Python for Scientists

Python is a free, open source, easy-to-use software tool that offers a significant alternative to proprietary packages such as Matlab and Mathematica. This book covers everything the working scientist needs to know to start using Python effectively.

The author explains scientific Python from scratch, showing how easy it is to implement and test non-trivial mathematical algorithms and guiding the reader through the many freely available add-on modules. A range of examples, relevant to many different fields, illustrate the program's capabilities. In particular, readers are shown how to use pre-existing legacy code (usually in Fortran77) within the Python environment, thus avoiding the need to master the original code.

Instead of exercises the book contains useful snippets of tested code which the reader can adapt to handle problems in their own field, allowing students and researchers with little computer expertise to get up and running as soon as possible.

Python for Scientists

JOHN M. STEWART

Department of Applied Mathematics & Theoretical Physics
University of Cambridge

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107061392

First published 2014

Printed in the United Kingdom by Clays, St Ives plc

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
Stewart, John, 1943 July 1–
Python for scientists / John M. Stewart.
pages cm
Includes bibliographical references and index.
ISBN 978-1-107-06139-2 (hardback)
1. Science – Data processing. 2. Python (Computer program language) I. Title.
Q183.9.S865 2014
005.13′3 – dc23 2014010571

ISBN 978-1-107-06139-2 Hardback
ISBN 978-1-107-68642-7 Paperback

Contents

Preface

I have used computers as an aid to scientific research for over 40 years. During that time, hardware has become cheap, fast and powerful. However, software relevant to the working scientist has become progressively more complicated. My favourite textbooks on Fortran90 and C++ run to 1200 and 1600 pages respectively. And then we need documentation on mathematics libraries and graphics packages. A newcomer going down this route is going to have to invest significant amounts of time and energy in order to write useful programmes. This has led to the emergence of "scientific packages" such as Matlab or Mathematica which avoid the complications of compiled languages, separate mathematics libraries and graphics packages. I have used them and found them very convenient for executing the tasks envisaged by their developers. However, I also found them very difficult to extend beyond these boundaries, and so I looked for alternative approaches.

Some years ago, a computer science colleague suggested that I should take a look at Python. At that time, it was clear that Python had great potential but a very flaky implementation. It was however free and open-source, and was attracting what has turned out to be a very effective army of developers. More recently, their efforts have coordinated to produce a formidable package consisting of a small core language surrounded by a wealth of add-on libraries or *modules*. A select group of these can and do replicate the facilities of the conventional scientific packages. More importantly an informed, intelligent user of Python and its modules can carry out major projects usually entrusted to dedicated programmers using Fortran, C etc. There is a marginal loss of execution speed, but this is more than compensated for by the vastly telescoped development time. The purpose of this book is to explain to working scientists the utility of this relatively unknown resource.

Most scientists will have some computer familiarity and programming awareness, although not necessarily with Python and I shall take advantage of this. Therefore, unlike many books which set out to "teach" a language, this one is not just a brisk trot through the reference manuals. Python has many powerful but unfamiliar facets, and these need more explanation than the familiar ones. In particular, if you encounter in this text a reference to the "beginner" or the "unwary", it signifies a point which is not made clear in the documentation, and has caught out this author at least once.

The first six chapters, plus Appendix A, cover almost everything the working scientist needs to know in order to get started in using Python effectively. My editor and some referees suggested that I should devote the second half of the book to problems in a particular field. This would have led to a series of books, “Python for Biochemists”, “Python for Crystallographers”,. . . , all with a common first half. Instead I have chosen to cover just three topics, which however, should be far more widely applicable in many different fields. Chapter 7 covers four radically different types of ordinary differential equations and shows how to use the various relevant black boxes, which are often Python wrappers around tried and trusted Fortran codes. The next chapter while ostensibly about pseudospectral approaches to evolutionary partial differential equations, actually covers a topic of great utility to many scientists, namely how to reuse legacy code, usually written in Fortran77 within Python at Fortran-like speeds, without understanding Fortran. The final chapter about solving very large linear systems via multigrid is also a case history in how to use object-oriented programming meaningfully in a scientific context. If readers look carefully and critically at these later chapters, they should gain the practical expertise to handle problems in their own field.

Acknowledgments are due to the many Python developers who have produced and documented a very useful tool, and also to the very many who have published code snippets on the web, a great aid to the tyro, such as this author. Many of my colleagues have offered valuable advice. Des Higham generously consented to my borrowing his ideas for the last quarter of Chapter 7. I am especially grateful to Oliver Rinne who read carefully and critically an early draft. At Cambridge University Press, my Production Editor, Jennifer Murphy and my Copy Editor, Anne Rix have exhibited their customary expertise. Last but not least I thank the Department of Applied Mathematics and Theoretical Physics, Cambridge for continuing to offer me office space after my retirement, which has facilitated the production of this book.

Writing a serious book is not a trivial task and so I am rather more than deeply grateful for the near-infinite patience of *Mary*, the “Python-widow”, which made this book possible!

1 Introduction

The title of this book is “Python for Scientists”, but what does that mean? The dictionary defines “Python” as either (a) a non-venomous snake from Asia or Saharan Africa or (b) a computer scripting language, and it is the second option which is intended here. (What exactly this second definition means will be explained later.) By “scientist”, I mean anyone who uses quantitative models either to obtain conclusions by processing pre-collected experimental data or to model potentially observable results from a more abstract theory, **and** who asks “what if?”. What if I analyze the data in a different way? What if I change the model? Thus the term also includes economists, engineers, mathematicians among others, as well as the usual concept of scientists. Given the volume of potential data or the complexity (non-linearity) of many theoretical models, the use of computers to answer these questions is fast becoming mandatory.

Advances in computer hardware mean that immense amounts of data or evermore complex models can be processed at increasingly rapid speeds. These advances also mean reduced costs so that today virtually every scientist has access to a “personal computer”, either a desktop work station or a laptop, and the distinction between these two is narrowing quickly. It might seem to be a given that suitable software will also be available so that the “what if” questions can be answered readily. However, this turns out not always to be the case. A quick pragmatic reason is that while there is a huge market for hardware improvements, scientists form a very small fraction of it and so there is little financial incentive to improve scientific software. But for scientists, this issue is important and we need to examine it in more detail.

1.1 Scientific software

Before we discuss what is available, it is important to note that all computer software comes in one of two types: proprietary and open-source. The first is supplied by a commercial firm. Such organizations have both to pay wages and taxes and to provide a return for their shareholders. Therefore, they have to charge real money for their products, and, in order to protect their assets from their competitors, they do not tell the customer how their software works. Thus, the end-users have therefore little chance of being able to adapt or optimize the product for their own use. Since wages and taxes are recurrent expenditure, the company needs to issue frequent charged-for updates and improvements (the *Danegeld effect*). Open-source software is available for free or at nominal cost (media, postage etc.). It is usually developed by computer literate individuals, often working for universities or similar organizations, who provide the service for their colleagues. It is distributed subject to anti-copyright licences, which give nobody the right to copyright it or to use it for commercial gain. Conventional economics might suggest that the gamut of open-source software should be inferior to its proprietary counterpart, or else the commercial organizations would lose their market. As we shall see, this is not necessarily the case.

Next we need to differentiate between two different types of scientific software. Computers operate according to a very limited and obscure set of instructions. A programming language is a somewhat less limited subset of human language in which sequences of instructions are written, usually by humans, to be read and understood by computers. The most common languages are capable of expressing very sophisticated mathematical concepts, albeit with a steep learning curve. Only a few language families, e.g., C and Fortran have been widely accepted, but they come with many different dialects, e.g., Fortran77, Fortran90, Ansi C, C++ etc. Compilers then translate code written by humans into machine code which can be optimized for speed and then processed. As such, they are rather like Formula 1 racing cars. The best of them are capable of breathtakingly fast performance, but driving them is not intuitive and requires a great deal of training and experience. Note that compilers need to be supplemented by libraries of software packages which implement frequently used numerical algorithms, and graphics packages will usually be needed. Fast versatile library packages are usually expensive, although good public domain packages are starting to appear.

A racing car is not usually the best choice for a trip to the supermarket, where speed is not of paramount importance. Similarly, compiled languages are not always ideal for trying out new mathematical ideas. Thus for the intended readers of this book the direct use of compilers is likely to be unattractive, unless their use is mandatory. We therefore look at the other type of software, usually called "scientific packages". Proprietary packages include Mathematica and Matlab, and open-source equivalents include Maxima, Octave, R and SciLab. They all operate in a similar fashion. Each provides its own idiosyncratic programming language in which problems are entered at a user interface. After a coherent group of statements, often just an individual statement, has been typed, the package writes equivalent core language code and compiles it on the fly. Thus errors and/or results can be reported immediately back to the user. Such packages are called "interpreters", and older readers may remember, perhaps with mixed feelings, the BASIC language. For small projects, the slow operation compared to a fully compiled code is masked by the speed of current microprocessors, but it does become apparent on larger jobs.

These packages are attractive for at least two reasons. The first is their ability to post-process data. For example, suppose that x is a real variable and there exists a (possibly unknown) function $y(x)$. Suppose also that for a set X of discrete instances of x we have computed a corresponding set Y of instances of y. Then a command similar to `plot(X,Y)` will display instantly a nicely formatted graph on the screen. Indeed, those generated by Matlab in particular are of publication quality. A second advantage is the apparent ability of some of the proprietary packages to perform in addition some algebraic and analytic processes, and to integrate all of them with their numerical and graphical properties. A disadvantage of all of these packages is the quirky syntax and limited expressive ability of their command languages. Unlike the compiled languages, it is often extremely difficult to programme a process which was not envisaged by the package authors.

The best of the proprietary packages are very easy to use with extensive on-line help and coherent documentation, which has not yet been matched by all of the open-source alternatives. However, a major downside of the commercial packages is the extremely high prices charged for their licences. Most of them offer a cut down “student version” at reduced price (but usable only while the student is in full-time education) so as to encourage familiarity with the package. This largesse is paid for by other users.

Let us summarize the position. On the one hand, we have the traditional compiled languages for numerics which are very general, very fast, very difficult to learn and do not interact readily with graphical or algebraic processes. On the other, we have standard scientific packages which are good at integrating numerics, algebra and graphics, but are slow and limited in scope.

What properties should an ideal scientific package have? A short list might contain:

1 a programming language which is both easy to understand and which has extensive expressive ability,
2 integration of algebraic, numerical and graphical functions,
3 the ability to generate numerical algorithms running with speeds within an order of magnitude of the fastest of those generated by compiled languages,
4 a user interface with adequate on-line help, and decent documentation,
5 an extensive range of textbooks from which the curious reader can develop greater understanding of the concepts,
6 open-source software, freely available,
7 implementation on all standard platforms, e.g., Linux, Mac OS X, Unix, Windows.

The bad news is that no single package satisfies all of these criteria.

The major obstruction here is the requirement of algebraic capability. There are two open-source packages, wx-Maxima and Reduce with significant algebraic capabilities worthy of consideration, but Reduce fails requirement 4 and both fail criteria 3 and 5. They are however extremely powerful tools in the hands of experienced users. It seems sensible therefore to drop the algebra requirement.[1]

In 1991, Guido van Rossum created Python as a open-source platform-independent general purpose programming language. It is basically a very simple language surrounded by an enormous library of add-on modules, including complete access to the underlying operating system. This means that it can manage and manipulate programmes built from other complete (even compiled) packages, i.e., a *scripting* language. This versatility has ensured both its adoption by power users such as Google, and a real army of developers. It means also that it can be a very powerful tool for the scientist. Of course,

[1] The author had initially intended to write a book covering both numerical and algebraic applications for scientists. However, it turns out that, apart from simple problems, the requirements and approaches are radically different, and so it seems more appropriate to treat them differently.

there are other scripting languages, e.g., Java and Perl, but none has the versatility or user-base to meet criteria 3–5 above.

Five years ago it would not have been possible to recommend Python for scientific work. The size of the army of developers meant that there were several mutually incompatible add-on packages for numerical and scientific applications. Fortunately, reason has prevailed and there is now a single numerical add-on package *numpy* and a single scientific one *scipy* around which the developers have united.

1.2 The plan of this book

The purpose of this intentionally short book is to show how easy it is for the working scientist to implement and test non-trivial mathematical algorithms using Python. We have quite deliberately preferred brevity and simplicity to encyclopaedic coverage in order to get the inquisitive reader up and running as soon as possible. We aim to leave the reader with a well-founded framework to handle many basic, and not so basic, tasks. Obviously, most readers will need to dig further into techniques for their particular research needs. But after reading this book, they should have a sound basis for this.

This chapter and Appendix A discuss how to set up a scientific Python environment. While the original Python interpreter was pretty basic, its replacement *IPython* is so easy to use, powerful and versatile that Chapter 2 is devoted to it.

We now describe the subsequent chapters. As each new feature is described, we try to illustrate it first by essentially trivial examples and, where appropriate, by more extended problems. This author cannot know the mathematical sophistication of potential readers, but in later chapters we shall presume some familiarity with basic calculus, e.g., the Taylor series in one dimension. However, for these extended problems we shall sketch the background needed to understand them, and suitable references for further reading will be given.

Chapter 3 gives a brief but reasonably comprehensive survey of those aspects of the core Python language likely to be of most interest to scientists. Python is an object-oriented language, which lends itself naturally to object-oriented programming (OOP), which may well be unfamiliar to most scientists. We shall adopt an extremely light touch to this topic, but need to point out that the container objects introduced in Section 3.5 do not all have precise analogues in say C or Fortran. Again the brief introduction to Python *classes* in Section 3.9 may be unfamiliar to users of those two families of languages. The chapter concludes with two implementations of the *sieve of Eratosthenes*, which is a classical problem: enumerate all of the prime numbers[2] less than a given integer n. A straightforward implementation takes 17 lines of code, but takes inordinately long execution times once $n > 10^5$. However, a few minutes of thought and using already described Python features suggests a shorter 13 line programme which runs 3000 times faster and runs out of memory (on my laptop) once $n > 10^8$. The point of this exercise is

[2] The restriction to integer arithmetic in this chapter is because our exposition of Python has yet to deal with serious calculations involving real or complex numbers efficiently.

that choosing the right approach (and Python often offers so many) is the key to success in Python numerics.

Chapter 4 extends the core Python language via the add-on module *numpy*, to give a very efficient treatment of real and complex numbers. In the background, lurk C/C++ routines to execute repetitive tasks with near-compiled-language speeds. The emphasis is on using structures via *vectorized* code rather than the traditional for-loops or do-loops. Vectorized code sounds formidable, but, as we shall show, it is much easier to write than the old-fashioned loop-based approach. Here too we discuss the input and output of data. First, we look at how *numpy* can read and write text files, human-readable data and binary data. Secondly, we look briefly at data analysis. We summarize also miscellaneous functions and give a brief introduction to Python's linear algebra capabilities. Finally, we review even more briefly a further add-on module *scipy* which greatly extends the scope of *numpy*.

Chapter 5 gives an introduction to the add-on module *matplotlib*. This was inspired by the striking graphics performance of the Matlab package and aspires to emulate or improve on it for two-dimensional x, y-plots. Indeed, almost all of the figures in Chapters 5–9 were produced using *matplotlib*. The original figures were produced in colour using the relevant code snippets. The exigencies of book publishing have required conversion to black, white and many shades of grey. After giving a range of examples to illustrate its capabilities, we conclude the chapter with a slightly more extended example, a fully functional 49-line code to compute and produce high-definition plots of Mandelbrot sets.

The difficulties of extending the discussion to three-dimensional graphics, e.g., representations of the surface $z = z(x, y)$ are discussed in Chapter 6. Some aspects of this can be handled by the *matplotlib* module, but for more generality we need to invoke the *mayavi* add-on module, which is given a brief introduction together with some example codes. If the use of such graphics is a major interest for you, then you will need to investigate further these modules.

If you already have some Python experience, you can of course omit parts of Chapters 3 and 4. You are however encouraged strongly to try out the relevant code snippets. Once you have understood them, you can deepen your understanding by modifying them. These "hacking" experiments replace the exercises traditionally included in textbooks. The same applies to Chapters 5 and 6, which cover Python graphics and contain more substantial snippets. If you already have an idea of a particular picture you would like to create, then perusal of the examples given here and also those in the *matplotlib gallery* (see Section 5.1) should produce a recipe for a close approximation which can be "hacked" to provide a closer realization of the desired picture.

These first chapters cover the basic tools that Python provides to enhance the scientist's computer experience. How should we proceed further?

A notable omission is that apart from a brief discussion in Section 4.5, the vast subject of data analysis will not be covered. There are three main reasons for this.

1 Recently an add-on module *pandas* has appeared. This uses *numpy* and *matplotlib*

to tackle precisely this issue. It comes with comprehensive documentation, which is described in Section 4.5.

2 One of the authors of *pandas* has written a book, McKinney (2012), which reviews *IPython*, *numpy* and *matplotlib* and goes on to treat *pandas* applications in great detail.

3 I do not work in this area, and so would simply have to paraphrase the sources above.

Instead, I have chosen to concentrate on the modelling activities of scientists. One approach would be to target problems in bioinformatics or cosmology or crystallography or engineering or epidemiology or financial mathematics or . . . etc. Indeed, a whole series of books with a common first half could be produced called "Python for Bioinformatics" etc. A less profligate and potentially more useful approach would be to write a second half applicable to **all** of these fields, and many more. I am relying here on the unity of mathematics. Problems in one field when reduced to a core dimensionless form often look like a similarly reduced problem from another field.

This property can be illustrated by the following example. In population dynamics we might study a single species whose population $N(T)$ depends on time T. Given a plentiful food supply we might expect exponential growth, $dN/dT = kN(T)$, where the growth constant k has dimension 1/time. However, there are usually constraints limiting such growth. A simple model to include these is the "logistic equation"

$$\frac{dN}{dT}(T) = kN(T)\,(N_0 - N(T)) \tag{1.1}$$

which allows for a stable constant population $N(T) = N_0$. The biological background to this equation is discussed in many textbooks, e.g., Murray (2002).

In (homogeneous spherically symmetric) cosmology, the density parameter Ω depends on the scale factor a via

$$\frac{d\Omega}{da} = \frac{(1+3w)}{a}\Omega(1-\Omega), \tag{1.2}$$

where w is usually taken to be a constant.

Now mathematical biology and cosmology do not have a great deal in common, but it is easy to see that (1.1) and (1.2) represent the same equation. Suppose we scale the independent variable T in (1.1) by $t = kN_0T$, which renders the new time coordinate t dimensionless. Similarly, we introduce the dimensionless variable $x = N/N_0$ so that (1.1) becomes the logistic equation

$$\frac{dx}{dt} = x(1-x). \tag{1.3}$$

In a general relativistic theory, there is no reason to prefer any one time coordinate to any other. Thus we may choose a new time coordinate t via $a = e^{t/(1+3w)}$, and then setting $x = \Omega$, we see that (1.2) also reduces to (1.3). Thus the same equations can arise in a number of different fields.[3] In Chapters 7–9, we have, for brevity and simplicity, used minimal equations such as (1.3). If the minimal form for your problem looks something

[3] This example was chosen as a pedagogic example. If the initial value $x(0) = x_0$ is specified, then the exact

like the one being treated in a code snippet, you can of course hack the snippet to handle the original long form for your problem.

Chapter 7 looks at four types of problems involving ordinary differential equations. We start with a very brief introduction to techniques for solving initial value problems and then look at a number of examples, including two classic non-linear problems, the van der Pol oscillator and the Lorenz equations. Next we survey two-point boundary value problems and examine both a linear Sturm–Liouville eigenvalue problem, and an exercise in continuation for the non-linear Bratu problem. Problems involving delay differential equations arise frequently in control theory and in mathematical biology, e.g., the logistic and Mackey–Glass equations, and a discussion of their numerical solution is given in the next section. Finally in this chapter we look briefly at stochastic calculus and stochastic ordinary differential equations. In particular, we consider a simple example closely linked to the Black–Scholes equation of financial mathematics.

There are two other major Python topics relevant to scientists that I would like to introduce here. The first is the incorporation of code written in other languages. There are two aspects of this: (a) the reuse of pre-existing legacy code, usually written in Fortran, (b) if one's code is being slowed down seriously by a few Python functions, as revealed by the profiler, see Section 2.6, how do we recode the offending functions in Fortran or C? The second topic is how can a scientific user make worthwhile use of the object-oriented programming (OOP) features of Python?

Chapter 8 addresses the first topic via an extended example. We look first at how pseudospectral methods can be used to attack a large number of evolution problems governed by partial differential equations, either initial value or initial-boundary value problems. For the sake of brevity, we look only at problems with one time and one spatial dimension. Here, as we explain, problems with periodic spatial dependence can be handled very efficiently using Fourier methods, but for problems which are more general, the use of Chebyshev transforms is desirable. However, in this case there is no satisfactory Python black box available. It turns out that the necessary tools have already been written in legacy Fortran77 code. These are listed in Appendix B, and we show how, with an absolutely minimal knowledge of Fortran77, we can construct extremely fast Python functions to accomplish the required tasks. Our approach relies on the *numpy* `f2py` tool which is included in all of the recommended Python distributions. If you are interested in possibly reusing pre-existing legacy code, it is worthwhile studying this chapter even if the specific example treated there is not the task that you have in mind. See also Section 1.3 for other uses for `f2py`.

One of the most useful features of object-oriented programming (OOP) from the point of view of the scientist is the concept of *classes*. Classes exist in C++ (but not C) and Fortran90 and later (but not Fortran77). However, both implementations are complicated and so are usually shunned by novice programmers. In contrast, Python's implementation is much simpler and more user-friendly, at the cost of omitting some of the more arcane features of other language implementations. We give a very brief introduction to

solution is $x(t) = x_0/[x_0 + (1 - x_0)e^{-t}]$. In the current context, $x_0 \geqslant 0$. If $x_0 \neq 1$, then all solutions tend monotonically towards the constant solution $x = 1$ as t increases. See also Section 7.5.3.

the syntax in Section 3.9. However, in Chapter 9 we present a much more realistic example: the use of *multigrid* to solve elliptic partial differential equations in an arbitrary number of dimensions, although for brevity the example code is for two dimensions. Multigrid is by now a classical problem which is best described recursively, and we devote a few pages to describing it, at least in outline. The pre-existing legacy code is quite complicated because the authors needed to simulate recursion in languages, e.g., Fortran77, which do not support recursion. Of course, we could implement this code using the `f2py` tool outlined in Chapter 8. Instead, we have chosen to use Python classes and recursion to construct a simple clear multigrid code. As a concrete example, we use the sample problem from the corresponding chapter in Press et al. (2007) so that the inquisitive reader can compare the non-recursive and OOP approaches. If you have no particular interest in multigrid, but do have problems involving linked mathematical structures, and such problems arise often in, e.g., bioinformatics, chemistry, epidemiology, solid state physics among others, then you should certainly peruse this final chapter to see how, if you state reasonably mathematically precisely what your problems are, then it is easy to construct Python code to solve them.

1.3 Can Python compete with compiled languages?

The most common criticism of Python and the scientific software packages is that they are far too slow, in comparison with compiled code, when handling complicated realistic problems. The speed-hungry reader might like to look at a recent study[4] of a straightforward "number-crunching" problem treated by various methods. Although the figures given in the final section refer to one particular problem treated on a single processor, they do however give a "ball park" impression of performance. As a benchmark, they use the speed of a fully compiled C++ programme which solves the problem. A Python solution using the technique of Chapter 3, i.e., core Python, is about 700 times slower. Once you use the floating-point module *numpy* and the techniques described in Chapter 4 the code is only about ten times slower, and the Matlab performance is estimated to be similar. However, as the study indicates there are a number of ways to speed up Python to about 80% of the C++ performance. A number of these are very rewarding exercises in computer science.

One in particular though is extremely useful for scientists: the `f2py` tool. This is discussed in detail in Chapter 8 where we show how we can reuse legacy Fortran code. It can also be used to access standard Fortran libraries, e.g., the NAG libraries.[5] Yet another use is to speed up *numpy* code and so improve performance! To see how this works, suppose we have developed a programme such as those outlined in the later sections of the book, which uses a large number of functions, each of which carries out a simple task. The programme works correctly, but is unacceptably slow. Note that getting detailed timing data for Python code is straightforward. Python includes a "profiler" which can be run on the working programme. This outputs a detailed list of the functions

[4] See `http://wiki.scipy.org/PerformancePython`.
[5] See, e.g., `http://www.nag.co.uk/doc/TechRep/pdf/TR1_08.pdf`.

ordered by the time spent executing them. It is very easy to use, and this is described in Section 2.6. Usually, there are one or two functions which take very long times to execute simple algorithms.

This is where `f2py` comes into its own. Because the functions are simple, even beginners can soon create equivalent code in say Fortran77 or Ansi C. Also, because what we are coding is simple, there is no need for the elaborate (and laborious to learn) features of say Fortran95 or C++. Next we encapsulate the code in Python functions using the `f2py` tool, and slot them into the Python programme. With a little experience we can achieve speeds comparable to that of a programme written fully in say Fortran95.

1.4 Limitations of this book

A comprehensive treatment of Python and its various branches would occupy several large volumes and would be out of date before it reached the bookshops. This book is intended to offer the reader a starting point which is sufficient to be able to use the fundamental add-on packages. Once the reader has a little experience with what Python can do, it is time to explore further those areas which interest the reader.

I am conscious of the fact that I have not even mentioned vitally important concepts, e.g., finite-volume methods for hyperbolic problems,[6] parallel programming and real-time graphics to name but a few areas in which Python is very useful. There is a very large army of Python developers working at the frontiers of research, and their endeavours are readily accessed via the internet. Please think of this little book as a transport facility towards the front line.

1.5 Installing Python and add-ons

Users of Matlab and Mathematica are used to a customized *Integrated Development Environment (IDE)*. From the start-up screen, you can investigate code, write, edit and save segments using the built-in editor, and run actual programmes. Since the operating systems Mac OS X and most flavours of Linux include a version of core Python as a matter of course, many computer officers and other seasoned hackers will tell you that it is simple to install the additional packages, and you can be up and coding within the hour, thus ameliorating the difference.

Unfortunately, the pundits are wrong. The Python system being advocated in this book runs the language to its extreme limits, and all of the add-ons must be compatible with each other. Like many others, this author has endured hours of frustration trying to pursue the pundits' policy. Please save your energy, sanity etc., and read Appendix A, which I have quite deliberately targeted at novices, for the obvious reason!

Admittedly, there is an amount, albeit slight and low-level, of hassle involved here. So what's the payoff? Well if you follow the routes suggested in Appendix A, you

[6] The well-regarded Clawpack package `http://depts.washington.edu/clawpack`, which is Fortran-based, has switched from Matlab to Python *matplotlib* for its graphics support.

should end up with a system which works seamlessly. While it is true that the original Python interpreter was not terribly user-friendly, which caused all of the established IDE purveyors to offer a "Python mode", the need which they purported to supply has been overtaken by the enhanced interpreter *IPython*. Indeed, in its latest versions *IPython* hopes to surpass the facilities offered by Matlab, Mathematica and the Python-related features of commercial IDEs. In particular, it allows you to use your favourite editor, not theirs, and to tailor its commands to your needs, as explained in Chapter 2.

2 Getting started with *IPython*

2.1 Generalities

This sounds like software produced by Apple, but it is in fact a Python interpreter on steroids. It has been designed and written by scientists with the aim of offering very fast exploration and construction of code with minimal typing effort, and offering appropriate, even maximal, on-screen help when required. Documentation and much more is available on the website.[1] This chapter is a brief introduction to the essentials of using *IPython*. A more extended discursive treatment can be found in, e.g., Rossant (2013).

IPython comes with three different user interfaces, *terminal*, *qtconsole* and *notebook*. If you have installed the recommended EPD distribution, then all three should be available to you. For the last two, additional software might be needed if you have used a different distribution. You can check what versions are available for your installation by issuing (at the command line) first `ipython` followed by the "return" key (RET). You can escape from *IPython* by typing `exit` followed by RET in the interpreter. Next try out the command `ipython qtconsole` following a similar strategy. Finally, try out the command `ipython notebook`. This should open in a new browser window. To escape from the third, you need CTL-`c` at the command line, plus closing the browser window. What is the difference between them?

Built into *IPython* is the GNU *readline* utility. This means that on the interpreter's current line, the left- and right-arrow keys move the cursor appropriately, and deletion and insertion are straightforward. Just about the only additional commands I ever use are CTL-`a` and CTL-`e` which move the cursor to the start and end of the line, and CTL-`k` which deletes the material to the right of the cursor. The up- and down-arrow keys replicate the previous and next command lines respectively, so that they can be edited. However, there is no way to edit two different lines simultaneously in *terminal* mode.

Both *qtconsole* and *notebook* mode remove this restriction, allowing multiline editing, which is a great convenience. All three modes allow access to the Python scientific graphics facility *matplotlib* explored in Chapter 5. This displays graphics in new windows, which themselves can be edited interactively. The latter two *IPython* modes also allow in-line graphics, i.e., the graphics are displayed in the console or browser windows. This looks neat but, alas, the interactivity has been lost. The *notebook* mode offers facilities which both emulate and transcend the Mathematica "notebook" concept. Besides including in-line graphics, we can insert text before and after the code snippets.

[1] It can be found at `www.ipython.org`.

Both RTF and LaTeX are supported. This is clearly a significant facility for science education. Tantalizingly, the *notebook* mode is still in very active development. You need to check the website cited above for up-to-date details. For this reason, we shall concentrate on *terminal* mode which is central to understanding and using all three modes.

2.2 Tab completion

While using the *IPython* interpreter, *tab completion* is always present. This means that whenever we start typing a Python-related name, we can pause and press the TAB key, to see a list of names valid in this context, which agree with the characters already typed. As an example, suppose we need to type `import matplotlib`.[2] Typing `iTAB` reveals 15 possible completions. By inspection, only one of them has second letter `m`, so that `imTAB` will complete to `import`. Augmenting this to `import mTAB` shows 25 possibilities, and by inspection we see that we need to complete the command by `import matpTAB` to complete the desired line.

That example was somewhat contrived. Here is a more compulsive reason for using tab completion. When developing code, we tend, lazily, to use short names for variables, functions etc. (In early versions of Fortran, we were indeed restricted to six or eight characters, but nowadays the length can be arbitrary.) Short names are not always meaningful ones, and the danger is that if we revisit the code in six months time, the intent of the code may no longer be self evident. By using meaningful names of whatever length is needed, we can avoid this trap. Because of tab completion, the long name only has to be typed once.

2.3 Introspection

IPython has the ability to inspect just about any Python construct, including itself, and to report whatever information its developers have chosen to make available. This facility is called *introspection*. It is accessed by the single character `?`. The easiest way to understand it is to use it, and so you are recommended to fire up the interpreter, e.g., by typing `ipython` followed by a RET on the command line. *IPython* will respond with quite a lengthy header, followed by an input line labelled `In [1]:`.

Now that you are in *IPython*, you can try out introspection by typing `?` (followed by RET) on the input line. *IPython* responds by issuing in pager mode a summary of all of the facilities available. If you exit this, see Section A.2.6, the command `quickref` (hint: use tab completion) gives a more concise version. Very careful study of both documents is highly recommended.

However, scientists are impatient folk, and the purpose of this chapter is to get them up and running with the most useful features. Therefore, we need to type in some Python code, which newcomers will have to take on trust until they have mastered Chapters 3 and 4. For example, please type in (spaces optional)

[2] *matplotlib* is essential to scientific graphics and forms the main topic of Chapter 5.

```
a=3
b=2.7
c=12 + 5j
s='Hello World!'
L=[a, b, c, s, 77.77]
```

The first two lines mean that `a` refers to an integer and `b` to a float (floating-point number). Python uses the engineers' convention whereby $\sqrt{-1} = j$. (Mathematicians would have preferred $\sqrt{-1} = i$.) Then `c` refers to a complex number.[3] Typing each of `a`, `b` and `c` on a line by itself reveals the value of the object to which the identifier refers. Now try `c?` on a line by itself. This confirms that `c` does indeed refer to a complex number, shows how it will be displayed and points out that another way of creating it would have been `c=complex(12,5)`. Next try `c.` immediately followed by TAB. The interpreter will immediately offer three possible completions. What do they mean? Here it is almost obvious, but try `c.real?`. (Using tab completion, you don't need to type `eal`.) This reveals that `c.real` is a float with the value 12, i.e., the real part of the complex number. The newcomer might like to check out `c.imag`. Next try out `c.conjugate?`. (Again only five keystrokes plus RETURN are needed!) This reveals that `c.conjugate` is a function, to be used, e.g., as `cc = c.conjugate()`.

The notation here might seem rather strange to users of say Fortran or C, where `real(c)` or `conjugate(c)` might have been expected. The change in syntax comes about because Python is an object-oriented language. The object here is `12+5j`, referred to as `c`. Then `c.real` is an enquiry as to the value of a component of the object. It changes nothing. However, `c.conjugate()` either alters the object or, as here, creates a new one, and is hence a function. This notation is uniform for all objects, and is discussed in more detail in Section 3.10.

Returning to the snippet, typing `s` by itself on a line prints a string. We can confirm this by the line `s?`, and `s.` followed by a TAB reveals 38 possible completions associated with string objects. The reader should use introspection to reveal what some of them do. Similarly, `L.?` shows that `L` is a list object with nine available completions. Do try a few of them out! As a general rule, the use of introspection and tab completion anywhere in Python code should generate focused documentation. There is a further introspection command `??` which, where appropriate, will reveal the original source code of a function, and examples will be given later in Section 2.6. (The object functions we have so far encountered are built-in, and were not coded in Python!)

2.4 History

If you look at the output from the code snippet in the previous section, you will see that *IPython* employs a history mechanism which is very similar to that in Mathematica notebooks. Input lines are labelled `In[1]`, `In[2]`, . . . , and if input `In[n]` produces any

[3] Note that in the code snippet there is no multiplication sign (*) between 5 and `j`.

output, it is labelled `Out[n]`. As a convenience, the past three input lines are available as `_i`, `_ii` and `_iii`, and the corresponding output lines are available as `_`, `__` and `___`. In practice though, you can insert the content of a previous input line into the current one by navigating using ↑ (or CTL-`p`) and ↓ (or CTL-`n`), and this is perhaps the most common usage. Unusually, but conveniently, history persists. If you close down *IPython* (using `exit`) and then restart it, the history of the previous session is still available via the arrow keys. There are many more elaborate things you can do with the history mechanism, try the command `%history?`.

2.5 Magic commands

The *IPython* window expects to receive valid Python commands. However, it is very convenient to be able to type commands which control either the behaviour of *IPython* or that of the underlying operating system. Such commands, which coexist with Python ones, are called *magic* commands. A very long very detailed description can be found by typing `%magic` in the interpreter, and a compact list of available commands is given by typing `%lsmagic`. (Do not forget tab completion!) You can get very helpful documentation on each command by using introspection. The rest of this section is devoted to explaining "magic".

Let us start by considering system commands. A harmless example is `pwd` which comes from the Unix operating system where it just prints the name of the current directory (**p**rint **w**orking **d**irectory) and exits. There are usually three ways to achieve this in the *IPython* window. You should try out the following snippet. (Do not type the numbers to the left of the box.)

```
1  !pwd
2  %pwd
3  pwd
4  pwd='hoho'
5  pwd
6  %pwd
7  del pwd
8  pwd
9  %automagic?
```

The first two commands are legitimate because no Python identifier starts with either of the symbols used. The first is interpreted as a system command and returns exactly what a regular system command would give. The second returns the same, but enclosed in apostrophes (indicating a string) and prefaced by a `u`. The `u` indicates that the string is encoded in Unicode, which enables a rich variety of outputs. Unicode was mentioned briefly in Section A.2.1. The `%` prefix indicates a magic command, as we now explain. Up to now, no significance has been assigned to the identifier `pwd`. Line 3 reveals the same output as line 2. Thus in general the % prefix is not needed. Line 4 says that `pwd` is now an identifier for the string `'hoho'`. Now typing `pwd` in line 5 reveals the string.

However, line 6 shows that the original magic command still works. In line 7, we delete all reference to the string. Then line 8 produces the same output as line 3. The "magic" is that `pwd` is a system command unless the user has assigned another value to it, while `%pwd` always works. Line 9 explains what is going on in much more detail.

2.6 The magic `%run` command

Because this command is so important, we shall present it via a somewhat more realistic example which, inevitably, requires a little preparation. In a later example, in Section 3.9, we shall consider how to implement arbitrary precision real arithmetic via fractions. There is an issue with fractions, e.g., 3/7 and 24/56 are usually regarded as the same number. Thus there is a problem here, to determine the "highest common factor" of two integers a and b, or as mathematicians are wont to say, their "greatest common divisor" (GCD), which can be used to produce a canonical form for the fraction a/b. (By inspection of factors, the GCD of 24 and 56 is 8, which implies 24/56 = 3/7 and no further reduction of the latter is possible.) Inspection of factors is not easily automated, and a little research, e.g., on the web, reveals *Euclid's algorithm*. To express this concisely, we need a piece of jargon. Suppose a and b are integers. Consider long division of a by b. The remainder is called a mod b, e.g., 13 mod 5 = 3, and 5 mod 13 = 5. Now denote the GCD of a and b by $gcd(a, b)$. Euclid's algorithm is most easily described recursively via

$$gcd(a, 0) = a, \qquad gcd(a, b) = gcd(b, a \bmod b), \quad (b \neq 0). \tag{2.1}$$

Try evaluating by hand $gcd(56, 24)$ according this recipe. It's very fast! It can be shown that the most laborious case arises when a and b are consecutive Fibonacci numbers, and so they would be useful for a test case. The *Fibonacci numbers* F_n are defined recursively via

$$F_0 = 0, \quad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}, \quad n \geqslant 2. \tag{2.2}$$

The sequence begins $0, 1, 1, 2, 3, 5, 8, \ldots$.

How do we implement all of this both efficiently and speedily in Python? We start with the Fibonacci question because it looks to be straightforward.

In order to get started with mastering *IPython*, the novice reader is asked to take on trust the next two code snippets. Partial explanation is offered here, but all of the features will be explained more thoroughly in Chapter 3.

Consider typing, using your chosen text editor, the following snippet into a file called `fib.py` in a convenient folder/directory. Unlike the C and Fortran families, *Python* deliberately eschews the use of parentheses () and braces {}. In order to do this, it insists on code being consistently indented. Any code line which ends with a colon (:) requires a new indented block, the unofficial standard being four spaces long. The end of the code block is shown by a return to the former level of indentation, see, e.g., lines 10–12 in the snippet below. *IPython* and any Python-aware editor will recognize the ending colon and do this automatically.[4]. See also section 3.1.

```
1  # File: fib.py Fibonacci numbers
2
3  """ Module fib: Fibonacci numbers.
4      Contains one function fib(n).
5  """
6
7  def fib(n):
8      """ Returns n’th Fibonacci number. """
9      a,b=0,1
10     for i in range(n):
11         a,b=b,a+b
12     return a
13
14 #####################################################
15 if __name__ == "__main__":
16     for i in range(1001):
17         print "fib(",i,") = ",fib(i)
```

The details of Python syntax are explained in Chapter 3. For the time being, note that lines 3–5 define a *docstring*, whose purpose will be explained shortly. Lines 7–12 define a Python function. Note the point made above that every colon (:) demands an indentation. Line 7 is the function declaration. Line 8 is the function *docstring*, again soon to be explained. Line 9 introduces identifiers `a` and `b`, which are local to this function, and refer initially to the values 0 and 1 respectively. Next examine line 11, ignoring for the moment its indentation. Here `a` is set to refer to the value that `b` originally referred to. Simultaneously, `b` is set to refer to the sum of values originally referred to by `a` and `b`. Clearly, lines 9 and 11 replicate the calculations implicit in (2.2). Now line 10 introduces a *for-loop* or *do-loop*, explained in Section 3.7.1, which extends over line 11. Here **`range`**`(n)` generates a dummy list with n elements, $[0, 1, \ldots, n-1]$, and so line 11 is executed precisely n times. Finally, line 12 exits the function with the return value set to that referred to finally by `a`. Next type one or more blank lines to remove the indentation.

Naturally, we need to provide a test suite to demonstrate that this function behaves as intended. Line 14 is simply a comment. Line 15 will be explained soon. (When typing it, note that there are four pairs of underscores.) Because it is an *if statement* terminated by a colon, all subsequent lines need to be indented. We have already seen the idea behind line 16. We repeat line 17 precisely 1001 times with $i = 0, 1, 2, \ldots, 1000$. It prints a string with four characters, the value of `i`, another string with four characters, and the value of `fib(i)`.

Assuming that you have created and saved this file, make sure that *IPython* is open

[4] Note that to un-indent in *IPython*, you need to type an empty line.

in the same directory. Then within the *IPython* window, issue the (magic) command `run fib`. If your creation was syntactically correct, the response will be 1001 lines of Fibonacci values. If not, the response will indicate the first error encountered. Returning to the editor window, correct and save the source. Then try the `run fib` command again. (Beginners must expect to go through this cycle several times, but it is quick!) Once the programme is verified we can ask how fast is it? Run the programme again but with the enhanced command `run -t fib` and *IPython* will produce timing data. On my machine, the "User time" is 0.05 s., but the "Wall time" is 4.5 s. Clearly, the discrepancy reflects the very large number of characters printed to the screen. To verify this, modify the snippet as follows. Comment out the print statement in line 17 by inserting a *hash* (#) character at the start of the line. Add a new line 18: `fib(i)`, being careful to get the indentation correct. (This evaluates the function, but does nothing with the value.) Now run the program again. On my machine it takes 0.03 s., showing that `fib(i)` is fast, but printing is not. (Don't forget to comment out line 18, and uncomment in line 17!)

We still need to explain the docstrings in lines 3–5 and 8, and the weird line 15. Close down *IPython* (use `exit`) and then reopen a fresh version. Type the single line `import fib`, which reflects the core of the filename. The tail `.py` is not needed. We have imported an object `fib`. What is it? Introspection suggests the command `fib?`, and *IPython*'s response is to print the *docstring* from lines 3–5 of the snippet. This suggests that we find out more about the function `fib.fib`, so try `fib.fib?`, and we are returned the *docstring* from line 8. The purpose of *docstrings*, which are messages enclosed in pairs of triple double-quotes, is to offer online documentation to other users and, just as importantly, you in a few days time! However, introspection has a further trick up its sleeve. Try `fib.fib??` and you will receive a listing of the source code for this function!

You should have noticed that `import fib` did not list the first 1001 Fibonacci numbers. Had we instead, in a separate session, issued the command `run fib`, they would have been printed! Line 15 of the snippet detects whether the file `fib.py` is being imported or run, and responds accordingly without or with the test suite. How it does this is explained in Section 3.4.

Now we return to our original task, which was to implement the `gcd` function implicit in equation (2.1). Once we recognize that (i) Python has no problem with recursion, and (ii) a mod b is implemented as `a%b`, then a minimal thought solution suggests itself, as in the following snippet. (Ignore for the time being lines 14–18.)

```
1  # File gcd.py Implementing the GCD Euclidean algorithm.
2
3  """ Module gcd: contains two implementations of the Euclid
4     GCD algorithm, gcdr and gcd.
5  """
6
7  def gcdr(a,b):
8     """ Euclidean algorithm, recursive vers., returns GCD. """
9     if b==0:
```

```
10          return a
11      else:
12          return gcdr(b,a%b)
13
14  def gcd(a,b):
15      """ Euclidean algorithm, non-recursive vers., returns GCD. """
16      while b:
17          a,b=b,a%b
18      return a
19
20  ##########################################################
21  if __name__ == "__main__":
22      import fib
23
24      for i in range(984):
25          print i, ' ', gcdr(fib.fib(i),fib.fib(i+1))
```

The only real novelty in this snippet is the **import** fib statement in line 22, and we have already discussed its effect above. The number of times the loop in lines 24 and 25 is executed is crucial. As printed, this snippet should run in a fraction of a second. Now change the parameter 984 in line 24 to 985, save the file, and apply `run gcd` again. You should find that the output appears to be in an infinite loop, but be patient. Eventually, the process will terminate with an error statement that the maximum recursion depth has been exceeded. While Python allows recursion, there is a limit on the number of self-calls that can be made.

This limitation may or may not be a problem for you. But it is worth a few moments thought to decide whether we could implement Euclid's algorithm (2.1) without using recursion. I offer one possible solution in the function `gcd` implemented in lines 14–18 of the snippet. Lines 16 and 17 define a *while loop*, note the colon terminating line 16. Between **while** and the colon, Python expects an expression which evaluates to one of the Boolean values `True` or `False`. As long as `True` is found, the loop executes line 17, and then retests the expression. If the test produces `False`, then the loop terminates and control passes to the next statement, line 18. In the expected context, `b` will always be an integer, so how can an integer take a Boolean value? The answer is remarkably simple. The integer value zero is always coerced to `False`, but all non-zero values coerce to `True`. Thus the loop terminates when `b` becomes zero, and then the function returns the value `a`. This is the first clause of (2.1). The transformation in line 17 is the second clause, and so this function implements the algorithm. It is shorter than the recursive function, can be called an arbitrary number of times and, as we shall see, runs faster.

So was the expenditure of thought worthwhile. Using the `run` command, we can obtain revealing statistics. First, edit the snippet to make 980 loops with the `gcdr` function, and save it. Now invoke `run -t gcd` to obtain the time spent. On my machine, the "User time" is 0.31s. Yours will vary, but it is relative timings that matter. The "Wall time" reflects the display overhead and is not relevant here. Next invoke `run -p gcd`, which invokes the Python profiler. Although you would need to read the documentation to understand every facet of the resulting display, a little scientific intuition can be very useful. This shows that there were 980 direct calls (as expected) of the function `gcdr`, within a total of 480,691 actual calls. The actual time spent within this function was 0.282s. Next there were 1960 calls (as expected) of the function `fib`, and the time expended was 0.087s. Note that these timings cannot be compared with the older `run -t gcd` ones, because the newer ones include the profiler overhead which is significant. However, we can conclude that about 75% of the time was spent in the function `gcdr`.

Next we need to repeat the exercise for the `gcd` function. Amend line 25 of the snippet to replace `gcdr` by `gcd` and resave the file. Now `run -t gcd` to get a "User time" of 0.19s. The other command `run -p gcd` reveals that the 1960 calls of function `fib` took 0.087s. However, the function `gcd` was only called 980 times (as expected) which occupied 0.093s. Thus `gcd` occupied about 50% of the time taken. Very approximately, these relative timings factor out the profiler overhead. Now 75% of the 0.31s. timing for the recursive version is 0.23s., while 50% of the 0.19s. time for the non-recursive version is 0.095s. Thus the expenditure of thought has produced a shortened code which runs in 40% of the time of the "thoughtless code"! There is a moral here.

3 A short Python tutorial

Although Python is a small language it is a very rich one. It is very tempting, when writing a textbook, to spell out all of the ramifications, concept by concept. The obvious example is the introductory tutorial from the originator of Python, Guido van Rossum. This is available in electronic form as the tutorial in your Python documentation or on-line[1] or as hard copy, van Rossum and Drake Jr. (2011). It is relatively terse at 150 printed pages, and does not mention *numpy*. My favourite textbook, Lutz (2009) runs to over 1200 pages, a marathon learning curve, and only mentions *numpy* in passing. It is excellent at explaining the features in detail, but is too expansive for a first course in Python. A similar criticism can be applied to two books with a more scientific orientation, Langtangen (2008) and Langtangen (2009), both around 700 pages with a significant overlap between them. I recommend these various books among many others for reference, but not for learning the language.

Very few people would learn a foreign language by first mastering a grammar textbook and then memorizing a dictionary. Most start with a few rudiments of grammar and a tiny vocabulary. Then by practice they gradually extend their range of constructs and working vocabulary. This allows them to comprehend and speak the language very quickly, and it is the approach to learning Python that is being adopted here. The disadvantage is that the grammar and vocabulary are diffused throughout the learning process, but this is ameliorated by the existence of textbooks, such as those cited in the first paragraph.

3.1 Typing Python

Although the narrative can simply be read, it is extremely helpful to have the *IPython* interpreter to hand. For longer code snippets, you may choose to use an editor as well, so that you can save the code. Your choices are described in Section A.2.1 of Appendix A. Some of the code snippets shown here have numbered lines. It is a good idea to type in these (omitting of course the line numbers to the left of the box) and terminating lines with the "return" (RET) key. You are strongly encouraged to try out your own experiments in the interpreter after saving each code snippet.

Every programming language includes *blocks* of code, which consist of one or more lines of code forming a syntactic whole. Python uses rather fewer parentheses () and

[1] It is available at `http://docs.python.org/2/tutorial`.

braces {} than other languages, and instead uses indentation as a tool for formatting blocks. After any line ending in a colon : a block is required, and it is differentiated from the surrounding code by being consistently indented. Although the amount is not specified, the unofficial standard is four spaces. *IPython* and any Python-aware text editor will do this automatically. To revert to the original indentation level, use the RET key to enter a totally empty line. Removing braces improves readability, but the disadvantage is that each line in a block must have the same indentation as the one before, or a syntax error will occur.

Python allows two forms of *comments*. A hash symbol # indicates that the rest of the current line is a comment, or more precisely a "tweet". A "documentation string" or *docstring* can run over many lines and include any printable character. It is delimited by a pair of triple quotes, e.g.

```
""" This is a very short docstring. """
```

For completeness, we note that we may place several statements on the same line provided we separate them with semicolons, but we should think about readability. Long statements can be broken up with the continuation symbol \. More usefully, if a statement includes a pair of brackets (), we can split the line at any point between them without the need for the continuation symbol. Here are simple examples.

```
a=4; b=5.5; c=1.5+2j; d='a'
e=6.0*a-b*b+\
       c**(a+b+c)
f=6.0*a-b*b+c**(
       a+b+c)
```

3.2 Objects and identifiers

Python deals exclusively with objects and identifiers. An *object* may be thought of as a region of computer memory containing both some data and information associated with those data. For a simple object, this information consists of its *type*, and its *identity*,[2] i.e., the location in memory, which is of course machine-dependent. The identity is therefore of no interest for most users. They need a machine-independent method for accessing objects. This is provided by an *identifier*, a label which can be attached to objects. It is made up of one or more characters. The first must be a letter or underscore, and any subsequent characters must be digits, letters or underscores. Identifiers are case-sensitive, `x` and `X` are different identifiers. (Identifiers which have leading and/or trailing underscores have specialized uses, and should be avoided by the beginner.) We must avoid using predefined words, e.g., **`list`**, and should always try to use meaningful identifiers. However, the choice between say `xnew`, `x_new` and `xNew` is a matter of taste. Consider the following code, which should be typed in the interpreter window.

[2] An unfortunate choice of name, not to be confused with the about-to-be-defined identifiers.

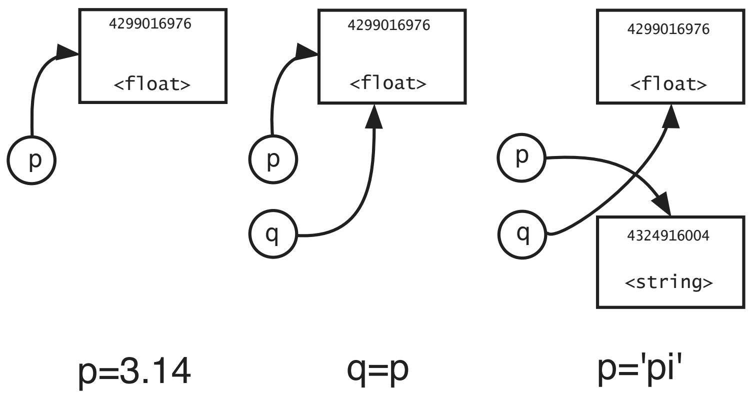

Figure 3.1 A schematic representation of assignments in Python. After the first command `p=3.14`, the float object `3.14` is created and identifier `p` is assigned to it. Here the object is depicted by its *identity*, a large number, its address in the memory of my computer (highly machine-dependent) where the data are stored, and the *type*. The second command `q=p` assigns identifier `q` to the same object. The third command `p='pi'` assigns `p` to a new "string" object, leaving `q` pointing to the original float object.

```
1  p=3.14
2  p
3  q=p
4  p='pi'
5  p
6  q
```

Note that we never declared the *type* of the object referred to by the identifier `p`. We would have had to declare `p` to be of type "double" in C and "real*8" in Fortran. This is no accident or oversight. A fundamental feature of Python is that the "type" belongs to the object, not to the identifier.[3]

Next in line 3, we set `q=p`. The right-hand side is replaced by whatever object `p` pointed to, and `q` is a new identifier which points to this object, see Figure 3.1. No equality of identifiers `q` and `p` is implied here! Notice that in line 4, we reassign the identifier `p` to a "string" object. However, the original float object is still pointed to by the identifier `q`, see Figure 3.1, and this is confirmed by the output of lines 5 and 6. Suppose we were to reassign the identifier `q`. Then, unless in the interim another identifier had been assigned to `q`, the original "float" object would have no identifier assigned to it and so becomes inaccessible to the programmer. Python will detect this automatically and silently free up the computer memory, a process known as *garbage collection*.

[3] The curious can find the type of an object with identifier `p` with the command **`type`**`(p)` and its identity with **`id`**`(p)`.

Because of its importance in what follows we emphasize the point that the basic building block in Python is, in pseudocode,

```
<identifier>=<object>
```

which will appear over and over again. As we have already stated earlier, the type of an object "belongs" to the object and not to any identifier assigned to it.

