5 Two-dimensional graphics

5.1 Introduction

The most venerable and perhaps best-known scientific graphics package is *Gnuplot*, and downloads of this open-source project can be obtained from its website.[1] The official documentation is Gnuplot Community (2012), some 238 pages, and a more descriptive introduction can be found in Janert (2010). *Gnuplot* is of course independent of Python. However, there is a *numpy* interface to it, which provides Python-like access to the most commonly used *Gnuplot* functions. This is available on line.[2] Although most scientific Python implementations install the relevant code as a matter of course, the documentation and example files from this online source are useful. For many applications requiring two-dimensional graphics, the output from *Gnuplot* is satisfactory, but only at its best is it of publication quality. Here *Matlab* is, until recently, the market leader in this respect, but Python aims to equal or surpass it in quality and versatility.

The *matplotlib* project[3] aims to produce Matlab-quality graphics as an add-on to *numpy*. Almost certainly, this should be part of your installation. It is installed by default in most Python packages designed for scientists. There is extensive "official documentation" (1255 pages) at Matplotlib Community (2013), and a useful alternative description in Tosi (2009). The reader is strongly urged to peruse the Matplotlib Gallery[4] where a large collection of publication quality figures, and the code to generate them, is displayed. This is an excellent way to explore the visual capabilities of *matplotlib*. Since *matplotlib* contains hundreds of functions, we can include here only a small subset. Note that almost all of the figures in this and subsequent chapters were generated using *matplotlib* and the relevant code snippets are included here. The exigencies of book publishing have required the conversion of these colour figures to black, white and many shades of grey.

As with all other powerful versatile tools, a potential user is encouraged strongly to read the instruction manual, but at well over one thousand pages few scientific users will attempt to do so. Indeed, the philosophy of this book is "observe and explore". We have therefore restricted the coverage to what seems to be the most useful for a scientist. Sections 5.2–5.8 look at a variety of single plots, and Section 5.7 shows how

[1] See `http://www.gnuplot.info`.
[2] See `http://gnuplot-py/sourceforge.net`.
[3] See `http://matplotlib.org`.
[4] See `http://matplotlib.org/gallery.html`.

to display mathematical formulae in a figure. Next Section 5.9 looks at the construction of compound figures. The following Section 5.10 looks at the production of animations for use within Python and the production of movies for use in presentations. The final Section 5.11 shows how to construct an intricate figure pixel by pixel. This is a long chapter. However, spending a few minutes reading the next section carefully can save prospective but impatient users hours of frustration.

5.2 Getting started: simple figures

Converting theoretical wishes into actual figures involves two distinct processes known as “front-ends” and “back-ends”.

5.2.1 Front-ends

The *front-end* is the user interface. As a prospective user, you will need to decide whether you will always want to prescribe figures from within the confines of an interpreter, or whether you want more generality, perhaps developing figures within an interpreter, but later invoking them by batch-processing a Python file, or even nesting them within some other application. The first, simpler but more restrictive choice is catered for by the *pylab* branch of *matplotlib*. You can gain access to it from the command line by, e.g.,

```
ipython --pylab
```

which loads *numpy* and *matplotlib* implicitly and without the safeguards advocated in the namespace/module discussion of Section 3.4. The motivation is to attract Matlab users by providing nearly identical syntax. But scientist Matlab users who are reading this chapter probably wish to escape its restrictions, and so we will eschew this approach. Instead, we shall follow the *pyplot* branch, the main topic of this chapter. As we shall see, this respects the namespace/module conventions and so offers a path to greater generality.

Working in the *IPython* interpreter, the recommended way to access *pyplot* is as follows

```
import matplotlib.pyplot as plt
```

Next, try some introspection, `plt?`. The query `plt.`TAB will reveal over 200 possibilities, so this is a large module. Fortunately, a wide variety of pictures can be drawn using surprisingly few commands.

5.2.2 Back-ends

The *front-end* supplies Python with user-requests. But how should Python turn them into visible results? Should they be to a screen (which screen?), to paper (which printer?), to another application? These questions are obviously hardware-dependent and it is the job of the software *back-end* to answer them. The *matplotlib* package contains a wide range of *back-ends*, but which one should be used? Fortunately, your Python installer should have recognized your hardware configuration and should have chosen the optimal *back-end*[5].

Within the code snippets described below, the command `plt.get_backend()` will reveal which *back-end* is being used. If you want to change this, you need first to locate the preference file. The commands

```
import matplotlib
matplotlib.matplotlib_fname()
```

will reveal this. Next you open this text file in your editor and navigate to about line 30. Above this you will see a list of available *back-ends* and you can edit the file by commenting out all but one choice, so as to specify the one actually used.

5.2.3 A simple figure

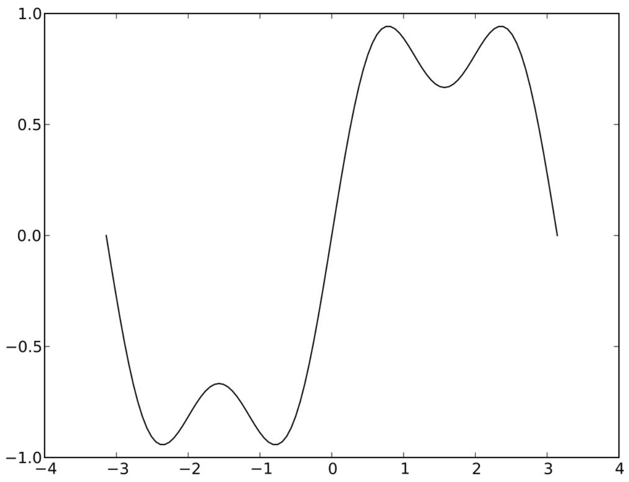

Figure 5.1 A simple plot using *matplotlib*.

Throughout this chapter, we shall be using the *IPython* interpreter. Minor differences will be encountered if an alternative interpreter is used. You should consider typing the following snippet into a new file, and then executing it via the magic `%run` command.

[5] Sometimes this does not happen; e.g., the installation of the *Enthought Python Distribution* 7.3-1 on a Macintosh produces two results. The 64-bit version uses the backend *MacOSX*, which works within *IPython*, but the 32-bit version uses *WXAgg* by default, which does not, at least on my machine.

