7 Ordinary differential equations

7.1 Initial value problems

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

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

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

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

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 $y(t)$

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

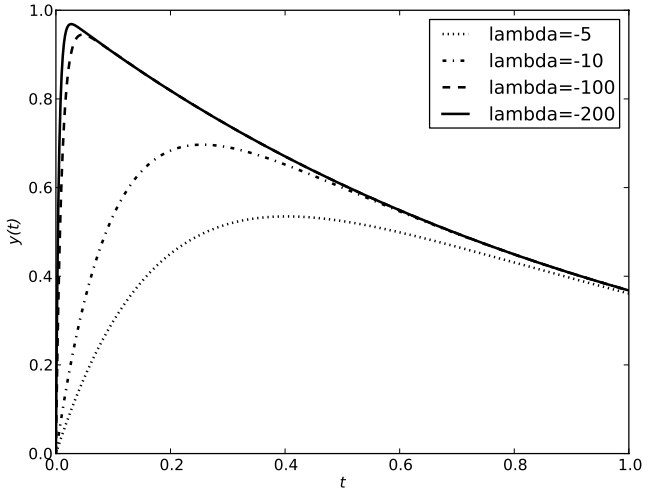

Figure 7.1 The exact solution of the model problem (7.5) with $y_0 = 0$, for various negative values of the parameter λ.

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*

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

or

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

[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

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

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.)

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from scipy.integrate import odeint
4
5  def rhs(Y,t,omega):
6      y,ydot=Y
7      return ydot,-omega**2*y
8
```

```
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.

```
1  import numpy as np
2  from scipy.integrate import odeint
3
```

```
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

$$\dot{x} = \sigma(y - x), \quad \dot{y} = \rho x - y - xz, \quad \dot{z} = xy - \beta z, \tag{7.17}$$

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.

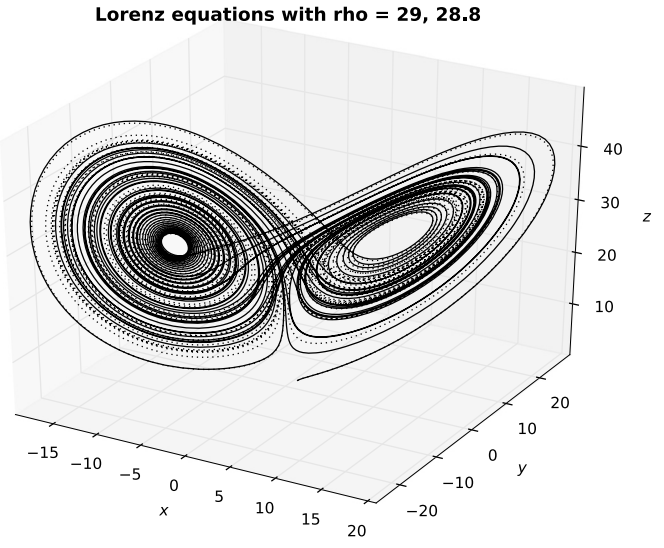

Figure 7.2 Two solution curves for the Lorenz equations, whose ρ parameter varies slightly. The solid and dotted regions show where the solutions differ markedly pointwise. However, the "butterfly structure" appears to be stable.

```
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 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)

$$y_1''(x) + h(x)y_1(x) = 0, \quad x > 0, \quad y_1(0) = 1,\ y_1'(0) = 0,$$
$$y_2''(x) + h(x)y_2(x) = 0, \quad x > 0, \quad y_2(0) = 0,\ y_2'(0) = 1.$$

Linearity implies that the general solution is $y(x) = C_1y_1(x) + C_2y_2(x)$ 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 $Ay_1(1)+C_2y_2(1) = B$. 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.

