmax = rk.minimum(c,ones((3,3)))
```

Python

Rank order filters are implemented by the `rank_filter` function in the `ndimage` module, and one median filter is also provided by `ndimage` as `median_filter`. Thus the two commands

```
In:  cm = ndi.median_filter(c,size=(3,3))
In:  cm2 = ndi.rank_filter(c,4,size=(3,3))
```

Python

produce the same results. (Recall that in Python arrays and lists are indexed starting with zero, so the central value in a 3×3 array has index value 4.) User-defined filters can be applied using `generic_filter`. For example, suppose we consider the *root-mean-square filter*, which returns the value

$$\sqrt{\frac{1}{N}\sum_{x \in M} x^2}$$

where M is the mask, and N is the number of its elements. First, this must be defined as a Python function:

```
In :  def rms(x):
...:      return sqrt(mean(x**2))
```

Python

Then it can be applied, for example, to the cameraman image `c`:

```
In : cr = ndi.generic_filter(c,rms,size=(3,3))
```

Python

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

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

where as for the root-mean-square filter M is the filter mask, and N 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

$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$$

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

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

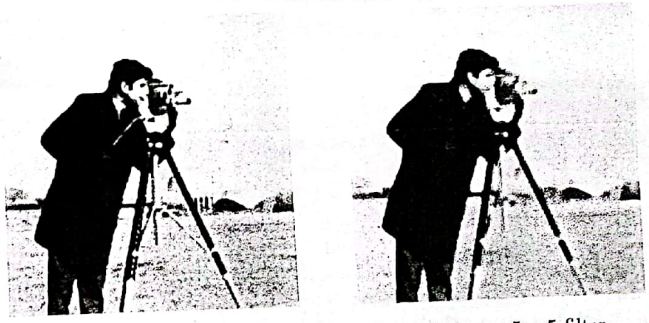
Both of these filters have uses for image restoration: again, see Chapter 8.

Non-linear filters are used extensively in image restoration, especially for cleaning noise; and these will be discussed in Chapter 8.

5.9 Edge-Preserving Blurring Filters

With the uniform (average) and Gaussian filters, blurring occurs across the entire image, and edges are blurred as well as backgrounds and other low frequency components. However, there is a large class of filters that blur low frequency components but keep the edges fairly sharp.

The median filter mentioned above is one such; we shall explore it in greater detail in Chapter 8. But just for a quick preview of its effects, consider median filters of size 3×3 and 5×5 applied to the cameraman image. These are shown in Figure 5.20. Even though



(a) Using 3×3 filter (b) Using a 5×5 filter

FIGURE 5.20: Using a median filter

much of the fine detail has gone, especially when using the larger filter, the edges are still sharp.

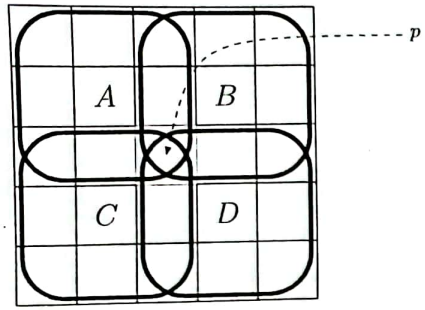


FIGURE 5.21: The neighborhoods used in the Kuwahara filter

Kuwahara Filters

These are a family of filters in which the variance of several regions in the neighborhood are computed, and the output is the mean of the region with the lowest variance. The simplest version takes a 5×5 neighborhood of a given pixel p looks at the four overlapping 3×3 neighborhoods of which it is a corner as shown in Figure 5.21.

The filter works by first computing the variances of the four neighborhoods around p, and then the output is the mean of the neighborhood with the lowest variance. So for example, consider the following neighborhood:

```
169  140  105  126  110
140   65  175  247   79
 40  178  240  171   37
 56   28  203   55   53
208  193   75  165  212
```

Then the variances of each of the neighborhoods can be found as

