- 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?

---

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

$$\text{Pixel info: } (c, r)\ p$$

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')
```

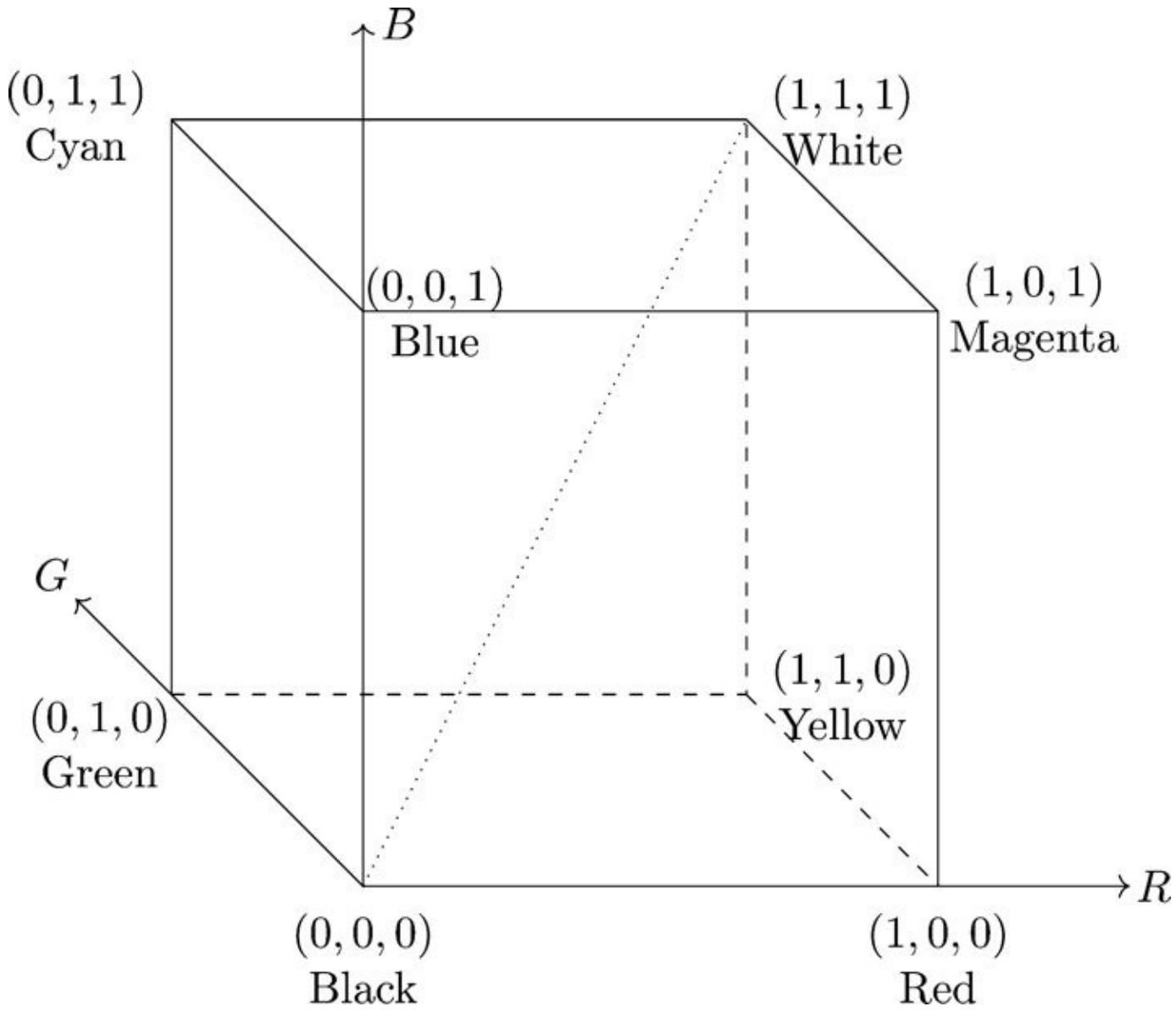
Python

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

Figure 2.3: The color cube for the RGB color model



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

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

MATLAB

Note now that the pixel values consist of a list of three values, giving the red, green, and blue 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:

$$\text{Pixel info: } (61, 109)\ \langle 37 \rangle\ [0.58\ 0.54\ 0.51]$$

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.

1

Available from http://www.sno.phy.queensu.ca/~phil/exiftool/

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”

Table 2.1 Some numeric data types

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

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. 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.

Table 2.2 Converting images in MATLAB and Octave

Function	Use	Format
`ind2gray`	Indexed to grayscale	`y=ind2gray(x,map);`

Function	Use	Format
`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);`

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.

Table 2.3 Converting images in Python

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)}`

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.

Figure 2.4: The start of a PGM file

```
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 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.

Bytes	Information	Description
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}.

Bytes	Information	Description
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 x 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."

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

Bytes	Information	Description
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.

Bytes	Information	Description
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 (which 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:

```
function hexdump(filenm, n)
% hexdump(filenm, n)
% Print the first n bytes of a file in hex and ASCII.
fid = fopen(filenm, 'r');
if (fid<0) disp(['Error␣opening␣',filenm]); return; 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 = 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-3:4*i);
    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(nread-16,6)), hexdisp,
        ascstring);
end;
fclose(fid);
```

MATLAB/Octave

Now Python:

```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] # all the bytes as 2
        character hex strings
    hl2 = hl +(-len(hl)%16)*['␣␣']  # fill up to a multiple of width 16
    asc = ['.']*num
    for i in range(num):            # creates ascii values (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 multiple 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] for i in range(8)]),
        print '␣|'+''.join([asc[16*n+i] for i in range(16)])+'|'
```

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  ................
```

2

http://www.imagemagick.org/

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