

Chapter 9

Boundary Value problems

Selected Reading

Numerical Recipes, 2nd edition: Chapter 19

Briggs, William L. 1987, A Multigrid Tutorial, SIAM, Philadelphia.

In previous sections we have been concerned with time dependent *initial value problems* where we start with some assumed initial condition, calculate how this solution will change in time and then simply march through time updating as we go. We have considered both explicit and implicit schemes, but the basic point is that given a starting place it's relatively straightforward to get to the next step. The principal difficulty with time dependent schemes is stability and accuracy (they're not the same thing), i.e. how to march through time without blowing up and arriving at a future time with most of your intended physics intact.

Boundary value problems, are a somewhat different animal. In a boundary value problem we are trying to satisfy a steady state solution everywhere in space that agrees with our prescribed boundary conditions. For a flux conservative problem, the problem becomes finding the set of fluxes at all the nodes such that for every node, what comes in goes out. In general, boundary value problems will reduce, when discretized, to a large and sparse set of linear (and sometimes non-linear) equations. While stability is no longer a problem, efficiency in solving these equations becomes tantamount. The following sections will first develop some physical intuition into the types and sources of boundary value problems, then show how to discretize them and finally present a potpourri of solution techniques.

9.1 Where BVP's come from: basic physics

Many Boundary value problems arise from steady state solutions of transient problems (or from implicit schemes in transient problems). For example if we were considering the general transient heat flow problem for diffusion plus a source

term we could solve the dimensionless equation

$$\frac{\partial T}{\partial t} = \nabla^2 T - \rho(\mathbf{x}) \quad (9.1.1)$$

starting with some initial condition (plus appropriate boundary conditions) and march through time to steady state where $\partial T/\partial t = 0$. If your initial guess is far from steady state, though, this may take some time. Alternatively if you were only interested in the final steady state temperature distribution then you would like to solve immediately for the temperature by finding the temperature distribution that satisfies

$$\nabla^2 T = \rho(\mathbf{x}) \quad (9.1.2)$$

This is simply the temperature distribution where the divergence of the heat flux ($\nabla \cdot k \nabla T$) out of any volume exactly balances the source terms in that volume. Of course, for this solution to make any sense, both the boundary conditions and source terms must be independent of time. An analogous problem shows up in solving for fluid flow in a porous medium where the flux is governed by Darcy's law. Here we want the net flux out of any region to balance any source or sink terms so we have

$$\nabla \cdot k \nabla \mathcal{P} = S(\mathbf{x}) \quad (9.1.3)$$

where k is the hydraulic conductivity (or permeability) and \mathcal{P} is the fluid pressure gradient (not exactly but close enough for jazz).

Similarly if we were interested in the flow of a very viscous fluid, we could solve the full Navier Stokes equation

$$\frac{\partial \mathbf{V}}{\partial t} + \mathbf{V} \cdot \nabla \mathbf{V} = \nu \nabla^2 \mathbf{V} - \frac{1}{\rho} \nabla P + \mathbf{g} \quad (9.1.4)$$

by considering the accelerations (RHS terms) of each particle of fluid and updating. However, for problems where the dynamic viscosity ν is very large, these accelerations are negligible and the stress propagates almost instantaneously across the medium. Thus rather than being constrained to solve on the short viscous relaxation time we would like to assume that the system comes into balance instantaneously such that

$$0 = \nu \nabla^2 \mathbf{V} - \frac{1}{\rho} \nabla P + \mathbf{g} \quad (9.1.5)$$

which is again an elliptic (2nd order) boundary value problem for the vector velocity \mathbf{V} .

Incompressible fluid flow also demonstrates another source of boundary value problems that arise simply from definitions or changes of variables. In the case of a 2-D incompressible fluid, $\nabla \cdot \mathbf{V} = 0$ and therefore we can write the velocity solely in terms of a stream-function as $\mathbf{V} = \nabla \times \psi \mathbf{k}$. If we also define the *vorticity* as $\omega = \nabla \times \mathbf{V}$ then the relationship between stream-function and vorticity is just the Poisson problem

$$\nabla^2 \psi = -\omega \quad (9.1.6)$$

even for time dependent fluid-flow problems. Similar problems arise in electromagnetics problems.

A final example comes from the behaviour of elastic materials which is completely analogous to that of fluids, i.e. we can either track the transient stress changes which propagate at the speed of sound as seismic waves through the medium until they settle down, or we can just assume they will eventually settle out and solve for the static case

$$\nabla \cdot \boldsymbol{\tau} = \mathbf{f} \quad (9.1.7)$$

for the deviatoric stress tensor $\boldsymbol{\tau}$. If we can get to the steady state case faster than it would take to track the seismic waves then we have made a huge improvement in efficiency. The big problem for boundary value problems is how to jump to the finish line efficiently. By the way, it is important to note that some problems may not have steady state solutions or if they do, these solutions are unstable to perturbations. In general, it pays to think and be careful, half the fun of any problem may be in getting there. . . .

. . . and how to solve them In the following sections we will describe several approaches for discretizing and solving elliptic (2nd order) boundary value problems. For illustration we will consider the simplest elliptic problem which is a *Poisson problem* of the form

$$\nabla^2 U = f(\mathbf{x}) \quad (9.1.8)$$

and its first generalization to a medium with spatially varying properties

$$\nabla \cdot k \nabla U = f(\mathbf{x}) \quad (9.1.9)$$

where k may be something like a conductivity or permeability and is a non-constant function of space (in worse cases, k may be a function of U and then the problem is non-linear. We won't deal with that complication here.)

