9 Case study: multigrid

In this final chapter, we present an extended example or "case study" of a topic which is relevant to almost all of the theoretical sciences, called multigrid. For many, multigrid is a closed and forbidding book, and so we first look at the type of problems it can be used to solve, and then outline how it works, finally describing broadly how it can be implemented very easily in Python. The rest of the chapter fleshes out the details.

In very many problems, we associate data with points on a spatial grid.[1] For simplicity, we assume that the grid is uniform. In a realistic case, we might want a resolution of say 100 points per dimension, and for a three-dimensional grid we would have 10^6 grid points. Even if we store only one piece of data per grid point, this is a lot of data which we can pack into a vector (one-dimensional array) $\mathbf{u}$ of dimension $N = O(10^6)$. These data are not free but will be restricted either by algebraic or differential equations. Using finite difference (or finite element) approximations, we can ensure that we are dealing with algebraic equations. Even if the underlying equations are non-linear, we have to linearize them (using, e.g., a Newton–Raphson procedure, see Section 9.3) for there is no hope of solving such a large set of non-linear equations. Thus we are faced with the solution of a very large linear system of equations of the form

$$A\mathbf{u} = \mathbf{f}, \tag{9.1}$$

where $\mathbf{f}$ is a N-dimensional vector and A is a $N \times N$ matrix.

Direct solution of the system (9.1), e.g., by matrix inversion, is usually a non-starter. Unless matrix A has a very special form, its inversion involve $O(N^3)$ operations! Instead, we have to use iterative methods (discussed in Section 9.2.1). These aim, by iterating on $\mathbf{u}$ to reduce the *residual* $\mathbf{r} = \mathbf{f} - A\mathbf{u}$ to zero. However, these methods, used in isolation, all suffer from a fatal flaw. To explain this, we need to consider a Fourier representation of the residual $\mathbf{r}$, which will consist of $O(N)$ modes. (This is discussed in more detail in Section 9.1.2.) The iterative methods will reduce very rapidly the magnitudes of about one half of the Fourier modes (the higher-frequency ones) of the residual to near zero, but will have negligible effect on the other half, the lower frequency ones. After that, the iteration stalls.

At this point, the first key idea behind multigrid emerges. Suppose we consider an additional grid of linear dimension $N/2$, i.e., double the spacing. Suppose too that we carefully transcribe the stalled solution $\mathbf{u}$, the source $\mathbf{f}$ and the operator A from the original *fine* grid to the new *coarse* grid. This makes sense because the fine grid residual

[1] In the finite element approach, the vocabulary is different, but the ideas are the same.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

was made up predominantly of low-frequency modes, which will all be accurately represented on the coarse grid. However, half of these will be high frequency, as seen in the coarse grid. By iteration, we can reduce them nearly to zero. If we then carefully transfer the solution data back to the fine grid, we will have removed a further quarter of the Fourier modes of the residual, making three quarters in total.

The second key idea is to realize that we are not restricted to the two-grid model. We could build a third, even-coarser, grid and repeat the process so as to eliminate seven-eighths of the residual Fourier modes. Indeed, we could carry on adding coarser grids until we reach the trivial one. Note too that the additional grids take up in total only a fraction of the original grid size, and this penalty is more than justified by the speed enhancement.

It is important to note that the operations on and information transfers between grids are, apart from the sizes, the same for all grids. This suggests strongly the use of a Python class structure. Further once the two grid model has been constructed, the rest of the procedure can be defined very simply using recursion. In the rest of this chapter, we fill out some of the details, and set up a suite of Python functions to implement them, ending with a non-trivial non-linear example.

Although we have tried to make this chapter reasonably self-contained, interested readers will want to explore further. Arguably the best place for beginners to start is Briggs et al. (2000), although for the first half of their book there are a number of web-based competitors. However, in the second half Briggs et al. (2000) sketch in a clear and concise way a number of applications. Trottenberg et al. (2001) covers much the same material in a more discursive manner which some may find more helpful. The textbook Wesseling (1992) is often overlooked, but its algorithmic approach, especially in chapter 8, is not matched elsewhere. The monograph by the father of the subject Brandt and Livne (2011) contains a wealth of useful information.

9.1 The one-dimensional case

We need a concrete example, which will act as a paradigm for a few fundamental ideas.

9.1.1 Linear elliptic equations

Here is a very simple example of an elliptic differential equation in one-dimension on the interval $[0, 1]$.

$$-u''(x) = f(x), \qquad u(0) = u_l, \quad u(1) = u_r, \tag{9.2}$$

where u_l and u_r are given constants. Note that by adding a linear function to $u(x)$, we can set $u_l = u_r = 0$, and here it is convenient to do so. We impose a uniform grid on $[0, 1]$ with n intervals, i.e., $n+1$ points labelled $x_i = i/n$, for $0 \leqslant i \leqslant n$, and set $u_i = u(x_i)$, and $f_i = f(x_i)$. We then discretize (9.2) as

$$-\frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} = f_i, \qquad 0 < i < n, \tag{9.3}$$

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

where $\Delta x = 1/n$, $u_0 = 0$ and $u_n = 0$. We can rewrite (9.3) as

$$A\mathbf{u} = \mathbf{f}, \tag{9.4}$$

where $\mathbf{u} = \{u_1, \ldots, , u_{n-1}\}$, $\mathbf{f} = \{f_1, \ldots, f_{n-1}\}$, $u_0 = u_n = 0$ and A is a $(n-1) \times (n-1)$ matrix. Many problems that are more complicated than (9.2) can be reduced to the form (9.4) which is the actual problem that we treat.

9.1.2 Smooth and rough modes

In the example above, how large should n be? Based on our experience of spectral methods, it is instructive to consider the use of discrete Fourier transforms. We may extend the domain of definition of u from $[0, 1]$ to $[-1, 1]$ by requiring the continuation to be an odd function of x. Finally, we can extend the domain to the real line by requiring $u(x)$ to be periodic with period 2. We define the discrete Fourier modes by

$$s_k(x) = \sin k\pi x, \quad k = 1, 2, \ldots, n-1. \tag{9.5}$$

Their values at the grid points are

$$s_{k,j} = s_k(x_j) = \sin(kj\pi/n). \tag{9.6}$$

There are precisely $n-1$ linearly independent Fourier modes, which is the same as the number of unknowns, and indeed the Fourier modes form a basis for the vector space of grid functions. Those modes with $1 \leqslant k < \frac{1}{2}n$ are said to be *smooth modes* while those with $\frac{1}{2}n \leqslant k < n$ are said to be *rough modes*. Thus we might choose to expand $u(x)$ and $f(x)$ as, e.g.,

$$u(x) = \sum_{k=1}^{n-1} u^k s_k(x). \tag{9.7}$$

Then $u(x)$ is said to be *smooth* if it is an acceptable approximation to ignore the contribution to the sum from rough modes. Requiring this $u(x)$ to be smooth poses a lower limit on the value of n if reasonable accuracy and definition are required.