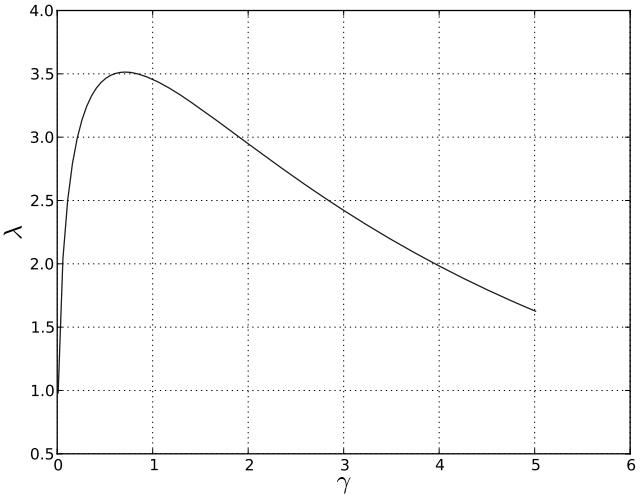

Figure 7.3 The eigenvalue λ for the Bratu problem as a function of the norm γ of the solution.

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)

```

```
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)
```

```
52 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.

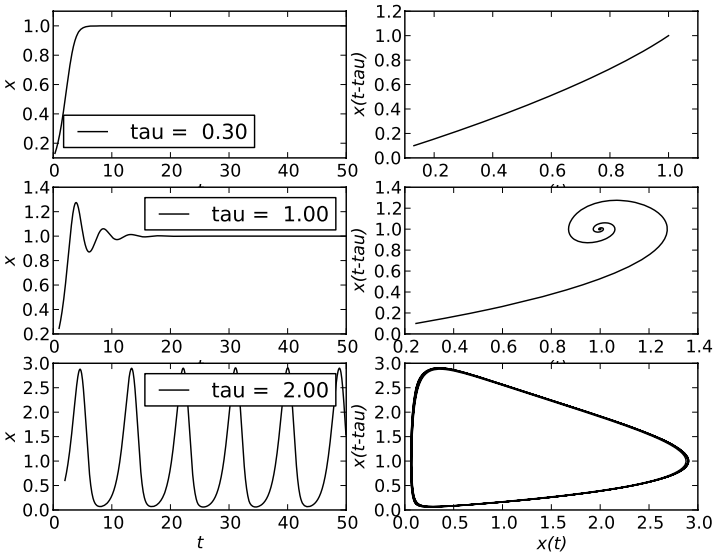

Figure 7.4 Numerical solution of the logistic differential delay equation (7.30) for three choices of the delay τ. The left graph is $x(t)$ as a function of time t. On the right is a quasi-phase plane plot, $x(t-\tau)$ versus $x(t)$.

