standard tridiagonal algorithm. Given $\mathbf{u}^n$, one solves (19.5.36) for $\mathbf{u}^{n+1/2}$, substitutes on the right-hand side of (19.5.37), and then solves for $\mathbf{u}^{n+1}$. The key question is how to choose the iteration parameter r, the analog of a choice of timestep for an initial value problem.

As usual, the goal is to minimize the spectral radius of the iteration matrix. Although it is beyond our scope to go into details here, it turns out that, for the optimal choice of r, the ADI method has the same rate of convergence as SOR. The individual iteration steps in the ADI method are much more complicated than in SOR, so the ADI method would appear to be inferior. This is in fact true if we choose the same parameter r for every iteration step. However, it is possible to choose a *different r* for each step. If this is done optimally, then ADI is generally more efficient than SOR. We refer you to the literature [1-4] for details.

Our reason for not fully implementing ADI here is that, in most applications, it has been superseded by the multigrid methods described in the next section. Our advice is to use SOR for trivial problems (e.g., 20×20), or for solving a larger problem once only, where ease of programming outweighs expense of computer time. Occasionally, the sparse matrix methods of §2.7 are useful for solving a set of difference equations directly. For production solution of large elliptic problems, however, multigrid is now almost always the method of choice.

CITED REFERENCES AND FURTHER READING:

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6.

Young, D.M. 1971, *Iterative Solution of Large Linear Systems* (New York: Academic Press). [1]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§8.3–8.6. [2]

Varga, R.S. 1962, *Matrix Iterative Analysis* (Englewood Cliffs, NJ: Prentice-Hall). [3]

Spanier, J. 1967, in *Mathematical Methods for Digital Computers, Volume 2* (New York: Wiley), Chapter 11. [4]

19.6 Multigrid Methods for Boundary Value Problems

Practical multigrid methods were first introduced in the 1970s by Brandt. These methods can solve elliptic PDEs discretized on N grid points in $O(N)$ operations. The "rapid" direct elliptic solvers discussed in §19.4 solve special kinds of elliptic equations in $O(N \log N)$ operations. The numerical coefficients in these estimates are such that multigrid methods are comparable to the rapid methods in execution speed. Unlike the rapid methods, however, the multigrid methods can solve general elliptic equations with nonconstant coefficients with hardly any loss in efficiency. Even nonlinear equations can be solved with comparable speed.

Unfortunately there is not a single multigrid algorithm that solves all elliptic problems. Rather there is a multigrid technique that provides the framework for solving these problems. You have to adjust the various components of the algorithm within this framework to solve your specific problem. We can only give a brief introduction to the subject here. In particular, we will give two sample multigrid routines, one linear and one nonlinear. By following these prototypes and by perusing the references [1-4], you should be able to develop routines to solve your own problems.

There are two related, but distinct, approaches to the use of multigrid techniques. The first, termed "the multigrid method," is a means for speeding up the convergence of a traditional relaxation method, as defined by you on a grid of pre-specified fineness. In this case, you need define your problem (e.g., evaluate its source terms) only on this grid. Other, coarser, grids defined by the method can be viewed as temporary computational adjuncts.

The second approach, termed (perhaps confusingly) "the full multigrid (FMG) method," requires you to be able to define your problem on grids of various sizes (generally by discretizing the same underlying PDE into different-sized sets of finite-difference equations). In this approach, the method obtains successive solutions on finer and finer grids. You can stop the solution either at a pre-specified fineness, or you can monitor the truncation error due to the discretization, quitting only when it is tolerably small.

In this section we will first discuss the "multigrid method," then use the concepts developed to introduce the FMG method. The latter algorithm is the one that we implement in the accompanying programs.

From One-Grid, through Two-Grid, to Multigrid

The key idea of the multigrid method can be understood by considering the simplest case of a two-grid method. Suppose we are trying to solve the linear elliptic problem

$$\mathcal{L}u = f \tag{19.6.1}$$

where $\mathcal{L}$ is some linear elliptic operator and f is the source term. Discretize equation (19.6.1) on a uniform grid with mesh size h. Write the resulting set of linear algebraic equations as

$$\mathcal{L}_h u_h = f_h \tag{19.6.2}$$

Let $\widetilde{u}_h$ denote some approximate solution to equation (19.6.2). We will use the symbol u_h to denote the exact solution to the difference equations (19.6.2). Then the *error* in $\widetilde{u}_h$ or the *correction* is

$$v_h = u_h - \widetilde{u}_h \tag{19.6.3}$$

The *residual* or *defect* is

$$d_h = \mathcal{L}_h \widetilde{u}_h - f_h \tag{19.6.4}$$

(Beware: some authors define residual as minus the defect, and there is not universal agreement about which of these two quantities 19.6.4 defines.) Since $\mathcal{L}_h$ is linear, the error satisfies

$$\mathcal{L}_h v_h = -d_h \tag{19.6.5}$$

At this point we need to make an approximation to $\mathcal{L}_h$ in order to find v_h. The classical iteration methods, such as Jacobi or Gauss-Seidel, do this by finding, at each stage, an approximate solution of the equation

$$\widehat{\mathcal{L}}_h \widehat{v}_h = -d_h \tag{19.6.6}$$

where $\widehat{\mathcal{L}}_h$ is a "simpler" operator than $\mathcal{L}_h$. For example, $\widehat{\mathcal{L}}_h$ is the diagonal part of $\mathcal{L}_h$ for Jacobi iteration, or the lower triangle for Gauss-Seidel iteration. The next approximation is generated by

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \widehat{v}_h \tag{19.6.7}$$

Now consider, as an alternative, a completely different type of approximation for $\mathcal{L}_h$, one in which we "coarsify" rather than "simplify." That is, we form some appropriate approximation $\mathcal{L}_H$ of $\mathcal{L}_h$ on a coarser grid with mesh size H (we will always take $H = 2h$, but other choices are possible). The residual equation (19.6.5) is now approximated by

$$\mathcal{L}_H v_H = -d_H \tag{19.6.8}$$

Since $\mathcal{L}_H$ has smaller dimension, this equation will be easier to solve than equation (19.6.5). To define the defect d_H on the coarse grid, we need a *restriction operator* $\mathcal{R}$ that restricts d_h to the coarse grid:

$$d_H = \mathcal{R} d_h \tag{19.6.9}$$

The restriction operator is also called the *fine-to-coarse operator* or the *injection operator*. Once we have a solution $\widetilde{v}_H$ to equation (19.6.8), we need a *prolongation operator* $\mathcal{P}$ that prolongates or interpolates the correction to the fine grid:

$$\widetilde{v}_h = \mathcal{P} \widetilde{v}_H \tag{19.6.10}$$

The prolongation operator is also called the *coarse-to-fine operator* or the *interpolation operator*. Both $\mathcal{R}$ and $\mathcal{P}$ are chosen to be linear operators. Finally the approximation $\widetilde{u}_h$ can be updated:

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \widetilde{v}_h \tag{19.6.11}$$

One step of this *coarse-grid correction scheme* is thus:

Coarse-Grid Correction

- Compute the defect on the fine grid from (19.6.4).
- Restrict the defect by (19.6.9).
- Solve (19.6.8) exactly on the coarse grid for the correction.
- Interpolate the correction to the fine grid by (19.6.10).