9.2 The tools of multigrid

9.2.1 Relaxation methods

In the system (9.4), the matrix A, although usually sparse, will have very large dimensions, especially if the underlying spatial grid has dimension 2 or larger. Direct solution methods, e.g., Gaussian elimination tend to be uncompetitive, and we usually seek an approximate solution by an iterative method, a process of "relaxation". Suppose we rewrite (9.3) as

$$u_i = \tfrac{1}{2}(u_{i-1} + u_{i+1} + \Delta x^2 f_i) \quad 1 \leqslant i \leqslant n-1. \tag{9.8}$$

Jacobi iteration is defined as follows. We start with some approximate solution $\widetilde{u}^{(0)}$ with $\widetilde{u}_0^{(0)} = \widetilde{u}_n^{(0)} = 0$. Using this on the right-hand side of (9.8), we can compute the

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

components of a new approximation $\widetilde{u}^{(1)}$. Suppose we write this process as $\widetilde{u}^{(1)} = J(\widetilde{u}^{(0)})$. Under reasonable conditions, we can show that the kth iterate $\widetilde{u}^{(k)} = J^k(\widetilde{u}^{(0)})$ converges to the exact solution u as $k \to \infty$. In practice, we make a small modification. Let ω be a parameter with $0 < \omega \leqslant 1$, and set

$$\widetilde{u}^{(k+1)} = W(\widetilde{u}^{(k)}) = (1 - \omega)\widetilde{u}^{(k)} + \omega J(\widetilde{u}^{(k)}). \tag{9.9}$$

This is *weighted Jacobi iteration*. For reasons explained below, $\omega = \frac{2}{3}$ is the usual choice.

Suppose we also write the exact solution u as

$$u = \widetilde{u}^{(k)} + e^{(k)},$$

where $e^{(k)}$ is the *error*. It is straightforward to show, using the linearity, that $e^{(k)}$ also satisfies (9.4) but with zero source term f.

We now illustrate this by a concrete example. We set $n = 16$ and choose as initial iterate for the error mode

$$e^{(0)} = s_1 + \tfrac{1}{3}s_{13},$$

where the s_k are the discrete Fourier modes defined in (9.5) and (9.6). This is the superposition of the fundamental mode, and one-third of a rapidly oscillating mode. Figure 9.1 shows this, and the results of the first eight Jacobi iterations. It was produced using

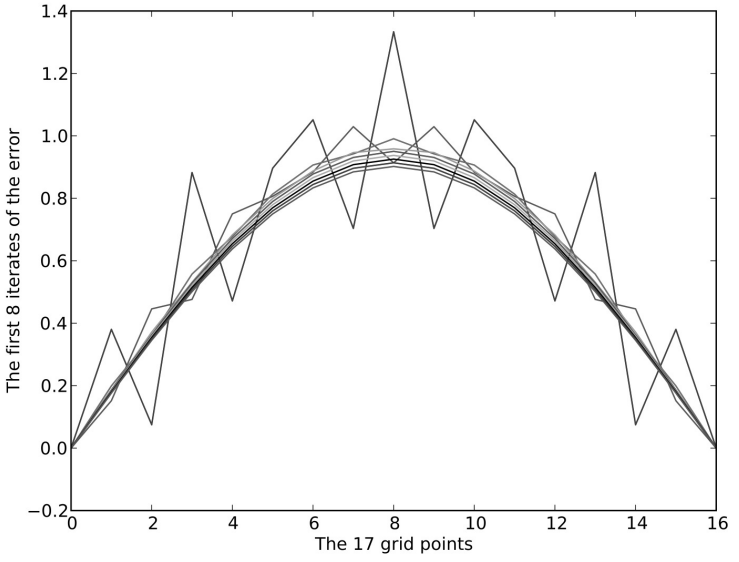

Figure 9.1 Jacobi iteration on a grid with 16 intervals. The initial iterate of the error is a mixture of the first and thirteenth harmonics, the extremely jagged curve. The results of 8 weighted Jacobi iterations are shown. Iteration damps the rough (thirteenth) mode very effectively but is extremely slow at reducing the smooth fundamental mode in the error.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

the following code snippet.

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

N=16
omega=2.0/3.0
max_its=8

x=np.linspace(0,1,N+1)
s1=np.sin(np.pi*x)
s13=np.sin(13*np.pi*x)
e_old=s1+s13/3.0
e=np.zeros_like(e_old)