Since we have introduced a "float" , albeit informally, we turn next to a simple class of objects.

3.3 Numbers

Python contains three simple types of number objects, and we introduce a fourth, not so simple, one.

3.3.1 Integers

Python refers to integers as **`int`**s . Although early versions only supported integers in the range $[-2^{31}, 2^{31}-1]$, in recent versions the range is considerably larger and is limited only by the availability of memory.

The usual operations of addition (+), subtraction (−) and multiplication ($*$) are of course available. There is a slight problem with division, for if p and q are integers, p/q may not be an integer. We may assume without loss of generality that $q > 0$ in which case there exist unique integers m and n with

$$p = mq + n, \qquad \text{where} \quad 0 \leqslant n < q.$$

Then *integer division* in Python is defined by `p//q`, which returns `m`. The *remainder* n is available as `p%q`. *Exponentiation* p^q is also available as `p**q`, and can produce a real number if $q < 0$.

3.3.2 Real numbers

Floating-point numbers are available as **`float`**s . In most installations, the default will be approximately 16 digits of precision for floats in the range $(10^{-308}, 10^{308})$. These correspond to *doubles* in the C-family of languages and *real*8* in the Fortran family. The notation for float constants is standard, e.g.

```
-3.14, -314e-2, -314.0e-2, -0.00314E3
```

all represent the same float.

The usual arithmetic rules for addition, subtraction, multiplication, division and exponentiation are available for floats. For the first three, mixed mode operations are implemented seamlessly, e.g, if addition of an int and a float is requested, then the int is automatically *widened* to be a float. The same applies for division if one operand is an int and the other is a float. However, if both are ints, e.g., ±`1/5`, what is the result? Earlier versions (< 3.0) of Python adopt integer division, `1/5=0` and `-1/5=-1`, while versions ⩾ 3.0 use real division `1/5=0.2`, `-1/5=-0.2`. This is a potential pitfall which is easily avoided. Either use integer division `//` or widen one of the operands to ensure an unambiguous result.

Widening of an int to a float is available explicitly, e.g., `float(4)` will return either `4.` or `4.0`. *Narrowing* of a float to an int is defined by the following algorithm. If x is real and positive, then there exist an integer m and a real y such that

$$x = m + y \qquad \text{where} \quad 0 \leqslant y < 1.0.$$

In Python, this narrowing is implemented via `int(x)`, which returns `m`. If x is negative, then `int(x)=-int(-x)`, succinctly described as "truncation towards zero", e.g., `int(1.4)=1` and `int(-1.4)=-1`.

In programming languages, we expect a wide range of familiar mathematical functions to be available. Fortran builds in the most commonly used ones, but in the C-family of languages, we need to import them with a statement such as `#include math.h` at the top of the programme. Python also requires a module to be imported, and as an example we consider the `math` module. (Modules are defined in Section 3.4.) Suppose first that we do not know what the `math` module contains. The following snippet first loads the module and then lists the identifiers of its contents.

```
import math
dir(math)
```

To find out more about the individual objects, we can either consult the written documentation or use the built-in help, e.g., in *IPython*

```
math.atan2?              # or help(math.atan2)
```

If one is already familiar with the contents, then a quick-and-dirty-fix is to replace the **`import`** command above by

```
 from math import *
```

anywhere in the code before invoking the functions. Then the function mentioned above is available as `atan2(y,x)` rather than `math.atan2(y,x)`, which at first sight looks appealing, but see Section 3.4. Note that unlike C, the **`import`** command can occur anywhere in the programme before its contents are needed.

3.3.3 Boolean numbers

For completeness, we include here Boolean numbers or `bool` which are a subset of the ints with two possible values `True` and `False`, roughly equivalent to 1 (one) and 0 (zero).

Suppose that `box` and `boy` refer to bools. Then expressions such as "`not box`", "`box and boy`" and "`box or boy`" take on their usual meanings.

The standard *equality* operators are defined for ints and floats `x`, `y`, e.g., `x==y` (equality), `x!=y` (inequality). As a simple exercise, to remind you of the limitations of Python *floats*, guess the result of the following line, next type it and then explain the result.

```
math.tan(math.pi/4.0)==1.0
```

However, for the comparison operators "`x>y`", "`x>=y`", "`x<y`" and "`x<=y`", widening takes place if necessary. Unusually, but conveniently, *chaining* of comparison operators is allowed, e.g., "`0<=x<1<y>z`" is equivalent to

```
 (0<=x) and (x<1) and (1<y) and (y>z)
```

Note that there is no comparison between `x` and `z` in this example.

3.3.4 Complex numbers

We have introduced three classes of numbers which form the simplest types of Python object. These are the basis for classes of numbers that are more complicated. For example *rational* numbers can be implemented in terms of pairs of integers. For our purposes, a probably more useful class is that of complex numbers which is implemented in terms of a pair of real numbers. Whereas mathematicians usually denote $\sqrt{-1}$ by i, many engineers prefer j, and Python has adopted the latter approach. Thus a Python `complex` number can be defined explicitly by, e.g., `c=1.5-0.4j`. Note carefully the syntax: the `j` (or equivalently `J`) follows the float with no intervening "$*$". Alternatively, a pair of floats `a` and `b` can be widened to a complex number via `c=complex(a,b)`. We can narrow a complex; e.g., with `c` as in the last sentence, `c.real` returns `a` and `c.imag` returns `b`. Another useful object is `c.conjugate()`, which returns the complex conjugate of `c`. The syntax of complex attributes will be explained in Section 3.10.

The five basic arithmetic operations work for Python complex numbers, and in mixed mode widening is done automatically. The library of mathematical functions for complex arguments is available. We need to import from `cmath` rather than `math`. However, for obvious reasons, the comparison operations involving ordering described above, are not defined for complex numbers, although the equality and inequality operators are available.

You have now seen enough of Python to use the interpreter as a sophisticated five function calculator, and you are urged to try out a few examples of your own.

3.4 Namespaces and modules

While Python is running, it needs to keep a list of those identifiers which have been assigned to objects. This list is called a *namespace*, and as a Python object it too has an identifier. For example, while working in the interpreter, the namespace has the unmemorable name `__main__`.

One of the strengths of Python is its ability to include files of objects, functions etc., written either by you or someone else. To enable this inclusion, suppose you have created a file containing objects, e.g., `obj1`, `obj2` that you want to reuse. The file should be saved as, e.g., `foo.py`, where the `.py` ending is mandatory. (Note that with most text editors you need this ending for the editor to realize that it is dealing with Python code.) This file is then called a *module*.

This module can be imported into subsequent sessions via

```
import foo
```

(When the module is first imported, it is compiled into bytecode and written back to storage as a file `foo.pyc`. On subsequent imports, the interpreter loads this precompiled bytecode unless the modification date of `foo.py` is more recent, in which case a new version of the file `foo.pyc` is generated automatically.)

One effect of this import is to make the namespace of the module available as `foo`. Then the objects from `foo` are available with, e.g., identifiers `foo.obj1` and `foo.obj2`. If you are absolutely sure that `obj1` and `obj2` will not clash with identifiers in the current namespace, you can import them via

```
from foo import obj1, obj2
```

and then refer to them as `obj1` etc.

The “quick-and-dirty-fix” of Section 3.3.2 is equivalent to the line

```
from foo import *
```

which imports *everything* from the module `foo`’s namespace. If an identifier `obj1` already existed, it will be overwritten by this import process. For example, suppose we had an identifier `gamma` referring to a float. Then

```
from math import *
```

overwrites this and `gamma` now refers to the (real) gamma-function. A subsequent

```
from cmath import *
```

overwrites `gamma` with the (complex) gamma-function! Note too that import statements can appear anywhere in Python code, and so chaos is lurking if we use this option.

Except for quick, exploratory work in the interpreter, it is far better to modify the import statements as, e.g.,

```
import math as re
import cmath as co
```

so that in the example above `gamma`, `re.gamma` and `co.gamma` are all available.

We now have sufficient background to explain the mysterious code line

```
if __name__ == "__main__"
```

which occurred in both of the snippets in Section 2.6. The first instance occurred in a file `fib.py`. Now if we import this module into the interpreter, its name is `fib` and not `__main__` and so the lines after this code line will be ignored. However, when developing the functions in the module, it is normal to make the module available directly, usually via the `%run` command. Then, as explained at the start of this section, the contents are read into the `__main__` namespace. Then the **`if`** condition of the code line is satisfied and the subsequent lines will be executed. In practice, this is incredibly convenient. While developing a suite of objects, e.g., functions, we can keep the ancillary test functions nearby. In production mode via **`import`**, these ancillary functions are effectively "commented out".

3.5 Container objects

The usefulness of computers is based in large part on their ability to carry out repetitive tasks very quickly. Most programming languages therefore provide container objects, often called arrays, which can store large numbers of objects of the same type, and retrieve them via an indexing mechanism. Mathematical vectors would correspond to one-dimensional arrays, matrices to two-dimensional arrays etc. It may come as a surprise to find that the Python core language has no array concept. Instead, it has container objects which are much more general, *lists*, *tuples*, *strings* and *dictionaries*. It will soon become clear that we can simulate an array object via a *list*, and this is how numerical work in Python used to be done. Because of the generality of *lists*, such simulations took a great deal longer than equivalent constructions in Fortran or C, and this gave Python a deservedly poor reputation for its slowness in numerical work. Developers produced various schemes to alleviate this, and they have now standardized on the *numpy* add-on module to be described in Chapter 4. Arrays in *numpy* have much of the versatility of Python *lists*, but are implemented behind the scenes as arrays in C significantly reducing, but not quite eliminating, the speed penalty. However, in this section we describe the core container objects in sufficient detail for much scientific work. Numerical arrays are deferred to the next chapter, but the reader particularly interested in numerics will need to understand the content of this section, because the ideas carry forward into the next chapter.

3.5.1 *Lists*

Consider typing the code snippet

```
[1,4.0,'a']
u=[1,4.0,'a']
v=[3.14,2.78,u,42]
v
len(v)
len?                # or help(len)
```

```
7  v*2
8  v+u
9  v.append('foo')
10 v
```

Line 1 is our first instance of a Python *list*, an ordered sequence of Python objects separated by commas and surrounded by square brackets. It is itself a Python object, and can be assigned to a Python identifier, as in line 2. Unlike arrays, there is no requirement that the elements of a *list* be all of the same type. In lines 3 and 4, we see that in creating the *list* an identifier is replaced by the object it refers to, e.g., one *list* can be an element in another. The beginner should consult Figure 3.1 again. It is the object, not the identifier, which matters. In line 5, we invoke a Python function which returns the length of the *list*, here 4. (Python functions will be discussed in Section 3.8. In the meantime we can find what **`len`** does by typing the line **`len`**? in *IPython*.) We can replicate *lists* by constructions like line 7, and concatenate *lists* as in line 8. We can append items to the ends of *lists* as in line 9. Here `v.append()` is another useful function. You should try `v.append?` or **`help`**`(v.append)` to see a description of it. Incidentally, **`list.`** followed by TAB completion or **`help`**`(`**`list`**`)` will give a catalogue of functions intrinsic to *lists*. Ignore those starting with double underscores and look particularly at those at the end of the catalogue, since these are likely to be the ones most useful for beginners.

3.5.2 *List* indexing

We can access elements of `u` by *indexing*, `u[i]` where $i \in [0, len(u))$ is an integer. Note that indexing starts with `u[0]` and ends with `u[`**`len`**`(u)-1]`. So far, this is very similar to what is available for arrays in, e.g., C or Fortran. However, a Python *list* such as `u` "knows" its length, and so we could also index the elements, in reverse order, by `u[`**`len`**`(u)-k]` where $k \in (0, len(u)]$, which Python abbreviates to `u[-k]`. This turns out to be very convenient. For example, not only is the first element of any *list* `w` referred to by `w[0]`, but the last element is `w[-1]`. The middle line of Figure 3.2 shows both sets of indices for a *list* of length 8. Using the code snippet above, you might like to guess the objects corresponding to `v[1]` and `v[-3]`, and perhaps use the interpreter to check your answers.

At first sight, this may appear to be a trivial enhancement, but it becomes very powerful when coupled to the concepts of slicing and mutability, which we address next. Therefore, it is important to make sure you understand clearly what negative indices represent.

3.5.3 *List* slicing

Given a *list* `u`, we can form more *lists* by the operation of *slicing*. The simplest form of a slice is `u[start:end]`, which is a *list* of length `end-start`, as shown in Figure 3.2. If the slice occurs on the right-hand side of an assignation, then a **new** *list* is created. For example, `su=u[2:6]` generates a new *list* with four elements, where `su[0]` is initialized with `u[2]`. If the slice occurs on the left, no new *list* is generated. Instead, it allows us to change a block of values in an existing *list*. There are important new constructs here which may well be unfamiliar to C and Fortran users.

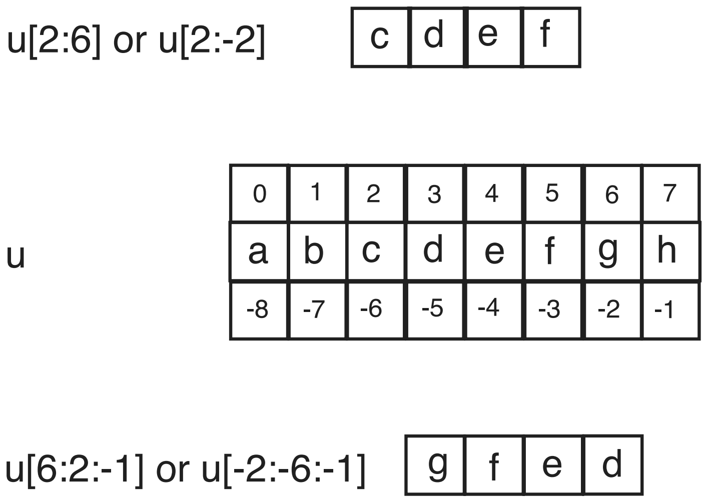

Figure 3.2 Indices and slicing for a *list* u of length 8. The middle line shows the contents of u and the two sets of indices by which the elements can be addressed. The top line shows the contents of a slice of length 4 with conventional ordering. The bottom line shows another slice with reversed ordering.

Consider the simple example below which illustrates the possibilities, and carry out further experiments of your own.

```
u=[0,1,2,3,4,5,6,7]
su=u[2:4]
su
su[0]=17
su
u
u[2:4]=[3.14,'a']
u
```

If `start` is zero, it may be omitted, e.g., `u[ :-1]` is a copy of `u` with the last element omitted. The same applies at the other end, `u[1: ]` is a copy with the first element omitted and `u[:]` is a copy of `u`. In these examples, we are assuming that the slice occurs on the right-hand side of an assignation. The more general form of *slicing* is `su = u[start:end:step]`. Then `su` contains the elements `u[start]`, `u[start+step]`, `u[start+2*step]`, ..., as long as the index is less than `end`. Thus with the list `u` chosen as in the example above we would have `u[2:-1:2]=[2,4,6]`. A particularly useful choice is `step=-1`, which allows traversal of the *list* in the reverse direction. See Figure 3.2 for an example.

3.5.4 *List* mutability

For any container object `u`, it may be possible to modify an element or a slice, without any apparent operation applied to the object's identifier. Such objects are said to be *mutable*. As an example, consider a politician's promises. In particular, *lists* are mutable. There is a trap here for the unwary. Consider the code

```
1  a=4
2  b=a
3  b='foo'
4  a
5  b
6  u=[0,1,4,9,16]
7  v=u
8  v[2]='foo'
9  v
10 u
```

The first five lines should be comprehensible: `a` is assigned to the object `4`; so is `b`. Then `b` is assigned to the object `'foo'`, and this does not change `a`. In line 6, `u` is assigned to a *list* object and so is `v` in line 7. Because *lists* are mutable, we may change the second element of the *list* object in line 8. Line 9 shows the effect. But `u` is pointing to the same object (see Figure 3.1) and it too shows the change in line 10. While the logic is clear, this may not be what was intended, for `u` was never changed explicitly.

It is important to remember the assertion made above: a slice of a *list* is always a **new** object, even if the dimensions of the slice and the original *list* agree. Therefore, compare lines 6–10 of the code snippet above with

```
1  u=[0,1,4,9,16]
2  v=u[ : ]
3  v[2]='foo'
4  v
5  u
```

Now line 2 makes a slice object, which is a *copy*[4] of the object defined in line 1. Changes to the `v`-*list* do not alter the `u`-*list* and vice-versa.

Lists are very versatile objects, and there exist many Python functions which can generate them. We shall discuss *list* generation at many points in the rest of this book.

[4] For the sake of completeness, we should note that this is a *shallow copy*. If `u` contains an element which is mutable, e.g., another *list* `w`, the corresponding element of `v` still accesses the original `w`. To guard against this, we need a *deep copy* to obtain a distinct but exact copy of both `u` and its current contents, e.g., `v=u.copy()`.

3.5.5 *Tuples*

The next container to be discussed is the *tuple*. Syntactically it differs from a *list* only by using () instead of [] as delimiters, and indexing and slicing work as for *lists*. However, there is a fundamental difference. We cannot change the values of its elements, a *tuple* is *immutable*. At first sight, the *tuple* would appear to be entirely redundant. Why not use a *list* instead? The rigidity of a *tuple* however has an advantage. We can use a *tuple* where a scalar quantity is expected, and in many cases we can drop the brackets () when there is no ambiguity, and indeed this is the commonest way of utilizing *tuples*. Consider the snippet below, where we have written a *tuple* assignation in two different ways.

```
(a,b,c,d)=(4,5.0,1.5+2j,'a')
a,b,c,d = 4,5.0,1.5+2j,'a'
```

The second line shows how we can make multiple scalar assignments with a single assignation operator. This becomes extremely useful in the common case where we need to swap two objects, or equivalently two identifiers, say `a` and `L1`. The conventional way to do this is

```
temp=a
a=L1
L1=temp
```

This would work in any language, assuming `temp`, `a` and `L1` all refer to the same type. However

```
a,L1 = L1,a
```

does the same job in Python, is clearer, more concise and works for arbitrary types.

Another use, perhaps the most important one for *tuples*, is the ability to pass a variable number of arguments to a function, as discussed in Section 3.8.4. Finally, we note a feature of the notation which often confuses the beginner. We sometimes need a *tuple* with only one element, say `foo`. The construction `(foo)` strips the parentheses and leaves just the element. The correct *tuple* construction is `(foo,)`.

3.5.6 *Strings*

Although we have already seen *string*s in passing, we note that Python regards them as immutable container objects for alphanumeric characters. There is no comma separator between items. The delimeters can be either single quotes or double quotes, but not a mixture. The unused delimiter can occur within the *string*, e.g.

```
s1="It's time to go"
s2=' "Bravo!" he shouted.'
```

Indexing and slicing work in the same way as for *lists*. *Strings* will turn out to be very useful in producing formatted output from the print function (see Section 3.8.6).

3.5.7 *Dictionaries*

As we have seen, a *list* object is an ordered collection of objects, A *dictionary* object is an unordered collection. Instead of accessing the elements by virtue of their position, we have to assign a keyword, an immutable object, usually a *string*, which identifies the element. Thus a *dictionary* is a collection of pairs of objects, where the first item in the pair is a *key* to the second. A key-object pair is written as `key:object`. We fetch items via keys rather than position. The *dictionary* delimeters are the braces { }. Here is a simple example that illustrates the basics.

```
1 empty={}
2 parms={'alpha':1.3,'beta':2.74}
3 #parms=dict(alpha=1.3,beta=2.74)
4 parms['gamma']=0.999
5 parms
```

The commented-out line 3 is an alternative to line 2. This illustrates the main numerical usage of *dictionaries*, to pass around an unknown and possibly variable number of parameters. Another important use will be keyword arguments in functions (see Section 3.8.5). A more sophisticated application is discussed in Section 7.5.2.

Dictionaries are extremely versatile, have many other properties and have plenty of applications in contexts which are more general, see the textbooks for details.

3.6 Python *if* statements

Normally, Python executes statements in the order that they are written. The **`if`** statement is the simplest way to modify this behaviour, and exists in every programming language. In its simplest form the Python syntax is

```
if < Boolean expression >:
    <block 1>
<block 2>
```

Here the expression must produce a `True` or `False` result. If `True`, then block1 is executed, followed by block 2, the rest of the programme; if the expression is `False`, then only block 2 is executed. Note that the **`if`** statement ends with a colon : indicating that a block must follow. The absence of delimeters such as braces makes it much easier to follow the logic, but the price required is careful attention to the indentation. Any Python-aware editor should take care of this automatically.

Here is a very simple example that also illustrates *strings* in action.

```
x=0.47
if 0 < x < 1:
    print "x lies between zero and one."
y=4
```

A simple generalization is

```
if < Boolean expression >:
   <block 1>
else:
   <block 2>
<block 3>
```

which executes either block 1 or block 2 followed by block 3.

We can chain **if** statements and there is a convenient abbreviation **elif**. Note that all of the logical blocks must be present. If nothing is required in a particular block, it should consist of the single command **pass**, e.g.

```
if < Boolean expression 1>:
   <block 1>
elif <Boolean expression 2>:
   <block 2>
elif <Boolean expression 3>:
   pass
else:
   <block 4>
<block 5>
```

Here the truth of Boolean expression 1 leads to the execution of blocks 1 and 5. If expression 1 is false and expression 2 is true, we get blocks 2 and 5. However if expressions 1 and 2 are false but 3 is true, only block 5 is executed, but if all three expressions are false, both blocks 4 and 5 are executed.

A situation which arises quite often is a construction with terse expressions, e.g.

```
if x>=0:
   y=f
else:
   y=g
```

As in the C-family of languages, there is an abbreviated form. The snippet above can be shortened in Python with no loss of clarity to

```
y=f if x>=0 else g
```

3.7 Loop constructs

Computers are capable of repeating sequences of actions with great speed, and Python has two loop constructs, **for** and **while** loops.

3.7.1 The Python *for* loop

This, the simplest loop construct, exists in all programming languages, as `for` loops in the C-family and as `do` loops in Fortran. The Python *loop* construct is a generalized, sophisticated evolution of these. Its simplest form is

```
for <iterator> in <iterable>:
    <block>
```

Here `<iterable>` is any container object. The `<iterator>` is any quantity which can be used to access, term by term, the elements of the container object. If `iterable` is an ordered container, e.g., a *list* `a`, then `iterator` could be an integer `i` within the range of the *list*. The block above would include references to, e.g., `a[i]`.

This sounds very abstract and needs to be elucidated. Many of the conventional C and Fortran uses will be postponed to Chapter 4 because the core Python being described here can offer only a very inefficient implementation of them. We start with a simple, but unconventional example.[5]

```
c = 4
for c in "Python":
    print c

c
```

Using `c` as the loop iterator here overwrites any previous use of `c` as an identifier. For each character in the *string* iterable, the block is executed, which here merely prints its value. When there are no more characters, the loop exits, and `c` refers to its last loop value.

At first sight it, looks as though `<iterator>` and `<iterable>` have to be single objects but, as will become normal, we can circumvent this requirement by using *tuples*. For example, suppose `Z` is a list of *tuples* each of length 2. Then a loop with two iterators can be constructed via

```
for (x,y) in Z:
    <block>
```

and is perfectly permissible. For another generalization, consider the **`zip`** function introduced in Section 4.4.1.

Before we can exhibit usage that is more traditional, we need to introduce the built-in Python **`range`** function. Its general form is

```
range(start,end,step)
```

which generates the *list* of integers `[start,start+step,start+2*step,...]` as long as each integer is less than `end`. (We have seen this before in the discussion of slicing in

[5] If you are using a version of Python $\geqslant 3.0$, this code will fail. See Section 3.8.6 to find out why, and the trivial change needed to repair the snippet.

Section 3.5.3.) Here, `step` is an optional argument, which defaults to one, and `start` is optional, defaulting to zero. Thus `range(4)` yields `[0,1,2,3]`. Consider the following example

```
L=[1,4,9,16,25,36]
for it in range(len(L)):
    L[it]+=1
L
```

Note that the loop is set up at the execution of the **for** statement. The block can change the iterator, but not the loop. Try the simple example

```
for it in range(4):
    it*=2
    print it
it
```

It should be emphasized that although the loop bodies in these examples are trivial, this need not be so. Python offers two different ways of dynamically altering the flow of control while the loop is executing. In a real-life situation, they could of course both occur in the same loop.

3.7.2 The Python *continue* statement

Consider the schematic example

```
for <iterator> in <iterable>:
    <block1>
    if <test1>:
        continue
    <block2>
<block5>
```

Here `<test1>` returns a Boolean value, presumably as a result of actions in `<block1>`. On each pass, it is checked, and if `True` control passes to the top of the loop, so that `<iterator>` is incremented, and the next pass commences. If `<test1>` returns `False`, then `<block2>` is executed, after which control passes to the top of the loop. At the termination of the loop (in the usual way), `<block5>` is executed.

3.7.3 The Python *break* statement

The **break** statement allows for premature ending of the loop and, optionally via the use of an **else** clause, different outcomes. The basic syntax is

```
for <iterator> in <iterable>:
    <block1>
    if <test2>:
        break
    <block2>
else:
    <block4>
<block5>
```

If on any pass `test2` produces a `True` value, the loop is exited and control passes to <`block5`>. If <`test2`> gives `False`, then the loop terminates in the usual way, and control passes first to <`block4`> and ultimately to <`block5`>. The author finds the flow of control here to be counterintuitive, but the use of the **`else`** clause in this context is both optional and rare. Here is a simple example.

```
y=107
for x in range(2,y):
   if y%x == 0:
       print y, " has a factor ", x
       break
else:
    print y, " is prime."
```

3.7.4 *List* comprehensions

A task which arises surprisingly often is the following. We have a *list* `L1`, and need to construct a second *list* `L2` whose elements are a fixed function of the corresponding elements of the first. The conventional way to do this is via a **`for`** loop. For example, let us produce an itemwise squared *list*.

```
L1=[2,3,5,7,11,14]
L2=[]                # Empty list
for i in range(len(L1)):
    L2.append(L1[i]**2)

L2
```

However, Python can execute the loop in a single line via a *list comprehension*.

```
L1=[2,3,5,7,11,14]
L2=[x**2 for x in L1]
L2
```

Not only is this shorter, it is faster, especially for long *lists*, because there is no need to construct the explicit for-loop.

List comprehensions are considerably more versatile than this. Suppose we want to build `L2`, but only for the odd numbers in `L1`.

```
L2=[x*x for x in L1 if x%2]
```

Suppose we have a *list* of points in the plane, where the coordinates of the points are stored in *tuples*, and we need to form a *list* of their Euclidean distances from the origin.

```
import math
lpoints=[(1,0),(1,1),(4,3),(5,12)]
ldists=[math.sqrt(x*x+y*y) for (x,y) in lpoints]
```

Next suppose that we have a rectangular grid of points with the *x*-coordinates in one *list* and the *y*-coordinates in the other. We can build the distance *list* with

```
l_x=[0,2,3,4]
l_y=[1,2]
l_dist=[math.sqrt(x*x+y*y) for x in l_x for y in l_y]
```

List comprehension is a Python feature which, despite its initial unfamiliarity, is well worth mastering.

3.7.5 Python *while* loops

The other extremely useful loop construct supported by Python is the **while** loop. In its simplest form, the syntax is

```
while <test>:
    <block1>
<block2>
```

Here `<test>` is an expression which yields a Boolean object. If it is `True`, then `<block1>` is executed. Otherwise, control passes to `<block2>`. Each time `<block1>` reaches its end, `<test>` is re-evaluated, and the process repeated. Thus the construct

```
while True :
    print "Type Control-C to stop this!"
```

will, without outside intervention, run for ever.

Just as with **for** loops, **else**, **continue** and **break** clauses are available, and the last two work in the same way, either curtailing execution of the loop or exiting it. Note in particular the code snippet above can become extremely useful with a **break** clause These were discussed in Section 3.7.3.

3.8 Functions

A *function* (or *subroutine*) is a device which groups together a sequence of statements that can be executed an arbitrary number of times within a programme. To increase generality, we may supply input arguments which can change between invocations. A function may, or may not, return data.

In Python, a function is, like everything else, an *object*. We explore first the basic syntax and the concept of scope and only finally in Sections 3.8.2–3.8.5 the nature of input arguments. (This may seem an illogical order, but the variety of input arguments is extremely rich.)

3.8.1 Syntax and scope

A Python function may be defined *anywhere* in a programme before it is actually used. The basic syntax is outlined in the following piece of pseudocode

```
def <name>(<arglist>):
   <body>
```

The **def** denotes the start of a function definition; <name> assigns an identifier or name to address the object. The usual rules on identifier names apply, and of course the identifier can be changed later. The brackets () are mandatory. Between them, we may insert zero, one or more variable names, separated by commas, which are called *arguments*. The final colon is mandatory.

Next follows the *body* of the function, the statements to be executed. As we have seen already, such blocks of code have to be indented. The conclusion of the function body is indicated by a return to the same level of indentation as was used for the **def** statement. In rare circumstances we may need to define a function, while postponing the filling-out of its body. In this preliminary phase, the body should be the single statement **pass**. It is not mandatory, but conventional and highly recommended, to include a *docstring*, a description of what the function does, between the definition and body. This is enclosed in a pair of triple quotes, and can be in free format extending over one or more lines. The inclusion of a docstring may seem to be an inessential frippery, but it is not. The author of the **len** function included one which you accessed in Section 3.5.1 via **len**?. Treat your readership (most likely **you**) with the same respect!

The body of the function definition introduces a new private namespace which is destroyed when the execution of the body code terminates. When the function is invoked, this namespace is populated by the identifiers introduced as arguments in the **def** statement, and will point to whatever the arguments pointed at when the function was invoked. New identifiers introduced in the body also belong in this namespace. Of course, the function is defined within a namespace which contains other external identifiers. Those which have the same names as the function arguments or identifiers already defined in the body do not exist in the private namespace, because they have been replaced by those private arguments. The others are visible in the private namespace, but it is strongly recommended not to use them unless the user is absolutely sure that they will point to the same object on every invocation of the function. In order to ensure portability when defining functions, try to use only the identifiers contained in the argument list and those which you have defined within, and intrinsic to, the private namespace.

Usually we require the function to produce some object or associated variable, say y, and this is done with a line **return** y. The function is exited after such a statement and so the **return** statement will be the last executed, and hence usually the last statement in the function body. In principle, `y` should be a scalar, but this is easily circumvented by using a *tuple*. For example, to return three quantities, say `u`, `v` and `w` one should use a *tuple*, e.g., **`return`** `(u,v,w)` or even **`return`** `u,v,w`. If however there is no return statement Python inserts an invisible **`return`** `None` statement. Here `None` is a special Python variable which refers to an empty or void object, and is the "value" returned by the function. In this way, Python avoids the Fortran dichotomy of *functions* and *subroutines*.

Here are some simple toy examples to illustrate a number of points. It is worthwhile typing them into the interpreter so that after verifying the points made here you can experiment further.

```
1  def add_one(x):
2      """ Takes x and returns x + 1. """
3      x=x+1
4      return x
5
6  x=23
7  add_one?                    # or help(add_one)
8  add_one(0.456)
9  x
```

In lines 1–4, we define the function `add_one(x)`. There is just one argument `x` introduced in line 1, and the object which it references is changed in line 3. In line 6, we introduce an integer object referenced by `x`. The next line tests the docstring and the one after that the function. The final line checks the value of `x`, which remains unchanged at `23` despite the fact that we implicitly assigned `x` to a float in line 8.

Next consider a faulty example

```
1  def add_y(x):
2      """ Adds y to x and returns x+y. """
3      return x+y
4
5  add_y(0.456)
6  y=1
7  add_y(0.456)
```

In line 5, private `x` is assigned to 0.456, and line 3 looks for a private `y`. None is found, and so the function looks for an identifier `y` in the enclosing namespace. None can be found and so Python stops with an error. After `y` has been introduced in line 6, the second invocation of `add_y` in line 7 works as expected. However, this is non-portable behaviour. We can only use the function inside namespaces where a `y` has already been defined. There are a few cases where this condition will be satisfied, but in general this type of function should be avoided.

Better coding is shown in the following example which also shows how to return multiple values via a *tuple*, and also that functions are objects.

```
1  def add_x_and_y(x, y):
2      """ Add x and y and return them and their sum. """
3      z = x+y
4      return x,y,z
5
6  a,b,c = add_x_and_y(1,0.456)
7  a
8  b
9  c
10 f=add_x_and_y
11 f(1,0.456)
12 add_x_and_y
13 f
```

The identifier z is private to the function, and is lost after the function has been left. Because we assigned c to the object pointed to by z, the object itself is not lost when the identifier z disappears. Lines 10–13 show that functions are objects, and we can assign new identifiers to them. (It may be helpful to look again at Figure 3.1 at this point.)

In all of these examples, the objects used as arguments have been immutable, and so have not been changed by the function's invocation. This is not quite true if the argument is a mutable container, as is shown in the following example.

```
L = [0,1,2]
id(L)
def add_with_side_effects(M):
    """ Increment first element of list. """
    M[0]+=1

add_with_side_effects(L)
L
id(L)
```

The *list* L has not been changed. However, its contents can, and have been changed, with no assignment operator (outside the function body). This is a *side effect* which is harmless in this context, but can lead to subtle hard-to-identify errors in real-life code. The cure is to make a private copy

```
L = [0,1,2]
id(L)
def add_without_side_effects(M):
    """ Increment first element of list. """
    MC=M[ : ]
    MC[0]+=1
    return MC
```

```

L = add_without_side_effects(L)
L
id(L)
```

In some situations, there will be an overhead in copying a long *list*, which may detract from the speed of the code, and so there is a temptation to avoid such copying. However, before using functions with side effects remember the adage “premature optimization is the root of all evil”, and use them with care.

3.8.2 Positional arguments

This is the conventional case, common to all programming languages. As an example, consider

```
def foo1(a,b,c):
    <body>
```

Every time `foo1` is called, precisely three arguments have to be supplied. An example might be `y=foo1(3,2,1)`, and the obvious substitution by order takes place. Another way of calling the function might be `y=foo1(c=1,a=3,b=2)`, which allows a relaxation of the ordering. It is an error to supply any other number of arguments.

3.8.3 Keyword arguments

Another form of function definition specifies *keyword* arguments, e.g.

```
def foo2(d=21.2,e=4,f='a'):
    <block>
```

In calling such a function we can give all arguments, or omit some, in which case the default value will be taken from the **`def`** statement. For example, invoking `foo2(f='b')` will use the default values `d=21.2` and `e=4` so as to make up the required three arguments. Since these are keyword arguments, the order does not matter.

It is possible to combine both of these forms of argument in the same function, provided all of the positional arguments come before the keyword ones. For example

```
def foo3(a,b,c,d=21.2,e=4,f='a')
    <block>
```

Now in calling this function, we must give between three and six arguments, and the first three refer to the positional arguments.

3.8.4 Variable number of positional arguments

It frequently happens that we do not know in advance how many arguments will be needed. For example, imagine designing a `print` function, where you cannot specify in advance how many items are to be printed. Python uses *tuples* to resolve this issue. Here is an example to illustrate syntax, method and usage.

```
1  def average(*args):
2      """ Return mean of a non-empty tuple of numbers. """
3      print args
4      sum=0.0
5      for x in args:
6          sum+=x
7      return sum/len(args)
8
9  average(1,2,3,4)
10 average(1,2,3,4,5)
```

It is traditional, but not obligatory, to call the *tuple* `args` in the definition. The asterisk is however mandatory. Line 3 is superfluous, and is simply to illustrate that the arguments supplied really do get wrapped into a *tuple*. Note that by forcing **`sum`** to be real in line 4, we ensure that the division in line 7 works as we would expect, even though the denominator is an integer.

3.8.5 Variable number of keyword arguments

Python has no problem coping with a function which takes a certain fixed number of positional arguments, followed by an arbitrary number of positional arguments, followed by an arbitrary number of keyword arguments, since the ordering "positional before keyword" is preserved. As the previous section showed, the additional positional arguments are packed into a *tuple* (identified by an asterisk). The additional keyword arguments are wrapped into a *dictionary* (identified by a repeated asterisk). The following example exemplifies the process.

```
def show(a, b, *args, **kwargs):
  print a, b, args, kwargs

show(1.3,2.7,3,'a',4.2,alpha=0.99,gamma=5.67)
```

Beginners are not likely to want to use proactively all of these types of arguments. However, they will see them occasionally in the docstrings of library functions, and so it is useful to know what they are.

3.8.6 The Python *print* function

Until the last two sections, we have not needed a print function. Working in the interpreter we can "print" any Python object by simply typing its identifier. However, once we start to write functions and programmes, we do need such a function. There is a small complication here. In versions of Python < 3.0, **`print`** looks like a command, and would be invoked by, e.g.

```
print <things to be printed>
```

In versions of Python ⩾ 3.0, **print** is implemented as a function, and the code line above becomes

```
print(<things to be printed>)
```

At the time of writing, *numpy* and its extensions are only available using the earlier versions. Since earlier versions of Python may eventually be deprecated, numerical users face a potential obsolescence dilemma, which is however resolved easily in one of two ways. The **print** function expects a variable number of arguments, implicitly a *tuple*. If we use the second of the two code lines above with earlier versions of Python, then the print command sees an explicit *tuple* because of the () and prints `<things to be printed>` surrounded by brackets. If the extra brackets prove annoying, then the alternative approach is to include the line

```
from __future__ import print_function
```

at the top of the code, which will remove them when using the Python ⩾ 3.0 **print** function within Python < 3.0 code.

The print command expects a *tuple* as its argument. Thus assuming `it` refers to an integer, and `y` to a float, we might write, dropping the *tuple* delimeters, simply

```
print "After iteration ",it,", the solution was ",y
```

Here we have no control over the formatting of the two numbers, and this is a potential source of problems, e.g., when producing a table, where uniformity is highly desirable. A first attempt at a more sophisticated approach is

```
print "After iteration %d, the solution was %f" % (it,y)
```

Here the *string* argument is printed with the `%d` term replaced by an integer and the `%f` term replaced by a float, both chosen from the final *tuple* in the argument slot. As written, the output of this version is identical to that of the previous line, but we can improve significantly on that. The format codes summarized below are based on those defined for the `printf` function in the C-family of languages.

We start by considering the format of the integer `it`, which we assume takes the value `41`. If we replace `%d` in the code line with `%5d`, the output will be right justified in a field of 5 characters, i.e., ␣␣␣41, where the character ␣ denotes a space. Similarly `%-5d` will produce left justified output 41␣␣␣. Also `%05d` will give 00041. If the number is negative, then a minus sign counts as a character in the field. This can lead to untidiness when printing both positive and negative integers. We can enforce the printing of a leading plus or minor sign by choosing `%+5d` for right justification and `%+-5d` for left justification. Of course, there is nothing special about the field width 5. It can be replaced by any suitable choice. Indeed, if the exact representation of the integer requires more characters than have been specified, then Python overrides the format instruction so as to preserve accuracy.

For the formatting of floats, there are three basic possibilities. Suppose `y` takes the value `123.456789`. If we replace `%f` by `%.3f`, the output will be 123.457, i.e., the value is truncated to three decimal places. The format code `%10.3f` prints it right justified in a field of width 10, i.e., ␣␣␣123.457, while `%-10.3f` does the same, left justified. As with integer formatting, a plus sign immediately after the percent sign forces the printing of a leading plus or minus sign. Also `%010.3f` replaces the leading spaces by zeros.

Clearly, the `%f` format will lose precision if `y` is very large or very small, and we consider the case `z=1234567.89`. On the printed page, we might write $z = 1.23456789 \times 10^6$ and in Python `z=1.23456789e6`. Now consider the print format code `%13.4e`, which applied to this `z` produces ␣␣␣1.2346e+06. There are precisely four integers after the decimal point, and the number is printed in a field thirteen characters wide. In this representation, only ten characters are needed and the number is right justified with three space to its left. Just as above, `%-13.4e` produces left justification and `%+13.4e` produces a leading sign, and only two spaces. (`%+-13.4e` does the same with left justification.) If the width is less than the minimum ten required here, then Python will increase it to ten. Finally, replacing ’e’ by ’E’ in these codes ensures an upper case character, e.g., `%+-13.4E` produces +1.2346E+06␣␣.

Sometimes we need to print a float whose absolute value can vary widely, but we wish to display a particular number of significant digits. With `z` as above, `%.4g` will produce 1.235e+06, i.e., correct to four significant digits. Note that `%g` defaults to `%.6g`. Python will use whichever of the ’e’ and ’f’ formats is the shorter. Also `%.4G` chooses between the ’E’ and ’f’ formats.

For completeness, we note that *string* variables are also catered for, e.g., `%20s` will print a *string* in a field of at least 20 characters, with left padding by spaces in needed.

3.8.7 Anonymous functions

It should be obvious that the choice of a name for a function’s arguments is arbitrary, $f(x)$ and $f(y)$ refer to the same function. On the other hand, we saw in the code snippet for the function `add_x_and_y` in Section 3.8.1 that we could change the name to `f` with impunity. This is a fundamental tenet of mathematical logic, usually described in the formalism of the *lambda-calculus* or *λ-calculus*. There will occur situations in Python coding where the name of the function is totally irrelevant, and Python mimics the lambda calculus. We could have coded `add_x_and_y` as

```
f = lambda x, y : x, y, x+y
```

or just

```
lambda x, y : x, y, x+y
```

See Sections 4.1.5 and 7.5.3 for more realistic cases of anonymous functions.

3.9 Introduction to Python classes

Classes are extremely versatile structures in Python. Because of this, the documentation is both long and complicated. Examples tend to be drawn from non-scientific data-processing and are often either too simple or unduly complicated. The basic idea is that you may have a fixed data structure or object which occurs frequently, together with operations directly associated to it. The Python *class* encapsulates both the object and its operations Our introductory presentation by way of an scientific example is concise but contains many of those features most commonly used by scientists. In this pedagogic context, we have of course to use integer arithmetic.

Our example is that of fractions, which can be thought of as arbitrary precision representations of real numbers. Python does itself include a `fraction` class, and so to avoid confusion our pedagogic example will be called `Frac`. We have in mind to implement a `Frac` as a pair of integers `num` and `den`, where the latter is non-zero. We have to be careful however for 3/7 and 24/56 are usually regarded as the same number. We looked at this particular problem in Section 2.6, and we will need access to the file `gcd.py` created there. The following snippet, which shows a skeleton `Frac` class, relies on it

```
# File frac.py

import gcd

class Frac:
    """ Fractional class. A Frac is a pair of integers num, den
        (with den!=0) whose GCD is 1.
    """

    def __init__(self,n,d):
        """ Construct a Frac from integers n and d.
            Needs error message if d = 0!
        """
        hcf=gcd.gcd(n, d)
        self.num, self.den = n/hcf, d/hcf

    def __str__(self):
        """ Generate a string representation of a Frac. """
        return "%d/%d" % (self.num,self.den)

    def __mul__(self,another):
        """ Multiply two Fracs to produce a Frac. """
        return Frac(self.num*another.num, self.den*another.den)

    def __add__(self,another):
        """ Add two Fracs to produce a Frac. """
```

```
27          return Frac(self.num*another.den+self.den*another.num,
28                      self.den*another.den)
29
30      def to_real(self):
31          """ return floating point value of Frac. """
32          return float(self.num)/float(self.den)
33
34  if __name__=="__main__":
35      a=Frac(3,7)
36      b=Frac(24,56)
37      print "a.num= ",a.num, ", b.den= ",b.den
38      print a
39      print b
40      print "floating point value of a is ", a.to_real()
41      print "product= ",a*b,", sum= ",a+b
```

The first novelty here is the **class** statement in line 5. Note the terminating colon. The actual class is defined through indentation and here extends from line 5 to line 32. Lines 6–8 define the class docstring for online documentation. Within the class. we have defined five class functions, which are indented (because they are in the class) and which have indented bodies (as usual).

The first one, lines 10–15, would in other languages be called a "constructor" function. Its purpose is to turn pairs of integers into `Frac` objects. The mandatory name **`__init__`** is strange, but, as we shall see, it is never used outside the class definition. The first argument is usually called **`self`**, and it too never appears outside the function definition. This all looks very unfamiliar, so look next at line 35 of the test suite, `a=Frac(3,7)`. This is asking for a `Frac` object for the integer pair 3,7 to be referenced by the identifier `a`. Implicitly, this line invokes the **`__init__`** function with **`self`** replaced by `a` and `n` and `d` replaced by 3 and 7. It then computes `hcf`, the GCD of 3 and 7 (here 1), and in line 15 it computes `a.num` as `n/hcf` and `a.den` as `hcf`. These are accessible in the usual way, and in line 37 the first of them gets printed. The same applies to the assignment in line 36, where **`__init__`** gets called with **`self`** replaced by `b`.

Almost every class would benefit from having lines like 38–39, where class objects are "printed". This is the purpose of the class string function `__str__`, which is defined in lines 17–19. When line 38 is executed, `__str__` is invoked with **`self`** replaced by `a`. Then line 19 returns the *string* `"3/7"`, and this is what is printed.

While these first two class functions are more or less essential, we may define many or few functions to carry out class operations. Let us concentrate on multiplication and addition, according to the standard rules

$$\frac{n_1}{d_1} * \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}, \qquad \frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + d_1 n_2}{d_1 d_2}.$$

Multiplication of two `Frac` objects requires the definition of the class function `__mul__` in lines 21–23. When we invoke `a*b` in line 41, this function is invoked with **`self`** replaced by the left operand, here `a`, and `another` replaced by the right operand, here `b`.

Note how line 23 computes the numerator and denominator of the product, and then calls **`__init__`** to create a new `Frac` object, so that `c=a*b` would create a new `Frac` with identifier `c`. In line 41 though, the new, as yet unnamed `Frac`, is passed immediately to `__str__`. As the code snippet shows, addition is handled in exactly the same way.

For the sake of brevity, we have left the class incomplete. It is a valuable exercise for the beginner to enhance the code gradually in ways which are more advanced.

1 Create division and subtraction class functions called `__div__` and `__sub__`, and test them.
2 It looks a little odd if the denominator turns out to be 1 to print, e.g., 7/1. Amend the `__str__` function so that if **`self`**`.den` is 1, then the *string* created is precisely **`self`**`.num`, and test that the new version works.
3 The **`__init__`** code will certainly fail if argument `d` is zero. Think of warning the user.

3.10 The structure of Python

Near the end of Section 3.2, we pointed out the relationship between *identifiers* and *objects*, and we now return to this topic. In our pedagogic example of a Python class `Frac` in Section 3.9, we noted that an instance of a `Frac`, e.g., `a=Frac(3,7)` produces an *identifier* `a` referring to an *object* of class `Frac`. Now a `Frac` object contains data, here a pair of integers, together with a set of functions to operate on them. We access the data relevant to the instance via the "dot mechanism", e.g., `a.num`. Similarly, we access the associated functions via, e.g., `a.real()`.

So far, this is merely gathering previously stated facts. However, Python is packed with objects, some very complicated, and this "dot mechanism" is used universally to access the objects' components. We have already seen enough of Python to point out some examples.

Our first example is that of complex numbers described in Section 3.3.4. Suppose we have set `c=1.5-0.4j` or equivalently `c=`**`complex`**`(1.5,-0.4)`. We should regard complex numbers as being supplied by class `Complex`, although to guarantee speed they are hard-coded into the system. Now just as with the `Frac` class, we access the data via `c.real` and `c.imag`, while `c.conjugate()` produces the complex conjugate number `1.5+0.4j`. We say that Python is *object oriented*. Compare this approach with that of a *function oriented* language such as Fortran77, where we would have used `C=CMPLX(1.5,-0.40)`, **`REAL`**`(C)`, `AIMAG(C)` and `CONJG(C)`. At this simple level, there is no reason to prefer one approach to the other.

Our next example refers to *modules* described in Section 3.4. Like most other features in Python, a module is an *object*. Thus **`import`** `math as re` includes the `math` module and gives it the identifier `re`. We can access data as, e.g., `re.pi` and functions by, e.g., `re.gamma(2)`.

Once you master the concept of the "dot mechanism", understanding Python should become a lot clearer. See, e.g., the discussion of container objects in Section 3.5. All of the more sophisticated packages, e.g., *numpy*, *matplotlib*, *mayavi* and *pandas* rely on it. It is at this level that the *object-oriented* approach pays dividends in offering a uniform environment which is not available in early versions of C or Fortran.

3.11 Prime numbers: a worked example

We finish this chapter on "pure" Python by looking at a real problem. The internet has revolutionized communications and has emphasized the need for security of transmission of data. Much of this security is based on the fact that it is exceedingly difficult to decide whether a given integer n which is large, say $n > 10^{100}$, can be written as the product of prime numbers. We look here at a much more fundamental problem, building a list of prime numbers.

Table 3.1 The Sieve of Eratosthenes for primes $\leqslant 18$. We start by writing down in a row the integers from 2 to 18. Starting with the leftmost integer, we delete all multiples of it in line 2. Then we move to the closest remaining integer, here 3, and delete all multiples of 3 in line 3. We continue this process. Clearly, the numbers left are not products of smaller integers; they are the primes.

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

Recall that an integer p is *prime* if it cannot be written as a product $q \times r$ of integers q and r, both greater than 1. Thus the first few primes are 2, 3, 5, 7,. . . . The problem of determining all primes less than or equal to a given integer n has been studied for millennia, and perhaps the best-known approach is the *sieve of Eratosthenes*. This is illustrated, for the case $n = 18$, in Table 3.1. The caption explains how the process works, and the first three steps are shown in the figure. Notice that since any composite number to be deleted $\leqslant n$, at least one of its factors $\leqslant \sqrt{n}$. In the example, $\sqrt{18} < 5$ and so there is no need to delete multiples of $5, 7, \ldots$ from the sieve.

The following Python function implements, directly, this procedure.

```python
def sieve_v1(n):
    """
      Use Sieve of Eratosthenes to compute list of primes <= n.
      Version 1
    """
    primes=range(2,n+1)
    for p in primes:
        if p*p>n:
```

```
 9             break
10         product=2*p
11         while product<=n:
12             if product in primes:
13                 primes.remove(product)
14             product+=p
15     return len(primes),primes
```

The first five lines are conventional. In line 6, we introduce the top line of Figure 3.1 coded as a Python *list* called `primes`, which is to be sieved. Next we introduce a **`for`** loop, which runs for lines 7–14. Each choice for the loop variable `p` corresponds to a new row in the figure. As we remarked above, we need not consider $p > \sqrt{n}$, and this is tested in lines 8 and 9. The **`break`** command transfers control to the line after the end of the loop, i.e., line 15. (This can be seen from the indentation.) Now look at the **`while`** loop which runs for lines 11–14. Because of the earlier **`break`** statement, it is guaranteed that the **`while`** loop is entered at least once. Then if `product` is still in the *list*, line 13 deletes it. In Section 3.5.1, we saw that `list.append(item)` appends `item` to the `list`. Here `list.remove(item)` deletes the first occurrence of `item` in `list`. It is an error if `item` is not in `list`, and this is why its presence is guaranteed by line 12. (Try `help(list)` to see the methods available for *lists*.) Having potentially removed `2*p`, we next construct `3*p` in line 14 and repeat the process. Once all relevant multiples of `p` have been removed by iteration of the while loop, we return to line 7 and set `p` to be the next prime in the *list*. Finally, we return the list of primes and its length. The latter is an important function in number theory, where it is usually denoted $\pi(n)$.

You are recommended to construct a file called `sieves.py` and type or paste this code into it. Then in *IPython* type

```
from sieves import sieve_v1
sieve_v1?
sieve_v1(18)
```

so as to check that things are working. You can even check the time taken with a command[6]

```
timeit sieve_v1(1000)
```

Inspection of Table 3.2 shows that while the performance of this simple straightforward function is satisfactory for small n, the time taken grows unacceptably large for even moderately large values. It is a very useful exercise to see how easily we can improve its performance.

We start by considering the algorithm, which involves two loops. We can hardly avoid iterating over the actual primes, but for each prime p we then loop removing composite numbers $n \times p$, where $n = 2, 3, 4, \ldots$, from the loop. We can make two very simple

[6] The *IPython* `%run -t` introduced in Section 2.6 only works on complete scripts. However, the magic `%timeit` command times individual lines.

