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.

```
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

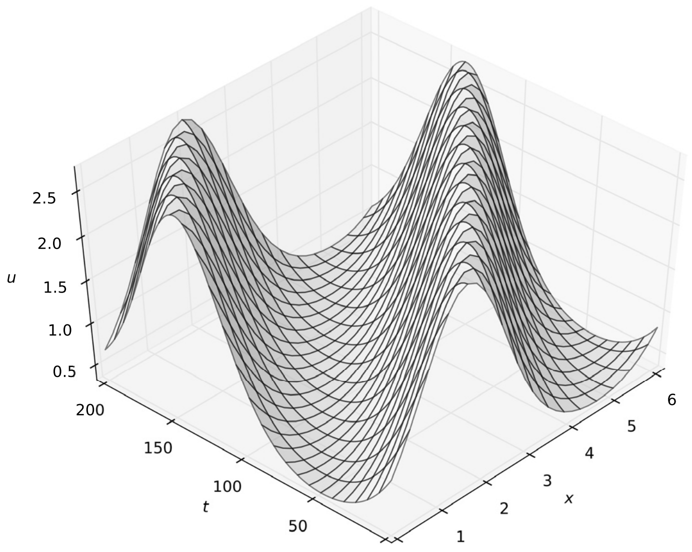

Figure 8.1 The numerical solution $u(t, x)$ of the periodic linear advection problem (8.9) for $x \in [0, 2\pi)$ and $t \in [0, 64\pi]$.

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 $j = 1, 2, \dots, N-1$, 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, $O(N^2)$.

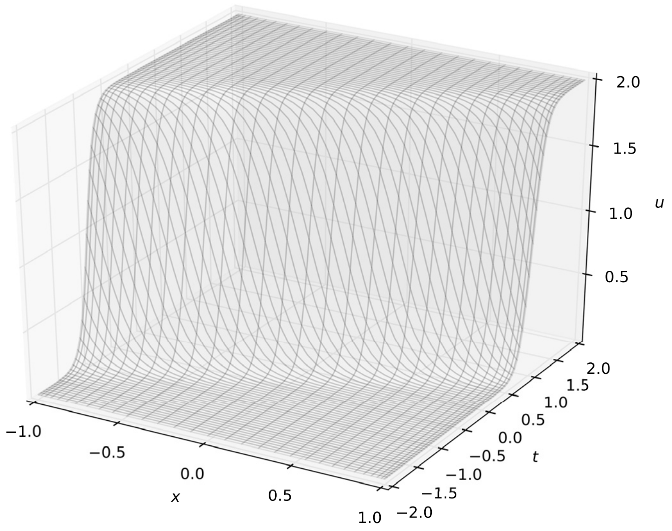

Figure 8.2 The numerical evolution of a "kink" solution (8.16) of the Burgers' equation (8.15) treated as an initial-boundary value problem.

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.

```python
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.