plt.ion()
plt.plot(e_old)
for it in range(max_its):
    e[1:-1]=(1.0-omega)*e_old[1:-1]+\
              0.5*omega*(e_old[0:-2]+e_old[2: ])
    plt.plot(e)
    e_old=e.copy()
plt.xlabel('The 17 grid points')
plt.ylabel('The first 8 iterates of the error')
```

Here our initial iterate was the superposition of the fundamental mode and one-third of a rapidly oscillating mode, the very spiky curve in Figure 9.1. The remaining curves show the results of the first eight Jacobi iterations. After three iterations, the amplitude of the spikes has all but vanished, but the reduction of the now smooth error $e^{(m)}$ to the exact value zero is extremely slow. You might like to alter the code snippet to verify that the same result holds for initial data made up of other mixtures of smooth and rough modes. The rough modes are damped rapidly, the smooth ones hardly at all. This is the heuristic reason for the name *relaxation*.

It is easy to see the theoretical foundation for these results. Note first that, using a standard trigonometrical identity, s_k defined by (9.6) is an eigenvector of the operator J with eigenvalue $\cos(k\pi/n)$. It follows that s_k is an eigenvector of operator W defined by (9.9), with eigenvalue

$$\lambda_k = 1 - \omega + \omega\cos(k\pi/n) = 1 - 2\omega\sin^2(\tfrac{1}{2}k\pi/n). \tag{9.10}$$

The damping of the kth Fourier mode is determined by $|\lambda_k|$. If $k \ll n$, we see that $\lambda_k \approx 1 - O(1/n^2)$. Thus damping of smooth modes is negligible, especially for large n. Thus although relaxation is convergent onto the exact solution, the ultraslow convergence of smooth modes renders it useless. Note however that if we consider the rough modes, $n/2 \leqslant k < n$, and we choose $\omega = 2/3$, then $|\lambda_k| < 1/3$. The weighted Jacobi

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

iteration then becomes a very efficient damper of rough modes. In multigrid terms, it is a *smoother*.

There are of course many other classical relaxation methods, the most common being *Gauss–Seidel*. The Gauss–Seidel method utilizes components of $u^{(k+1)}$ on the right-hand side of (9.8) as soon as they have been computed. Thus it is really a family of methods depending on the order in which the components are computed. Here for example is a single *red–black Gauss–Seidel* iteration applied to the vector $\widetilde{\mathbf{u}}^{(k)}$

$$\begin{aligned} \widetilde{u}_i^{(k+1)} &= \tfrac{1}{2}(\widetilde{u}_{i-1}^{(k)} + \widetilde{u}_{i+1}^{(k)} + \Delta x^2 f_i) \qquad && i = 2, 4, \ldots, n-2, \\ \widetilde{u}_i^{(k+1)} &= \tfrac{1}{2}(\widetilde{u}_{i-1}^{(k+1)} + \widetilde{u}_{i+1}^{(k+1)} + \Delta x^2 f_i) \qquad && i = 1, 3, \ldots, n-1. \end{aligned} \tag{9.11}$$

Here points with i even are "red", while those with i odd are called "black". This extends to two dimensions, where points with i and j both even or both odd are "red", while the remaining points are "black". The similarity to a chess board gives rise to the name. The behaviour of Gauss–Seidel methods is similar to that of weighted Jacobi. They usually damp the rough modes a little faster, but suffer from the same defect when treating the smooth ones. Thus none of them taken in isolation can be used to solve the original problem (9.4).

Fortunately, multigrid offers a very elegant solution. The central tenet of multigrid is not to consider the problem on a single fixed grid, $n = 16$ above, but instead to look at it on a succession of grids, e.g., $n = 16, 8, 4, 2$. Scaling by factors other than 2 is possible but this is the usual choice. Henceforth, we shall assume that the finest grid size n is a power of 2.

9.2.2 Residual and error

We recall that our aim is to solve the system (9.4)

$$A\mathbf{u} = \mathbf{f}$$

on a "finest grid" of size n with u_0 and u_n specified. Suppose we have some approximation $\widetilde{\mathbf{u}}$ to $\mathbf{u}$. We define the *error* as $\mathbf{e} = \mathbf{u} - \widetilde{\mathbf{u}}$, and the *residual* as $\mathbf{r} = \mathbf{f} - A\widetilde{\mathbf{u}}$. Then it is easy to verify that the *residual equation*

$$A\mathbf{e} = \mathbf{r}, \quad e_0 = e_n = 0, \tag{9.12}$$

holds. It is apparently trite but important to recognize that systems (9.4) and (9.12) are formally identical.

We recall also that our finest grid is a partition of $0 \leqslant x \leqslant 1$ by n intervals. Setting $h = \Delta x = 1/n$, we have set $x_i = ih$, $u(x_i) = u_i$ etc. for $0 \leqslant i \leqslant n$. Also n is sufficiently large that $\mathbf{u}$ and $\mathbf{f}$ are smooth on the finest grid, and n is a power of 2. For the moment, we call the finest grid the *fine grid*. We consider also a *coarse grid* with spacing $H = 2h$, i.e., there are $n/2$ intervals, which is intended to be a replica of the fine grid. It is convenient to distinguish the fine grid quantities by, e.g., $\mathbf{u}^h$ from the corresponding coarse grid ones, e.g., $\mathbf{u}^H$. How are they related? Well for the linear problems under consideration we can require A^H to be obtained from A^h under the transformation $n \to n/2$, $h \to H = 2h$. Note that $\mathbf{u}^h$ is a $n-1$-dimensional vector, whereas $\mathbf{u}^H$ has dimension $n/2 - 1$.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

9.2.3 Prolongation and restriction

Suppose we have a vector $\mathbf{v}^H$ defined on the coarse grid. The process of interpolating it onto the fine grid, called *prolongation* which we denote by ${I_H}^h$, can be done in several ways. The obvious way to handle those fine grid points, which correspond to the coarse grid ones, is a straight copy of the components. The simplest way to handle the others is linear interpolation; thus

$$v^h_{2j} = v^H_j \quad 0 \leqslant j \leqslant \tfrac{1}{2}n, \qquad v^h_{2j+1} = \tfrac{1}{2}(v^H_j + v^H_{j+1}) \quad 0 \leqslant j \leqslant \tfrac{1}{2}n - 1, \tag{9.13}$$

which we denote schematically as $\mathbf{v}^h = {I_H}^h \mathbf{v}^H$. Note that this may leave discontinuous slopes for the v^h_{2j} at internal points, which could be described as the creation of rough modes. We could "improve" on this by using a more sophisticated interpolation process. But in practice, we can remove these unwanted rough modes by applying a few smoothing steps to the fine grid after prolongation.

Next we consider the converse operation, i.e., mapping a fine grid vector $\mathbf{v}^h$ to a coarse grid one $\mathbf{v}^H$, which is called *restriction*, denoted by $\mathbf{v}^H = {I_h}^H \mathbf{v}^h$. The obvious choice is a straight copy

$$v^H_j = v^h_{2j} \qquad 0 \leqslant j \leqslant \tfrac{1}{2}n, \tag{9.14}$$

but the *full weighting process*

$$v^H_0 = v^h_0, \quad v^H_{n/2} = v^h_n, \qquad v^H_j = \tfrac{1}{4}(v^h_{2j-1} + 2v^h_{2j} + v^h_{2j+1}) \quad 1 \leqslant j \leqslant \tfrac{1}{2}n - 1, \tag{9.15}$$

is more commonly used.

Note that there is a problem with all restriction processes. The "source" space is that of vectors of dimension $n - 1$, and the "target" space is that of vectors of dimension $n/2 - 1$. Thus there must be a "kernel" vector space K of dimension $n/2$ such that the restriction of vectors in K is the zero vector. To see this explicitly, consider a Fourier mode $\mathbf{s}^h_k$ on the fine grid with

$$s^h_{k,j} = s^h_k(x_j) = \sin(kj\pi/n),$$

from (9.6). Assuming that j is even and using (9.14), we find

$$s^H_{k,j/2} = s^h_{k,j} = \sin(kj\pi/n).$$

