18 Images

Contents

Biological images

Often in biology and medicine the data people use comes in the form of an image. This could be as simple as a photograph of some cells or an image that has been constructed from other data, e.g. from an MRI scan. The images that we will be discussing in this chapter, whatever their source, will be *pixmap* images, also known as *raster* images. They will be constructed as rectangular arrays of colour or grey values, the smallest square element of which we refer to as a *pixel*. We will not be considering the *vector graphics* approach to making pictures, where the data is described in terms of lines and shape outlines. Here we will concentrate on pixel arrays, the kind of image data that comes from our digital cameras and various scientific instruments.

We will deal with pixmap images in a general, slightly mathematical way. It will not matter what the image actually represents for the most part, although we will endeavour to give examples with a biological flavour. Not so long ago images would largely be acquired by using photographic film, but now the digital camera is ubiquitous, and without the need to buy expensive film a scientist can capture as many images as time and storage capacity allow. Thus the examples presented here will often have an emphasis towards automation, and if you need to write programs dealing with biological data this will allow you to construct efficient analytical pipelines.

Pixmaps

A pixmap image can be stored in a variety of different ways on a computer, such as the common file formats like JPEG, PNG or GIF. However, whatever the means of storage, which is often just a cunning way of saving space (or download bandwidth), all pixmaps can be imagined as an array of different colour values. Here the usual convention is that the first pixel (array position 0, 0) is viewed as the top left of the image, i.e. the other pixels go right and down relative to the first. A pixmap image will have no resolution as such, just one fixed size in terms of points in a matrix; how big it ends up looking is a matter for the display or printer. Each pixel element of such an array will have its own colour specification and placing these all together makes the whole image. There are several common ways of representing colour in computing, some of which are described below. The basic principle is that one or more numbers are allocated to each pixel and these describe the components or properties of the colour. It is then up to the display (or printer) to know how to interpret the pixel's values and to show the colours correctly.

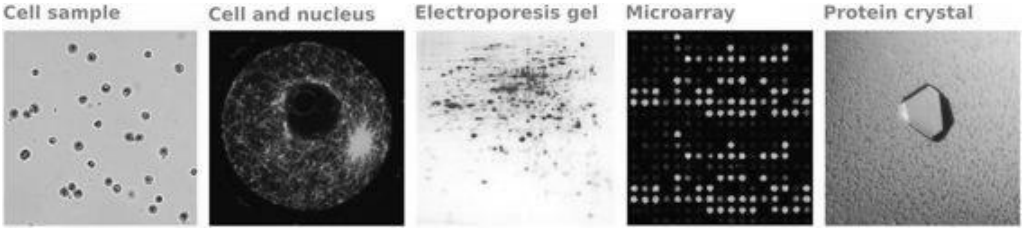

Figure 18.1 (Plate 5). Examples of a variety of different kinds of images used in biology. Shown from left to right are: a microscope image of a mammalian cell culture (courtesy Dr. Anja Winter, University of Leicester); a red-green fluorescence microscope image of an oocyte and its nucleus (courtesy Dr. Melina Schuh, MRC Laboratory of Molecular Biology); a two-dimensional electrophoresis gel of a plant proteome (courtesy Prof. Paul Dupree, University of Cambridge); an image of a DNA microarray (courtesy Karen Howarth, University of Cambridge); a protein crystal that has been grown for structure determination by X-ray crystallography (courtesy Dr. Aleksandra Watson, University of Cambridge).

A few of the more common colour models used in computing:

- Greyscale: each pixel is represented by a single value, which determines how bright it is. Zero will represent black and the maximum value will be white, with the grey shades in between. Sometimes greyscale is referred to as *luminance* (although this has a proper meaning in physics).
- RGB: represents each pixel with three numbers which specify the amount of red (R), green (G) and blue (B) component colours that are in the pixel. The mixtures of these components specify other colours. This is similar to the way that most computer screens operate.
- RGBA: this is the same as RGB, but carries an extra number for each pixel called the *alpha* (A) value, which specifies how transparent it is; this is only really useful when making things pretty and overlaying images, to say how much of the background comes through. This is certainly a form to be aware of but not something we usually have to think about too much for science.
- CMYK: represents each pixel with four numbers indicating cyan (C), magenta (M),

yellow (Y) and black (K) components. This is a specification useful for printing, where the components match the colours of inks (which are better for mixing on paper than red, green and blue).

- HSV: represents each pixel in terms of hue (H), saturation (S) and value (V). The hue indicates where the pure colour lies in a rainbow (or colour wheel), the saturation specifies how colourful the pixel is compared to grey, and the value says how dark (close to black) the colour is.

A technical aspect that will impinge on our ability to deal with images is the way that different number ranges are used in different circumstances. Thinking about RGB images, we can imagine the pixels' red, green and blue components as taking values between 0.0 (minimum) and 1.0 (maximum), and this may be convenient for us when doing mathematical manipulations. However, such components are not generally held as floating point values between zero and one, rather they are stored as integers. For example, they commonly range from zero up to 255. For RGB this means using 8 bits for each colour (2^8 = 256), which in turn gives rise to the whole image being described as 24-bit (8 red + 8 green + 8 blue). Naturally allowing values to be stored as larger numbers takes up more memory but allows for many more gradations, and so better colour representation. In order to interpret image data correctly we must know what this maximum value is, i.e. whether it is 8-bit, 16-bit etc., otherwise the data will be nonsense.

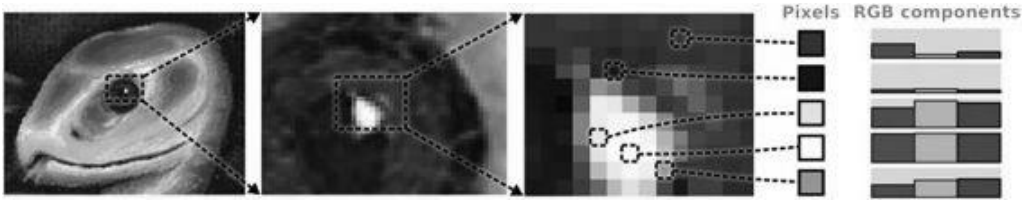

Figure 18.2 (Plate 6). An image, its component pixels and their RGB colour-space values. A section of the book cover picture is shown magnified at different levels to reveal the array of square pixels that the digital image is composed of. For the highest magnification, example pixels with different colours are selected and the component red, green and blue (RBG) values that constitute each colour are shown as histograms.