This should display a result rather like Figure 5.1. If it does not, then you need to experiment with other *back-ends* as described above.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x=np.linspace(-np.pi,np.pi,101)
5  y=np.sin(x)+np.sin(3*x)/3.0
6
7  plt.ion()
8  plt.plot(x, y)
9  plt.savefig('foo.pdf')
10 #plt.show()
```

Lines 1, 4 and 5 should be familiar. As we suggested in Section 5.2.1, line 2 shows the recommended way to import *matplotlib* for general use. The single line 8 should construct a figure, and line 9 saves it as a file in the current directory. (The types of file which can be written depend on the implementation and *back-end*. Most support `png`, `pdf`, `ps`, `eps` and `svg` files, and your implementation should support other formats.) We then have two choices as to how to display it. For developing a figure, and for use in the interpreter, line 7 toggles “interactive mode” on (and the less frequently used `plt.ioff` switches it off). Then, as we add further lines, their effect is displayed instantly. This can absorb resources, and the default is to make changes but not display them. Finally, when the figure is ready to be displayed, and you are not using the interpreter, the commented out line 10 displays the completed figure. Thus you need to invoke either `plt.ion` or `plt.show` to send any output to the screen. Within the interpreter, both give identical outputs, but `plt.show` has a blocking property, and further progress is halted until the picture is deleted. Neither the docstrings, `plt.ion?` etc., or the manual, Matplotlib Community (2013), are particularly helpful on this point. You need to find by experiment which command best suits your work style.

The figure itself is plain and unadorned, although the default axes ranges look reasonable. In the next section, we shall see how to enhance it. It is remarkable though that we need only three *matplotlib* functions to draw, display and save a figure!

5.2.4 Interactive controls

The seven interactive buttons at the bottom of the *matplotlib* window allow interactive manipulations of the current figure to produce a sequence of new ones.

We start with button 4, the pan/zoom tool. It should be clicked first to enable it and then the mouse pointer should be moved into the figure. There are two operations:

- Pan: Click the left mouse button and hold it down while dragging the mouse to a new position, and then release it. By simultaneously holding down the *x* or *y* key, the panning action is limited to the selected direction.
- Zoom: Click the right mouse button at a chosen point and hold it to zoom the figure.

Horizontal motion to the right or left generates a proportional zoom in or out of the x-axis, keeping the chosen point fixed. Vertical motion does the same for the y-direction. The x and y keys work in the same way as for panning. Simultaneously holding down the *Ctrl* key preserves the aspect ratio.

Button 5 is in many ways more convenient. Click to enable it and then, holding down the left mouse button, drag a rectangle on the figure. The view will be zoomed to the interior of the rectangle.

Rather like a browser, button 1 returns to the original figure and buttons 2 and 3 allow browsing down and up the stack of modified figures. Button 6 allows control of the margins of the figure. In principle, button 7 allows you to save to file the current figure. This option is implementation-dependent, and will not function in all installations.

5.3 Cartesian plots

5.3.1 The *matplotlib* `plot` function

This is a very powerful and versatile function with a long and complicated docstring to match. However the principles are straightforward. The simplest call, see, e.g., the snippet above, would be `plt.plot(x,y)`, where `x` and `y` are *numpy* vectors of the same length. This would generate a curve linking the points $(x[0], y[0]), (x[1], y[1]), \ldots,$ using the default style options. The most concise call would be `plt.plot(y)`, where `y` is a *numpy* vector of length `n`. Then a default `x` vector is created with integer spacing to enable the curve to be drawn. The syntax allows many curves to be drawn with one call. Suppose `x`, `y` and `z` are vectors of the same length. Then `plt.plot(x,y,x,z)` would produce two curves. However, I recommend that you usually draw only one curve per call, because it is then much easier to enhance individual curves with a call of the type `plt.plot(x,y,fmt)`, where the format parameter(s) `fmt` is about to be described.

5.3.2 Curve styles

Table 5.1 The standard *matplotlib* colour choices. If the colours of a set of curves are not specified, then *matplotlib* cycles through the first five colours.

character	colour
b	blue (default)
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

By the style of a curve, we mean its colour, nature and thickness. *Matplotlib* allows a variety of descriptions for colour. The most-used ones are given in Table 5.1. Thus if `x` and `y` are vectors of the same length, then we could draw a magenta curve with `plt.plot(x,y,color='magenta')`, or more concisely via `plt.plot(x,y,'m')`. If several curves are drawn and no colours are specified, then the colours cycle through the first five possibilities listed above.

Table 5.2 The standard *matplotlib* line styles.

character(s)	description
-	solid curve (default)
--	dashed curve
-.	dash-dot curve
:	dotted curve

Curves are usually solid lines, the default, but other forms are possible, see Table 5.2. The clearest way to use these is to concatenate them with the colour parameter, e.g., `'m-.'` as a parameter produces a magenta dash–dotted curve. The case where we want no curve at all is dealt with below.

The final line style is width, measured by **`float`** values in printer's points. We could use, e.g., `linewidth=2` or more concisely `lw=2`. Thus to draw a magenta dash–dotted curve of width four points, we would use

```
plt.plot(x,y,'m-.',lw=4)
```

5.3.3 Marker styles

Table 5.3 The 12 most commonly used marker styles.

character	colour
.	point (default)
o	circle
*	star
+	plus
x	x
v	triangle down
^	triangle up
<	triangle left
>	triangle right
n	square
p	pentagon
h	hexagon

Matplotlib offers a menagerie of marker styles, 22 at the last count, and the 12 most commonly used ones are given in Table 5.3. For a complete list, see the `plt.plot` docstring. Again the easiest way to use is them is by concatenation, e.g., `'m-.x'` as a parameter produces a magenta dash–dotted curve with “x” markers. Notice that the parameter `'mo'` produces magenta circle markers at each of the points, but no curve joining them. Thus invoking `plt.plot(x,y,'b--')` followed by `plt.plot(x,y,'ro')` will draw a blue dashed curve with red circle markers. We can also control the colour and size with parameters that are more verbose, e.g.

```
plt.plot(x,y,'o',markerfacecolor='blue',markersize=2.5)
```

This form is usually used if we make two-tone markers with a different colour edge, e.g., by adding the parameters `markeredgecolor='red',markeredgewidth=2`. There are many many possibilities, not all of them aesthetically pleasant.

5.3.4 Axes, grid, labels and title

In the section above, we discussed regular Cartesian axes. Quite often though, one or more logarithmic axes are desired. The three cases are given by `plt.semilogx`, `plt.semilogy` and `plt.loglog`, which can be regarded as substitutes for `plt.plot`.

Matplotlib usually makes an excellent choice for the choice of axis extents and the ticks along them. However, the extents are easily changed with

```
plt.axis([xmin,xmax,ymin,ymax])
```

The functions `plt.xticks` and `plt.yticks` control the ticks along the axes, and their docstrings should be consulted if you wish to change the default settings.

By default, *matplotlib* does not include a grid. It can be added by using `plt.grid()`. This function has many optional parameters and if you wish to fine tune the grid, perusal of the docstring is recommended.

Suppose we are producing a figure with two or more curves. We may attach a label to any curve, by including the parameter `label='string'` in the call to `plt.plot`. After all of the curves have been plotted, the function `plt.legend(loc='best')` draws a box in the most suitable position. Inside the box, there will be a line for each labelled curve showing its style and label. As usual, the function can take a variety of parameters. Consult the docstring for details.

The title of a plot and its axes are most conveniently labelled with, e.g.

```
plt.title('Position as a function of time')
plt.xlabel('time')
plt.ylabel('position')
```

Note that for simple single plots, `plt.suptitle` is usually a preferable alternative.[6]

[6] If you are drawing several graphs in the same figure (see Section 5.9), then `title()` can be used to label each of them, while `suptitle()` generates an overall figure title.

5.3.5 A not-so-simple example: partial sums of Fourier series

We now give a not-so-simple explicit example using the concepts introduced above. We define $f(x)$ on $(-\pi, \pi]$ via

$$f(x) = \begin{cases} -1, & \text{if } -\pi < x < 0, \\ 1, & \text{if } 0 \leqslant x \leqslant \pi, \end{cases}$$

and by 2π-periodicity outside that interval. Then its Fourier series is

$$\mathcal{F}(x) = \frac{4}{\pi} \sum_{n \text{ odd}}^{\infty} \frac{\sin nx}{n}.$$

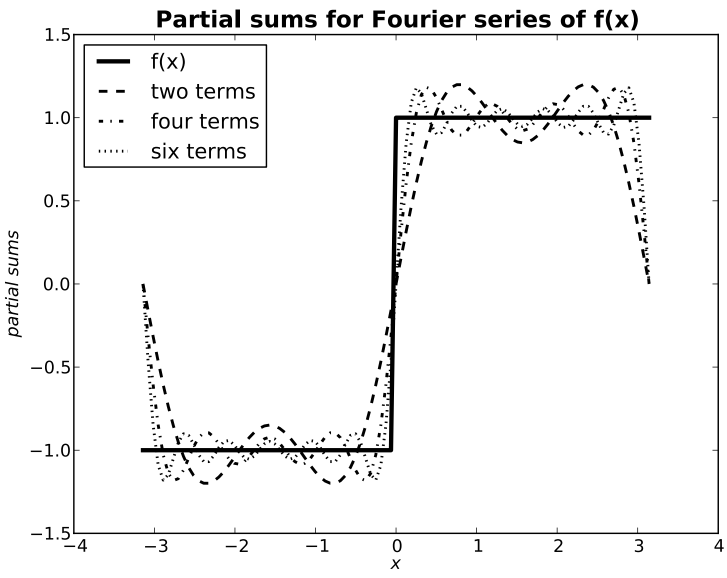

Figure 5.2 An enhanced not-so-simple plot using *matplotlib*.

A coloured version of Figure 5.2 was produced by the following snippet