- Compute the next approximation by (19.6.11).

Let's contrast the advantages and disadvantages of relaxation and the coarse-grid correction scheme. Consider the error v_h expanded into a discrete Fourier series. Call the components in the lower half of the frequency spectrum the *smooth components* and the high-frequency components the *nonsmooth components*. We have seen that relaxation becomes very slowly convergent in the limit $h \to 0$, i.e., when there are a large number of mesh points. The reason turns out to be that the smooth components are only slightly reduced in amplitude on each iteration. However, many relaxation methods reduce the amplitude of the nonsmooth components by large factors on each iteration: They are good *smoothing operators*.

For the two-grid iteration, on the other hand, components of the error with wavelengths $\lesssim 2H$ are not even representable on the coarse grid and so cannot be reduced to zero on this grid. But it is exactly these high-frequency components that can be reduced by relaxation on the fine grid! This leads us to combine the ideas of relaxation and coarse-grid correction:

Two-Grid Iteration

- Pre-smoothing: Compute $\bar{u}_h$ by applying $\nu_1 \geq 0$ steps of a relaxation method to $\widetilde{u}_h$.
- Coarse-grid correction: As above, using $\bar{u}_h$ to give $\bar{u}_h^{\text{new}}$.
- Post-smoothing: Compute $\widetilde{u}_h^{\text{new}}$ by applying $\nu_2 \geq 0$ steps of the relaxation method to $\bar{u}_h^{\text{new}}$.

It is only a short step from the above two-grid method to a multigrid method. Instead of solving the coarse-grid defect equation (19.6.8) exactly, we can get an approximate solution of it by introducing an even coarser grid and using the two-grid iteration method. If the convergence factor of the two-grid method is small enough, we will need only a few steps of this iteration to get a good enough approximate solution. We denote the number of such iterations by γ. Obviously we can apply this idea recursively down to some coarsest grid. There the solution is found easily, for example by direct matrix inversion or by iterating the relaxation scheme to convergence.

One iteration of a multigrid method, from finest grid to coarser grids and back to finest grid again, is called a *cycle*. The exact structure of a cycle depends on the value of γ, the number of two-grid iterations at each intermediate stage. The case $\gamma = 1$ is called a V-cycle, while $\gamma = 2$ is called a W-cycle (see Figure 19.6.1). These are the most important cases in practice.

Note that once more than two grids are involved, the pre-smoothing steps after the first one on the finest grid need an initial approximation for the error v. This should be taken to be zero.

Smoothing, Restriction, and Prolongation Operators

The most popular smoothing method, and the one you should try first, is Gauss-Seidel, since it usually leads to a good convergence rate. If we order the mesh points from 1 to N, then the Gauss-Seidel scheme is

$$u_i = -\Big(\sum_{\substack{j=1 \\ j\neq i}}^{N} L_{ij}u_j - f_i\Big)\frac{1}{L_{ii}} \qquad i = 1,\ldots,N \tag{19.6.12}$$

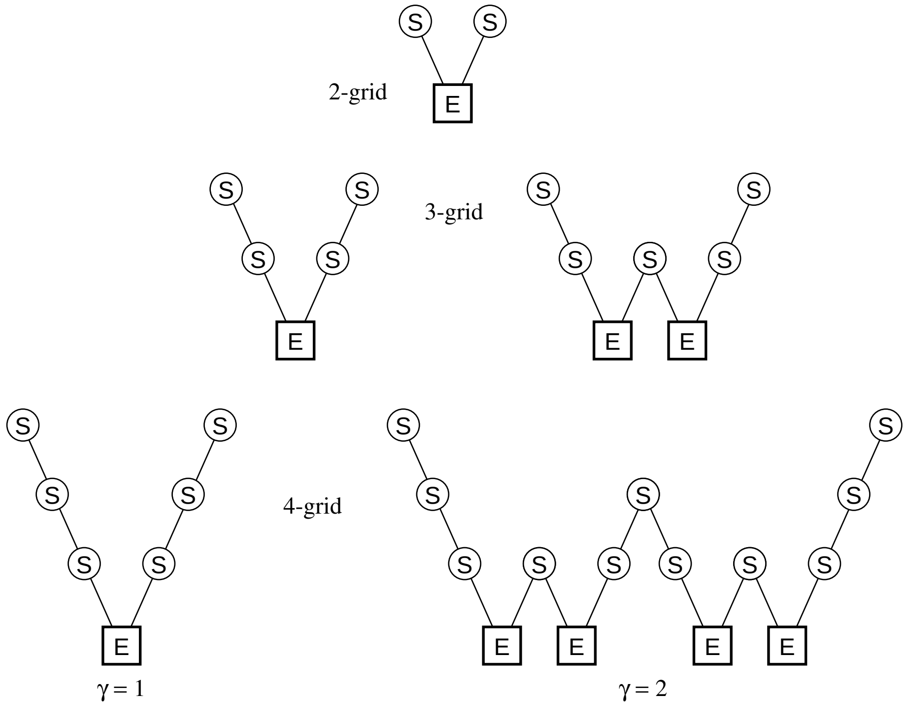

Figure 19.6.1. Structure of multigrid cycles. S denotes smoothing, while E denotes exact solution on the coarsest grid. Each descending line \ denotes restriction ($\mathcal{R}$) and each ascending line / denotes prolongation ($\mathcal{P}$). The finest grid is at the top level of each diagram. For the V-cycles ($\gamma = 1$) the E step is replaced by one 2-grid iteration each time the number of grid levels is increased by one. For the W-cycles ($\gamma = 2$), each E step gets replaced by two 2-grid iterations.

where new values of u are used on the right-hand side as they become available. The exact form of the Gauss-Seidel method depends on the ordering chosen for the mesh points. For typical second-order elliptic equations like our model problem equation (19.0.3), as differenced in equation (19.0.8), it is usually best to use red-black ordering, making one pass through the mesh updating the "even" points (like the red squares of a checkerboard) and another pass updating the "odd" points (the black squares). When quantities are more strongly coupled along one dimension than another, one should relax a whole line along that dimension simultaneously. Line relaxation for nearest-neighbor coupling involves solving a tridiagonal system, and so is still efficient. Relaxing odd and even lines on successive passes is called zebra relaxation and is usually preferred over simple line relaxation.