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.

```
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

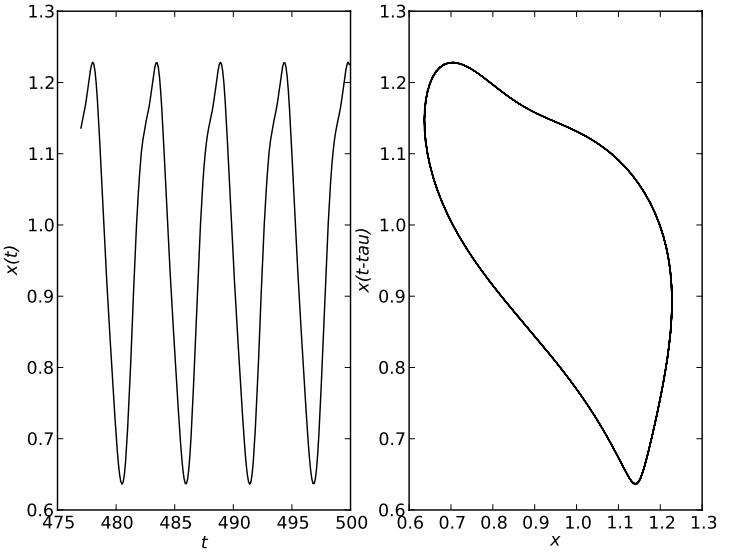

Figure 7.5 Numerical solution of the Mackey–Glass equation with $a = 2$, $b = 1$, $m = 7$, $\tau = 2$ and initial data $x_0(t) = 0.5$. The left graph is $x(t)$ as a function of time t. On the right is a quasi-phase plane plot, $x(t - \tau)$ versus $x(t)$.

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,

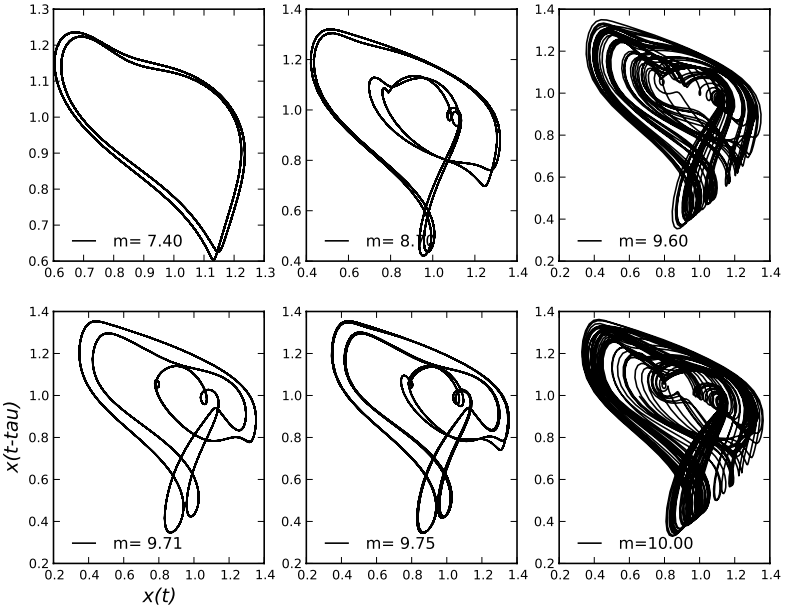

Figure 7.6 Quasi-phase plane portraits for the Mackey–Glass equation. The parameters used are specified in Figure 7.5, except that $m = 7$ there, is changed to the values shown. At $m = 7$, the solution has period 1. At $m = 7.4$, this has doubled, and at $m = 8.7$, the period has quadrupled. Increasing 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.

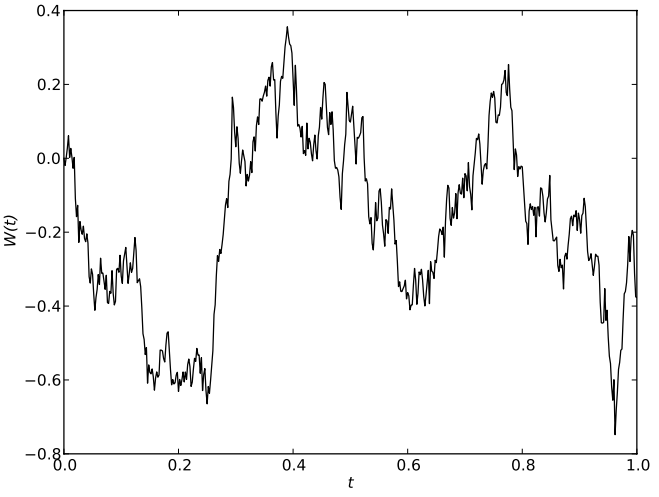

Figure 7.7 A sample of a Wiener process or discrete Brownian motion.

```
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.

```
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

```

```
17 import matplotlib.pyplot as plt
18 plt.ion()
19
20 plt.plot(t,Umean,'b-', label="mean of %d paths" % M)
21 plt.plot(t, Uexact, 'r-',label="exact U")
22 for i in range(5):
23   plt.plot(t,U[i, : ], '--')
24
25 plt.xlabel('t')
26 plt.ylabel('U')
27 plt.title('U= exp(-W(t)/2) for Wiener Process W(t)',
28         weight='bold',size=16)
29 plt.legend(loc='best')
30
31 maxerr=np.max(np.abs(Umean-Uexact))
32 print "With %d paths and %d intervals the max error is %g" % \
33          (M,N,maxerr)
```