Table 3.2 The number of primes $\pi(n) \leqslant n$, and the approximate time taken on the author's laptop to compute them using the Python functions given here. The *relative* timings (not the *absolute* ones) are of interest here.

n	$\pi(n)$	time_v1(secs)	time_v2(secs)
10	4	3.8×10^{-6}	3.2×10^{-6}
10^2	25	1.1×10^{-4}	8.3×10^{-6}
10^3	168	8.9×10^{-3}	4.3×10^{-5}
10^4	1,229	9.3×10^{-1}	3.3×10^{-4}
10^5	9,592	95	3.7×10^{-3}
10^6	78,498		4.1×10^{-2}
10^7	664,579		4.5×10^{-1}
10^8	5,761,455		4.9

improvements here. Note that, assuming $p > 2$, any composite number less than p^2 will have been removed already from the sieve, and so we may start by removing the composite p^2. Further, the composite $p^2 + n \times p$ is even if n is odd, and so was removed from the sieve on the first pass. Therefore, we can improve the algorithm for each prime $p > 2$, by removing just p^2, $p^2 + 2p$, $p^2 + 4p, \ldots$ This is not the best we could do, e.g., 63 is sieved twice, but it will suffice.[7]

Next we consider the implementation. Here we have been rather wasteful in that five loops are involved. The **for** and **while** loops are explicit. The **if** statement in line 12 involves iterating through the primes *list*. This is repeated in line 13, and after `product` has been identified and discarded, all of the remaining *list* elements have to be shuffled down one position. Consider the following non-obvious re-implementation.

```
1  def sieve_v2(n):
2      """
3          Sieve of Eratosthenes to compute list of primes <= n.
4          Version 2.
5      """
6      sieve = [True]*(n+1)
7      for i in xrange(3,n+1,2):
8          if i*i > n:
9              break
10         if sieve[i]:
11             sieve[i*i: :2*i]=[False]*((n - i*i) // (2*i) + 1)
12     answer = [2] + [i for i in xrange(3,n+1,2) if sieve[i]]
13     return len(answer), answer
```

Here the initial sieve is implemented as a *list* of Booleans, all initialized to `True` in line 6. This will be a block of code, fast to set up, and occupying less memory than a *list* of the same length of integers. The outer loop is a **for** loop running from lines 7 to 11. In line 7, the function **xrange** is new. This behaves just like range, but rather

[7] Sieves which are more elaborate exist, e.g., those of Sundaram and of Atkin.

than creating an entire *list* in memory, it generates the members on demand. For large *lists*, this is faster and less memory-hungry. Here the loop covers the odd numbers in the closed interval $[3, n]$. Lines 8 and 9 stop the outer loop once $i > \sqrt{n}$, just as in the earlier version. Now consider lines 10 and 11. Initially, `i` is 3, and `sieve[i]` is `True`. We need to set the sieve elements for i^2, $i^2 + 2i, \ldots$ to `False`, the equivalent of discarding them in this implementation. Rather than use a loop, we carry out the operation as a single slice in line 11. (The integer factor on the right gives the dimension of the slice.) At the end of the **`for`** loop, the *list* `sieve` has had set to `False` the items with indices corresponding to all odd composite integers. Finally, in line 12 we build the list of primes. We start with a *list* containing just 2. We use a *list* comprehension to construct a *list* of all odd unsieved numbers, and then we concatenate both *lists*, before returning the result.

Although at first reading it may take some time to understand this version, nothing new (other than **`xrange`**) has been used, the code is 25% shorter and runs considerably faster, as shown in the final column in Table 3.2. Indeed, it delivers the millions of primes less than 10^8 in a few seconds. The extension to 10^9 would take a few minutes if tens of gigabytes of memory were available. Clearly, for very large primes, sieve methods are not the most efficient.

Notice too that within the *lists* in this second version, the items all have the same type. Although Python does not impose this restriction, it turns out to be worthwhile to introduce a new object, lists of homogeneous type, and this leads naturally to the *numpy*, the topic of the next chapter.

4 Numpy

Numpy is an add-on package which brings enhancements allowing Python to be used constructively for scientific computing, by offering a performance close to that of compiled languages but with the ease of the Python language. The basic object in *Numpy* is the *ndarray*. These are (possibly multi-dimensional) arrays of objects, all of the same data type, and whose size is fixed at creation. (Note that the Python *list* object does not impose homogeneity on its items, and that a *list* object can enlarge or reduce dynamically via the intrinsic `append` and `remove` functions.) The homogeneity requirement ensures that each item occupies the same space in memory. This enables the *numpy* designers to implement many operations involving *ndarray*s as precompiled C-code. Because of this, operations on *ndarrays* can be executed much more efficiently and with much less support code than that required for Python *lists*. Let us illustrate this point with an often-quoted simple example. Suppose that `a` and `b` are two Python *lists* of the same size and we want to multiply them element-wise. Using the Pythonic approach of Chapter 3, we might use

```
c=[]
for i in range(len(a)):
    c.append(a[i]*b[i])
```

However, if the *lists* contain thousands or millions of items, this will be extremely slow. Had we been using a compiled language, e.g., C, and ignoring all variable declarations etc., we might have written

```
for(i=0; i<rows; i++) {
    c[i]=a[i]*b[i];
}
```

and in two dimensions

```
for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
        c[i][j]=a[i][j]*b[i][j];
    }
}
```

Although superficially complicated, this code will execute extremely quickly. However, with *ndarrays* the code would be

```
c=a*b
```

and would use precompiled code to achieve (very nearly) the same speed. To be fair, modern compiled languages, e.g., C++ and Fortran90 can achieve the same simplicity of expression for such a simple example. *Numpy*'s library of "vectorized" functions and operations is however at least as rich as those available in compiled languages, and becomes considerably richer once *scipy*, see Section 4.9.1, is taken into account.

This chapter is an introduction to *numpy*. While the core is pretty stable, extensions near the edges are an ongoing process. The definitive documentation is a recent "user guide", Numpy Community (2013*b*) at 103 pages and the "reference manual", Numpy Community (2013*a*) at 1409 pages. Earlier, more discursive accounts, including a wealth of examples, can be found in Langtangen (2008) and/or Langtangen (2009).

Before we start, we must import the *numpy* module. The preferred approach is to preface the code with

```
import numpy as np
```

Then a *numpy* function `func` needs to be written as `np.func`. It is of course tempting to use the quick-and-dirty-fix

```
from numpy import *
```

and to drop the `np.` prefix. While this is fine for small scale experiments, experience shows that it tends to lead to namespace difficulties for real-life problems. We shall assume that the first **`import`** statement above has always been used.

Once *numpy* has been imported, we need to think about on-line documentation, for this is a very large module. It is very straightforward to see how large *numpy* is if you are following my recommendation to use the *IPython* interpreter, which utilizes *tab completion*, i.e., if we type part of a command and press the TAB key, the range of possible completions is shown, unless there is precisely one, in which case it will be completed. Trying

```
np.<TAB>
```

shows that over 500 possibilities are available, and asks whether you want to see them? Normally we will not want to see all of them. However, it is instructive to answer 'y' once. How do we make sense of this plethora?

In my view, the best place to start is the `np.lookfor` function, try

```
np.lookfor? # or help(np.lookfor)
```

for documentation. As an example, consider looking for functions whose docstrings make use of the word cosine.

```
np.lookfor('cosine')
```

reveals a surprising collection. We explore further with, e.g.

```
np.cos?  # or help(np.cos)
```

which describes the `np.cos` function.

4.1 One-dimensional arrays

Vectors or one-dimensional arrays are the basic building blocks for numerical computation. We look first at how to construct them, and then at how to use them. It makes sense here to distinguish two types of constructors, build a vector from scratch, or construct a vector to "look like" another object.

4.1.1 Ab initio constructors

Perhaps the most useful constructor is `np.linspace`, which builds an equally spaced array of **`float`**s. Its calling sequence is

```
x=np.linspace(start,stop,num=50,endpoint=True,retstep=False)
```

x is an array of length `num`. If `num` is not specified, it defaults to the value 50. The first element is `x[0]=start`. If `endpoint` is `True`, the default, then the final value is `x[-1]=stop`, and the interval spacing is `step=(stop-start)/(num-1)`. However, if we choose `endpoint` to be `False`, then `step= (stop-start)/num`, and so the final value is `x[-1]=stop - step`. Thus the parameter `endpoint` controls whether the interval is closed [start, stop] or half open [start, stop). If `retstep` is `True`, then the function returns a *tuple* consisting of the array and `step`. The following code snippet illustrates the basic use of `linspace`.

```
import numpy as np
xc,dx=np.linspace(0,1,11,retstep=True)
xc,dx
xo=np.linspace(0,1,10,endpoint=False)
xo
```

The function `np.logspace`, is similar, but the numbers are equally spaced on a logarithmic scale. See its docstring for the details and examples of use.

Somewhat closer to the **`range`** function of Python is the function, `np.arange`, which returns an array rather than a *list*. The calling sequence is

```
x=np.arange(start=0,stop,step=1,dtype=None)
```

which generates the open interval [start, stop) with intervals of `step`. Python will try to deduce the type of the array, e.g., **`int`**, **`float`** or **`complex`** from the input arguments, but this choice can be overridden by specifying the type in the last argument. Two examples are

```
import numpy as np
yo=np.arange(1,10)
yo
yoc=np.arange(1,10,dtype=complex)
yoc
```

Three further vector constructors turn out to be surprisingly useful.

```
z=np.zeros(num,dtype=float)
```

constructs an array of length `num` filled with zeros. The function `np.ones` does the same but packs the array with ones, and `np.empty` constructs an array of the same length but leaves the values of the contents unspecified.

Last but not least, we need to introduce the function `np.array`. Its simplest and most used form is

```
ca=np.array(c,dtype=None,copy=True)
```

Here `c` is any container object which can be indexed, e.g., a *list*, a *tuple* or another array. The function will try to guess an appropriate type, but this can be overwritten by using the `dtype` parameter. Possible choices include **`bool`**, **`int`**, **`float`**, **`complex`** and even user-defined objects, see Section 3.9. Here are two examples, one from a *list* and one converting an array of floats to a complex array.

```
la=np.array([1,2,3.0])
la
x = np.linspace(0,1,11)
x
cx=np.array(x, dtype=complex)
cx
```

4.1.2 Look alike constructors

Quite often these will be generated automatically, e.g., with `x` as above the line

```
y=np.sin(x)
```

will generate a new array with identifier `y` whose components are the sines of the corresponding components of `x`.

Otherwise, a very useful constructor is `np.empty_like`

```
xe=np.empty_like(x)
```

sets up an empty array (i.e., the contents have unspecified values) of the same size and type as `x`. Almost as useful are `np.zeros_like` and `np.ones_like`, which behave very much like `np.empty_like`.

Surprisingly often, we need vectors that span an interval but with different spacings on different subintervals. To take a concrete example, suppose we require `xs` to span $[0, 1]$ with spacing of 0.1, except on $[0.5, 0.6]$ where we require a spacing of 0.01. We achieve this by constructing three subintervals, two half-closed and one closed, and then joining them together. It is a useful exercise to consider carefully the following snippet.

```
1 xl=np.linspace(0,0.5,5,endpoint=False)
2 xm=np.linspace(0.5,0.6,10,endpoint=False)
3 xr=np.linspace(0.6,1.0,5)
4 xs=np.hstack((xl,xm,xr))
```

Note that `np.hstack` accepts only one argument, and so we must use a *tuple* in line 4.

We end this section by remarking that a vector is a one-dimensional instance of a *ndarray*. Any *ndarray* is a mutable container object. We studied some of the properties of such objects in the discussion of *lists* in Section 3.5. The remarks we made there, especially on slicing and copies of *lists*, apply equally well to vectors, and indeed to more general *ndarray*s.

4.1.3 Arithmetical operations on vectors

Arithmetical operations between arrays *of the same size* can be performed as the next snippet shows.

```
1 a=np.linspace(0,1,5)
2 c=np.linspace(1,3,5)
3 a+c
4 a*c
5 a/c
```

Let us look in detail at say line 3. The sum is an array of the same size as its operands. The ith component is the sum of the i-components of `a` and `c`. In this sense, the + operator is said to act *component-wise*. All of the arithmetical operations in this code snippet are acting component-wise. Incidentally, there is an efficiency issue, relevant for large arrays, which should be highlighted here. Suppose in line 3 of the snippet above we had set `a=a+c`. Python would have created a temporary array to hold the sum required on the right-hand side, filled it, then attached the identifier `a` to it, and finally deleted the original `a`-array. It is faster to use `a+=c`, which avoids the creation of a temporary array. Similar constructions are available for the other arithmetic operators acting on vectors. There is however a pitfall for the unwary. In Python, if a scalar `a` has type **`int`** and a second scalar `b` has type **`float`**, then the operation `a+=b` widens the type of `a` to float. In fact, the precise opposite holds for numpy arrays! The reader is encouraged to try

```
a=np.ones(4,dtype=int)
b=np.linspace(0,1,4)
a+=b
a
```

to see that the type of a is unchanged. The numbers in the new a have been narrowed back to int by truncation towards zero.

In general, arithmetical operations between vectors of different sizes which produce another vector cannot be defined unambiguously, and so cause an error. However, arithmetical operations between an array and a scalar can be given an unambiguous meaning, as shown in the next snippet.

```
import numpy as np
a=np.linspace(0,1,5)
2*a
a*2
a/5
a**3
a+2
```

Here the last line deserves comment. The sum (or difference) of an array, here a, and a scalar, here 2, is not defined. However, *numpy* effectively[1] chooses to *widen* or *broadcast* 2 to 2*ones_like(a) and performs the addition component-wise. Broadcasting is discussed in more detail in Section 4.2.1.

We now have enough information to consider a simple but non-trivial example, the smoothing of data by three-point averaging. Suppose f refers to a Python vector of data. We might smooth the data at interior points as follows

```
f_av=f.copy() # make a copy, just as for lists
for i in range(1,len(f)-1): # loop over interior points
    f_av[i]=(f[i-1]+f[i]+f[i+1])/3.0
```

This works well for small arrays but becomes very slow for larger vectors. Consider instead

```
f_av=f.copy() # as above
f_av[1:-1]=(f[ :-2]+f[1:-1]+f[2: ])/3.0
```

This *vectorized code* will execute much faster for large arrays since the implied loop will be executed using precompiled C code. One way to get the slicings correct is to note that in each slice [a:b], the difference a-b is the same.[2] Finally, we need to point out to beginners that there is an ***extremely important difference*** between arrays and *lists*. If l is a Python *list*, then l[:] is ***always*** a (shallow) copy, but for *numpy* arrays, slicings ***always*** reference the original array. That is why we had to enforce an explicit (deep) copy in the code snippets above.

[1] In fact, it does no such thing! However, the method it uses simulates this effect while being much more memory- and time-efficient.

[2] Recall that the slicing conventions imply [:b] is [0:b] and [a:] is [a:-0].

4.1.4 Ufuncs

The class of *universal functions* or *ufuncs* adds considerably to the usefulness of *numpy*. A *ufunc* is a function which when applied to a scalar generates a scalar, but when applied to an array produces an array of the same size, by operating component-wise. Some of the ufuncs which are most useful for scientists are shown in Table 4.1. Many of these will be familiar, but the docstrings are readily available, e.g.

```
np.fix? # or help(np.fix)
```

Table 4.1 Some commonly used *ufuncs* which can be applied to vectors. Those in the last column require two arguments. For documentation, try, e.g., `np.sign?` or `help(np.sign)`.

sign	cos	cosh	exp	power
abs	sin	sinh	log	dot
angle	tan	tanh	log10	vdot
real	arccos	arccosh	sqrt	
imag	arcsin	arcsinh		
conj	arctan	arctanh		
fix	arctan2			

One point which is not made clear in the documentation is the domain and range of each of these functions. For example, how does *numpy* interpret $\sqrt{-1}$ or $\cos^{-1} 2$? In pure Python, `math.sqrt(-1)` or `math.acos(2)` will produce an error and the program will stop. However, `cmath.sqrt(-1)` returns `1j`, for the argument is widened to a complex value.

The module *numpy* behaves differently, and the type of the argument is critical. Note that `np.sqrt(-1+0j)` takes a complex square root and returns `1j`, but `np.sqrt(-1)` produces a warning and returns `np.nan` or *not a number*, a float of indeterminate value. Further arithmetic may be performed on it, but the result will always be `np.nan`. Similar remarks apply to $\cos^{-1} 2$.

In *numpy*, direct division by zero `1.0/0.0` produces an error and the execution halts. However, if it occurs indirectly, e.g., within a loop, this is not the case. Consider

```
x = np.linspace(-2, 2, 5)
x
1.0/x
```

which produces a warning and `array([-0.5,-1.,inf,1.,0.5])`. Here `inf` behaves pretty much like infinity in further calculations. It available is as `np.inf`. Its negative is given by `np.NINF`. (There is an unfortunate asymmetry in notation here!)

There are of course many more *numpy* functions which can be applied to vectors and some of them are reviewed in Section 4.6.

In almost every non-trivial programme, there will be user-defined functions, and it is highly desirable that, where appropriate, they behave like *ufuncs*, in the sense that when applied to arrays of consistent dimensions, they return arrays of an appropriate dimension without invoking explicit loops over the components. In many cases, e.g., where only arithmetical operations and *ufuncs* are involved, this will be manifestly true.

However, if logical statements are involved, this may not be the case. Consider, e.g., the "top hat" function

$$h(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } 0 \leqslant x \leqslant 1, \\ 0 & \text{if } x > 1. \end{cases}$$

Using the techniques of Chapter 3, we might try to apply this definition via the code

```
import numpy as np
def h(x):
    """ return 1 if 0<=x<=1 else 0. """
    if x < 0.0:
        return 0.0
    elif x <= 1.0:
        return 1.0
    else:
        return 0.0

v=np.linspace(-2, 2, 401)
hv=h(v)
```

However, this fails, and the reason is very simple. If `x` has more than one element, then `x<0.0` is ambiguous. The next subsection shows how to resolve this problem.

4.1.5 Logical operations on vectors

If `x` is a scalar, then `x<0` is unambiguous. It must evaluate to `True` or `False`. But what if `x` is a *numpy* vector? Clearly, the right-hand side of the inequality is a scalar, and a moment's thought should suggest that the assertion should be interpreted component-wise, so as to produce a vector of outcomes. Let us test this hypothesis using the interpreter.

```
1  import numpy as np
2  x=np.linspace(-2,2,9)
3  y=x<0
4  x
5  y
6  z=x.copy()
7  z[y]=-z[y]
8  z
```

The primary result, see lines 3 and 5, is that `y` is a vector of length 9 of type *bool*, of which precisely the first four components are `True`. This shows that the surmise above was indeed correct. Lines 6–8 of the code demonstrate an extremely useful feature of *numpy*. We may treat logical arrays such as `y` as slice definitions on one or both sides of an assignment. First, in line 6 we make a copy `z` of `x`. The right-hand side of line 7 first selects those components of z which are negative, i.e., those for which the corresponding component of `y` is `True`, then multiplies them by -1. The assignment then inserts precisely the modified components back into `z`. Thus we have computed `z=|x|` using implicit, C-style, loops. It should be clear that the intermediate logical array `y` is redundant. We could compute `z=|x|` more quickly and clearly by

```
z=x.copy()
z[z<0]=-z[z<0]
```

The reason for the copy is to the leave the original `x` unchanged. See the warning at the end of Section 4.1.3.

Acting on scalars, we can combine logical operators, e.g., `x>0 and x<1` by chaining them, e.g., `0<x<1`. Acting on arrays, we have to carry out the comparisons individually. As an example, we might compute the hat function $h(x)$, defined in the previous subsection, for the current vector `x` via

```
h=np.ones_like(x)
h[x<0]=0.0
h[x>1]=0.0
h
```

Suppose we want to construct a more complicated function, e.g., the example $k(x)$ below. The *numpy* module offers a function `select` of considerable generality to encapsulate a number, say M, of possible choices. Because a formal description is rather complicated, the reader is advised to look ahead to the example and corresponding code snippet while reading it. We assume that `x` is a one-dimensional array of length n and label its elements x_i $(0 \leqslant i < n)$. We assume the choices form a *list* C with members C_J $(0 \leqslant J < M)$. For example, C_0 might be $x \geqslant 2$. Next we might construct a *list* B of answers of length M. Each member B_J is an array of Boolean of length n whose elements are defined by $B_{Ji} = C_J(x_i)$. Finally, we need to supply a *list* of outcomes R of length M. Each member R_J is an array of length n whose elements are defined according to

$$R_{Ji} = \begin{cases} \text{desired outcome} & \text{if } B_{Ji} = True, \\ \text{arbitrary} & \text{if } B_{Ji} = False. \end{cases}$$

Then the `select` function produces a one-dimensional array `k` of length n, and operates as follows. There is an implicit outer loop over i where $0 \leqslant i < n$, and an implicit inner loop over J where $0 \leqslant J < M$. For each fixed i, we search through the array B_{Ji} skipping all the `False` values until we reach the first `True` one. We then set k_i equal to the corresponding R_{Ji}, break out of the J-loop and move to the next i. There is a potential complication. If B_{Ji} returns `False` for fixed i and all J, then the above procedure would produce an arbitrary value for k_i. To guard against this possibility, it might be prudent to supply a default scalar value for k_i to be used in such cases.

The formal syntax is, assuming that `x`, `C` and `R` are already defined, and $m = 3$

```
result=np.select([C0, C1, C2], [R0, R1, R2], default=0)
```

Let us illustrate this by an artificial but non-trivial example

$$k(x) = \begin{cases} -x & \text{if } x < 0, \\ x^3 & \text{if } 0 \leqslant x < 1, \\ x^2 & \text{if } 1 \leqslant x < 2, \\ 4 & \text{otherwise.} \end{cases}$$

The *numpy* code might look like

```
x=np.linspace(-1,3,9)
choices=[ x>=2, x>=1, x>=0, x<0 ]
outcomes=[ 4.0, x**2, x**3, -x ]
y=np.select(choices, outcomes)
```

or more succinctly

```
x=np.linspace(-1,3,9)
y=np.select([ x>=2, x>=1, x>=0, x<0 ],
            [ 4.0, x**2, x**3, -x ])
```

Note that the order of the choices (and their outcomes) needs some care. Can you find a different ordering?

In a certain sense, the `select` function is wasteful, for all possible outcomes need to be evaluated in advance. If this is a performance issue, then *numpy* offers an alternative approach called `piecewise`, involving functions. Consider the example

$$m(x) = \begin{cases} e^{2x} & \text{if } x < 0, \\ 1 & \text{if } 0 \leqslant x < 1, \\ e^{1-x} & \text{if } 1 \leqslant x. \end{cases}$$

We could code this definition as follows

```
def m1(x):
    return np.exp(2*x)
def m2(x):
    return 1.0
def m3(x):
    return np.exp(1.0-x)
conditions=[ x<0, x<1,1<=x ]
functions=[ m1, m2, m3 ]
x=np.linspace(-10,10,2001)
m=np.piecewise(x, conditions, functions)
```

Now the functions will only be called where they are needed. Note that the length of the functions *list* should either be the same as that of the conditions *list*, or one larger.

In this case, the final function is the default choice. There are other options, see the `np.piecewise` docstring for details.

It is perhaps unlikely that the functions `m1`, `m2` and `m3` in the snippet above will be used elsewhere. However, the syntax of `np.piecewise` requires functions to be present, and so this is a situation where the anonymous functions of Section 3.8.7 can be useful. Using them, a more self-contained definition might be

```
m=np.piecewise(x, [x<0, x<1, 1<=x], [lambda x: np.exp(2*x),
                 lambda x: 1.0, lambda x: np.exp(1.0-x)])
```

4.2 Two-dimensional arrays

We turn next to two-dimensional arrays. For this, and the more general case of *n*-dimensional arrays, the official documentation is far from perfect. Fortunately, once we have mastered the basic definitions, it is easy to see that the definitions are consistent with those for vectors, and that much of what we have already learned about one-dimensional arrays or vectors carries through.

A general *numpy* array carries three important attributes: `ndim` the number of dimensions or *axes*, `shape` a *tuple* of dimension `ndim`, which gives the extent or length along each axis, and `dtype`, which gives the type of each element. Let see us them first in a familiar context.

```
v=np.linspace(0,1.0,11)
v.ndim
v.shape
v.dtype
```

Here the shape is `(11,)`, which is a *tuple* with one element and is almost, but not quite, synonymous with the number 11.

A conceptually simple method of generating two-dimensional arrays explicitly is from a *list* of *lists*, e.g.

```
1 x=np.array([[0,1,2,3],[10,11,12,13],[20,21,22,23]])
2 x.ndim
3 x,shape
4 x.dtype
5 x
6 x[2]
7 x[ : ,1]
8 x[2][1]
9 x[2,1]
```

Note first the display produced by line 5 of the code snippet. The trailing axis, corresponding to the last element of the shape is displayed horizontally, then the next axis corresponding to the penultimate element of the shape is displayed vertically. (For large arrays, the default option is that only the corners are printed.) The next two lines 6 and 7 show how we can access individual rows and columns. The good news is that the Python slicing conventions still apply. Line 8 shows one way to access an individual element of `x`. First we create an intermediate temporary vector from the last row, and then accesses an element of that vector. It is however more efficient, especially for large arrays, to access the required element directly as in line 9.

4.2.1 Broadcasting

Suppose that `y` is an array with precisely the same shape as `x`. Then `x+y`, `x-y`, `x*y` and `x/y` are arrays of the same shape, where the operations are carried out componentwise, i.e., component by component. In certain circumstances, these operations are well-defined even when the shapes of `x` and `y` differ, and this is called *broadcasting*. We generalize to a situation where we have a number of arrays, not necessarily of the same shape, to which we are applying arithmetical operations. At first sight broadcasting seems somewhat strange, but it is easier to grasp if we remember two rules:

- The *first rule of broadcasting* is that if the arrays do not have the same number of dimensions, then a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of axes.
- The *second rule of broadcasting* ensures that arrays with a size of 1 along a particular dimension or axis act as if they had the size of the array with the largest size along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcasted" array.

As a simple example, consider the array `v` with shape `(11,)` introduced on the previous page. What does `2*v` mean? Well `2` has shape `(0,)` and so by the first rule we augment the shape of "2" to `(1,)`. Next we use the second rule to increase the shape of "2" to `(11,)`, with identical components. Finally, we perform componentwise multiplication to double each element of `v`. The same holds for other arithmetic operations.[3] However, if `w` is a vector with shape `(5,)`, the broadcasting rules do not allow for the construction of `v*w`.

To see array arithmetic with broadcasting in action, consider

```
x=np.array([[0,1,2,3],[10,11,12,13],[20,21,22,23]])
r=np.array([2,3,4,5])
c=np.array([[5],[6],[7]])
2*x
x*r
r*x
c*x
```

[3] We have already seen this in Section 4.1.3. The discussion there is fully consistent with the broadcasting rules here.

Note that since `x*r=r*x`, `x*r` is NOT the same as matrix multiplication, nor is `x*x` matrix multiplication. For these, try `np.dot(x,r)` or `np.dot(x,x)`. A brief introduction to matrix arithmetic will be given in Section 4.8.

4.2.2 Ab initio constructors

Suppose we are given vectors of x-values x_i, where $0 \leqslant i < m$, and y-values y_k, where $0 \leqslant k < n$, and we want to represent a function $u(x, y)$ by grid values $u_{ik} = u(x_i, y_k)$. Mathematically, we might use a $m \times n$ array ordered in *matrix form*, e.g., for $m = 3$ and $n = 4$

$$\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ u_{10} & u_{11} & u_{12} & u_{13} \\ u_{20} & u_{21} & u_{22} & u_{23} \end{array}$$

We are thinking of x increasing downwards and y increasing rightwards. However, in the image-processing world many prefer to require x to increase rightwards and y to increase upwards, leading to *image form*

$$\begin{array}{ccc} u_{03} & u_{13} & u_{23} \\ u_{02} & u_{12} & u_{22} \\ u_{01} & u_{11} & u_{21} \\ u_{00} & u_{10} & u_{20} \end{array}$$

Clearly, the two arrays are linear transformations of each other. Because examples in the literature often use arrays corresponding to symmetric matrices, the differences are rarely spelled out explicitly. We therefore need to exhibit care.

For vectors, we found that the most useful ab initio constructors were the *numpy* functions `np.linspace` and `np.arange`, depending on whether we want to model closed or half-open intervals. For two-dimensional arrays, there are four possible intervals. As a concrete example, we try to construct explicitly a grid with $-1 \leqslant x \leqslant 1$ and $0 \leqslant y \leqslant 1$ with a spacing of 0.25 in both directions, and perform a simple arithmetical operation on it.

Perhaps the easiest to understand is the `np.meshgrid` constructor. We first construct vectors `xv` and `yv` which define the two coordinate axes for the intervals, then two arrays `xa` and `ya` on which y and x respectively are held constant. Finally, we compute their product.

```
xv=np.linspace(-1, 1, 9)
yv=np.linspace(0, 1, 5)
[xa,ya]=np.meshgrid(xv,yv)
xa
ya
xa*ya
```

By constructing either or both of the vectors `xv` and `yv` using `np.arange`, we can deal with the half-open possibilities. Notice from the shape of `xa` or `ya` that `np.meshgrid` uses *image form*.

The `np.mgrid` and `np.ogrid` operators use rather different syntax, cobbled together from slicing notation, and the representation of complex numbers. We illustrate this first in one dimension. Try out

```
np.mgrid[-1:1:9j]
np.mgrid[-1:1:0.25]
```

Note that the first line, with pure imaginary spacing, mimics the effect of the one-dimensional `np.linspace` while the second emulates `np.arange`. Although unfamiliar, this notation is succinct and generalizes to two or more dimensions, where it uses *matrix form*.

```
[xm,ym]=np.mgrid[-1:1:9j, 0:1:5j]
xm
ym
xm*ym
```

This is shorter than the first code snippet of this section, but achieves the same result up to a linear transformation. The adaptation to half-closed intervals is handled by line 2 of the snippet before this one.

For large arrays, especially with more dimensions, much of the data in `xm` and `ym` may be redundant. This deficiency is addressed by the `np.ogrid` variant which also uses *matrix form*.

```
[xo,yo]=np.ogrid[-1:1:9j, 0:1:5j]
xo
yo
xo.shape
yo.shape
xo*yo
```

You should verify that `xo` and `yo` have shapes chosen so that the broadcasting rules apply, and that `xm*ym` (from the last snippet) and `xo*yo` are identical.

In the previous section, we introduced the three functions `np.zeros`, `np.ones` and `np.empty` as vector constructors. They work equally well as constructors for more general arrays. All that is necessary is to replace the first argument, which was the length of the vector, by a *tuple* defining the shape of the array, e.g.

```
x=np.zeros((4,3),dtype=float)
```

defines a 4×3 array of floats, in *matrix form*, initialized to zeros. Of course, a *tuple* with one element, e.g., `(9,)` can be replaced by an integer, e.g., 9 in this context.

4.2.3 Look alike constructors

Much of what was said earlier for vector constructors applies here. In particular, the extremely useful `np.zeros_like`, `np.ones_like` and `np.empty_like` constructors can be used with array arguments.

Another very useful look alike constructor is `np.reshape`, which takes an existing array (or even a *list*) and a *tuple*, and, if possible, recasts the array into another whose shape is determined by the *tuple*. A common example is

```
l=range(6)
a=np.reshape(l,(2,3))
a
```

but a potentially more useful idea is

```
v=np.linspace(0, 1.0, 5)
vg=np.reshape(v, (5, 1))
vg.shape
vg
```

4.2.4 Operations on arrays and ufuncs

For arrays of the same shape, the usual arithmetic operations work as expected. There are no surprises. Operations between arrays of different shapes, or even different dimensions, are permitted if the broadcasting rules allow coercion into a common shape. Consider the example

```
u=np.linspace(10,20,3)
vg=np.reshape(np.linspace(0,7,15),(5,3))
u+vg
```

Ufuncs which take a single argument, e.g., `np.sin(x)`, work exactly as expected for arrays. Those which take more arguments, e.g., `np.power(x,y)`, also work exactly as expected if the arguments have the same shape, and are subject to the broadcasting rules otherwise.

Very little needs to be said about slicing. Although various short cuts do exist, it is safest and clearest to exhibit both dimensions, and we can slice one or both dimensions in the obvious way, e.g.

```
vg=np.reshape(np.linspace(0,7,15),(5,3))
vg
vg[1:-1,1: ]=9
vg
```

4.3 Higher-dimensional arrays

The good news here is that almost everything that has been said about two-dimensional arrays carries over into higher dimensions. The single exception is the `np.meshgrid` function, which is restricted to two dimensions. Here is a simple but instructive example of using `np.ogrid` in three dimensions.

```
[xo,yo,zo]=np.ogrid[-1:1:9j, 0:10:5j, 100:200:3j]
xo
yo
zo
xo+yo+zo
```

4.4 Domestic input and output

In this section, we look at how to communicate with humans, other programmes or store intermediate results. We can distinguish at least three scenarios and we consider simple examples of input and output processes for them. The first supposes that we are given a text file which contains both words and numbers, and we wish to read in the latter to *numpy* arrays. Conversely, we might want to output numbers to a text file. The second is similar but simpler, in that we want to process just numbers into and from text files. The third scenario is similar to the second but for reasons of speed and economy of space, we wish to use binary files which can be processed by another *numpy* programme, possibly on a different platform.

There remains the generalization of dealing with data produced by another programme, perhaps a spreadsheet or a "number crunching" non-Pythonic programme. This is discussed in Section 4.5.

4.4.1 Discursive output and input

For the sake of brevity, we assume we are reporting the result, a single float, for each of four quarters. Given *numpy* arrays containing them, we first produce a file called `q4.txt` which can be read both by humans and by other programmes. In fact, this task uses mainly core Python ideas.

```
1  import numpy as np
2  quarter=np.array([1,2,3,4],dtype=int)
3  results=np.array([37.4,47.3,73.4,99])
4  outfile=open("q4.txt","w")
5  outfile.write("The results for the first four quarters\n\n")
6  for q,r in zip(quarter, results):
7      outfile.write("For quarter %d the result is %5.1f\n" %
8                    (q,r))
9  outfile.close()
```

Lines 1–3 generate some artificial data. The Python function **open** in line 4 creates a file called `q4.txt` and opens it for writing.[4] See the docstring for further details of the **open**

[4] The names of files depend strongly on the choice of operating system. The choice here assumes *Unix* and creates or overwrites a file in the current working directory. You should replace the first *string* by the valid address of the desired file in your operating system.

function. Within the programme, the file has the arbitrary identifier `outfile`. In line 5, we write a header *string* terminated by a newline character `n` and a second newline to insert a blank line below the header. Line 6 introduces a new feature, the Python **`zip`** function. This takes a number of iterable objects, here the two *numpy* arrays, and returns the next object from each of them as a *tuple*. The **`for`** loop terminates when the shortest iterable object is exhausted. Within the loop, we write a formatted *string* containing elements from each of the arrays, and terminated with a newline. Finally, in line 9 we close the file. You can check the correctness of this snippet by looking for the file `q4.txt` (it should be in your current directory), and reading it with your favourite text editor.

Now let us consider the reverse process. We have a text file `q4.txt`, about which we know the following. The first two lines form a header, which can be discarded. Next follow an unknown number of lines, which all have the same form: a sequence of "words" separated by white space. Labelling the words in each line from zero, we wish to build two *numpy* arrays: one containing word 2 as an **`int`** and the other word 6 as a **`float`**. (To see why we want words 2 and 6, look at the text file.) The dimension of the arrays will be the same as the number of lines to be read. The following snippet carries out this task and, like the previous one, it is mainly pure Python.

```
1   infile=open("q4.txt","r")
2   lquarter=[]
3   lresult=[]
4   temp=infile.readline()
5   temp=infile.readline()
6   for line in infile:
7       words=line.split()
8       lquarter.append(int(words[2]))
9       lresult.append(float(words[6]))
10  infile.close()
11  import numpy as np
12  aquarter=np.array(lquarter,dtype=int)
13  aresult=np.array(lresult)
```

Lines 1 connects the text file to the programme for reading, and line 10 eventually disconnects it. Lines 2 and 3 create empty *lists* to hold the numbers. The file object `infile` is a *list* of *strings*, one per line. Line 4 then reads the first *string* into a *string* in memory with identifier `temp`, and so effectively deletes the first element of `infile`. Line 5 repeats the process, and so we have discarded the header. (For a more verbose header, a **`for`** loop might be more appropriate.) Now we loop through the remaining lines in `infile`. Acting on the *string* `line`, the string function `split` breaks it into a *list* of substrings, here called `words`. The breaks occur at each white space inside `line` and the white space itself is eliminated. (We could use a different splitting character, e.g., a comma. See the docstring for details. Now `words[2]` contains a *string* which we wish to convert to an integer, and the **`int`** function in line 8 does this. Next we append the integer to the `lquarter` *list*. Line 9 repeats the process for the floating-point number. Finally, in lines 12 and 13 we construct *numpy* arrays from the *lists*. Note that at no point have we needed to know how many data lines were present.

4.4.2 *Numpy* text output and input

Text-based output and input of *numpy* data are much more succinct. As an example, we first construct four vectors and an array and show how to save them to text files.

```
 1  len=21
 2  x=np.linspace(0,2*np.pi,len)
 3  c=np.cos(x)
 4  s=np.sin(x)
 5  t=np.tan(x)
 6  arr=np.empty((4,len),dtype=float)
 7  arr[0, : ]=x
 8  arr[1, : ]=c
 9  arr[2, : ]=s
10  arr[3, : ]=t
11  np.savetxt('x.txt',x)
12  np.savetxt('xcst.txt',(x,c,s,t))
13  np.savetxt('xarr.txt',arr)
```

The first ten lines merely set up some data. Line 11 creates a text file, opens it, copies the vector `x` to it using the default format `%.18e`, and closes the file. Line 12 shows how to copy many vectors (or arrays), but of equal shapes. We merely parcel them into a *tuple*. Line 13 shows how to save an array into a file. For more possibilities, see the docstring for `np.savetxt`.

With the current setup, reading the files is easy.

```
xc=np.loadtxt('x.txt')
xc,cc,sc,tc=np.loadtxt('xcst.txt')
arrc=np.loadtxt('xarr.txt')
```

There are of course possibilities that are more elaborate. See the docstring for details.

4.4.3 *Numpy* binary output and input

Writing and reading to binary files should be faster and produce more compact files because the conversions to and from text are not needed. The obvious disadvantage is that the files cannot be read by humans. Since different platforms encode numbers in different ways, there is a danger that binary files may be highly platform-dependent. *Numpy* has its own binary format which should guarantee platform-independence. However, these files cannot be read easily by other non-Python programmes.

A single vector or array is easily written or read. Using the array definitions above, we would write the array to file with

```
np.save('array.npy',arr)
```

and we could recover the array with

```
arrc = np.load('array.npy')
```

File opening and closing is handled silently provided we use the `.npy` postfix in the file name.

A collection of arrays of possibly varying shapes is handled by creating a zipped binary archive. Using the array definitions above, we might write the vectors with a one liner

```
np.savez('test.npz',x=x,c=c,s=s,t=t)
```

Recovering the arrays is a two stage process.

```
1 temp=np.load('test.npz')
2 temp.files
3 xc=temp['x']
4 cc=temp['c']
5 sc=temp['s']
6 tc=temp['t']
```

In line 2, we display the names of the files in the archive, which has the identifier `temp`. Lines 3–6 show how to recover the arrays. Notice that in both processes, file opening, zipping, unzipping and closing is handled silently. There are many other ways of performing binary input/output, in particular the *struct* module. See the documentation for details.

4.5 Foreign input and output

We now turn, albeit very briefly, to the problem of how to read in data from a non-Python source, and the answer depends on the size of the data.

4.5.1 Small amounts of data

Relatively small amounts of data are often available as *comma separated value (CSV)* files, time-series data, observational or statistical data or SQL tables or spreadsheet data. We could try to construct a reader along the lines of Section 4.4.1, but we would need to safeguard against missing or malformed data entries. Fortunately, this problem has already been handled by the *Python data analysis library* which is available as the package *pandas*.[5] This should have been included with the recommended Python distributions (see section A.1), but if not, then it can be downloaded from its website or

[5] Its website, `http://pandas.pydata.org`, is highly informative.

from the Python Package Index (see Section A.3) and that section also gives instructions for installing the package. However, before installation, interested readers should consult the excellent documentation available from the website. A recent more discursive textbook by one of the *pandas* developers, McKinney (2012), contains a wealth of potentially useful examples. It would be a waste of resources to repeat them or similar ones here, and that is why this section is so brief.

4.5.2 Large amounts of data

Large amounts of data, typically terabytes, are usually produced in a standard data format, and we shall consider here the popular *Hierarchical Data Format (HDF)* currently in version 5. The *HDF5* package[6] by itself consists merely of a format definition and associated libraries. To actually use it you need an *application programming interface (API)*. There are officially supported interfaces for C, C++, Fortran90 and Java. Clearly, HDF5 fulfills a need in scientific number crunching! Therefore, the scientific packages Mathematica, Matlab, R and SciLab offer “third party” APIs. Python is perhaps unique in that it offers two and one half, aimed at slightly different audiences.

The half refers to a minimalistic interface within *pandas*, see the relevant documentation. The package *h5py*[7] offers both low- and high-level access to HDF5, aiming for very similar functionality to its compiled language API cousins. The other package *PyTables*[8] offers only a high-level interface, but then provides additional facilities such as sophisticated indexing and query capabilities that we would expect to find only in a database.

The HDF5 documentation is not helpful in deciding which Python API to use, and you really need to study all three of them before making a decision. Note that installation of a sufficiently up-to-date version of HDF5 on Unix/Linux platforms is not for the faint-hearted. This is one instance where Windows users get a helping hand in that both *h5py* and *PyTables* offer binary installations, which include a private copy of HDF5.

4.6 Miscellaneous ufuncs

Numpy contains a number of miscellaneous ufuncs, and we group some of the most useful ones here. For a more complete review, see the *numpy* reference manual (Numpy Community 2013*a*).

4.6.1 Maxima and minima

The basic maximum function has the syntax `np.max(array,axis=None)`. Its usage is illustrated below, where the last two lines build subarrays maximizing over columns and rows respectively.

[6] Its website is `http://www.hdfgroup.org/HDF5/`.
[7] Its website, `http://www.h5py.org` offers documentation and downloads.
[8] There is a very informative website at `http://www.pytables.org`.

```
x=np.array([[5,4,1],[7,3,2]])
np.max(x)
np.max(x,axis=0)
np.max(x,axis=1)
```

Of course, if one of the elements is a nan, it is automatically the maximum of the array and any subarrays containing it. The function np.nanmax behaves almost identically, but ignores nan values. Minima are handled by np.min and np.nanmin. The range (**p**eak **t**o **p**eak) of the array, or subarray thereof, is given by np.ptp(x). Look at the function docstrings for further information.

Incidentally, np.isnan(x) returns an array of Booleans of the same shape as x with a True entry for each instance of a nan. The function np.isfinite(x) does the opposite, returning False for each nan or inf.

4.6.2 Sums and products

We can sum the elements of any array x with np.sum(x). If x has more than one dimension (we use two for illustrative purposes), then np.sum(x,axis=0) sums over the individual columns, and np.sum(x,axis=1) sums over the rows. With the same syntax, the function np.cumsum(x) produces an array of the same shape as x but with cumulative sums. The functions np.prod and np.cumprod do the same but for products. As always, consult the function docstrings for examples of usage.

4.6.3 Simple statistics

The numpy functions np.mean and np.median have the same syntax as the functions discussed above and produce the mean or median either for the whole array or along the specified axis. The averaging function

```
np.average(x,axis=None,weights=None)
```

is slightly different. If the parameter weights is specified, it must be an array with either the same shape as x or the shape appropriate to the chosen axis. The result is an appropriately weighted average.

The syntax of the variance function is

```
np.var(x,axis=0,ddof=0)
```

The first two arguments should be familiar. Note that if the elements of x are complex, the squaring operation uses a complex conjugate so as to generate a real value. The computation of the variance involves a division by n, where n is the number of elements involved. If ddof, the "delta degrees of freedom", is specified, then the divisor is replaced by n-ddof. The function np.std is very similar and computes the standard deviation.

Numpy also contains a number of functions for correlating data. The most used ones are `np.corrcoeff`, `np.correlate` and `np.cov`. Their syntax is rather different to the functions discussed above, and so the relevant docstrings need careful perusal before use.

4.7 Polynomials

Polynomials in a single variable occur very frequently in data analysis, and *numpy* offers several approaches for manipulating them. A concise way to describe a polynomial is in terms of its coefficients, e.g.

$$c_0x^4 + c_1x^3 + c_2x^2 + c_3x + c_4 \leftrightarrow \{c_0, c_1, c_2, c_3, c_4\} \leftrightarrow [c[0], c[1], c[2], c[3], c[4]]$$

stored as a Python *list*.

4.7.1 Converting data to coefficients

A rather abstract approach to define a polynomial is to specify a *list* of its roots. This defines the coefficient *list* only up to an overall factor, and the function `np.poly` always chooses `c[0]=1`, i.e., a *monic* polynomial. An example is given below.

More often we have a list or array `x` of x-values and a second one `y` of y-values and we seek a “best fit” *least squares approximation* by an unknown polynomial of given order `n`. This is called *polynomial interpolation* or *polynomial regression*, and the function `np.polyfit(x,y,n)` does precisely that.

4.7.2 Converting coefficients to data

If you are given the coefficient array, then `np.roots` delivers the roots. More usefully, given the coefficient array and a single x-value or an array of x-values, `np.polyval` returns the corresponding y-values. The following snippet illustrates the ideas of this and the preceding subsection.

```python
import numpy as np

roots=[0,1,1,2]
coeffs=np.poly(roots)
coeffs
np.roots(coeffs)
x = np.linspace(0,0.5*np.pi,7)
y=np.sin(x)
c=np.polyfit(x,y,3)
c
y1=np.polyval(c,x)
y1-y
```

4.7.3 Manipulating polynomials in coefficient form

The functions `np.polyadd`, `np.polysub`, `np.polymult` and `np.polydiv` handle the four basic arithmetic functions. The function `np.polyder` obtains the x-derivative of a given polynomial, while `np.polyint` performs x-integration, where the arbitrary constant is set to zero. As ever, more information is given in the docstrings.

4.8 Linear algebra

4.8.1 Basic operations on matrices

For arrays of the same shape, addition and multiplication by a scalar have already been defined and they act componentwise. Mathematicians often think of two-dimensional arrays as matrices.

If `A`, `B`, . . . , are *numpy* arrays with two dimensions, they will be called *matrices*. Note that unlike the usual algebra conventions, indices start with zero rather than one. The transpose of `A` is available as `A.transpose()`, or more succinctly as `A.T`. Note that *numpy* does not distinguish between column and row vectors. This means that if `u` is a one-dimensional array or vector, then `u.T = u`.

We already know how to build zero matrices, e.g., `z=np.zeros((4,4))`. The function `np.identity` creates identity matrices, e.g.

```
I=np.identity(3,dtype=float).
```

A more general form of this is the eye function. `np.eye(m,n,k,dtype=float)` returns a $m\times n$ matrix, where the kth diagonal consists of ones and the other elements are zero. Examine carefully

```
C=2*np.eye(3,4,-1)+3*np.eye(3,4,0)+4*np.eye(3,4,1)
C
```

Next consider a different situation where we have a set of m vectors `v1`, `v2`, . . . , `vm` all of length n and we want to construct a $m\times n$ matrix with the vectors as rows. This is accomplished easily with the `np.vstack` function as the following example shows.

```
1  v1=np.array([1,2,3])
2  v2=np.array([4,5,6])
3  rows=np.vstack((v1,v2))
4  rows
5  cols=rows.T
6  cols
```

Note that because the actual number of arguments in line 3 is variable, we have to wrap them in a *tuple* first, because `np.vstack` takes precisely one argument. We could of course encode them as columns creating a $n\times m$ matrix, as the last two lines of the code snippet show.

Addition and subtraction of matrices, and multiplication by a scalar are covered by the standard *numpy* componentwise operators. Matrix multiplication is via the `np.dot` function, see below for an example.

4.8.2 More specialized operations on matrices

The module *numpy* contains a submodule `linalg` which handles operations on matrices that are more specialized. Suppose that A is a square $n \times n$ matrix. The determinant of A is given by `np.linalg.det` and, assuming A is non-singular, its inverse is obtained with `np.linalg.inv`. The snippet illustrates the use of these functions.

```python
import numpy as np

a=np.array([[4,2,0],[9,3,7],[1,2,1]])
a
np.linalg.det(a)
b=np.linalg.inv(a)
b
np.dot(b,a)
```

The function `np.linalg.eig` can be used to generate eigenvalues and eigenvectors. This function delivers the eigenvectors as columns of a n-row matrix. Each column has unit Euclidean length, i.e., the eigenvectors are normalized.

```python
import numpy as np

a=np.array([[4,2,0],[9,3,7],[1,2,1]])

evals, evecs = np.linalg.eig(a)
eval1 = evals[0]
evec1 = evecs[:,0]
np.sum(evec1*evec1)
np.dot(a, evec1) - eval1*evec1
```

There are many more functions available in the `np.linalg` module. Try `np.linalg?` for details. They are all Python wrappers around the corresponding LAPACK library subroutines.

4.8.3 Solving linear systems of equations

A very common problem is the need to obtain a "solution" $\mathbf{x}$ to a linear system of equations

$$A\mathbf{x} = \mathbf{b}, \tag{4.1}$$

where A is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors.

The simplest case is where A is $n \times n$ and is non-singular, while $\mathbf{x}$ and $\mathbf{b}$ are n-vectors.

Then the solution vector **x** is well-defined and unique. It is straightforward to obtain a numerical approximation to it, as the next snippet shows. We shall treat two cases simultaneously, with

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 5 & 5 & 5 \\ 1 & 4 & 6 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 5 \\ 5 \\ -3 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ -\frac{7}{2} \end{pmatrix}$$

```
import numpy as np
a=np.array([[3,2,1],[5,5,5],[1,4,6]])
b=np.array([[5,1],[5,0],[-3,-7.0/2]])
x=np.linalg.solve(a,b)
x
np.dot(a,x)-b
```

Here the last line checks the solution.

This is but the briefest introduction to the linear algebra capabilities of *numpy*. See also Section 4.9.1.

4.9 More *numpy* and beyond

The *numpy* module contains far more resources than those sketched here, in particular several specialized groups of functions including:

- *numpy.fft*, a collection of discrete Fourier transform routines;
- *numpy.random*, which generates random numbers drawn from a variety of distributions. We offer a snapshot of some of its possibilities in Section 7.6 on stochastic differential equations.

Information about these, as well as some specialist functions is available via the interpreter, and also in the reference manual, Numpy Community (2013*a*).

4.9.1 *Scipy*

The *scipy* module contains a wide variety of further specialized groups of functions. A subset of the available groups is:

- *scipy.special* makes many special functions, e.g., Bessel functions, readily available;
- *scipy.integrate* contains various quadrature functions and, most importantly, routines for solving the initial value problems for systems of ordinary differential equations (see Section 7.1);
- *scipy.optimize*, which covers optimization and root finding functions, for which a simple example is given below;
- *scipy.fftpack* which contains a more extensive set of routines involving discrete Fourier transforms;

- *scipy.linalg*, *scipy.sparse*, *scipy.sparse.linalg*, which extend considerably the linear algebra capabilities of *numpy*.

Accessing these within the interpreter is slightly different to the *numpy* case, e.g.

```
import scipy.optimize
scipy.optimize?
scipy.optimize.rootf?
```

As a simple example, we address a problem which arises later in Section 7.4.5, to determine all positive solutions of

$$\coth v = v.$$

A rough sketch of the graphs of each side of the equation as functions of v shows that there is precisely one solution, and that it is greater than 1. Using the information we have gleaned from the previous code snippet, the following code solves the problem.

```
import numpy as np
import scipy.optimize as sco

def fun(x):
    return np.cosh(x)/np.sinh(x)-x

rooots=sco.fsolve(fun,1.0)
root= roots[0]
print "root is %15.12f and value is %e" % (root, fun(root))
```

The required root is approximately 1.1996786402577135, for which the function value is about 3×10^{-14}.

Fuller details, including groups not mentioned here, can be found in the reference manual, Scipy Community (2012). On the whole, the documentation reaches a high standard, if not quite the equal of that for *numpy*.

4.9.2 *Scikits*

In addition, there are a number of scientific packages which are not included in *scipy* for various reasons. They may be too specialized for inclusion, they may involve software licences, e.g., GPL, which are incompatible with *scipy*'s BSD licence or quite simply they reflect work in progress. Many of these can be accessed via the Scikits web page,[9] or via the Python Package Index (see Section A.3). We shall demonstrate their usage with concrete examples. First, in the treatment of boundary value problems for ordinary differential equations, in Section 7.4, the package `scikits.bvp1lg` will be needed. In Section 7.5, which discusses simple delay differential equations, we have chosen not to use the Scikits package `skikits.pydde`, because the package `pydelay`, downloadable

[9] `http://scikits.appspot.com/`

from its web page,[10] appears to be both more versatile and user friendly. Installation of both packages is identical, and is described in Section A.3.

[10] `http://pydelay.sourceforge.net`

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

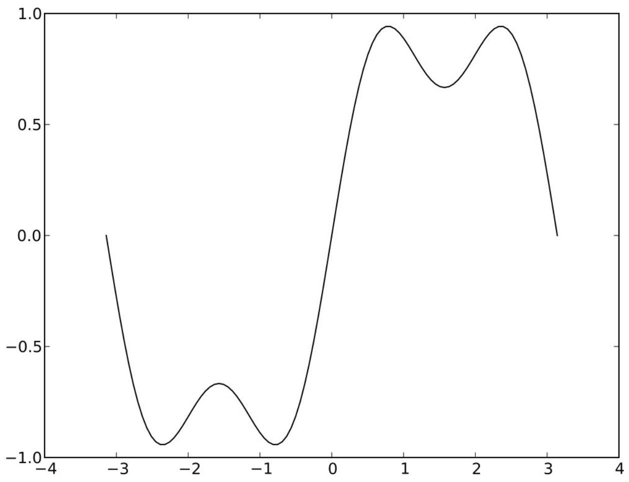

Figure 5.1 A simple plot using *matplotlib*.