```
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(-np.pi,np.pi,101)
f=np.ones_like(x)
f[x<0]=-1
y1=(4/np.pi)*(np.sin(x)+np.sin(3*x)/3.0)
y2=y1+(4/np.pi)*(np.sin(5*x)/5.0+np.sin(7*x)/7.0)
y3=y2+(4/np.pi)*(np.sin(9*x)/9.0+np.sin(11*x)/11.0)
plt.ion()
plt.plot(x,f,'b-',lw=3,label='f(x)')
plt.plot(x,y1,'c--',lw=2,label='two terms')
plt.plot(x,y2,'r-.',lw=2,label='four terms')
plt.plot(x, y3,'b:',lw=2,label='six terms')
plt.legend(loc='best')
plt.xlabel('x',style='italic')
plt.ylabel('partial sums',style='italic')
plt.suptitle('Partial sums for Fourier series of f(x)',
             size=16,weight='bold')
```

Several points need to be made about the decorations. Although it is more appropriate for a presentation than a book figure, we have included a title in a larger and bolder font. The axes labels have been set in italic font. More sophisticated textual decorations are possible in *matplotlib*, and they will be discussed in Section 5.7.

If this figure appears somewhat intricate, see another approach to the visual representation of complicated data in Section 5.9.

5.4 Polar plots

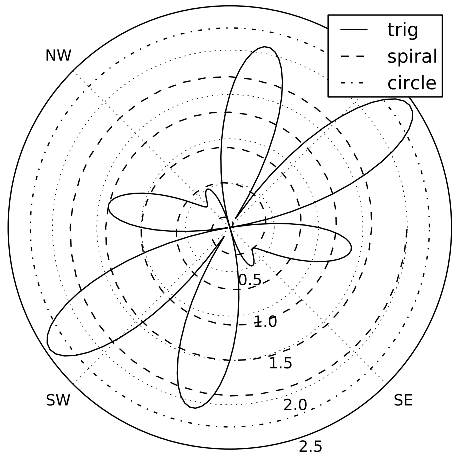

Figure 5.3 A simple polar plot using *matplotlib*.

Suppose we use polar coordinates (r, θ) and define a curve by $r = f(\theta)$. It is straightforward to plot this using `plt.polar`, which behaves rather like `plt.plot`. However, there are some significant differences, so perusal of the docstring before use is recommended. Figure 5.3 was created using the following snippet.

```
theta=np.linspace(0,2*np.pi,201)
r1=np.abs(np.cos(5.0*theta) - 1.5*np.sin(3.0*theta))
r2=theta/np.pi
r3=2.25*np.ones_like(theta)
plt.ion()
plt.polar(theta, r1,label='trig')
plt.polar(5*theta, r2,label='spiral')
plt.polar(theta, r3,label='circle')
plt.thetagrids(np.arange(45,360,90), ('NE','NW','SW','SE'))
plt.rgrids((0.5,1.0,1.5,2.0,2.5),angle=290)
plt.legend(loc='best')
```

The interactive pan/zoom button behaves differently when applied to polar plots. The radial coordinate labels can be rotated to a new position using the left mouse button, while the right button zooms the radial scale.

5.5 Error bars

When measurements are involved we often need to display error bars. *Matplotlib* handles these efficiently using the function `plt.errorbar`. This behaves like `plt.plot`, but with extra parameters. Consider first errors in the y-variable, which we specify with the `yerr` variable. Suppose the length of `y` is `n`. If `yerr` has the same dimension, then symmetric error bars are drawn. Thus for the kth point at `y[k]` the error bar extends from `y[k]-yerr[k]` to `y[k]+yerr[k]`. If the errors are not symmetric, then `yerr` should be a $2\times n$ array, and the kth error bar extends from `y[k]-yerr[0,k]` to `y[k]+yerr[1,k]`. Analogous remarks apply to x-errors and the variable `xerr`. By default the colour and line width are derived from the main curve. Otherwise, they can be set with the parameters `ecolor` and `elinewidth`. For other parameters, see the `plt.errorbar` docstring.

The thoroughly artificial errors shown in Figure 5.4 were produced using the following code snippet.

```
import numpy.random as npr
x=np.linspace(0,4,21)
y=np.exp(-x)
xe=0.08*npr.randn(len(x))
ye=0.1*npr.randn(len(y))
plt.ion()
plt.errorbar(x,y,fmt='bo',lw=2,xerr=xe,yerr=ye,
             ecolor='r',elinewidth=1)
```

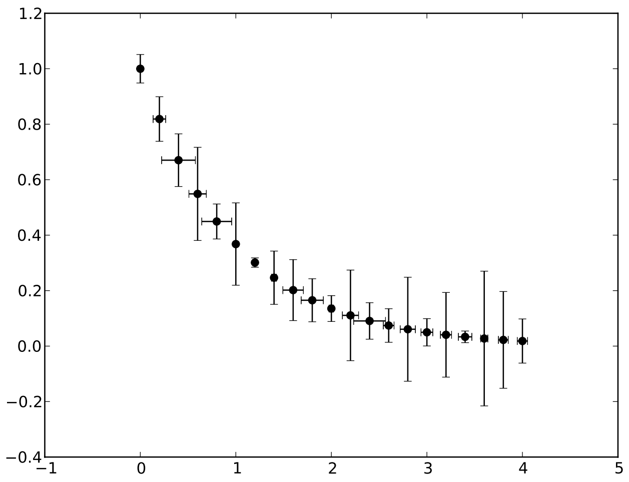

Figure 5.4 A very plain plot using error bars.

5.6 Text and annotations

Suppose we wish to place a plain text *string* in a figure starting at (x, y) in *user* coordinates, i.e., those defined by the figure axes. The extremely versatile *matplotlib* function `plt.text(x,y,'some plain text')` does precisely that. There are various ways of enhancing it, colours, boxes etc., which are described in the function's docstring.

Sometimes we wish to refer to a particular feature on the figure, and this is the purpose of `plt.annotate`, which has a slightly idiosyncratic syntax, see the snippet and the function's docstring.

```
x=np.linspace(0,2,101)
y=(x-1)**3+1
plt.plot(x,y)
plt.annotate('point of inflection at x=1',xy=(1,1),
             xytext=(0.8,0.5),
             arrowprops=dict(facecolor='black',width=1,
             shrink=0.05))
```

This produces Figure 5.5. For examples of annotations which are more sophisticated, see the *matplotlib gallery*.

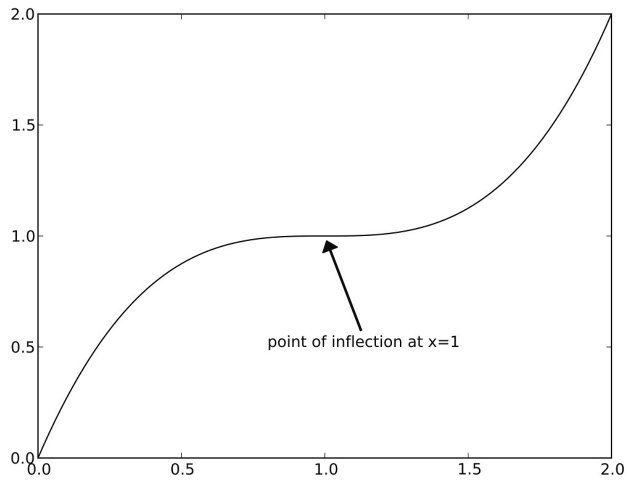

Figure 5.5 A simple example of annotation using *matplotlib*.

5.7 Displaying mathematical formulae

It is an opportune moment to discuss the Achilles heel of all plotting software. How do we display decently formatted mathematical formulae? Most word processors offer add-on tools to display mathematical formulae but quite often, and especially for intricate formulae, they look ugly. Mathematicians have learned how to produce book quality displays using the open-source LaTeX package.[7] Indeed, this book was produced using it.

First a few words about LaTeX. The quasi-official and most comprehensive package is *TexLive*.[8] This is available with extensive documentation and ready-to-run binaries for almost any platform and includes all commonly used TeX-programmes, macro packages and fonts. Many other less comprehensive collections are available.

However, *matplotlib* has made a very brave effort to make TeX-like features available for both non-LaTeX and LaTeX users. For small-scale one-off use, this can be highly effective.

5.7.1 Non-LaTeX users

Matplotlib includes a primitive engine called *mathtext* to provide TeX style expressions. This is far from a complete LaTeX installation, but it is self-contained, and the *matplotlib* documentation has a succinct review of the main TeX commands. The engine can be

[7] The original package was called TeX but development of this halted at version 3.1416. LaTeX includes TeX as a subset, and is where development is still to be found.
[8] Its homepage is at `http://www.tug.org/texlive`.

accessed from `plt.text` (or any of its derivatives, e.g., `plt.title`, `plt.suptitle`, `plt.xlabel`, `plt.ylabel` or `plt.annotate`) which expect to receive a *string*. The documentation examples work superbly! However, as a concrete real-life example, the figures in the next Section 5.8 need a title "The level contours of $z = x^2 - y^2$". How are we to achieve this?

The simplest solution would be