However

$$s^H_{n-k,j/2} = \sin((n-k)j\pi/n) = \sin(j\pi - kj\pi/n) = -\sin(kj\pi/n)$$

since j is even. Thus the images of s^h_k and s^h_{n-k} under restriction are equal and opposite. (The same holds for full weighting.) This phenomenon is well known in Fourier analysis where it is called *aliasing*. Thus there is a serious loss of information in the restriction process. Note however that if s^h_k is smooth, i.e., $k < n/2$, then the aliased mode s^h_{n-k} is coarse. If we applied restriction only to smooth vectors, then the information loss would be minimal. Therefore, a good working rule is to smooth before restricting. Thus we have a "golden rule": smooth after prolongation but before restriction.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

9.3 Multigrid schemes

Before we introduce specific schemes, we shall widen the discussion to consider a more general problem

$$L(\mathbf{u}(\mathbf{x})) = \mathbf{f}(\mathbf{x}), \tag{9.16}$$

on some domain $\Omega \subseteq \mathbf{R}^n$. Here L is a, possibly non-linear, elliptic operator, and $\mathbf{u}(\mathbf{x})$ and $\mathbf{f}(\mathbf{x})$ are smooth vector-valued functions defined on Ω. In order to keep the discussion simple, we shall assume homogeneous *Dirichlet boundary conditions*

$$\mathbf{u}(\mathbf{x}) = \mathbf{0}, \quad \mathbf{x} \in \partial\Omega \tag{9.17}$$

on the boundary $\partial\Omega$ of the domain Ω. More general boundary conditions are treated in the already cited literature.

Suppose $\widetilde{\mathbf{u}}(\mathbf{x})$ is some approximation to the exact solution $\mathbf{u}(\mathbf{x})$ of (9.16). As before, we define the *error* $\mathbf{e}(\mathbf{x})$ and *residual* $\mathbf{r}(\mathbf{x})$ by

$$\mathbf{e}(\mathbf{x}) = \mathbf{u}(\mathbf{x}) - \widetilde{\mathbf{u}}(\mathbf{x}), \quad \mathbf{r}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) - L(\widetilde{\mathbf{u}}(\mathbf{x})). \tag{9.18}$$

We now define the *residual equation* to be

$$L(\widetilde{\mathbf{u}} + \mathbf{e}) - L(\widetilde{\mathbf{u}}) = \mathbf{r}, \tag{9.19}$$

which follows from (9.16) and (9.18). Note that if the operator L is linear, then the left-hand side of (9.19) simplifies to $L(\mathbf{e})$, and so there is no inconsistency with the earlier definition (9.12).

A further pair of remarks is appropriate at this stage. We assume that we have constructed a fine grid Ω^h with spacing h which covers Ω. Using the previous notation, we aim to solve

$$L^h(\mathbf{u}^h) = \mathbf{f}^h \text{ on } \Omega^h, \quad \mathbf{u}^h = 0 \text{ on } \partial\Omega^h, \tag{9.20}$$

rather than the continuous problem (9.16) and (9.17). The important point to note is that it is not necessary to solve the grid equation (9.20) exactly. For even if we found the exact solution, $\mathbf{u}^{h,exact}$, this would not furnish an exact solution $\mathbf{u}^{exact}$ for the continuous problem. Instead, we might expect to obtain

$$\|\mathbf{u}^{exact} - \mathbf{u}^{h,exact}\| \leqslant \epsilon^h, \tag{9.21}$$

for an appropriate norm and some bound ϵ^h which tends to zero as $h \to 0$. Thus it suffices to obtain an approximate grid solution $\widetilde{\mathbf{u}}^h$ with the property

$$\|\mathbf{u}^{h,exact} - \widetilde{\mathbf{u}}^h\| = O(\epsilon^h). \tag{9.22}$$

The second remark concerns what we mean by "smoothing" in this more general non-linear context. Suppose we label the grid points in the interior of Ω^h by $i = 1, 2, \ldots, n-1$ in some particular order. Then (9.20) becomes

$$L_i^h(u_1^h, u_2^h, \ldots, u_{i-1}^h, u_i^h, u_{i+1}^h, \ldots, u_{n-1}^h) = f_i^h, \quad 1 \leqslant i \leqslant n-1.$$

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

The Gauss–Seidel (GS) iteration process solves these equations in sequence using new values as soon as they are computed. Thus

$$L_i^h(u_1^{h,new}, u_2^{h,new}, \ldots, u_{i-1}^{h,new}, u_i^{h,new}, u_{i+1}^{h,old}, \ldots, u_{n-1}^{h,old}) = f_i^h,$$

generates a single scalar equation for the unknown $u_i^{h,new}$. If this is non-linear, then we may use one or more Newton–Raphson (NR) iterations to solve it approximately, e.g.,

$$u_i^{h,new} = u_i^{h,old} - \left[\frac{L_i(\mathbf{u}) - f_i}{\partial L_i(\mathbf{u})/\partial u_i}\right]_{(u_1^{h,new}, u_2^{h,new}, \ldots, u_{i-1}^{h,new}, u_i^{h,new}, u_{i+1}^{h,old}, \ldots, u_{n-1}^{h,old})}. \tag{9.23}$$

The NR iterations should converge quadratically if the iterate is within the domain of attraction, and usually one iteration suffices.

In practice, this GS–NR process acts as an effective smoother, just as GS does in the linear case.

9.3.1 The two-grid algorithm

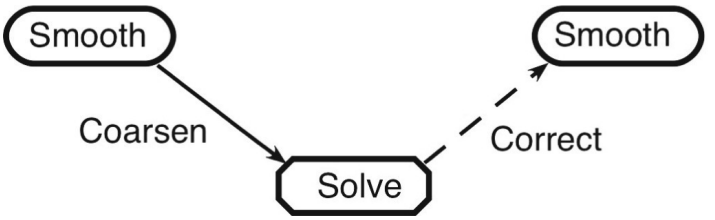

Figure 9.2 The two-grid scheme. The upper level corresponds to a grid with spacing h while the lower one refers to a grid with spacing H where $H > h$. Time is flowing from left to right. The scheme produces a (hopefully) more accurate solution on the finer grid. It may be iterated many times.

Now that we have a smoother, we can introduce an embryonic multigrid algorithm. Suppose that besides the grid Ω^h introduced above, we have a second grid Ω^H where, usually, $H = 2h$. We then apply in order the following steps:

Initial approximation We start with some arbitrary choice $\widetilde{\mathbf{u}}^h$ for an approximation to $\mathbf{u}^h$.

Smooth step We apply a small number ν_1 (typically 1, 2 or 3) of relaxation iteration steps to $\widetilde{\mathbf{u}}^h$, effectively removing the coarse components. Next we compute the residual

$$\mathbf{r}^h = \mathbf{f}^h - L^h(\widetilde{\mathbf{u}}^h). \tag{9.24}$$

Coarsen step Since $\widetilde{\mathbf{u}}^h$ and $\mathbf{r}^h$ are smooth, we can restrict them onto the coarse grid via

$$\widetilde{\mathbf{u}}^H = {I_h}^H \widetilde{\mathbf{u}}^h, \qquad \mathbf{f}^H = {I_h}^H \mathbf{r}^h, \tag{9.25}$$

without losing significant information.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

Solution step We now solve the residual equation

$$L^H(\widetilde{\mathbf{u}}^H + \mathbf{e}^H) - L^H(\widetilde{\mathbf{u}}^H) = \mathbf{f}^H \tag{9.26}$$

for $\mathbf{e}^H$. How this is to be done will be explained later. Notice that, setting $\mathbf{u}^H = \widetilde{\mathbf{u}}^H + \mathbf{e}^H$, the residual equation can be written as

$$L^H(\mathbf{u}^H) = I_h{}^H\mathbf{f}^h + \tau^H, \tag{9.27}$$

where the *"tau-correction"* is

$$\tau^H = L^H(I_h{}^H\widetilde{\mathbf{u}}^h) - I_h{}^H(L^h(\widetilde{\mathbf{u}}^h)), \tag{9.28}$$

and the linearity of I_h^H has been used. The tau-correction is a useful measure of the errors inherent in the grid transfer process.

Correct step Prolong $\mathbf{e}^H$ back to the fine grid. Since $\widetilde{\mathbf{u}}^H + \mathbf{e}^H$ is presumably smooth on the coarse grid, this is an unambiguous operation. Then (hopefully) improve $\widetilde{\mathbf{u}}^h$ by the step $\widetilde{\mathbf{u}}^h \to \widetilde{\mathbf{u}}^h + \mathbf{e}^h$.

Smooth step Finally, we apply a small number ν_2 of relaxation iteration steps to $\widetilde{\mathbf{u}}^h$.

Figure 9.2 illustrates this process. Each horizontal level refers to a particular grid spacing, and lower level(s) refer to coarser grid(s). As we progress from left to right at the top level, we obtain a (hopefully) better approximation to the solution of the initial problem, although we may need to iterate the scheme several times. There are no convincing theoretical arguments for establishing values for the parameters ν_1 and ν_2. Their choice has to be made empirically.