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
ˆ	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

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

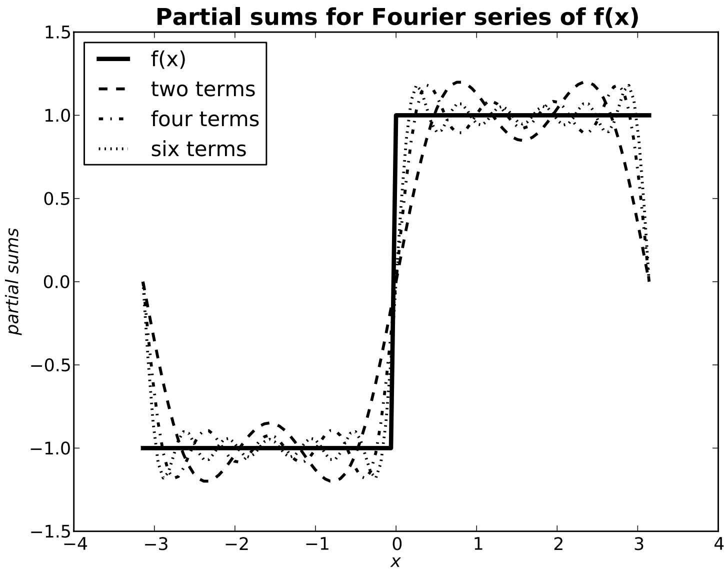

Figure 5.2 An enhanced not-so-simple plot using *matplotlib*.

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

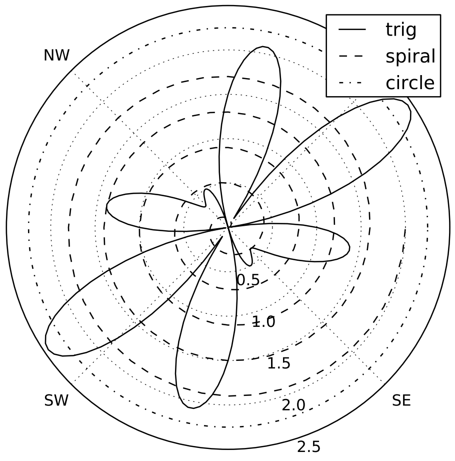

Figure 5.3 A simple polar plot using *matplotlib*.

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)
```

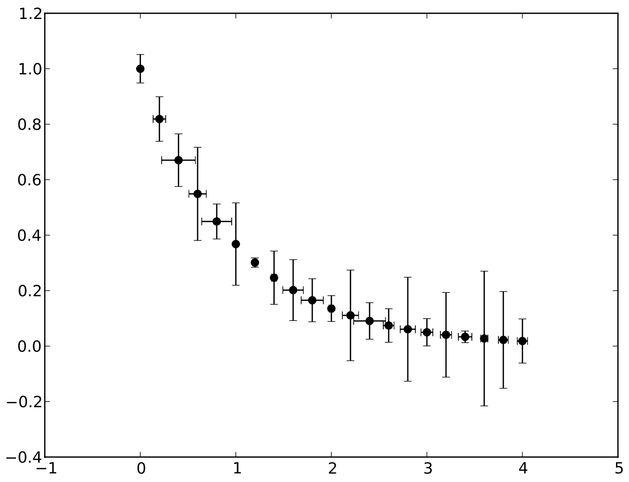

Figure 5.4 A very plain plot using error bars.

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*.

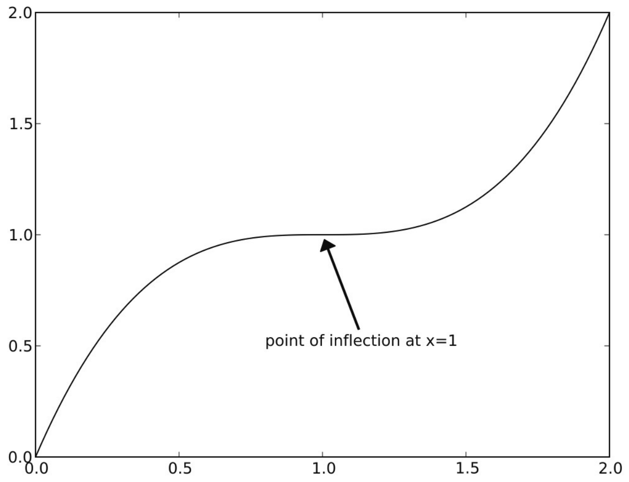

Figure 5.5 A simple example of annotation using *matplotlib*.

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`.

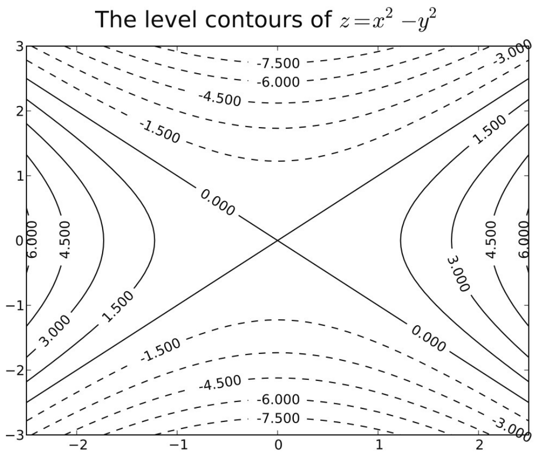

Figure 5.6 A plain contour plot using *matplotlib*.

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

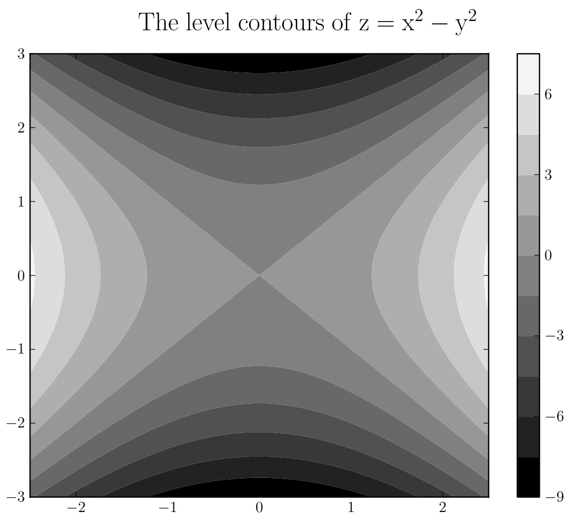

Figure 5.7 A filled contour plot using *matplotlib*.

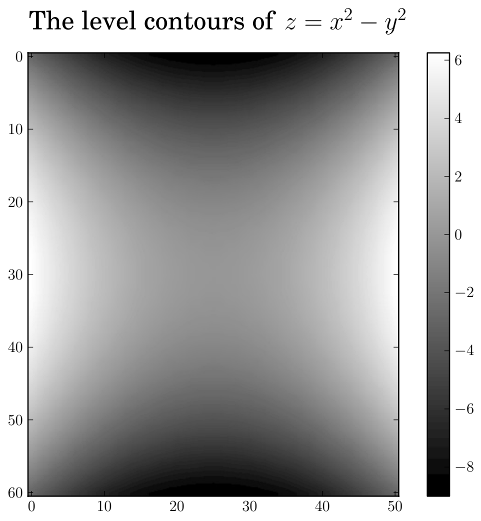

Figure 5.8 Contours implied in a continuous image using *matplotlib*.

```
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

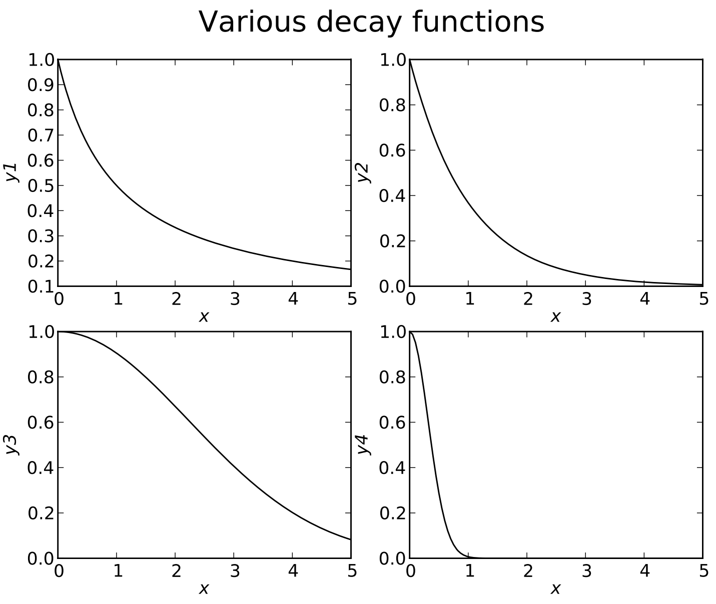

Figure 5.9 An example of compound plots using *matplotlib*.

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 $z(t, x) = \text{sech}^2(0.5 * (t - x))$, 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.

```
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

```
(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.

```
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.

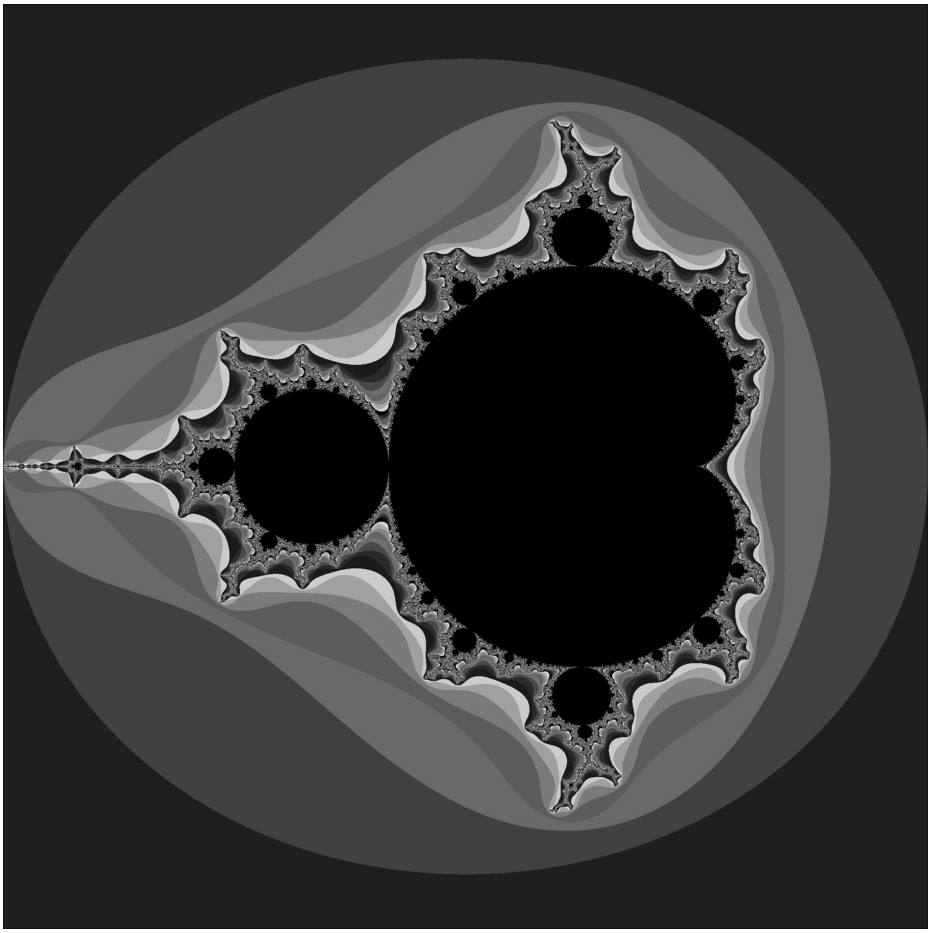

Figure 5.10 An example of a Mandelbrot set

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

6 Three-dimensional graphics

6.1 Introduction

6.1.1 Three-dimensional data sets

In the previous chapter, we saw that the Python *matplotlib* module is excellent for producing graphs of a single function, $y = f(x)$. Actually, it can do a little more than this. Suppose u is an independent variable, and a number of values of u, usually uniformly spaced, are given. Suppose we define a parametrized curve $x = x(u)$, $y = y(u)$. The *matplotlib* `plt.plot` function can easily draw such curves. As a concrete simple example, suppose θ is defined on $[0, 2\pi]$. Then $x = \cos\theta$, $y = \sin\theta$ defines a familiar curve, the unit circle. As a more intricate example consider *Lissajous' figures*, of which an example is $x = \cos(3\theta)$, $y = \sin(5\theta)$. A code snippet to draw an unadorned version of this is

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

theta=np.linspace(0,2*np.pi,401)
x=np.cos(3*theta)
y=np.sin(5*theta)
plt.ion()
plt.plot(x,y,lw=2)
```

Also *matplotlib* is good at producing contour curves. More specifically, suppose z is a function of two variables, $z = Z(x, y)$. Then, at least locally and subject to certain conditions, and possibly interchanging x and y, we can invert the relation as $y = Y(x, z)$. If we now fix the value of z, say $z = z_0$, then we obtain a family of curves $y = Y(x, z_0)$ parametrized by z_0. These are the contour curves that *matplotlib* can plot with the `plt.contour` function.

Now suppose we want to extend these capabilities to three dimensions. We shall consider three cases, with a concrete example for each. Our examples are artificial in the sense that they are predefined analytically. However, in setting them up we have to construct finite sets of discrete data. In the real world, we would instead use our own finite sets of discrete data, derived either from experiment or a complicated numerical simulation.

The first is a parametrized curve $\mathbf{x}(t) = (x(t), y(t), z(t))$, and as a specific example we consider the curve $C_{nm}(a)$

$$x = (1 + a\cos(nt))\cos(mt),\ y = (1 + a\cos(nt))\sin(mt),\ z = a\sin(nt)$$

where $t \in [0, 2\pi]$, n and m are integers and $0 < a < 1$. This is a spiral wrapped round a circular torus, with major and minor radii 1 and a respectively, a three-dimensional generalization of Lissajous' figures discussed above.

The second is a surface of the form $z = z(x, y)$. Here we consider the artificial but specific example

$$z = e^{-2x^2-y^2}\cos(2x)\cos(3y) \quad \text{where } -2 \leqslant x \leqslant 2,\ -3 \leqslant y \leqslant 3.$$

For the third case, we shall treat the more general case of a parametrized surface $x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$, which reduces to the case above if we choose $x = u$, $y = v$. As a concrete example, we consider the self-intersecting minimal surface discovered by Enneper

$$x = u(1 - u^2/3 + v^2),\ y = v(1 - v^2/3 + u^2),\ z = u^2 - v^2 \quad \text{where } -2 \leqslant u, v \leqslant 2.$$

It is not difficult to construct similar examples with a higher number of dimensions.

There is an issue which is almost trivial in two dimensions, namely the relationship, if any, between the data values. In two dimensions, when we call `plt.plot(x,y)`, `x` and `y` are treated as linear lists, and a line is drawn between contiguous data points. If we believe the data to be uncorrelated, then `plt.scatter(x,y)` would simply show the data points.

In three dimensions, there is more freedom. For example, if we were to plot the second example above when x and y were uniform arrays, e.g., as generated by `np.mgrid`, we would be tessellating a two-dimensional surface by rectangles. The Enneper surface above is a more general tiling by quadrilaterals. However, we might consider also a tessellation by triangles, or even impose no structure at all and simply plot the points. In higher dimensions, there are many more possibilities.

6.1.2 The reduction to two dimensions

The screen of a VDU and a paper page are both two dimensional, and so any representation of a three- or higher-dimensional object must ultimately be reduced to two dimensions. There are (at least) two standard processes to achieve this. In order to avoid too much abstraction in the presentation, let us assume that the object has been described in terms of coordinates $(x, y, z, w, \ldots)$.

The first reduction process, often called the "cut plane" technique is to impose some arbitrary condition or conditions on the coordinates so as to reduce the dimensionality. The simplest and most common approach is to assume constant values for all but two of the coordinates, e.g., $z = z_0$, $w = w_0, \ldots,$ thus giving a restricted but two dimensional view of the object. Hopefully, by doing this several times for a (not too large) set of choices for $z_0, w_0, \ldots,$ we will be able to recover valuable information about the object.

An example of this approach is the contour curve procedure in *matplotlib* mentioned in the previous section.

The second common reduction process is *projection*. Suppose first that the object is three dimensional. Suppose we choose an "observer direction", a unit vector in the three-dimensional space, and project the object onto the two-plane orthogonal to the observer direction. (If distinct object points project to the same image point we need to consider whether, and if so how, we want to distinguish them. This difficult issue is being ignored in this introduction.) In the standard notation for three-dimensional vectors, let $\mathbf{n}$ be the observer direction with $\mathbf{n} \cdot \mathbf{n} = 1$. Let $\mathbf{x}$ be an arbitrary three-vector. Then the projection operator $\mathcal{P}$ is defined by

$$\mathcal{P}(\mathbf{x}) = \mathbf{x} - (\mathbf{x} \cdot \mathbf{n})\mathbf{n},$$

and it is easy to see that this vector is orthogonal to $\mathbf{n}$, i.e., $\mathbf{n} \cdot \mathcal{P}(\mathbf{x}) = 0$ for all $\mathbf{x}$. In physical terms, the projection onto the plane spanned by the $\mathcal{P}(\mathbf{x})$ is what a distant observer would see. The projection technique also works in higher dimensions, but the physical interpretation is not as direct. Hopefully, by choosing enough directions we can hope to "visualize" the object.

Another point which should be noted is that if an object is to be resolved with N points per dimension and there are d dimensions, the number of points needed will be N^d. Thus if $d \geqslant 3$, we cannot expect to achieve resolutions as fine as in the two-dimensional case.

6.2 Visualization software

There are two tasks here which need to be considered separately, although the first is often glossed over.

In principle we need to convert the data to one of the more-or-less standard data formats. Perhaps the best known of these are the *HDF (Hierarchical Data Format)* mentioned in Section 4.5.2 and the *VTK (Visualization Tool Kit)*,[1] although there are plenty of others in the public domain. These formats aim at providing maximum generality in their descriptions of "objects" to be visualized, and so come with a fairly steep learning curve. The choice of a particular format is usually predicated by a choice of visualization package.

The task of a visualization package is to take data in a specified format and produce two-dimensional views of them. Packages must be fast, so that whether either cut plane or projection techniques are used, figures can be produced and altered with a minimal time lag. Given that they also need to produce a variety of outputs, images, movies etc., they tend to be very complicated with again a steep learning curve. Because they are not trivial to write, they are often commercial products with a very expensive price tag. Fortunately, there are some top quality examples in the public domain, and a far from complete list includes (in alphabetical order) *Mayavi*,[2] *Paraview*,[3] both of which are

[1] Its website is `http://www.vtk.org`.
[2] Its website is `http://mayavi.sourceforge.net`.
[3] It can be found at `http://www.paraview.org`.

based on *VTK*, and *Visit*[4] which uses *silo* or *HDF*. All three offer a Python interface to access their features. There is however a snag with the last two of the three examples listed. Like Python, they are complicated programmes which are still being developed. Because of the complexity involved in compiling them they all offer ready-to-run binary versions for all of the major platforms. Naturally, the Python interfaces only work with the Python version that was used when the binary programme was created, and this may not be the same version as that of the end user. However, because it is part of many Python distribution packages, *mayavi* should not suffer from this problem.

The philosophy of this book is both to try to minimize the addition of external software to the Python base, and, more importantly, to eliminate steep learning curves. Python avoids both of these disadvantages by offering two different approaches which are very similar to what we have already discussed. The first is a *mplot3d* module for *matplotlib*. The second is a *mlab* module which can be loaded from *mayavi* and which is itself a Python package. This makes many of the features of *mayavi* available with a *matplotlib*-like interface. For the documentation, see the user guide Ramachandandran and Variquaux (2009).

In the next three sections, we look at the two approaches to visualization for each of our three examples. The fastest and most satisfactory way to master visualization is to try out the code snippets and then experiment with them and with other examples. In each section, we first supply a code snippet to construct the data set. This will be common to both approaches, and so this first snippet should be pasted into the second or third snippet, which actually carries out the visualization. This should make it easier to reuse the latter snippets with the reader's own data set.

As usual, the use of the *IPython* interpreter is strongly recommended. No special considerations are needed when using *matplotlib*. However, if the *mayavi* package *mlab* is to be used, then *ipython* should be invoked from the command line with

```
ipython --gui=wx
```

assuming your version is 0.11 or later. Older versions need

```
ipython -wthread
```

6.3 A three-dimensional curve

We now consider the curve $C_{mn}(a)$ from the first example of Section 6.1.1. The data to be visualized are easily constructed using the following snippet, which we refer to as snippet A. This snippet is common to both of the drawing approaches illustrated below.

```
# This is snippet A

import numpy as np
```

[4] It is at `http://wci.llnl.gov/codes/visit`.

```

theta=np.linspace(0,2*np.pi,401)
a=0.3   # specific but arbitrary choice of the parameters
m=11
n=9
x=(1+a*np.cos(n*theta))*np.cos(m*theta)
y=(1+a*np.cos(n*theta))*np.sin(m*theta)
z=a*np.sin(n*theta)
```

6.3.1 Visualizing the curve with *mplot3d*

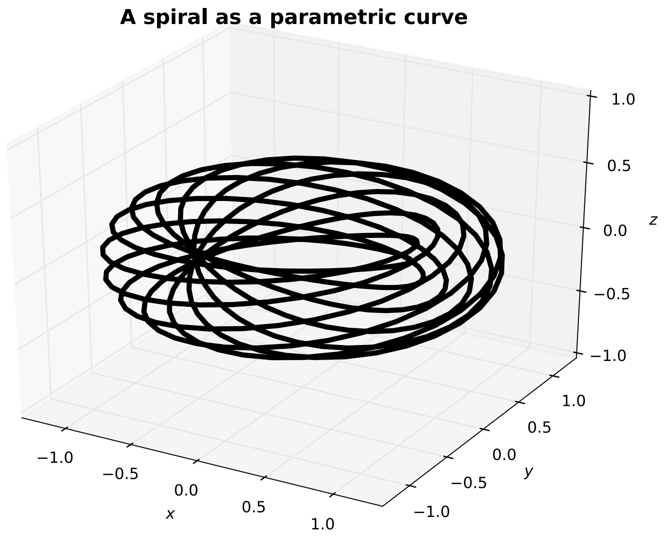

Figure 6.1 An example of a curve wrapped round a torus using *matplotlib.mplot3d* .

The next snippet uses the first to visualize the curve using *matplotlib* package *mplot3d*.

```
# Insert snippet A here.
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

plt.ion()
fig=plt.figure()
ax=Axes3D(fig)
ax.plot(x,y,z,'g',linewidth=4)
ax.set_zlim3d(-1.0,1.0)
ax.set_xlabel('x')
```

```
11  ax.set_ylabel('y')
12  ax.set_zlabel('z')
13  ax.set_title('A spiral as a parametric curve',
14              weight='bold',size=16)
15  #ax.elev, ax.azim = 60, -120
16  fig.savefig('torus.pdf')
```

Line 2 should be familiar, and line 3 introduces the `Axes3D` class object which will be needed for carrying out the visualization. Next lines 6 and 7 join up the two concepts, `ax` is an instance of the object `Axes3D` tied to an instance `fig` of the *matplotlib* `Figure` class. Thus the remaining code follows the style of Section 5.9. Line 8 actually draws the curve. The parameters of `ax.plot` are almost identical to those of the two-dimensional function `plt.plot`. Likewise lines 9–14 and 16 carry out operations familiar from the two-dimensional case.

The figure should be shown in the familiar *matplotlib* window, where the interactive buttons behave as before (see Section 5.2.4). There is important new functionality however. Simply moving the mouse over the figure with the left button depressed will change the observer direction, and the azimuth and elevation of the current direction is displayed at the lower right corner.

As written, the snippet will save the figure drawn with the default values of azimuth and elevation, −60° and 30° respectively, which are rarely the most useful ones. By experimenting, we can establish more desirable values. The commented out line 15 shows how to set them to values that are more desirable for this particular image. If you rerun the snippet with this line uncommented and amended, then the desired figure will be saved, as in, e.g., Figure 6.1. Figures for presentations often look better if a title is present, whereas they are usually unnecessary for books. As a compromise, both Figures 6.1 and 6.2 carry titles, to show how to implement them, but they are omitted for the rest of this chapter.

6.3.2 Visualizing the curve with *mlab*

The following snippet shows the default way to visualize the curve in *mayavi*. (Remember to invoke *ipython* with the `--gui=wx` parameter.)

```
# Insert snippet A here.
from mayavi import mlab

mlab.plot3d(x,y,z,np.sin(n*theta),
            tube_radius=0.025,colormap='spectral')
mlab.axes(line_width=2,nb_labels=5)
mlab.title('A spiral wrapped around a torus',size=0.6)
#mlab.show() # not needed when using ipython
mlab.savefig('torus1.png')
```

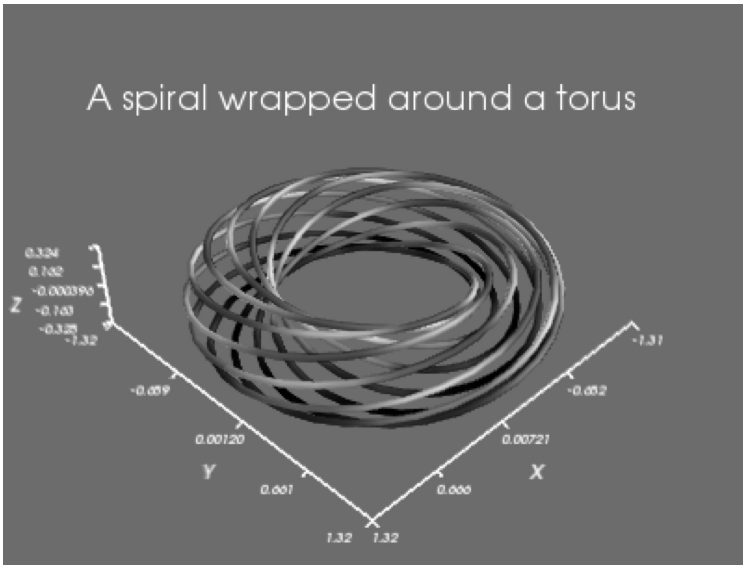

Figure 6.2 An example of a curve wrapped round a torus using the *mayavi mlab* module with the default settings.

The result is shown in Figure 6.2. Lines 4 and 5 draw the curve. Here we have chosen to colour the curve according the value of sin($n\theta$), where θ is the curve's parameter, using the colour map `spectral`. (Colour maps are discussed below.) Lines 6 and 7 produces axes and a title. The docstrings give further possibilities. (Note that the *mayavi* developers have chosen the Matlab-compatible version of names for functions to decorate figures. See Section 5.9 for a discussion of this issue.) Finally, line 9 saves a copy to disk. Note that although the documentation suggests that `mlab.savefig` can save to a variety of formats, this appears to depend on the back-end used. My back-end only saves successfully `png` files. However, if you installed *ImageMagick* as mentioned in Section 5.10.2, its command line `convert` can be used to convert a `png` file to many other formats.

The window here is rather different to that of *matplotlib*. Once the figure has been drawn we can change the viewing angle by dragging the mouse across it with the left button depressed. The right button offers pan and zoom facilities. At the top left, there are 13 small buttons for interacting with the figure. By default, the background colour is black, and button 13 (on the right) allows us to change this. Button 12 is used for saving the current scene. The formats available are implementation-dependent, but all implementations should allow the scene to be saved as a `png` file. Buttons 2–11 do fairly obvious things, but button 1 (on the left) is far more intriguing . Pressing it reveals the *mayavi* "pipeline" that *mlab* constructed to draw the figure. For example, clicking on "Colors and legends" shows the chosen colour map or "Look Up Table", abbreviated to "LUT". Clicking on the colour bar reveals the 60 available choices. The reader is urged to experiment further with the other pipeline elements!

The default choices of grey background, white axes, labels and title are very effective for visual presentations, but print formats usually expect a white background with black axes, labels and title. The easiest way to do this is via button 13, as mentioned in the preceding paragraph. Alternatively, we can modify our code. In *mayavi*, colour is managed by a triple of floating-point numbers (rgb values) stored as a tuple, and so we need to manage these for the foreground and background. Thus in the snippet above we might add the definitions

```
 black=(0,0,0)
 white=(1,1,1)
```

and then modify the figure, axis and title commands, e.g.

```
mlab.figure(bgcolor=white)
mlab.axes(line_width=2,nb_labels=5,color=black)
mlab.title('A spiral wrapped around a torus',size=0.6,
          color=black)
```

The interactive approach is simpler and more versatile.

6.4 A simple surface

In this section, we consider the simple surface defined above in Section 6.1.1 to be the graph of

$$z = e^{-2x^2-y^2}\cos(2x)\cos(3y) \quad \text{where } -2 \leqslant x \leqslant 2,\ -3 \leqslant y \leqslant 3.$$

The following snippet, referred to later as snippet B, sets up the data.

```
import numpy as np

xx, yy=np.mgrid[-2:2:81j, -3:3:91j]
zz=np.exp(-2*xx**2-yy**2)*np.cos(2*xx)*np.cos(3*yy)
```

6.4.1 Visualizing the simple surface with *mplot3d*

Figure 6.3 was drawn using the following code snippet

```
# Insert snippet B here to set up the data
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

plt.ion()
fig=plt.figure()
ax=Axes3D(fig)
ax.plot_surface(xx,yy,zz,rstride=4,cstride=3,color='c',alpha=0.9)
```

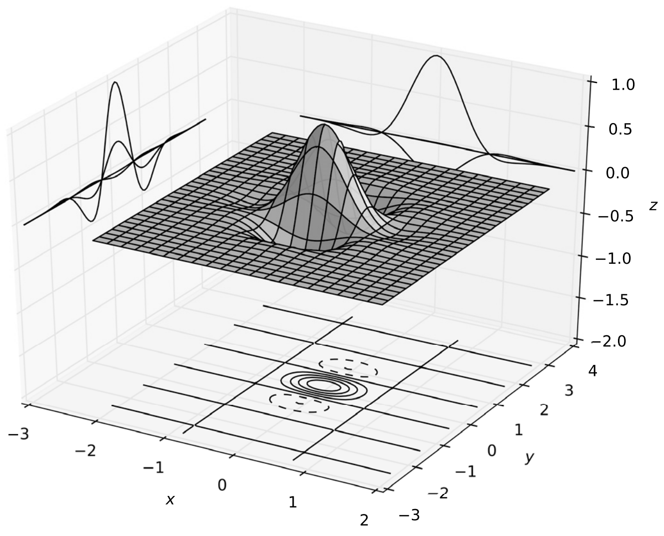

Figure 6.3 An example of a simple surface with three contour plots using the *matplotlib.mplot3d* module.

```
 9  ax.contour(xx,yy,zz,zdir='x',offset=-3.0,colors='black')
10  ax.contour(xx,yy,zz,zdir='y',offset=4.0,colors='blue')
11  ax.contour(xx,yy,zz,zdir='z',offset=-2.0)
12  ax.set_xlim3d(-3.0,2.0)
13  ax.set_ylim3d(-3.0,4.0)
14  ax.set_zlim3d(-2.0,1.0)
15  ax.set_xlabel('x')
16  ax.set_ylabel('y')
17  ax.set_zlabel('z')
18  #ax.set_title('Surface plot with contours',
19  #              weight='bold',size=18)
```

Here lines 1–7 have appeared before in the snippet used to visualize a parametric curve. Line 8 draws the surface. The parameters `rstride=4, cstride=3` mean that every fourth row and third column are actually used to construct the plot. The parameter `alpha=0.9` controls the opacity[5] of the surface, a floating-point number in the range [0, 1]. See the function's docstring for further potential parameters. Again we have seen lines 15–19 before.

If a "wire frame" representation of the surface is preferred, it can be obtained by replacing line 8 by

```
ax.plot_wireframe(xx,yy,zz,rstride=4,cstride=3)
```

[5] A high alpha means that the surface is very opaque, i.e., mesh lines behind the hump are not seen, whereas a low alpha renders the hump transparent.

The function's docstring lists the further possibilities.

Another new feature here is the drawing of contour plots. Recall that in Section 5.8 we showed how *matplotlib* could visualize the level contours of a surface $z = z(x, y)$. In line 9 of the current snippet, we do the same after rearranging the functional relationship to be $x = x(y, z)$. We have to position the plot as a yz-plane somewhere and the parameter `offset=-3` puts it in the plane $x = -3$. Now the default x-range was set by snippet B to be $x \in [-2, 2]$, which would render the contour plot invisible. Therefore, line 12 resets the x-range to be $x \in [-3, 2]$. Lines 10 and 11 and 13 and 14 deal with the other coordinate directions. Of course, there is no obligation to draw all three contour plots, or even one or two of them. Your figures will certainly look less cluttered without them!

6.4.2 Visualizing the simple surface with *mlab*

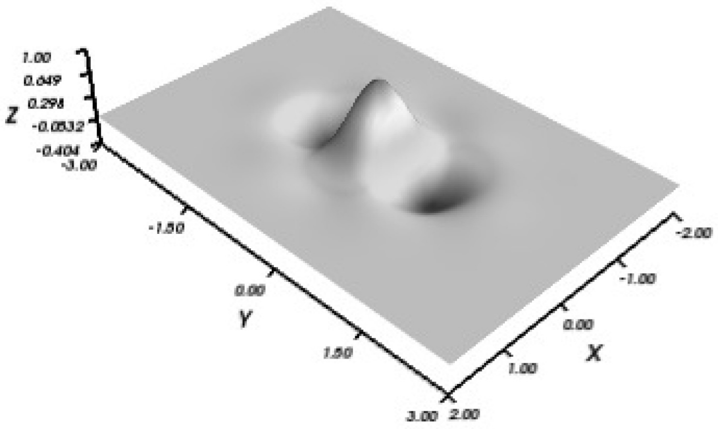

Figure 6.4 An example of a simple surface visualized using *mayavi*'s *mlab* module.

Figure 6.4 was drawn using the following code snippet

```
# Insert snippet B here to set up the data

from mayavi import mlab
fig=mlab.figure()
s=mlab.surf(xx,yy,zz,representation='surface')
ax=mlab.axes(line_width=2,nb_labels=5)
#mlab.title('Simple surface plot',size=0.4)
```

The new feature here is line 5, and perusal of the relevant docstring is worthwhile. Note in particular `'surface'` is the default, and replacing it with `'wireframe'` produces an alternative representation. Setting `warp_scale` to a float value will magnify (or contract) the z-axis.

Besides the interactive controls described in Section 6.3.2, there are other ways to obtain information about and to control the figure. Typing `mlab.view()` into the interpreter will produce six numbers. The first is the "azimuth" of the observer or "camera" direction, measured in degrees in the range $[0, 360]$. The second is the "elevation" in the range $[0, 180]$. The third is the distance of the observer, and the final three give the focal point of the camera, which by default is the origin. Any of these parameters can be set by invoking the function, e.g.,

```
mlab.view(azimuth=300,elevation=60)
```

Although interactive button 12 can be used to save the figure

```
mlab.savefig('foo.png')
```

will do the same job. If, like the author, you find it difficult to rotate the figure while keeping the z-axis vertical, this can be ameliorated by adding the lines

```
from tvtk.api import tvtk
fig.scene.interactor.interactor_style=tvtk.InteractorStyleTerrain()
```

after line 4 of the snippet.

6.5 A parametrically defined surface

We turn now to the visualization of Enneper's surface, defined in Section 6.1.1 by

$$x = u(1 - u^2/3 + v^2),\ y = v(1 - v^2/3 + u^2),\ z = u^2 - v^2 \quad \text{where } -2 \leqslant u, v \leqslant 2.$$

A code snippet, henceforth referred to as snippet C, to generate the data is

```
# This is snippet C
import numpy as np
[u,v]=np.mgrid[-2:2:51j, -2:2:61j]
x,y,z=u*(1-u**2/3+v**2),v*(1-v**2/3+u**2),u**2-v**2
```

Here the difference between the two plotting tools becomes significant.

6.5.1 Visualizing Enneper's surface using *mplot3d*

A code snippet to draw Figure 6.5 is

```
# Insert snippet C here
import matplotlib.pyplot as plt
```

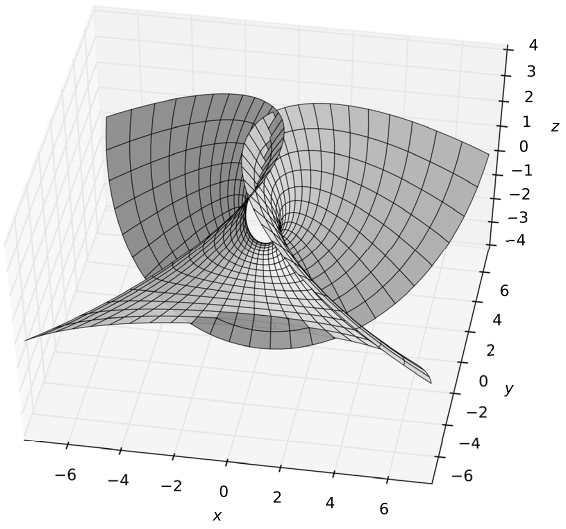

Figure 6.5 Enneper's surface visualized using *matplotlib*'s *mplot3d* module.

```
3  from mpl_toolkits.mplot3d import Axes3D
4
5  plt.ion()
6  fig=plt.figure()
7  ax=Axes3D(fig)
8  ax.plot_surface(x.T,y.T,z.T,rstride=2,cstride=2,color='r',
9                  alpha=0.2,linewidth=0.5)
10 ax.elev, ax.azim = 50, -80
11 ax.set_xlabel('x')
12 ax.set_ylabel('y')
13 ax.set_zlabel('z')
14 #ax.set_title('A parametric surface plot',
15 #                weight='bold',size=18)
```

The new features are all in line 8. Notice first that the underlying code is based on image-processing conventions, and to accord with *matplotlib* conventions we need to supply the transpose of the *x*-, *y*- and *z*-arrays. This is a non-trivial self-intersecting surface, and its visualization requires a judicious choice of the opacity, line width and stride parameters.

There is a great deal of computation going on here, which will tax even the fastest processors. In particular, interactive panning and tilting will be slowed down significantly.

6.5.2 Visualizing Enneper's surface using *mlab*

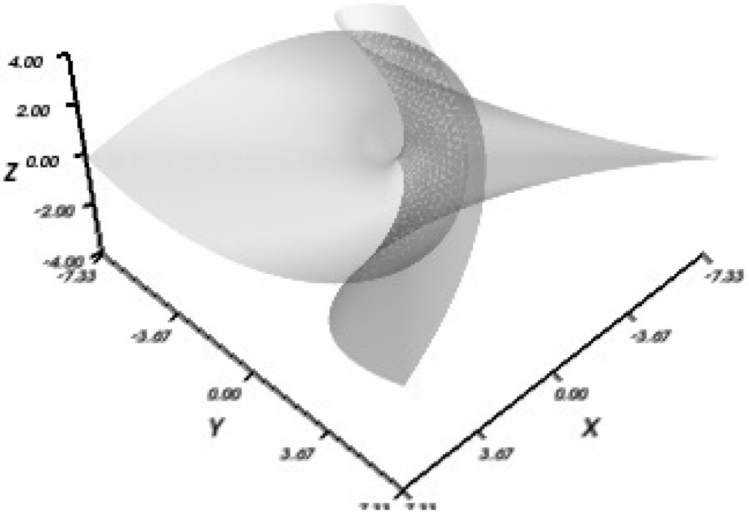

Figure 6.6 Enneper's surface visualized using *mayavi*'s *mlab* module.

The *mlab* module can also be used to visualize Enneper's surface. Figure 6.6 can be drawn with the code snippet

```
# Insert snippet C here to set up the data

from mayavi import mlab
fig=mlab.figure()
s=mlab.mesh(x,y,z,representation='surface',
            line_width=0.5,opacity=0.5)
mlab.axes(line_width=2,nb_labels=5)
#mlab.title('A parametrically defined surface',size=0.4)
```

Interactive panning and tilting is noticeably faster using *Mayavi's mlab* module, than using *mplot3d*. Why is this so? Well *matplotlib* and *mplot3d* use "vector graphics" which deal with shapes and are infinitely stretchable. *Mayavi* and *mlab* use "bitmapped graphics" which create and store each pixel of the picture. Its files are large but easy to create. The big disadvantage is that only a limited amount of expansion or contraction is possible without significant loss of visual quality. This why the function `mlab.savefig` only works satisfactorily with a bitmap format such as `png`.

6.6 Three-dimensional visualization of a Julia set

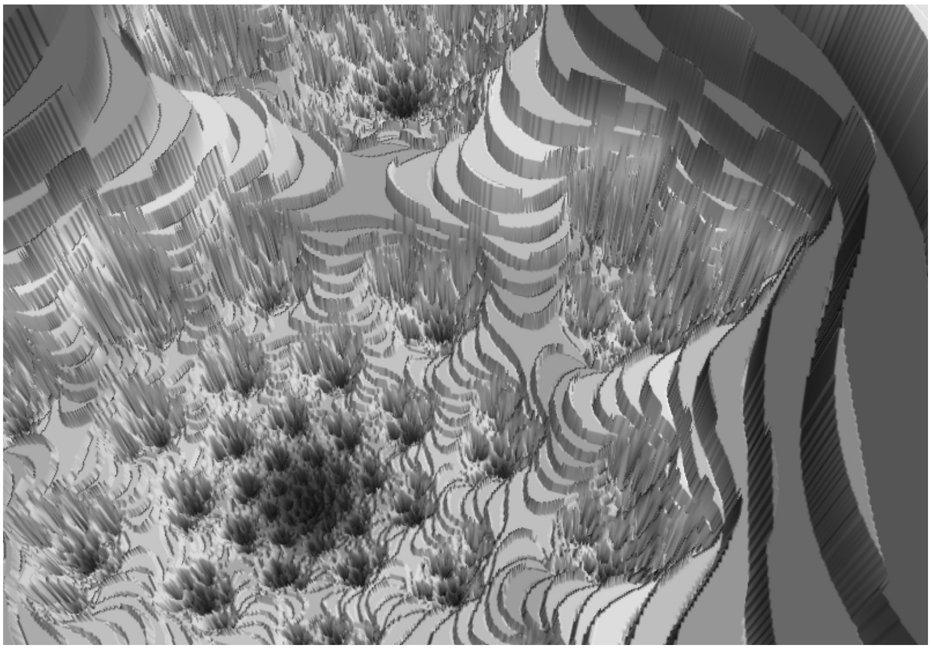

Figure 6.7 A Julia set for the mapping $z \to z^2 + c$ with $c = -0.7 - 0.4i$. Points near the top of the "canyon" correspond to trajectories which escaped to infinity early, while those at the bottom have yet to escape.

We turn now to the use of *mlab* to visualize figures that are more complicated. In Section 5.11, we sketched an introduction to fractal sets and showed how to construct the classic visualization of the Mandelbrot set for the mapping $z \to z^2 + c$ as realized by the iteration (5.1). Here we shall be concerned with the Julia set for the same mapping. Now we hold the parameter c fixed and let the iterates z_n depend on the initial value z_0. Then the escape parameter ϵ depends on z_0 rather than c, $\epsilon = \epsilon(z_0)$. Instead of the high-resolution, but fixed, image of Section 5.11, we look for an interactive, but lower-resolution picture, a so-called *canyon view*, as exemplified in the following self-contained snippet, which is adapted from the Mayavi documentation, Ramachandandran and Variquaux (2009), see also Mayavi Community (2011), and is not optimized for speed.

```
import numpy as np

# Set up initial grid
x,y=np.ogrid[-1.5:0.5:1000j,-1.0:1.0:1000j]
z=x+1j*y
julia = np.zeros(z.shape)
c=-0.7-0.4j
```

```
8
9   # Build the Julia set
10  for it in range(1,101):
11      z=z**2+c
12      escape=z*z.conj()>4
13      julia+=(1/float(it))*escape
14
15  from mayavi import mlab
16  mlab.figure(size=(800,600))
17  mlab.surf(julia,colormap='gist_ncar',
18            warp_scale='auto',vmax=1.5)
19  mlab.view(15,30,500,[-0.5,-0.5,2.0])
20  mlab.show()
```

Lines 1–7 are straightforward. The grid `julia` is set initially to zeros, and an arbitrary choice for c is made. In lines 10–13, we perform 100 iteration steps. `escape` is a grid of booleans which are `False` if the corresponding $|z_n| \leq 2$ and are `True` otherwise. For the escaped points, we increment `julia` by a small amount. Finally, lines 16–19 draw the figure. The parameters in lines 17–19 were chosen by trial and error. The output of this snippet is shown in Figure 6.7.

The reader is urged to try out this code, and then to make changes. The most passive way to do this is simply to change the code and then rerun it. More versatility can be obtained by using the cursor to modify the picture and button 13 to change values. Then typing, e.g., `mlab.view()` into the interpreter will reveal the current chosen values. Pressing button 1 brings up a *mayavi* window. The *mayavi* "pipeline" is shown upper left. Left clicking on any pipeline item will reveal the options chosen and offer the means to change them. With a little practice it's actually very easy and adaptable.

Obviously, in an introductory text we cannot survey all of the possibilities for three-dimensional graphics. In particular, we have not covered the important topic of animations. The reader is therefore urged to explore further the abilities of the *mlab* module. It is fast, versatile and easy to use, although it lacks the extremely high resolution of *matplotlib*.

7 Ordinary differential equations

7.1 Initial value problems

Consider first a single second-order differential equation, e.g., the *van der Pol equation*

$$\ddot{y} - \mu(1 - y^2)\dot{y} + y = 0, \qquad t \geqslant t_0, \tag{7.1}$$

for a function $y(t)$, with initial data

$$y(t_0) = y_0, \quad \dot{y}(t_0) = v_0. \tag{7.2}$$

Here μ, t_0, y_0 and v_0 are constants, $\dot{y} = dy/dt$, $\ddot{y} = d^2y/dt^2$. We can rewrite this in a standard first-order form as follows. Let

$$\mathbf{y} = \begin{pmatrix} y \\ \dot{y} \end{pmatrix}, \quad \mathbf{y}_0 = \begin{pmatrix} y_0 \\ v_0 \end{pmatrix}, \quad \mathbf{f}(\mathbf{y}) = \begin{pmatrix} \mathbf{y}[1] \\ \mu(1 - \mathbf{y}[0]^2)\mathbf{y}[1] - \mathbf{y}[0] \end{pmatrix}. \tag{7.3}$$

Then (7.1) and (7.2) combine to the standard form

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t), t), \qquad t \geqslant t_0, \qquad \mathbf{y}(t_0) = \mathbf{y}_0. \tag{7.4}$$

Because we specify sufficient conditions at an initial time $t = t_0$ to fix the solution, this is called an *initial value problem*. A very wide range of problems can be written in the standard form (7.4), where $\mathbf{y}$, $\mathbf{y}_0$ and $\mathbf{f}$ are s-vectors for some finite s.

A great deal of research has been devoted to the initial value problem (7.4). The classic text is Coddington and Levinson (1955). Useful reviews with an emphasis on numerical methods include Ascher et al. (1998), Butcher (2008) and Lambert (1992).

Based on this body of work Python offers, via the add-on module *scipy*, a "black box" package for the solution of the initial value problem (7.4). Black boxes offer convenience but also dangers. It is essential to understand something of how the black box works and hence what are its limitations, and how to influence its behaviour. Therefore, the next section sketches the basic ideas for the numerical integration of initial value problems.

7.2 Basic concepts

Here for simplicity we shall consider a single first-order equation for a function $y(t)$

$$\dot{y} = \lambda(y - e^{-t}) - e^{-t}, \qquad t \geqslant 0, \quad y(0) = y_0, \tag{7.5}$$

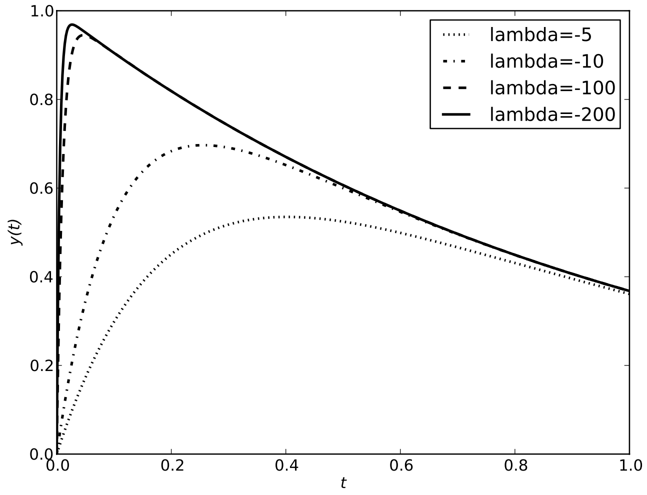

Figure 7.1 The exact solution of the model problem (7.5) with $y_0 = 0$, for various negative values of the parameter λ.

where the parameter λ is a constant. The exact solution is

$$y(t) = (y_0 - 1)e^{\lambda t} + e^{-t}. \tag{7.6}$$

We shall concentrate on the initial condition $y_0 = 0$, and consider non-positive values for λ. If $\lambda = 0$, the exact solution is $y(t) = -1 + e^{-t}$, and for $\lambda = -1$ the solution is $y(t) = 0$. Figure 7.1 shows the solution for a few negative values for λ. As $|\lambda|$ increases, the solution rises rapidly from 0 at $t = 0$ to $O(1)$ when $t = O(|\lambda|^{-1})$ and then decays like e^{-t}.

In order to solve this equation numerically for say $0 \leqslant t \leqslant 1$, we introduce a discrete grid. Choose some large integer N, and set $h = 1/N$. Define

$$t_n = n/N, \qquad n = 0, 1, 2, \ldots, N.$$

We have chosen an equidistant grid for simplicity, but in practice this is not essential. Let $y_n = y(t_n)$. Thus $y(t)$ is represented on this grid by the sequence $\{y_0, y_1, y_2, \ldots, y_N\}$. Let the corresponding numerical approximation be denoted $\{Y_0, Y_1, Y_2, \ldots, Y_N\}$.

Perhaps the simplest approximation scheme is the *forward Euler* one

$$Y_0 = y_0, \quad Y_{n+1} = Y_n + h(\lambda Y_n - (\lambda + 1)e^{-t_n}), \quad n = 0, 1, 2, \ldots, N - 1, \tag{7.7}$$

which corresponds to retaining only the first two terms in a Taylor series. Indeed, the *single step error* or *local truncation error* is

$$\tau_n = \frac{y_{n+1} - y_n}{h} - (\lambda y_n - (\lambda + 1)e^{-t_n}) = \tfrac{1}{2}h\ddot{y}(t_n) + O(h^2). \tag{7.8}$$

By choosing h sufficiently small (N sufficiently large), we can make the single step error as small as we want. However, we are really interested in the *actual error* $E_n = Y_n - y_n$. Using (7.7) and (7.8) to eliminate Y_{n+1} and y_{n+1} respectively, we can easily obtain

$$E_{n+1} = (1 + h\lambda)E_n - h\tau_n. \tag{7.9}$$

This is a recurrence relation whose solution for $n > 0$ is

$$E_n = (1 + h\lambda)^n E_0 - h\sum_{m=1}^{n}(1 + h\lambda)^{n-m}\tau_{m-1}. \tag{7.10}$$

(Equation (7.10) is easy to prove by induction.)

Now as n increases, $|E_n|$ stays bounded only if $|1 + h\lambda| \leqslant 1$, otherwise it grows exponentially. Thus there is a stability criterion[1]

$$|1 + h\lambda| \leqslant 1 \tag{7.11}$$

that has to be satisfied if forward Euler is to be numerically satisfactory. For λ close to -1, any reasonable value of h will produce acceptable results. But consider the case $\lambda = -10^6$. Stability requires $h < 2 \times 10^{-6}$. This is understandable when computing the rapid initial transient, but the same incredibly small steplength is required over the rest of the range where the solution appears to be decaying gently like e^{-t} (see Figure 7.1), and so the numerical evolution proceeds incredibly slowly. This phenomenon is called *stiffness* in the problem (7.5). Clearly, forward Euler is unsatisfactory on stiff problems.

In fact, forward Euler has a number of other defects. Consider, e.g., simple harmonic motion $\ddot{y}(t) + \omega^2 y(t) = 0$ with ω real. Reducing this to standard form, (7.4) gives

$$\dot{\mathbf{y}}(t) = A\mathbf{y}(t), \qquad \text{where } \mathbf{y} = \begin{pmatrix} y \\ \dot{y} \end{pmatrix}, \text{ and } A = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}. \tag{7.12}$$

The stability criterion (7.11) now has to apply to each of the eigenvalues λ of the matrix A. Here $\lambda = \pm i\omega$, and so forward Euler is always unstable on such problems.

Forward Euler has the property of being *explicit*, i.e., Y_{n+1} is given explicitly in terms of known data. Numerical analysts have built an extensive theory of explicit numerical schemes which produce much smaller single step errors and have less stringent stability criteria. In particular, by using simultaneously several methods each with different accuracies it is possible to estimate the *local truncation error*, and if we have specified the accuracy in the solution that we require (see later), then we can locally change the step size h to keep it as large as possible (for the sake of efficiency), while maintaining accuracy and stability. This is called *adaptive time-stepping*. However, these explicit methods are all ineffective on stiff problems.