```
In:  x[0:3,0:3].var(), x[0:3,2:5].var(), x[2:5,0:3].var(), x[2:5,2:5].var()
```

Python

or as

```
>>> var(x(1:3,1:3)(:),1),var(x(1:3,3:5)(:),1),var(x(3:5,1:3)(:),1),...
    var(x(3:5,3:5)(:),1)
```

MATLAB/Octave

The variances of the neighborhoods A, B, C, and D will be found to be 3372.54, 4465.11, 5278.0, 5566.69, respectively. Of these four values, the variance of A is the smallest, so the output is the mean of A, which (when rounded to an integer) is 139.

None of MATLAB, Octave, or Python support the Kuwahara filter directly, but it is very easy to program it. Recall that for a random variable X, its variance can be computed as

$$\overline{X^2} - \left(\overline{X}\right)^2$$

where $\overline{(\cdot)}$ is the mean. This means that the variances in all 3×3 neighborhoods of an image can be found by first filtering $\mathtt{x}^2$ with the averaging filter, then squaring the result of filtering $\mathtt{x}$ with the averaging filter, and subtracting them:

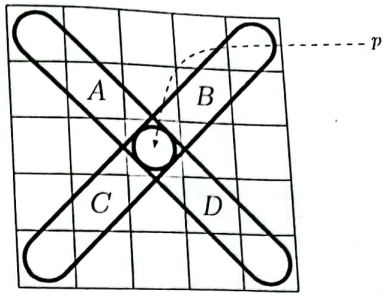


FIGURE 5.22: Alternative neighborhoods for a Kuwahara filter

```
>> cd = float(c);
>> cdm = imfilter(cd,ones(3)/9,'symmetric');
>> cd2f = imfilter(cd.^2,ones(3)/9,'symmetric');
>> cdv = cd2f - cdm.^2;
```

MATLAB/Octave