In a perfect world, Eq. (9.1.9) may have analytic solutions or integral solutions by way of Green functions or Fourier integrals. Any book on PDE's, complex analysis or mathematical physics will have details. It should be noted, however, that often, the "analytic" solution requires at least as much computational power to solve as its discrete numerical solution. Still if you can find it, an analytic solution often is often more informative.

9.2 Discretization

But the world is never perfect and this is a course on modeling afterall, so if you want to solve your problem numerically we need to begin by making it discrete. This section will discuss discretization by finite-difference techniques. A later section will discuss discretization into Fourier components.

Actually, we have already discussed finite difference approximations to n-dimensional spatial derivatives in the previous section on n-D Initial value problems. For boundary value problems, nothing has changed except that we don't have any time derivatives to kick around any more. For example, the standard 5-point

discretization of Eq. (9.1.8) on a regular 2-D cartesian mesh with grid spacing Δx and Δy is

$$\frac{1}{\Delta x^2}[U_{i-1,j} - 2U_{i,j} + U_{i+1,j}] + \frac{1}{\Delta y^2}[U_{i,j-1} - 2U_{i,j} + U_{i,j+1}] = f_{i,j} \quad (9.2.1)$$

or multiplying through by Δy^2 and writing in stencil form

$$\begin{bmatrix} & & 1 & & \\ \alpha^2 & & -2(1 + \alpha^2) & & \alpha^2 \\ & & 1 & & \end{bmatrix} U_{i,j} = \Delta y^2 f_{i,j} \quad (9.2.2)$$

where $\alpha = \Delta x/\Delta y$ is the aspect ratio of a grid cell. For a uniform grid spacing $\Delta x = \Delta y = \Delta$, this is the familiar 5-point stencil

$$\begin{bmatrix} & & 1 & & \\ 1 & & -4 & & 1 \\ & & 1 & & \end{bmatrix} U_{i,j} = \Delta^2 f_{i,j} \quad (9.2.3)$$

Which is readily generalized to Eq. (9.1.9) on a staggered mesh to be

$$\begin{bmatrix} & & k_{i,j+1/2} & & \\ \alpha^2 k_{i-1/2,j} & & -\Sigma & & \alpha^2 k_{i+1/2,j} \\ & & k_{i,j-1/2} & & \end{bmatrix} U_{i,j} = \Delta y^2 f_{i,j} \quad (9.2.4)$$

where

$$\Sigma = \alpha^2(k_{i-1/2,j} + k_{i+1/2,j}) + k_{i,j-1/2} + k_{i,j+1/2} \quad (9.2.5)$$

is simply the sum of the off-center components of the stencil and $k_{i+1/2,j}$ is the conductivity mid-way between points i and $i + 1$ (etc.). It is straightforward to show that for constant k , Eqs. (9.2.4) and (9.2.2) are identical up to the scale of k .

Equations (9.2.1) through (9.2.5) are all systems of linear equations that can be written in the general matrix form

$$\mathbf{Ax} = \mathbf{b} \quad (9.2.6)$$

where \mathbf{A} is a penta-diagonal matrix that looks like Figure 9.1. Where each stencil forms one line of the matrix. The important feature of these matrices is that they are incredibly sparse and the problem is to solve for \mathbf{x} efficiently in terms of time and storage. Here we will discuss three general approaches to solving Eq. (9.2.6): *Direct matrix methods* that attempt to solve for \mathbf{x} by gaussian elimination; *rapid methods* that make use of the specific properties of certain classes of problems to provide quick solutions; and *iterative methods* that try to approximate the inverse matrix and apply this approximate matrix repetitively to reduce the error. Each of these methods has its strengths and weaknesses and (as usual) we will highlight the basic do's and don't's.

Finally, a note about boundary conditions. In addition to specifying the interior stencil for determining \mathbf{A} it is necessary to specify the boundary conditions (after all it's a boundary value problem). In general, boundary conditions add auxiliary information that modifies the matrix or the right hand-side or both. However, there are many ways to implement the boundary conditions and these depend somewhat on the method of solution and will be dealt with in each of the sections below.

9.3 Direct Methods

Given that the basic problem is to somehow solve $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, one might think that the best way to proceed is to use Matlab or some other *direct-solver* to calculate \mathbf{A}^{-1} directly and do a matrix multiplication on \mathbf{b} . In practice, this is usually always a bad idea for two reasons. First, matrix inversion (for a dense matrix) is a rather expensive operation (of order N^3 operations where N is the number of unknowns... i.e. the number of grid points in your mesh). Second, even if the matrix \mathbf{A} is sparse, the corresponding inverse matrix is often not (see Figure 9.2) and the additional storage required for \mathbf{A}^{-1} can be considerable.