In order to deal with the stiffness problem, we consider briefly the *backward Euler scheme*

$$Y_{n+1} = Y_n + h[\lambda Y_{n+1} - (\lambda + 1)e^{-t_n}], \qquad n = 0, 1, 2, \ldots, N - 1,$$

or

$$Y_{n+1} = (1 - h\lambda)^{-1}[Y_n - h(\lambda + 1)e^{-t_n}], \qquad n = 0, 1, 2, \ldots, N - 1. \tag{7.13}$$

[1] There are a number of different stability criteria in the literature. This one corresponds to 0-*stability*.

We can repeat the earlier analysis to show that, with the obvious modification to τ_n in equation (7.8), the single step error $\tau_n = O(h)$ again, and the actual error satisfies

$$E_{n+1} = (1 - h\lambda)^{-1}(E_n - h\tau_n),$$

with solution

$$E_n = (1 - h\lambda)^{-n}E_0 - h\sum_{m=0}^{n-1}(1 - h\lambda)^{m-n}\tau_m.$$

The stability criterion is now

$$|1 - h\lambda| > 1.$$

This is clearly satisfied for our stiff problem with large negative λ for all positive values of h.

For linear systems such as (7.12), we need to replace the factor $1 - h\lambda$ in (7.13) by $I - hA$, a $s \times s$ matrix which needs to be inverted, at least approximately. For this reason, backward Euler is called an *implicit scheme*. Again numerical analysts have built an extensive theory of implicit schemes which are primarily used to deal with stiff problems.

In this brief introduction, we have dealt with only the simplest linear cases. In general the function $\mathbf{f}(\mathbf{y}, t)$ of problem (7.4) will be non-linear, and its Jacobian (matrix of partial derivatives with respect to $\mathbf{y}$) takes the place of the matrix A. A further complication is that stiffness (if it occurs) may depend on the actual solution being sought. This poses a significant challenge to black box solvers.

7.3 The `odeint` function

It is quite difficult to write an effective efficient black box solver for the initial value problem, but a few well-tried and trusted examples exist in the literature. One of the best-known ones is the Fortran code *lsoda*, an integrator developed at Lawrence Livermore National Laboratory, as part of the *odepack* package. This switches automatically between stiff and non-stiff integration routines, depending on the characteristics of the solution, and does adaptive time-stepping to achieve a desired level of solution accuracy. The `odeint` function in the *scipy.integrate* module is a Python wrapper around the Fortran code for *lsoda*. Before we discuss how to use it, we need to understand a few of the implementation details.

7.3.1 Theoretical background

Suppose we are trying to solve numerically the standard problem (7.4). We will certainly need a Python function `f(y,t)`, which returns the right-hand side as an array of the same shape as `y`. Similarly we will need an array `y0` which contains the initial data. Finally, we need to supply `tvals` an array of parameter t-values for which we would like the corresponding `y`-values returned. Note that the first entry, `tvals[0]`, should be the t_0 of (7.4). Then the code

```
y=odeint(f,y0,tvals)
```

will return an approximate solution in `y`. However, a large number of significant details are being glossed over here.

Perhaps the first question is what step length (or lengths) h should be used, and how does this relate to `tvals`? Well the step length h that is chosen at each point is constrained by the need to maintain accuracy, which leads to the question of how is the accuracy to be specified? The function `odeint` tries to estimate the local error E_n. Perhaps the simplest criterion is to choose some small value of *absolute error* ϵ_{abs} and to require

$$|E_n| < \epsilon_{abs},$$

and to adjust the step length h accordingly. However, if Y_n becomes very large this criterion may lead to very small values of h and so become very inefficient. A second possibility is to define a *relative error* ϵ_{rel} and to require

$$|E_n| < |Y_n|\epsilon_{rel},$$

but this choice runs into problems if $|Y_n|$ becomes small, e.g., if Y_n changes sign. Thus the usual criterion is to require

$$|E_n| < |Y_n|\epsilon_{rel} + \epsilon_{abs}. \tag{7.14}$$

Here we have assumed that Y_n is a scalar. Very little changes if Y_n is an array and all of its components take comparable values. If this is not the case, then we would require ϵ_{abs} and ϵ_{rel} to be arrays. In `odeint`, these quantities are keyword arguments called `atol` and `rtol` respectively and can be scalars or arrays (with the same shape as `y`). The default value for both is a scalar, about 1.5×10^{-8}. Thus the package attempts to integrate from `tvals[0]` to a final value not less than `tvals[-1]` using internally chosen steps to try to satisfy at least one of these criteria at each step. It then constructs the `y`-values at the `t`-values specified in `tvals` by interpolation and returns them.

We should recognize that `atol` and `rtol` refer to local one step errors, and that the global error may be much larger. For this reason, it is unwise to choose these parameters to be so large that the details of the problem are poorly approximated. If they are chosen so small that `odeint` can satisfy neither of the criteria, a run time error will be reported.

As stated above, `odeint` can cope automatically with equations or systems which are or become stiff. If this is a possibility, then it is strongly recommended to supply the Jacobian of $\mathbf{f}(\mathbf{y}, t)$ as a function say `jac(y,t)` and to include it with the keyword argument `Dfun=jac`[2]. If it is not supplied, then `odeint` will try to construct one by numerical differentiation, which can be potentially dangerous in critical cases.

For a complete list of keyword arguments, the reader should consult the docstring for `odeint`. There are however two more arguments which are commonly used. Suppose the function `f` depends on parameters, e.g., `f(y,t,alpha,beta)`, and similarly

[2] `jac` requires a $s \times s$ matrix, which might be large and sparse. In Section 4.8.1, we pointed out a space-efficient method for specifying such objects. The function `odeint` can use such methods. See the docstring for details.

for jac. The function odeint needs to be told this as a keyword argument specifying a *tuple*, e.g., args=(alpha,beta). Finally, while developing experience with a new package it is very helpful to be able to find out what went on inside the package. The command

```
y,info=odeint(f,y0,tvals,full_output=True)
```

will generate a *dictionary* info giving a great deal of output information. See the docstring for details.

7.3.2 Practical usage

To see how easy it is to use odeint, consider the ultra-simple problem

$$\dot{y}(t) = y(t), \qquad y(0) = 1,$$

and suppose we want the solution at $t = 1$. The following code generates this,

```
import numpy as np
from scipy.integrate import odeint
odeint(lambda y,t:y,1,[0,1])
```

which generates [[1.],[2.71828193]] as expected. Here we have used an anonymous function (see Section 3.8.7) to generate $f(y,t) = y$ as the first argument. The second is the initial y-value. The third and final argument is a *list* consisting of the (mandatory) initial t-value and the final t-value, which is coerced implicitly to a *numpy* array of floats. The output consists of the t- and y-arrays with the initial values omitted.

Here is a simple example of a system of equations. The problem is simple harmonic motion

$$\ddot{y}(t) + \omega^2 y(t) = 0, \qquad y(0) = 1,\ \dot{y}(0) = 0,$$

to be solved and plotted for $\omega = 2$ and $0 \leqslant t \leqslant 2\pi$. We first rewrite the equation as a system

$$\mathbf{y} = (y, \dot{y})^T, \quad \dot{\mathbf{y}}(t) = (\dot{y}, -\omega^2 y)^T.$$

The following complete code snippet will produce an undecorated picture of the solution. (In real-life problems we would want to decorate the figure, following the methods in Chapter 5.)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def rhs(Y,t,omega):
    y,ydot=Y
    return ydot,-omega**2*y

```

```
9  t_arr=np.linspace(0,2*np.pi,101)
10 y_init=[1, 0]
11 omega=2.0
12 y_arr=odeint(rhs,y_init,t_arr,args=(omega,))
13 y,ydot=y_arr[:,0],y_arr[:,1]
14 plt.ion()
15 plt.plot(t_arr,y,t_arr,ydot)
```

Four points should be noted here. The first is that in line 6 within the function `rhs`, we have "unwrapped" the vector argument for the sake of clarity in the return values. Secondly, in line 7 we have returned a *tuple* which will silently be converted to an array. Thirdly, see line 12, in `odeint` the argument `args` must be a *tuple*, even if there is only one value. The reason for this is explained in Section 3.5.5. Finally, the output is packed as a two-dimensional array, and line 13 unpacks the solution arrays.

When, as here, the motion is *autonomous*, i.e., no explicit t-dependence, it is often illuminating to produce a *phase plane portrait*, plotting `ydot` against `y` and, temporarily ignoring the t-dependence. This is easily achieved with a simple addition

```
plt.figure() # to set up a new canvas
plt.plot(y,ydot)
plt.title("Solution curve when omega = %5g" % omega)
```

The function `plt.figure` in line 1 generates a new figure window. This was explained in Section 5.9.1.

However, with a little sophistication we can do a great deal more. First, we can create a grid in phase space and draw direction fields, the vector $\dot{\mathbf{y}}(t)$ at each point of the grid. Secondly, we can draw the solution curve starting from an arbitrary point of the figure, i.e., arbitrary initial conditions. The first step is achieved with the following code, which should be added at the end of the previous numbered snippet.

```
1 plt.figure()
2 y,ydot=np.mgrid[-3:3:21j,-6:6:21j]
3 u,v=rhs(np.array([y,ydot]),0.0,omega)
4 mag=np.hypot(u,v)
5 mag[mag==0]=1.0
6 plt.quiver(y,ydot,u/mag,v/mag,color='red')
```

In line 2, we choose $(y, ydot)$ coordinates to cover a relatively coarse grid in phase space. (Note that the arrays `y` and `ydot` created in line 13 of the original snippet are now lost.) In line 3, we compute the components (u, v) of the tangent vector field at each point of this grid. A line like `plt.quiver(y,ydot,u,v)` will then draw these values as little arrows whose magnitude and direction are those of the vector field and whose base is the grid point. However, near an equilibrium point ($u = v = 0$) the arrows end up as uninformative dots. In order to alleviate this, line 4 computes the magnitude $\sqrt{u^2 + v^2}$ of each vector, and line 6 plots the normalized unit vector field. Line 5 ensures that no "division by zero" takes place. The docstring for `plt.quiver` offers plenty of scope for further decoration.

The next snippet draws trajectories with arbitrary initial conditions on this phase plane, and should be adjoined to the one above.

```
1  # Enable drawing of arbitrary number of trajectories
2  print "\n\n\nUse mouse to select each starting point."
3  print "Timeout after 30 seconds"
4  choice=plt.ginput()
5  while len(choice)>0 :
6      y01=np.array([choice[0][0],choice[0][1]])
7      y= odeint(rhs,y01,t_arr,args=(omega,))
8      plt.plot(y[:, 0],y[:, 1],lw=2)
9      choice=plt.ginput()
10 print "Timed out!"
```

Line 4 shows the simple use of a very versatile function. The `ginput` function waits for n mouse clicks, here the default $n = 1$, and returns the coordinates of each point in the array `choice`. We then enter a **`while`** loop in line 5, and set `y01` to be the initial data corresponding to the clicked point in line 6. Now line 7 computes the solution, and line 8 plots it. The programme then waits, line 9, for further mouse input. The default "timeout" for `ginput` is 30 seconds, and so if no input is given, line 10 will be reached eventually. If the reader wishes to use this approach proactively, then it is highly worthwhile to embellish the bare-bones code presented here.

For a first example of a non-linear problem, we turn to the *van der Pol* equation (7.1), (7.2) in the first-order form (7.4), (7.3). This is often used in the literature as part of a testbed for numerical software. We follow convention and set initial conditions via $\mathbf{y}_0 = (2, 0)^T$ and consider various values of the parameter μ. Note that if $\mu = 0$, we have the previous example of simple harmonic motion with period $\tau = 2\pi$. If $\mu > 0$, then for any initial conditions the solution trajectory tends rapidly to a limit cycle, and analytic estimates of the period give

$$\tau = \begin{cases} 2\pi(1 + O(\mu^2)) & \text{as } \mu \to 0, \\ \mu(3 - 2\log 2) + O(\mu^{-1/3}) & \text{as } \mu \to \infty. \end{cases} \tag{7.15}$$

The rapid relaxation to a periodic orbit suggests that this example could well become stiff. Recalling the right-hand side vector from (7.3), we compute the Jacobian matrix

$$\mathbf{J}(\mathbf{y}) = \begin{pmatrix} 0 & 1 \\ -2\mu\mathbf{y}[0]\mathbf{y}[1] - 1 & \mu(1 - \mathbf{y}[0]^2) \end{pmatrix}. \tag{7.16}$$

We set up next a Python script to integrate this equation, which extends the one used earlier to integrate simple harmonic motion.

```
import numpy as np
from scipy.integrate import odeint

```

```
 4  def rhs(y,t,mu):
 5      return [ y[1], mu*(1-y[0]**2)*y[1]-y[0] ]
 6
 7  def jac(y,t,mu):
 8      return [ [0, 1],[-2*mu*y[0]*y[1]-1, mu*(1-y[0]**2)] ]
 9
10  mu=1
11  t=np.linspace(0,30,1001)
12  y0=np.array([2.0,0.0])
13  y,info=odeint(rhs,y0,t,args=(mu,),Dfun=jac,full_output=True)
14
15  print " mu = %g, number of Jacobian calls is %d" % \
16              (mu, info['nje'][-1])
```

Notice that the call to `odeint` in line 13 takes two further named arguments. The first `Dfun` tells the integrator where to find the Jacobian function. The second sets `full_output`. While the integrator is running, it collects various statistics which can aid understanding of what precisely it is doing. If `full_output` is set to `True` (the default is `False`), then these data are made accessible as a *dictionary* via the second identifier on the left-hand side of line 13. For a complete list of available data, the reader should consult the docstring of the function `odeint`. Here we have chosen to display the “number of Jacobian evaluations” which is available as `info['nje']`. For small to moderate values of the parameter μ (and default values for the error tolerances), the integrator does not need to use implicit algorithms, but they are needed for $\mu > 15$.

Of course, the visualization scripts introduced earlier can be used in this context. The reader is encouraged to experiment here, either to become familiar with the *van der Pol* equation, or `odeint`’s capabilities, or both. Note that increasing the parameter μ implies increasing the period, see equation (7.15), and so increasing the final time, and the number of points to be plotted.

For our final example illustrating the capabilities of the `odeint` function, we turn to the *Lorenz* equations. This non-linear system arose first as a model for the earth’s weather which exhibited *chaotic* behaviour, and has been studied widely in a variety of contexts. There are many popular articles, and at a slightly more advanced level Sparrow (1982) gives a comprehensive introduction. The unknowns are $x(t)$, $y(t)$ and $z(t)$, and the equations are

$$\dot{x} = \sigma(y - x), \qquad \dot{y} = \rho x - y - xz, \qquad \dot{z} = xy - \beta z, \tag{7.17}$$

where β, ρ and σ are constants, and we specify initial conditions at say $t = 0$. Lorenz originally studied the case where $\sigma = 10$ and $\beta = 8/3$ and this practice is widely followed, allowing only ρ to vary. For smallish values of ρ, the behaviour of solutions is predictable, but once $\rho > \rho_H \approx 24.7$ the solutions become aperiodic. Further they exhibit very sensitive dependence on initial conditions. Two solution trajectories corresponding to slightly different initial data soon look quite different. The following self-contained code snippet can be used to investigate the aperiodicity.

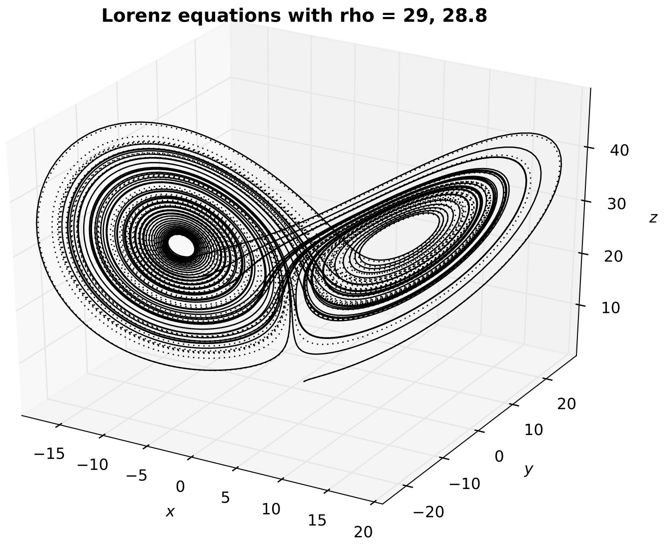

Figure 7.2 Two solution curves for the Lorenz equations, whose ρ parameter varies slightly. The solid and dotted regions show where the solutions differ markedly pointwise. However, the "butterfly structure" appears to be stable.

```
import numpy as np
from scipy.integrate import odeint

def rhs(u,t,beta,rho,sigma):
    x,y,z = u
    return [sigma*(y-x), rho*x-y-x*z, x*y-beta*z]

sigma=10.0
beta=8.0/3.0
rho1=29.0
rho2=28.8

u01=[1.0,1.0,1.0]
u02=[1.0,1.0,1.0]

t=np.linspace(0.0,50.0,10001)
u1=odeint(rhs,u01,t,args=(beta,rho1,sigma))
u2=odeint(rhs,u02,t,args=(beta,rho2,sigma))

x1,y1,z1=u1[:, 0],u1[:, 1],u1[:, 2]
x2,y2,z2=u2[:, 0],u2[:, 1],u2[:, 2]

```

```
23 import matplotlib.pyplot as plt
24 from mpl_toolkits.mplot3d import Axes3D
25 
26 plt.ion()
27 fig=plt.figure()
28 ax=Axes3D(fig)
29 ax.plot(x1,y1,z1,'b-')
30 ax.plot(x2,y2,z2,'r:')
31 ax.set_xlabel('x')
32 ax.set_ylabel('y')
33 ax.set_zlabel('z')
34 ax.set_title('Lorenz equations with rho = %g, %g' % (rho1,rho2))
```

Here x, y and z are packaged as a three-vector $\mathbf{u}$. However, for clarity they are unpacked locally in lines 5, 20 and 21. Two solution trajectories whose ρ parameters vary only slightly are plotted as solid and dotted curves in Figure 7.2.

Notice that if $\rho > 1$, there are equilibrium points (zero velocity) at

$$(0,0,0), \quad (\pm\sqrt{\beta(\rho-1)}, \pm\sqrt{\beta(\rho-1)}, \rho-1),$$

while if $0 < \rho < 1$ the only equilibrium point is the origin. Although the solution does not approach any equilibrium point in particular, it does seem to wind its way around the two non-trivial equilibrium points. From nearby one of these points, the solution spirals out slowly. When the radius gets too large, the solution jumps into a neighbourhood of the other equilibrium, where it begins another outward spiral, and the process repeats itself. The resulting picture looks somewhat like the wings of a butterfly. See again Figure 7.2.

Note that there are significant regions of only solid or only dotted curves. The two solution curves although initially close, diverge pointwise as time increases. This is easier to see in a colour plot. The "butterfly structure" however remains intact. Much of our knowledge of the Lorenz and related systems is based on numerical experiments. There is a detailed discussion in Sparrow (1982).

7.4 Two-point boundary value problems

7.4.1 Introduction

Here is a very simple model for a two-point boundary value problem

$$y''(x) = f(x, y, y') \quad \text{for } a < x < b \quad \text{with } y(a) = A,\ y(b) = B. \tag{7.18}$$

(We have changed the independent variable from t to x to accord with historical convention.) As we shall see, this is a global problem and so much harder than the initial value problem of Section 7.1, which is local. Existence theorems are much harder to formulate, let alone prove, and numerical treatments are not so straightforward as those for the earlier case.

Consider, e.g., the simple linear example

$$y''(x) + h(x)y(x) = 0, \quad 0 < x < 1, \text{ with } y(0) = A, \; y(1) = B. \tag{7.19}$$

Because of the linearity, we might choose to use a "shooting approach". We first solve two initial value problems (see Section 7.1)

$$\begin{aligned} y_1''(x) + h(x)y_1(x) &= 0, \quad x > 0, \quad y_1(0) = 1, \; y_1'(0) = 0, \\ y_2''(x) + h(x)y_2(x) &= 0, \quad x > 0, \quad y_2(0) = 0, \; y_2'(0) = 1. \end{aligned}$$

Linearity implies that the general solution is $y(x) = C_1 y_1(x) + C_2 y_2(x)$ where C_1 and C_2 are constants. We satisfy the boundary condition at $x = 0$ by requiring $C_1 = A$.

Now the boundary condition at $x = 1$ requires $Ay_1(1)+C_2 y_2(1) = B$. If $y_2(1) \neq 0$, then there is a unique solution for C_2, and the problem has a unique solution. However, if $y_2(1) = 0$, then there are infinitely many solutions for C_2 if $Ay_1(1) = B$ and no solutions at all otherwise.

We should note from this that the existence and uniqueness of solutions of boundary value problems is far less clear cut than for the initial value problem, and depends in an intrinsic way on the behaviour of the solution throughout the integration interval. We need a new approach. The textbook Ascher et al. (1995) offers a careful balance between theory and application, and is recommended for investigating further the background concepts of this topic. A selection of that material, treated at a more elementary level, can be found in Ascher et al. (1998).

7.4.2 Formulation of the boundary value problem

We now set up a problem for an array of dependent variables

$$\mathbf{Y}(x) = (y_0(x), y_1(x), \ldots, y_{d-1}(x))$$

of dimension d. We shall not require the differential equations to be written as a first-order system. Instead, we assume that the system can be written in the form

$$y_i^{(m_i)}(x) = f_i, \quad a < x < b, \quad 0 \leqslant i < d,$$

where the right-hand sides have to be specified. In other words, $y_i(x)$ is determined by an equation for its m_ith derivative. In order to be more precise, we introduce an augmented array of the dependent variables and all lower-order derivatives

$$\mathbf{Z}(x) = \mathbf{Z}(\mathbf{Y}(x)) = \left(y_0, y_0', \ldots, y_0^{(m_0-1)}, y_1, y_1', \ldots, y_{d-1}, y_{d-1}', \ldots, y_{d-1}^{(m_{d-1})}\right)$$

of dimension $N = \sum_{i=0}^{d-1} m_i$. Then our system of differential equations is of the form

$$y_i^{(m_i)}(x) = f_i(x, \mathbf{Z}(\mathbf{Y}(x))), \quad a < x < b, \quad 0 \leqslant i < d. \tag{7.20}$$

If we were to write (7.20) as a first-order system, then it would have dimension N.

Next we need to set up the N boundary conditions. We impose them at points $\{z_j\}$, $j = 0, 1, \ldots, N-1$ where $a \leqslant z_0 \leqslant z_1 \leqslant \ldots \leqslant z_{N-1} \leqslant b$. Each of the N boundary conditions is required to be of the form

$$g_j(z_j, \mathbf{Z}(\mathbf{Y}(z_j))) = 0, \quad j = 0, 1, \ldots, N-1. \tag{7.21}$$

See below for concrete examples.

There is a restriction here: the boundary conditions are said to be *separated*, i.e., condition g_j depends only on values at z_j. It is not difficult to see that many more general boundary conditions can be written in this form. As a simple example, consider the problem

$$u''(x) = f(x, u(x), u'(x)), \quad 0 < x < 1, \quad \text{with } u(0) + u(1) = 0,\ u'(0) = 1,$$

which includes a non-separated boundary condition. In order to handle this, we adjoin a trivial differential equation to the system, $v'(x) = 0$ with $v(0) = u(0)$. The system now has two unknowns $\mathbf{Y} = (u, v)$ and is of order 3 and the three separated boundary conditions for $\mathbf{Z} = (u, u', v)$ are $u'(0) = 1$, $u(0) - v(0) = 0$ and $u(1) + v(1) = 0$ at $z_0 = 0$, $z_1 = 0$ and $z_2 = 1$ respectively.

This trick can be used to deal with unknown parameters and/or normalization conditions. Again consider a simple-at-first-glance eigenvalue example

$$u''(x) + \lambda u(x) = 0, \quad u(0) = u(1) = 0, \text{ with } \int_0^1 u^2(s)\,ds = 1. \tag{7.22}$$

We introduce two auxiliary variables $v(x) = \lambda$ and $w(x) = \int_0^x u^2(s)\,ds$. Thus our unknowns are $\mathbf{Y} = (u, v, w)$ with $\mathbf{Z} = (u, u', v, w)$. There are three differential equations of orders 2, 1, 1

$$u''(x) = -u(x)v(x), \quad v'(x) = 0, \quad w'(x) = u^2(x), \tag{7.23}$$

and four separated boundary conditions

$$u(0) = 0, \quad w(0) = 0, \quad u(1) = 0, \quad w(1) = 1. \tag{7.24}$$

Thus a large class of boundary value problems can be coerced into our standard form (7.20) and (7.21).

The example above shows another facet of the boundary value problem. It is easy to see that problem (7.22) has a countably infinite set of solutions $u_n(x) = \sqrt{2}\sin(n\pi x)$ with $\lambda = \lambda_n = (n\pi)^2$ for $n = 1, 2, \ldots$. Although the u-equation is linear in u, it is non-linear in $\mathbf{Y}$, and the eigenvalues are the solutions of a non-linear equation, here $\sin(\sqrt{\lambda}) = 0$. Note also that non-linear equations are not in general soluble in closed form, and can have many solutions. In such situations, solution techniques usually involve iterative methods, and we need to be able to specify which solution we are interested in. For example we might supply a more or less informed guess as to the unknown solution's behaviour.

Shooting methods “guess” starting values for initial value problems. However, and especially for non-linear problems, the wrong guess frequently produces trial solutions which blow up before the distant boundary is reached. Thus we need to consider also different techniques for the numerical solution of boundary value problems. We are considering the interval $a \leqslant x \leqslant b$, and so we represent this numerically by a discrete grid $a \leqslant x_0 \leqslant x_1 \leqslant \ldots \leqslant x_{M-1} \leqslant b$. This grid might, perhaps, include the grid of boundary points $\{z_j\}$ introduced above. Next we face two related issues: (i) how do we represent the unknown functions on the grid, and (ii) how do we approximate the differential equations and boundary conditions on the grid?

Clearly, choices which are tailored to one particular problem may be less than optimal for another. Given the diversity of problems that are being considered, many may choose to select the method apposite to their problem. This assumes that they know, or can get good advice on, the optimal choice. For others, a black box solution, while very rarely providing the optimal solution, may generate a reliable answer quickly, and that is the approach we adopt here. There are many black box packages described in the literature, usually available either as Fortran or C++ packages. One of the more widely respected packages is COLNEW, Bader and Ascher (1987), which is described extensively in appendix B of Ascher et al. (1995). The Fortran code has been given a Python wrapper in the *scikit* package `scikits.bvp1lg`. (*Scikit* packages were discussed in Section 4.9.2.) Its installation, which requires an accessible Fortran compiler, is described in Section A.3.

7.4.3 A simple example

Here is a simple example of a two-point boundary value problem

$$u''(x) + u(x) = 0, \qquad u(0) = 0, \quad u'(\pi) = 1, \quad 0 \leqslant x \leqslant \pi, \tag{7.25}$$

for which the exact solution is $u(x) = -\sin x$. The following code snippet can be used to obtain and plot the numerical solution.

```
import numpy as np
import scikits.bvp1lg.colnew as colnew

degrees=[2]
boundary_points=np.array([0,np.pi])
tol=1.0e-8*np.ones_like(boundary_points)

def fsub(x,Z):
    """The equations"""
    u,du=Z
    return np.array([-u])

def gsub(Z):
    """The boundary conditions"""
    u,du=Z
    return np.array([u[0],du[1]-1.0])

solution=colnew.solve(boundary_points,degrees,fsub,gsub,
                      is_linear=True,tolerances=tol,
                      vectorized=True,maximum_mesh_size=300)

```

```
22 import matplotlib.pyplot as plt
23 plt.ion()
24 x=solution.mesh
25 u_exact=-np.sin(x)
26 plt.plot(x,solution(x)[:,0],'b.')
27 plt.plot(x,u_exact,'g-')
```

Here line 2 imports the boundary value problem solver package `colnew`. There is precisely one differential equation of degree 2 and this is specified by the Python *list* in line 4. There are two boundary points, and these are given as a *list* coerced to an array in line 5. We need to specify an array of error tolerances that we will permit for each of the boundary points, and an arbitrary choice for this is made in line 6.

We write the single equation as $u''(x) = -u(x)$ and we specify the right-hand side in the function `fsub`. Its arguments are x and the enlarged vector of dependent variables $Z = \{u, u'\}$, which (for convenience) we unpack in line 10. Note that we must return the single value $-u$ as an array, which involves first packing it into a *list*, line 11, to ensure that when `fsub` is called by `colnew` with arrays as arguments then the output is an array of the appropriate size. Similar considerations apply to the boundary conditions which we first rewrite as $u(0) = 0$ and $u'(\pi) - 1 = 0$. The function `gsub` returns them as an array coerced from a *list* in line 16. Note that the array subscripts refer to the points specified in `boundary_points`. Thus `u[0]` refers to $u(0)$ but `du[1]` refers to $u'(\pi)$. It is important to get the syntax right for these functions, for otherwise the error messages are somewhat obscure.

After this preliminary work, we call the function `colnew.solve` in line 18. The first four arguments are mandatory. The remainder are keyword arguments, and the ones we have used here should suffice for simple problems. For a full list of the optional arguments we should examine the function's docstring in the usual way. The remaining lines show how to use the output from `colnew.solve` to which we assigned the identifier `solution`. The array of x-values for which the solution was determined is stored as `solution.mesh`, to which we assign the identifier `x` in line 24. Now the enlarged solution vector $Z = \{u, u'\}$ is available as the two-dimensional array `solution(x)[ , ]`. The second argument specifies the Z-component (u or du), while the first labels the point at which it is computed. Thus the second argument of `plt.plot` in line 26 returns a vector of u-values.

This is clearly a great deal of effort for such a simple problem. Fortunately, the work load hardly changes as the difficulty increases, as the next two examples show.

7.4.4 A linear eigenvalue problem

We now treat in more detail the eigenvalue problem (7.22) in the form (7.23) with (7.24). It is well known that there is a countable infinity of solutions for problems such as (7.22), and so as an example we ask for the third eigenvalue. Because this is a Sturm–Liouville problem, we know that the corresponding eigenfunction will have two roots inside the interval as well as roots at the end points. We therefore suggest a suitable quartic polynomial as an initial guess, since we want to approximate the third eigenfunction. Here is a code snippet which solves the problem. Again for the sake of brevity we have left the figure undecorated, which is not good practice.

```
import numpy as np
import scikits.bvp1lg.colnew as colnew

degrees=[2,1,1]
boundary_points=np.array([0.0, 0.0, 1.0, 1.0])
tol=1.0e-5*np.ones_like(boundary_points)

def fsub(x,Z):
    """The equations"""
    u,du,v,w=Z
    ddu=-u*v
    dv=np.zeros_like(x)
    dw=u*u
    return np.array([ddu,dv,dw])

def gsub(Z):
    """The boundary conditions"""
    u,du,v,w =Z
    return np.array([u[0],w[1],u[2],w[3]-1.0])


guess_lambda=100.0
def guess(x):
    u=x*(1.0/3.0-x)*(2.0/3.0-x)*(1.0-x)
    du=2.0*(1.0-2.0*x)*(1.0-9.0*x+9.0*x*x)/9.0
    v=guess_lambda*np.ones_like(x)
    w=u*u
    Z_guess=np.array([u,du,v,w])
    f_guess=fsub(x,Z_guess)
    return Z_guess,f_guess


solution=colnew.solve(boundary_points, degrees,fsub,gsub,
                      is_linear=False,initial_guess=guess,
                      tolerances=tol,vectorized=True,
                      maximum_mesh_size=300)

# plot solution

import matplotlib.pyplot as plt
```

```
41
42 plt.ion()
43 x=solution.mesh
44 u_exact=np.sqrt(2)*np.sin(3.0*np.pi*x)
45 plt.plot(x,solution(x)[:,0],'b.',x,u_exact,'g-')
46 print ‘‘Third eigenvalue is %16.10e .’’ % solution(x)[0,2]
```

Here the new feature is the initial guess, lines 22–30. The function `guess(x)` is required to return the initial guess for both the enlarged solution vector Z and the corresponding right-hand sides. Line 24 specifies `u` as a suitable quartic polynomial, and line 25 its derivative. The rest of the code should be self-explanatory. The third eigenvalue is obtained as 88.826439647, which differs from the exact value $9\pi^2$ by a factor $1+4\times10^{-10}$.

7.4.5 A non-linear boundary value problem

As a final example in this section, we consider the *Bratu problem*

$$u''(x) + \lambda e^{u(x)} = 0, \quad 0 < x < 1, \quad \lambda > 0, \quad u(0) = u(1) = 0, \tag{7.26}$$

which arises in several scientific applications including combustion theory.

As stated this is an enigma. Does there exist a solution for all values of the parameter λ, or only for a certain set, in which case λ is an eigenvalue? If the latter, then is the set discrete, as in Section 7.4.4, or continuous? If λ is an eigenvalue, is the eigenfunction unique?

As with most real-life problems, we start from a position of profound ignorance. We might intuit that some solutions must exist since this problem has physical origins. Let us choose an arbitrary value of λ, say $\lambda = 1$, and ask whether we can solve this problem? Because it is non-linear we need an initial guess for the solution, and $u(x) = \mu x(1-x)$ satisfies the boundary conditions for all choices of the arbitrary parameter μ. As a first attempt, we attempt a direct numerical approach.

```
1  import numpy as np
2  import scikits.bvp1lg.colnew as colnew
3
4  degrees=[2]
5  boundary_points=np.array([0.0, 1.0])
6  tol=1.0e-8*np.ones_like(boundary_points)
7
8  def fsub(x, Z):
9      """The equations"""
10     u,du=Z
11     ddu=-lamda*np.exp(u)
12     return np.array([ddu])
13
14
15 def gsub(Z):
```

```
16      """The boundary conditions"""
17      u,du=Z
18      return np.array([u[0],u[1]])
19
20  def initial_guess(x):
21     """Initial guess depends on parameter mu"""
22      u=mu*x*(1.0-x)
23      du=mu*(1.0-2.0*x)
24      Z_guess=np.array([u, du])
25      f_guess=fsub(x,Z_guess)
26      return Z_guess,f_guess
27
28  lamda=1.0
29  mu=0.2
30  solution=initial_guess
31
32  solution=colnew.solve(boundary_points,degrees,fsub,gsub,
33                        is_linear=False,initial_guess=solution,
34                        tolerances=tol,vectorized=True,
35                        maximum_mesh_size=300)
36
37  # plot solution
38
39  import matplotlib.pyplot as plt
40
41  plt.ion()
42  x=solution.mesh
43  plt.plot(x,solution(x)[:,0],'b.')
44  plt.show(block=False)
```

This is very similar to the code snippet in Section 7.4.4, but with two changes. As explained in Section 3.8.7, **lambda** has a reserved meaning. It cannot be used as an identifier and so we use `lamda` in line 11 for the parameter in the equation (7.26). Next note that we have offered a guess for the solution in lines 20–26. If we run this snippet with the parameter values $\lambda = 1$ and $\mu = 0.2$ in lines 28 and 29, we obtain a smooth solution. However, if we retain $\lambda = 1$ but set $\mu = 20$, we obtain a *different* smooth solution. If we change λ, say $\lambda = 5$, then no solution is generated! It would seem that λ is not a good parameter for delineating the solutions.

Experience suggests that a better parameter for describing the solutions might be a norm of the solution. We therefore consider an enlarged problem with extra dependent variables

$$v(x) = \lambda, \quad w(x) = \int_0^x u^2(s)\, ds,$$

so that the system becomes

$$u''(x) = -v(x)e^{u(x)}, \quad v'(x) = 0, \quad w'(x) = u^2(x),$$

with boundary conditions

$$u(0) = 0, \quad w(0) = 0, \quad u(1) = 0, \quad w(1) = \gamma.$$

Note the new parameter γ, which measures the square of the norm of the solution. This suggests a new strategy. Can we construct a one-parameter family of solutions $u = u(\gamma; x)$, $\lambda = \lambda(\gamma)$ parametrized by γ?

We propose to examine this strategy numerically, using a continuation argument. We start with a very small value for γ, for which we expect the unique small-μ solution above, and so we should obtain a numerical solution. Next we increase γ by a small amount and re-solve the problem using the previously computed solution as an initial guess. By repeating this "continuation process", we can develop a set of solutions with γ as the parameter. Figure 7.3 shows how λ varies as a function of γ.

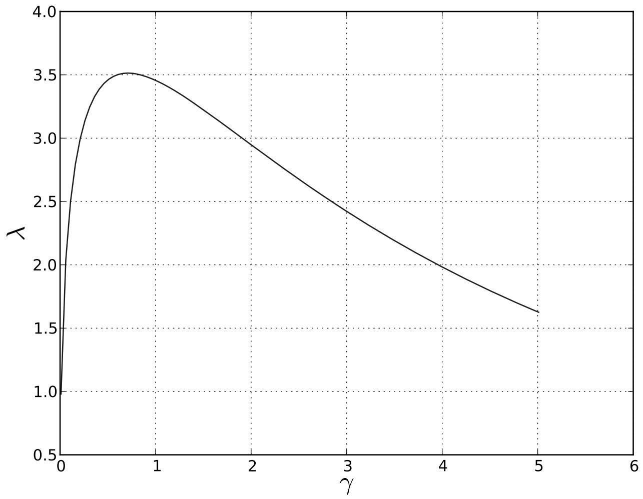

Figure 7.3 The eigenvalue λ for the Bratu problem as a function of the norm γ of the solution.

The following snippet carries out the task and draws Figure 7.3.

```
import numpy as np
import scikits.bvp1lg.colnew as colnew

degrees=[2,1,1]
boundary_points=np.array([0.0,0.0,1.0,1.0])
tol=1.0e-8*np.ones_like(boundary_points)

```

```python
def fsub(x,Z):
    """The equations"""
    u,du,v,w=Z
    ddu=-v*np.exp(u)
    dv=np.zeros_like(x)
    dw=u*u
    return np.array([ddu,dv,dw])


def gsub(Z):
    """The boundary conditions"""
    u,du,v,w=Z
    return np.array([u[0],w[1],u[2],w[3]-gamma])

def guess(x):
    u=0.5*x*(1.0-x)
    du=0.5*(1.0-2.0*x)
    v=np.zeros_like(x)
    w=u*u
    Z_guess=np.array([u,du,v,w])
    f_guess=fsub(x,Z_guess)
    return Z_guess,f_guess

solution=guess
gaml=[]
laml=[]

for gamma in np.linspace(0.01,5.01,1001):
    solution = colnew.solve(boundary_points,degrees,fsub,gsub,
                            is_linear=False,initial_guess=solution,
                            tolerances=tol,vectorized=True,
                            maximum_mesh_size=300)
    x=solution.mesh
    lam=solution(x)[:,2]
    gaml.append(gamma)
    laml.append(np.max(lam))

# plot solution
import matplotlib.pyplot as plt

plt.ion()
plt.plot(gaml,laml)
plt.xlabel(r'$\gamma$',size=20)
plt.ylabel(r'$\lambda$',size=20)
plt.grid(b=True)
```

We see from Figure 7.3 that λ is a smooth function of γ with a maximum at $\gamma \approx 0.7$. If we rerun the code snippet for $\gamma \in [0.65, 0.75]$ (by modifying line 35), we start to home in on the maximum at $\gamma = \gamma_c$. We can get the maximum value of λ via `np.max(laml)` and the code estimates $\lambda_c = 3.513830717996$. For $\lambda < \lambda_c$, there are two solutions with different values of γ, but if $\lambda > \lambda_c$, then there is no solution at all. This confirms and adds detail to our first investigation.

In fact, we can investigate the analytic solution of (7.26). If we set $v(x) = u'(x)$, then $u = \log(-v'/\lambda)$ and we find that $v'' = vv' = \frac{1}{2}(v^2)'$, so that $v' - \frac{1}{2}v^2 = k$, a constant. First, we assume $k < 0$, setting $k = -8\nu^2$. (The analysis for $k > 0$ is similar but leads to negative eigenvalues λ, and the case $k = 0$ leads to singular solutions for $u(x)$.) Thus we need to solve

$$v'(x) - \tfrac{1}{2}v^2(x) = -8\nu^2,$$

and the general solution is

$$v(x) = -4\nu \tanh(2\nu(x - x_0)),$$

where x_0 is an arbitrary constant. This implies

$$u(x) = -2\log\left[\cosh(2\nu(x - x_0))\right] + \text{const.}$$

We use the constants to fit the boundary conditions $u(0) = u(1) = 0$ finding

$$u(x) = -2\log\left[\frac{\cosh(2\nu(x - \frac{1}{2}))}{\cosh \nu}\right].$$

Next we determine $\lambda = -u''/\exp(u)$ as

$$\lambda = \lambda(\nu) = 8\left(\frac{\nu}{\cosh \nu}\right)^2.$$

Clearly, $\lambda(\nu) > 0$ for $\nu > 0$ and has a single maximum at $\nu = \nu_c$ where ν_c is the single positive root of $\coth \nu = \nu$. We found in Section 4.9.1 that $\nu_c \approx 1.19967864026$, and so $0 < \lambda < \lambda_c$ where $\lambda_c = \lambda(\nu_c) \approx 3.51383071913$. The numerical estimate above differs by a few parts in 10^9, a confirmation of its accuracy.

This has been a very brief survey of what Python can do with boundary value problems. Many important problems, e.g., equations on infinite or semi-infinite domains, have been omitted. They are however covered in Ascher et al. (1995), to which the interested reader is directed.

7.5 Delay differential equations

Delay differential equations arise in many scientific disciplines and in particular in control theory and in mathematical biology, from where our examples are drawn. For a recent survey, see, e.g., Erneux (2009). Because they may not be familiar to all users, we start by considering in some detail a very simple case. Readers can find a systematic treatment in textbooks, e.g., Driver (1997).

7.5.1 A model equation

Consider an independent variable t and a single dependent variable $x(t)$ which satisfies a delay differential equation

$$\frac{dx}{dt}(t) = x(t-\tau) \qquad t > 0, \tag{7.27}$$

where $\tau \geqslant 0$ is a constant. For $\tau = 0$, we have an ordinary differential equation and the solution is determined once the “initial datum” $x(0) = x_0$ has been specified. We say that this problem has precisely “one degree of freedom” (x_0). However, if $\tau > 0$, we can see that the appropriate initial data are the values of

$$x(t) = x_0(t), \qquad t \in [-\tau, 0], \tag{7.28}$$

for some *function* $x_0(t)$. The number of “degrees of freedom” is infinite. Now in the study of the initial value problem for first-order ordinary differential equations, exotic phenomena such as limit cycles, Hopf bifurcations and chaos require two or more degrees of freedom, i.e., a *system* of equations. But, as we shall see, all of these phenomena are present in scalar first-order delay differential equations.

Next we need to note a characteristic property of solutions of delay differential equations. Differentiating (7.27) gives

$$\frac{d^2x}{dt^2}(x) = x(t-2\tau),$$

and more generally

$$\frac{d^nx}{dt^n}(t) = x(t-n\tau), \qquad n = 1, 2, 3, \ldots \tag{7.29}$$

Now consider the value of dx/dt as t approaches 0 both from above and below. Mathematicians denote these limits by $0+$ and $0-$ respectively. From (7.27), we see that at $t = 0+$ we have $dx/dt = x_0(-\tau)$, while at $t = 0-$ equation (7.28) implies that $dx/dt = dx_0/dt$ evaluated at $t = 0-$. For general data $x_0(t)$, these limits will not be the same, and so we must expect a jump discontinuity in dx/dt at $t = 0$. Then equation (7.29) implies a jump discontinuity in d^2x/dt^2 at $t = \tau$, and in d^nx/dt^n at $t = (n-1)\tau$ and so on. Because our model equation is so simple, we can see this explicitly. Suppose, e.g., we choose $x_0(t) \equiv 1$. We can solve (7.27) analytically for $t \in [-1, 1]$ to give

$$x(t) = \begin{cases} 1 & \text{if } t \leqslant 0, \\ 1+t & \text{if } 0 < t \leqslant 1, \end{cases}$$

showing a jump of 1 in dx/dt at $t = 0$. Repeating the process gives

$$x(t) = \tfrac{3}{2} + \tfrac{1}{2}t^2 \quad 1 < t \leqslant 2,$$

giving a jump of 1 in d^2x/dt^2 at $t = 1$. This is the so called “method of steps”, which is useful only as a pedagogic exercise.

7.5.2 More general equations and their numerical solution

Clearly, we can generalize (7.27) to a system of equations, we can include many different delays $\tau_1, \tau_2, \ldots$, and even allow the delay to depend on time and state, $\tau = \tau(t, x)$. As we have suggested, delay differential equations have a remarkably rich structure, and are not a simple extension of the initial value problem for ordinary differential equations discussed in Section 7.1, and for this reason the software used there, to construct numerical solutions is of little use here. For a useful introduction to the numerical solution of delay differential equations see Bellen and Zennaro (2003).

It is extremely difficult to construct a robust "black box" integrator like the `odeint` function of 7.1, which can handle general delay differential equations. Fortunately, most of those encountered in mathematical biology have one or more constant delays and no state-dependent ones. In this simpler case, there are a number of robust algorithms, see Bellen and Zennaro (2003) for a discussion. A commonly used integrator, which goes under the generic name `dde23`, is due to Bogacki and Shampine (1989). (This is made up of Runge–Kutta integrators of orders 2 and 3 and cubic interpolators.) Because of its complexity, it is usually used in "packaged" form, as for example within *Matlab*. Within Python, it is accessible via the `pydelay` package, which can be downloaded from its website.[3] Instructions for installing such packages are given in Section A.3

The *pydelay* package contains comprehensive documentation and some well-chosen example codes, including, inter alia, the Mackey–Glass equation from mathematical biology. Here we shall consider first a superficially simpler example which however exhibits surprising behaviour. But first we need to say something about how the package works. Python is ultimately based on the *C* language. We saw in Chapter 4 how *numpy* could convert Python *list* operations into *C* array operations "on the fly" dramatically increasing Python's speed. Buried in *scipy* is the `swig` tool. This allows the user to pass to the interpreter valid *C*-code via a string of characters (often in docstring format for substantial chunks) which will be compiled "on the fly" and executed. Using `swig` proactively is not easy for the beginner, and most scientific users interested in using compiled code will gain more utility by exploring the `f2py` tool discussed in Section 8.7. The developer of *pydelay* has offered a clever interface between `swig` and the casual user. The user needs to input the equations (including various parameters) as *strings* to enable the package to write a valid *C*-programme, which is then compiled and hidden away in a temporary file. Next data required to execute the `dde23`-programme are required. Finally, after execution we need to recover the output as *numpy* arrays. In Python, these input/output operations are handled seamlessly by *dictionaries* of objects. Recall from Section 3.5.7 that a *dictionary* member consists of a pair. The first, the *key* must be a *string*, which identifies the member. The second, the *content* can be of any valid type. This background is essential for understanding the code snippets in the remainder of this section.

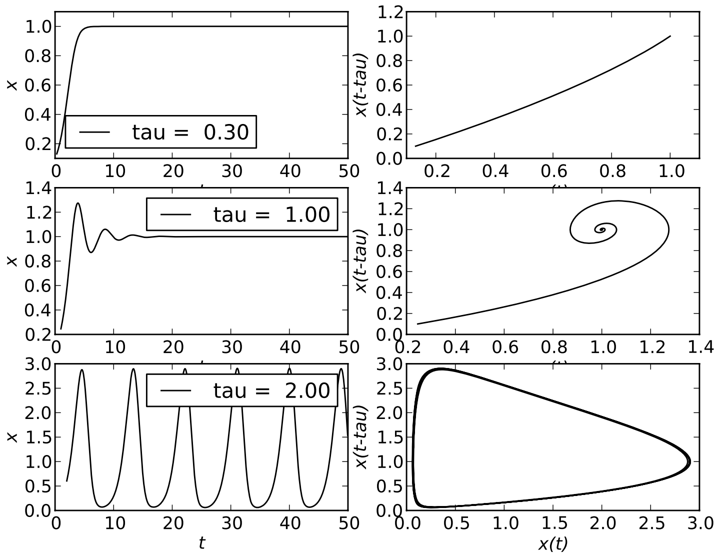

Figure 7.4 Numerical solution of the logistic differential delay equation (7.30) for three choices of the delay τ. The left graph is $x(t)$ as a function of time t. On the right is a quasi-phase plane plot, $x(t-\tau)$ versus $x(t)$.

7.5.3 The logistic equation

The dimensionless logistic equation was mentioned earlier (1.3) as a simple example in population dynamics. Now consider its delay differential equation counterpart

$$\frac{dx}{dt}(t) = x(t)(1 - x(t-\tau)), \tag{7.30}$$

where the constant τ is non-negative. For the biological background suggesting this equation, see, e.g., Murray (2002). For $\tau = 0$, we recover the ordinary differential equation whose solutions have a very simple behaviour. There are two stationary solutions $x(t) = 0$ and $x(t) = 1$, and the first is a repeller the second an attractor. For arbitrary initial data $x(0) = x_0$, the solution tends monotonically to the attractor as $t \to \infty$.

For small positive values of τ we might reasonably expect this behaviour to persist. Suppose we consider initial data

$$x(t) = x_0 \qquad \text{for } -\tau \leqslant t \leqslant 0. \tag{7.31}$$

Then a linearized analysis shows that for $0 < \tau < e^{-1}$ this is indeed the case, see row 1 of Figure 7.4. However, for $e^{-1} < \tau < \frac{1}{2}\pi$ the convergence to the attractor $y = 1$ becomes oscillatory, as can be seen row two of Figure 7.4. At $\tau = \frac{1}{2}\pi$, a Hopf bifurcation occurs, i.e., the attractor becomes a repeller and the solution tends to a *limit cycle*. This is evident in row three of Figure 7.4.

[3] It can be found at `http://pydelay.sourceforge.net`.

The supercritical case $\tau > \frac{1}{2}\pi$ is very surprising in that such a simple model can admit periodic behaviour. Further for large choices for τ, the minimum value of $x(t)$ can be extremely small. In population dynamics, we might interpret this as extermination, suggesting that the population is wiped out after one or two cycles. For further discussion of the interpretation of these solutions, see, e.g., Erneux (2009) and Murray (2002).

Each row of Figure 7.4 was produced with the aid of the following code.