Once we know how image data is stored we will need to be able to manipulate it. In this chapter we will generally be working with greyscale or the RGB colour model for simplicity, but we will illustrate how to convert to and from the other representations. When performing more mathematical operations with numeric Python we will think of the pixels in an image as being elements of a matrix (a 2D array) and often we can think of the different red, green and blue components as separate layers (also called *channels*) each with a separate matrix, which are then arranged depth-wise to make the full pixmap.

Image manipulation

Much of what is covered in this chapter is about manipulation of pixmaps. Naturally when making changes we have to be mindful of biasing the scientific investigation. It is important to be objective so that we show what is actually there, not just what we expect or what looks pretty. Of course this is not different in principle to any other scientific data, but it is often very easy to manipulate an image and forget to keep the original data. Hence we encourage keeping the original data in important situations. This also allows the

development of better analytic methods in the future.

Sometimes when manipulating images we will be working with the pixel data directly in numeric Python. However, such low-level coding is not always required. When performing common operations on an image, such as adjusting contrast or colour balance, we can make good use of existing high-level graphics libraries that work with Python such as the *Python Imaging Library*[1] (PIL) and the Python wrapper to *Imagemagick*.[2] Although these libraries are both worth considering we will focus mainly on PIL in this chapter. Generally we will use PIL for high-level functions, especially the ability to display, load and save images, and then use numeric Python for the detailed work. However, for some operations we will illustrate using both PIL and numeric Python and leave it up to the reader to choose which is most convenient or useful.

Basic image operations

The first Python examples in this chapter will illustrate how we can use PIL to get hold of pixmaps from the data stored in various kinds of image file. The data can then be manipulated, if required, and saved back again, potentially to a different type of image file. When working with images we often simply need to convert from one type of image file to another, given computer programs are sometimes particular about the format. Also, by converting we can sometimes improve storage efficiency, e.g. going from an uncompressed TIFF to JPEG.

Python Imaging Library

To work with PIL, the following examples naturally assume that the library is installed.[3] Then to begin we import the Image module, which will allow us to construct Image class objects.[4].Inbuilt into this kind of object are lots of useful operations that we can access directly for the image just by calling a function (i.e. a bound method) on the object. The image module can use the .open() method to load a file from disk and make an Image object. Note that the PIL function automatically guesses at the type of file at load time so that we only have to specify a file name. The example files we are working with here are available in the downloadable data that supports this book.

```
from PIL import Image

img = Image.open('examples/Cells.jpg')
```

With the image object made we can access its properties, and most importantly call the .show() method to display it on screen.

```
print(img.size)
print(img.mode)

img.show()
```

A given image object can be saved back to a file using several different file formats. In the example below we use PNG and GIF format. At the time of writing PIL can use any of

the following formats: BMP, DCX, EPS, GIF, IM, JPEG, PCD, PDF, PNG, PPM, PSD, TIFF, XBM and XPM. Not all of these will store images in the same way. For the common web formats, JPEG gives the smallest files, but will change the data and may lose quality (it uses *lossy compression*), PNG will preserve all the data but the files will be larger (*lossless compression*). GIF is similar to PNG, but can only handle 256 colours at once (although this *palette* of colours can be chosen from a larger set) so if the image has more colours saving as GIF will lose information.

```
img.save('Cells.png', 'PNG')
```

If we need to have an image which describes its pixel values in a different way we can use the .convert() function prior to saving, or some other operation. It is notable that converting to greyscale (code ‘L’) in PIL takes account of the sensitivity of the human eye to colours where, for the same physical intensity, green seems brightest followed by red and then blue. Thus, such a greyscale conversion preserves aesthetic brightness, but this will be a biased average of the pixel values and may not be what we want scientifically.

```
img.convert('CMYK')  #  Cyan, Magenta, Yellow, blacK
img.convert('L')    #  Luminance = greyscale
```

Next we will run though a few of the more basic ways of changing images, which we can admire by using .show() or by saving and viewing in another program. The .crop() method chops the edges off the pixmap. We need to specify the rectangle to use as left, top, right and bottom edges (in order) and pass these values as arguments in a tuple, not separately. The convention used in the Image object is that the pixel with positional indices (0,0) is at the top left. Accordingly, in the example the right and bottom edge points are calculated by subtracting from the original width and height. Also, note that we are sending the result back to a variable called img, thus we are overwriting the original data, but of course we are free to use a different name if required.

```
w, h = img.size
img = img.crop((10, 10, w-10, h-10))
```

Another easy manipulation is rotation, which is inbuilt. Here we specify the angle of rotation in degrees and then save the rotated pixmap. As before, we are overwriting img with new data.

```
img = img.rotate(270)
img.save('CellsAdj.png', 'PNG')
```

To change the size of an image we have one of two options: the first method is .resize(), which gives back a new image, preserving the one we passed in. For this operation we need to say how the image will be resized (how to combine the original pixels together to make the new array). Thus we enter what the new width and height will be: here half of the original values. Also, we can optionally supply a resizing method, which specifies which algorithm will be used. The example uses the Image.ANTIALIAS option (coming directly from the module Image, not the object). Antialiasing is commonly what you would want for making smaller images, though .BILINEAR, .BICUBIC or .NEAREST can all be used for resizing in general.

```
img2 = img.resize((w/2, h/2), Image.ANTIALIAS)
img2.save('CellsHalfSize.png', 'PNG')
```

An alternative way of resizing images is to make smaller preview versions called *thumbnails*. Unlike the previous examples, the thumbnail() function actually changes the image in-place. If we do not want the original to be affected we need to make a copy first, which is fortunately easy:

```
img2 = img.copy()
img2.thumbnail((50, 50), Image.ANTIALIAS)
img2.save('CellsThumb.png', 'PNG')
```

Using NumPy for images

Next we move on from the inbuilt PIL methods and place the image data into a numeric Python array. Naturally this array will have the same width and height as the image, i.e. we have an array element for each pixel. The array will have different depths depending on the kind of image data that is being interpreted. For example, if the image is greyscale (‘black and white’), then we need an array that is only one element deep, to contain the brightness value. If the image is RGB, then we need an array that is three deep; you can imagine this pixmap as consisting of three stacked planes, for red, green and blue layers respectively. Or put another way, a pixel that makes an RGB element is a vector of the form (red, green, blue).