Note that SOR should *not* be used as a smoothing operator. The overrelaxation destroys the high-frequency smoothing that is so crucial for the multigrid method.

A succint notation for the prolongation and restriction operators is to give their *symbol*. The symbol of $\mathcal{P}$ is found by considering v_H to be 1 at some mesh point (x, y), zero elsewhere, and then asking for the values of $\mathcal{P}v_H$. The most popular prolongation operator is simple bilinear interpolation. It gives nonzero values at the 9 points (x, y), $(x + h, y), \ldots, (x - h, y - h)$, where the values are $1, \frac{1}{2}, \ldots, \frac{1}{4}$.

Its symbol is therefore

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{19.6.13}$$

The symbol of $\mathcal{R}$ is defined by considering v_h to be defined everywhere on the fine grid, and then asking what is $\mathcal{R}v_h$ at (x, y) as a linear combination of these values. The simplest possible choice for $\mathcal{R}$ is *straight injection*, which means simply filling each coarse-grid point with the value from the corresponding fine-grid point. Its symbol is "[1]." However, difficulties can arise in practice with this choice. It turns out that a safe choice for $\mathcal{R}$ is to make it the adjoint operator to $\mathcal{P}$. To define the adjoint, define the scalar product of two grid functions u_h and v_h for mesh size h as

$$\langle u_h|v_h\rangle_h \equiv h^2 \sum_{x,y} u_h(x, y)v_h(x, y) \tag{19.6.14}$$

Then the adjoint of $\mathcal{P}$, denoted $\mathcal{P}^\dagger$, is defined by

$$\langle u_H|\mathcal{P}^\dagger v_h\rangle_H = \langle \mathcal{P}u_H|v_h\rangle_h \tag{19.6.15}$$

Now take $\mathcal{P}$ to be bilinear interpolation, and choose $u_H = 1$ at (x, y), zero elsewhere. Set $\mathcal{P}^\dagger = \mathcal{R}$ in (19.6.15) and $H = 2h$. You will find that

$$(\mathcal{R}v_h)_{(x,y)} = \tfrac{1}{4}v_h(x, y) + \tfrac{1}{8}v_h(x + h, y) + \tfrac{1}{16}v_h(x + h, y + h) + \cdots \tag{19.6.16}$$

so that the symbol of $\mathcal{R}$ is

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{19.6.17}$$

Note the simple rule: The symbol of $\mathcal{R}$ is $\frac{1}{4}$ the transpose of the matrix defining the symbol of $\mathcal{P}$, equation (19.6.13). This rule is general whenever $\mathcal{R} = \mathcal{P}^\dagger$ and $H = 2h$.

The particular choice of $\mathcal{R}$ in (19.6.17) is called *full weighting*. Another popular choice for $\mathcal{R}$ is *half weighting*, "halfway" between full weighting and straight injection. Its symbol is

$$\begin{bmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{2} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{bmatrix} \tag{19.6.18}$$

A similar notation can be used to describe the difference operator $\mathcal{L}_h$. For example, the standard differencing of the model problem, equation (19.0.6), is represented by the *five-point difference star*

$$\mathcal{L}_h = \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{19.6.19}$$

If you are confronted with a new problem and you are not sure what $\mathcal{P}$ and $\mathcal{R}$ choices are likely to work well, here is a safe rule: Suppose m_p is the order of the interpolation $\mathcal{P}$ (i.e., it interpolates polynomials of degree $m_p - 1$ exactly). Suppose m_r is the order of $\mathcal{R}$, and that $\mathcal{R}$ is the adjoint of some $\mathcal{P}$ (not necessarily the $\mathcal{P}$ you intend to use). Then if m is the order of the differential operator $\mathcal{L}_h$, you should satisfy the inequality $m_p + m_r > m$. For example, bilinear interpolation and its adjoint, full weighting, for Poisson's equation satisfy $m_p + m_r = 4 > m = 2$.

Of course the $\mathcal{P}$ and $\mathcal{R}$ operators should enforce the boundary conditions for your problem. The easiest way to do this is to rewrite the difference equation to have homogeneous boundary conditions by modifying the source term if necessary (cf. §19.4). Enforcing homogeneous boundary conditions simply requires the $\mathcal{P}$ operator to produce zeros at the appropriate boundary points. The corresponding $\mathcal{R}$ is then found by $\mathcal{R} = \mathcal{P}^\dagger$.

Full Multigrid Algorithm

So far we have described multigrid as an iterative scheme, where one starts with some initial guess on the finest grid and carries out enough cycles (V-cycles, W-cycles,...) to achieve convergence. This is the simplest way to use multigrid: Simply apply enough cycles until some appropriate convergence criterion is met. However, efficiency can be improved by using the *Full Multigrid Algorithm* (FMG), also known as *nested iteration*.

Instead of starting with an arbitrary approximation on the finest grid (e.g., $u_h = 0$), the first approximation is obtained by interpolating from a coarse-grid solution:

$$u_h = \mathcal{P} u_H \tag{19.6.20}$$

The coarse-grid solution itself is found by a similar FMG process from even coarser grids. At the coarsest level, you start with the exact solution. Rather than proceed as in Figure 19.6.1, then, FMG gets to its solution by a series of increasingly tall "N's," each taller one probing a finer grid (see Figure 19.6.2).

Note that $\mathcal{P}$ in (19.6.20) need not be the same $\mathcal{P}$ used in the multigrid cycles. It should be at least of the same order as the discretization $\mathcal{L}_h$, but sometimes a higher-order operator leads to greater efficiency.

It turns out that you usually need one or at most two multigrid cycles at each level before proceeding down to the next finer grid. While there is theoretical guidance on the required number of cycles (e.g., [2]), you can easily determine it empirically. Fix the finest level and study the solution values as you increase the number of cycles per level. The asymptotic value of the solution is the exact solution of the difference equations. The difference between this exact solution and the solution for a small number of cycles is the iteration error. Now fix the number of cycles to be large, and vary the number of levels, i.e., the smallest value of h used. In this way you can estimate the truncation error for a given h. In your final production code, there is no point in using more cycles than you need to get the iteration error down to the size of the truncation error.