This two-grid algorithm is often referred to as the *Full Approximation Scheme (FAS)*. It is designed to handle non-linear problems and so has to implement the non-linear version of the residual equation (9.19) which requires the transfer of an approximate solution from the fine to the coarse grid. In the linear case, it is only necessary to transfer the error, see, e.g., (9.12).

As presented the two-grid algorithm suffers from two very serious defects. The first appears in the initial approximation step. How is the initial choice of $\widetilde{\mathbf{u}}^h$ to be made? In the linear case, there is no issue, for we might as well choose $\widetilde{\mathbf{u}}^h = \mathbf{0}$. But in the non-linear case, this choice may not lie in the domain of attraction for the Newton–Raphson iteration, which would upset the smoothing step. In Subsection 9.3.3, we remove this obstacle. The second defect is how are we to generate the solution of the residual equation in the solution step? We deal with this in the next section.

9.3.2 The V-cycle scheme

One flaw with the two-grid scheme is the need to solve the residual equation (9.26) on the coarser grid. Because there are rather fewer operations, involved this is less intensive than the original problem, but the cost may still be unacceptable. In this case, a possible solution arises if we augment the coarse grid solution step by another two-grid scheme using an even coarser grid. As can be seen from Figure 9.3, we have an embryonic *V-cycle*. Of course, there is no need to stop with three levels. We may repeat the process

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

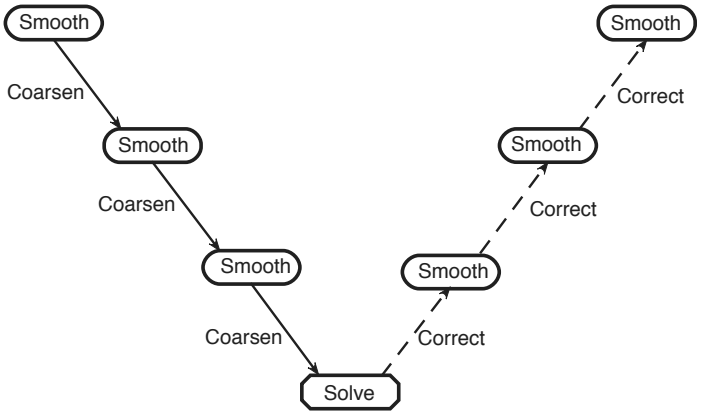

Figure 9.3 The V-cycle scheme. The upper level corresponds to a grid with spacing h while the lower ones refer to progressively coarser grids. Time is flowing from left to right. The scheme produces a (hopefully) more accurate solution on the finer grid. It may be iterated many times, hence the name "cycle". Notice how the "golden rule", smooth before coarsening and after refining, is applied automatically.

adding ever coarser grids until we reach one with precisely one interior point, where we construct an "exact" solution on this grid. Notice that each *V-cycle* comes with the two parameters ν_1 and ν_2 inherited from the two-grid algorithm. It should be apparent from Figure 9.3 that we could iterate on *V-cycles*, and in practice this is usually done ν_0 times.

9.3.3 The full multigrid scheme (FMG)

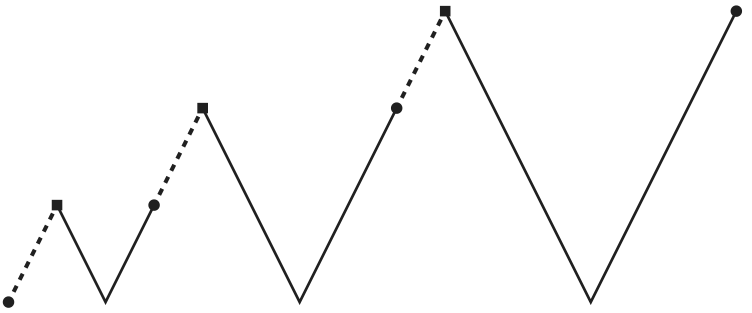

Figure 9.4 The FMG scheme. The circular dots correspond to the final solution and the square dots indicate the initial approximate solution for one or more V-cycles at the given level, while the dotted lines correspond (as before) to an interpolation step. As time progresses from left to right, the transition from the initial to the final solution is via a sequence of V-cycles.

We turn now to the initialization problem for the two-grid algorithm. For linear problems, we can always start from the trivial solution, but this is not possible, usually, for non-linear ones. The V-cycle presented above suggests an obvious solution. Suppose

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

we jump to the coarsest grid, solve the problem there and prolong the solution back to the finer grids to act as a first iterate. In this simple form, the idea doesn't work. Interpolation leads to two types of errors. High-frequency errors may be invisible on the coarse grid, but should be reduced by smoothing. Aliasing errors, i.e., smooth errors introduced by high-frequency effects because on the coarse grid it is mistaken for smooth data, can also occur. However the *Full Multigrid Scheme (FMG)* gets round this problem in a very ingenious way. First we solve the problem on the coarsest grid. Then interpolate the solution to the next coarsest grid. Now perform ν_0 V-cycles on this grid (which turns out to be a two-grid scheme). Next interpolate the solution to the third coarsest grid, perform V-cycles and so on. The scheme is depicted in Figure 9.4. Note that the interpolation steps (the dotted lines in the figure) are not part of the V-cycles. While it may be convenient to use the standard prolongation operator for these steps, there are strong arguments, see, e.g., Brandt and Livne (2011), for using a more accurate scheme.

9.4 A simple Python multigrid implementation

The aim of this chapter is to produce a simple Python implementation of Multigrid using the components we have just outlined. Clearly, V-cycles and FMG are implemented most elegantly using recursion. Most early implementations used Fortran which then did not include recursion, and so lengthy but reasonably efficient kludges were developed to circumvent this. Also it should be clear that the levels or grids have a clear structure, which can be used to clarify the code. Until recently, Fortran lacked the means to implement such structures. Thus most existing codes are very difficult to understand. We could of course replicate them in Python using the `f2py` tool from Chapter 8, but this is not an approach that we would wish to encourage. We shall instead produce a Python implementation which is clear, concise and simple.

The most efficient way to undertake programming a complex task like Multigrid is to split it into groups of subtasks, and we can identify three obvious groups. The first is those tasks which depend only on the number of dimensions and not on the actual problem. Prolongation and restriction fall naturally into this category. Tasks in the second group depend on the number of dimensions and the specific problem. The obvious examples are smoothing and the computation of residuals. The final group consists of tasks specific to multigrid. Our implementation of this final group will make assumptions neither about the number of dimensions nor the details of the specific problem being studied.

Experience shows that tasks in the first two groups are the computationally expensive ones. The code used here is a reasonably efficient *numpy* one. However, it would be very easy, using e.g., `f2py`, to implement them using a compiled language such as Fortran with a Python wrapper, because the algorithms used involve just simple loops. However, the situation is reversed for the third group. The clearest formal definition of multigrid is recursive in nature. Recursion is fully implemented in Python, but only partially so in more recent dialects of Fortran. The concept of grids with different levels of refinement

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

(the vertical spacing in the figures above) is easily implemented using the Python *class* construct, which is much simpler to implement than its C++ counterpart.[2]

For the sake of brevity, we have eliminated all extraneous features and have reduced comments to the absolute minimum. The scripts presented here should be thought of as a framework for a more user-friendly suite of programmes.

9.4.1 Utility functions

The file `util.py` contains three functions covering L^2-norms (mean square), prolongation and restriction in two dimensions. It should be a routine matter to write down the equivalents in one, three or any other number of dimensions.