```
plt.title(r'The level contours of $z=x^2-y^2$')
```

To understand this line, consider first the *string* itself. You need to know that mathematics is defined in TEX between a pair of dollar ($) signs. Then TEX strips the dollar signs and sets the mathematics (by default) in Computer Modern Roman (CMR) italic font with the font size determined by the (TEX) context. The `r` before the first *string* delimiter indicates that this is a **raw** *string*, so that *matplotlib* will call up its *mathtext* engine. (Without it, the dollar signs would be rendered verbatim.) The code line above is used in the first snippet in Section 5.8, producing Figure 5.6. The visual aspect is grotesque! The four words are rendered in a sans-serif font of the *matplotlib* default size. The mathematics is rendered immaculately, but in CMR italic font at the TEX default size, which is much smaller! Apart from the font mismatch, there would appear to be no easy way to resolve the size discrepancy.

Fortunately, TEX can come to the rescue. One obvious solution to the font/size disparity is to render the entire *string* in TEX mathematics mode. We want the first four words though in the default TEX font, and we can achieve this by enclosing them in braces preceded by `\rm`. However, you need to know that TEX mathematics mode gobbles spaces, and so you need to insert `\␣` or `\,` or `\;` to obtain small, medium or larger spacing. Our revised command is

```
plt.title(r'$\rm{The\ level\ contours\ of\;} z=x^2-y^2$',
          fontsize=20)
```

The code line above is used in the second snippet in Section 5.8, leading to Figure 5.7. The font/size mismatch is removed, and the result scales consistently. (In fact, this book has been typeset via LATEX using the Times family of fonts as the default, and TEX has been told to respect this. Thus the formulae in the titles to Figures 5.6 and 5.7 are displayed in different fonts.) Do not forget that the same treatment can be applied to all of the other text instances that might occur in a figure.

For another realistic example, see Section 7.6.4.

5.7.2 LATEX users

Of course, *matplotlib* "knows" about LATEX and if you already have a working LATEX installation we now indicate how this can be used to simplify the production of the title *string* discussed above. The first step is to import the `rc` module, which manages *matplotlib* parameters. Next we need to tell *matplotlib* to use LATEX and what font we want to employ within it. This is done by inserting the following code into the code snippet immediately after the first two **`import`** lines.

```
from matplotlib import rc

rc('font',family='serif')
rc('text',usetex = True)
```

(For other possibilities, consult the `rc` docstring.) Next create the title with

```
plt.title(r'The level contours of $z=x^2-y^2$',fontsize=20)
```

where the *string* contains regular LaTeX syntax. This is used to create the title in Figure 5.8.

5.7.3 Alternatives for LaTeX users

Clearly, either of the strategies outlined above for creating TeX-formatted strings is non-trivial, and both are restricted to figures created within *matplotlib*. There are at least two approaches which extend the facility to position precisely TeX strings in an already created figure and which, for the sake of clarity, we assume to be in PDF format.

The first works for LaTeX users on all platforms, and assumes the diagram has been included in a LaTeX source file. Then the widely available LaTeX package *pinlabel* does the job.

The open-source software *LaTeXit*[9] has a very intuitive graphical user interface and allows text positioning in any PDF file, and is my preferred choice. The downside is that although it is now a mature product, at present it is only available on the Macintosh OS X platform.

There is another open-source package *KlatexFormula*[10] with some of *LaTeXit*'s functionality which is available on all platforms, but I have not tested it.

5.8 Contour plots

Suppose we have a relation $F(x, y) = z$, and let z_0 be a fixed value for z. Subject to the conditions of the implicit function theorem, we can, after possibly interchanging the rôles of x and y, solve this relation, at least locally, for $y = f(x, z_0)$. These are the "contour curves" or "contours" of z. What do they look like as z_0 varies? In principle, we need to specify two-dimensional arrays of equal shape for each of x, y and z, and a vector of values for z_0. *Matplotlib* allows for a number of short cuts in this process. For example, according to the documentation we can omit the x- and y-arrays. However, *Matplotlib* then creates the missing arrays as integer-spaced grids using `np.meshgrid`, which creates the transpose of what is usually needed, and so we should be careful if using this option! If we do not specify the z_0 vector, we can give instead the number of contour curves that should be drawn, or accept the default value. By default, the z_0

[9] It can be downloaded from `http://www.chachatelier.fr/latexit/`.
[10] Its website is `http://klatexformula.sourceforge.net`.

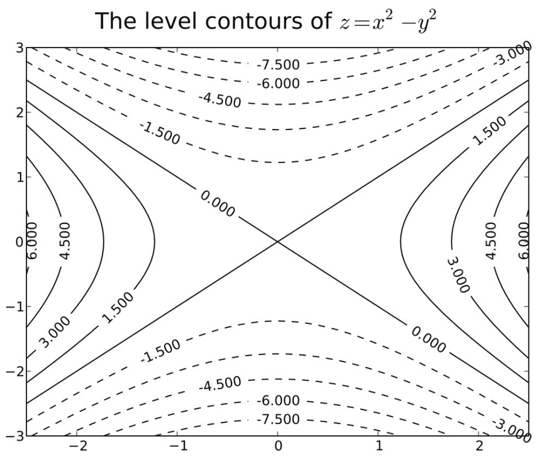

Figure 5.6 A plain contour plot using *matplotlib*.

values are not shown and so the `plt.clabel` function can be used to generate them, using as argument the value returned by `plt.contour`. The docstrings of these functions will reveal further options. Figure 5.6 was produced using the following snippet.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  plt.ion()
5  [X,Y] = np.mgrid[-2.5:2.5:51j,-3:3:61j]
6  Z=X**2-Y**2
7  curves=plt.contour(X,Y,Z,12,colors='k')
8  plt.clabel(curves)
9
10 plt.suptitle(r'The level contours of $z=x^2-y^2$',fontsize=20)
```

We have seen the ideas behind lines 1–6 before. `X`, `Y` and `Z` are 51×61 arrays of float. In line 7, the function `plt.contour` attempts to draw 12 contour curves. Its return value is the set of curves. Then in line 8 we attach Z-value labels to the curves. With the standard defaults, *matplotlib* will display the curves using a rainbow of colours. In this particular case, they do not reproduce well in this monochrome book. The option `colors='k'` in line 7 ensures that they all appear in black (the colour corresponding to `'k'`, see Section 5.3.2). There are lots of other options available. The two function docstrings list the details. Finally, line 10 creates a title according to the first prescription in Section 5.7.1.