The simple multigrid iteration (cycle) needs the right-hand side f only at the finest level. FMG needs f at all levels. If the boundary conditions are homogeneous,

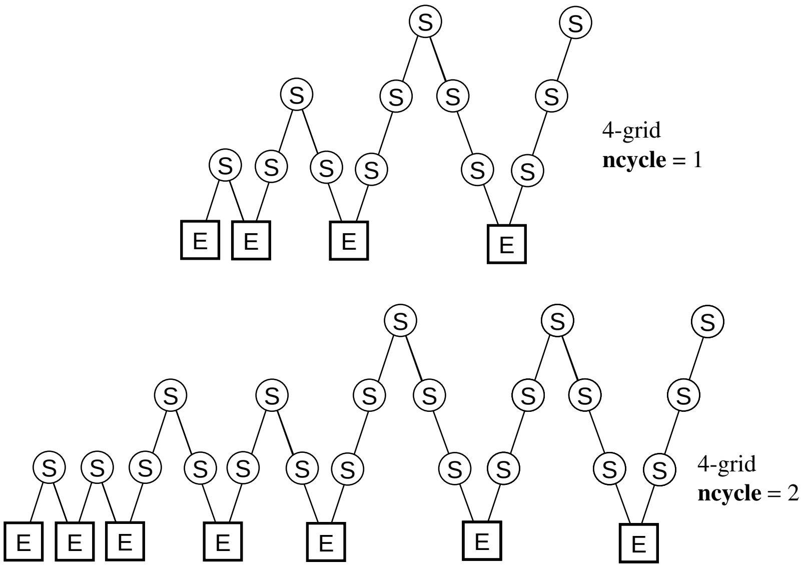

Figure 19.6.2. Structure of cycles for the full multigrid (FMG) method. This method starts on the coarsest grid, interpolates, and then refines (by "V's"), the solution onto grids of increasing fineness.

you can use $f_H = \mathcal{R} f_h$. This prescription is not always safe for inhomogeneous boundary conditions. In that case it is better to discretize f on each coarse grid.

Note that the FMG algorithm produces the solution on all levels. It can therefore be combined with techniques like Richardson extrapolation.

We now give a routine `mglin` that implements the Full Multigrid Algorithm for a linear equation, the model problem (19.0.6). It uses red-black Gauss-Seidel as the smoothing operator, bilinear interpolation for $\mathcal{P}$, and half-weighting for $\mathcal{R}$. To change the routine to handle another linear problem, all you need do is modify the functions `relax`, `resid`, and `slvsml` appropriately. A feature of the routine is the dynamical allocation of storage for variables defined on the various grids.

```
#include "nrutil.h"
#define NPRE 1                     Number of relaxation sweeps before ...
#define NPOST 1                    ...and after the coarse-grid correction is com-
#define NGMAX 15                       puted.

void mglin(double **u, int n, int ncycle)
```
Full Multigrid Algorithm for solution of linear elliptic equation, here the model problem (19.0.6). On input `u[1..n][1..n]` contains the right-hand side ρ, while on output it returns the solution. The dimension `n` must be of the form $2^j + 1$ for some integer j. (j is actually the number of grid levels used in the solution, called `ng` below.) `ncycle` is the number of V-cycles to be used at each level.
```
{
    void addint(double **uf, double **uc, double **res, int nf);
    void copy(double **aout, double **ain, int n);
    void fill0(double **u, int n);
    void interp(double **uf, double **uc, int nf);
    void relax(double **u, double **rhs, int n);
    void resid(double **res, double **u, double **rhs, int n);
    void rstrct(double **uc, double **uf, int nc);
    void slvsml(double **u, double **rhs);
```

```
    unsigned int j,jcycle,jj,jpost,jpre,nf,ng=0,ngrid,nn;
    double **ires[NGMAX+1],**irho[NGMAX+1],**irhs[NGMAX+1],**iu[NGMAX+1];

    nn=n;
    while (nn >>= 1) ng++;
    if (n != 1+(1L << ng)) nrerror("n-1 must be a power of 2 in mglin.");
    if (ng > NGMAX) nrerror("increase NGMAX in mglin.");
    nn=n/2+1;
    ngrid=ng-1;
    irho[ngrid]=dmatrix(1,nn,1,nn);          Allocate storage for r.h.s. on grid ng − 1,
    rstrct(irho[ngrid],u,nn);                and fill it by restricting from the fine grid.
    while (nn > 3) {                         Similarly allocate storage and fill r.h.s. on all
        nn=nn/2+1;                               coarse grids.
        irho[--ngrid]=dmatrix(1,nn,1,nn);
        rstrct(irho[ngrid],irho[ngrid+1],nn);
    }
    nn=3;
    iu[1]=dmatrix(1,nn,1,nn);
    irhs[1]=dmatrix(1,nn,1,nn);
    slvsml(iu[1],irho[1]);                   Initial solution on coarsest grid.
    free_dmatrix(irho[1],1,nn,1,nn);
    ngrid=ng;
    for (j=2;j<=ngrid;j++) {                 Nested iteration loop.
        nn=2*nn-1;
        iu[j]=dmatrix(1,nn,1,nn);
        irhs[j]=dmatrix(1,nn,1,nn);
        ires[j]=dmatrix(1,nn,1,nn);
        interp(iu[j],iu[j-1],nn);
        Interpolate from coarse grid to next finer grid.
        copy(irhs[j],(j != ngrid ? irho[j] : u),nn);       Set up r.h.s.
        for (jcycle=1;jcycle<=ncycle;jcycle++) {           V-cycle loop.
            nf=nn;
            for (jj=j;jj>=2;jj--) {                        Downward stoke of the V.
            for (jpre=1;jpre<=NPRE;jpre++)                 Pre-smoothing.
                relax(iu[jj],irhs[jj],nf);
            resid(ires[jj],iu[jj],irhs[jj],nf);
            nf=nf/2+1;
            rstrct(irhs[jj-1],ires[jj],nf);
            Restriction of the residual is the next r.h.s.
            fill0(iu[jj-1],nf);                            Zero for initial guess in next
            }                                                  relaxation.
            slvsml(iu[1],irhs[1]);                         Bottom of V: solve on coars-
            nf=3;                                              est grid.
            for (jj=2;jj<=j;jj++) {                        Upward stroke of V.
            nf=2*nf-1;
            addint(iu[jj],iu[jj-1],ires[jj],nf);
            Use res for temporary storage inside addint.
            for (jpost=1;jpost<=NPOST;jpost++)             Post-smoothing.
                relax(iu[jj],irhs[jj],nf);
            }
        }
    }
    copy(u,iu[ngrid],n);                                   Return solution in u.
    for (nn=n,j=ng;j>=2;j--,nn=nn/2+1) {
        free_dmatrix(ires[j],1,nn,1,nn);
        free_dmatrix(irhs[j],1,nn,1,nn);
        free_dmatrix(iu[j],1,nn,1,nn);
        if (j != ng) free_dmatrix(irho[j],1,nn,1,nn);
    }
    free_dmatrix(irhs[1],1,3,1,3);
    free_dmatrix(iu[1],1,3,1,3);
}
```

```
void rstrct(double **uc, double **uf, int nc)
```