```
# File: util.py: useful 2D utilities.

import numpy as np

def l2_norm(a,h):
    return h*np.sqrt(np.sum(a**2))

def prolong_lin(a):
    pshape=(2*np.shape(a)[0]-1,2*np.shape(a)[1]-1)
    p=np.empty(pshape,float)
    p[0: :2,0: :2]=a[0: ,0: ]
    p[1:-1:2,0: :2]=0.5*(a[0:-1,0: ]+a[1: ,0: ])
    p[0: :2,1:-1:2]=0.5*(a[0: ,0:-1]+a[0: ,1: ])
    p[1:-1:2,1:-1:2]=0.25*(a[0:-1,0:-1]+a[1: ,0:-1]+
        a[0:-1,1: ]+a[1: ,1: ])
    return p

def restrict_hw(a):
    rshape=(np.shape(a)[0]/2+1,np.shape(a)[1]/2+1)
    r=np.empty(rshape,float)
    r[1:-1,1:-1]=0.5*a[2:-1:2,2:-1:2]+ \
                 0.125*(a[2:-1:2,1:-2:2]+a[2:-1:2,3: :2]+
                        a[1:-2:2,2:-1:2]+a[3: :2,2:-1:2])
    r[0,0: ]=a[0,0: :2]
    r[-1,0: ]=a[-1,0: :2]
    r[0: ,0]=a[0: :2,0]
    r[0: ,-1]=a[0: :2,-1]
    return r

```

[2] Much of the complexity in C++ classes comes from the security requirement; a normal class user should not be able to see the implementation details for the class algorithms. This feature is irrelevant for most scientific users and is not offered in Python.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

```
30 #-----------------------------------------------
31 if __name__=='__main__':
32     a=np.linspace(1,81,81)
33     b=a.reshape(9,9)
34     c=restrict_hw(b)
35     d=prolong_lin(c)
36     print "original grid\n",b
37     print "with spacing 1 its norm is ",l2_norm(b,1)
38     print "\n restricted grid\n",c
39     print "\n prolonged restricted grid\n",d
```

The function in lines 5 and 6 returns an approximation to the L^2-norm of any array `a` with spacing `h`. Lines 8–16 implement prolongation using linear interpolation for a two-dimensional array `a`, using the obvious generalization of the one-dimensional version (9.13). Next lines 18–23 show how to implement restriction for the interior points by *half weighting*, which is the simplest generalization of full weighting defined in one dimension by (9.15) in a two-dimensional context. Finally, lines 24–27 implement a simple copy for the boundary points. The supplementary test suite, lines 31–39, offers some simple partial checks on these three functions.

9.4.2 Smoothing functions

The smoothing functions depend on the problem being considered. We treat here the test problem for non-linear multigrid, which is discussed in Press et al. (2007),

$$L(u) = u_{xx} + u_{yy} + u^2 = f(x, y) \tag{9.29}$$

on the domain $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$ with

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0. \tag{9.30}$$

We may write the discretized version in the form

$$F_{i,j} \equiv \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} + {u_{i,j}}^2 - f_{i,j} = 0. \tag{9.31}$$

We regard $F_{i,j} = 0$ as a non-linear equation to be solved for $u_{i,j}$. Noting that

$$\frac{\partial F_{i,j}}{\partial u_{i,j}} = -\frac{4}{h^2} + 2u_{i,j},$$

the Newton–Raphson iteration scheme is

$$(u_{i,j})_{new} = u_{i,j} - \frac{F_{i,j}}{-4/h^2 + 2u_{i,j}}. \tag{9.32}$$

Here is a module with functions to carry out a single iteration of Gauss–Seidel red–black smoothing, and to compute the residual. As usual docstrings and helpful comments have been omitted to ensure brevity.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

```
# File: smooth.py: problem dependent 2D utilities.