An alternative view is provided by filling the spaces between the contour lines with colour, and supplying a colour bar. For example Figure 5.7 was created by replacing lines 7 and 8 in the snippet above by

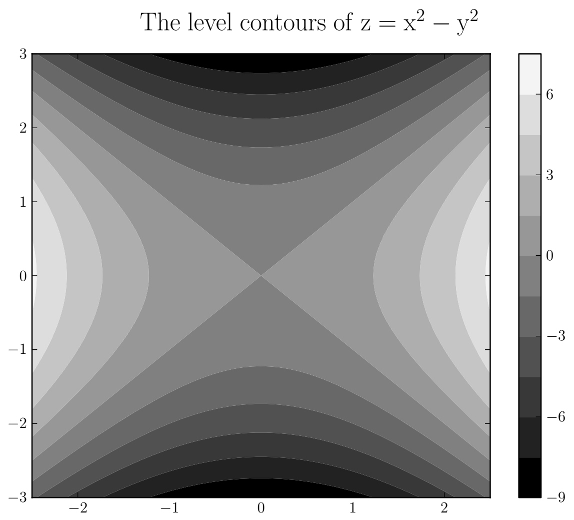

Figure 5.7 A filled contour plot using *matplotlib*.

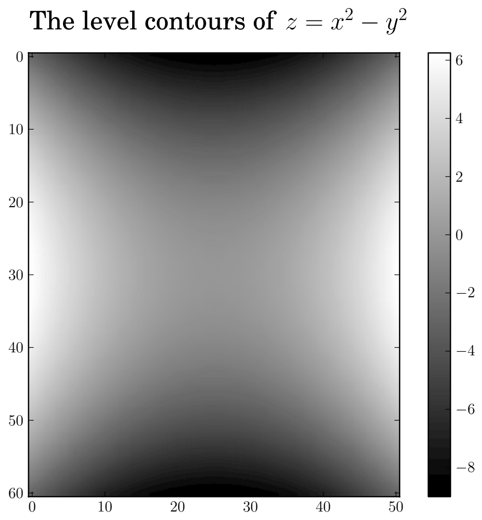

Figure 5.8 Contours implied in a continuous image using *matplotlib*.

```
plt.contourf(X,Y,Z,12)
plt.colorbar()
```

and the second prescription in Section 5.7.1 was used to replace line 10 generating the title.

Finally, we can construct a third view, shown as Figure 5.8 with continuous rather than discrete colours. This was produced by simply replacing the first line of the amendment above by

```
plt.imshow(Z)
```

We also used the prescription from Section 5.7.2 to generate the title. When using the `plt.imshow` option, we need to bear in mind that `np.meshgrid` is used with default integer grids of the appropriate size. In order to preserve the spatial information, we may need to alter the `extent` and/or `aspect` parameters. The docstring contains the details. Also an array transpose may be needed to alleviate the meshgrid problem.

5.9 Compound figures

We saw in Figure 5.2 how we could present a great deal of information in a single figure. In many cases, this is not the optimal approach. In order to avoid unnecessary nomenclature, we introduce a commonplace paradigm. A creative artist would choose to use either one or the other or a combination of two strategies: start again on additional new pages of the sketching block, or place several smaller figures on the same page. The first strategy is conceptually the simpler and so we treat its *matplotlib* analogue first.

Before starting this, we need to note an important point, which is underplayed in the *matplotlib* literature. The *matplotlib* module owes its power and versatility to its strongly *class*-based structure. (Section 3.9 contains an introduction to *classes*.) However, its developers went to some lengths to disguise this, with the intention of attracting Matlab users to convert to *matplotlib*, by trying to use Matlab's syntax wherever possible.

5.9.1 Multiple figures

In fact, this subsection requires minimal understanding of the *matplotlib class* structures. All we need to know is that what an artist would regard as a page of the sketching block, and a *matplotlib* user would see as a screen window, is actually an instance of the `Figure` class. We create a new instance (new page, new screen) with the command `plt.figure()`. In order to obtain consistency with Matlab, *matplotlib* does this silently for the first instance.

Here is a simple example for the creation of two very simple figures.

```
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(0,2*np.pi,301)
y=np.cos(x)
z=np.sin(x)

plt.ion()
```

```
 9  plt.plot(x,y) # First figure
10  plt.figure()
11  plt.plot(x,z) # Second figure
```

While this is very simple, it is important to realize that all work on the first figure (including decorating, saving or printing) must be done before we choose a new page with the `plt.figure()` command in line 10. With this approach we cannot review the "previous page". Note that this snippet creates two *matplotlib* windows. In most installations, the second will be superposed on the first, and so needs to be dragged aside so that both are visible.

5.9.2 Multiple plots

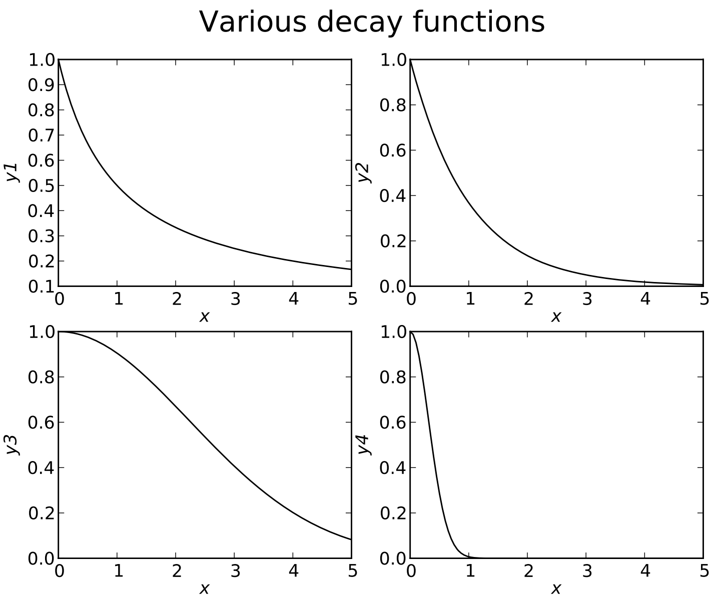

Figure 5.9 An example of compound plots using *matplotlib*.

Sometimes there is a need to present several plots in the same figure, i.e., a compound figure. In many plotting packages, this is an onerous complicated procedure. However, this task is performed easily in *matplotlib* at the expense of learning some minor syntactical changes. We present here a totally artificial example, showing four different decay rates in the same figure. Obviously, the code snippet needed to produce Figure 5.9 needs to be longer than usual.

```
x=np.linspace(0,5,101)
y1=1.0/(x+1.0)
y2=np.exp(-x)
```

```
4  y3=np.exp(-0.1*x**2)
5  y4=np.exp(-5*x**2)
6  plt.ion()
7  fig1=plt.figure()
8  ax1=fig1.add_subplot(2,2,1)
9  ax1.plot(x,y1)
10 ax1.set_xlabel('x')
11 ax1.set_ylabel('y1')
12 ax2=fig1.add_subplot(222)
13 ax2.plot(x,y2)
14 ax2.set_xlabel('x')
15 ax2.set_ylabel('y2')
16 ax3=fig1.add_subplot(223)
17 ax3.plot(x,y3)
18 ax3.set_xlabel('x')
19 ax3.set_ylabel('y3')
20 ax4=fig1.add_subplot(224)
21 ax4.plot(x,y4)
22 ax4.set_xlabel('x')
23 ax4.set_ylabel('y4')
24 fig1.suptitle('Various decay functions')
```

Consider the code snippet. Lines 1–5 construct four vectors of data. Next we need to introduce some slightly unfamiliar notation. Readers who have mastered Section 3.9 will recognize what is going on here. First, we construct in line 7 a "Figure class object" using the `plt.figure` function. As we saw in the last subsection, a "Figure object" corresponds more or less with what an artist would draw on a single page. Notice too that we have discarded, unused, the "Figure object" that *matplotlib* automatically creates, and have given the new figure the identifier `fig1`. (This means that if we subsequently generate a new "Figure object" with say `fig2=plt.figure()`, we can move between them, akin to the artist flipping pages, which was not possible with the approach of the previous subsection.)