Half-weighting restriction. nc is the coarse-grid dimension. The fine-grid solution is input in uf[1..2*nc-1][1..2*nc-1], the coarse-grid solution is returned in uc[1..nc][1..nc].

```
{
    int ic,iif,jc,jf,ncc=2*nc-1;

    for (jf=3,jc=2;jc<nc;jc++,jf+=2) {         Interior points.
        for (iif=3,ic=2;ic<nc;ic++,iif+=2) {
            uc[ic][jc]=0.5*uf[iif][jf]+0.125*(uf[iif+1][jf]+uf[iif-1][jf]
                    +uf[iif][jf+1]+uf[iif][jf-1]);
        }
    }
    for (jc=1,ic=1;ic<=nc;ic++,jc+=2) {        Boundary points.
        uc[ic][1]=uf[jc][1];
        uc[ic][nc]=uf[jc][ncc];
    }
    for (jc=1,ic=1;ic<=nc;ic++,jc+=2) {
        uc[1][ic]=uf[1][jc];
        uc[nc][ic]=uf[ncc][jc];
    }
}
```

```
void interp(double **uf, double **uc, int nf)
```

Coarse-to-fine prolongation by bilinear interpolation. nf is the fine-grid dimension. The coarse-grid solution is input as uc[1..nc][1..nc], where $nc = nf/2 + 1$. The fine-grid solution is returned in uf[1..nf][1..nf].

```
{
    int ic,iif,jc,jf,nc;
    nc=nf/2+1;
    for (jc=1,jf=1;jc<=nc;jc++,jf+=2)          Do elements that are copies.
        for (ic=1;ic<=nc;ic++) uf[2*ic-1][jf]=uc[ic][jc];
    for (jf=1;jf<=nf;jf+=2)                    Do odd-numbered columns, interpolat-
        for (iif=2;iif<nf;iif+=2)                  ing vertically.
            uf[iif][jf]=0.5*(uf[iif+1][jf]+uf[iif-1][jf]);

    for (jf=2;jf<nf;jf+=2)                     Do even-numbered columns, interpolat-
        for (iif=1;iif <= nf;iif++)                ing horizontally.
            uf[iif][jf]=0.5*(uf[iif][jf+1]+uf[iif][jf-1]);
}
```

```
void addint(double **uf, double **uc, double **res, int nf)
```

Does coarse-to-fine interpolation and adds result to uf. nf is the fine-grid dimension. The coarse-grid solution is input as uc[1..nc][1..nc], where $nc = nf/2 + 1$. The fine-grid solution is returned in uf[1..nf][1..nf]. res[1..nf][1..nf] is used for temporary storage.

```
{
    void interp(double **uf, double **uc, int nf);
    int i,j;

    interp(res,uc,nf);
    for (j=1;j<=nf;j++)
        for (i=1;i<=nf;i++)
            uf[i][j] += res[i][j];
}
```

```
void slvsml(double **u, double **rhs)
```

Solution of the model problem on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in rhs[1..3][1..3] and the solution is returned in u[1..3][1..3].

```
{
    void fill0(double **u, int n);
    double h=0.5;

    fill0(u,3);
```

```
    u[2][2] = -h*h*rhs[2][2]/4.0;
}
```

```
void relax(double **u, double **rhs, int n)
```
Red-black Gauss-Seidel relaxation for model problem. Updates the current value of the solution u[1..n][1..n], using the right-hand side function rhs[1..n][1..n].
```
{
    int i,ipass,isw,j,jsw=1;
    double h,h2;

    h=1.0/(n-1);
    h2=h*h;
    for (ipass=1;ipass<=2;ipass++,jsw=3-jsw) {        Red and black sweeps.
        isw=jsw;
        for (j=2;j<n;j++,isw=3-isw)
            for (i=isw+1;i<n;i+=2)                    Gauss-Seidel formula.
                u[i][j]=0.25*(u[i+1][j]+u[i-1][j]+u[i][j+1]
                    +u[i][j-1]-h2*rhs[i][j]);
    }
}
```

```
void resid(double **res, double **u, double **rhs, int n)
```
Returns *minus* the residual for the model problem. Input quantities are u[1..n][1..n] and rhs[1..n][1..n], while res[1..n][1..n] is returned.
```
{
    int i,j;
    double h,h2i;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=2;j<n;j++)              Interior points.
        for (i=2;i<n;i++)
            res[i][j] = -h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+rhs[i][j];
    for (i=1;i<=n;i++)             Boundary points.
        res[i][1]=res[i][n]=res[1][i]=res[n][i]=0.0;
}
```

```
void copy(double **aout, double **ain, int n)
```
Copies ain[1..n][1..n] to aout[1..n][1..n].
```
{
    int i,j;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            aout[j][i]=ain[j][i];

}
```

```
void fill0(double **u, int n)
```
Fills u[1..n][1..n] with zeros.
```
{
    int i,j;
    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            u[i][j]=0.0;
}
```

The routine mglin is written for clarity, not maximum efficiency, so that it is easy to modify. Several simple changes will speed up the execution time:

- The defect d_h vanishes identically at all black mesh points after a red-black Gauss-Seidel step. Thus $d_H = \mathcal{R}d_h$ for half-weighting reduces to simply copying half the defect from the fine grid to the corresponding coarse-grid point. The calls to `resid` followed by `rstrct` in the first part of the V-cycle can be replaced by a routine that loops only over the coarse grid, filling it with half the defect.
- Similarly, the quantity $\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \mathcal{P}\widetilde{v}_H$ need not be computed at red mesh points, since they will immediately be redefined in the subsequent Gauss-Seidel sweep. This means that `addint` need only loop over black points.
- You can speed up `relax` in several ways. First, you can have a special form when the initial guess is zero, and omit the routine `fill0`. Next, you can store $h^2 f_h$ on the various grids and save a multiplication. Finally, it is possible to save an addition in the Gauss-Seidel formula by rewriting it with intermediate variables.
- On typical problems, `mglin` with `ncycle` $= 1$ will return a solution with the iteration error bigger than the truncation error for the given size of h. To knock the error down to the size of the truncation error, you have to set `ncycle` $= 2$ or, more cheaply, `npre` $= 2$. A more efficient way turns out to be to use a higher-order $\mathcal{P}$ in (19.6.20) than the linear interpolation used in the V-cycle.