```python
import numpy as np
import matplotlib.pyplot as plt
from pydelay import dde23

t_final=50
delay=2.0
x_initial=0.1

equations={'x' : 'x*(1.0-x(t-tau))'}
parameters={'tau' : delay}
dde=dde23(eqns=equations,params=parameters)
dde.set_sim_params(tfinal=t_final,dtmax=1.0,
                   AbsTol=1.0e-6,RelTol=1.0e-3)
histfunc={'x': lambda t: x_initial}
dde.hist_from_funcs(histfunc,101)
dde.run()

t_vis=0.1*t_final
sol=dde.sample(tstart=t_vis+delay,tfinal=t_final,dt=0.1)
t=sol['t']
x=sol['x']
sold=dde.sample(tstart=t_vis,tfinal=t_final-delay,dt=0.1)
xd=sold['x']

plt.ion()
plt.plot(t,x)
plt.xlabel('t')
plt.ylabel('x(t)')
#plt.title('Logistic equation with x(0)=%5.2f, delay tau=%5.2f'
#          % (x_initial,delay))

plt.figure()
plt.plot(x,xd)
plt.xlabel('tx')
plt.ylabel('x(t-tau)')
#plt.title('Logistic equation with x(0)=%5.2f, delay tau=%5.2f'
#         % (x_initial,delay))
```

Lines 1 and 2 are routine. Line 3 imports `dde23` as well as *scipy* and various other modules. Lines 5–7 just declare the length of the evolution, the delay parameter τ and the initial value x_0, and need no explanation.

Lines 9–16 contain new ideas. We have to tell `dde23` our equations, supplying them in valid *C*-code. Fortunately, most arithmetic expressions in core Python are acceptable. Here there is precisely one equation and we construct a *dictionary* `equations` with precisely one member. The equation is for dx/dt and so the member key is `'x'`, and the value is a Python *string* containing the formula for dx/dt, line 9. The equation involves a parameter `tau`, and so we need a second *dictionary* `parameters` to supply it. The *dictionary* contains an element for each member, whose key is the *string* representing the parameter, and the value is the numerical value of the parameter, line 10. Next line 11 writes *C*-functions to simulate this problem. Finally, we have to supply input parameters to it. Examples of the usual numerical ones are given in lines 12–13. Here the "initial data" corresponding to x are given by (7.31) and we construct a third *dictionary* with one member to supply them, using the anonymous function syntax from Section 3.8.7 in line 14 and supply them to the *C*-code in line 15. Finally, line 16 runs the background *C*-code.

Next, we have to extract the solution. Usually we do not know the precise initial data, and will therefore choose to display the steady state after (hopefully) initial transient behaviour has disappeared. The purpose of `t_vis` is to choose a starting time for the figures. The solution generated by the *C*-programme is available, but at values of t which have chosen by `dde23`, and these will not usually be evenly spaced. The purpose of the `sample` function in line 19 is to perform cubic interpolation and to return the data evenly spaced in t for the parameters specified in the arguments to the function. In the right-hand column of Figure 7.4, we show quasi-phase plane plots of $x(t-\tau)$ versus $x(t)$. Thus, in line 19, we choose the range to be t in $[t_{vis} + \tau, t_{final}]$, but in line 22 we use t in $[t_{vis}, t_{final} - \tau]$, both with the same spacing. The function `sample` returns a *dictionary*. Then line 20 extracts the member with key `'t'`, which turns out to be *numpy* array of evenly spaced t-values. Lines 21 and 23 return $x(t)$ and $x(t - \tau)$ values as arrays. More arguments can be supplied to the `dde` functions. For further information, see the very informative docstrings.

Finally, in lines 25–37 we draw $x(t)$ as a function of t and, in a separate figure, a plot of $x(t - \tau)$ against $x(t)$ a "pseudo phase plane portrait". The output of this snippet corresponds to the last row in Figure 7.4. The entire compound figure was constructed using the techniques of Section 5.9, and is left as an exercise for the reader.

7.5.4 The Mackey–Glass equation

The Mackey–Glass equation arose originally as a model for a physiological control system, Mackey and Glass (1977), and can be found in many textbooks on mathematical biology, e.g., Murray (2002). It is also of great interest in dynamical systems theory for it is a simple one-dimensional system (albeit with an infinite number of degrees of freedom) which exhibits limit cycles, period doubling and chaos. In dimensionless

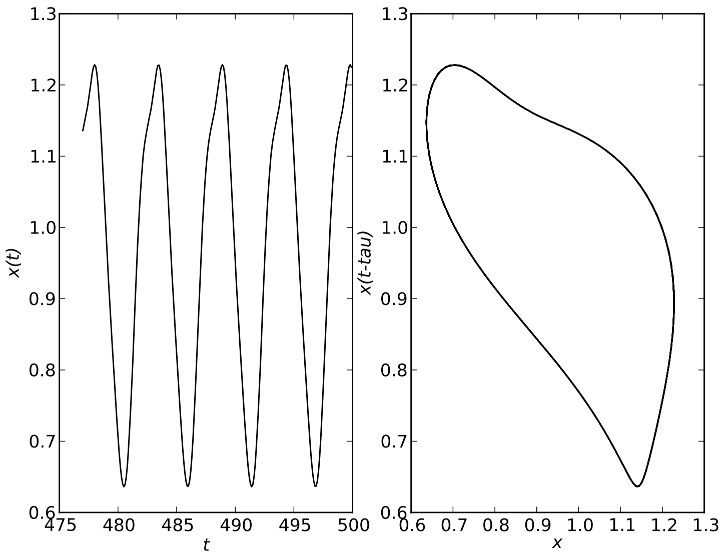

Figure 7.5 Numerical solution of the Mackey–Glass equation with $a = 2$, $b = 1$, $m = 7$, $\tau = 2$ and initial data $x_0(t) = 0.5$. The left graph is $x(t)$ as a function of time t. On the right is a quasi-phase plane plot, $x(t - \tau)$ versus $x(t)$.

form, it is

$$\frac{dx}{dt}(t) = a\frac{x(t-\tau)}{1 + (x(t-\tau))^m} - bx(t), \tag{7.32}$$

where the constants a, b, m and τ are non-negative. For $-\tau \leqslant t \leqslant 0$, we shall choose initial data $x(t) = x_0$ as before.

Because of its importance, example code to solve it is included in the `pydelay` package. However, if you have already experimented with the code snippet for the logistic delay differential equation in the previous subsection, it is easiest merely to edit a copy of that, as we now show.

Clearly, we need to change the constants, and so lines 5–7 of the previous snippet should be replaced by the arbitrary values

```
t_final=500
a=2.0
b=1.0
m=7.0
delay=2.0
x_initial=0.5
```

Next we need to change the equation and parameters. Now `dde23` knows about the *C* `math.h` library, so that $\cos u$ would be encoded as `cos(u)` etc. First consider the exponentiation in (7.32). In Python (and Fortran), we would represent u^v by `u**v`. However,

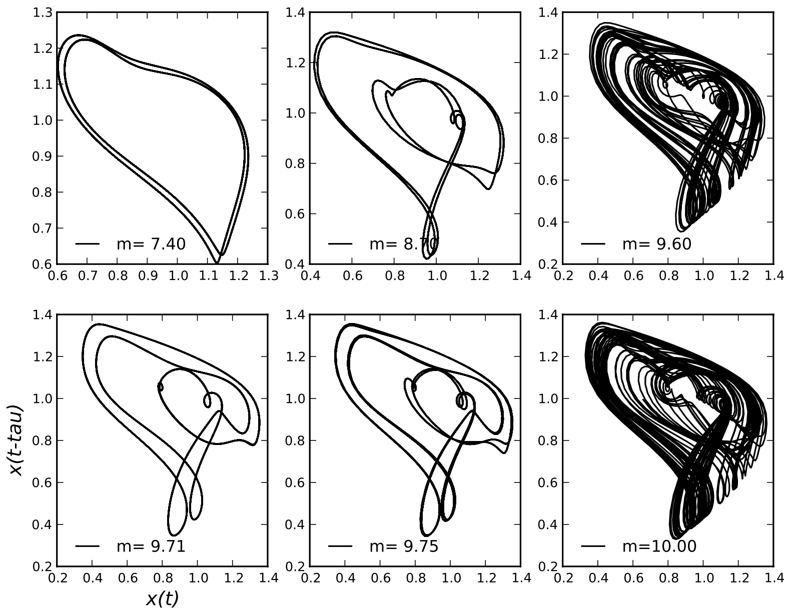

Figure 7.6 Quasi-phase plane portraits for the Mackey–Glass equation. The parameters used are specified in Figure 7.5, except that $m = 7$ there, is changed to the values shown. At $m = 7$, the solution has period 1. At $m = 7.4$, this has doubled, and at $m = 8.7$, the period has quadrupled. Increasing m further leads to chaos, as shown at $m = 9.6$. However, larger values of m lead again to regular motion. At $m = 9.71$, the solution has period 3, but this doubles if $m = 9.75$, and chaos ensues again at $m = 10$. Larger values of m lead again to regular motion. Note that the actual transitions between different behaviours take place at values of m intermediate to those shown.

in *C* we have to use `pow(u,v)`. To encode the equation and parameters, we need to replace lines 9 and 10 of the earlier snippet by

```
equations={ 'x' : 'a*x(t-tau)/(1.0+pow(x(t-tau), m))-b*x' }
parameters = { 'a' : a,'b' : b,'m' : m,'tau': delay }
```

Next we choose to examine the late stages of the trajectories and so change line 18 to

```
t_vis = 0.95*t_final
```

Finally, we need to change a number of strings in lines 29 and 36. In particular, change the *string* `'Logistic'` to `'Mackey-Glass'`, `'delay tau'` to `'power m'` and `delay` to `m`. The amended snippet was used to produce each of the two components of Figure 7.5, which exhibits limit cycle behaviour, qualitatively similar to the last row of Figure 7.4.

Because the Mackey–Glass equation contains four adjustable parameters a complete exploration of its properties is a non-trivial undertaking. Here, following the original authors, we shall experiment with changing m, while keeping the other parameters fixed. We display the quasi-phase plane plots (after running the evolution for rather longer times than in the snippet above) in Figure 7.6. The details are shown in the figure caption, but we see that the limit cycle behaviour exhibits period doubling and chaos. Biologists may prefer to examine the corresponding wave forms, which will require adjustment of the `t_vis` parameter to limit the number of periods displayed.

It would be possible to supply examples that are more complicated, such as those enclosed in the *pydelay* package, but the interested reader should be able to build on the two exhibited here.

7.6 Stochastic differential equations

The previous three sections have dealt with scientific problems where the models can be fairly precisely described, and the emphasis has been on methods with high orders of accuracy. There exist many areas of scientific interest, e.g., parts of mathematical biology and mathematical finance, where such precision is simply unavailable. Typically, there are so many sources of influence that they cannot be counted and we need to consider stochastic evolution. For a heuristic but wide-ranging survey of the possibilities see, e.g., Gardiner (2009). We shall specialize here to autonomous stochastic differential equations. A careful treatment of the theory can be found in, e.g., Øksendal (2003), and for the underlying theory of numerical treatments, the standard reference is Kloeden and Platen (1992). However, this author has found the numerical treatment provided by Higham (2001) particularly enlightening, and some of his ideas are reflected in the presentation here. Of necessity, Higham omits many of the mathematical details, but for these the stimulating lectures of Evans (2013) are highly recommendable. In this sketch of the theory, we shall also take a very heuristic approach, relying on the cited references to supply the theoretical justification.

7.6.1 The Wiener process

The most commonly used mechanism for modelling random behaviour is the *Wiener process* or *Brownian motion*. The standard scalar process or motion over $[0, T]$ is a random variable $W(t)$, which depends continuously on t, and satisfies:

- $W(0) = 0$ with probability 1,
- if $0 \leqslant s < t \leqslant T$, then the increment $W(t) - W(s)$ is a random variable which is normally distributed with mean 0 and variance $t - s$ (or standard deviation $\sqrt{t-s}$),
- for $0 \leqslant s < t < u < v \leqslant T$ the increments $W(t) - W(s)$ and $W(v) - W(u)$ are independent variables.

These are essentially the axioms that Einstein used in his explanation of Brownian motion.

It is easy to construct a numerical approximation of this using Python. The module *numpy* contains a submodule called *random*, and this contains a function `normal` which generates instances of pseudo-random variables which should be normally distributed.

See the docstring for more details. In the following code snippet, we generate and plot a Brownian motion for $0 \leqslant t \leqslant T$ with 500 steps.

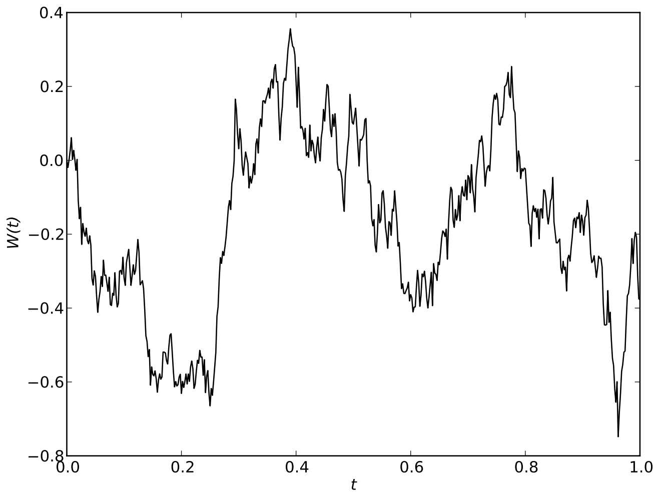

Figure 7.7 A sample of a Wiener process or discrete Brownian motion.

```
1  import numpy as np
2  import numpy.random as npr
3
4  T=1
5  N=500
6  t,dt=np.linspace(0,T,N+1,retstep=True)
7  dW=npr.normal(0.0,np.sqrt(dt),N+1)
8  dW[0]=0.0
9  W=np.cumsum(dW)
10
11 import matplotlib.pyplot as plt
12 plt.ion()
13
14 plt.plot(t,W)
15 plt.xlabel('t')
16 plt.ylabel('W(t)')
17 plt.title('Sample Wiener Process',weight='bold',size=16)
```

The random variables are generated in line 7. In its simplest form, the arguments to be supplied to the `normal` function are the mean, standard deviation and the shape of the required output array. Here we have generated precisely one more random number than we require, so we that we can change the first value to zero in line 8. This will ensure that $W(0) = 0$. Finally, we generate $W(t)$ as a cumulative sum of the increments in line 9. (The function `cumsum` was introduced in Section 4.6.2.) The rest of the snippet is routine, and we show one actual output in Figure 7.7. "Almost surely", your attempt will not be the same. Why?

7.6.2 The Itô calculus

We need to examine more closely Wiener processes. In order to explain what is happening, we make a small change to the notation, setting

$$dW(t) = \Delta W(t) = W(t + \Delta t) - W(t).$$

We know that the expectation value of ΔW is $\langle \Delta W \rangle = 0$, and its variance is $\langle (\Delta W)^2 \rangle = \Delta t$. This suggests that the Wiener process $W(t)$, while continuous, is nowhere differentiable, since $\Delta W/\Delta t$ will diverge as $\Delta t \to 0$. (In fact, it can be shown "almost surely", i.e., with probability 1, that $W(t)$ is indeed nowhere differentiable.) Thus we may talk about the differential $dW(t)$ but not about $dW(t)/dt$.

Consider next the autonomous deterministic initial value problem

$$\dot{x}(t) = a(x(t)) \qquad \text{with } x(0) = x_0. \tag{7.33}$$

We choose to write this in an equivalent form as

$$dx(t) = a(x(t))\, dt \qquad \text{with } x(0) = x_0, \tag{7.34}$$

which is to be regarded as a shorthand for

$$x(t) = x_0 + \int_0^t a(x(s))\, ds. \tag{7.35}$$

We now introduce a "noise" term on the right-hand side of equation (7.34) and replace the deterministic $x(t)$ by a random variable $X(t)$ parametrized by t, writing our *stochastic differential equation (SDE)* as

$$dX(t) = a(X(t))\, dt + b(X(t))\, dW(t) \qquad \text{with } X(0) = X_0, \tag{7.36}$$

which is shorthand for

$$X(t) = X_0 + \int_0^t a(X(s))\, ds + \int_0^t b(X(s))\, dW(s), \tag{7.37}$$

where the second integral on the right-hand side will be defined below. Assuming this has been done, we say that $X(t)$ given by equation (7.37) is a *solution* of the stochastic differential equation (7.36). Before proceeding further with this, we need to point out a trap for the unwary.

Suppose $X(t)$ satisfies (7.36) and let $Y(t) = f(X(t))$ for some smooth function f. Naïvely, we might expect

$$dY = f'(X)\, dX = bf'(X)\, dW + af'(X)\, dt, \tag{7.38}$$

but, in general, this is incorrect! To see this, suppose we construct the first two terms in the Taylor series

$$\begin{aligned} dY =& f'(X)\,dX + \tfrac{1}{2}f''(X)dX^2 + \dots \\ =& f'\,[a\,dt + b\,dW] + \tfrac{1}{2}f''\left[a^2\,dt^2 + 2ab\,dt\,dW + b^2\,dW^2\right] + \dots \\ =& bf'\,dW + af'\,dt + \tfrac{1}{2}b^2 f''\,dW^2 + abf''\,dW\,dt + \tfrac{1}{2}a^2\,dt^2 + \dots. \end{aligned}$$

We have grouped the terms in decreasing order of size because we know $dW = O(dt^{1/2})$. Now Itô suggested that we should replace

$$dW^2 \to dt, \quad dW\,dt \to 0,$$

in the above formula and neglect quadratic and higher-order terms, obtaining

$$dY = bf'\,dW + \left(af' + \tfrac{1}{2}b^2 f''\right)dt. \tag{7.39}$$

This is called *Itô's formula* and it can be used to replace (7.38). Let us see it in action in a simple example. Suppose we set

$$dX = dW \qquad \text{with } X(0) = 0,$$

i.e., X *is* the Wiener process $W(t)$ and $a = 0$ and $b = 1$.

Let us choose $f(x) = e^{-x/2}$. Then the Itô formula predicts

$$dY = -\tfrac{1}{2}Y\,dW + \tfrac{1}{8}Y\,dt \qquad \text{with } Y(0) = 1.$$

We do not know how to solve this, but we can take its expectation value finding

$$d\langle Y\rangle = \tfrac{1}{8}\langle Y\rangle\,dt.$$

Also $\langle Y\rangle(0) = 1$ and so $\langle Y\rangle(t) = e^{t/8}$. However, a naïve approach, i.e., ignoring the Itô correction, might suggest $\langle X(t)\rangle = \langle W(t)\rangle = 0$ and so $\langle Y\rangle(t) = 1$.

The following Python code snippet can be used to test the alternatives.

```python
import numpy as np
import numpy.random as npr

T=1
N=1000
M=5000
t,dt=np.linspace(0,T,N+1,retstep=True)
dW=npr.normal(0.0,np.sqrt(dt),(M,N+1))
dW[ : ,0]=0.0
W=np.cumsum(dW,axis=1)
U=np.exp(- 0.5*W)
Umean=np.mean(U,axis=0)
Uexact=np.exp(t/8)

# Plot it

```

```
import matplotlib.pyplot as plt
plt.ion()

plt.plot(t,Umean,'b-', label="mean of %d paths" % M)
plt.plot(t, Uexact, 'r-',label="exact U")
for i in range(5):
   plt.plot(t,U[i, : ], '--')

plt.xlabel('t')
plt.ylabel('U')
plt.title('U= exp(-W(t)/2) for Wiener Process W(t)',
          weight='bold',size=16)
plt.legend(loc='best')

maxerr=np.max(np.abs(Umean-Uexact))
print "With %d paths and %d intervals the max error is %g" % \
           (M,N,maxerr)
```

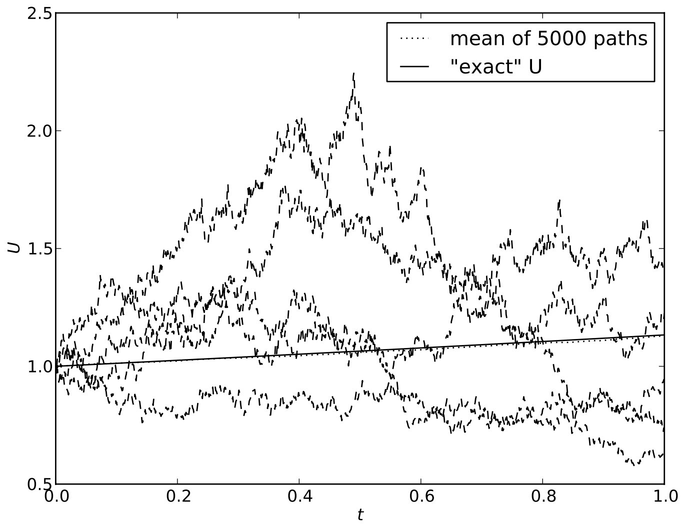

Figure 7.8 A function $U(t) = \exp(-\frac{1}{2}W(t))$ of a discrete Brownian motion $W(t)$. The dashed curves are five sample paths. The solid curve gives the "exact" solution $\langle U\rangle(t) = \exp(\frac{1}{8}t)$ predicted by the Itô formula. Almost hidden beneath it is the dotted curve which maps the mean over all of the 5000 samples, The maximum difference between the two for these Brownian motion samples is about 1×10^{-2}.

Most of this code should be familiar. The first novelty is line 8 where we generate `M` instances of a Brownian motion, each with `N+1` steps. `dW` is a two-dimensional array in which the first index refers to the sample number and the second to the time. Line 9 sets the initial increment for each sample to zero, and then line 10 generates the `M` Brownian motion samples, by summing over the second index. `U` has of course the same shape as `W` and `dW`. Line 11 generates the function values and line 12 calculates their average over the `M` samples. (Beginners might like to consult the docstrings for `cumsum` and `mean`.) Notice the total absence of explicit loops with the consequent tiresome, prone-to-error, book-keeping details. In Figure 7.8, we plot the first five sample paths, the mean over the `M` samples and the Itô suggestion $\langle U(t)\rangle = e^{t/8}$, and we also compute and print the infinity norm of the error. As always, these code snippets are not tablets set in stone, but suggestions for a basis for numerical experiments. Hacking them is the best way to gain experience!

7.6.3 Itô and Stratanovich stochastic integrals

In this subsection, we try to make sense of the concept

$$\int_{T_1}^{T_2} f(t)\, dW(t),$$

which we saw already in equation (7.37). We adopt the standard Riemann–Stieltjes approach and choose some partition of the interval

$$T_1 = t_0 < t_1 < \ldots < t_N = T_2,$$

and set $\Delta t_k = t_{k+1} - t_k$, $W_k = W(t_k)$ and $\Delta W_k = W_{k+1} - W_k$. We also choose an arbitrary point in each subinterval $\tau_k \in [t_{k-1}, t_k]$, and for simplicity we choose the point uniformly

$$\tau_k = (1-\lambda)t_k + \lambda t_{k+1},$$

where $\lambda \in [0, 1]$ is a fixed parameter. We will be considering an infinite refinement, $\max_k \Delta t_k \to 0$, which we denote informally by $N \to \infty$. Then our definition would look like

$$\int_{T_1}^{T_2} f(t)\, dW(t) = \lim_{N\to\infty} \sum_{k=0}^{N-1} f(\tau_k)\, \Delta W_k,$$

in the sense that the expectation values of both sides are the same. This turns out to be well-defined but unfortunately, and unlike the deterministic case, it depends critically on the choice of τ_k, or more precisely on the choice of $f(\tau_k)$. To see this, we consider a particular example, $f(t) = W(t)$, and denote $W(\tau_k)$ by $\widehat{W}_k$. We start from the algebraic identity

$$\widehat{W}_k \Delta W_k = \tfrac{1}{2}(W_{k+1}^2 - W_k^2) - \tfrac{1}{2}\Delta W_k^2 + (\widehat{W}_k - W_k)^2 + (W_{k+1} - \widehat{W}_k)(\widehat{W}_k - W_k).$$

We now sum this identity from $k = 0$ to $k = N-1$, and take expectation values. The first term on the right produces $\frac{1}{2}\langle W_N^2 - W_0^2\rangle = \frac{1}{2}\langle (W^2(T_2) - W^2(T_1))\rangle$. The second gives $-\frac{1}{2}\sum \Delta t_k = -\frac{1}{2}(T_2 - T_1)$. Similarly, the third term produces $\sum(\tau_k - t_k) = \lambda(T_2 - T_1)$.

The expectation value of the final term gives zero, because the two time intervals are disjoint and hence uncorrelated. Therefore

$$\int_{T_1}^{T_2} W(t)\,dW(t) = \tfrac{1}{2}\langle W^2(T_2) - W^2(T_1)\rangle + (\lambda - \tfrac{1}{2})(T_2 - T_1), \tag{7.40}$$

in the sense of expectation values. If we choose $\lambda = \frac{1}{2}$, we have the "common sense" result, and this is usually called the *Stratanovich integral*. In many cases, it makes more sense to choose τ_k to be the left end point of the interval, i.e., $\lambda = 0$, and this gives rise to the *Itô integral*, which is more widely used. It is straightforward to modify the last Python code snippet to verify the formula (7.40), at least in the Itô case.

7.6.4 Numerical solution of stochastic differential equations

We return to our stochastic differential equation (7.36)

$$dX(t) = a(X(t))\,dt + b(X(t))\,dW(t) \qquad \text{with } X(0) = X_0, \tag{7.41}$$

to be solved numerically for $t \in [0, T]$. We partition the t-interval into N equal subintervals of length $\Delta t = T/N$, set $t_k = k\Delta t$ for $k = 0, 1, \dots, N$ and abbreviate $X_k = X(t_k)$. Formally

$$X_{k+1} = X_k + \int_{t_k}^{t_{k+1}} a(X(s))\,ds + \int_{t_k}^{t_{k+1}} b(X(s))\,dW(s). \tag{7.42}$$

Next consider a smooth function $Y = Y(X(s))$ of $X(s)$. By Itô's formula (7.39)

$$dY = \mathcal{L}[Y]\,dt + \mathcal{M}[Y]\,dW,$$

where

$$\mathcal{L}[Y] = a(X)\frac{dY}{dX} + \tfrac{1}{2}b(X)\frac{d^2Y}{dX^2}, \qquad \mathcal{M}[Y] = b(X)\frac{dY}{dX}.$$

Therefore

$$Y(X(s)) = Y(X_k) + \int_{t_k}^{s} \mathcal{L}[Y(X(\tau))]\,d\tau + \int_{t_k}^{s} M[Y(X(\tau))]\,dW(\tau).$$

Next replace $Y(X)$ in the equation above first by $a(X)$ and then by $b(X)$, and substitute the results into equation (7.42), obtaining

$$\begin{aligned} X_{k+1} =& X_k + \int_{t_k}^{t_{k+1}} \left\{ a(X_k) + \int_{t_k}^{s} \mathcal{L}[a(X(\tau))]\,d\tau + \int_{t_k}^{s} \mathcal{M}[a(X(\tau))]\,dW(\tau) \right\} ds + \\ & \int_{t_k}^{t_{k+1}} \left\{ b(X_k) + \int_{t_k}^{s} \mathcal{L}[b(X(\tau))]\,d\tau + \int_{t_k}^{s} \mathcal{M}[b(X(\tau))]\,dW(\tau) \right\} dW(s). \end{aligned}$$

We rearrange this as

$$\begin{aligned} X_{k+1} =& X_k + a(X_k)\Delta t + b(X_k)\Delta W_k \\ & + \int_{t_k}^{t_{k+1}} dW(s) \int_{t_k}^{s} dW(\tau)\mathcal{M}[b(X(\tau))] \\ & \quad \text{plus integrals over } ds\,d\tau, ds\,dW(\tau) \text{ and } dW(s)\,d\tau. \end{aligned}$$

The *Euler–Maruyama method* consists of retaining just the first line of the expression above

$$X_{k+1} = X_k + a(X_k)\Delta t + b(X_k)(W_{k+1} - W_k). \tag{7.43}$$

Milstein's method includes an approximation of the second line via

$$\begin{aligned} X_{k+1} = &X_k + a(X_k)\Delta t + b(X_k)(W_{k+1} - W_k) + \\ &b(X_k)\frac{db}{dX}(X_k)\int_{t_k}^{t_{k+1}} dW(s)\int_{t_k}^{s} dW(\tau). \end{aligned} \tag{7.44}$$

From the calculations in the previous subsection, we know how to compute the double integral, in the Itô form, as

$$\begin{aligned} \int_{t_k}^{t_{k+1}} dW(s)\int_{t_k}^{s} dW(\tau) &= \int_{t_k}^{t_{k+1}} (W(s) - W_k)\, dW(s) \\ &= \tfrac{1}{2}(W_{k+1}^2 - W_k^2) - \tfrac{1}{2}\Delta t - W_k(W_{k+1} - W_k) \\ &= \tfrac{1}{2}[(W_{k+1} - W_k)^2 - \Delta t]. \end{aligned}$$

Thus Milstein's method finally becomes

$$X_{k+1} = X_k + a(X_k)\Delta t + b(X_k)\Delta W_k + \tfrac{1}{2}b(X_k)b'(X_k)[(\Delta W_k)^2 - \Delta t]. \tag{7.45}$$

Here we have treated only a single scalar equation. The generalization to systems of SDEs is not entirely straightforward, but is discussed in the cited references.

An important consideration is the accuracy of these methods. There are two definitions in common use: strong and weak convergence. Suppose we fix some t-value, say $\tau \in [0, T]$, and suppose that $\tau = n\Delta t$. In order to estimate the accuracy at this value τ we need to compare the computed trajectory X_n with the exact one $X(\tau)$. But both are random variables, and so more than one comparison method is plausible. If we want to measure closeness of trajectories we might look at the expectation value of the difference. Then a method has a *strong order of convergence* γ if

$$\langle |X_n - X(\tau)| \rangle = O(\Delta t^\gamma) \qquad \text{as } \Delta t \to 0. \tag{7.46}$$

However, for some purposes the difference of the expectation values might be more relevant. Then a method has a *weak order of convergence* γ if

$$|\langle X_n \rangle - \langle X(\tau) \rangle| = O(\Delta t^\gamma) \qquad \text{as } \Delta t \to 0. \tag{7.47}$$

Both the Euler–Maruyama and the Milstein methods have weak order of convergence equal to one. However, looking at the method of derivation we might guess that Euler-Marayuma has strong order of convergence $\gamma = \frac{1}{2}$ and that the Milstein method has $\gamma = 1$, and this turns out to be the case. A theoretical justification can be found in the textbooks, while an empirical verification is provided by a code snippet later in this section.

As a concrete example, we consider equation (7.41) with $a(X) = \lambda X$ and $b(X) = \mu X$ where λ and μ are constants

$$dX(t) = \lambda X(t)\, dt + \mu X(t)\, dW(t) \qquad \text{with } X(0) = X_0. \tag{7.48}$$

This arises in financial mathematics as an asset price model, and is a key ingredient in the derivation of the *Black–Scholes* partial differential equation, Hull (2009). Its formal Itô solution is

$$X(t) = X_0 \exp\left[(\lambda - \tfrac{1}{2}\mu^2)t + \mu W(t)\right]. \tag{7.49}$$

We choose the arbitrary parameter values $\lambda = 2$, $\mu = 1$ and $X_0 = 1$.

First we implement the Milstein algorithm and compare it with the analytic solution (7.49). This can be performed using the following snippet, which uses no new Python features.

```
import numpy as np
import numpy.random as npr

# Set up grid
T=1.0
N=1000
t,dt=np.linspace(0,T,N+1,retstep=True)

# Get Brownian motion
dW=npr.normal(0.0, np.sqrt(dt),N+1)
dW[0]=0.0
W=np.cumsum(dW)

# Equation parameters and functions
lamda=2.0
mu=1.0
Xzero=1.0
def a(X): return lamda*X
def b(X): return mu*X
def bd(X): return mu*np.ones_like(X)

# Analytic solution
Xanal=Xzero*np.exp((lamda-0.5*mu*mu)*t+mu*W)

# Milstein solution
Xmil=np.empty_like(t)
Xmil[0]=Xzero
for n in range(N):
    Xmil[n+1]=Xmil[n]+dt*a(Xmil[n]) + dW[n+1]*b(Xmil[n]) + 0.5*(
        b(Xmil[n])*bd(Xmil[n])*(dW[n+1]**2-dt))

import matplotlib.pyplot as plt

plt.ion()
```

```
35 plt.plot(t,Xanal,'b-',label='analytic')
36 plt.plot(t,Xmil,'g-.',label='Milstein')
37 plt.legend(loc='best')
38 plt.xlabel('t')
39 plt.ylabel('X(t)')
40 #plt.suptitle('Comparison of Milstein method' +
41 #           'and analytic solution of a SDE',
42 #           weight='bold',size=16)
```

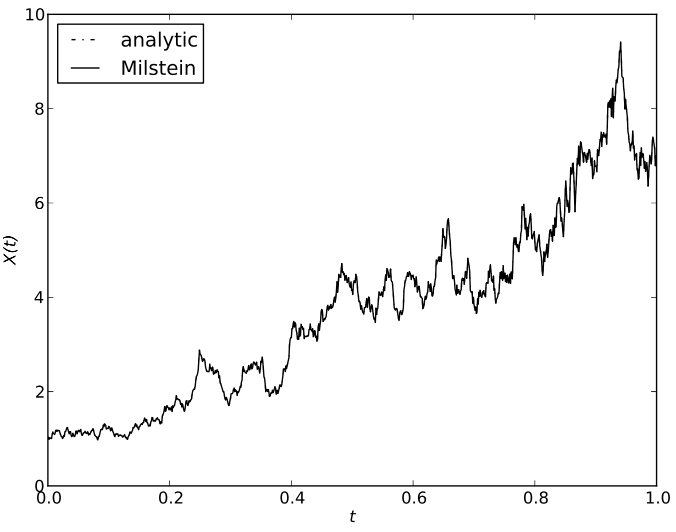

Figure 7.9 An instance of an analytic solution (7.49) of the equation (7.48) and its solution using the Milstein algorithm over 1000 intervals. There is no discernible difference.

As can be seen in Figure 7.9, the Milstein algorithm generates a solution which appears, visually, to be very close to the analytic one. As an exercise, we might like to modify lines 29 and 30 to produce the equivalent figure using the Euler–Maruyama algorithm. We should be cautious though, for the figure relies on one instance of a Brownian motion. Running the snippet repeatedly shows that the apparent convergence can be better or worse than the example shown here.

Therefore, we next consider, numerically, the convergence rate of the two numerical algorithms. The following code snippet investigates empirically the strong convergence of the Euler–Maruyama and Milstein algorithms. For the sake of brevity, it contains only minimal comments, but a commentary on the code follows the snippet.

```
1  import numpy as np
2  import numpy.random as npr
3
```

```
# Problem definition
M=1000          # Number of paths sampled
P=6                 # Number of discretizations
T=1                 # Endpoint of time interval
N=2**12          # Finest grid size
dt=1.0*T/N

# Problem parameters
lamda=2.0
mu=1.0
Xzero=1.0

def a(X): return lamda*X
def b(X): return mu*X
def bd(X): return mu*np.ones_like(X)

# Build the Brownian paths.
dW=npr.normal(0.0,np.sqrt(dt), (M,N+1))
dW[ : , 0]=0.0
W=np.cumsum(dW,axis=1)

# Build the exact solutions at the ends of the paths
ones=np.ones(M)
Xexact=Xzero*np.exp((lamda-0.5*mu*mu)*ones+mu*W[ : , -1])
Xemerr=np.empty((M, P))
Xmilerr=np.empty((M, P))

# Loop over refinements
for p in range(P):
    R=2**p
    L=N/R                       # must be an integer!
    Dt=R*dt
    Xem=Xzero*ones
    Xmil=Xzero*ones
    Wc=W[ : , : :R]
    for j in range(L):      # integration
        deltaW=Wc[ : , j+1]-Wc[ : , j]
        Xem+=Dt*a(Xem)+deltaW*b(Xem)
        Xmil+=Dt*a(Xmil)+deltaW*b(Xmil)+ \
              0.5*b(Xmil)*bd(Xmil)*(deltaW**2-Dt)
    Xemerr[ : ,p]=np.abs(Xem-Xexact)
    Xmilerr[ : ,p]=np.abs(Xmil-Xexact)

# Do some plotting
```

```
48  import matplotlib.pyplot as plt
49  plt.ion()
50
51  Dtvals=dt*np.array([2**p for p in range(P)])
52  lDtvals=np.log10(Dtvals)
53  Xemerrmean=np.mean(Xemerr,axis=0)
54  plt.plot(lDtvals,np.log10(Xemerrmean),'bo')
55  plt.plot(lDtvals,np.log10(Xemerrmean),'b:',label='EM actual')
56  plt.plot(lDtvals,0.5*np.log10(Dtvals),'b-.',
57           label='EM theoretical')
58  Xmilerrmean=np.mean(Xmilerr,axis=0)
59  plt.plot(lDtvals,np.log10(Xmilerrmean),'bo')
60  plt.plot(lDtvals,np.log10(Xmilerrmean),'b--',label='Mil actual')
61  plt.plot(lDtvals,np.log10(Dtvals),'b-',label='Mil theoretical')
62  plt.legend(loc='best')
63  plt.xlabel(r'$\log_{10}\Delta t$',size=16)
64  plt.ylabel(r'$\log_{10}\left(\langle|X_n-X(\tau)|\rangle\right)$',
65             size=16)
66  #plt.title('Strong convergence of Euler--Maruyama and Milstein',
67  #         weight='bold',size=16)
68
69  emslope=((np.log10(Xemerrmean[-1])-np.log10(Xemerrmean[0])) /
70            (lDtvals[-1]-lDtvals[0]))
71  print 'Empirical EM slope is %g' % emslope
72  milslope=((np.log10(Xmilerrmean[-1])-
73            np.log10(Xmilerrmean[0])) / (lDtvals[-1]-lDtvals[0]))
74  print 'Empirical MIL slope is %g' % milslope
```

The idea here is to do a number of integrations simultaneously, corresponding to grids of size $N/2^p$ for $p = 0, 1, 2, 3, 4, 5$. For each choice of p, we will sample over M integrated paths. The various parameters are set in lines 5–10 and 12–14, and the functions $a(X)$, $b(X)$ and $b')X)$ are set in lines 16–18. Lines 21–23 construct M Brownian paths for the finest discretization. Up to here should be familiar. Lines 26 and 27 build the exact solution for each of the sample paths. We shall be computing a Euler-Mayurama and a Milstein error for each sample path and each discretization, and lines 28 and 29 reserve space for them. The actual calculations are done in lines 32–45, where we loop over p. Thus we consider grids of size `L` with spacing `Dt`. Lines 36 and 37 set up the starting values for each method and sample, and line 38 sets up Brownian path samples for the appropriate spacing. The loop in lines 39–43 carries out the two integrations for each sample path. We compute the error for each method corresponding to strong convergence, equation (7.46) with $\tau = T$, in lines 44 and 45. Most of the rest of the code is routine. In line 53. we compute the mean of the Euler-Marayuma error averaged over the M samples, and then plot it as a function of discretization time Δt on a log–log plot.

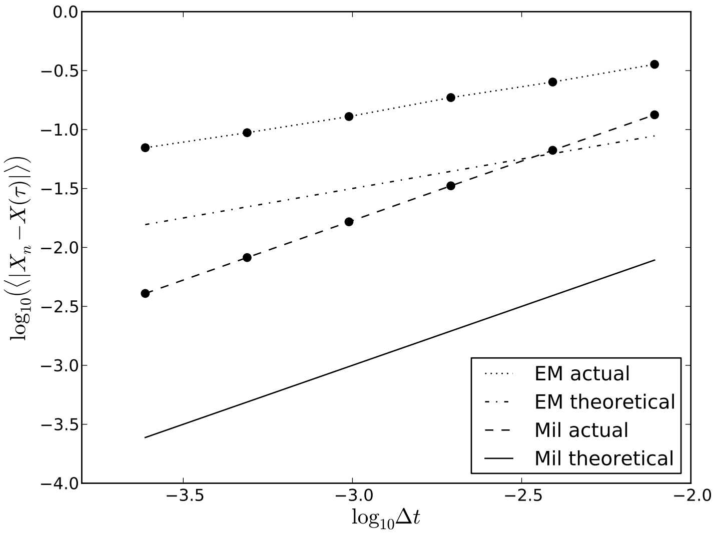

Figure 7.10 The expectation value of the final error in the numerical solution of equation (7.48) as a function of Δt. What matters here are the slopes of the "lines". The Milstein method converges like Δt, while the Euler–Maruyama one converges like $(\Delta t)^{1/2}$.

For reference, we plot $\sqrt{\Delta t}$ on the same axes to give a visual indication that $\gamma \approx \frac{1}{2}$. We then do the same for the Milstein error.

Lines 62–65 create TeX-formatted labels for the axes. Purists will complain with some justification that the fonts used are incongruous in this context. This is because we have followed the prescriptions for non-LaTeX users outlined in Section 5.7.1. There, reasonably enough, it is assumed that the text surrounding the figure is set in the default TeX font, "Computer Modern Roman", and were this the case here, then the axes labels would be beyond reproach. Although this book has been produced using LaTeX, the fonts employed all belong to the "Times" family, and so the raw TeX choice is discordant. The remedy is explained in Section 5.7.2, and its invocation is left as an exercise for the concerned reader. Finally, in lines 69–74 we estimate the empirical values of γ, finding values close to $\frac{1}{2}$ and 1 for the two methods. This is indicated in Figure 7.10. This code snippet could be shortened considerably, but it has been left in this form, so that it can be hacked to compute and display different information, and handle equations that are more complicated. The reader is encouraged strongly to use it as a starting point for further experiments.

When treating the conventional initial and boundary value problems we placed considerable emphasis on high-order high-accuracy algorithms. This is justified because there is an underlying assumption that we know both the equations and initial or boundary data exactly. These assumptions are rarely justified for stochastic equations, and so higher-order methods are uncommon.

8 Partial differential equations: a pseudospectral approach

In this chapter, we present two rather different topics, which we have linked together. First we shall study some initial value problems and initial-boundary value problems, where the forward-in-time integration is handled by the "method of lines". We can treat the spatial derivatives by one of two methods:

1 finite differencing, the standard approach, which is discussed in every textbook, and some aspects of which will be treated in the next chapter,
2 spectral methods which, for smooth solutions, will give near exponential accuracy.

For simplicity, we shall study only scalar partial differential equations in one spatial dimension, but everything can be generalized to systems of equations in two or more dimensions. We look first at Fourier methods capable of handling problems with periodic spatial dependence. Then in Section 8.6 we suggest a promising approach using Chebyshev transforms for spatial dependencies that are more general. Unfortunately, there is no pre-existing Python black box package to implement this, but there is legacy Fortran77 code which we list in Appendix B.

The second main topic of this chapter is to present the *numpy* `f2py` tool in Section 8.7, so that we can re-use the legacy code of Appendix B to construct Python functions to implement the ideas of Section 8.6. If you are interested in reusing legacy code, then you should study the ideas presented in Section 8.7, to find out how to do it, even if you have no interest in Chebyshev transforms.

8.1 Initial-boundary value problems

Essentially, all of the features we intend to describe here can be found in a single example, *Burgers' equation*. Suppose $u(t, x)$ is defined for $0 \leqslant t \leqslant T$ and $a \leqslant x \leqslant b$ and satisfies

$$u_t = -u\,u_x + \mu\,u_{xx}, \tag{8.1}$$

where $\mu > 0$ is a constant parameter. Here we are using a common notation for partial derivatives viz., $u_x = \partial u/\partial x$, $u_{xx} = \partial^2 u/\partial x^2$ etc. The first term on the right-hand side of (8.1) represents self-convection, while the second implements diffusion. There are very many physical processes where these two are the dominant effects, and they will be governed by some variant of *Burgers' equation*. In order to fix on a particular solution, we need to impose an initial condition

$$u(0,x) = f(x), \qquad a \leqslant x \leqslant b, \tag{8.2}$$

and one or more boundary conditions, e.g.,

$$u(t,a) = g_1(t), \quad u(t,b) = g_2(t), \qquad 0 \leqslant t \leqslant T. \tag{8.3}$$

The problem (8.1), (8.2), (8.3) is called an *initial-boundary value problem (IBVP)*. If we set both $a = -\infty$ and $b = \infty$, i.e., there are no boundaries, then we have an *initial value problem (IVP)*.

8.2 Method of lines

We can recast an evolution equation such as (8.1) as

$$u_t(t,x) = \mathcal{S}[u], \tag{8.4}$$

where $\mathcal{S}[u]$ is a functional of $u(t,x)$, which does not include t-derivatives of u. Here we could write it explicitly as

$$\mathcal{S}[u] = F(u(t,x), u_x(t,x), u_{xx}(t,x)) = -u\,u_x + \mu\,u_{xx}. \tag{8.5}$$

The main point is that for any given t we can compute the right-hand side of (8.4) if we know the values of $u(t,x)$ for the same t and all x.

We choose to regard (8.4) as an infinite collection of ordinary differential equations, one for each value of x, with t as the independent variable and initial data furnished by (8.2). Further, we replace the infinite set of x with $a \leqslant x \leqslant b$, by a representative finite set, and the spatial derivatives by some discrete approximation. This is the *method of lines (MoL)*. This means that we can utilize the experience and techniques that have already been built in the study of initial value problems for ordinary differential equations.

8.3 Spatial derivatives via finite differencing

Suppose we choose to represent the interval $a \leqslant x \leqslant b$ by a finite set of equidistant values $a = x_0 < x_1 < \cdots < x_N = b$, with spacing $dx = (b-a)/N$.

Assuming that $u(t,x)$ is say four times differentiable with respect to x, we have

$$u_x(t,x_n) = \frac{u(t,x_{n+1}) - u(t,x_{n-1})}{2dx} + O(dx^2), \tag{8.6}$$

and

$$u_{xx}(t,x_n) = \frac{u(t,x_{n+1}) - 2u(t,x_n) + u(t,x_{n-1})}{dx^2} + O(dx^2), \tag{8.7}$$

which gives the simplest method of computing $\mathcal{S}[u]$ in (8.4) for, e.g., Burgers' equation. Notice that this will only deliver $\mathcal{S}[u]$ for x_n with $1 \leqslant n \leqslant N-1$, i.e., not at the end points x_0 and x_N.

Now suppose we try to evolve $u(t, x)$ forwards in time by using, e.g., the Euler method with a time step dt. We can compute $u(t + dt, x_n)$ for interior points, but not $u(t + dt, a)$ ($n = 0$) or $u(t + dt, b)$ ($n = N$). This is precisely where the boundary conditions (8.3) come in, to furnish the missing values.

Of course, dt cannot be chosen arbitrarily. If we construct a linearized version of (8.1), then a stability analysis shows that for explicit time-stepping to be stable we need

$$dt < C dx^2 = O(N^{-2}), \tag{8.8}$$

where C is a constant of order unity. Since dx will be chosen small enough to satisfy accuracy requirements, the restriction (8.8) on dt makes explicit time-stepping unattractive in this context. Implicit time-stepping needs to be considered, but this too is unattractive if there are significant non-linearities, because we have to solve a sequence of non-linear equations.

8.4 Spatial derivatives by spectral techniques for periodic problems

Spectral methods offer a useful alternative to finite differencing. In order to illustrate the ideas, we first consider a special case where the spatial domain $[a, b]$ has been mapped to $[0, 2\pi]$ and we are assuming that $u(t, x)$ is 2π-periodic in x. If we denote the Fourier transform of $u(t, x)$ with respect to x by $\widetilde{u}(t, k)$, then, as is well known, the Fourier transform of $d^n u/dx^n$ is $(ik)^n\widetilde{u}(t, k)$ and so by Fourier inversion we can recover the spatial derivatives of $u(t, x)$. Thus for Burgers' equation (8.1), we need one Fourier transform and two inversions to recover the two derivatives on the right-hand side. We need to turn this into a spectral algorithm.

Suppose we represent $u(t, x)$ for each t by function values at N equidistant points on $[0, 2\pi)$. We can construct the *discrete Fourier transform (DFT)* which, loosely speaking, is the first N terms in the Fourier series expansion of $u(t, x)$. We insert the appropriate multiplier for each term and then compute the inverse DFT. The precise details are well documented, see e.g., Boyd (2001), Fornberg (1995), Hesthaven et al. (2007), Press et al. (2007) or Trefethen (2000). Unfortunately, different authors have different conventions, and it is tedious and error-prone to translate between them. At first sight, this approach might seem to be of academic interest only. Because each DFT is a linear operation, it can be implemented as a matrix multiplication which requires $O(N^2)$ operations, and so the evaluation of the right-hand side of (8.1) requires $O(N^2)$ operations, while finite differencing requires only $O(N)$.

If we know, or guess, that $u(t, x)$ is *smooth*, i.e., arbitrarily many x-derivatives exist and are bounded, then the truncation error in the DFT is $o(N^{-k})$ for arbitrarily large k. In practice, our algorithm is likely to return errors of order $O(10^{-12})$ for $N \approx 20$. We would need $N \sim 10^6$ to achieve the same accuracy with a finite differencing approach. If N has only small prime factors, e.g., $N = 2^M$, then the DFT and its inverse can be computed using *fast Fourier transform (FFT)* techniques, which require $O(N \log N)$ rather than $O(N^2)$ operations. Of course, it is crucial here that $u(t, x)$ be periodic as well as smooth. If $u(t, 0) \neq u(t, 2\pi)$, then the periodic continuation would be discontinuous, and the Gibbs' phenomenon would wreck all of these accuracy estimates.

The question now arises as to whether to implement the DFT using matrix multiplication or the FFT approach. If $N \lesssim 30$, then matrix multiplication is usually faster. All the techniques needed to construct a *numpy* implementation have been outlined already and the interested reader is invited to construct one[1]. For larger values of N an efficient implementation of the FFT approach is needed, and because this involves important new ideas, most of the second half of this chapter is devoted to it. Most of the standard FFT routines for DFT operations are available in the `scipy.fftpack` module, and in particular there is a very useful function of the form `diff(u,order=1,period=2*pi)`. If `u` is a *numpy* array representing the evenly spaced values of $u(x)$ on $[0, 2\pi)$, then the function returns an array of the same shape as `u` containing the values of the first derivative for the same x-values. Higher derivatives and other periodicities are handled by the shown parameters.

We consider a concrete example. Let $f(x) = \exp(\sin(x))$ for $x \in [0, 2\pi)$, and compute df/dx on $[0, 2\pi)$. The following code snippet implements a comparison of finite differencing and the spectral approach.

```
import numpy as np
from scipy.fftpack import diff

def fd(u):
    """ Return 2*dx* finite-difference x-derivative of u. """
    ud=np.empty_like(u)
    ud[1:-1]=u[2: ]-u[ :-2]
    ud[0]=u[1]-u[-1]
    ud[-1]=u[0]-u[-2]
    return ud

for N in [4,8,16,32,64,128,256]:
    dx=2.0*np.pi/N
    x=np.linspace(0,2.0*np.pi,N,endpoint=False)
    u=np.exp(np.sin(x))
    du_ex=np.cos(x)*u
    du_sp=diff(u)
    du_fd=fd(u)/(2.0*dx)
    err_sp=np.max(np.abs(du_sp-du_ex))
    err_fd=np.max(np.abs(du_fd-du_ex))
    print "N=%d, err_sp=%.4e err_fd=%.4e" % (N,err_sp,err_fd)
```

Table 8.1 shows the output from the code snippet. The infinity norm of the finite differencing error decreases by a factor of about 4 for each doubling of the number of points, i.e., the error $= O(N^{-2})$, consistent with the error estimate in (8.6). The spectral error

[1] Boyd (2001) is one of many references which contain specific algorithms.

squares for each doubling, until N is "large", i.e., $N \gtrsim 30$ here. This fast error decrease is often called *exponential convergence*. Then two effects degrade the accuracy. With software producing arbitrary accuracy, we might have expected the error to be $O(10^{-30})$ for $N = 64$. But in normal use, most programming languages handle floating-point numbers with an accuracy of about one part in 10^{16}, and so we cannot hope to see the tiny errors. Also the eigenvalues of the equivalent "differentiation matrix" become large, typically $O(N^{2p})$ for p-order differentiation, and this magnifies the effect of rounding errors, as can be seen here for $N \gtrsim 60$.

Table 8.1 The maximum error in estimating the spatial derivative of $\exp(\sin x)$ on $[0, 2\pi)$ as a function of N, the number of function values used. Doubling the number of points roughly squares the spectral error, but only decreases the finite difference error by a factor of 4. For very small spectral errors, i.e., very large N, this rule fails for artificial reasons explained in the text.

N	Spec. error	FD error
4	1.8×10^{-1}	2.5×10^{-1}
8	4.3×10^{-3}	3.4×10^{-1}
16	1.8×10^{-7}	9.4×10^{-2}
32	4.0×10^{-15}	2.6×10^{-2}
64	9.9×10^{-15}	6.5×10^{-3}
128	2.2×10^{-14}	1.6×10^{-3}
256	5.8×10^{-14}	4.1×10^{-4}

8.5 The IVP for spatially periodic problems

It should be noted for spatially periodic problems that if $u(t, x)$ is specified for fixed t and $x \in [0, 2\pi)$, then using the techniques of the previous section we can compute the x-derivatives of u on the same interval, without needing boundary conditions like (8.3), because $u(t, 2\pi) = u(t, 0)$. This is used in lines 8 and 9 of the snippet above. Therefore, when considering spatially periodic problems, only the initial value problem is relevant.

When we use the method of lines with say an explicit scheme, we need to consider carefully the choice of time interval dt we are going to use. Suppose, for simplicity, we are considering a linear problem. Then the $\mathcal{S}[u]$ of (8.5) will be a linear functional which we can represent by a $N \times N$ matrix A. We may order the (in general complex) eigenvalues by the magnitude of their absolute value. Let Λ be the largest magnitude. We know from the discussion in Section 7.2 that the stability of explicit schemes for the time integration will require $\Lambda dt \lessapprox O(1)$. Now if the highest order of spatial derivative occurring in $\mathcal{S}[u]$ is p, then, for finite differencing, calculations show that $\Lambda = O(N^p)$. A common case is the parabolic equation for heat conduction where $p = 2$. Since $N \gg 1$ to ensure spatial accuracy, the stability requirement $dt = O(N^{-2})$. This may be unacceptably small, and so implicit time integration schemes are often used. If instead we use the spectral scheme for evaluating spatial periodic derivatives, then it turns out that a similar stability estimate obtains. (When we consider non-periodic problems later in this chapter, then we can show that $\Lambda = O(N^{2p})$, leading to $dt = O(N^{-2p})$, which, at first sight, might rule out explicit time integration schemes.) However, we saw in the last section, that the values of N needed for a given spatial accuracy, are much smaller than those required for finite differencing. Further, because of the phenomenal accuracy achievable with even modest values of N, a small value of dt will be needed anyway to achieve comparable temporal accuracy.

Once we have decided on a suitable size for dt we need to choose a, preferably explicit, scheme for the time integration process. Here the literature cited above has plenty of advice, but for many purposes Python users do not need to make a choice, but can rely instead on the `odeint` function of Section 7.3, as we show by means of a very simple example. It is difficult to construct a non-linear initial value problem which is periodic in space and so we consider linear advection

$$u_t = -2\pi u_x, \qquad u(0,x) = \exp(\sin x), \tag{8.9}$$

for which the exact solution is $u(t,x) = \exp(\sin(x - 2\pi t))$. The following snippet carries out a time integration and performs two common tasks, displaying the solution and computing the final error. Most of the frills, comments etc. have been omitted for the sake of brevity.