For the following example we will first make the NumPy imports that will be needed. Many of these will be familiar, but uint8 warrants some explanation. The uint8 object represents a data type that specifies *unsigned*[5] 8-bit integer numbers. Most integers in Python will be 32-bit and have signs. In essence a uint8 number will only go from 0 to 255, and this is how many RGB image values are actually stored. To interpret the pixmap data we will need to use this data type, rather than the regular Python number types.

```
from numpy import array, dot, dstack, ones, random, uint8, zeros
```

To make an image from scratch we make an array of the required size. Here we specify a height and width of 200 pixels and use a depth of three, because we want to make RGB pixmaps. Note that when we make the array first we use height h, then width w, then depth d. This may seem counterintuitive to some people, given the custom for using x, y order in coordinates and stating dimensions as width × height. However, putting height first better reflects the way that pixels are stored in image files, and so matches what we will pass on to PIL at the end to save the image.

```
h = 200
w = 200
d = 3
```

The pixmap is constructed as an array using these sizes, and the values that we put in the array will specify the colour of the pixels. All zeros will give rise to black, all ones will become white and random could be anything.

```
pixmap = zeros((h, w, d)) # black
pixmap = ones((h, w, d)) # white
```

```
pixmap = random.random((h, w, d)) # random colours
pixmap *= 255
```

The arrays constructed above contain values that go from zero to one, but the RGB values will go from 0 to 255. Thus, in the above example *= 255 multiplies each value in the array by 255.

Also, the Python arrays usually contain standard integers or floating point numbers, not the required 8-bit variety, so we have to explicitly convert the data type to uint8. The converted array can be directly interpreted by PIL to make an Image object which we can show on screen, save to file or whatever. Here we encapsulate the conversion operations into a function, pixmapToImage, so we can use it later. In the function we check to make sure pixel values do not exceed 255, scaling them back if they do:

```
def pixmapToImage(pixmap, mode='RGB'):

  if pixmap.max() > 255:
    pixmap *= 255.0 / pixmap.max()

  pixmap = array(pixmap, uint8)
  img = Image.fromarray(pixmap, mode)

  return img

img1 = pixmapToImage(pixmap)
img1.show()
```

Next we will consider the construction of images by combining separate matrices for the red, green and blue image components. So, for example, if we wanted to make a pure yellow image, which has maximum red and green components, we can do the following to make component matrices of the same size and use dstack to combine them in the right manner:

```
size = (h,w)
redMatrix   = ones(size)
greenMatrix = ones(size)
blueMatrix  = zeros(size)

pixmap = dstack([redMatrix, greenMatrix, blueMatrix])
pixmap *= 255

img1 = pixmapToImage(pixmap)
img1.show()
```

In order to copy existing PIL image data into a numeric Python array the handy .getdata() method can be used, which is built into all PIL Image objects. It is notable that the unit8 is used again so that the numbers in the image are of the unsigned 8-bit data type. Also, we put in a .convert() step to check that the image is of the RGB type. However, it should be noted that our pixmap/image conversion functions should perform many more checks if they were used in real-world scenarios. Initially the data will be a plain list of colour data; all pixels will effectively be in a long line. To reconstruct the original pixmap’s dimensions reshape can be used to rearrange the elements, remembering that the

width and height are used in the opposite order to what .size gives.

```
def imageToPixmapRGB(img):

  img2 = img.convert('RGB')
  w, h = img2.size
  data = img2.getdata()
  pixmap = array(data, uint8)
  pixmap = pixmap.reshape((h,w,3))

  return pixmap
```

The loaded image pixmap can now be manipulated as required, although it will be convenient to use PIL to actually visualise what is happening.

As an example of working with image data stored in arrays, below we perform an operation which may be useful to red-green colour-blind people. The notion here is that some types of biological images are bright red and green, because of special fluorescent marker compounds, and must be converted (here to yellow and blue) so that a red-green colour-blind person can view them effectively (see Figure 18.3 for an illustration). The image is loaded and converted to a pixmap in the manner described:

```
img = Image.open('examples/CellNucleusRedGreen.png')
pixmap = imageToPixmapRGB(img)
```

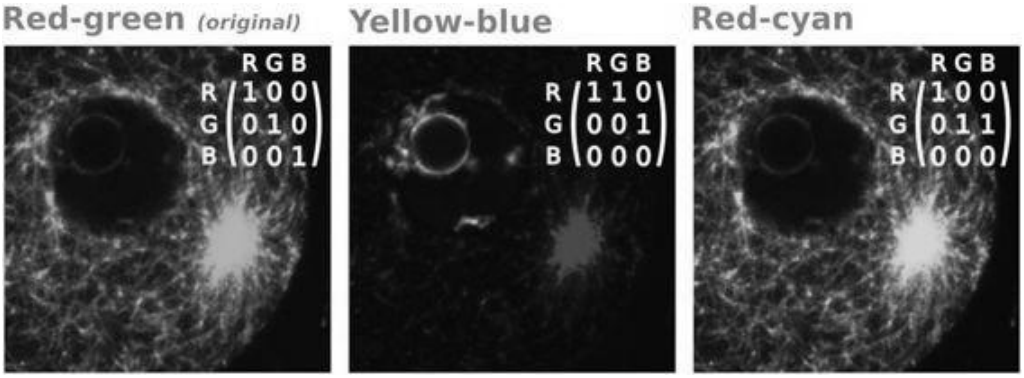

Figure 18.3 (Plate 7). Matrix transformations of pixmap colours. A red-green coloured (i.e. two-channel) fluorescence microscope image of a cell is shown alongside colour-adjusted yellow-blue and red-cyan versions. Inset in each image is the RGB colour transformation matrix relative to the red-green image.