```
In:  cd = float32(c)
In:  cdm = ndi.uniform_filter(cd,(3,3))
In:  cd2f = ndi.uniform_filter(cd**2,(3,3))
In:  cdv = cd2f - cdm**2
```

Python

At this stage, the array `cdm` contains all the mean values, and `cdv` contains all the variances. At every point (i, j) in the image then:

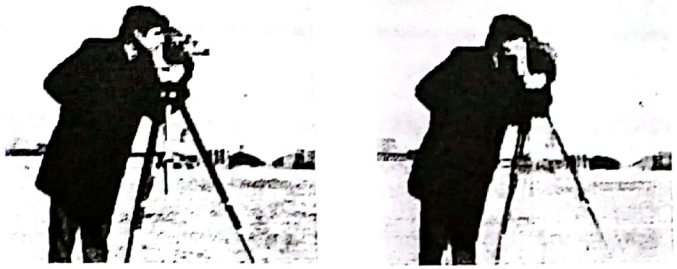
1. Compute the list
$$\text{vars} = [\mathtt{cdv}(i-1, j-1), \mathtt{cdv}(i-1, j+1), \mathtt{cdv}(i+1, j-1), \mathtt{cdv}(i+1, j+1)].$$
2. Also compute the list
$$\text{means} = [\mathtt{cdm}(i-1, j-1), \mathtt{cdm}(i-1, j+1), \mathtt{cdm}(i+1, j-1), \mathtt{cdm}(i+1, j+1)].$$
3. Output the value of "means" corresponding to the lowest value of "vars."

Programs are given at the end of the chapter. Note that the neighborhoods can of course be larger than 3×3, any odd square size can used, or indeed any other shape, such as shown in Figure 5.22. Figure 5.23 shows the cameraman image first with the Kuwahara filter using 3×3 neighborhoods and with 7×7 neighborhoods.

Note that even with the significant blurring with the larger filter, the edges are still remarkably sharp.

Bilateral Filters

Linear filters (and many non-linear filters) are *domain filters*; the weights attached to each neighboring pixel depend on only the *position* of those pixels. Another family of filters are the *range filters*, where the weights depend on the relative *difference* between pixel



(a) Using 3×3 neighborhoods (b) Using 7×7 neighborhoods

FIGURE 5.23: Using Kuwahara filters

values. For example, consider the Gaussian filter $e^{(-x^2-y^2)/2}$ (so with variance $\sigma^2 = 1$) over the region $-1 \leq x, y \leq 1$:

$$G = \begin{matrix} 0.36788 & 0.60653 & 0.36788 \\ 0.60653 & 1.00000 & 0.60653 \\ 0.36788 & 0.60653 & 0.36788 \end{matrix}$$

Applied to the neighborhood

$$N = \begin{matrix} 0.8 & 0.1 & 0.6 \\ 0.3 & 0.5 & 0.7 \\ 0.4 & 0.9 & 0.2 \end{matrix}$$

the output is simply the sum of all the products of corresponding elements of G and N.

As a *range filter*, however, with variance σ_r^2, the output would be the Gaussian function applied to the difference of the elements of N with its central value:

$$N - 0.5 = \begin{matrix} 0.3 & -0.4 & 0.1 \\ -0.2 & 0.0 & 0.2 \\ -0.1 & 0.4 & -0.3 \end{matrix}$$

For each of these values v the filter consists $e^{-v^2/2\sigma_r^2}$.

Thus, a domain filter is based on the *closeness* of pixels, and a range filter is based on the *similarity* of pixels. Clearly, the larger the value of σ_r the flatter the filter, and so there will be less distinction of closeness.

The following three arrays show the results with $\sigma_r = 0.1, 1, 10$, respectively:

$\sigma_r = 0.1$:

$$\begin{matrix} 0.01111 & 0.00034 & 0.60653 \\ 0.13534 & 1.00000 & 0.13534 \\ 0.60653 & 0.00034 & 0.01111 \end{matrix}$$

$\sigma_r = 1$:

$$\begin{matrix} 0.95600 & 0.92312 & 0.99501 \\ 0.98020 & 1.00000 & 0.98020 \\ 0.99501 & 0.92312 & 0.95600 \end{matrix}$$

$\sigma_r = 10$:

$$\begin{matrix} 0.99955 & 0.99920 & 0.99995 \\ 0.99980 & 1.00000 & 0.99980 \\ 0.99995 & 0.99920 & 0.99955 \end{matrix}$$

In the last example, the values are very nearly equal, so this range filter treats all values as being (roughly) equally similar.

The idea of the *bilateral filter* is to use *both* domain and range filtering, both with Gaussian filters, and with variances σ_d^2 and σ_r^2. For each pixel in the image, a range filter is created using σ_r^2, which maps the similarity of pixels in that neighborhood. That filter is then convolved with the domain filter which uses σ_d.

By adjusting the size of the filter mask, and of the variances, it is possible to obtain varying amounts of blurring, as well as keeping the edges sharp.

At the time of writing, MATLAB's Image Processing Toolbox does not include bilateral filtering, but both Octave and Python do: the former with the `bilateral` parameter of the `imsmooth` function; the latter with the `denoise_bilateral` method in the `restoration` module of `skimage`.

However, a simple implementation can be written, and one is given at the chapter end.

Using any of these functions and applied to the cameraman image produces the results shown in Figure 5.24, where in each case w gives the size of the filter. Thus, the image in Figure 5.24(a) can be created with our function by:

```
>> cb = bilateral(c,2,2,0,2);
```

MATLAB/Octave

where the first parameter w (in this case 2) produces filters of size $2w + 1 \times 2w + 1$.

Notice that even when a large flat Gaussian is used as the domain filter, as in Figure 5.24(d), the corresponding use of an appropriate range filter keeps the edges sharp.

In Python, a bilateral filter can be applied to produce, for example, the image in Figure 5.24(b) with

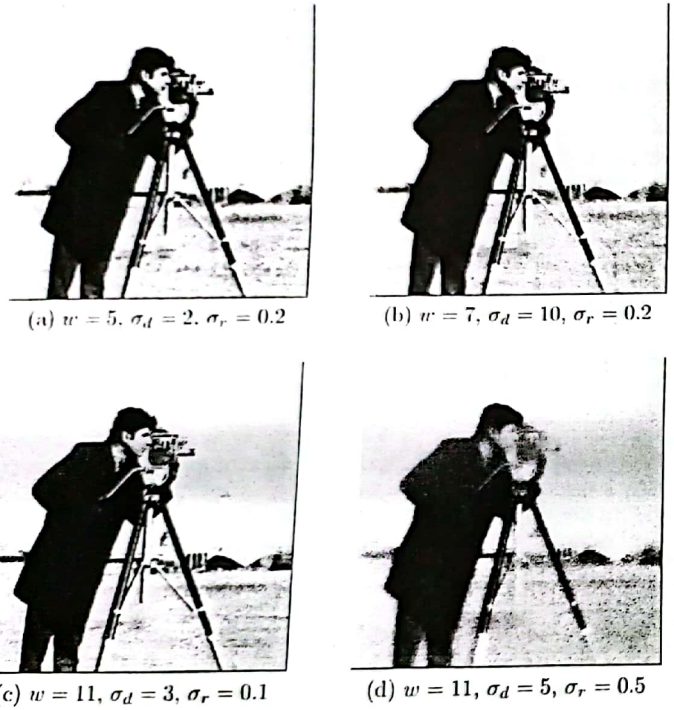
```
In : import skimage.restoration as re
In : cb = re.denoise_bilateral(c,win_size=7,sigma_range=0.2,\
...: sigma_spatial=10)
```

Python

The following is an example of the use of Octave's `imsmooth` function:

```
>> cb = imsmooth(c,'bilateral',sigma_d=2,sigma_r=0.1);
```

Octave



(a) $w = 5$, $\sigma_d = 2$, $\sigma_r = 0.2$ (b) $w = 7$, $\sigma_d = 10$, $\sigma_r = 0.2$

(c) $w = 11$, $\sigma_d = 3$, $\sigma_r = 0.1$ (d) $w = 11$, $\sigma_d = 5$, $\sigma_r = 0.5$

FIGURE 5.24: Bilateral filtering

and the window size of the filter is automatically generated from the σ values.

5.10 Region of Interest Processing

Often we may not want to apply a filter to an entire image, but only to a small region within it. A non-linear filter, for example, may be too computationally expensive to apply to the entire image, or we may only be interested in the small region. Such small regions within an image are called *regions of interest* or *ROIs*, and their processing is called *region of interest processing*.

Regions of Interest in MATLAB

Before we can process a ROI, we have to define it. There are two ways: by listing the coordinates of a polygonal region; or interactively, with the mouse. For example, suppose we take part of a monkey image:

```
>> m2 = imread('monkey.png');
>> m = m2(56:281,221:412);
```

MATLAB/Octave

and attempt to isolate its head. If the image is viewed with `impixelinfo`, then the coordinates of a hexagon that enclose the head can be determined to be (60, 14), (27, 38), (84, 127), (78, 177), (130, 160) and (139, 69), as shown in Figure 5.25. We can then define a region of interest using the `roipoly` function:

```
>> xi = [60 27 14 78 130 139]
>> yi = [14 38 127 177 160 69]
>> roi=roipoly(m,yi,xi);
```

MATLAB/Octave

Note that the ROI is defined by two sets of coordinates: first the columns and then the rows, taken in order as we traverse the ROI from vertex to vertex. In general, a ROI mask will be a binary image the same size as the original image, with 1s for the ROI, and 0s elsewhere. The function `roipoly` can also be used interactively:

```
>> roi=roipoly(m);
```

MATLAB/Octave

This will bring up the monkey image (if it isn't shown already). Vertices of the ROI can be selected with the mouse: a left click selects a new vertex, backspace or delete removes the most recently chosen vertex, and a right click finishes the selection.

Region of Interest Filtering

One of the simplest operations on a ROI is spatial filtering; this is implemented with th function `roifilt2`. With the monkey image and the ROI found above, we can experimen

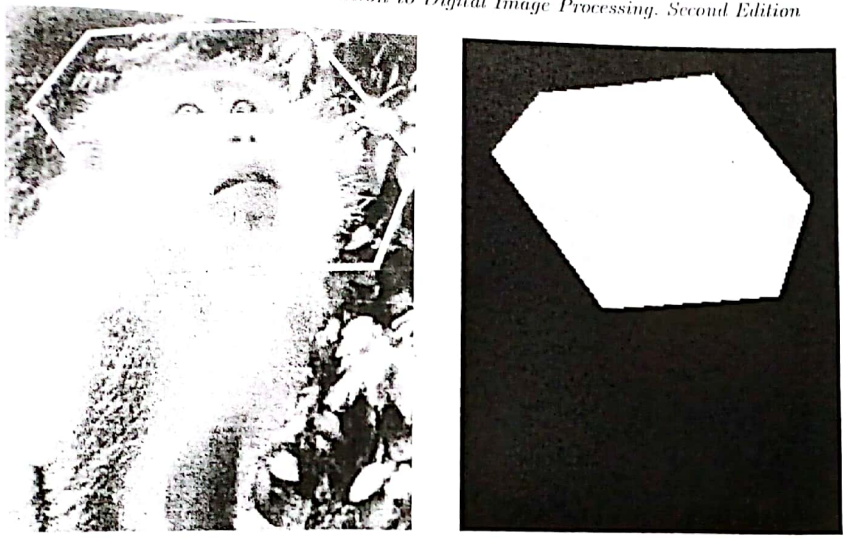


FIGURE 5.25: An image with an ROI, and the ROI mask

```
>> a = fspecial('average',15);
>> ma = roifilt2(a,m,roi);
>> u = fspecial('unsharp');
>> mu = roifilt2(u,m,roi);
>> l = fspecial('log');
>> ml = roifilt2(l,m,roi);
>> imshow(ma),figure,imshow(mu),figure,imshow(ml)
```

MATLAB

The images are shown in Figure 5.26.

Average filtering Unsharp masking Laplacian of Gaussian

FIGURE 5.26: Examples of the use of `roifilt2`

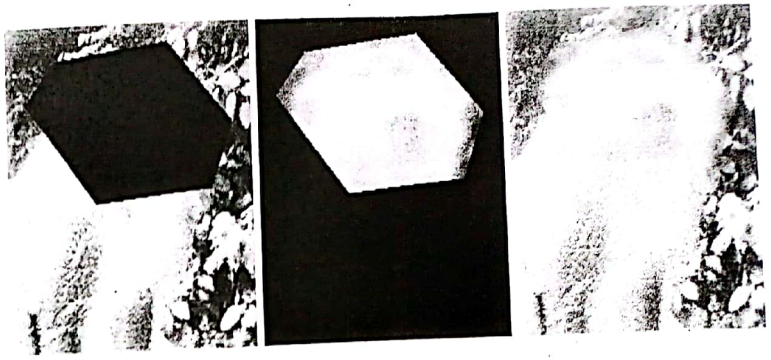


FIGURE 5.27: ROI filtering with a polygonal mask

Regions of Interest in Octave and Python

Neither Octave nor Python have `roipoly` or `roifilt2` commands. However, in each it is quite straightforward to create a mask defining the region of interest, and use that mask to restrict the act of filtering to the region.

With the monkey above, and the head region, suppose that the matrices for the image and the ROI are M and R, respectively. If M_f is the result of the image matrix after filtering, then

$$M_f R + M(1 - R)$$

will provide the result we want. This is shown in Figure 5.27 where the rightmost image is the sum of the other two.

In Octave, this can be achieved with:

```
> m2 = imread('monkey.png'); m = m2(56:281,221:412);
> [r,c] = size(m);
> xi = [60 27 14 78 130 139]
> yi = [14 38 127 177 160 69]
> roi = poly2mask(yi,xi,r,c);
> f = fspecial('gaussian',9,3);
> mg = imfilter(m,f));
> mr = imadd(mg.*roi,m.*~roi)
```

Octave

And in Python, using `zeros_like` which creates an array of zeros the same size as its input, as well as `polygon` from the `draw` module of `skimage`:

```
In: m2 = io.imread('monkey.png'); m = m2[55:281,220:412]
In: r,c = m.shape

In: xi = np.array([60,27,14,78,130,139])
In: yi = np.array([14,38,127,177,160,69])
In: roi = np.zeros_like(m)
In: r,c = polygon(yi,xi)
In: roi[c,r] = 1

In: mg = ut.img_as_ubyte(fl.gaussian_filter(m,g))
In: mr = mg*roi + m*(1-roi)
```

Python

5.11 Programs

This is a simple program (which can be run in MATLAB or Octave) for bilateral filtering.

```
function out = bilateral(im,w,sigma_d,sigma_r)
  im = im2double(im);
  [r,c] = size(im);
  out = zeros(r,c);
  A = padarray(im,[w,w],'symmetric');
  G = fspecial('gaussian',2*w+1,sd);          # the domain filter
  for i = 1+w:r+w-1
      for j = 1+w:c+w-1
          R = A(i-w:i+w,j-w:j+w);              # region to be computed
          H = exp(-(R-A(i,j)).^2/(2*sr^2));   # the range filter
          F = H.*G;
          out(i-w,j-w) = sum(F(:).*R(:))/sum(F(:));
      end;
  end;
  close(h);
end
```

MATLAB/Octave

Exercises

1. The array below represents a small grayscale 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 zeros.)

```
20  20  20  10  10  10  10  10  10
20  20  20  20  20  20  20  20  10
20  20  20  10  10  10  10  20  10
20  20  10  10  10  10  10  20  10
```



```
20  10  10  10  10  10  10  20  10
10  10  10  10  20  10  10  20  10
10  10  10  10  10  10  10  10  10
20  10  20  20  10  10  10  20  20
20  10  10  20  10  10  20  10  20
```

(a)
```
-1  -1   0
-1   0   1
 0   1   1
```

(b)
```
 0  -1  -1
 1   0  -1
 1   1   0
```

(c)
```
-1  -1  -1
 2   2   2
-1  -1  -1
```

(d)
```
-1   2  -1
-1   2  -1
-1   2  -1
```

(e)
```
-1  -1  -1
-1   8  -1
-1  -1  -1
```

(f)
```
1  1  1
1  1  1
1  1  1
```

(g)
```
-1  0  1
-1  0  1
-1  0  1
```

(h)
```
 0  -1   0
-1   4  -1
 0  -1   0
```

2. Check your answers to the previous question with using `imfilter` (if you are using MATLAB or Octave), or `ndi.correlate` if you are using Python.

3. Describe what each of the masks in the previous question might be used for. If you can't do this, wait until Question 5 below.

4. Devise a 3×3 mask for an "identity filter," which causes no change in the image.

5. Choose an image that has a lot of fine detail, and load it.

   Apply all the filters listed in Question 1 to this image. Can you now see what each filter does?

6. Apply larger and larger averaging filters to this image. What is the smallest sized filter for which the fine detail cannot be seen?

7. Repeat the previous question with Gaussian filters with the following parameters:

Size	Standard deviation		
[3,3]	0.5	1	2
[7,7]	1	3	6
[11,11]	1	4	8
[21,21]	1	5	10

   At what values do the fine details disappear?

8. Can you see any observable difference in the results of average filtering and of using a Gaussian filter?

9. If you are using MATLAB or Octave, read through the help page of the `fspecial` function, and apply some of the other filters to the cameraman image and to the mandrill image.

10. Apply different Laplacian filters to an image of your choice at and to the cameraman images. Which produces the best edge image?

11. Is the 3×3 median filter separable? That is, can this filter be implemented by a 3×1 filter followed by a 1×3 filter?

12. Repeat the above question for the maximum and minimum filters.

13. Apply a 3×3 averaging filter to the middle 9 values of the matrix

$$\begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{bmatrix}$$

and then apply another 3×3 averaging filter to the result.

Using your answer, describe a 5×5 filter that has the effect of two averaging filters. Is this filter separable?

14. Use the appropriate commands to produce the outputs shown in Figure 5.5, starting with the diagonally increasing image