```python
import numpy as np
from scipy.fftpack import diff
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def u_exact(t,x):
    """ Exact solution. """
    return np.exp(np.sin(x-2*np.pi*t))


def rhs(u, t):
    """ Return rhs. """
    return -2.0*np.pi*diff(u)

N=32
x=np.linspace(0,2*np.pi,N,endpoint=False)
u0=u_exact(0,x)
t_initial=0.0
t_final=64*np.pi
t=np.linspace(t_initial,t_final,101)
sol=odeint(rhs,u0,t,mxstep=5000)

plt.ion()
```

```
25 fig=plt.figure()
26 ax=Axes3D(fig)
27 t_gr,x_gr=np.meshgrid(x,t)
28 ax.plot_surface(t_gr,x_gr,sol,alpha=0.5)
29 ax.elev,ax.azim=47,-137
30 ax.set_xlabel('x')
31 ax.set_ylabel('t')
32 ax.set_zlabel('u')
33
34 u_ex=u_exact(t[-1],x)
35 err=np.abs(np.max(sol[-1,: ]-u_ex))
36 print "With %d Fourier nodes the final error = %g" % (N, err)
```

Lines 7–21 set up the problem and line 22 obtains the solution. Lines 24–32 cope with the visualization. (Note the transposed arguments in line 27, see Section 4.2.2.) Lines 34–36 construct a minimal error check. The graphical output of this snippet is displayed

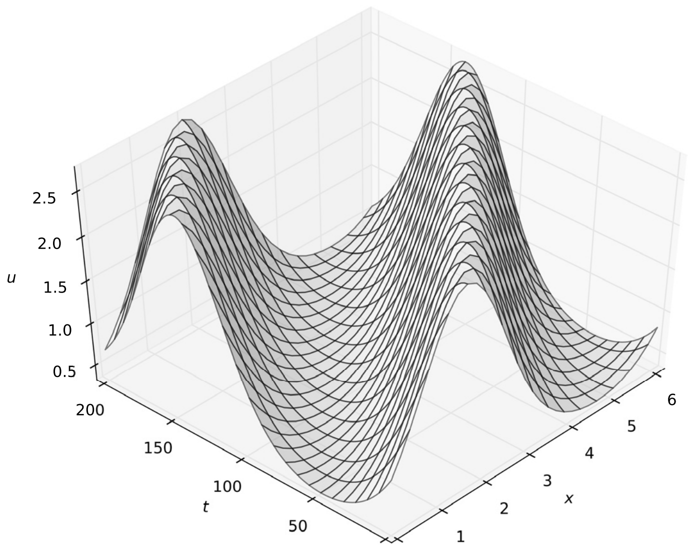

Figure 8.1 The numerical solution $u(t, x)$ of the periodic linear advection problem (8.9) for $x \in [0, 2\pi)$ and $t \in [0, 64\pi]$.

as Figure 8.1. This figure illustrates the virtue of the pseudospectral approach. Even a relatively coarse grid spacing can produce a smooth result, and the final error here is about 10^{-4}. It would require a much finer grid, with the associated work overhead, to produce a comparable result using purely finite difference techniques.

8.6 Spectral techniques for non-periodic problems

The vast majority of interesting initial-boundary value problems do not exhibit spatial periodicity, and so the Fourier-series-related spectral techniques described above cannot be applied. An alternative approach might seem to be approximation of functions by means of polynomials.

For the sake of simplicity, let us map the interval $a \leqslant x \leqslant b$ to $-1 \leqslant x \leqslant 1$, e.g., by a linear transformation. Next we choose a set of $N+1$ discrete points $-1 = x_0 < x_1 < \cdots < x_N = 1$. We now try to obtain a useful polynomial approximation to a function $u(x)$. It is easy to see that there exists a unique polynomial $p_N(x)$ of order N which interpolates u, i.e., $p_N(x_n) = u(x_n)$ for all n in $[0, N]$. Indeed, there exists a pair of *numpy* functions `polyfit` and `poly1d` (see Section 4.7) which carry out these calculations. Assuming uniform convergence we could differentiate the interpolating polynomial to provide an estimate of $u'(x)$ at the grid points.

Surprisingly, for uniformly spaced grid points, the assumed convergence does not occur. This is known as the *Runge effect*, which is exemplified by the following code snippet, which examines the perfectly smooth function $u(x) = 1/(1+25x^2)$ for $x \in [0, 1]$, whose values range between 1/26 and 1.

```
1  import numpy as np
2
3  def u(x): return 1.0/(1.0+25.0*x**2)
4
5  N=20
6  x_grid=np.linspace(-1.0,1.0,N+1)
7  u_grid=u(x_grid)
8  z=np.polyfit(x_grid,u_grid,N)
9  p=np.poly1d(z)
10 x_fine=np.linspace(-1.0, 1.0, 5*N+1)
11 u_fine=p(x_fine)
```

In line 8, the array `z` is filled with the coefficients of the interpolating polynomial, and `p` created in line 9 is a function which returns the interpolating polynomial itself. `p(x)` agrees with `u(x)` at the grid points. However, if we create a finer grid in line 10 and evaluate `p(x)` on it, then the values obtained range between −58 and +4. This polynomial is useless for approximating the function! Increasing the value of N does not help.[2]

Fortunately, the situation improves markedly if we turn attention to certain grids with non-uniform spacing. Suppose we build first a uniform θ-grid on $[0, \pi]$, and use the transformation $x = -\cos\theta$ to construct the non-uniform *Chebyshev grid nodes* on $[-1, 1]$

$$\theta_k = \frac{k\pi}{N}, \qquad x_k = -\cos\theta_k, \qquad k \in [0, N]. \tag{8.10}$$

[2] For continuous but non-differentiable functions, the situation is much worse, e.g., if $u(x) = |x|$, we obtain convergence at $x = 0, \pm 1$ and at no other points!

Let $Q_k(x)$ be the Nth-order polynomial in x defined by

$$Q_k(x) = \frac{(-1)^k}{Nc_k} \frac{\sin\theta \sin N\theta}{(\cos\theta_k - \cos\theta)}, \tag{8.11}$$

where $c_k = 1$ for $j = 1, 2, \ldots, N-1$, while $c_0 = c_N = 2$. Then for $0 \leqslant j, k \leqslant N$ it is straightforward to show that

$$Q_k(x_j) = \delta_{jk}. \tag{8.12}$$

Let $f(x)$ be an absolutely continuous function on $[-1, 1]$ and set $f_n = f(x_n)$. It follows that the Nth-order polynomial which interpolates $f(x)$ at the Chebyshev nodes is

$$f_N(x) = \sum_{k=0}^{N} f_k Q_k(x) \quad \text{where } f_k = f(x_k). \tag{8.13}$$

Then it can be shown that for every absolutely continuous function $f(x)$ on$[-1, 1]$, the sequence of interpolating polynomials $f_N(x)$ converges to $f(x)$ uniformly as N increases. (Note that we do **not** require $f(-1) = f(1)$.) These are the ones we shall use.

Next we need to consider how to transform between $f(x)$ and the set of $\{f_k\}$. This is most economically performed using a specific choice of orthogonal polynomials. That choice depends on the norms chosen, but if as here we use the maximum (C^∞) norm, then the optimal choice is the set of *Chebyshev polynomials* $\{T_n(x)\}$

$$T_n(x) = \cos(n\theta) = \cos(n \arccos(-x)), \tag{8.14}$$

see (8.10), Boyd (2001), Fornberg (1995). There is another compelling reason for this choice. In general, the number of operations required to estimate a derivative will be $O(N^2)$. However, it is easy to see that $T_m(x_n) = \cos(mn\pi/N)$. Thus if we use Chebyshev polynomials evaluated at Chebyshev nodes then the transformation, from physical to Chebyshev space, its inverse and the process of differentiation in Chebyshev space can be handled using the discrete fast Fourier cosine transform, which takes $O(N \log N)$ operations. Equally importantly, we may expect *exponential convergence* for smooth functions.

Besides being a very useful reference for the theoretical ideas we have used in this section, Fornberg (1995) contains, in its Appendix A, detailed code in Fortran77 for the practical implementation of these ideas. The philosophy of this book is to try to avoid programming in compiled languages. How then are we to proceed? Two strategies are:

1 We could try to figure out what these Fortran subroutines and functions do, and then try to implement them in Python. Unfortunately, only the base FFT function is available from *scipy*, and it uses a different convention.
2 We could try to implement them using the original Fortran77 code, but with a wrapper around them which makes them look and behave like Python functions.

This second approach is the one we shall use here. Besides solving the immediate current problem, we shall see how, in general, one can reuse legacy Fortran code. (This procedure was already used to generate many of the *scipy* functions.) The next two sections discuss how to wrap pre-existing Fortran code, and the discussion of spectral methods continues in the one after that.

8.7 An introduction to f2py

The f2py tool was originally in *scipy* but as it matured it has gravitated to *numpy*. You can check that it is installed, and obtain a summary of its options by typing in the command line `f2py --verbose`. What does it do? Well originally it was designed to compile Fortran77 subroutines and place a wrapper around the result so that it could be used as a Python function. Later that was extended to C functions and more recently to those parts of Fortran90[3] which could have been coded in Fortran77. The philosophy of this book is to use Python wherever possible, so why should f2py be of interest? There are two major applications which could be useful for the intended readership, who I assume have no knowledge of Fortran.

1. There could be, as here, legacy code, usually in the form of Fortran77 subroutines, that we would like to reuse, even if our knowledge of Fortran is minimal
2. When developing a major project, it is advisable to move the time-expensive number-crunching operations into simple Python functions. An example of arranging a Python project to facilitate this approach is given in Chapter 9. If the Python profiler shows that they are too slow, we can then recode the operations as simple Fortran subroutines, compile and wrap them using f2py and replace the slow Python functions with the superfast ones.

Unfortunately, f2py documentation for beginners is patchy. The official documentation[4] is somewhat out of date. There are various third-party accounts available on the internet, none of them entirely satisfactory. However, for most scientists the tasks enumerated above can be accomplished using a small number of concepts, which are illustrated in the following examples.

8.7.1 Simple examples with scalar arguments

We start with a very simple task (for which the Python functions already exist) to explain the f2py process. Given three components of a vector, u, v and w, compute the Euclidean norm $s = \sqrt{u^2 + v^2 + w^2}$. We start with a Fortran90 listing.

```fortran
! File: norm3.f90 A simple subroutine in f90

subroutine norm(u,v,w,s)
real(8), intent(in) :: u,v,w
real(8), intent(out) :: s
s=sqrt(u*u+v*v+w*w)
end subroutine norm
```

Comments in Fortran 90 follow an exclamation mark (!). We need to know that all of the input arguments and output identifiers have to be supplied as input arguments to a

[3] By Fortran90, we include implicitly also many elements of Fortran95.
[4] See `http://cens.ioc.ee/projects/f2py2e/`.

subroutine, lines 3–8 of the snippet. Line 4 declares that three arguments are **real**(8) (roughly a Python float) and are to be regarded as input variables. Line 5 declares s to be a real, but more importantly it delivers a result which can be accessed from the calling programme.

It is important to supply f2py with syntactically correct Fortran. We check this by trying to compile the code, using a command line like

```
gfortran norm3.f90
```

(You should use the name of your compiler if it differs from mine.) If there are syntax errors, they will be described, after which the compiler should complain about undefined “symbols”. Make sure that the syntax errors are eliminated. Now try

```
f2py -c --fcompiler=gfortran norm3.f90 --f90flags=-O3 -m normv3
```

or more simply, if you only have the one compiler,

```
f2py -c norm3.f90 --f90flags=-O3 -m normv3
```

or, if you do not care about optimized code,

```
f2py -c norm3.f90 -m normv3
```

You should find a file called normv3.so in your current directory.[5] Now fire up the *IPython* interpreter and type import normv3 followed by normv3? You will find that module *normv3* contains a function *norm*, so try normv3.norm? This function looks like a Python one which accepts three floats and produces one. Try it out with, e.g.,

```
print normv3.norm(3,4,5)
```

which will produce 7.07106781187 as expected.

It is highly illuminating to consider the identical problem coded in Fortran 77. In that language, comments are preceded by a C in column 1; the body of the code lies between columns 7 and 72. Here is the equivalent code snippet, which we assume is available in a file norm3.f.

```
C     FILE NORM3.F A SIMPLE SUBROUTINE IN F77

      SUBROUTINE NORM(U,V,W,S)
      REAL*8 U,V,W,S
      S=SQRT(U*U+V*V+W*W)
      END
```

Next delete normv3.so and feed this file to f2py via the line[6]

```
f2py -c --fcompiler=gfortran norm3.f --f77flags=-O3 -m normv3
```

[5] A file called, e.g., foo.o is an *object file*, created by most compilers. foo.so would be a *shared object file* shared by more than one language.

[6] The shortened command line versions given on the previous page also work here.

and within *IPython* import `normv3` and inspect the new file via `normv3.norm?`. We have a problem! The function `normv3.norm` expects four input arguments. A moment's thought reveals the problem. There is no way in Fortran77 to specify "out" variables. Here though `f2py` has the solution. Amend the code above to

```
C     FILE NORM3.F A SIMPLE SUBROUTINE IN F77

      SUBROUTINE NORM(U,V,W,S)
      REAL*8 U,V,W,S
Cf2py intent(in) U,V,W
Cf2py intent(out) S
      S=SQRT(U*U+V*V+W*W)
      END
```

We have inserted Fortran90-like directives into Fortran77 code via comment statements. Now delete the old `normv3.so` file and create a new one. The resulting `normv3.norm` function has the expected form and behaves identically to its Fortran90 predecessor.

8.7.2 Vector arguments

The next level of complexity is when some arguments are vectors, i.e., one-dimensional arrays. The case of scalar input and vector output is dealt with in the next section, so we consider here vector input and scalar output. We continue with our norm example extending it to `N`-dimensional vectors `U`. For brevity, we discuss here only the Fortran77 version, as given by the following snippet.

```
 1  C     FILE NORMN.F  EXAMPLE OF N-DIMENSIONAL NORM
 2
 3        SUBROUTINE NORM(U,S,N)
 4        INTEGER N
 5        REAL*8 U(N)
 6        REAL*8 S
 7  Cf2py intent(in) N
 8  Cf2py intent(in) U
 9  Cf2py depend(U) N
10  Cf2py intent(out) S
11        REAL*8 SUM
12        INTEGER I
13        SUM = 0
14        DO 100 I=1,N
15   100     SUM = SUM + U(I)*U(I)
16        S = SQRT(SUM)
17        END
```

Ignore for the moment lines 7–10. Unlike Python, Fortran arrays do not know their size, and so it is mandatory to supply both `U` and `N` as arguments, and we need of course `S`, the output argument. Lines 11–16 carry out the calculation. The construction in lines 14 and 15 is a do-loop, the Fortran equivalent of a for-loop. The subroutine arguments look strange to a Python user, but we can remedy this. Lines 7, 8 and 10 merely declare the input and output variables as before. Line 9 is new. It tells f2py that N and U are related and that the value of N can be inferred from that for U. We now create a new module with, e.g.,

```
f2py -c --fcompiler=gfortran normn.f --f77flags=-O3 -m normn
```

and within *IPython* import normn and inspect the function normn.normn. We find that it requires u as an input argument, and n is an optional input argument, while s is the output argument. Thus

```
print normn.norm([3,4,5,6,7])
```

works just like a Python function. You should try also normn.norm([3,4,5,6,7],3). The adventurous reader might like to test normn.norm([3,4,5,6,7],6).

8.7.3 A simple example with multi-dimensional arguments

We need to point out an important difference between Python and Fortran. Suppose a is a 2×3 array. In Python, its components would be stored linearly as ("row order"), just as they would in the C family of languages,

$$a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12} \; ,$$

but in the Fortran family the order would be ("column order")

$$a_{00}, a_{10}, a_{01}, a_{11}, a_{02}, a_{12} \; .$$

For vectors, i.e., one-dimensional arrays, there is of course no difference. However, in the more general case there may be a need for copying an array from one data format to the other. This is potentially time-consuming if the arrays are large.

Usually there is no need to worry about how the arrays are stored in memory. Modern versions of the f2py utility do a very good job of managing the problem, minimizing the number of copies, although it is hard to avoid at least one copy operation. Here are two very simple ways to minimize copying.

1. If the Fortran is expecting a **real***8 array, make sure that Python supplies a *numpy* array of float. (In the example above, we supplied a *list* of integers, which certainly causes copying!) If you are using the np.array function, see Section 4.2, you can actually specify order='F' to get round the problem.
2. Although it is a common Fortran idiom to input, modify and output the same large array in a single subroutine, this often leads to programming errors, and also to copying in f2py. A more explicit Pythonic style, which leads to a single copy, is to be recommended.

Both of these ideas are illustrated in the following snippet and *IPython* session.

```
C     FILE: MODMAT.F MODIFYING A MULTIDIMENSIONIAL ARRAY

      SUBROUTINE MMAT(A,B,C,R,M,N)
      INTEGER M,N
      REAL*8 A(M,N),B(M),C(N),R(M,N)
Cf2py intent(in) M,N
Cf2py intent(in) A,B,C
Cf2py depend(A) M,N
Cf2py intent(out) R
      INTEGER I,J
      DO 10 I=1,M
        DO 10 J=1,N
 10       R(I,J)=A(I,J)
      DO 20 I=1,M
 20     R(I,1)=R(I,1)+B(I)
      DO 30 J=1,N
 30     R(1,J)=R(1,J)+C(J)
      END
```

In words, array R is a copy of A with the first column increased by adding B and the first row increased by adding C. Suppose this snippet is compiled to a module modmat using f2py. Then consider the following *IPython* session

```
import numpy as np
import modmat

a = np.array([[1,2,3],[4,5,6]],dtype='float',order='F')
b=np.array([10,20],dtype='float')
c=np.array([7,11,13],dtype='float')
r=modmat.mmat(a,b,c)
r
```

8.7.4 Undiscussed features of f2py

This section has treated only a few aspects of f2py, but those which are likely to be the most useful for scientist readers We have discussed only Fortran code, but most features work for C code also. We have not discussed the more advanced features such as strings and common blocks, because they are unlikely to be used in the envisaged contexts. Finally, we have not discussed *signature files*. In some circumstances, e.g., proprietary Fortran libraries, the user may compile legacy files but not modify them with the Cf2py lines. We can obtain the required module by a two-stage process. First, we run f2py -h to generate a signature file, e.g., modmat.pyf This is written in a Fortran90 style and gives the default shape of the Fortran subroutine, here mmat without the Cf2py lines.

We then edit the signature file to create the function we want. In the second stage, f2py takes the signature and Fortran files and creates the required shared object file. The details are explained in the f2py documentation.

8.8 A real-life f2py example

The examples presented in the previous section are of course over-simplified. (The author has to strike a balance between informing and confusing the reader.) In this section, we look at a more realistic task of turning legacy Fortran code into a useful suite of Python functions, presuming only a minimal knowledge of Fortran. Clearly, only one such task can be incorporated into this book, and so it is unlikely that the reader's favourite application will be covered. Nevertheless, it is very worthwhile to follow the process carefully, so as to see how to apply the same procedure to other applications. Our example is the task set aside in Section 8.6, to implement the Chebyshev transform tools described in Fornberg (1995) and given in legacy Fortran code there.

We start by typing or copying the first subroutine from the code in Appendix B to a file, say cheb.f. Actually, this is not given entirely in Fornberg (1995). The working lines 8–10 are given as a snippet at the start of section F.3 on page 187 of that reference, and the earlier lines 1–4 and later lines 11 and 12 are confected using the code examples given earlier on that page. As the comment line 2 makes clear, N is an input argument, X is an output one, and it refers to a vector of dimension N+1. We need to help f2py with the comment lines 5–7.

We move next to the second code snippet, the basic fast discrete Fourier transform **SUBROUTINE** FFT which runs (before enhancements) to 63 lines of rather opaque code. Comment lines 10–12 reveal a common Fortran idiom. The arrays A and B are input variables which will be altered in situ and then become output variables. Fortunately, f2py is equipped to handle this, and line 21 deals with the problem. As can be seen in lines 14–17, the integer arguments IS and ID are input variables, and so line 22 reminds f2py of this. Line 16 informs us that N is the actual dimension of the arrays A and B. As in the previous section, we could handle this with a line

```
Cf2py depend(A) N
```

which would treat n as an optional argument to the corresponding Python function. However, there is no conceivable case in which we would want to use this optional value of n. Instead, line 23 states that n will not appear in the final argument list, and will instead be inferred[7] from the dimension of the input array a. The third code snippet, the fast discrete Fourier cosine transform, is handled in an identical manner.

Finally, we need to add the three subroutines from pages 188–189 of Fornberg (1995). They do not contain useful comment lines, but the commentary on page 188 explains the nature of the subroutine arguments. The enhanced comment lines include no new ideas.

[7] Note that **intent**(hide) is not a legal statement in Fortran90.

If all of this Fortran77 code has been typed or copied into a file, say `cheb.f`, then we can use `f2py` to create a module, say `cheb`. (It might well be worth including the code optimization flag here.) We should next use *IPython*'s introspection facility to check that the function signatures are what we would reasonably expect.

It is of course extremely important to verify the resulting Python functions. The following snippet uses all of the routines (`fft` and `fct` only indirectly) and gives a partial check that all is well.

```python
import numpy as np
import cheb
print cheb.__doc__

for N in [8,16,32,64]:
    x= cheb.chebpts(N)
    ex=np.exp(x)
    u=np.sin(ex)
    dudx=ex*np.cos(ex)

    uc=cheb.tocheb(u,x)
    duc=cheb.diffcheb(uc)
    du=cheb.fromcheb(duc,x)
    err=np.max(np.abs(du-dudx))
    print "with N = %d error is %e" % (N, err)
```

With $N = 8$ the error is $O(10^{-3})$ dropping to $O(10^{-8})$ for $N = 16$ and and $O(10^{-14})$ for $N = 32$.

This completes our construction of a suite of functions for carrying out spatial differentiation in Chebyshev space. The same principles could be used to extend Python in other directions. We can wrap Fortran90 or 95 code in exactly the same way, provided we remember that comments start (not necessarily at the beginning of the line) with an exclamation mark (`!`) rather than a `C`. Well-written Fortran90 code will include specifications like `intent(in)`, but there is no harm in including the `f2py` version, provided both are consistent with each other.

8.9 Worked example: Burgers' equation

We now consider a concrete example of pseudospectral methods, the initial-boundary value problem for Burgers' equation (8.1),

$$u_t = -u\, u_x + \mu\, u_{xx}, \tag{8.15}$$

where μ is a positive constant parameter. Here the time development of $u(t, x)$ is controlled by two effects, the non-linear advection represented by the first term on the right-hand side of (8.15), and the linear diffusion represented by the second term. A number of exact solutions are known, and we shall use the *kink* solutions

$$\widehat{u}(t,x) = c\left[1 + \tanh\left(\frac{c}{2\mu}(ct - x)\right)\right], \tag{8.16}$$

where c is a positive constant, to test our numerical approach. (For fixed t and large negative x, $u \approx 2c$, while if x is large and positive, $u \approx 0$. There is a time-dependent transition between two uniform states, hence the name *kink*.)

We shall restrict consideration to the intervals $-2 \leqslant t \leqslant 2$ and $-1 \leqslant x \leqslant 1$. We shall need to impose initial data at $t = -2$, and so we set

$$u(-2,x) = f(x) = c\left[1 - \tanh\left(\frac{c}{2\mu}(x + 2c)\right)\right], \tag{8.17}$$

consistent with (8.16).

8.9.1 Boundary conditions: the traditional approach

Notice that the traditional approach of estimating spatial derivatives by finite differencing, (8.6)–(8.7), fails to deliver values at the end points of the spatial interval. We need more information to furnish these values. (The earlier assumption of periodicity circumvented this problem.) Mathematically, because this equation involves two spatial derivatives we need to impose two boundary conditions at $x = \pm 1$, $-2 \leqslant t \leqslant 2$, in order that the solution to the initial-boundary value problem is unique. Here we follow the example of Hesthaven et al. (2007) and require as an illustrative case

$$\mathcal{B}_1[t,u] \equiv u^2(t,-1) - \mu u_x(t,-1) - g_1(t) = 0, \quad \mathcal{B}_2[t,u] \equiv \mu u_x(t,1) - g_2(t) = 0, \tag{8.18}$$

for $-2 \leqslant t \leqslant 2$. Here $g_1(t)$ and $g_2(t)$ are arbitrary functions. (For the sake of definiteness, we shall later choose them so that the conditions (8.18) are exactly satisfied by the exact solution (8.16).) Implementing the finite difference method to deal with boundary conditions such as (8.18) is dealt with in the standard textbooks, and will not be repeated here. Subject to stability and convergence requirements, discrete changes in $u(t,x)$ at the boundary grid points can easily be accommodated in a finite difference scheme. More care is needed in a pseudospectral scheme to avoid the occurrence of the Gibbs phenomenon due to a perceived discontinuity.

8.9.2 Boundary conditions: the penalty approach

The pseudospectral approach estimates spatial derivatives at every point of the grid. So how are we to modify it to take account of boundary conditions like (8.18)? The so-called "penalty method" offers a very versatile and remarkably successful method of implementing such boundary conditions. At first sight, it looks to be absurd. Instead of solving (8.15) subject to boundary conditions (8.18) and initial conditions (8.17), we solve

$$u_t = -uu_x + \mu u_{xx} - \tau_1 S_1(x)\mathcal{B}_1[t,u] - \tau_2 S_2(x)\mathcal{B}_2[t,u], \tag{8.19}$$

subject only to the initial conditions (8.17). Here τ_i are constants, and $S_i(x)$ are suitable functions of x. It appears that we solve neither Burgers' equation nor the boundary conditions, but a combination of the two. The basic idea is that if the τ_i are chosen large enough, then any deviations of $u(t, x)$ from a solution of the boundary conditions (8.19) will be reduced to zero exponentially quickly. Indeed, the $\tau_i \to \infty$ limit of (8.19) is the boundary condition (8.18). But a solution of (8.19) which satisfies the boundary conditions also satisfies the evolution equation (8.15). This idea is not new, nor is it restricted to boundary conditions. For example, it appears in numerical general relativity, where the evolution-in-time equations are subject to elliptic (constant-time) constraints as "constraint damping".

The penalty method applied to boundary conditions takes a surprisingly simple form in the pseudospectral approach being advocated here. As pointed out first by Funaro and Gottlieb (1988) and later by Hesthaven (2000), it is advantageous to choose $S_1(x)$ in (8.19) to be the $Q_0(x)$ defined by (8.11), and similarly to replace $S_2(x)$ by $Q_N(x)$. Recalling equation (8.12), we see that as far as the grid approximation is concerned, we are imposing the left (right) boundary condition at the leftmost (rightmost) grid point, and the interior grid points do not "see" the boundary conditions. Detailed calculations for linear problems suggest that the method will be stable and accurate if the parameters τ_i are large, $O(N^2)$.

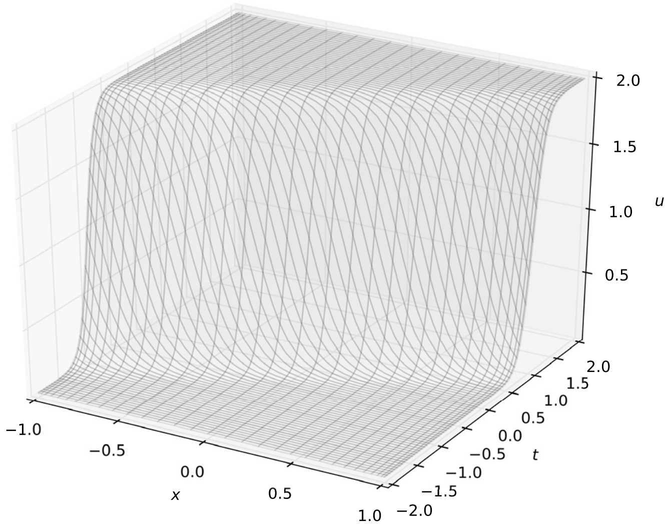

Figure 8.2 The numerical evolution of a "kink" solution (8.16) of the Burgers' equation (8.15) treated as an initial-boundary value problem.

Making due allowance for the more intricate equation and the need to impose boundary conditions, the code snippet given below for solving Burgers' equation for the kink solution is hardly more complicated than that used to solve the linear advection equation with periodic boundary conditions in Section 8.5. Note that the snippet computes the numerical solution, displays it graphically and estimates the error.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import cheb

c=1.0
mu=0.1

N=64
x=cheb.chebpts(N)
tau1=N**2
tau2=N**2
t_initial=-2.0
t_final=2.0

def u_exact(t,x):
    """ Exact kink solution of Burgers' equation. """
    return c*(1.0+np.tanh(c*(c*t-x)/(2*mu)))

def mu_ux_exact(t,x):
    """ Return mu*du/dx for exact solution. """
    arg=np.tanh(c*(c*t-x)/(2*mu))
    return 0.5*c*c*(arg*arg - 1.0)

def f(x):
    """ Return initial data. """
    return u_exact(t_initial,x)

def g1(t):
    """ Return function needed at left boundary. """
    return (u_exact(t,-1.0))**2-mu_ux_exact(t,-1.0)

def g2(t):
    """ Return function needed at right boundary. """
    return mu_ux_exact(t,1.0)

def rhs(u, t):
    """  Return du/dt. """
    u_cheb=cheb.tocheb(u,x)
    ux_cheb=cheb.diffcheb(u_cheb)
```

```
42      uxx_cheb=cheb.diffcheb(ux_cheb)
43      ux=cheb.fromcheb(ux_cheb,x)
44      uxx cheb. fromcheb(uxx_cheb,x)
45      dudt=-u*ux+mu*uxx
46      dudt[0]-=tau1*(u[0]**2-mu*ux[0]-g1(t))
47      dudt[-1]-=tau2*(mu*ux[-1]-g2(t))
48      return dudt
49
50  t=np.linspace(t_initial, t_final,81)
51  u_initial=f(x)
52  sol=odeint(rhs,u_initial,t,rtol=10e-12,atol=1.0e-12,mxstep=5000)
53  xg,tg=np.meshgrid(x,t)
54  ueg=u_exact(tg,xg)
55  err=sol-ueg
56  print "With %d points error is %e" % (N,np.max(np.abs(err)))
57
58  plt.ion()
59  fig=plt.figure()
60  ax=Axes3D(fig)
61  ax.plot_surface(xg,tg,sol,rstride=1,cstride=2,alpha=0.1)
62  ax.set_xlabel('x',style='italic')
63  ax.set_ylabel('t',style='italic')
64  ax.set_zlabel('u',style='italic')
```

Table 8.2 The maximum absolute error as a function of Chebyshev grid size N. The *relative* sizes are of interest here.

N	max. abs. error
16	1.1×10^{-2}
32	4.7×10^{-5}
64	1.2×10^{-9}
128	4.0×10^{-11}
256	5.7×10^{-10}

Lines 7 and 8 set up the parameters for the exact solution, and lines 10–15 set up the grid and *ad hoc* values for τ_i. Line 45 sets up the right hand side of (8.19) at the interior points and lines 46 and 47 add in the penalty terms at the end points. The rest of the code should be a variant of earlier code snippets. The picture generated by this snippet is shown in Figure 8.2. The maximum absolute error computed by the snippet for different choices of N is shown in Table 8.2. Notice that doubling the number of grid points squares the error, but in the transitions from $N = 64$ to $N = 256$ both the numerical accuracy errors discussed earlier in Section 8.4 and the errors generated by the `odeint` evolution function are starting to intrude, spoiling the picture. The evolution errors can be mitigated by changing the accuracy parameter, but the others are inherent in the method.

This completes the survey of spectral methods coupled to the method of lines for solving parabolic problems, or hyperbolic ones where the solution is known a-priori to be smooth. Non-linearities and complicated boundary conditions have been covered, and the principal omission is a discussion of systems of equations. However, the philosophy behind the method of lines approach is to reduce the time integration of an evolutionary partial differential to solving a set of ordinary differential equations. Indeed both code snippets used the *scipy* `odeint` function. When we treated ordinary differential equations in Chapter 7, we showed there how to integrate systems of equations, and all that is required to deal with omission here is to merge those techniques into the snippets given here.

9 Case study: multigrid

In this final chapter, we present an extended example or "case study" of a topic which is relevant to almost all of the theoretical sciences, called multigrid. For many, multigrid is a closed and forbidding book, and so we first look at the type of problems it can be used to solve, and then outline how it works, finally describing broadly how it can be implemented very easily in Python. The rest of the chapter fleshes out the details.

In very many problems, we associate data with points on a spatial grid.[1] For simplicity, we assume that the grid is uniform. In a realistic case, we might want a resolution of say 100 points per dimension, and for a three-dimensional grid we would have 10^6 grid points. Even if we store only one piece of data per grid point, this is a lot of data which we can pack into a vector (one-dimensional array) $\mathbf{u}$ of dimension $N = O(10^6)$. These data are not free but will be restricted either by algebraic or differential equations. Using finite difference (or finite element) approximations, we can ensure that we are dealing with algebraic equations. Even if the underlying equations are non-linear, we have to linearize them (using, e.g., a Newton–Raphson procedure, see Section 9.3) for there is no hope of solving such a large set of non-linear equations. Thus we are faced with the solution of a very large linear system of equations of the form

$$A\mathbf{u} = \mathbf{f}, \tag{9.1}$$

where $\mathbf{f}$ is a N-dimensional vector and A is a $N \times N$ matrix.

Direct solution of the system (9.1), e.g., by matrix inversion, is usually a non-starter. Unless matrix A has a very special form, its inversion involve $O(N^3)$ operations! Instead, we have to use iterative methods (discussed in Section 9.2.1). These aim, by iterating on $\mathbf{u}$ to reduce the *residual* $\mathbf{r} = \mathbf{f} - A\mathbf{u}$ to zero. However, these methods, used in isolation, all suffer from a fatal flaw. To explain this, we need to consider a Fourier representation of the residual $\mathbf{r}$, which will consist of $O(N)$ modes. (This is discussed in more detail in Section 9.1.2.) The iterative methods will reduce very rapidly the magnitudes of about one half of the Fourier modes (the higher-frequency ones) of the residual to near zero, but will have negligible effect on the other half, the lower frequency ones. After that, the iteration stalls.

At this point, the first key idea behind multigrid emerges. Suppose we consider an additional grid of linear dimension $N/2$, i.e., double the spacing. Suppose too that we carefully transcribe the stalled solution $\mathbf{u}$, the source $\mathbf{f}$ and the operator A from the original *fine* grid to the new *coarse* grid. This makes sense because the fine grid residual

[1] In the finite element approach, the vocabulary is different, but the ideas are the same.

was made up predominantly of low-frequency modes, which will all be accurately represented on the coarse grid. However, half of these will be high frequency, as seen in the coarse grid. By iteration, we can reduce them nearly to zero. If we then carefully transfer the solution data back to the fine grid, we will have removed a further quarter of the Fourier modes of the residual, making three quarters in total.

The second key idea is to realize that we are not restricted to the two-grid model. We could build a third, even-coarser, grid and repeat the process so as to eliminate seven-eighths of the residual Fourier modes. Indeed, we could carry on adding coarser grids until we reach the trivial one. Note too that the additional grids take up in total only a fraction of the original grid size, and this penalty is more than justified by the speed enhancement.

It is important to note that the operations on and information transfers between grids are, apart from the sizes, the same for all grids. This suggests strongly the use of a Python class structure. Further once the two grid model has been constructed, the rest of the procedure can be defined very simply using recursion. In the rest of this chapter, we fill out some of the details, and set up a suite of Python functions to implement them, ending with a non-trivial non-linear example.

Although we have tried to make this chapter reasonably self-contained, interested readers will want to explore further. Arguably the best place for beginners to start is Briggs et al. (2000), although for the first half of their book there are a number of web-based competitors. However, in the second half Briggs et al. (2000) sketch in a clear and concise way a number of applications. Trottenberg et al. (2001) covers much the same material in a more discursive manner which some may find more helpful. The textbook Wesseling (1992) is often overlooked, but its algorithmic approach, especially in chapter 8, is not matched elsewhere. The monograph by the father of the subject Brandt and Livne (2011) contains a wealth of useful information.

9.1 The one-dimensional case

We need a concrete example, which will act as a paradigm for a few fundamental ideas.

9.1.1 Linear elliptic equations

Here is a very simple example of an elliptic differential equation in one-dimension on the interval [0, 1].

$$-u''(x) = f(x), \qquad u(0) = u_l, \quad u(1) = u_r, \tag{9.2}$$

where u_l and u_r are given constants. Note that by adding a linear function to $u(x)$, we can set $u_l = u_r = 0$, and here it is convenient to do so. We impose a uniform grid on [0, 1] with n intervals, i.e., $n+1$ points labelled $x_i = i/n$, for $0 \leqslant i \leqslant n$, and set $u_i = u(x_i)$, and $f_i = f(x_i)$. We then discretize (9.2) as

$$-\frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} = f_i, \qquad 0 < i < n, \tag{9.3}$$

where $\Delta x = 1/n$, $u_0 = 0$ and $u_n = 0$. We can rewrite (9.3) as

$$A\mathbf{u} = \mathbf{f}, \tag{9.4}$$

where $\mathbf{u} = \{u_1, \dots, u_{n-1}\}$, $\mathbf{f} = \{f_1, \dots, f_{n-1}\}$, $u_0 = u_n = 0$ and A is a $(n-1) \times (n-1)$ matrix. Many problems that are more complicated than (9.2) can be reduced to the form (9.4) which is the actual problem that we treat.

9.1.2 Smooth and rough modes

In the example above, how large should n be? Based on our experience of spectral methods, it is instructive to consider the use of discrete Fourier transforms. We may extend the domain of definition of u from $[0, 1]$ to $[-1, 1]$ by requiring the continuation to be an odd function of x. Finally, we can extend the domain to the real line by requiring $u(x)$ to be periodic with period 2. We define the discrete Fourier modes by

$$s_k(x) = \sin k\pi x, \quad k = 1, 2, \dots, n-1. \tag{9.5}$$

Their values at the grid points are

$$s_{k,j} = s_k(x_j) = \sin(kj\pi/n). \tag{9.6}$$

There are precisely $n-1$ linearly independent Fourier modes, which is the same as the number of unknowns, and indeed the Fourier modes form a basis for the vector space of grid functions. Those modes with $1 \leqslant k < \frac{1}{2}n$ are said to be *smooth modes* while those with $\frac{1}{2}n \leqslant k < n$ are said to be *rough modes*. Thus we might choose to expand $u(x)$ and $f(x)$ as, e.g.,

$$u(x) = \sum_{k=1}^{n-1} u^k s_k(x). \tag{9.7}$$

Then $u(x)$ is said to be *smooth* if it is an acceptable approximation to ignore the contribution to the sum from rough modes. Requiring this $u(x)$ to be smooth poses a lower limit on the value of n if reasonable accuracy and definition are required.

9.2 The tools of multigrid

9.2.1 Relaxation methods

In the system (9.4), the matrix A, although usually sparse, will have very large dimensions, especially if the underlying spatial grid has dimension 2 or larger. Direct solution methods, e.g., Gaussian elimination tend to be uncompetitive, and we usually seek an approximate solution by an iterative method, a process of "relaxation". Suppose we rewrite (9.3) as

$$u_i = \tfrac{1}{2}(u_{i-1} + u_{i+1} + \Delta x^2 f_i) \quad 1 \leqslant i \leqslant n-1. \tag{9.8}$$

Jacobi iteration is defined as follows. We start with some approximate solution $\widetilde{u}^{(0)}$ with $\widetilde{u}_0^{(0)} = \widetilde{u}_n^{(0)} = 0$. Using this on the right-hand side of (9.8), we can compute the components of a new approximation $\widetilde{u}^{(1)}$. Suppose we write this process as $\widetilde{u}^{(1)} = J(\widetilde{u}^{(0)})$. Under reasonable conditions, we can show that the kth iterate $\widetilde{u}^{(k)} = J^k(\widetilde{u}^{(0)})$ converges to the exact solution u as $k \to \infty$. In practice, we make a small modification. Let ω be a parameter with $0 < \omega \leqslant 1$, and set

$$\widetilde{u}^{(k+1)} = W(\widetilde{u}^{(k)}) = (1-\omega)\widetilde{u}^{(k)} + \omega J(\widetilde{u}^{(k)}). \tag{9.9}$$

This is *weighted Jacobi iteration*. For reasons explained below, $\omega = \frac{2}{3}$ is the usual choice.

Suppose we also write the exact solution u as

$$u = \widetilde{u}^{(k)} + e^{(k)},$$

where $e^{(k)}$ is the *error*. It is straightforward to show, using the linearity, that $e^{(k)}$ also satisfies (9.4) but with zero source term f.

We now illustrate this by a concrete example. We set $n = 16$ and choose as initial iterate for the error mode

$$e^{(0)} = s_1 + \tfrac{1}{3} s_{13},$$

where the s_k are the discrete Fourier modes defined in (9.5) and (9.6). This is the superposition of the fundamental mode, and one-third of a rapidly oscillating mode. Figure 9.1 shows this, and the results of the first eight Jacobi iterations. It was produced using

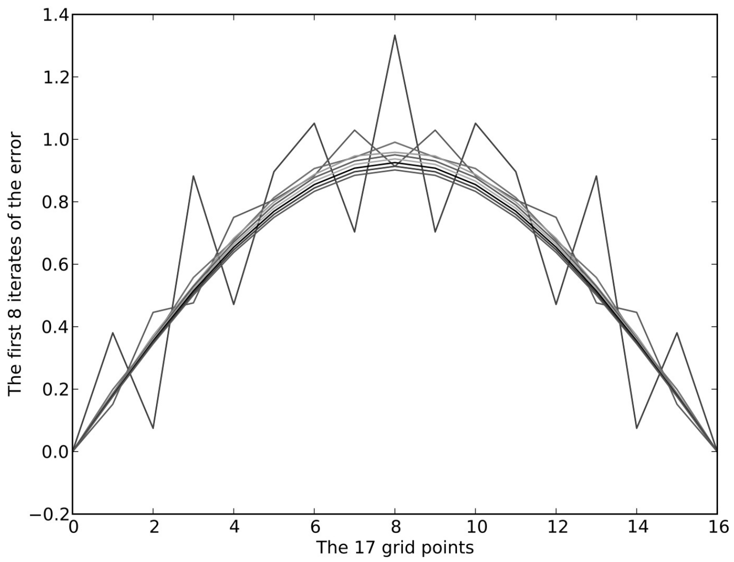

Figure 9.1 Jacobi iteration on a grid with 16 intervals. The initial iterate of the error is a mixture of the first and thirteenth harmonics, the extremely jagged curve. The results of 8 weighted Jacobi iterations are shown. Iteration damps the rough (thirteenth) mode very effectively but is extremely slow at reducing the smooth fundamental mode in the error.

the following code snippet.

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

N=16
omega=2.0/3.0
max_its=8

x=np.linspace(0,1,N+1)
s1=np.sin(np.pi*x)
s13=np.sin(13*np.pi*x)
e_old=s1+s13/3.0
e=np.zeros_like(e_old)