Here it is notable that there is an alternative approach using SciPy (a scientific Python package which is often installed alongside NumPy): the imread() function is provided in the ndimage module, to create an array directly from the file data. Though it may be convenient to avoid PIL, without an Image object we cannot invoke show() to easily visualise the loaded data. Also, doing things this way will include the ‘alpha’ transparency layer, so to get the same kind of array as above we take a slice of the first three (red, green and blue) colour layers with [:,:,:3].

```
from scipy import ndimage
pixmap = ndimage.imread('examples/CellNucleusRedGreen.png')
pixmap = pixmap[:,:,:3]
```

The next task is to define a colour transformation matrix. The way to think of this is

that the three rows dictate how to modify red, green and blue respectively and the [r, g, b] values within each row specify what the colour will become. Accordingly, in transform the first row specifies that the red channel will be transformed into equal red and green (i.e. yellow), the second row means that the green channel will become blue and the third row is all zero and so any original blue is removed entirely (not that there is much in the test image). The transformation is applied using a dot product, remembering that the order of the arguments is important.

```
transform = array([[1.0, 1.0, 0.0],
                   [0.0, 0.0, 1.0],
                   [0.0, 0.0, 0.0]])

pixmap2 = dot(pixmap, transform)

img2 = pixmapToImage(pixmap2)
img2.show()
```

Pixmap data can be manipulated as floating point numbers, rather than just integers in the range 0 to 255. In the next example the pixmap is converted to an array of float data type, so the numbers fall in the range 0.0 to 255.0. Dividing by 255.0 the range becomes 0.0 to 1.0, which is handy for the next step, where the values of the pixmap are squared (an element-by-element operation). Squaring colour values in the range zero to one is a convenient way of changing the brightness of the image; values will move towards zero, but the upper limit is still 1.0. With the brightness adjusted the array can be converted back into the range 0 to 255 to go back to PIL.

```
pixmap = array(pixmap, float)
pixmap /= 255.0
pixmap = pixmap ** 2
pixmap = array(255*pixmap, uint8)

img2 = pixmapToImage(pixmap)
img2.show()
```

Adjustments and filters

The previous example which adjusts the brightness of a pixmap leads neatly into thinking about more general ways that we can change the global properties of an image. Helpfully many kinds of adjustment, for contrast, brightness, sharpness etc., can be made directly in PIL. This is more convenient than doing things in numeric Python arrays. Nonetheless, we will go on to show some of the equivalent operations with arrays, because this gives more control and teaches something about how the adjustments work at a low level.

The `ImageEnhance` module

To adjust PIL images the ImageEnhance module is imported, as well as the Image used earlier, and then an example image loaded to experiment with:

```
from PIL import Image, ImageEnhance
img = Image.open('examples/Cells.jpg')
```

Firstly, we will use this image enhancement module to change the contrast in the image. The way that this works with PIL is to create a processing object (here processObj) for a given image, which is then called with the parameters needed to control the image adjustment. Here we make a processing object to adjust the contrast and then call its .enhance() method with a value of 2.0. For all of the kinds of ImageEnhance enhancement calls, a value of 1.0 will preserve the image as it was. Thus here we are increasing the contrast:

```
processObj = ImageEnhance.Contrast(img)
img2 = processObj.enhance(2.0)
img2.show()
```

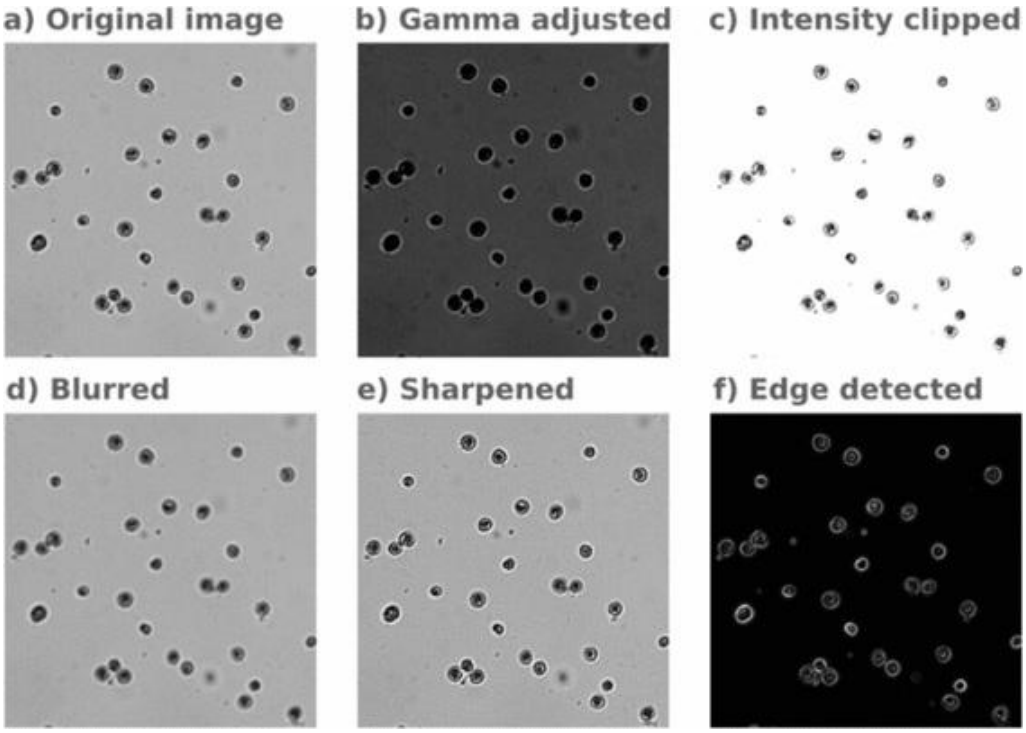

Figure 18.4. Example results from common image-processing operations. An original microscope image of mammalian cells is shown alongside five adjusted versions created with image-processing routines presented here. The gamma adjustment used a factor of 4.0. The blurred, sharpened and edge-detected images were generated by matrix convolution. The intensity clipping set the lightest pixels to mid-grey and then normalised the intensities to the full black-white range.