If you look at Figure 5.9, you will see that the figure comprises a title and four subplots. Let us deal with the title first. It is constructed in line 24 of the snippet using the `suptitle` function associated with the class instance `fig1`. Now for the four subplots. We construct the first of these using line 8. The function `fig1.add_subplot(2,2,1)` takes into account that there will be an array of subplots with two rows and two columns. The final argument notes that we are dealing with the first of these. The return value of that function is an "Axes subplot class object" with the identifier `ax1`. If you try `ax1.`TAB within *IPython*, you will discover that over 300 possible functions are available, and three of them are used to plot the curve and label the axes in lines 9–11. Lines 12–23 repeat the process for each of the other three plots. Provided that the total number of plots is less than ten (the usual case), there is a standard abbreviation used in, e.g., line 12, where `add_subplot(2,2,2)` has been shortened to `add_subplot(222)`. There is a myriad of other features which can be used, and a careful perusal of the various function docstrings will reveal them. Note in particular that `ax1.title(string)` prints a title relevant to that subplot.

When constructing elaborate compound figures, it is important to keep in mind the limitations of space. If subplots start overlapping, we should consider invoking the function `plt.tight_layout`.

New users may feel justifiably irritated that the syntax used in the earlier sections of this chapter for creating single, i.e., non-compound, plots, e.g., `plt.xlabel`, differs from that used in this section, e.g., `ax1.set_xlabel`. Fundamentally, the concepts of this section take precedence. For example, if we were to replace line 8 of the snippet above by

```
ax=fig.add_subplot(111)
```

and used the 300 "axis" based functions, we could recreate all of the earlier examples. As we shall see in Section 6.3.1, this syntax is also used by the *mplot3d* extension of *matplotlib* to pseudo-three-dimensional plots. However, the *matplotlib* developers decided that if an `add_subplot` call was not present, then they had the freedom to create an environment consistent with that of *Matlab*, so as to assist the hoped-for exodus from *Matlab* to *matplotlib*, and so sophisticated *matplotlib* users must either live with two sets of commands or use the approach indicated above.

5.10 Animations

So far, we have looked mainly at functions of a single variable, $x = x(t)$ or $y = y(x)$. We considered briefly $z = z(x, y)$ and showed how to construct the contour curves $z = z_0$ constant, by interpolation so as to get $y = y(x, z_0)$. We next consider the visualization of functions of two variables, $z = z(t, x)$ or $z = z(x, y)$. These equations define a surface in three-dimensional space with coordinates (t, x, z) or (x, y, z). There are two approaches. We can either project the slice into two dimensions, the subject of the next chapter, or we can build a set of slices, each at say constant t, and project them in sequence so as to form an animation, the subject of this section.

There are a number of different ways of constructing animations, most depending on the specific platform on which Python is running, and also on the options with which *matplotlib* was compiled. We present here two very different approaches which aim, as far as possible, to remove these dependencies. In the first, we construct an animation within *matplotlib*. In the second, we construct a movie file which can then be used externally, e.g., in presentations. As a concrete example, we consider a very simple curve $z(t, x) = \operatorname{sech}^2(0.5 * (t - x))$, which arises when considering a single soliton travelling wave solution of the Korteweg–de Vries equation in dimensionless coordinates.

5.10.1 In situ animations

In order to get a clearer idea of what is involved, consider first the following snippet which just draws a single curve.

```python
import numpy as np
import matplotlib.pyplot as plt

def sol(t,x):
    return 0.5/np.cosh(0.5*(x-t))**2

x=np.linspace(0,60.0,1001)
plt.ion()
plt.plot(x, sol(10,x))
```

We want to leave the axes and any labels fixed. Note also that in this example, the range of the `sol` function is fixed as $[0, 0.5]$. We certainly do not want the y-extent enlarging or contracting during the animation. If this is likely to be a problem, then we need to invoke `plt.ylim(ymin,ymax)` to set the y-extent. We see that what we need to change during the animation is just the curve. So how do we get an identifier for it?

Until now we have regarded `plt.plot(x,y)` as a command. In fact, as its docstring shows, its return value is the lines that were added, packed in a *tuple*. Thus supposing there were two of them, we could get identifiers for them by writing

```python
(line1,line2)=plt.plot(x,y)
```

Here there is only one. We can unpack the identifier with `(line,)=plt.plot(x,y)`, where the trailing comma reminds us that we have a *tuple*, see Section 3.5.5, which is coercible to a *list*. Clearly, the parentheses are redundant and so it is more idiomatic to write `line,=plt.plot(x,y)`. If we then try `line.?` (in *IPython*), we discover what attributes `line` has. The useful ones here are `line.set_xdata` and `line.set_ydata`, which accept one-dimensional arrays as input. We do not want to change the x-values, but we shall change the y-values.

To make the animation, we shall loop over t-values. Where the data are as simple as they are here, this loop will run so fast, that we will not see the individual plots. We therefore need to pause each iteration. The Python `time` module includes a useful function `time.sleep(secs)`, which does precisely that. It is well worth experimenting with the following snippet, which is self-contained. Type and save it in a file, say `foo.py` and run it in the interpreter with the line `run foo`. Note in particular that the choice of the sleep interval is a matter of taste, and depends on your installation.

```python
import numpy as np
import matplotlib.pyplot as plt
import time

def sol(t,x):
    return 0.5/np.cosh(0.5*(x-t))**2

x=np.linspace(0,60.0,1001)

plt.ion()
plt.xlabel('x')
plt.ylabel('y')
line,=plt.plot(x,sol(10,x))
for t in np.linspace(-10,70,161):
    line.set_ydata(sol(t,x))
    plt.draw()
    plt.title('Soliton wave at t = %5.1f' % t)
    time.sleep(0.1)
```

Of course, in a realistic situation the data we wish to plot may not be so simple. The purpose of this example is to illustrate simply and clearly the basics of in situ animation, without getting too involved in the computation of individual images.

5.10.2 Movies

In situ animations are simple, but they offer little control, pause, rewind etc., and they require a running Python programme. For many purposes, especially presentations, you might prefer a stand-alone movie. In this section, we show how to achieve this. There are two steps to this process. First, we have to build a set of frame files, one for each t-value. Next we have to consider how to convert this collection of frame files into a movie. The manual, Matplotlib Community (2013), suggests using the `mencoder` function, or the `convert` utility from the *ImageMagick* package. This author recommends the `ffmpeg` package,[11] which is readily available for most platforms, as is the *ImageMagick* package.[12] In fact, the latter has binary packages available on its website for all of the major platforms. Our code snippet below covers both cases.

Our movie will be made up of many frames, one for each t-value. Since we have seen how each frame is constructed already, we encapsulate it in a function `draw_frame(i)`, which carries out the plot for the `i`th frame.

```
import numpy as np
import matplotlib.pyplot as plt

def sol(t,x):
    return 0.5/np.cosh(0.5*(x-t))**2

def draw_frame(t,x):
     """ Draw a frame. """
```

[11] See `http://www.ffmpeg.org`.
[12] See `http://www.ImageMagick.org`.

```
 9      plt.plot(x,sol(t,x))
10      plt.axis((0,60.0,0,0.5))
11      plt.xlabel('x')
12      plt.ylabel('y')
13      plt.title('Soliton wave at t = %05.1f' % t)
14
15  x=np.linspace(0,60.0,1001)
16  t=np.linspace(-10,70,901)
17
18  for i in range(len(t)):
19     file_name='_temp%05d.png' % i
20     draw_frame(t[i],x)
21     plt.savefig(file_name)
22     plt.clf()
23
24  import os
25  os.system("rm _movie.mpg")
26  os.system("/opt/local/bin/ffmpeg -r 25 " +
27          " -i _temp%05d.png -b:v 1800 _movie.mpg")
28  #os.system("/opt/local/bin/convert _temp*.png _movie.mpg")
29  os.system("rm _temp*.png")
```

The main loop, lines 18–22, now carries out the following steps for each `i`. Create a file name made up of `_temp`, `i` and the suffix `png`. Then plot the frame and save it to file using the name just generated. Finally, line 22, clear the frame for the next plot. At the end of this step, we have very many files `_temp00000.png`, `_temp00001.png`, ... We could of course have chosen another file format rather than `png`, but this one is probably a reasonable compromise between quality and size.