Implementing all the above features typically gives up to a factor of two improvement in execution time and is certainly worthwhile in a production code.

Nonlinear Multigrid: The FAS Algorithm

Now turn to solving a nonlinear elliptic equation, which we write symbolically as

$$\mathcal{L}(u) = 0 \tag{19.6.21}$$

Any explicit source term has been moved to the left-hand side. Suppose equation (19.6.21) is suitably discretized:

$$\mathcal{L}_h(u_h) = 0 \tag{19.6.22}$$

We will see below that in the multigrid algorithm we will have to consider equations where a nonzero right-hand side is generated during the course of the solution:

$$\mathcal{L}_h(u_h) = f_h \tag{19.6.23}$$

One way of solving nonlinear problems with multigrid is to use Newton's method, which produces linear equations for the correction term at each iteration. We can then use linear multigrid to solve these equations. A great strength of the multigrid idea, however, is that it can be applied *directly* to nonlinear problems. All we need is a suitable *nonlinear* relaxation method to smooth the errors, plus a procedure for approximating corrections on coarser grids. This direct approach is Brandt's Full Approximation Storage Algorithm (FAS). No nonlinear equations need be solved, except perhaps on the coarsest grid.

To develop the nonlinear algorithm, suppose we have a relaxation procedure that can smooth the residual vector as we did in the linear case. Then we can seek a smooth correction v_h to solve (19.6.23):

$$\mathcal{L}_h(\widetilde{u}_h + v_h) = f_h \tag{19.6.24}$$

To find v_h, note that

$$\begin{aligned}\mathcal{L}_h(\widetilde{u}_h + v_h) - \mathcal{L}_h(\widetilde{u}_h) &= f_h - \mathcal{L}_h(\widetilde{u}_h)\\ &= -d_h\end{aligned} \tag{19.6.25}$$

The right-hand side is smooth after a few nonlinear relaxation sweeps. Thus we can transfer the left-hand side to a coarse grid:

$$\mathcal{L}_H(u_H) - \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) = -\mathcal{R}d_h \tag{19.6.26}$$

that is, we solve

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}d_h \tag{19.6.27}$$

on the coarse grid. (This is how nonzero right-hand sides appear.) Suppose the approximate solution is $\widetilde{u}_H$. Then the coarse-grid correction is

$$\widetilde{v}_H = \widetilde{u}_H - \mathcal{R}\widetilde{u}_h \tag{19.6.28}$$

and

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h) \tag{19.6.29}$$

Note that $\mathcal{PR} \neq 1$ in general, so $\widetilde{u}_h^{\text{new}} \neq \mathcal{P}\widetilde{u}_H$. This is a key point: In equation (19.6.29) the interpolation error comes only from the correction, not from the full solution $\widetilde{u}_H$.

Equation (19.6.27) shows that one is solving for the full approximation u_H, not just the error as in the linear algorithm. This is the origin of the name FAS.

The FAS multigrid algorithm thus looks very similar to the linear multigrid algorithm. The only differences are that both the defect d_h and the relaxed approximation u_h have to be restricted to the coarse grid, where now it is equation (19.6.27) that is solved by recursive invocation of the algorithm. However, instead of implementing the algorithm this way, we will first describe the so-called *dual viewpoint*, which leads to a powerful alternative way of looking at the multigrid idea.

The dual viewpoint considers the *local truncation error*, defined as

$$\tau \equiv \mathcal{L}_h(u) - f_h \tag{19.6.30}$$

where u is the exact solution of the oiginal continuum equation. If we rewrite this as

$$\mathcal{L}_h(u) = f_h + \tau \tag{19.6.31}$$

we see that τ can be regarded as the correction to f_h so that the solution of the fine-grid equation will be the exact solution u.

Now consider the *relative truncation error* τ_h, which is defined on the H-grid relative to the h-grid:

$$\tau_h \equiv \mathcal{L}_H(\mathcal{R}u_h) - \mathcal{R}\mathcal{L}_h(u_h) \tag{19.6.32}$$

Since $\mathcal{L}_h(u_h) = f_h$, this can be rewritten as

$$\mathcal{L}_H(u_H) = f_H + \tau_h \tag{19.6.33}$$

In other words, we can think of τ_h as the correction to f_H that makes the solution of the coarse-grid equation equal to the fine-grid solution. Of course we cannot compute τ_h, but we do have an approximation to it from using $\widetilde{u}_h$ in equation (19.6.32):

$$\tau_h \simeq \widetilde{\tau}_h \equiv \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}\mathcal{L}_h(\widetilde{u}_h) \tag{19.6.34}$$

Replacing τ_h by $\widetilde{\tau}_h$ in equation (19.6.33) gives

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}d_h \tag{19.6.35}$$

which is just the coarse-grid equation (19.6.27)!

Thus we see that there are two complementary viewpoints for the relation between coarse and fine grids:

- Coarse grids are used to accelerate the convergence of the smooth components of the fine-grid residuals.

- Fine grids are used to compute correction terms to the coarse-grid equations, yielding fine-grid accuracy on the coarse grids.