If we want to perform a different kind of adjustment we need to make a different processing object. The next example is for image sharpness (changing the local contrast at the edges within an image), where we can make things sharper with a value greater than one, or more blurred with a value less than one:

```
processObj = ImageEnhance.Sharpness(img)

img2Sharp = processObj.enhance(4.0)
img2Sharp.show()

imgBlur = processObj.enhance(0.5)
imgBlur.show()
```

Likewise, we can adjust brightness and overall colourfulness, noting that in the next examples we use the method .enhance() straight away, and don't make an explicit processing object; the object is still made after .Brightness(img), but it is not given a named variable.

```
imgBrighter = ImageEnhance.Brightness(img).enhance(0.5)
imgDull = ImageEnhance.Color(img).enhance(0.1)
```

Intensity adjustments using NumPy

Moving to numeric Python, we will show the same kinds of image adjustment for pixmap arrays and also show how things can be taken further. The examples will be constructed as Python functions and testing is demonstrated after the function definitions. As usual, the required NumPy imports are made upfront:

```
from numpy import array, dstack, exp, mgrid, sqrt, uint8
```

The mgrid object imported here may not be familiar. This is used to quickly create arrays that can be used together to form a grid of row and column numbers. For example, mgrid[0:3,0:3] gives the following sub-arrays: [[0,0,0],[1,1,1],[2,2,2]] and [[0,1,2],[0,1,2],[0,1,2]]. This is handy because the first sub-array gives the row number of the elements and the second gives the column number. This can be thought of as being analogous to using the regular range() function, for arrays.

The first example function controls the brightness of an object using what is known as *gamma correction* (see Figure 18.4b). This sort of correction is often used to adjust an image so that it can be presented by different kinds of display, accounting for different innate responses to brightness. The mathematical operation used is very simple: the pixmap values (albeit greyscale, RGB etc.) are converted into the range 0.0 to 1.0 and all the values are raised to the power of gamma. The effect is that if gamma is greater than one the image will look darker, and below one brighter. Taking the mid point, 0.5 as an example, gamma=2.0 changes this to 0.25 and gamma=0.5 gives 0.707. Adjusting the brightness in this way still preserves the extremes (i.e. $0^{\gamma} = 0$ and $1^{\gamma} = 1$) but the 'curve' of intermediate values is distorted.

The function takes a pixmap array and the gamma factor as input. Inside the function, the pixmap (which we are assuming takes values from 0 to 255) is scaled, so the maximum possible value is 1.0. The gamma power is applied, and the values are then re-scaled back to their original range. Because we made a new array in the function we pass this back at the return.

```
def gammaAdjust(pixmap, gamma=1.0):

  pixmap = array(pixmap, float)/255.0
  pixmap = pixmap ** gamma
  pixmap *= 255

  return pixmap
```

The next brightness-related function is designed to automatically adjust the values in the pixmap so that they are 'normalised' to take up the full range. So, for example, if we

had a dull grey image, with no black or white, the darkest shade would be moved to black (0) and the brightest to white (255). The function works by first subtracting the smallest (.min()) value in the pixmap from all the elements, so that the minimum is set to zero. Next the maximum value is set to be 255, by dividing by the adjusted pixmap's maximum (giving a 0.0 to 1.0 range initially) and then multiplying by 255.0. Note that we deliberately use the floating point number 255.0 so that the division also gives a floating point result and also that we guard against dividing by a maximum of zero (in an all-black image). The scaled pixmap is then passed back at the end.

```
def normalisePixmap(pixmap):

  pixmap -= pixmap.min()
  maxVal = pixmap.max()

  if maxVal > 0:
    pixmap *= 255.0 / maxVal

  return pixmap
```

The next example is a little more complicated. It sets the minimum and maximum brightness values in an image by clipping, i.e. it only adjusts the edges of the brightness range and doesn't affect the middle. For greyscale images the clipPixmapValues function can be used with normalisePixmap above to stretch the values to black and white again, thus removing any dark or light image detail, as illustrated in Figure 18.4c.

The function is defined as taking a pixmap and two threshold values. These thresholds have default values so that if they are not set the image does not change, i.e. 0 and 255 are the normal limits and all values will lie between. In the function the pixmap is first copied, so we don't affect the original. Then we define grey, which will be a greyscale pixmap (a map of the brightness) by taking an average over all the colour values, i.e. in the depth dimension of the pixmap, hence axis=2.[6] It should be noted that, because we take an average of colours, individual red, green and blue components may lie outside the thresholds, so in essence the clipping is according to how close a pixel is to black or white.

With the grey pixmap defined, we set any limiting values, first minimum then maximum. In both cases we define boolArray, which contains an array of True and False values depending on whether the test condition was met: if the intensity values of the elements were smaller or larger than the threshold. The arrays of truth values are converted to indices with .nonzero(), which pulls out the array coordinates (row and column) of the True values. These indices are the ones that are to be changed, and are simply used to set those values in the pixmap to the specified limit.

```
def clipPixmapValues(pixmap, minimum=0, maximum=255):

  pixmap2 = pixmap.copy()
  grey = pixmap2.mean(axis=2)

  boolArray = grey < minimum
  indices = boolArray.nonzero()
  pixmap2[indices] = minimum
```

```
  boolArray = grey > maximum
  indices = boolArray.nonzero()
  pixmap2[indices] = maximum

  return pixmap2
```

Note that an alternative way of clipping the values of a bitmap would be to use the .clip() function of NumPy arrays, e.g:

```
minimum, maximum = 64, 192
pixmap2 = pixmap.clip(minimum, maximum)
```

In contrast to clipPixmapValues() this will limit the values in the colour layers separately, rather than the combined, average signals. Naturally which function is more useful will depend on the context.

Ancillary to the above functions that adjust brightness values, it is commonplace to look at a histogram of the values to see what their distribution is. This is a good way of looking at the statistical effect of the operations, and also allows people to make intelligent choices when using thresholds, e.g. to separate signal from noise, or foreground from background. First a grey pixmap of brightness is made by averaging over the depth (colour) axis. This array is then flattened to a one-dimensional array and converted to a regular Python list. This list is passed to the pyplot.hist() function from matplotlib to make a histogram with 256 bits (or we could use a smaller number for less detail).

```
def showHistogram(pixmap):

  grey = pixmap.mean(axis=2)
  values = grey.flatten().tolist()

  from matplotlib import pyplot

  pyplot.hist(values, 256)
  pyplot.show()
```