import numpy as np

def get_lhs(u,h2):
    w=np.zeros_like(u)
    w[1:-1,1:-1]=(u[0:-2,1:-1]+u[2: ,1:-1]+
                u[1:-1,0:-2]+u[1:-1,2: ]-
                4*u[1:-1,1:-1])/h2+u[1:-1,1:-1]*u[1:-1,1:-1]
    return w

def gs_rb_step(v,f,h2):
    u=v.copy()
    res=np.empty_like(v)

    res[1:-1:2,1:-1:2]=(u[0:-2:2,1:-1:2]+u[2: :2,1:-1:2]+
                    u[1:-1:2,0:-2:2]+u[1:-1:2,2: :2]-
                    4*u[1:-1:2,1:-1:2])/h2 +\
                    u[1:-1:2,1:-1:2]**2-f[1:-1:2,1:-1:2]
    u[1:-1:2, 1:-1:2]-=res[1:-1:2,1:-1:2]/(
                             -4.0/h2+2*u[1:-1:2,1:-1:2])

    res[2:-2:2,2:-2:2]=(u[1:-3:2,2:-2:2]+u[3:-1:2,2:-2:2]+
                    u[2:-2:2,1:-3:2]+u[2:-2:2,3:-1:2]-
                    4*u[2:-2:2,2:-2:2])/h2+\
                    u[2:-2:2,2:-2:2]**2-f[2:-2:2,2:-2:2]
    u[2:-2:2,2:-2:2]-=res[2:-2:2,2:-2:2]/(
                            -4.0/h2+2*u[2:-2:2,2:-2:2])

    res[2:-2:2,1:-1:2]=(u[1:-3:2,1:-1:2]+u[3:-1:2,1:-1:2]+
                    u[2:-2:2,0:-2:2]+u[2:-2:2,2: :2]-
                    4*u[2:-2:2,1:-1:2])/h2 +\
                    u[2:-2:2,1:-1:2]**2-f[2:-2:2,1:-1:2]
    u[2:-2:2,1:-1:2]-=res[2:-2:2,1:-1:2]/(
                            -4.0/h2+2*u[2:-2:2,1:-1:2])

    res[1:-1:2,2:-2:2]=(u[0:-2:2,2:-2:2]+u[2: :2,2:-2:2]+
                    u[1:-1:2,1:-3:2]+u[1:-1:2,3:-1:2]-
                    4*u[1:-1:2,2:-2:2])/h2+\
                    u[1:-1:2,2:-2:2]**2-f[1:-1:2,2:-2:2]
    u[1:-1:2,2:-2:2]-=res[1:-1:2,2:-2:2]/(
                            -4.0/h2+2*u[1:-1:2,2:-2:2])

```

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

```
    return u


def solve(rhs):
    h=0.5
    u=np.zeros_like(rhs)
    fac=2.0/h**2
    dis=np.sqrt(fac**2+rhs[1,1])
    u[1,1]=-rhs[1,1]/(fac+dis)
    return u
```

The first function `get_lhs` takes a current $\mathbf{u}$ and steplength squared h^2 and returns $L(\mathbf{u})$ for this level. The second function `gs_rb_step` requires in addition the right-hand side $\mathbf{f}$ and combines a Newton–Raphson iteration with a red–black Gauss–Seidel one to produce a smoothing step. Note that nowhere do we need to specify the sizes of the arrays: the information about which level is being used is contained in the `h2` parameter. The final function returns the exact solution of the discretized equation on the coarsest 3×3 grid.

These are the most lengthy and complicated functions we shall need. By abstracting them into a separate module, we can construct a test suite (not shown) in order to ensure that they deliver what is intended.

9.4.3 Multigrid functions

We now present a Python module which defines a class `Grid` which will carry out FAS V-cycles and FMG. The basic idea is for a `Grid` to represent all of the data and functions for each of the horizontal levels in Figures 9.3 and 9.4. The basic difference between levels is merely one of size, and so it is appropriate to encapsulate them in a class. The data include a pointer to a coarser `Grid`, i.e., the next level below the one under discussion. Because of the complexity of the ideas being used, we have included the docstrings and various **print** statements. Once the reader understands what is going on, the **print** statements may safely be deleted.

```
# File: grid.py: linked grid structures and associated algorithms

import numpy as np
from util import l2_norm, restrict_hw, prolong_lin
from smooth import gs_rb_step, get_lhs, solve

class Grid:
    """
        A Grid contains the structures and algorithms for a
        given level together with a pointer to a coarser grid.
    """
    def __init__(self,name,steplength,u,f,coarser=None):
```

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

```
        self.name=name
        self.co=coarser         # pointer to coarser grid
        self.h=steplength       # step length h
        self.h2=steplength**2   # h**2
        self.u=u                # improved variable array
        self.f=f                # right hand side array

    def __str__(self):
        """ Generate an information string about this level. """
        sme='Grid at %s with steplength = %0.4g\n' % (
                            self.name, self.h)
        if self.co:
            sco='Coarser grid with name %s\n' % self.co.name
        else:
            sco='No coarser grid\n'
        return sme+sco

    def smooth(self,nu):
        """
            Carry out Newton--Raphson/Gauss--Seidel red--black
            iteration u-->u, nu times.
        """
        print 'Relax in %s for %d times' % (self.name,nu)
        v=self.u.copy()
        for i in range(nu):
            v=gs_rb_step(v,self.f,self.h2)
        self.u=v

    def fas_v_cycle(self,nu1,nu2):
        """ Recursive implementation of (nu1, nu2) FAS V-Cycle."""
        print 'FAS-V-cycle called for grid at %s\n' % self.name
        # Initial smoothing
        self.smooth(nu1)
        if self.co:
            # There is a coarser grid
            self.co.u=restrict_hw(self.u)
            # Get residual
            res=self.f-get_lhs(self.u,self.h2)
            # get coarser f
            self.co.f=restrict_hw(res)+get_lhs(self.co.u,
                                        self.co.h2)
            oldc=self.co.u
            # Get new coarse solution
            newc=self.co.fas_v_cycle(nu1,nu2)
```

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