One benefit of this new viewpoint is that it allows us to derive a natural stopping criterion for a multigrid iteration. Normally the criterion would be

$$\|d_h\| \le \epsilon \qquad (19.6.36)$$

and the question is how to choose ϵ. There is clearly no benefit in iterating beyond the point when the remaining error is dominated by the local truncation error τ. The computable quantity is $\widetilde{\tau}_h$. What is the relation between τ and $\widetilde{\tau}_h$? For the typical case of a second-order accurate differencing scheme,

$$\tau = \mathcal{L}_h(u) - \mathcal{L}_h(u_h) = h^2\tau_2(x,y) + \cdots \qquad (19.6.37)$$

Assume the solution satisfies $u_h = u + h^2u_2(x,y) + \cdots$. Then, assuming $\mathcal{R}$ is of high enough order that we can neglect its effect, equation (19.6.32) gives

$$\begin{aligned}\tau_h &\simeq \mathcal{L}_H(u + h^2u_2) - \mathcal{L}_h(u + h^2u_2)\\ &= \mathcal{L}_H(u) - \mathcal{L}_h(u) + h^2[\mathcal{L}'_H(u_2) - \mathcal{L}'_h(u_2)] + \cdots\\ &= (H^2 - h^2)\tau_2 + O(h^4)\end{aligned} \qquad (19.6.38)$$

For the usual case of $H = 2h$ we therefore have

$$\tau \simeq \tfrac{1}{3}\tau_h \simeq \tfrac{1}{3}\widetilde{\tau}_h \qquad (19.6.39)$$

The stopping criterion is thus equation (19.6.36) with

$$\epsilon = \alpha\|\widetilde{\tau}_h\|, \qquad \alpha \sim \tfrac{1}{3} \qquad (19.6.40)$$

We have one remaining task before implementing our nonlinear multigrid algorithm: choosing a nonlinear relaxation scheme. Once again, your first choice should probably be the nonlinear Gauss-Seidel scheme. If the discretized equation (19.6.23) is written with some choice of ordering as

$$L_i(u_1,\ldots,u_N) = f_i, \qquad i = 1,\ldots,N \qquad (19.6.41)$$

then the nonlinear Gauss-Seidel schemes solves

$$L_i(u_1,\ldots,u_{i-1},u_i^{\text{new}},u_{i+1},\ldots,u_N) = f_i \qquad (19.6.42)$$

for u_i^{new}. As usual new u's replace old u's as soon as they have been computed. Often equation (19.6.42) is linear in u_i^{new}, since the nonlinear terms are discretized by means of its neighbors. If this is not the case, we replace equation (19.6.42) by one step of a Newton iteration:

$$u_i^{\text{new}} = u_i^{\text{old}} - \frac{L_i(u_i^{\text{old}}) - f_i}{\partial L_i(u_i^{\text{old}})/\partial u_i} \qquad (19.6.43)$$

For example, consider the simple nonlinear equation

$$\nabla^2 u + u^2 = \rho \qquad (19.6.44)$$

In two-dimensional notation, we have

$$\mathcal{L}(u_{i,j}) = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j})/h^2 + u_{i,j}^2 - \rho_{i,j} = 0 \qquad (19.6.45)$$

Since

$$\frac{\partial\mathcal{L}}{\partial u_{i,j}} = -4/h^2 + 2u_{i,j} \qquad (19.6.46)$$

the Newton Gauss-Seidel iteration is

$$u_{i,j}^{\text{new}} = u_{i,j} - \frac{\mathcal{L}(u_{i,j})}{-4/h^2 + 2u_{i,j}} \qquad (19.6.47)$$

Here is a routine `mgfas` that solves equation (19.6.44) using the Full Multigrid Algorithm and the FAS scheme. Restriction and prolongation are done as in `mglin`. We have included the convergence test based on equation (19.6.40). A successful multigrid solution of a problem should aim to satisfy this condition with the maximum number of V-cycles, `maxcyc`, equal to 1 or 2. The routine `mgfas` uses the same functions `copy`, `interp`, and `rstrct` as `mglin`.

```
#include "nrutil.h"
#define NPRE 1                      Number of relaxation sweeps before ...
#define NPOST 1                     ...and after the coarse-grid correction is computed.
#define ALPHA 0.33                  Relates the estimated truncation error to the norm
#define NGMAX 15                        of the residual.

void mgfas(double **u, int n, int maxcyc)
```
Full Multigrid Algorithm for FAS solution of nonlinear elliptic equation, here equation (19.6.44). On input `u[1..n][1..n]` contains the right-hand side ρ, while on output it returns the solution. The dimension `n` must be of the form $2^j + 1$ for some integer j. (j is actually the number of grid levels used in the solution, called `ng` below.) `maxcyc` is the maximum number of V-cycles to be used at each level.
```
{
    double anorm2(double **a, int n);
    void copy(double **aout, double **ain, int n);
    void interp(double **uf, double **uc, int nf);
    void lop(double **out, double **u, int n);
    void matadd(double **a, double **b, double **c, int n);
    void matsub(double **a, double **b, double **c, int n);
    void relax2(double **u, double **rhs, int n);
    void rstrct(double **uc, double **uf, int nc);
    void slvsm2(double **u, double **rhs);
    unsigned int j,jcycle,jj,jm1,jpost,jpre,nf,ng=0,ngrid,nn;
    double **irho[NGMAX+1],**irhs[NGMAX+1],**itau[NGMAX+1],
        **itemp[NGMAX+1],**iu[NGMAX+1];
    double res,trerr;

    nn=n;
    while (nn >>= 1) ng++;
    if (n != 1+(1L << ng)) nrerror("n-1 must be a power of 2 in mgfas.");
    if (ng > NGMAX) nrerror("increase NGMAX in mglin.");
    nn=n/2+1;
    ngrid=ng-1;
    irho[ngrid]=dmatrix(1,nn,1,nn);          Allocate storage for r.h.s. on grid ng − 1,
    rstrct(irho[ngrid],u,nn);                and fill it by restricting from the fine grid.
    while (nn > 3) {                         Similarly allocate storage and fill r.h.s. on all
        nn=nn/2+1;                               coarse grids.
        irho[--ngrid]=dmatrix(1,nn,1,nn);
        rstrct(irho[ngrid],irho[ngrid+1],nn);
    }
    nn=3;
    iu[1]=dmatrix(1,nn,1,nn);
    irhs[1]=dmatrix(1,nn,1,nn);
    itau[1]=dmatrix(1,nn,1,nn);
    itemp[1]=dmatrix(1,nn,1,nn);
    slvsm2(iu[1],irho[1]);                   Initial solution on coarsest grid.
    free_dmatrix(irho[1],1,nn,1,nn);
    ngrid=ng;
    for (j=2;j<=ngrid;j++) {                 Nested iteration loop.
        nn=2*nn-1;
        iu[j]=dmatrix(1,nn,1,nn);
        irhs[j]=dmatrix(1,nn,1,nn);
        itau[j]=dmatrix(1,nn,1,nn);
        itemp[j]=dmatrix(1,nn,1,nn);
        interp(iu[j],iu[j-1],nn);
        Interpolate from coarse grid to next finer grid.
        copy(irhs[j],(j != ngrid ? irho[j] : u),nn);     Set up r.h.s.
        for (jcycle=1;jcycle<=maxcyc;jcycle++) {         V-cycle loop.
        nf=nn;
            for (jj=j;jj>=2;jj--) {                      Downward stoke of the V.
                for (jpre=1;jpre<=NPRE;jpre++)           Pre-smoothing.
                    relax2(iu[jj],irhs[jj],nf);
                lop(itemp[jj],iu[jj],nf);                L_h(ũ_h).
                nf=nf/2+1;
```

```
            jm1=jj-1;
            rstrct(itemp[jm1],itemp[jj],nf);          $\mathcal{R}\mathcal{L}_h(\widetilde{u}_h)$.
            rstrct(iu[jm1],iu[jj],nf);                $\mathcal{R}\widetilde{u}_h$.
            lop(itau[jm1],iu[jm1],nf);
            $\mathcal{L}_H(\mathcal{R}\widetilde{u}_h)$ stored temporarily in $\widetilde{\tau}_h$.
            matsub(itau[jm1],itemp[jm1],itau[jm1],nf);        Form $\widetilde{\tau}_h$.
            if (jj == j)
                trerr=ALPHA*anorm2(itau[jm1],nf);     Estimate truncation error $\tau$.
            rstrct(irhs[jm1],irhs[jj],nf);            $f_H$.
            matadd(irhs[jm1],itau[jm1],irhs[jm1],nf);    $f_H + \widetilde{\tau}_h$.
        }
        slvsm2(iu[1],irhs[1]);                        Bottom of V: Solve on coars-
        nf=3;                                              est grid.
        for (jj=2;jj<=j;jj++) {                       Upward stroke of V.
        jm1=jj-1;
        rstrct(itemp[jm1],iu[jj],nf);                 $\mathcal{R}\widetilde{u}_h$.
        matsub(iu[jm1],itemp[jm1],itemp[jm1],nf);     $\widetilde{u}_H - \mathcal{R}\widetilde{u}_h$.
        nf=2*nf-1;
        interp(itau[jj],itemp[jm1],nf);               $\mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h)$ stored in $\widetilde{\tau}_h$.
        matadd(iu[jj],itau[jj],iu[jj],nf);            Form $\widetilde{u}_h^{\text{new}}$.
        for (jpost=1;jpost<=NPOST;jpost++)            Post-smoothing.
            relax2(iu[jj],irhs[jj],nf);
        }
        lop(itemp[j],iu[j],nf);                       Form residual $\|d_h\|$.
        matsub(itemp[j],irhs[j],itemp[j],nf);
        res=anorm2(itemp[j],nf);
        if (res < trerr) break;                       No more V-cycles needed if
    }                                                      residual small enough.
}
copy(u,iu[ngrid],n);                                  Return solution in u.
for (nn=n,j=ng;j>=1;j--,nn=nn/2+1) {
    free_dmatrix(itemp[j],1,nn,1,nn);
    free_dmatrix(itau[j],1,nn,1,nn);
    free_dmatrix(irhs[j],1,nn,1,nn);
    free_dmatrix(iu[j],1,nn,1,nn);
    if (j != ng && j != 1) free_dmatrix(irho[j],1,nn,1,nn);
}
}
```

```
void relax2(double **u, double **rhs, int n)
```