As any introductory book on linear algebra (such as Strang) will tell you, if you're solving $\mathbf{Ax} = \mathbf{b}$ you're really not interested in \mathbf{A}^{-1} at all but rather the solution \mathbf{x} and the best direct approach is by gaussian elimination, or in its alternative form, LU decomposition which factors a matrix into the product of a lower and upper triangular matrix i.e. $\mathbf{A} = \mathbf{LU}$ which are easily solved by forward and back-substitution. The basic algorithms of Gaussian elimination are trivial, however, numerically robust versions that can maintain precision for large and possibly ill-conditioned matrices require some care, particularly in the choice of pivot elements for elimination. Nevertheless, this is a well developed field and many options exist. For dense matrices, high-performance fortran open-source routines can be found in `Lapack` (and many of these algorithms probably underlie matlab). Numerical Recipes provides source code at a small cost but Matlab is probably the easiest interface to use, i.e. gaussian elimination is as simple as `x=A\b` and LU decomposition is just `[L,U]=lu(A)`. **Note:** for matrices more complicated

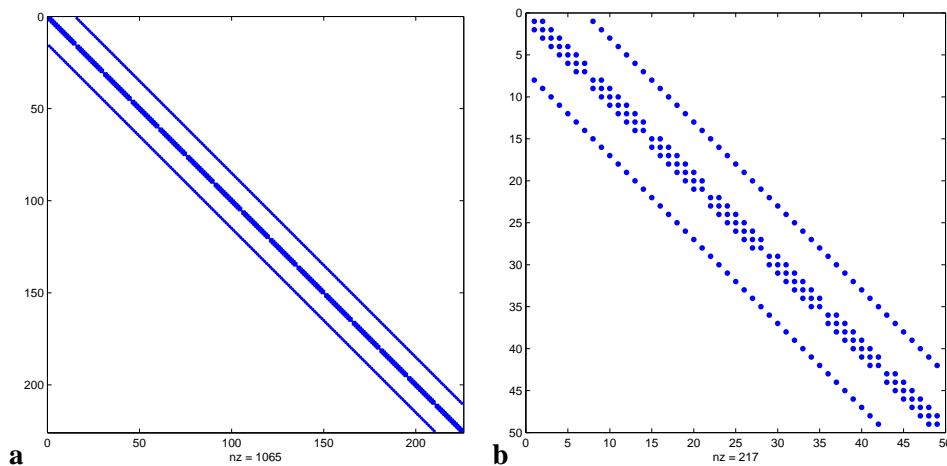


Figure 9.1: (a) Structure of a 5-point Laplacian operator on a 17×17 grid with dirichlet boundary conditions. Each dot marks the position of a single non-zero entry. The actual size of matrix \mathbf{A} is $(15 \times 15)^2 = 225^2 = 50625$ entries for the 225 unknown interior points. Note the incredible sparseness of this matrix. Of the 50625 entries, only 1065 are non-zero. (b) Same array but for a 9×9 grid (49 unknowns) to show the structure of the array better. Note: both of these arrays are incredibly small for real 2-D problems.

than tri-diagonal matrices, matlab’s gaussian elimination routines are significantly faster than LU decomposition (see Fig. 9.3). Matlab also makes it easy to move between dense matrices and sparse matrices as many of the calling interfaces are transparent to matrix type. For modeling problems, however, always use sparse matrices. Finally `y12m` is a freely available sparse LU solver in fortran available from www.netlib.org. `y12m` is an example of an *analyze, factorize, solve* package that attempts to reorder the matrix to make the most use of the sparsity structure. All of these routines can be quite useful, particularly for poorly conditioned small matrices. However, for many problems they are simply too expensive given the alternatives. Nevertheless, they should always be part of any numerical toolkit.

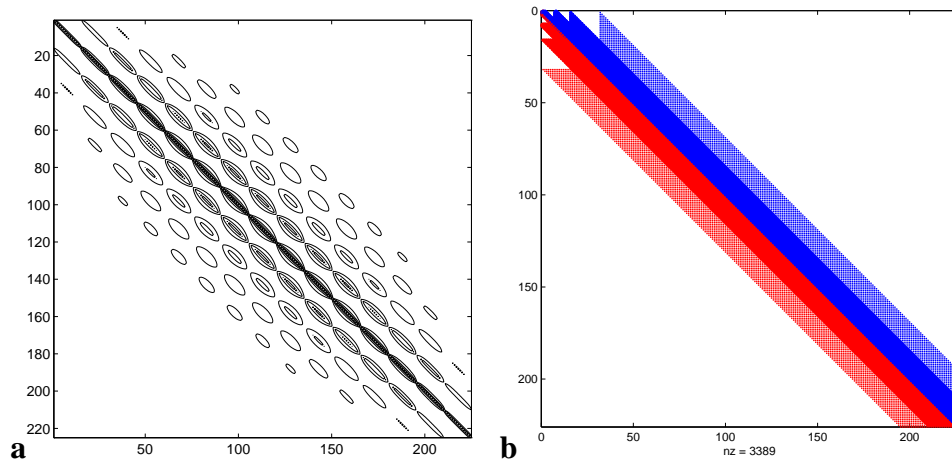


Figure 9.2: **a** Contour map of the inverse of A (for a 17×17 poisson problem) showing just the largest non-zero entries (the matrix is actually dense). Note that the sparsity of A is not necessarily retained by A^{-1} . **b** Sparsity structure of the LU decomposition of A is an improvement but shows significant fill-in between the bands (ignore the thin blue line which is some weird plotting artifact). As the matrices get larger, this fill-in becomes more significant.