The above functions can be tried with a test image, using the PIL Image object to take care of loading and display, as discussed previously.

```
from PIL import Image

img = Image.open('examples/Cells.jpg')
pixmap = imageToPixmapRGB(img)

showHistogram(pixmap)

pixmap2 = gammaAdjust(pixmap, 0.7)
pixmap3 = clipPixmapValues(pixmap2, 0, 145)
pixmap4 = normalisePixmap(pixmap3)

pixmapToImage(pixmap4, mode='L').show()
```

Convolving image filters

The next examples move on from pixel brightness to the concept of *filters*. In this context

a filter is a way of transforming an image, combining original pixel values together to make new values. A simple example of this is the blurring of an image (see Figure 18.4d). The blurred version of a pixel is constructed by setting its value to be an average of the surrounding pixels. The filters used will be described as matrices. For example, the 3×3 matrix [[1,1,1], [1,8,1], [1,1,1]], can be used to blur an image. The way to think of this is that the centre of the matrix (which has the value 8 here) represents the position of the original pixel, and the other elements are the square of pixels that surrounds it. The values in the filter matrix dictate how much influence each of the pixels has when used to create a new pixel. For the 3×3 example with 1 at the edges and 8 in the centre the new pixel will be an average (in terms of RGB or whatever) of the eight surrounding pixels and the central one, which here has as much influence as all the rest combined. As a consequence the new pixmap will be a blurred version of the original; the pixel values will spread to their neighbours slightly. When applying a filter matrix it is either normalised (elements sum to one) or the image is normalised afterwards so that the final pixel value cannot exceed the image maximum.

Many of the filtering and processing examples that we will illustrate have implementations in the scipy.ndimage module.[7] This module is well worth considering, especially in view of its speed and large range of functionality. For example, instead of the Gaussian blurring example that we give below, which uses NumPy alone, the scipy.ndimage.filters.gaussian_filter() function can be used instead. However, in this chapter we will mostly use NumPy, to better illustrate what is happening at a low level, and only use SciPy a little for some generic functionality.

The actual application of the filter is mathematically a *convolution*, which we can perform using the handy ndimage.convolve()function from the scipy module.[8] The convolution operation takes two arrays, which in this instance are the image pixmap and the filter matrix. One caveat to the convolve function is that both input arrays must have the same number of axes (dimensions), so when we are applying a flat matrix (2D) to an RGB or CMYK pixmap (3D) we convolve the 2D matrix separately with each of the colour layers. The check for this is simple given the .ndim attribute of the arrays: we insist that the matrix is 2D (triggering an exception if not) and that the pixmap is either 2D or 3D. If the pixmap is 2D we can perform the convolution directly. Otherwise, for a 3D pixmap the colour components are convolved separately (extracting each layer with slice notation pixmap[:,:,i]) and the transformed colour layers are then stacked in the usual way (depth means colour) and returned from the function as a complete pixmap array.

For the convolution the mode can be specified to determine how the limits of the arrays, where the filter would overlap the pixmap edge, are treated. By default this mode is ‘reflect’,[9] which means that the image is effectively extended by using a mirror image at the edge. This kind of edge treatment can introduce processing artefacts, but it at least keeps the size of the output array the same as the input.

```
from scipy import signal

def convolveMatrix2D(pixmap, matrix, mode='reflect'):

  matrix = array(matrix)
  if matrix.ndim != 2:
```

```
    raise Exception('Convolution matrix must be 2D')
  if pixmap.ndim not in (2,3):
    raise Exception('Pixmap must be 2D or 3D')

  if pixmap.ndim == 2:
    pixmap2 = ndimage.convolve(pixmap, matrix, mode=mode)

  else:
    layers = []
    for i in range(3):
      layer = ndimage.convolve(pixmap[:,:,i], matrix, mode=mode)
      layers.append(layer)

    pixmap2 = dstack(layers)

  return pixmap2
```

To process an image with a filter we simply call the above with pixmap and matrix arguments:

```
matrix = [[1, 1, 1],
          [1, 8, 1],
          [1, 1, 1]]

pixmapBlur = convolveMatrix2D(pixmap, matrix)
```

To view the result we need to normalise the image, given that the pixmap was convolved in a way that increased the intensity values by a factor of 16 (8 from the original intensity plus 1 from each of eight neighbouring pixels, according to matrix). Hence, we divide the pixmap so that the intensities of the pixels are put back in the original range.

```
pixmapBlur /= array(matrix).sum()
pixmapToImage(pixmapBlur).show()
```

Sharpen, blur and edge-detection filters

Although we can use any filtering matrix, there are several common operations that are applied to pixmaps, so we will encapsulate some of these in functions. The first example of these sharpens an image, as illustrated in Figure 18.4e. It uses a filter matrix that accentuates the differences between pixels.

```
def sharpenPixmap(pixmap):

  matrix = [[-1,-1,-1],
            [-1, 8, -1],
            [-1,-1,-1]]
```

The procedure is to convert the input pixmap into a grey (brightness) pixmap. The grey pixmap is then convolved with the filter matrix, which increases the contrast at the edges of features (where there are changes in brightness), and normalised to use the full range: 0 to 255.

```
grey = pixmap.mean(axis=2)

pixmapEdge = convolveMatrix2D(grey, matrix)
normalisePixmap(pixmapEdge)
```

The grey pixmap with the enhanced edges has its values centred on the average brightness. So, for example, if the average brightness of pixmapEdge is 127, the range of values changes from 0…255 to −127…128. These centred values, either side of zero, represent how much adjustment we will apply to sharpen the original image. Before making the adjustment pixmapEdge is stacked so that it is three layers deep, and thus will operate on red, green and blue.

```
pixmapEdge -= pixmapEdge.mean()
pixmapEdge = dstack([pixmapEdge, pixmapEdge, pixmapEdge])
```

The new, sharpened image is created by adding the pixmap edge adjustment to the original pixmap. With the pixels adjusted the clip function (inbuilt into NumPy arrays) is used to make sure that adding the pixmaps does not exceed the limits of 0 and 255.

```
pixmapSharp = pixmap + pixmapEdge
pixmapSharp = pixmapSharp.clip(0, 255)

return pixmapSharp
```