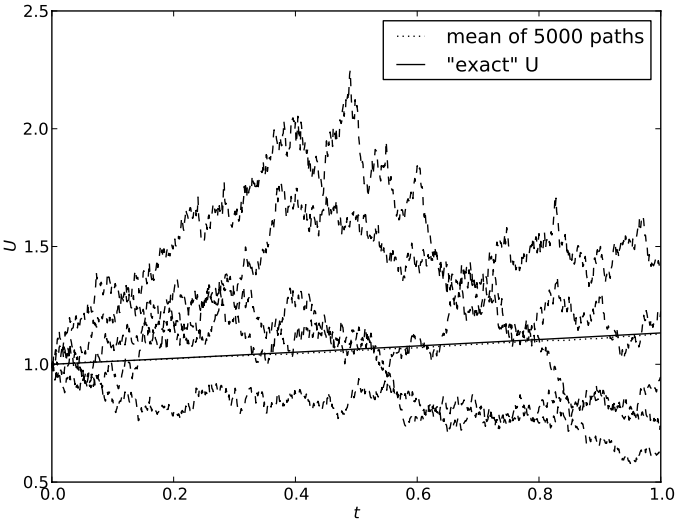

Figure 7.8 A function $U(t) = \exp(-\frac{1}{2}W(t))$ of a discrete Brownian motion $W(t)$. The dashed curves are five sample paths. The solid curve gives the "exact" solution $\langle U\rangle(t) = \exp(\frac{1}{8}t)$ predicted by the Itô formula. Almost hidden beneath it is the dotted curve which maps 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 $k = 0, 1, \ldots, N$ 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

$$\begin{aligned} X_{k+1} =& X_k + a(X_k)\Delta t + b(X_k)\Delta W_k \\ & + \int_{t_k}^{t_{k+1}} dW(s) \int_{t_k}^{s} dW(\tau)\mathcal{M}[b(X(\tau))] \\ & \text{plus integrals over } ds\,d\tau, ds\,dW(\tau) \text{ and } dW(s)\,d\tau. \end{aligned}$$

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)
```

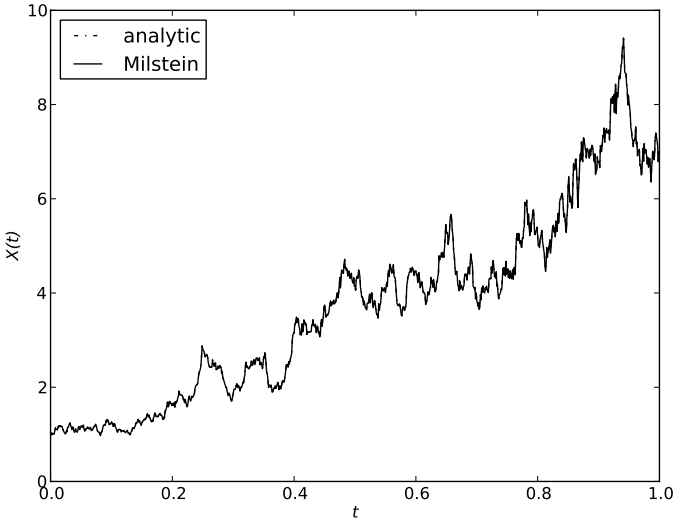

Figure 7.9 An instance of an analytic solution (7.49) of the equation (7.48) and its solution using the Milstein algorithm over 1000 intervals. There is no discernible difference.

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.

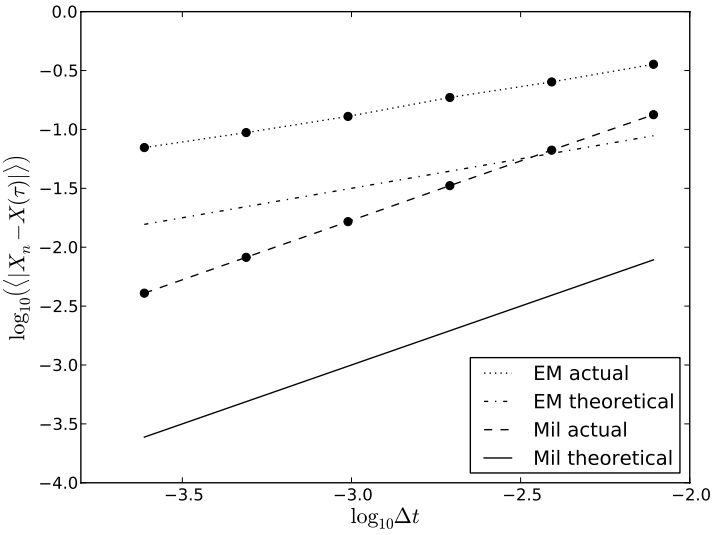

Figure 7.10 The expectation value of the final error in the numerical solution of equation (7.48) as a function of Δt. What matters here are the slopes of the "lines". The Milstein method converges like Δt, while the Euler–Maruyama one converges like $(\Delta t)^{1/2}$.

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.