9.3.1 Boundary conditions

As promised, we need to talk about implementing boundary conditions in these direct schemes. Because there is no “grid” in these problems (as least as far as the matrix solver is concerned), any modifications due to boundary conditions must be included either in the matrix A or in the right hand side vector \mathbf{b} . For *dirichlet* boundary conditions either approach can be used depending on whether you want to include the boundary points in the matrix inversion or not. In general, because matrix methods can be so expensive, the fewer points the better so one approach is just to solve for the unknown interior points¹. With this method, discretize the

¹This is why the matrix shown in Fig. 9.1b has $7 \times 7 = 49$ unknowns for a 9×9 grid with dirichlet conditions.

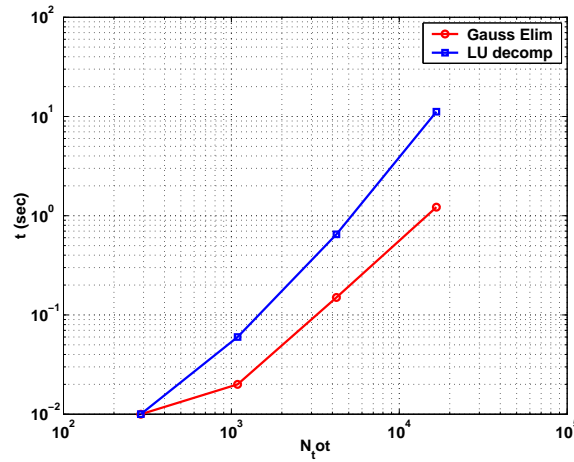


Figure 9.3: Comparison of solution times for a discrete 5-point Poisson problem with dirichlet boundary conditions using matlab's gaussian elimination routines and LU decomposition. Unlike the tridiagonal matrices, the pentadiagonal matrices are solved much faster with gaussian elimination. Note that for the larger sparse matrices, the solution time for sparse gaussian elimination scales roughly as $N^{1.5}$ where N is the total number of unknowns. The largest problem here corresponds to a 129×129 problem. Solution times shown here are for a 850Mhz PentiumIII running linux (aka my laptop).

unknown points next to the boundary and take all the known values over to the right hand side. For example, if the left edge $i = 1$ is dirichlet, the stencil equation for any non-corner point at $i = 2$

$$\begin{bmatrix} d \\ a & b & c \\ e \end{bmatrix} x_{2,j} = b_{2,j} \quad (9.3.1)$$

would be replaced by

$$\begin{bmatrix} d \\ 0 & b & c \\ e \end{bmatrix} x_{2,j} = b_{2,j} - ax_{1,j} \quad (9.3.2)$$

because $x_{1,j}$ is known. Again in sparse-schemes, the zero element in the stencil would not be stored (and actually doesn't lie within the matrix). The other approach for dirichlet conditions is to solve for the boundaries explicitly by writing a stencil equation for the boundary points like

$$\begin{bmatrix} 0 \\ 0 & 1 & 0 \\ 0 \end{bmatrix} x_{1,j} = f_{1,j} \quad (9.3.3)$$

where $f_{1,j}$ is a known function. This technique allows all the points in the grid to be solved for once without having to deal with subarrays. The actual cost of a dirichlet point is small because Eq. (9.3.3) can be inverted immediately.

For *Neumann or mixed* conditions we modify the matrix as explained in previous sections. I.e. if the left edge ($i = 1$) of the problem is a reflection boundary then we replace the centered stencil equation

$$\begin{bmatrix} & d & \\ a & b & c \\ & e & \end{bmatrix} x_{1,j} = b_{1,j} \quad (9.3.4)$$

with

$$\begin{bmatrix} & d & \\ 0 & b & a + c \\ & e & \end{bmatrix} x_{1,j} = b_{1,j} \quad (9.3.5)$$

For *periodic wrap-around* boundaries, additional matrix entries must be added to make sure the stencil includes the points near the far boundary.

A warning about boundary conditions Note, for elliptic problems and direct solvers, not all boundary conditions will produce unique answers in which case the problem is *ill posed*. For example, the solution of a Poisson problem like Eq. 9.1.8 with either doubly periodic or all Neumann boundaries is ill posed because given any solution u_0 then any solution such $u^* = u_0 + c$ where c is a constant is also a solution that matches the boundary conditions. This might seem to be a trivial problem, however, it actually forces the matrix A to be *singular* and therefore to have no unique inverse. This will cause a direct solver to fail although other methods can still be used for this problem.

9.4 Rapid Methods

Direct methods can be useful for small problems with strange matrix structure (e.g. for some of the matrices produced by finite element methods on unstructured grids). However for simpler problems you can usually do much better. In fact for certain classes of problems (of which the Poisson problem is one), there are *rapid* methods that can take advantage of some of the special properties of these the underlying matrix. Here we will deal with two rapid methods *Fourier Transform* spectral methods and combined *Fourier-cyclic reduction techniques*.

9.4.1 Fourier Transform Solutions

Problems with regular boundaries and constant coefficient stencils can often be solved using Fourier transform or *spectral techniques*. What makes them practical is that the underlying FFT (fast-Fourier transform) is an order $N \log_2 N$ transformation that takes N points in the space-domain exactly to N points in the frequency domain (and back again).