For the second part of the process, we import the `os` module, and in particular the function `os.system()`. This is very simple. It accepts a *string*, interprets it as a command line instruction and carries it out. This snippet assumes a Unix/Linux operating system, and Windows users will need to modify it. The first invocation of `os.system` deletes a file `_movie.mpg` (if it exists) from the current directory. As written, the second invocation creates a command line invocation for `ffmpeg`. Note that Python does not know your user profile, and so the first item in the *string* is the absolute path name of my copy of the `ffmpeg` binary. Then follow the parameters that `ffmpeg` needs. The basic ones are `-r 25`, i.e., 25 frames per second, and `-b:v 1800`, the bit rate. The choices here work for me. The input files follow `-i`, and `_temp%05d.png` is interpreted as the collection of frame files we have just created. The last item is the output file `_movie.mpg`. If you want to use ImageMagick's `convert` utility, comment out the lines 26 and 27 and uncomment line 28. Again the absolute path name is mandatory. The final line deletes all of the frame files we created earlier, leaving the movie file which can be viewed using any standard utility. Of course, other choices of movie format rather than `mpg` are possible.

This snippet creates and then destroys 901 frame files. Its speed depends on your installation. The reason for using Python to carry out these system commands should be evident. Once we are happy with the code, we can wrap it as a self-contained function for making movies! This is an instance of the use of Python as a scripting language.

5.11 Mandelbrot sets: a worked example

We finish this chapter with a somewhat longer example to show how simple Python commands can push at the limits of image-processing. Since we have not yet considered the numerical solution of differential equations, we (artificially) restrict ourselves to discrete processes. My chosen example, while relatively simple to present, has extremely complicated dynamics and the challenge is to represent them graphically. Although the reader may have no professional interest in Mandelbrot sets, the discussion of their implementation raises a number of technical points that are more general, e.g.:

- performing operations hundreds of millions of times efficiently,
- removing points dynamically from multi-dimensional arrays,
- creating high-definition images pixel by pixel.

This is why the example was chosen.

The book Peitgen and Richter (1986) drew attention to the utility of computer graphics for giving insight into fractals. There are many sites on the internet drawing attention both to the remarkable figures which can illustrate the boundary of the Mandelbrot set, the best-known fractal, and to the programmes which generate them. Although programmes as short as 100–150 characters exist, the aim of this section is to build on what has been introduced so far and show how to create a programme which produces insightful images quickly. We first describe very briefly the underlying mathematics, and then discuss an algorithm to display the set boundary. Finally, we explain how the code snippet below implements the algorithm.

We shall describe the complex plane in terms of cartesian coordinates x, y or in terms of the complex variable $z = x + iy$, where $i^2 = -1$. We need also a map $z \to f(z)$ of the complex plane to itself. Mandelbrot chose $f(z) = z^2 + c$, where c is a constant, but many other choices are possible. An iterated sequence is then defined by

$$z_{n+1} = {z_n}^2 + c, \qquad \text{with } z_0 \text{ given, and } n = 0, 1, 2, \ldots \tag{5.1}$$

It is customary to choose $z_0 = 0$, but by a trivial renumbering we choose $z_0 = c$. The equation (5.1) then defines $z_n = z_n(c)$.

Here are two examples, corresponding to $c = 1$ and $c = i$ respectively.

$$z_n = \{1, 2, 5, 26, 677, \ldots\}, \qquad z_n = \{i, -1 + i, -i, -1 + i, -i, \ldots\}.$$

We focus attention on the behaviour of $z_n(c)$ as $n \to \infty$. If $|z_n(c)|$ remains bounded, we say that c lies in the *Mandelbrot set* $\mathcal{M}$. Clearly, $c = i$ is in $\mathcal{M}$, but $c = 1$ is not. A deep result is that $\mathcal{M}$ is a connected set, i.e., it possesses an inside and outside, and we have exhibited elements of both.

Recall the triangle inequality: if u and v are complex numbers, then $|u + v| \leqslant |u| + |v|$. Using this result, it is straightforward to show that if $|c| \geqslant 2$ then $z_n(c) \to \infty$, so that c lies outside $\mathcal{M}$. Now suppose that $|c| < 2$. If we find that $|z_N(c)| > 2$ for some N, then the triangle inequality can be used to show that $|z_n(c)|$ increases as n increases beyond N and so $|z_n(c)| \to \infty$ as $n \to \infty$. If c lies outside $\mathcal{M}$, there will be a smallest such N, and for the purposes of our algorithm, we shall call it the escape parameter $\epsilon(c)$. The parameter is not defined if c lies in $\mathcal{M}$, but it is convenient to say $\epsilon(c) = \infty$ in this case.

We can carry out an analogous study for Julia sets. Recall that we iterated the recurrence relation (5.1) for fixed $z_0 = 0$ with the parameter c varying in order to obtain the Mandelbrot set. Now consider the alternative approach holding the parameter c fixed and varying the starting point z_0. Then if $|z_n(c)|$ remains bounded, we say that z_0 lies in the *Julia set* $\mathcal{J}(c)$.

The role of the computer is to help to visualize $\epsilon(c)$ when c lies in some domain in the complex plane. It turns out that this function has extremely complicated behaviour and so surface plots are inappropriate. Because structure is observed to exist on all scales, contour plots are misleading. Instead, we use colour to represent the value of the escape parameter, and this can be done on a pixel-by-pixel basis. This creates the figures which have captured the imagination of so many. Here we explore the classic visualization of the Mandelbrot set. Later, in Section 6.6, we look at an alternative view of the Julia set for the process (5.1).

We first choose a rectangular domain $x_{lo} \leqslant x \leqslant x_{hi}$, $y_{lo} \leqslant y \leqslant y_{hi}$. We know how to construct `x` and `y` on an evenly spaced rectangular grid using `np.mgrid`, and we can then build the parameter `c = x + 1j*y` at each grid point. Next for each grid point we make `z` a copy of `c`. We need to specify an integer parameter `max_iter`, the maximum number of Mandelbrot iterations that we are prepared to carry out. We then construct an iteration loop to evolve `z` to `z**2+c`. At each stage of the loop, we test if `np.abs(z)>2`. On the first time this happens we fix the escape parameter `eps` to be the iteration number. If this never happens, we set `eps` to `max_iter`. We then proceed to the next grid point. At the end of this process, we have a rectangular array of eps we we can dispatch for visualization. This is straightforward. There is however an issue which we need to take into account. The grid dimensions will be $> 10^3$ in each direction, and `max_iter` $> 10^2$. Thus we are looking at a minimum of 10^8 Mandelbrot iterations! We need to vectorize this calculation. In principle, this too is simple. We use a **`for`** loop to carry out the Mandelbrot and itEration use *numpy* arrays to evolve all of the grid points simultaneously. However, if, early on, we set the value of the escape parameter at a grid point, then we do not want to include that point in subsequent iterations.

This involves logical operations, and, with care, the discussion of logical operations on vectors in Section 4.1.5 can be extended to multi-dimensional arrays. However, the removal of points from a two-dimensional array is non-trivial in comparison with its one-dimensional analogue. We therefore need to "flatten" our two-dimensional arrays to produce one-dimensional vectors. We take the `z`-array and use the `reshape` function to construct a vector containing the same points. This is done by pointers. There is no actual copying of the data. After we have iterated the vector, we check for "escaped points", i.e., those for which $|z| > 2$. For all escaped points, we write simultaneously the escape parameter. Next we form a new vector with the escaped points deleted. Again this can be done with no actual copying. Then we iterate the smaller vector and so on.