Red-black Gauss-Seidel relaxation for equation (19.6.44). The current value of the solution `u[1..n][1..n]` is updated, using the right-hand side function `rhs[1..n][1..n]`.

```
{
    int i,ipass,isw,j,jsw=1;
    double foh2,h,h2i,res;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    foh2 = -4.0*h2i;
    for (ipass=1;ipass<=2;ipass++,jsw=3-jsw) {        Red and black sweeps.
        isw=jsw;
        for (j=2;j<n;j++,isw=3-isw) {
            for (i=isw+1;i<n;i+=2) {
                res=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+u[i][j]*u[i][j]-rhs[i][j];
                u[i][j] -= res/(foh2+2.0*u[i][j]);    Newton Gauss-Seidel formula.
            }
        }
    }
}
```

```
#include <math.h>

void slvsm2(double **u, double **rhs)
```
Solution of equation (19.6.44) on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in rhs[1..3][1..3] and the solution is returned in u[1..3][1..3].
```
{
    void fill0(double **u, int n);
    double disc,fact,h=0.5;

    fill0(u,3);
    fact=2.0/(h*h);
    disc=sqrt(fact*fact+rhs[2][2]);
    u[2][2] = -rhs[2][2]/(fact+disc);
}

void lop(double **out, double **u, int n)
```
Given u[1..n][1..n], returns $\mathcal{L}_h(\widetilde{u}_h)$ for equation (19.6.44) in out[1..n][1..n].
```
{
    int i,j;
    double h,h2i;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=2;j<n;j++)                Interior points.
        for (i=2;i<n;i++)
            out[i][j]=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                4.0*u[i][j])+u[i][j]*u[i][j];
    for (i=1;i<=n;i++)               Boundary points.
        out[i][1]=out[i][n]=out[1][i]=out[n][i]=0.0;
}

void matadd(double **a, double **b, double **c, int n)
```
Adds a[1..n][1..n] to b[1..n][1..n] and returns result in c[1..n][1..n].
```
{
    int i,j;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            c[i][j]=a[i][j]+b[i][j];
}

void matsub(double **a, double **b, double **c, int n)
```
Subtracts b[1..n][1..n] from a[1..n][1..n] and returns result in c[1..n][1..n].
```
{
    int i,j;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            c[i][j]=a[i][j]-b[i][j];
}

#include <math.h>

double anorm2(double **a, int n)
```
Returns the Euclidean norm of the matrix a[1..n][1..n].
```
{
    int i,j;
    double sum=0.0;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            sum += a[i][j]*a[i][j];
    return sqrt(sum)/n;
}
```

CITED REFERENCES AND FURTHER READING:

Brandt, A. 1977, *Mathematics of Computation*, vol. 31, pp. 333–390. [1]

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag). [2]

Stuben, K., and Trottenberg, U. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag), pp. 1–176. [3]

Brandt, A. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag). [4]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill).

Briggs, W.L. 1987, *A Multigrid Tutorial* (Philadelphia: S.I.A.M.).

Jespersen, D. 1984, *Multrigrid Methods for Partial Differential Equations* (Washington: Mathematical Association of America).

McCormick, S.F. (ed.) 1988, *Multigrid Methods: Theory, Applications, and Supercomputing* (New York: Marcel Dekker).

Hackbusch, W., and Trottenberg, U. (eds.) 1991, *Multigrid Methods III* (Boston: Birkhauser).

Wesseling, P. 1992, *An Introduction to Multigrid Methods* (New York: Wiley).