The next example is the Gaussian filter, which blurs pixels with a weighting that has a normal (‘bell curve’) distribution (see Figure 22.4 for an illustration). For this, two values are passed in: r is the half-width of the filter excluding the centre and sigma is the amount of spread in the distribution. These parameters respectively control the size and strength of the blur. Larger filters with wider distributions (i.e. influence away from the centre) will give more blurring. It is notable that the mgrid object is used to give a range of initial grid values for the filter, specifying the separation of each point from the centre in terms of rows and columns; this is similar to using range() to generate a list.

```
def gaussFilter(pixmap, r=2, sigma=1.4):

  x, y = mgrid[-r:r+1, -r:r+1]
```

The Gaussian function is applied by taking the row and column values (x and y), squaring them, scaling by two times sigma squared and finally taking the negative exponent of the sum. The exact centre row and column will be zero and so the exponent will be at a maximum here, but the further x and y row and column values are from the centre the smaller the value is.

```
s2 = 2.0 * sigma * sigma

x2 = x * x / s2
y2 = y * y / s2

matrix = exp( -(x2 + y2))
matrix /= matrix.sum()
```

Once the filter matrix is defined it is applied to the pixmap using convolution, to each

of the colour components.

```
pixmap2 = convolveMatrix2D(pixmap, matrix)

return pixmap2
```

The final filter example is for edge detection and uses what is known as the *Sobel operator*. In essence this is a filter that detects the intensity gradient between nearby pixels (see Figure 18.4f). It is applied horizontally, vertically or in both directions and gives bright pixels at those edges. As can be seen in the Python code the filter is a 3×3 matrix where there is a line of negative numbers, then zeros, then positive numbers. This matrix is transposed to switch between horizontal and vertical operations. The matrix means that, for a given orientation, a transformed pixel has none of its original value, but rather a value which represents the difference between values on either side.

```
def sobelFilter(pixmap):

  matrix = array([[-1, 0, 1],
                  [-2, 0, 2],
                  [-1, 0, 1]])
```

The Sobel filter matrix is applied to the grey average of the input pixmap. This is done twice for both orientations so we get two edge maps.

```
grey = pixmap.mean(axis=2)
edgeX = convolveMatrix2D(grey, matrix)
edgeY = convolveMatrix2D(grey, matrix.T)
```

The final pixmap of edges is then a combination of horizontal and vertical edge maps. Taking the square root of the sum of the squares of the two edge maps means the values will always be positive; it won’t make a difference between an edge going from light to dark or dark to light in an image. The edge-detected pixmap is also normalised so we can see the full range of values and finally it is returned from the function.

```
pixmap2 = sqrt(edgeX * edgeX + edgeY * edgeY)
normalisePixmap(pixmap2) # Put min, max at 0, 255

return pixmap2
```

The filter functions can all be tested with the example image, using Image.show() to see the results, after the appropriate array conversions. Note that for the sobelFilter() output we pass the ‘L’ mode to the PIL conversion function because it is a greyscale image, not RGB.

```
from PIL import Image
img = Image.open('examples/Cells.jpg')
pixmap = imageToPixmapRGB(img)

pixmap = sharpenPixmap(pixmap)
pixmapToImage(pixmap).show()

pixmap = gaussFilter(pixmap)
```

```
pixmapGrey = sobelFilter(pixmap)
pixmapToImage(pixmapGrey, mode='L').show()
```

In the above examples we have demonstrated using Python functions, rather than classes (our own kind of Python object), to keep things simple. However, custom object classes can be really convenient when you know what you're doing. So, for the image examples the programmer may consider making a bespoke Pixmap class (or whatever name seems best). This could have the ability to work with PIL automatically, doing the right array conversions and perform common operations, i.e. using Pixmap.sobelFilter() methods etc.

Feature detection

Lastly in this chapter we show a practical example that aims to automatically extract information about the physical objects that are represented by an image. This touches lightly on the field of *image recognition*, although what we show is simple compared to state-of-the art techniques. Nonetheless it aims to give a basic idea of what kind of thing is possible. The objective will be to count the number of cells in a digital photograph taken using a microscope, which is a fairly routine data-gathering task in biology.

Counting cells

Initially the image is loaded and converted to an array representing the pixmap. Then we apply the Gaussian filter (with default parameters) to blur the image slightly, assigning the result to pixmap2 to keep the original pixmap. The blurring acts to remove the small-scale components of the image; this reduces image noise but does not significantly affect the images of the cells. Then we apply the Sobel edge-detection filter to the image, and normalise. This makes a greyscale image of just the outlines, which we can inspect. An alternative at this point would be to convert the image into black and white (only) using an intensity threshold; however, edge detection will work better where the background colour of the photograph is uneven.

```
from PIL import Image

img = Image.open('examples/Cells.jpg')

pixmap =  imageToPixmapRGB(img)
pixmap2 = gaussFilter(pixmap)
pixmap2 = sobelFilter(pixmap2)

normalisePixmap(pixmap2)

pixmapToImage(pixmap2).show()
```

Next the pixmapCluster function is constructed, which will analyse our pre-processed image by clustering the bright pixels (the edges) so that we can identify blobs that represent cells. The blobs can then be analysed to select those of the required size and shape etc.

Before the clustering a helper function is defined which will find the neighbours of a

pixel, investigating those above, below, to the left and to the right, and checking whether each is present in a pre-specified set of points. This points set will represent all the bright pixels that come from the edge-detection step. The check list specifies the neighbouring locations relative to the input point to test, and neighbours is the list from among these that are acceptable because they are in points. Note that some of the checked positions will be off the edge of the pixmap, but this does not matter because they would never be found in points in the first place.

```
def getNeighbours(point, points):

  i, j = point
  check = [(i-1, j), (i+1, j),
           (i, j-1), (i, j+1)]

  neighbours = [p for p in check if p in points]

  return neighbours
```