plt.ion()
plt.plot(e_old)
for it in range(max_its):
    e[1:-1]=(1.0-omega)*e_old[1:-1]+\
              0.5*omega*(e_old[0:-2]+e_old[2: ])
    plt.plot(e)
    e_old=e.copy()
plt.xlabel('The 17 grid points')
plt.ylabel('The first 8 iterates of the error')
```

Here our initial iterate was the superposition of the fundamental mode and one-third of a rapidly oscillating mode, the very spiky curve in Figure 9.1. The remaining curves show the results of the first eight Jacobi iterations. After three iterations, the amplitude of the spikes has all but vanished, but the reduction of the now smooth error $e^{(m)}$ to the exact value zero is extremely slow. You might like to alter the code snippet to verify that the same result holds for initial data made up of other mixtures of smooth and rough modes. The rough modes are damped rapidly, the smooth ones hardly at all. This is the heuristic reason for the name *relaxation*.

It is easy to see the theoretical foundation for these results. Note first that, using a standard trigonometrical identity, s_k defined by (9.6) is an eigenvector of the operator J with eigenvalue $\cos(k\pi/n)$. It follows that s_k is an eigenvector of operator W defined by (9.9), with eigenvalue

$$\lambda_k = 1 - \omega + \omega\cos(k\pi/n) = 1 - 2\omega\sin^2(\tfrac{1}{2}k\pi/n). \tag{9.10}$$

The damping of the kth Fourier mode is determined by $|\lambda_k|$. If $k \ll n$, we see that $\lambda_k \approx 1 - O(1/n^2)$. Thus damping of smooth modes is negligible, especially for large n. Thus although relaxation is convergent onto the exact solution, the ultraslow convergence of smooth modes renders it useless. Note however that if we consider the rough modes, $n/2 \leqslant k < n$, and we choose $\omega = 2/3$, then $|\lambda_k| < 1/3$. The weighted Jacobi iteration then becomes a very efficient damper of rough modes. In multigrid terms, it is a *smoother*.

There are of course many other classical relaxation methods, the most common being *Gauss–Seidel*. The Gauss–Seidel method utilizes components of $u^{(k+1)}$ on the right-hand side of (9.8) as soon as they have been computed. Thus it is really a family of methods depending on the order in which the components are computed. Here for example is a single *red–black Gauss–Seidel* iteration applied to the vector $\widetilde{\mathbf{u}}^{(k)}$

$$\begin{aligned} \widetilde{u}_i^{(k+1)} &= \tfrac{1}{2}(\widetilde{u}_{i-1}^{(k)} + \widetilde{u}_{i+1}^{(k)} + \Delta x^2 f_i) \qquad & i &= 2, 4, \ldots, n-2, \\ \widetilde{u}_i^{(k+1)} &= \tfrac{1}{2}(\widetilde{u}_{i-1}^{(k+1)} + \widetilde{u}_{i+1}^{(k+1)} + \Delta x^2 f_i) \qquad & i &= 1, 3, \ldots, n-1. \end{aligned} \tag{9.11}$$

Here points with i even are "red", while those with i odd are called "black". This extends to two dimensions, where points with i and j both even or both odd are "red", while the remaining points are "black". The similarity to a chess board gives rise to the name. The behaviour of Gauss–Seidel methods is similar to that of weighted Jacobi. They usually damp the rough modes a little faster, but suffer from the same defect when treating the smooth ones. Thus none of them taken in isolation can be used to solve the original problem (9.4).

Fortunately, multigrid offers a very elegant solution. The central tenet of multigrid is not to consider the problem on a single fixed grid, $n = 16$ above, but instead to look at it on a succession of grids, e.g., $n = 16, 8, 4, 2$. Scaling by factors other than 2 is possible but this is the usual choice. Henceforth, we shall assume that the finest grid size n is a power of 2.

9.2.2 Residual and error

We recall that our aim is to solve the system (9.4)

$$A\mathbf{u} = \mathbf{f}$$

on a "finest grid" of size n with u_0 and u_n specified. Suppose we have some approximation $\widetilde{\mathbf{u}}$ to $\mathbf{u}$. We define the *error* as $\mathbf{e} = \mathbf{u} - \widetilde{\mathbf{u}}$, and the *residual* as $\mathbf{r} = \mathbf{f} - A\widetilde{\mathbf{u}}$. Then it is easy to verify that the *residual equation*

$$A\mathbf{e} = \mathbf{r}, \quad e_0 = e_n = 0, \tag{9.12}$$

holds. It is apparently trite but important to recognize that systems (9.4) and (9.12) are formally identical.

We recall also that our finest grid is a partition of $0 \leqslant x \leqslant 1$ by n intervals. Setting $h = \Delta x = 1/n$, we have set $x_i = ih$, $u(x_i) = u_i$ etc. for $0 \leqslant i \leqslant n$. Also n is sufficiently large that $\mathbf{u}$ and $\mathbf{f}$ are smooth on the finest grid, and n is a power of 2. For the moment, we call the finest grid the *fine grid*. We consider also a *coarse grid* with spacing $H = 2h$, i.e., there are $n/2$ intervals, which is intended to be a replica of the fine grid. It is convenient to distinguish the fine grid quantities by, e.g., $\mathbf{u}^h$ from the corresponding coarse grid ones, e.g., $\mathbf{u}^H$. How are they related? Well for the linear problems under consideration we can require A^H to be obtained from A^h under the transformation $n \to n/2$, $h \to H = 2h$. Note that $\mathbf{u}^h$ is a $n-1$-dimensional vector, whereas $\mathbf{u}^H$ has dimension $n/2 - 1$.

9.2.3 Prolongation and restriction

Suppose we have a vector $\mathbf{v}^H$ defined on the coarse grid. The process of interpolating it onto the fine grid, called *prolongation* which we denote by ${I_H}^h$, can be done in several ways. The obvious way to handle those fine grid points, which correspond to the coarse grid ones, is a straight copy of the components. The simplest way to handle the others is linear interpolation; thus

$$v^h_{2j} = v^H_j \quad 0 \leqslant j \leqslant \tfrac{1}{2}n, \qquad v^h_{2j+1} = \tfrac{1}{2}(v^H_j + v^H_{j+1}) \quad 0 \leqslant j \leqslant \tfrac{1}{2}n - 1, \tag{9.13}$$

which we denote schematically as $\mathbf{v}^h = {I_H}^h \mathbf{v}^H$. Note that this may leave discontinuous slopes for the v^h_{2j} at internal points, which could be described as the creation of rough modes. We could "improve" on this by using a more sophisticated interpolation process. But in practice, we can remove these unwanted rough modes by applying a few smoothing steps to the fine grid after prolongation.

Next we consider the converse operation, i.e., mapping a fine grid vector $\mathbf{v}^h$ to a coarse grid one $\mathbf{v}^H$, which is called *restriction*, denoted by $\mathbf{v}^H = {I_h}^H \mathbf{v}^h$. The obvious choice is a straight copy

$$v^H_j = v^h_{2j} \qquad 0 \leqslant j \leqslant \tfrac{1}{2}n, \tag{9.14}$$

but the *full weighting process*

$$v^H_0 = v^h_0, \quad v^H_{n/2} = v^h_n, \qquad v^H_j = \tfrac{1}{4}(v^h_{2j-1} + 2v^h_{2j} + v^h_{2j+1}) \quad 1 \leqslant j \leqslant \tfrac{1}{2}n - 1, \tag{9.15}$$

is more commonly used.

Note that there is a problem with all restriction processes. The "source" space is that of vectors of dimension $n - 1$, and the "target" space is that of vectors of dimension $n/2 - 1$. Thus there must be a "kernel" vector space K of dimension $n/2$ such that the restriction of vectors in K is the zero vector. To see this explicitly, consider a Fourier mode $\mathbf{s}^h_k$ on the fine grid with

$$s^h_{k,j} = s^h_k(x_j) = \sin(kj\pi/n),$$

from (9.6). Assuming that j is even and using (9.14), we find

$$s^H_{k,j/2} = s^h_{k,j} = \sin(kj\pi/n).$$

However

$$s^H_{n-k,j/2} = \sin((n-k)j\pi/n) = \sin(j\pi - kj\pi/n) = -\sin(kj\pi/n)$$

since j is even. Thus the images of s^h_k and s^h_{n-k} under restriction are equal and opposite. (The same holds for full weighting.) This phenomenon is well known in Fourier analysis where it is called *aliasing*. Thus there is a serious loss of information in the restriction process. Note however that if s^h_k is smooth, i.e., $k < n/2$, then the aliased mode s^h_{n-k} is coarse. If we applied restriction only to smooth vectors, then the information loss would be minimal. Therefore, a good working rule is to smooth before restricting. Thus we have a "golden rule": smooth after prolongation but before restriction.

9.3 Multigrid schemes

Before we introduce specific schemes, we shall widen the discussion to consider a more general problem

$$L(\mathbf{u}(\mathbf{x})) = \mathbf{f}(\mathbf{x}), \tag{9.16}$$

on some domain $\Omega \subseteq \mathbf{R}^n$. Here L is a, possibly non-linear, elliptic operator, and $\mathbf{u}(\mathbf{x})$ and $\mathbf{f}(\mathbf{x})$ are smooth vector-valued functions defined on Ω. In order to keep the discussion simple, we shall assume homogeneous *Dirichlet boundary conditions*

$$\mathbf{u}(\mathbf{x}) = \mathbf{0}, \quad \mathbf{x} \in \partial\Omega \tag{9.17}$$

on the boundary $\partial\Omega$ of the domain Ω. More general boundary conditions are treated in the already cited literature.

Suppose $\widetilde{\mathbf{u}}(\mathbf{x})$ is some approximation to the exact solution $\mathbf{u}(\mathbf{x})$ of (9.16). As before, we define the *error* $\mathbf{e}(\mathbf{x})$ and *residual* $\mathbf{r}(\mathbf{x})$ by

$$\mathbf{e}(\mathbf{x}) = \mathbf{u}(\mathbf{x}) - \widetilde{\mathbf{u}}(\mathbf{x}), \quad \mathbf{r}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) - L(\widetilde{\mathbf{u}}(\mathbf{x})). \tag{9.18}$$

We now define the *residual equation* to be

$$L(\widetilde{\mathbf{u}} + \mathbf{e}) - L(\widetilde{\mathbf{u}}) = \mathbf{r}, \tag{9.19}$$

which follows from (9.16) and (9.18). Note that if the operator L is linear, then the left-hand side of (9.19) simplifies to $L(\mathbf{e})$, and so there is no inconsistency with the earlier definition (9.12).

A further pair of remarks is appropriate at this stage. We assume that we have constructed a fine grid Ω^h with spacing h which covers Ω. Using the previous notation, we aim to solve

$$L^h(\mathbf{u}^h) = \mathbf{f}^h \text{ on } \Omega^h, \quad \mathbf{u}^h = 0 \text{ on } \partial\Omega^h, \tag{9.20}$$

rather than the continuous problem (9.16) and (9.17). The important point to note is that it is not necessary to solve the grid equation (9.20) exactly. For even if we found the exact solution, $\mathbf{u}^{h,exact}$, this would not furnish an exact solution $\mathbf{u}^{exact}$ for the continuous problem. Instead, we might expect to obtain

$$||\mathbf{u}^{exact} - \mathbf{u}^{h,exact}|| \leqslant \epsilon^h, \tag{9.21}$$

for an appropriate norm and some bound ϵ^h which tends to zero as $h \to 0$. Thus it suffices to obtain an approximate grid solution $\widetilde{\mathbf{u}}^h$ with the property

$$||\mathbf{u}^{h,exact} - \widetilde{\mathbf{u}}^h|| = O(\epsilon^h). \tag{9.22}$$

The second remark concerns what we mean by “smoothing” in this more general non-linear context. Suppose we label the grid points in the interior of Ω^h by $i = 1, 2, \ldots, n-1$ in some particular order. Then (9.20) becomes

$$L_i^h(u_1^h, u_2^h, \ldots, u_{i-1}^h, u_i^h, u_{i+1}^h, \ldots, u_{n-1}^h) = f_i^h, \quad 1 \leqslant i \leqslant n-1.$$

The Gauss–Seidel (GS) iteration process solves these equations in sequence using new values as soon as they are computed. Thus

$$L_i^h(u_1^{h,new}, u_2^{h,new}, \dots, u_{i-1}^{h,new}, u_i^{h,new}, u_{i+1}^{h,old}, \dots, u_{n-1}^{h,old}) = f_i^h,$$

generates a single scalar equation for the unknown $u_i^{h,new}$. If this is non-linear, then we may use one or more Newton–Raphson (NR) iterations to solve it approximately, e.g.,

$$u_i^{h,new} = u_i^{h,old} - \left[\frac{L_i(\mathbf{u}) - f_i}{\partial L_i(\mathbf{u})/\partial u_i}\right]_{(u_1^{h,new}, u_2^{h,new}, \dots, u_{i-1}^{h,new}, u_i^{h,new}, u_{i+1}^{h,old}, \dots, u_{n-1}^{h,old})}. \tag{9.23}$$

The NR iterations should converge quadratically if the iterate is within the domain of attraction, and usually one iteration suffices.

In practice, this GS–NR process acts as an effective smoother, just as GS does in the linear case.

9.3.1 The two-grid algorithm

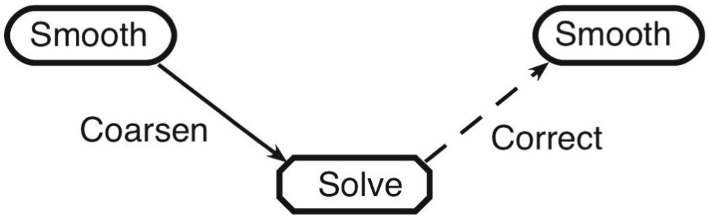

Figure 9.2 The two-grid scheme. The upper level corresponds to a grid with spacing h while the lower one refers to a grid with spacing H where $H > h$. Time is flowing from left to right. The scheme produces a (hopefully) more accurate solution on the finer grid. It may be iterated many times.

Now that we have a smoother, we can introduce an embryonic multigrid algorithm. Suppose that besides the grid Ω^h introduced above, we have a second grid Ω^H where, usually, $H = 2h$. We then apply in order the following steps:

Initial approximation We start with some arbitrary choice $\widetilde{\mathbf{u}}^h$ for an approximation to $\mathbf{u}^h$.

Smooth step We apply a small number ν_1 (typically 1, 2 or 3) of relaxation iteration steps to $\widetilde{\mathbf{u}}^h$, effectively removing the coarse components. Next we compute the residual

$$\mathbf{r}^h = \mathbf{f}^h - L^h(\widetilde{\mathbf{u}}^h). \tag{9.24}$$

Coarsen step Since $\widetilde{\mathbf{u}}^h$ and $\mathbf{r}^h$ are smooth, we can restrict them onto the coarse grid via

$$\widetilde{\mathbf{u}}^H = I_h{}^H \widetilde{\mathbf{u}}^h, \qquad \mathbf{f}^H = I_h{}^H \mathbf{r}^h, \tag{9.25}$$

without losing significant information.

Solution step We now solve the residual equation

$$L^H(\widetilde{\mathbf{u}}^H + \mathbf{e}^H) - L^H(\widetilde{\mathbf{u}}^H) = \mathbf{f}^H \tag{9.26}$$

for $\mathbf{e}^H$. How this is to be done will be explained later. Notice that, setting $\mathbf{u}^H = \widetilde{\mathbf{u}}^H + \mathbf{e}^H$, the residual equation can be written as

$$L^H(\mathbf{u}^H) = I_h{}^H\mathbf{f}^h + \tau^H, \tag{9.27}$$

where the *"tau-correction"* is

$$\tau^H = L^H(I_h{}^H\widetilde{\mathbf{u}}^h) - I_h{}^H(L^h(\widetilde{\mathbf{u}}^h)), \tag{9.28}$$

and the linearity of I_h^H has been used. The tau-correction is a useful measure of the errors inherent in the grid transfer process.

Correct step Prolong $\mathbf{e}^H$ back to the fine grid. Since $\widetilde{\mathbf{u}}^H + \mathbf{e}^H$ is presumably smooth on the coarse grid, this is an unambiguous operation. Then (hopefully) improve $\widetilde{\mathbf{u}}^h$ by the step $\widetilde{\mathbf{u}}^h \to \widetilde{\mathbf{u}}^h + \mathbf{e}^h$.

Smooth step Finally, we apply a small number ν_2 of relaxation iteration steps to $\widetilde{\mathbf{u}}^h$.

Figure 9.2 illustrates this process. Each horizontal level refers to a particular grid spacing, and lower level(s) refer to coarser grid(s). As we progress from left to right at the top level, we obtain a (hopefully) better approximation to the solution of the initial problem, although we may need to iterate the scheme several times. There are no convincing theoretical arguments for establishing values for the parameters ν_1 and ν_2. Their choice has to be made empirically.

This two-grid algorithm is often referred to as the *Full Approximation Scheme (FAS)*. It is designed to handle non-linear problems and so has to implement the non-linear version of the residual equation (9.19) which requires the transfer of an approximate solution from the fine to the coarse grid. In the linear case, it is only necessary to transfer the error, see, e.g., (9.12).

As presented the two-grid algorithm suffers from two very serious defects. The first appears in the initial approximation step. How is the initial choice of $\widetilde{\mathbf{u}}^h$ to be made? In the linear case, there is no issue, for we might as well choose $\widetilde{\mathbf{u}}^h = \mathbf{0}$. But in the non-linear case, this choice may not lie in the domain of attraction for the Newton–Raphson iteration, which would upset the smoothing step. In Subsection 9.3.3, we remove this obstacle. The second defect is how are we to generate the solution of the residual equation in the solution step? We deal with this in the next section.

9.3.2 The V-cycle scheme

One flaw with the two-grid scheme is the need to solve the residual equation (9.26) on the coarser grid. Because there are rather fewer operations, involved this is less intensive than the original problem, but the cost may still be unacceptable. In this case, a possible solution arises if we augment the coarse grid solution step by another two-grid scheme using an even coarser grid. As can be seen from Figure 9.3, we have an embryonic *V-cycle*. Of course, there is no need to stop with three levels. We may repeat the process

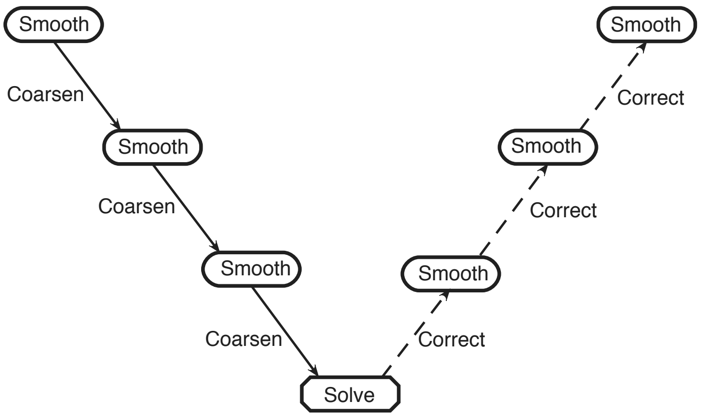

Figure 9.3 The V-cycle scheme. The upper level corresponds to a grid with spacing h while the lower ones refer to progressively coarser grids. Time is flowing from left to right. The scheme produces a (hopefully) more accurate solution on the finer grid. It may be iterated many times, hence the name "cycle". Notice how the "golden rule", smooth before coarsening and after refining, is applied automatically.

adding ever coarser grids until we reach one with precisely one interior point, where we construct an "exact" solution on this grid. Notice that each *V-cycle* comes with the two parameters ν_1 and ν_2 inherited from the two-grid algorithm. It should be apparent from Figure 9.3 that we could iterate on *V-cycles*, and in practice this is usually done ν_0 times.

9.3.3 The full multigrid scheme (FMG)

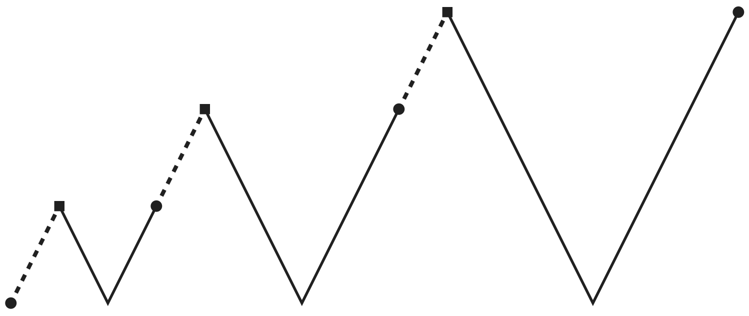

Figure 9.4 The FMG scheme. The circular dots correspond to the final solution and the square dots indicate the initial approximate solution for one or more V-cycles at the given level, while the dotted lines correspond (as before) to an interpolation step. As time progresses from left to right, the transition from the initial to the final solution is via a sequence of V-cycles.

We turn now to the initialization problem for the two-grid algorithm. For linear problems, we can always start from the trivial solution, but this is not possible, usually, for non-linear ones. The V-cycle presented above suggests an obvious solution. Suppose we jump to the coarsest grid, solve the problem there and prolong the solution back to the finer grids to act as a first iterate. In this simple form, the idea doesn't work. Interpolation leads to two types of errors. High-frequency errors may be invisible on the coarse grid, but should be reduced by smoothing. Aliasing errors, i.e., smooth errors introduced by high-frequency effects because on the coarse grid it is mistaken for smooth data, can also occur. However the *Full Multigrid Scheme (FMG)* gets round this problem in a very ingenious way. First we solve the problem on the coarsest grid. Then interpolate the solution to the next coarsest grid. Now perform ν_0 V-cycles on this grid (which turns out to be a two-grid scheme). Next interpolate the solution to the third coarsest grid, perform V-cycles and so on. The scheme is depicted in Figure 9.4. Note that the interpolation steps (the dotted lines in the figure) are not part of the V-cycles. While it may be convenient to use the standard prolongation operator for these steps, there are strong arguments, see, e.g., Brandt and Livne (2011), for using a more accurate scheme.

9.4 A simple Python multigrid implementation

The aim of this chapter is to produce a simple Python implementation of Multigrid using the components we have just outlined. Clearly, V-cycles and FMG are implemented most elegantly using recursion. Most early implementations used Fortran which then did not include recursion, and so lengthy but reasonably efficient kludges were developed to circumvent this. Also it should be clear that the levels or grids have a clear structure, which can be used to clarify the code. Until recently, Fortran lacked the means to implement such structures. Thus most existing codes are very difficult to understand. We could of course replicate them in Python using the `f2py` tool from Chapter 8, but this is not an approach that we would wish to encourage. We shall instead produce a Python implementation which is clear, concise and simple.

The most efficient way to undertake programming a complex task like Multigrid is to split it into groups of subtasks, and we can identify three obvious groups. The first is those tasks which depend only on the number of dimensions and not on the actual problem. Prolongation and restriction fall naturally into this category. Tasks in the second group depend on the number of dimensions and the specific problem. The obvious examples are smoothing and the computation of residuals. The final group consists of tasks specific to multigrid. Our implementation of this final group will make assumptions neither about the number of dimensions nor the details of the specific problem being studied.

Experience shows that tasks in the first two groups are the computationally expensive ones. The code used here is a reasonably efficient *numpy* one. However, it would be very easy, using e.g., `f2py`, to implement them using a compiled language such as Fortran with a Python wrapper, because the algorithms used involve just simple loops. However, the situation is reversed for the third group. The clearest formal definition of multigrid is recursive in nature. Recursion is fully implemented in Python, but only partially so in more recent dialects of Fortran. The concept of grids with different levels of refinement (the vertical spacing in the figures above) is easily implemented using the Python *class* construct, which is much simpler to implement than its C++ counterpart.[2]

For the sake of brevity, we have eliminated all extraneous features and have reduced comments to the absolute minimum. The scripts presented here should be thought of as a framework for a more user-friendly suite of programmes.

9.4.1 Utility functions

The file `util.py` contains three functions covering L^2-norms (mean square), prolongation and restriction in two dimensions. It should be a routine matter to write down the equivalents in one, three or any other number of dimensions.

```python
# File: util.py: useful 2D utilities.

import numpy as np

def l2_norm(a,h):
    return h*np.sqrt(np.sum(a**2))

def prolong_lin(a):
    pshape=(2*np.shape(a)[0]-1,2*np.shape(a)[1]-1)
    p=np.empty(pshape,float)
    p[0: :2,0: :2]=a[0: ,0: ]
    p[1:-1:2,0: :2]=0.5*(a[0:-1,0: ]+a[1: ,0: ])
    p[0: :2,1:-1:2]=0.5*(a[0: ,0:-1]+a[0: ,1: ])
    p[1:-1:2,1:-1:2]=0.25*(a[0:-1,0:-1]+a[1: ,0:-1]+
        a[0:-1,1: ]+a[1: ,1: ])
    return p

def restrict_hw(a):
    rshape=(np.shape(a)[0]/2+1,np.shape(a)[1]/2+1)
    r=np.empty(rshape,float)
    r[1:-1,1:-1]=0.5*a[2:-1:2,2:-1:2]+ \
                0.125*(a[2:-1:2,1:-2:2]+a[2:-1:2,3: :2]+
                       a[1:-2:2,2:-1:2]+a[3: :2,2:-1:2])
    r[0,0: ]=a[0,0: :2]
    r[-1,0: ]=a[-1,0: :2]
    r[0: ,0]=a[0: :2,0]
    r[0: ,-1]=a[0: :2,-1]
    return r

```

[2] Much of the complexity in C++ classes comes from the security requirement; a normal class user should not be able to see the implementation details for the class algorithms. This feature is irrelevant for most scientific users and is not offered in Python.

```
30 #-----------------------------------------------
31 if __name__=='__main__':
32     a=np.linspace(1,81,81)
33     b=a.reshape(9,9)
34     c=restrict_hw(b)
35     d=prolong_lin(c)
36     print "original grid\n",b
37     print "with spacing 1 its norm is ",l2_norm(b,1)
38     print "\n restricted grid\n",c
39     print "\n prolonged restricted grid\n",d
```

The function in lines 5 and 6 returns an approximation to the L^2-norm of any array a with spacing h. Lines 8–16 implement prolongation using linear interpolation for a two-dimensional array a, using the obvious generalization of the one-dimensional version (9.13). Next lines 18–23 show how to implement restriction for the interior points by *half weighting*, which is the simplest generalization of full weighting defined in one dimension by (9.15) in a two-dimensional context. Finally, lines 24–27 implement a simple copy for the boundary points. The supplementary test suite, lines 31–39, offers some simple partial checks on these three functions.

9.4.2 Smoothing functions

The smoothing functions depend on the problem being considered. We treat here the test problem for non-linear multigrid, which is discussed in Press et al. (2007),

$$L(u) = u_{xx} + u_{yy} + u^2 = f(x, y) \tag{9.29}$$

on the domain $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$ with

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0. \tag{9.30}$$

We may write the discretized version in the form

$$F_{i,j} \equiv \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} + {u_{i,j}}^2 - f_{i,j} = 0. \tag{9.31}$$

We regard $F_{i,j} = 0$ as a non-linear equation to be solved for $u_{i,j}$. Noting that

$$\frac{\partial F_{i,j}}{\partial u_{i,j}} = -\frac{4}{h^2} + 2u_{i,j},$$

the Newton–Raphson iteration scheme is

$$(u_{i,j})_{new} = u_{i,j} - \frac{F_{i,j}}{-4/h^2 + 2u_{i,j}}. \tag{9.32}$$

Here is a module with functions to carry out a single iteration of Gauss–Seidel red–black smoothing, and to compute the residual. As usual docstrings and helpful comments have been omitted to ensure brevity.

```
# File: smooth.py: problem dependent 2D utilities.

import numpy as np

def get_lhs(u,h2):
    w=np.zeros_like(u)
    w[1:-1,1:-1]=(u[0:-2,1:-1]+u[2: ,1:-1]+
                  u[1:-1,0:-2]+u[1:-1,2: ]-
                  4*u[1:-1,1:-1])/h2+u[1:-1,1:-1]*u[1:-1,1:-1]
    return w

def gs_rb_step(v,f,h2):
    u=v.copy()
    res=np.empty_like(v)

    res[1:-1:2,1:-1:2]=(u[0:-2:2,1:-1:2]+u[2: :2,1:-1:2]+
                        u[1:-1:2,0:-2:2]+u[1:-1:2,2: :2]-
                        4*u[1:-1:2,1:-1:2])/h2 +\
                        u[1:-1:2,1:-1:2]**2-f[1:-1:2,1:-1:2]
    u[1:-1:2, 1:-1:2]-=res[1:-1:2,1:-1:2]/(
                                 -4.0/h2+2*u[1:-1:2,1:-1:2])

    res[2:-2:2,2:-2:2]=(u[1:-3:2,2:-2:2]+u[3:-1:2,2:-2:2]+
                        u[2:-2:2,1:-3:2]+u[2:-2:2,3:-1:2]-
                        4*u[2:-2:2,2:-2:2])/h2+\
                        u[2:-2:2,2:-2:2]**2-f[2:-2:2,2:-2:2]
    u[2:-2:2,2:-2:2]-=res[2:-2:2,2:-2:2]/(
                                -4.0/h2+2*u[2:-2:2,2:-2:2])

    res[2:-2:2,1:-1:2]=(u[1:-3:2,1:-1:2]+u[3:-1:2,1:-1:2]+
                        u[2:-2:2,0:-2:2]+u[2:-2:2,2: :2]-
                        4*u[2:-2:2,1:-1:2])/h2 +\
                        u[2:-2:2,1:-1:2]**2-f[2:-2:2,1:-1:2]
    u[2:-2:2,1:-1:2]-=res[2:-2:2,1:-1:2]/(
                                -4.0/h2+2*u[2:-2:2,1:-1:2])

    res[1:-1:2,2:-2:2]=(u[0:-2:2,2:-2:2]+u[2: :2,2:-2:2]+
                        u[1:-1:2,1:-3:2]+u[1:-1:2,3:-1:2]-
                        4*u[1:-1:2,2:-2:2])/h2+\
                        u[1:-1:2,2:-2:2]**2-f[1:-1:2,2:-2:2]
    u[1:-1:2,2:-2:2]-=res[1:-1:2,2:-2:2]/(
                                -4.0/h2+2*u[1:-1:2,2:-2:2])

    return u


def solve(rhs):
    h=0.5
    u=np.zeros_like(rhs)
    fac=2.0/h**2
    dis=np.sqrt(fac**2+rhs[1,1])
    u[1,1]=-rhs[1,1]/(fac+dis)
    return u
```

The first function `get_lhs` takes a current **u** and steplength squared h^2 and returns $L(\mathbf{u})$ for this level. The second function `gs_rb_step` requires in addition the right-hand side **f** and combines a Newton–Raphson iteration with a red–black Gauss–Seidel one to produce a smoothing step. Note that nowhere do we need to specify the sizes of the arrays: the information about which level is being used is contained in the `h2` parameter. The final function returns the exact solution of the discretized equation on the coarsest 3×3 grid.

These are the most lengthy and complicated functions we shall need. By abstracting them into a separate module, we can construct a test suite (not shown) in order to ensure that they deliver what is intended.

9.4.3 Multigrid functions

We now present a Python module which defines a class `Grid` which will carry out FAS V-cycles and FMG. The basic idea is for a `Grid` to represent all of the data and functions for each of the horizontal levels in Figures 9.3 and 9.4. The basic difference between levels is merely one of size, and so it is appropriate to encapsulate them in a class. The data include a pointer to a coarser `Grid`, i.e., the next level below the one under discussion. Because of the complexity of the ideas being used, we have included the docstrings and various **`print`** statements. Once the reader understands what is going on, the **`print`** statements may safely be deleted.

```
# File: grid.py: linked grid structures and associated algorithms

import numpy as np
from util import l2_norm, restrict_hw, prolong_lin
from smooth import gs_rb_step, get_lhs, solve

class Grid:
    """
        A Grid contains the structures and algorithms for a
        given level together with a pointer to a coarser grid.
    """
    def __init__(self,name,steplength,u,f,coarser=None):
        self.name=name
        self.co=coarser          # pointer to coarser grid
        self.h=steplength        # step length h
        self.h2=steplength**2    # h**2
        self.u=u                 # improved variable array
        self.f=f                 # right hand side array

    def __str__(self):
        """ Generate an information string about this level. """
        sme='Grid at %s with steplength = %0.4g\n' % (
                            self.name, self.h)
        if self.co:
            sco='Coarser grid with name %s\n' % self.co.name
        else:
            sco='No coarser grid\n'
        return sme+sco

    def smooth(self,nu):
        """
            Carry out Newton--Raphson/Gauss--Seidel red--black
            iteration u-->u, nu times.
        """
        print 'Relax in %s for %d times' % (self.name,nu)
        v=self.u.copy()
        for i in range(nu):
            v=gs_rb_step(v,self.f,self.h2)
        self.u=v

    def fas_v_cycle(self,nu1,nu2):
        """ Recursive implementation of (nu1, nu2) FAS V-Cycle."""
        print 'FAS-V-cycle called for grid at %s\n' % self.name
        # Initial smoothing
        self.smooth(nu1)
        if self.co:
            # There is a coarser grid
            self.co.u=restrict_hw(self.u)
            # Get residual
            res=self.f-get_lhs(self.u,self.h2)
            # get coarser f
            self.co.f=restrict_hw(res)+get_lhs(self.co.u,
                                               self.co.h2)
            oldc=self.co.u
            # Get new coarse solution
            newc=self.co.fas_v_cycle(nu1,nu2)
```

```
57              # Correct current u
58              self.u+=prolong_lin(newc-oldc)
59          self.smooth(nu2)
60          return self.u
61
62      def fmg_fas_v_cycle(self,nu0,nu1,nu2):
63          """ Recursive implementation of FMG-FAS-V-Cycle"""
64          print 'FMG-FAS-V-cycle called for grid at %s\n' % self.name
65          if not self.co:
66              # Coarsest grid
67              self.u=solve(self.f)
68          else:
69              # Restrict f
70              self.co.f=restrict_hw(self.f)
71              # Use recursion to get coarser u
72              self.co.u=self.co.fmg_fas_v_cycle(nu0,nu1,nu2)
73              # Prolong to current u
74              self.u=prolong_lin(self.co.u)
75          for it in range(nu0):
76              self.u=self.fas_v_cycle(nu1, nu2)
77          return self.u
```

Thanks to the use of classes, the code is remarkably simple and concise. Note that `None` in line 12 is a special Python variable which can take any type and always evaluates to zero or `False`. By default there is not a coarser grid, and so when we actually construct a *list* of levels we need to link manually each grid to the next coarser one. The function `__init__`, lines 12–18 constructs a Grid level. The function `__str__`. lines 20–28 constructs a string describing the current grid. Their use will be described soon. Lines 30–39 give the `smooth` function, which simply carries out `nu` iterations of the Newton–Raphson/Gauss–Seidel smoothing process on the current grid. The function `fas_v_cycle` on lines 41–60 is non-trivial. If we are at the coarsest grid level, it ignores lines 47–58 and simply carries out $\nu_1 + \nu_2$ smoothing steps and then returns. In the more general case, lines 45–54 carry out ν_1 "smooth" steps followed by a "coarsen" step of the two-grid scheme. Line 56 then carries out the "solve" step by recursion, and finally lines 58 and 59 carry out the remaining "correct" and finally ν_2 "smooth" steps. It is not the most efficient code. For useful hints on achieving efficiency, see the discussion of an equivalent code in Press et al. (2007). Finally, lines 62–77 implement a non-trivial function for carrying out full multigrid FAS cycles. The parameter ν_0 controls how many *V-cycles* are carried out. Notice how closely the Python code mimics the earlier theoretical description, which makes it easier to understand and to develop, a great saving of time and effort.

We shall assume that the code snippet above has been saved as `grid.py`.

Finally, we create the following snippet in a file `rungrid.py` to show how to use the functions. We have chosen to replicate the worked example from Press et al. (2007)

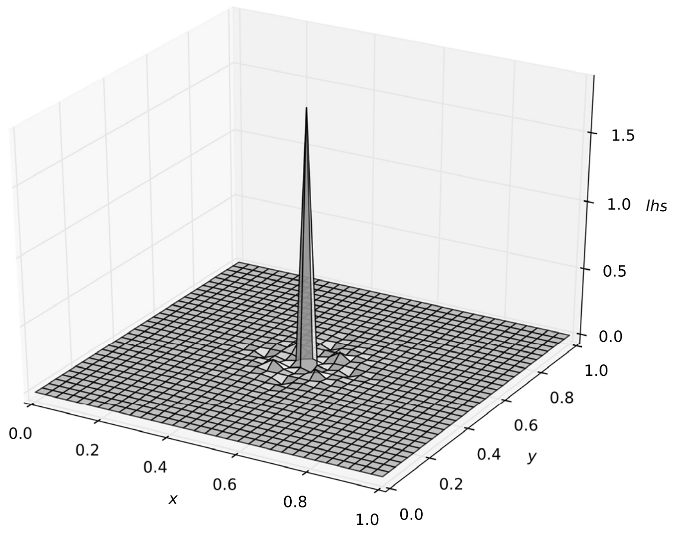

Figure 9.5 The left-hand side of equation (9.15) plotted on the 32×32 computational grid. The original source was zero everywhere apart from the central point where it took the value 2.

which studies (9.29) on a 32×32 grid with $f(x, y) = 0$ except at the mid-point where it takes the value 2.

```python
# File rungrid.py: runs multigrid programme.

import numpy as np
from grid import Grid
from smooth import get_lhs

n_grids=5
size_init=2**n_grids+1
size=size_init-1
h=1.0/size
foo=[]                                # container for grids

for k in range(n_grids):              # set up list of grids
    u=np.zeros((size+1,size+1),float)
    f=np.zeros_like(u)
    name='Level '+str(n_grids-1-k)
    temp=Grid(name,h,u,f)
    foo.append(temp)
    size/=2
    h*=2

```

```
22  for k in range(1,n_grids):          # set up coarser links
23     foo[k-1].co=foo[k]
24
25  # Check that the initial construction works
26  for k in range(n_grids):
27     print foo[k]
28
29  # Set up data for the NR problem
30  u_init=np.zeros((size_init,size_init))
31  f_init=np.zeros_like(u_init)
32  f_init[size_init/2,size_init/2]=2.0
33  foo[0].u=u_init                      # trial solution
34  foo[0].f=f_init
35
36  foo[0].fmg_fas_v_cycle(1,1,1)
37
38  # As a check get lhs of equation for final grid
39
40  lhs=get_lhs(foo[0].u,foo[0].h2)
41
42  import matplotlib.pyplot as plt
43  from mpl_toolkits.mplot3d import Axes3D
44
45  plt.ion()
46  fig=plt.figure()
47  ax=Axes3D(fig)
48
49  xx,yy=np.mgrid[0:1:1j*size_init,0:1:1j*size_init]
50  ax.plot_surface(xx,yy,lhs,rstride=1,cstride=1,alpha=0.4)
```

Lines 7–11 set up initial parameters, including an empty *list* `foo` which is a container for the `Grid` hierarchy. Each pass of the loop in lines 13–20 constructs a `Grid` (line 17) and adds it to the *list* (line18). Next the loop in lines 22 and 23 links `Grids` to the coarser neighbour. Lines 26–27 check that nothing is amiss by exercising the `__str__` function. The lines 30–34 set up some initial data for the problem actually treated by Press et al. (2007). Finally, line 36 constructs the solution.

In order to perform one (of many) checks on the solution, we reconstruct the left-hand side of the equation evaluated on the finest grid in line 40. The remainder of the snippet plots this as a surface, shown in Figure 9.5. The initial source was zero everywhere except at the central point where it took the value 2. The numerical results, which are identical to those obtained with the code of Press et al. (2007), give the peak value as 1.8660. Increasing the grid resolution to 64×64 or 128×128 does not change this value significantly. The L^2-norm of the error is always $O(h^2)$, but this disguises the effect of the discontinuity.

This example was chosen to run in a fraction of a second on most reasonably modern machines. If you hack the code to handle a more realistic (and complicated) problem for you, it might run rather more slowly. How do we “improve” the code? The first step is to verify carefully that the code does produce convergent results. Next we might think about improving the efficiency of the Python code. The first step is to *profile* the code. Assuming you are using *IPython*, then replace the command `run rungrid` with `run -p rungrid`. This appends to the output the results from the python *profiler* which show the time spent in each function. If there is a “bottleneck”, it will show up at the top of the list. Almost certainly, any bottleneck will be in the file `smooth.py`. It may be that the python code for such functions is obviously inefficient and can be improved. More rarely, vectorized numpy code is too slow. Then the simplest effective solution is to use `f2py`, as advocated in Chapter 8. This does not mean recoding the whole programme in Fortran, but only the bottleneck functions, which should be computationally intensive but with a simple overall structure, so that no great Fortran expertise is needed.

Appendix A Installing a Python environment

In order to use Python we need to install at least two distinct types of software. The first type is relatively straightforward, consisting of core Python itself, and associated packages, and is discussed in Section A.1 below. The second type addresses a much more complex issue, the interaction of human and machine, i.e., how to instruct the machine to do what we want it to do. An optimal resolution depends in part on what other computer software you wish to use. Some pointers are offered in Section A.2.

Most users of Matlab or Mathematica never see these issues. Invoking either application sets up an editor window or *notebook*, and an integrated editor accepts instructions from the keyboard. Then a key press invokes the interpreter, and delivers the desired output with an automatic return to the editor window. They obviously get top marks for immediate convenience and simplicity, but in the long term they may not win out on efficiency and versatility.

A.1 Installing Python packages

As has been outlined in Section 1.2, we shall need not only core Python, but also the add-on packages *IPython* (see Chapter 2), *numpy*, *scipy* (packages discussed in Chapter 4), *matplotlib* (see Chapter 5) and potentially *mayavi* (discussed in Chapter 6). Although data analysis is merely mentioned in Section 4.5, that section recommends the *pandas* package, and for many this will be a must-have.

The core Python package may already be available on your machine, and it is very tempting to load up the others from your favourite software repository, or from the main Python site,[1] where you can find both documentation and downloadable packages for the major platforms. In my experience, this is far too often the advice offered by "computer officers", who are oblivious of the consequences of this policy. The snag with this approach is that it is very difficult to ensure that all of these packages are mutually compatible. Even if they are at one instant, a subsequent "upgrade improvement" to say *numpy* may well break the link to the other add-on packages. Instead, I recommend strongly that you download a complete integrated system, and stick with it. You may then wish to upgrade the complete system as a whole every other year or so.

There are a number of integrated systems available, so which one should you choose?

[1] Its website is `http://www.python.org`.

For many people, the simplest solution is to use the *Enthought Python Distribution*.[2] New users should note that it is now called *Canopy*, and is available on all of the standard platforms. Every user can download and use free-of-charge *Canopy Express*. This is a "light" version of *Canopy* omitting many specialized packages. It does however contain all of the packages mentioned above except *mayavi*. However, if you are a student or staff member at a degree-granting institution, you can obtain a free-of-charge academic licence, renewable annually, for the full *Canopy* package, which of course includes *mayavi*. If you are ineligible for the academic licence, then you need to do a careful cost–benefit analysis because *Canopy* offers a wide range of goodies, intended for a business-oriented audience, which might justify the cost of a licence. Whichever approach you choose, the huge advantage here is a well-tested and popular integrated environment out-of-the-box. All of the code in this book has been generated and tested seamlessly using it.

There exist similar free-to-use packages, with less restricted availability, including *Python(x,y)*[3] for the Windows platform only, which includes all of the recommended packages, including *mayavi*, *Scipy-Superpack*[4] for Mac OS X and *Source Python Distribution*[5] for many platforms, and they may well suit your requirements better, but I have not tested any of these packages.

In order to use parts of *scipy* and the `f2py` utility, a Fortran compiler is needed. If you already have one, then the Python integrated system should recognize it. Otherwise, the ideal companion is *gfortran*, and free-to-use binaries for all platforms are available from its web site[6].

A.2 Communicating with Python

Invoking Python, e.g., typing `python` on a command line, places one directly in a rather basic interpreter. Instead, we strongly recommend using the enhanced interpreter *IPython*, as advocated in Chapter 2. Commands restricted to a few lines can be entered on the keyboard for immediate execution. However, to enter longer programmes and, more importantly to amend and save them for future use, an editor is required. Choosing a suitable editor requires care, and is discussed in Section A.2.1 below.

A.2.1 Editors for programming

For many users, the concept of a computer text editor usually means a word processor such as Microsoft *Word* or one of its many imitators. These do a more or less fine job of word-processing, covering many languages and characters. Each symbol input at the keyboard has to be converted to a number, and the current standard is "Unicode", which

[2] It is available at `http://www.enthought.com/products/epd`.
[3] Its website is `http://code.google.com/p/pythonxy`.
[4] This moves around but is currently at `http://fonnesbeck.github.io/ScipySuperpack`.
[5] It can be found at `http://code.google.com/p/spdproject/downloads/list`.
[6] Its address is `http://gcc.gnu.org/wiki/GFortran/`.

carries conversion codes for more than 110,000 characters. Of course, the conversion works also in the reverse direction for output to the screen or printer and so the casual user never sees this complexity.

Programming languages predate word processors and were developed when computer memory was scarce. The standard conversion is "ASCII", which allows for 93 printable characters—more than enough for most programming languages, including Python. Unfortunately, Unicode-based editors make a very poor job of producing ASCII and in practice we need to use an ASCII-based editor.

Before choosing a "programmer's editor", you should take stock of which programming languages you use or intend to use. The reason for this is that all of these languages have their idiosyncrasies, and it is highly desirable that the editor be aware of them.[7] Fortunately, there are excellent ASCII-based editors in the public domain. *Emacs* and *vim* (formerly *vi*) are ultra-versatile and are available on almost all platforms. Their versatility depends on built-in or add-on modules for the main programming languages, written by enthusiasts. However, the cost of that versatility is a slightly steeper learning curve. Perhaps this is the reason why many users of one or the other are fiercely (almost irrationally) loyal to their choice and deprecate strongly the other.

Of course, many scientists will be prepared to sacrifice some aspects of versatility in return for a much shallower learning curve, and there are some extremely competent open-source alternatives. In this context, many Python users on the Windows platform prefer to use *Notepad++*[8] which is a versatile editor with a much shallower learning curve, but it is native to the Windows platform. Intrinsic to the Mac OS X platform, the editor *TextWrangler*[9] is similarly the preferred choice. There are of course many other excellent public domain and proprietary programmer's editors which are aware of the common programming languages. Tables listing and comparing their features can be found on the internet.

A.2.2 The *IPython*-editor interaction

As can be seen in Chapter 2, non-trivial programme development involves frequent switching between the *IPython* interpreter and the text editor. There are three different ways of doing this. Which you choose depends on a trade-off between user convenience and the complexity of the initial setup process.

A.2.3 The two windows approach

This is the simplest option. You open both an *IPython* window and an editor window. Transfer of code from interpreter to editor is performed using your operating system's "copy and paste" facility. Transfer of saved edited code to interpreter is accomplished

[7] For example, in the creation of this book I used LaTeX, Python, Reduce, Fortran and C++. Fairly obviously, I prefer a single editor which "understands" all of these.

[8] Its website is `http://notepad-plus-plus.org`.

[9] Free-to-use from `http://www.barebones.com/products/textwrangler`.

using the latter's magic `%run` command—see Section 2.6. This is trivial to set up, but can become cluttered with large windows on small screens.

A.2.4 Calling the editor from within *IPython*

Rather more sophisticated is the use of the magic `%edit` command within the interpreter. Then *IPython* will fire up an editor in a new window. Once editing is complete, you save the file and quit the editor. *IPython* resumes control and by default runs the new code. Note that this approach is suited to fast lightweight editors only. The default is *vim* or *notepad* (Windows). If you are unhappy with this default choice, you should investigate the documentation to discover where to reset the default editor.

A.2.5 Calling *IPython* from within the editor

Even more sophisticated is to use only an editor window, and to run *IPython* from within it in a subwindow. At the time of writing, it only works with the first three text editors cited in Section A.2.1, and is slightly more difficult to set up.[10] Suppose you have developed and saved a script `foo.py` in a normal editor window. Next you open an *IPython* subwindow, e.g., CTL-C ! in *emacs*, and type `run foo` in it to execute the script. Now one can try out new ideas in this interpreter subwindow. Once they have been finalized they can be pasted back into the editor window. For those who have significant experience with their editor, this is the most versatile approach.

A.2.6 The *IPython* pager

As we shall see, Python contains a great deal of online information which can be accessed as and when required. The *IPython* interpreter has an *introspection* feature to access this. This is perhaps most useful when faced with a very specific enquiry which can be answered in one or two lines. However, when learning a new feature we usually start from profound ignorance, and there will be a procession of enquiries starting from very general ones and ending with the required specific one. Python can cope with this, but, inevitably, general questions may need up to a hundred lines or so to answer. *IPython* will call up a "pager" to handle this problem. The standard default is the *less* pager which will be present on all Unix-based platforms, and is available for Windows. If you do not already use this, you need to know just four commands:

- SPACE gives the next screen,
- `b` gives the previous screen,
- `h` gives the help screen, which includes less frequently used commands,
- `q` quits.

If you are unhappy with this choice of pager, then the default can be changed.

[10] Unfortunately, the official documentation is far from adequate. I got started by searching the web for `emacs ipython`. An easily overlooked point is that to use `python mode` you need to tell the editor where to find the Python executable. You should give instead the address of the *IPython* executable.

A.3 The Python Package Index

Occasionally you may wish to obtain and install a Python package which was not included in the distribution already installed. There is a central repository, the Python Package Index,[11] which contains more than 25,000 additional packages. Bear in mind that there are a number of excellent packages, e.g., *pydelay*, see Section 7.5.2, which are not in the package index. There are various ways of installing them, each "easier than the previous method", but the following procedure is simple and should work with any Python distribution. Suppose you have located a package from the index, say `foo.tar.gz` and downloaded it to any convenient directory. After unpacking it, you should move to the source directory `foo` which will contain several files including one called `setup.py`. Assuming you have, or have obtained, "super user" privileges, the following command line

```
python setup.py install
```

will compile and install the package where your Python interpreter can find it, typically via a command like

```
import foo
```

See also Section 4.9.2. Concrete examples are given in Sections 7.4.2 and 7.5.2.

[11] Currently it is at `http://pypi.python.org/pypi`.

Appendix B Fortran77 subroutines for pseudospectral methods

Appendix F of Fornberg (1995) includes a collection of related Fortran77 subroutines for constructing pseudospectral algorithms for solving non-periodic problems. In Section 8.8 we explain how we can use `f2py` to create a corresponding suite of what look like Python functions, but are in fact the compiled Fortran codes with Pythonic wrappers. We can carry out this sophisticated process with at most a minimal knowledge and understanding of Fortran. We list here the relevant functions, copied directly from Fornberg (1995), but with certain enhancements, namely comment lines which start with `Cf2py`. These enhancements are explained in Section 8.8.

The first subroutine does not exist in Fornberg (1995) but is based on the Fortran snippet at the start of Section F.3 on page 187 of that reference, and the header and footer are confected using the code examples given earlier on that page.

```
      SUBROUTINE CHEBPTS(X, N)
C     CREATE N+1 CHEBYSHEV DATA POINTS IN X
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION X(0:N)
Cf2py intent(in) N
Cf2py intent(out) X
Cf2py depend(N) X
      PI = 4.D0*ATAN(1.D0)
      DO 10 I=0,N
 10   X(I) = -COS(PI*I/N)
      RETURN
      END
```

The next subroutine is the fast discrete Fourier transform given on pages 176–9, together with a detailed explanation of how it works. Lines 21–23 contain additional comments to aid the `f2py` tool.

```
1        SUBROUTINE FFT (A,B,IS,N,ID)
2  C-- +-------------------------------------------------------------
3  C-- | A CALL TO FFT REPLACES THE COMPLEX DATA VALUES A(J)+i B(J),
4  C-- | J=0,1,... ,N-1 WITH THEIR TRANSFORM
5  C-- |
6  C-- |                     2 i ID PI K J / N
7  C-- | SUM    (A(K) + iB(K))e              , J=0,1,.. .,N-1
```

```
C-- | K=0..N-1
C-- |
C-- | INPUT AND OUTPUT PARAMETERS
C-- |     A   ARRAY A (0: *), REAL PART OF DATA/TRANSFORM
C-- |     B   ARRAY B (0: *), IMAGINARY PART OF DATA/TRANSFORM
C-- | INPUT PARAMETERS
C-- |   IS    SPACING BETWEEN CONSECUTIVE ELEMENTS IN A AND B
C-- |         (USE IS=+1 FOR ELEMENTS STORED CONSECUTIVELY)
C-- |   N     NUMBER OF DATA VALUES, MUST BE A POWER OF TWO
C-- |   ID    USE +1 OR -1 TO SPECIFY DIRECTION OF TRANSFORM
C-- +-------------------------------------------------------------
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:*),B(0:*)
Cf2py intent(in, out) A, B
Cf2py intent(in) IS, ID
Cf2py integer intent(hide), depend(A) n
      J=0
C---  APPLY PERMUTATION MATRIX ----
      DO 20 I=0,(N-2)*IS,IS
         IF (I.LT.J) THEN
            TR = A(J)
            A(J) = A(I)
            A(I) = TR
            TI   = B(J)
            B(J) = B(I)
            B(I) = TI
         ENDIF
         K = IS*N/2
 10      IF (K.LE.J) THEN
            J = J-K
            K = K/2
            GOTO 10
         ENDIF
 20      J = J+K
C---  PERFORM THE LOG2 N MATRIX-VECTOR MULTIPLICATIONS ---
         S = 0.0D0
         C = -1.0D0
         L = IS
 30      LH=L
         L = L+L
         UR = 1.0D0
         UI = 0.0D0
         DO 50 J=0,LH-IS,IS
            DO 40 I=J,(N-1)*IS,L
```

```
52              IP = I+LH
53              TR = A(IP)*UR-B(IP)*UI
54              TI = A(IP)*UI+B(IP)*UR
55              A(IP) = A(I)-TR
56              B(IP) = B(I)-TI
57              A(I) = A(I)+TR
58  40          B(I) = B(I)+TI
59            TI = UR*S+UI*C
60            UR = UR*C-UI*S
61  50        UI=TI
62          S = SQRT (0.5D0*(1.0D0-C))*ID
63          C = SQRT (0.5D0*(1.0D0+C))
64          IF (L.LT.N*IS) GOTO 30
65        RETURN
66        END
```

Next we include the fast discrete Fourier cosine transform from pages 182–183. We have added comment lines 22–24 as above.

```
1        SUBROUTINE FCT (A,X,N,B)
2   C-- +------------------------------------------------------------------
3   C-- | A CALL TO FCT PLACES IN B(0:N) THE COSINE TRANSFORM OF THE
4   C-- | VALUES IN A(0:N)
5   C-- |
6   C-- | B(J) = SUM C(K)*A(K)*COS(PI*K*J/N) , J=0,1,...,N, K=0..N
7   C-- |
8   C-- | WHERE C(K) = 1.0 FOR K=0,N, C(K) =2.0 FOR K=1,2,...,N-1
9   C__ |
10  C-- | INPUT PARAMETERS:
11  C-- | A  A(0:N) ARRAY WITH INPUT DATA
12  C-- | X  X(0:N) ARRAY WITH CHEBYSHEV GRID POINT LOCATIONS
13  C-- |    X(J) = -COS(PI*J/N) , J=0,1,...,N
14  C-- | N  SIZE OF TRANSFORM - MUST BE A POWER OF TWO
15  C-- | OUTPUT PARAMETER
16  C-- | B  B(0:N) ARRAY RECEIVING TRANSFORM COEFFICIENTS
17  C-- |    (MAY BE IDENTICAL WITH THE ARRAY A)
18  C-- |
19  C-- +------------------------------------------------------------------
20       IMPLICIT REAL*8 (A-H,O-Z)
21       DIMENSION A(0:*),X(0:*),B(0:*)
22  Cf2py intent(in) A, X
23  Cf2py intent(out) B
24  Cf2py integer intent(hide), depend(A) N
25       N2 = N/2
26       A0 = A(N2-1)+A(N2+1)
```

```
      A9 = A(1)
      DO 10 I=2,N2-2,2
         A0 = A0+A9+A(N+1-I)
         A1 = A( I+1)-A9
         A2 = A(N+1-I)-A(N-1-I)
         A3 = A(I)+A(N-I)
         A4 = A(I)-A(N-I)
         A5 = A1-A2
         A6 = A1+A2
         A1 = X(N2-I)
         A2 = X(I)
         A7 = A1*A4+A2*A6
         A8 = A1*A6-A2*A4
         A9 = A(I+1)
         B(I  ) = A3+A7
         B(N-I) = A3-A7
         B(I+1 ) = A8+A5
 10      B(N+1-I) = A8-A5
      B(1) = A(0)-A(N)
      B(0) = A(0)+A(N)
      B(N2 ) = 2.D0*A(N2)
      B(N2+1) = 2.D0*(A9-A(N2+1))
      CALL FFT(B(0),B(1),2,N2,1)
      A0 = 2.D0*A0
      B(N) = B(0)-A0
      B(0) = B(0)+A0
      DO 20 I=1,N2-1
         A1 = 0.5 D0      *(B(I)+B(N-I))
         A2 = 0.25D0/X(N2+I)*(B(I)-B(N-I))
         B(I  ) = A1+A2
 20      B(N-I) = A1-A2
      RETURN
      END
```

Finally, we add three subroutines from Fornberg (1995) pages 188–189 which convert arrays in physical space to those in Chebyshev space and vice versa, and carry out the spatial differentiation in Chebyshev space.

```
      SUBROUTINE FROMCHEB (A,X,N,B)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:N),X(0:N),B(0:N)
Cf2py intent(in) A, X
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) N
      B(0) = A(0)
      A1 = 0.5D0
      DO 10 I=1,N-1
         A1 = -A1
 10      B(I) = A1*A(I)
      B(N) = A(N)
      CALL FCT(B,X,N,B)
      RETURN
      END

      SUBROUTINE TOCHEB (A,X,N,B)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:N),X(0:N),B(0:N)
Cf2py intent(in) A, X
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) N
      CALL FCT(A,X,N,B)
      B1 = 0.5D0/N
      B(0) = B(0)*B1
      B(N) = B(N)*B1
      B1 = 2.D0*B1
      DO 10 I=1,N-1
         B1 = -B1
 10      B(I) = B(I)*B1
      RETURN
      END

      SUBROUTINE DIFFCHEB (A,N,B)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(0:N),B(0:N)
Cf2py intent(in) A
Cf2py intent(out) B
Cf2py integer intent(hide), depend(A) N
      A1 = A(N)
      A2 = A(N-1)
      B(N) = 0.D0
      B(N-1) = 2.D0*N*A1
      A1 = A2
      A2 = A(N-2)
      B(N-2) = 2.D0*(N-1)*A1
      DO 10 I=N-2,2,-1
         A1 = A2
         A2 = A(I-1)
 10      B(I-1) = B(I+1)+2.D0*I*A1
      B(0) = 0.5D0*B(2)+A2
      RETURN
      END
```

References

Ascher, U. M., Mattheij, R. M. M. and Russell, R. D. (1995), *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, SIAM.

Ascher, U. M., Mattheij, R. M. M., Russell, R. D. and Petzold, L. R. (1998), *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM.

Bader, G. and Ascher, U. M. (1987), 'A new basis implementation for a mixed order boundary value odesolver', *SIAM J. Sci. Stat. Comp.* **8**, 483–500.

Bellen, A. and Zennaro, M. (2003), *Numerical Methods for Delay Differential Equations*, Oxford.

Bogacki, P. and Shampine, L. F. (1989), 'A 3(2) pair of Runge–Kutta formulas', *Appl. Math. Lett.* **2**, 321–325.

Boyd, J. P. (2001), *Chebyshev and Fourier Spectral Methods*, second edn, Dover.

Brandt, A. and Livne, O. E. (2011), *Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics*, revised edn, SIAM.

Briggs, W. L., Henson, V. E. and McCormick, S. (2000), *A Multigrid Tutorial*, second edn, SIAM.

Butcher, J. C. (2008), *Numerical Methods for Ordinary Differential Equations*, second edn, Wiley.

Coddington, E. A. and Levinson, N. (1955), *Theory of Ordinary Differential Equations*, McGraw-Hill.

Driver, R. D. (1997), *Ordinary and Delay Differential Equations*, Springer.

Erneux, Y. (2009), *Appplied Delay Differential Applications*, Springer.

Evans, L. C. (2013), *Introduction to Stochastic Differential Equations*, AMS.

Fornberg, B. (1995), *A Practical Guide to Pseudospectral Methods*, Cambridge.

Funaro, D. and Gottlieb, D. (1988), 'A new method of imposing boundary conditions in pseudospectral approximations of hyperbolic equations', *Math. Comp.* **51**, 599–613.

Gardiner, C. W. (2009), *Handbook of Stochastic Methods*, fourth edn, Springer.

Gnuplot Community (2012), 'Gnuplot 4.6, an interactive plotting program', available from `www.gnuplot.info/docs_4.6/gnuplot.pdf`.

Hesthaven, J. S. (2000), 'Spectral penalty methods', *Appl. Num. Maths.* **33**, 23–41.

Hesthaven, J. S., Gottlieb, S. and Gottlieb, D. (2007), *Spectral Methods for Time-Dependent Problems*, Cambridge.

Higham, D. J. (2001), 'An algorithmic introduction to numerical solution of stochastic differential equations', *SIAM Rev.* **43**, 525–546.

Hull, J. (2009), *Options, Futures and Other Derivatives*, seventh edn, Pearson.

Janert, K. (2010), *Gnuplot in Action*, Manning Publications Co.

Kloeden, P. E. and Platen, E. (1992), *Numerical Solution of Stochastic Differential Equations*, Springer.

Lambert, J. D. (1992), *Numerical Methods for Ordinary Differential Systems*, Wiley.

Langtangen, H. P. (2008), *Python Scripting for Computational Science*, third edn, Springer.

Langtangen, H. P. (2009), *A Primer on Scientific Programming with Python*, Springer.

Lutz, M. (2009), *Learning Python*, fourth edn, O'Reilly.

Mackey, M. C. and Glass, L. (1977), 'Oscillation and chaos in physiological control systems', *Science* **197**, 287–289.

Matplotlib Community (2013), 'Matplotlib release 1.2.1', available from `http://matplotlib.org/Matplotlib.pdf`.

Mayavi Community (2011), 'Mayavi: 3d scientific data visualization and plotting in Python', available from `http://docs.enthought.com/mayavi/mayavi/`.

McKinney, W. W. (2012), *Python for Data Analysis*, O'Reilly.

Murray, J. D. (2002), *Mathematical Biology I. An Introduction*, Springer.

Numpy Community (2013*a*), 'Numpy reference release 1.7.0', available from `http://docs.scipy.org/doc/numpy/numpy-ref-1.7.0.pdf`.

Numpy Community (2013*b*), 'Numpy user guide release 1.7.0', available from `http://docs.scipy.org/doc/numpy/numpy-user-1.7.0.pdf`.

Øksendal, B. (2003), *Stochastic Differential Equations*, sixth edn, Springer.

Peitgen, H–O. and Richter, P. H. (1986), *The Beauty of Fractals: Images of Complex Dynamical Systems*, Springer.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2007), *Numerical Recipes: The Art of Scientific Computing*, third edn, Cambridge.

Ramachandandran, P. and Variquaux, G. (2009), 'Mayavi user guide release 3.3.1', available from `http://code.enthought.com/projects/mayavi/docs/development/latex/mayavi/mayavi_user_guide.pdf`.

Rossant, C. (2013), *Learning IPython for Interactive Computing and Data Visualization*, Packt Publishing.

Scipy Community (2012), 'SciPy reference guide release 0.12.0', available from `http://docs.scipy.org/doc/scipy/scipy-ref.pdf`.

Sparrow, C. (1982), *The Lorenz Equations*, Springer.

Tosi, S. (2009), *Matplotlib for Python Developers*, Packt Publishing.

Trefethen, L. N. (2000), *Spectral Methods in MATLAB*, SIAM.

Trottenberg, U., Oosterlee, C. W. and Schüller, A. (2001), *Multigrid*, Academic Press.

van Rossum, G. and Drake Jr., F. L. (2011), *An Introduction to Python*, Network Theory Ltd.

Wesseling, P. (1992), *An Introduction to Multigrid Methods*, Wiley.

Index