These methods start by noting that any continuous *periodic* function on a periodic domain of $N_i \times N_j$ points can be expressed exactly by its discrete inverse Fourier transform

$$u_{ij} = \mathcal{F}^{-1}[\hat{u}] = \frac{1}{N_i N_j} \sum_{m=1}^{m=N_i} \sum_{n=1}^{n=N_j} \hat{u}_{mn} e^{-ik_m x} e^{-ik_n y} \quad (9.4.1)$$

where the i 's in the exponentials are actually the imaginary number $\sqrt{-1}$ and

$$x = (i-1)\Delta x \quad y = (j-1)\Delta y \quad (9.4.2)$$

$$k_m = \frac{2\pi(m-1)}{(N_i-1)\Delta x} \quad k_n = \frac{2\pi(n-1)}{(N_j-1)\Delta y} \quad (9.4.3)$$

where k_m, k_n are the horizontal and vertical discrete wave numbers. The important features of the discrete Fourier transform is that it is linear, invertible and by the magic of the *Fast Fourier Transform* (FFT) can be evaluated in order $N \log_2 N$ operations in each direction² (not to mention there are lots of highly optimized versions around).

To use the FFT to solve Eq. (9.1.8), notice that because ∇^2 is a linear operator, we can find $\nabla^2 u$ by taking the Laplacian of each term in the summation³, i.e.

$$\nabla^2 u_{ij} = \frac{1}{N_i N_j} \sum_{m=1}^{m=N_i} \sum_{n=1}^{n=N_j} -(k_m^2 + k_n^2) \hat{u}_{mn} e^{-ik_m x} e^{-ik_n y} \quad (9.4.4)$$

but we also know that we can write the right hand side as

$$f_{ij} = \mathcal{F}^{-1}[\hat{f}] = \frac{1}{N_i N_j} \sum_{m=1}^{m=N_i} \sum_{n=1}^{n=N_j} \hat{f}_{mn} e^{-ik_m x} e^{-ik_n y} \quad (9.4.5)$$

thus term by term it must be true that

$$-(k_m^2 + k_n^2) \hat{u}_{mn} = \hat{f}_{mn} \quad (9.4.6)$$

Now $\hat{f}_{mn} = \mathcal{F}[f]$ is simply the Fourier transform of the right hand side and is readily evaluated. Thus to solve Eq.(9.1.8) we first find the FFT of the right-hand side, then divide each component by $(k_m^2 + k_n^2)$ (being particularly careful around the D-C component which should be zero⁴) and then finding u_{ij} by inverse transformation, i.e.

$$u_{ij} = \mathcal{F}^{-1} \left[\frac{-\hat{f}_{mn}}{(k_m^2 + k_n^2)} \right] \quad (9.4.7)$$

²so for simple power of 2 grids where $N_i = 2^p$ and $N_j = 2^q$ then the 2-D FFT takes $O(N_i N_j pq)$ operations e.g. a 64×64 grid takes $O(36 \times 65^2)$ operations

³Although this is only strictly true if the function shares the same boundary conditions as the underlying orthogonal functions. For other boundary conditions one cannot take the derivatives of a fourier series term by term.

⁴it is the possibility of a non-zero mean of the right-hand side that causes this problem to be singular with doubly periodic boundary conditions. This is the same problem that causes the direct methods to fail.

This approach will automatically work for doubly periodic boundaries because the discrete FFT's implicitly assume the functions are periodic. For Dirichlet boundaries where $u = 0$, you can expand the function in terms of discrete sin series, likewise for reflection boundaries you can use cos series. Numerical Recipes discusses these and more general boundary conditions in some detail as well as a slightly different approach for solving the discrete equation $\mathbf{A}\mathbf{u} = \mathbf{f}$ directly.

As an aside, spectral methods also can be used in initial value problems although they most often occur as *pseudo-spectral* techniques where the equations are not actually solved in the wave-number domain, however FFT's are used to evaluate spatial derivatives. For example, so far we have been evaluating advection terms like $\mathbf{V} \cdot \nabla T$ using a local stencil for the gradient operator. However, using the same trick we used to evaluate $\nabla^2 U$ in Eq. (9.4.4) we could also evaluate ∇T as

$$\nabla T = \mathcal{F}^{-1} [-i\mathbf{k}\mathcal{F}[T]] \quad (9.4.8)$$

where \mathbf{k} is the *wave-vector* $k_m\mathbf{i} + k_n\mathbf{j}$. Thus we first forward transform our temperature file, multiply by $i\mathbf{k}$ and then transform back. The net effect is to calculate the gradients of all the Fourier components simultaneously resulting in an extremely high-order method. Once we have ∇T we simply form the dot-product with the velocity field and away we go. For problems with smoothly varying fields and regular boundaries, these techniques can be highly accurate with much fewer grid points.

9.4.2 Cyclic Reduction Solvers

Spectral methods work for problems with constant coefficients and regular boundaries. A slightly more general set of rapid methods, however exists for problems that are *separable* in the sense of separation of variables. These methods include *cyclic reduction* and *FACR* (Fourier-analysis and cyclic reduction). Numerical Recipes gives a brief explanation of how these work which I will not repeat. What I will do is tell you about a lovely collection of FACR codes by Adams, Swartztrauber and Sweet called FISHPAK⁵. These are a collection of Fortran routines for solving more general Helmholtz problems such as

$$\nabla^2 U + \lambda U = f(\mathbf{x}) \quad (9.4.9)$$

in Cartesian, cylindrical and polar coordinates on regular and staggered meshes. They also can solve for 3-D Cartesian coordinates and the general 2-D separable elliptic problem

$$a(x)\frac{\partial^2 u}{\partial x^2} + b(x)\frac{\partial u}{\partial x} + c(x)u + d(y)\frac{\partial^2 u}{\partial y^2} + e(y)\frac{\partial u}{\partial y} + f(y)u = g(x, y) \quad (9.4.10)$$

for any combination of periodic or mixed boundary conditions. These codes are available from netlib (<http://www.netlib.org>) and are extremely fast with solution time scaling like $N_i N_j \log_2 N_i$. The principal (minor) drawback however