Next comes the main pixel clustering function. The details of this will not be described here because it is very similar to the simpleCluster() function described fully in Chapter 23. Essentially a threshold value is used to get a list of bright pixel points (using the same indexing strategy as clipPixmap()). These are then grouped into clusters according to whether they are deemed to be neighbours, as determined in getNeighbours. Comparing to simpleCluster() the key differences are that we are working directly with the pixel objects, rather than via indices, and that the neighbour-detecting algorithm does not need to search all pairs of data points; here only a local area of the pixmap needs to be inspected, which is very much quicker. At the end of the clustering, clusters, a list of lists, is passed back, where each sub-list represents all the pixel points (x, y locations) in each detected blob.

```
def brightPixelCluster(pixmap, threshold=60):

  boolArray = pixmap > threshold
  indices = array(boolArray.nonzero()).T
  points = set([tuple(point) for point in indices])

  clusters = []
  pool = set(points)
  clustered = set()

  while pool:
    pointA = pool.pop()
    neighbours = getNeighbours(pointA, points)

    cluster = []
    cluster.append(pointA)
    clustered.add(pointA)

    pool2 = set(neighbours)
    while pool2:
      pointB = pool2.pop()

      if pointB in pool:
        pool.remove(pointB)
```

```
        neighbours2 = getNeighbours(pointB, points)
        pool2.update(neighbours2)
        cluster.append(pointB)

    clusters.append(cluster)

  return clusters
```

The bright pixel clustering may then be used on a pre-processed pixmap with highlighted edges. Each cluster of pixels will represent a blob, which we can group into large, medium and small varieties. Here the blob size thresholds were determined by looking at a histogram of the number of points in the clusters:

```
clusters = brightPixelCluster(pixmap2)
sizes = [len(c) for c in clusters]
from matplotlib import pyplot
pyplot.hist(sizes, 40)
pyplot.show()
```

Then the clusters were grouped by size and placed in separate lists for reporting:

```
smallBlobs = []
mediumBlobs = []
bigBlobs = []

for cluster in clusters:
  n = len(cluster)

  if n < 80:
    smallBlobs.append(cluster)
  elif n < 320:
    mediumBlobs.append(cluster)
  else:
    bigBlobs.append(cluster)

print('Found %d small blobs' % len(smallBlobs))
print('Found %d medium blobs' % len(mediumBlobs))
print('Found %d big blobs' % len(bigBlobs))
```

To visualise the clustering results we will add colour codes to a grey version of the original image. Note that the grey pixmap has had its edges removed by slicing ([2:-2, 2:-2]) because the pre-processed pixmap2 lost two edge points when it went though the convolution filters. We could improve the filtering process to deal with edges better if required, for example, by extending a pixmap with copied data so that it retains its original size after filtering.

The grey pixmap is stacked three layers deep so we can make an RGB image with the colour codes:

```
grey = pixmap.mean(axis=2)[2:-2, 2:-2]
colorMap = dstack([grey, grey, grey])
```

A list of colours containing (red, green, blue) arrays and the corresponding blob data is constructed:

```
colors = [(255, 0, 0), (255, 255, 0), (0, 0, 255)]
categories = [smallBlobs, mediumBlobs, bigBlobs]
```

Then by going through the clusters in each category we can colour the initially grey pixmap. Each cluster contains a list of row and column locations within the pixmap, and by extracting these into two separate lists (x, y) we have a means of selecting a subset of the colorMap and setting the colour of blob points to reflect the category. Once it is coloured, we can admire our handiwork with Image.show().

```
for i, blobs in enumerate(categories):
  color = colors[i]

  for cluster in blobs:
    x, y = zip(*cluster)

    colorMap[x,y] = color

Image.fromarray(array(colorMap, uint8), 'RGB').show()
```

The analysis performed on the blobs has only considered their total pixel area, but more sophisticated properties can be used. For example, the shape of the blobs, such as how circular they are, could be measured. Looking at the example blob-detection results (see Figure 18.5) there is an obvious extension to our cell-counting routine, which is to attempt to subdivide the larger blobs to estimate how many cells are overlapped:

```
numCells = len(mediumBlobs)  # initial guess
cellAreas = [len(blob) for blob in mediumBlobs]
meanCellArea = sum(cellAreas) / float(numCells)

for blob in bigBlobs:
  numCells += int( len(blob) // meanCellArea )

print('Estimated number of cells: %d' %  numCells)
```

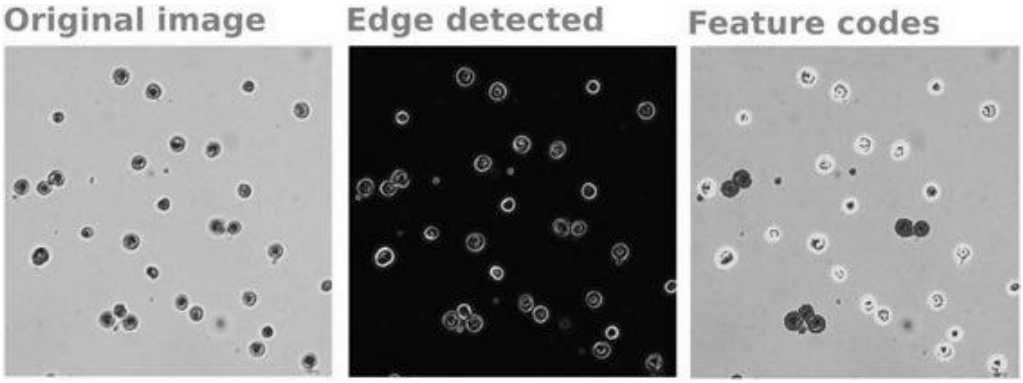

Figure 18.5 (Plate 8). Images from micrograph cell-counting procedure. An original microscope image of mammalian cells is shown alongside the results of the edge detection and a greyscale version where the different cell-edge features are labelled with colours after blob analysis. Isolated cells are yellow, overlapping cells are blue and small fragments are red.

[1] As of 2013, PIL has not been ported to Python 3, but an alternative implementation, Pillow, has been: http://python-imaging.github.io.

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

[3] Download PIL via links at http://www.cambridge.org/pythonforbiology.

[4] These Image objects sometimes cause problems if they are associated with a local variable, e.g. inside a function, and get *garbage collected* sooner than expected. If this occurs then the general solution is to make sure that the image object has a reference from another object that will persist in memory, for example by putting it in a non-local list or adding it as a self. attribute.

[5] No positive or negative sign.

[6] The array axis indices are 0 for the row/height, 1 for column/width and 2 for colour values.

[7] SciPy is typically installed with NumPy.

[8] SciPy also has a signal.convolve() function, which could be used, though this doesn't deal with the edges of images so well.

[9] The other options include 'constant', which uses a user-defined (cval=) constant value outside the image edge; 'nearest', which extends the edge values outwards; and 'wrap', which takes values from the opposite edge.