```
57            # Correct current u
58            self.u+=prolong_lin(newc-oldc)
59        self.smooth(nu2)
60        return self.u
61
62    def fmg_fas_v_cycle(self,nu0,nu1,nu2):
63        """ Recursive implementation of FMG-FAS-V-Cycle"""
64        print 'FMG-FAS-V-cycle called for grid at %s\n' % self.name
65        if not self.co:
66            # Coarsest grid
67            self.u=solve(self.f)
68        else:
69            # Restrict f
70            self.co.f=restrict_hw(self.f)
71            # Use recursion to get coarser u
72            self.co.u=self.co.fmg_fas_v_cycle(nu0,nu1,nu2)
73            # Prolong to current u
74            self.u=prolong_lin(self.co.u)
75        for it in range(nu0):
76            self.u=self.fas_v_cycle(nu1, nu2)
77        return self.u
```

Thanks to the use of classes, the code is remarkably simple and concise. Note that `None` in line 12 is a special Python variable which can take any type and always evaluates to zero or `False`. By default there is not a coarser grid, and so when we actually construct a *list* of levels we need to link manually each grid to the next coarser one. The function `__init__`, lines 12–18 constructs a Grid level. The function `__str__`. lines 20–28 constructs a string describing the current grid. Their use will be described soon. Lines 30–39 give the `smooth` function, which simply carries out `nu` iterations of the Newton–Raphson/Gauss–Seidel smoothing process on the current grid. The function `fas_v_cycle` on lines 41–60 is non-trivial. If we are at the coarsest grid level, it ignores lines 47–58 and simply carries out $\nu_1 + \nu_2$ smoothing steps and then returns. In the more general case, lines 45–54 carry out ν_1 "smooth" steps followed by a "coarsen" step of the two-grid scheme. Line 56 then carries out the "solve" step by recursion, and finally lines 58 and 59 carry out the remaining "correct" and finally ν_2 "smooth" steps. It is not the most efficient code. For useful hints on achieving efficiency, see the discussion of an equivalent code in Press et al. (2007). Finally, lines 62–77 implement a non-trivial function for carrying out full multigrid FAS cycles. The parameter ν_0 controls how many *V-cycles* are carried out. Notice how closely the Python code mimics the earlier theoretical description, which makes it easier to understand and to develop, a great saving of time and effort.

We shall assume that the code snippet above has been saved as `grid.py`.

Finally, we create the following snippet in a file `rungrid.py` to show how to use the functions. We have chosen to replicate the worked example from Press et al. (2007)

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

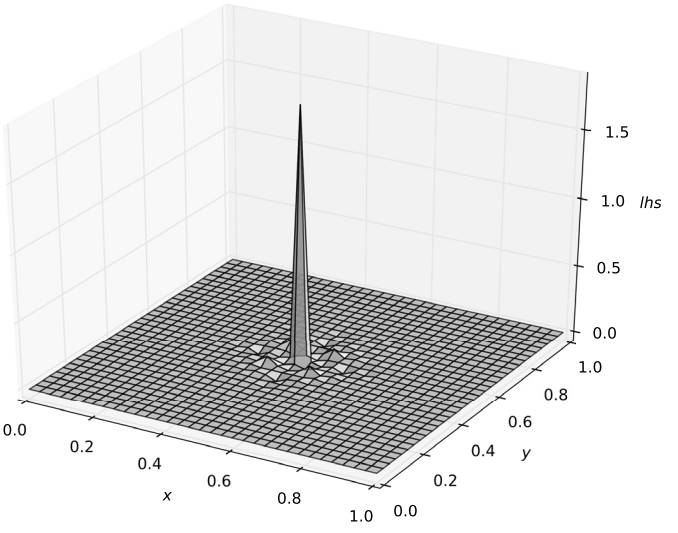

Figure 9.5 The left-hand side of equation (9.15) plotted on the 32×32 computational grid. The original source was zero everywhere apart from the central point where it took the value 2.

which studies (9.29) on a 32×32 grid with $f(x, y) = 0$ except at the mid-point where it takes the value 2.

```
# File rungrid.py: runs multigrid programme.

import numpy as np
from grid import Grid
from smooth import get_lhs

n_grids=5
size_init=2**n_grids+1
size=size_init-1
h=1.0/size
foo=[]                          # container for grids

for k in range(n_grids):        # set up list of grids
   u=np.zeros((size+1,size+1),float)
   f=np.zeros_like(u)
   name='Level '+str(n_grids-1-k)
   temp=Grid(name,h,u,f)
   foo.append(temp)
   size/=2
   h*=2

```

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

```
22 for k in range(1,n_grids):          # set up coarser links
23    foo[k-1].co=foo[k]
24
25 # Check that the initial construction works
26 for k in range(n_grids):
27    print foo[k]
28
29 # Set up data for the NR problem
30 u_init=np.zeros((size_init,size_init))
31 f_init=np.zeros_like(u_init)
32 f_init[size_init/2,size_init/2]=2.0
33 foo[0].u=u_init                     # trial solution
34 foo[0].f=f_init
35
36 foo[0].fmg_fas_v_cycle(1,1,1)
37
38 # As a check get lhs of equation for final grid
39
40 lhs=get_lhs(foo[0].u,foo[0].h2)
41
42 import matplotlib.pyplot as plt
43 from mpl_toolkits.mplot3d import Axes3D
44
45 plt.ion()
46 fig=plt.figure()
47 ax=Axes3D(fig)
48
49 xx,yy=np.mgrid[0:1:1j*size_init,0:1:1j*size_init]
50 ax.plot_surface(xx,yy,lhs,rstride=1,cstride=1,alpha=0.4)
```

Lines 7–11 set up initial parameters, including an empty *list* `foo` which is a container for the `Grid` hierarchy. Each pass of the loop in lines 13–20 constructs a `Grid` (line 17) and adds it to the *list* (line18). Next the loop in lines 22 and 23 links `Grid`s to the coarser neighbour. Lines 26–27 check that nothing is amiss by exercising the `__str__` function. The lines 30–34 set up some initial data for the problem actually treated by Press et al. (2007). Finally, line 36 constructs the solution.

In order to perform one (of many) checks on the solution, we reconstruct the left-hand side of the equation evaluated on the finest grid in line 40. The remainder of the snippet plots this as a surface, shown in Figure 9.5. The initial source was zero everywhere except at the central point where it took the value 2. The numerical results, which are identical to those obtained with the code of Press et al. (2007), give the peak value as 1.8660. Increasing the grid resolution to 64×64 or 128×128 does not change this value significantly. The L^2-norm of the error is always $O(h^2)$, but this disguises the effect of the discontinuity.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press

This example was chosen to run in a fraction of a second on most reasonably modern machines. If you hack the code to handle a more realistic (and complicated) problem for you, it might run rather more slowly. How do we "improve" the code? The first step is to verify carefully that the code does produce convergent results. Next we might think about improving the efficiency of the Python code. The first step is to *profile* the code. Assuming you are using *IPython*, then replace the command `run rungrid` with `run -p rungrid`. This appends to the output the results from the python *profiler* which show the time spent in each function. If there is a "bottleneck", it will show up at the top of the list. Almost certainly, any bottleneck will be in the file `smooth.py`. It may be that the python code for such functions is obviously inefficient and can be improved. More rarely, vectorized numpy code is too slow. Then the simplest effective solution is to use `f2py`, as advocated in Chapter 8. This does not mean recoding the whole programme in Fortran, but only the bottleneck functions, which should be computationally intensive but with a simple overall structure, so that no great Fortran expertise is needed.

https://doi.org/10.1017/CBO9781107447875.010 Published online by Cambridge University Press