⁵For all you francophones Poisson is fish in French

is that they were written (extremely well) back in the early 80's and the fortran is inscrutable and therefore hard to modify for more general boundary conditions. In addition they still will only work for separable problems and could not, for example solve the more general problem $\nabla \cdot k \nabla T = \rho$, for a generally spatially varying conductivity. However, the only solvers that can compete in time with these routines and handle spatially varying coefficients are the iterative multi-grid solvers (see below). So if you have a relatively straightforward Poisson problem (particularly in non-cartesian geometries) these codes are highly recommended.

Because these codes are both highly useful and extremely inscrutable, the main thrust of this section is not to explain how they work but how you might use it. As an example, the documentation for the driver for HWSCRT (Helmholtz, regular grid, cartesian) is in the program and looks like.

```

SUBROUTINE HWSCRT (A,B,M,MBDCND,BDA,BDB,C,D,N,NBDCND,BDC,BDD,
1          ELMBDA,F,IDIMF,PERTRB,IERROR,W)
C
C
C      * * * * *
C      *
C      *           F I S H P A K
C      *
C      *   A PACKAGE OF FORTRAN SUBPROGRAMS FOR THE SOLUTION OF
C      *   SEPARABLE ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS
C      *   (VERSION 3.1 , OCTOBER 1980)
C      *
C      *           BY
C      *   JOHN ADAMS, PAUL SWARZTRAUBER AND ROLAND SWEET
C      *
C      *           OF
C      *   THE NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
C      *   BOULDER, COLORADO (80307) U.S.A.
C      *
C      *           WHICH IS SPONSORED BY
C      *   THE NATIONAL SCIENCE FOUNDATION
C      * * * * *
C
C      * * * * * PURPOSE * * * * *
C
C      SUBROUTINE HWSCRT SOLVES THE STANDARD FIVE-POINT FINITE
C      DIFFERENCE APPROXIMATION TO THE HELMHOLTZ EQUATION IN CARTESIAN
C      COORDINATES:
C
C      (D/DX)(DU/DX) + (D/DY)(DU/DY) + LAMBDA*U = F(X,Y).
C
C
C      * * * * * PARAMETER DESCRIPTION * * * * *
C
C      * * * * * ON INPUT * * * * *
C
C      A,B
C      THE RANGE OF X, I.E., A .LE. X .LE. B.  A MUST BE LESS THAN B.
C
C      M
C      THE NUMBER OF PANELS INTO WHICH THE INTERVAL (A,B) IS
C      SUBDIVIDED.  HENCE, THERE WILL BE M+1 GRID POINTS IN THE

```

C X-DIRECTION GIVEN BY $X(I) = A + (I-1)DX$ FOR $I = 1, 2, \dots, M+1$,
 C WHERE $DX = (B-A)/M$ IS THE PANEL WIDTH. M MUST BE GREATER THAN 3.
 C
 C MBDCND
 C INDICATES THE TYPE OF BOUNDARY CONDITIONS AT $X = A$ AND $X = B$.
 C
 C = 0 IF THE SOLUTION IS PERIODIC IN X , I.E., $U(I, J) = U(M+I, J)$.
 C = 1 IF THE SOLUTION IS SPECIFIED AT $X = A$ AND $X = B$.
 C = 2 IF THE SOLUTION IS SPECIFIED AT $X = A$ AND THE DERIVATIVE OF
 C THE SOLUTION WITH RESPECT TO X IS SPECIFIED AT $X = B$.
 C = 3 IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X IS
 C SPECIFIED AT $X = A$ AND $X = B$.
 C = 4 IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X IS
 C SPECIFIED AT $X = A$ AND THE SOLUTION IS SPECIFIED AT $X = B$.
 C
 C BDA
 C A ONE-DIMENSIONAL ARRAY OF LENGTH $N+1$ THAT SPECIFIES THE VALUES
 C OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X AT $X = A$.
 C WHEN MBDCND = 3 OR 4,
 C
 C $BDA(J) = (D/DX)U(A, Y(J))$, $J = 1, 2, \dots, N+1$.
 C
 C WHEN MBDCND HAS ANY OTHER VALUE, BDA IS A DUMMY VARIABLE.
 C
 C BDB
 C A ONE-DIMENSIONAL ARRAY OF LENGTH $N+1$ THAT SPECIFIES THE VALUES
 C OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X AT $X = B$.
 C WHEN MBDCND = 2 OR 3,
 C
 C $BDB(J) = (D/DX)U(B, Y(J))$, $J = 1, 2, \dots, N+1$.
 C
 C WHEN MBDCND HAS ANY OTHER VALUE BDB IS A DUMMY VARIABLE.
 C
 C C,D
 C THE RANGE OF Y , I.E., $C \leq Y \leq D$. C MUST BE LESS THAN D .
 C
 C N
 C THE NUMBER OF PANELS INTO WHICH THE INTERVAL (C, D) IS
 C SUBDIVIDED. HENCE, THERE WILL BE $N+1$ GRID POINTS IN THE
 C Y -DIRECTION GIVEN BY $Y(J) = C + (J-1)DY$ FOR $J = 1, 2, \dots, N+1$, WHERE
 C $DY = (D-C)/N$ IS THE PANEL WIDTH. N MUST BE GREATER THAN 3.
 C
 C NDBCND
 C INDICATES THE TYPE OF BOUNDARY CONDITIONS AT $Y = C$ AND $Y = D$.
 C
 C = 0 IF THE SOLUTION IS PERIODIC IN Y , I.E., $U(I, J) = U(I, N+J)$.
 C = 1 IF THE SOLUTION IS SPECIFIED AT $Y = C$ AND $Y = D$.
 C = 2 IF THE SOLUTION IS SPECIFIED AT $Y = C$ AND THE DERIVATIVE OF
 C THE SOLUTION WITH RESPECT TO Y IS SPECIFIED AT $Y = D$.
 C = 3 IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y IS
 C SPECIFIED AT $Y = C$ AND $Y = D$.
 C = 4 IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y IS
 C SPECIFIED AT $Y = C$ AND THE SOLUTION IS SPECIFIED AT $Y = D$.
 C
 C BDC
 C A ONE-DIMENSIONAL ARRAY OF LENGTH $M+1$ THAT SPECIFIES THE VALUES
 C OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y AT $Y = C$.
 C WHEN NDBCND = 3 OR 4,
 C
 C $BDC(I) = (D/DY)U(X(I), C)$, $I = 1, 2, \dots, M+1$.
 C
 C WHEN NDBCND HAS ANY OTHER VALUE, BDC IS A DUMMY VARIABLE.
 C
 C BDD
 C A ONE-DIMENSIONAL ARRAY OF LENGTH $M+1$ THAT SPECIFIES THE VALUES
 C OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y AT $Y = D$.
 C WHEN NDBCND = 2 OR 3,
 C
 C $BDD(I) = (D/DY)U(X(I), D)$, $I = 1, 2, \dots, M+1$.
 C
 C WHEN NDBCND HAS ANY OTHER VALUE, BDD IS A DUMMY VARIABLE.
 C
 C ELMBDA