```
 20   40   60   80  100
 40   60   80  100  120
 60   80  100  120  140
 80  100  120  140  160
100  120  140  160  180
```

15. Display the difference between the `cmax` and `cmin` images obtained in Section 5.8. You can do this with an image subtraction.

What are you seeing here? Can you account for the output of these commands?

16. If you are using MATLAB or Octave, then use the `tic` and `toc` timer functions to compare the use of `nlfilter` and `colfilt` functions. If you are using the ipython enhanced shell of Python, try the `%time` function

17. Use `colfilt` (MATLAB/Octave) or `generic_filter` (Python) to implement the geometric mean and alpha-trimmed mean filters.

18. If you are using MATLAB or Octave, show how to implement the root-mean-square filter.

19. Can unsharp masking be used to reverse the effects of blurring? Apply an unsharp masking filter after a 3×3 averaging filter, and describe the result.

20. Rewrite the Kuwahara filter as a single function that can be applied with either `colfilt` (MATLAB/Octave) or `generic_filter` (Python).

Chapter 6

Image Geometry

There are many situations in which we might want to change the shape, size, or orientation of an image. We may wish to enlarge an image, to fit into a particular space, or for printing; we may wish also to reduce its size, say for inclusion on a web page. We might also wish to rotate it: maybe to adjust for an incorrect camera angle, or simply for affect. Rotation and scaling are examples of *affine transformations*, where lines are transformed to lines, and in particular parallel lines remain parallel after the transformation. Non-affine geometrical transformations include warping, which we will not consider.

6.1 Interpolation of Data

We will start with a simple problem: suppose we have a collection of 4 values, which we wish to enlarge to 8. How do we do this? To start, we have our points x_1, x_2, x_3, and x_4, which we suppose to be evenly spaced, and we have the values at those points: $f(x_1)$, $f(x_2)$, $f(x_3)$, and $f(x_4)$. Along the line $x_1 \ldots x_4$ we wish to space eight points $x'_1, x'_2, \ldots, x'_8$. Figure 6.1 shows how this would be done.

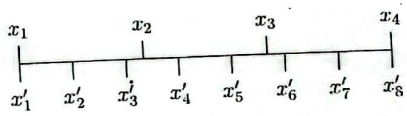


FIGURE 6.1: Replacing four points with eight

Suppose that the distance between each of the x_i points is 1; thus, the length of the line is 3. Thus, since there are seven increments from x'_i to x'_8, the distance between each two will be $3/7 \approx 0.4286$. To obtain a relationship between x and x' we draw Figure 6.1 slightly differently as shown in Figure 6.2. Then

$$x' = \frac{1}{3}(7x - 4),$$
$$x = \frac{1}{7}(3x' + 4).$$

As you see from Figure 6.1, none of the x'_i coincides exactly with an original x_j, except for the first and last. Thus we are going to have to "guess" at possible function values $f(x'_i)$. This guessing at function values is called *interpolation*. Figure 6.3 shows one way of doing this: we assign $f(x'_i) = f(x_j)$, where x_j is the original point closest to x'_i. This is called *nearest neighbor interpolation*.