This will be very fast, but there is a complication. We need to build a two-dimensional array of escape parameters, and this positional information has been lost in the flattened truncated vector. The solution is to create a pair of "index vectors" carrying the *x* and *y*-positions and to truncate them in exactly the same way as we truncate the *z*-vector. Thus for any point in the *z*-vector we can always recover its coordinates by looking at the corresponding points in the two index vectors.

The code snippet below can be divided into five sections. The first, lines 5–9, is simple, because we merely set the parameters. (If we were to recast the operation as a function, these would be the input arguments.)

The second, lines 12–15, sets up the arrays using hopefully familiar functions. The array `esc_parms` needs comment though. The convention in image-processing is to reverse the order of *x*- and *y*-coordinates (effectively a transpose) and for each point we need a triple of unsigned integers of length 8 which will hold the red, green, blue (rgb) data for that pixel.

Next, in lines 18–21 of the snippet, we "flatten" the arrays for `ix`, `iy` and `c` using the reshape function. No copying occurs and the flattened arrays contain the same number of components as their two-dimensional analogues. At this stage, we introduce in line 22 a `z`-vector which, initially, is a copy of the `c`-vector, the starting point for the iteration.

```
1   import numpy as np
2   from time import time
3
4   # Set the parameters
5   max_iter=256                # maximum number of iterations
6   nx, ny=1024, 1024           # x- and y-image resolutions
7   x_lo, x_hi=-2.0,1.0         # x bounds in complex plane
8   y_lo, y_hi=-1.5,1.5         # y bounds in complex plane
9   start_time=time()
10
11  #  Construct the two dimensional arrays
12  ix,iy=np.mgrid[0:nx,0:ny]
13  x,y=np.mgrid[x_lo:x_hi:1j*nx,y_lo:y_hi:1j*ny]
14  c=x+1j*y
15  esc_parm=np.zeros((ny,nx,3),dtype='uint8') # holds pixel rgb data
16
17  # Flattened arrays
18  nxny=nx*ny
19  ix_f=np.reshape(ix,nxny)
20  iy_f=np.reshape(iy,nxny)
21  c_f=np.reshape(c,nxny)
22  z_f=c_f.copy()                          # the iterated variable
23
```

```
24  for iter in xrange(max_iter):      # do the iterations
25      if not len(z_f):               # all points have escaped
26          break
27      # rgb values for this choice of iter
28      n=iter+1
29      r,g,b=n % 4 * 64,n % 8 * 32,n % 16 * 16
30      # Mandelbrot evolution
31      z_f*=z_f
32      z_f+=c_f
33      escape=np.abs(z_f) > 2.0      # points which are escaping
34      # Set the rgb pixel value for the escaping points
35      esc_parm[iy_f[escape],ix_f[escape],:]=r, g, b
36      escape=-escape                # points not escaping
37      # Remove batch of newly escaped points from flattened arrays
38      ix_f=ix_f[escape]
39      iy_f=iy_f[escape]
40      c_f=c_f[escape]
41      z_f=z_f[escape]
42
43  print "Time taken = ", time() - start_time
44
45  from PIL import Image
46
47  picture=Image.fromarray(esc_parm)
48  picture.show()
49  picture.save("mandelbrot.jpg")
```

The **for** loop, lines 24–41, carries out the Mandelbrot iteration. We first test whether the z-vector is empty, i.e., all of the points have escaped, and if so we halt the iteration. In lines 28 and 29 we choose a triple of rgb values given by an arbitrary encoding of the iteration counter. (This can be replaced by another of your choice.) Lines 31 and 32 carry out the iteration. It would have been simpler and clearer to have used one line, `z_f=z_f*z_f+c_f`. However, this involves creating two temporary arrays, which are not needed in the version shown. This version runs in 80% of the time taken by `z_f=z_f*z_f+c_f`. Line 33 tests for escaping. `escape` is a vector of Booleans of the same size as `z_f`, with the value `True` if `abs(z_f)>2` and `False` otherwise. The next line writes the rgb data for those components where escape is `True`. It is at this point that we need the `ix_f` and `iy_f` vectors which carry the positions of the points.

As we can readily check, if `b` is an vector of Booleans, then `-b` is a vector of the same size with `True` and `False` interchanged. Thus after line 36, escape is `True` for all points except the escaped ones. Now lines 38–41 remove the escaped points from the arrays `ix_f`, `iy_f`, `c_f` and `z_f`, after which we traverse the loop again. Note that if a point has not escaped after `max_iter` iterations, its rgb value is the default, zero, and the corresponding colour is black.

The final part of the code, lines 45–49, invoke the Python Imaging Library (PIL) to convert the rgb array to a picture, to display it and to save it as a file. PIL should have been included in your Python installation package. If not, it can be downloaded together with documentation from its website.[13]

The black and white version of the output of the code snippet above is shown below as Figure 5.10. It is worthwhile experimenting with other smaller domains in the complex plane, in order to appreciate the richness of the Mandelbrot set boundary. You might prefer a less sombre colour scheme, and it is easy to change line 29 of the snippet to achieve this.

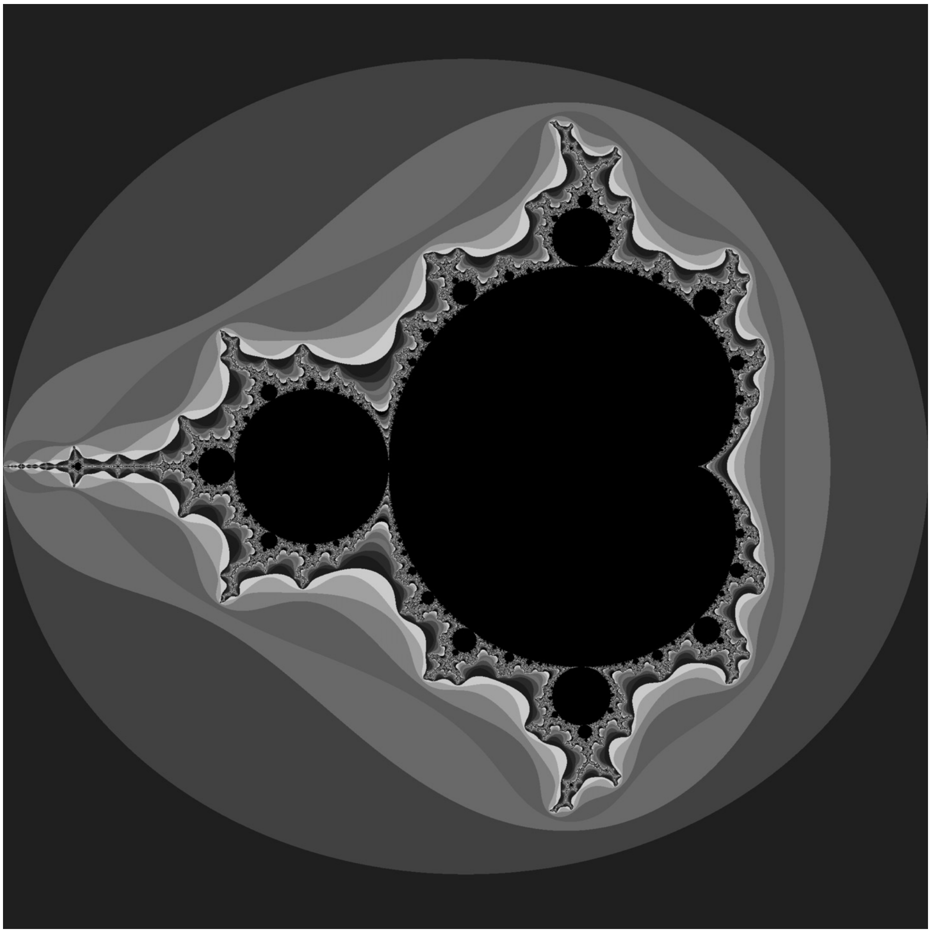

Figure 5.10 An example of a Mandelbrot set

[13] The website is `http://www.pythonware.com/products/pil`.