C THE CONSTANT LAMBDA IN THE HELMHOLTZ EQUATION. IF
C LAMBDA .GT. 0, A SOLUTION MAY NOT EXIST. HOWEVER, HWSCRT WILL
C ATTEMPT TO FIND A SOLUTION.

C
C F
C A TWO-DIMENSIONAL ARRAY WHICH SPECIFIES THE VALUES OF THE RIGHT
C SIDE OF THE HELMHOLTZ EQUATION AND BOUNDARY VALUES (IF ANY).
C FOR I = 2,3,...,M AND J = 2,3,...,N
C
C $F(I,J) = F(X(I),Y(J)).$
C
C ON THE BOUNDARIES F IS DEFINED BY

MBDCND	F(1,J)	F(M+1,J)	
-----	-----	-----	
0	F(A,Y(J))	F(A,Y(J))	
1	U(A,Y(J))	U(B,Y(J))	
2	U(A,Y(J))	F(B,Y(J))	J = 1,2,...,N+1
3	F(A,Y(J))	F(B,Y(J))	
4	F(A,Y(J))	U(B,Y(J))	

NBDCND	F(I,1)	F(I,N+1)	
-----	-----	-----	
0	F(X(I),C)	F(X(I),C)	
1	U(X(I),C)	U(X(I),D)	
2	U(X(I),C)	F(X(I),D)	I = 1,2,...,M+1
3	F(X(I),C)	F(X(I),D)	
4	F(X(I),C)	U(X(I),D)	

C F MUST BE DIMENSIONED AT LEAST (M+1)*(N+1).
C
C NOTE
C IF THE TABLE CALLS FOR BOTH THE SOLUTION U AND THE RIGHT SIDE F
C AT A CORNER THEN THE SOLUTION MUST BE SPECIFIED.

C
C IDIMF
C THE ROW (OR FIRST) DIMENSION OF THE ARRAY F AS IT APPEARS IN THE
C PROGRAM CALLING HWSCRT. THIS PARAMETER IS USED TO SPECIFY THE
C VARIABLE DIMENSION OF F. IDIMF MUST BE AT LEAST M+1 .

C
C W
C A ONE-DIMENSIONAL ARRAY THAT MUST BE PROVIDED BY THE USER FOR
C WORK SPACE. W MAY REQUIRE UP TO $4*(N+1) +$
C $(13 + \text{INT}(\text{LOG}_2(N+1)))*(M+1)$ LOCATIONS. THE ACTUAL NUMBER OF
C LOCATIONS USED IS COMPUTED BY HWSCRT AND IS RETURNED IN LOCATION
C W(1).

C
C * * * * * ON OUTPUT * * * * *
C
C F
C CONTAINS THE SOLUTION U(I,J) OF THE FINITE DIFFERENCE
C APPROXIMATION FOR THE GRID POINT (X(I),Y(J)), I = 1,2,...,M+1,
C J = 1,2,...,N+1 .

C
C PERTRB
C IF A COMBINATION OF PERIODIC OR DERIVATIVE BOUNDARY CONDITIONS
C IS SPECIFIED FOR A POISSON EQUATION (LAMBDA = 0), A SOLUTION MAY
C NOT EXIST. PERTRB IS A CONSTANT, CALCULATED AND SUBTRACTED FROM
C F, WHICH ENSURES THAT A SOLUTION EXISTS. HWSCRT THEN COMPUTES
C THIS SOLUTION, WHICH IS A LEAST SQUARES SOLUTION TO THE ORIGINAL
C APPROXIMATION. THIS SOLUTION PLUS ANY CONSTANT IS ALSO A
C SOLUTION. HENCE, THE SOLUTION IS NOT UNIQUE. THE VALUE OF
C PERTRB SHOULD BE SMALL COMPARED TO THE RIGHT SIDE F. OTHERWISE,
C A SOLUTION IS OBTAINED TO AN ESSENTIALLY DIFFERENT PROBLEM.
C THIS COMPARISON SHOULD ALWAYS BE MADE TO INSURE THAT A
C MEANINGFUL SOLUTION HAS BEEN OBTAINED.

C
C IERROR

```

C      AN ERROR FLAG THAT INDICATES INVALID INPUT PARAMETERS.  EXCEPT
C      FOR NUMBERS 0 AND 6, A SOLUTION IS NOT ATTEMPTED.
C
C      = 0  NO ERROR.
C      = 1  A .GE. B.
C      = 2  MBDCND .LT. 0 OR MBDCND .GT. 4  .
C      = 3  C .GE. D.
C      = 4  N .LE. 3
C      = 5  NDBCND .LT. 0 OR NDBCND .GT. 4  .
C      = 6  LAMBDA .GT. 0  .
C      = 7  IDIMF .LT. M+1  .
C      = 8  M .LE. 3
C
C      SINCE THIS IS THE ONLY MEANS OF INDICATING A POSSIBLY INCORRECT
C      CALL TO HWSCRT, THE USER SHOULD TEST IERROR AFTER THE CALL.
C
C      W
C      W(1) CONTAINS THE REQUIRED LENGTH OF W.
C
C      * * * * *

```

Well if you understood all that, an example call for a rectangular domain $x_{min} \leq x \leq x_{max}$, $y_{min} \leq y \leq y_{max}$ with $NI \times NJ$ points with dirichlet boundary conditions would look like

```

call hwsqrt(xmin,xmax,(ni-1),1,tmp,tmp,
&          ymin,ymax,(nj-1),1,tmp,tmp,
&          0.,rhs,ni,pertrb,ierror,wk)

```

where `wk` and `tmp` are temporary work arrays. In general, these routines (like many useful pieces of freeware) are very hard to understand on a line-by-line level. However they are easy to test because we can always work out a simple test solution by forward substitution. As an example, if we assume our true solution is

$$u(x, y) = \sin\left(\frac{n\pi x}{x_{max}}\right) \sin\left(\frac{m\pi y}{y_{max}}\right) \quad (9.4.11)$$

(which is a $m \times n$ set of shear cells on a rectangular domain that stretches from $0 \leq x \leq x_{max}$, $0 \leq y \leq y_{max}$ see Fig. 9.4) then u is a solution of Eq. (9.1.8) if

$$f(x, y) = -\pi^2 \left[\left(\frac{n}{x_{max}}\right)^2 + \left(\frac{m}{y_{max}}\right)^2 \right] \sin\left(\frac{n\pi x}{x_{max}}\right) \sin\left(\frac{m\pi y}{y_{max}}\right) \quad (9.4.12)$$

and we use dirichlet boundaries $u = 0$ on all boundaries. Similar tests for Neumann boundaries can be made by swapping `cos` for `sin`. The problem set will provide a host of methods that apply this test .

9.5 Iterative Methods

In both direct and rapid solutions, we supply a right hand side (and possibly a matrix) and the technique returns the solution. Iterative techniques take a slightly different tack and are actually more related to solving the time dependent problem

$$\frac{\partial u}{\partial t} = \mathcal{L}u - f(\mathbf{x}) \quad (9.5.1)$$

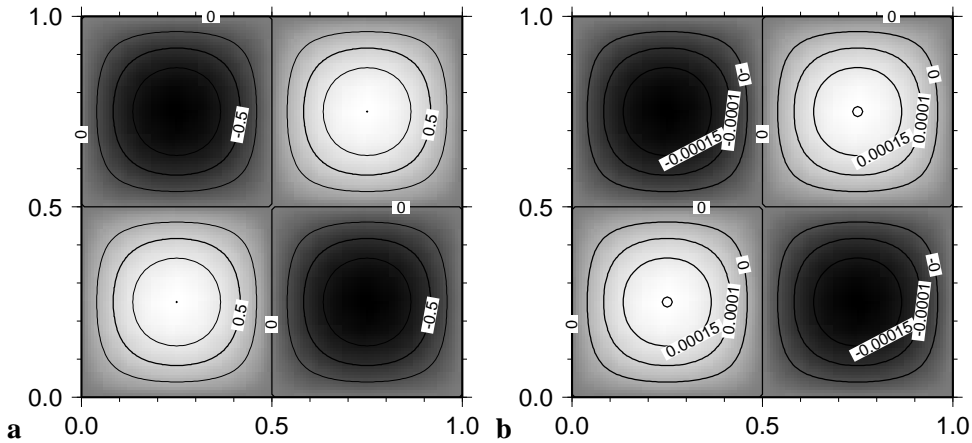


Figure 9.4: example of a 2×2 sin-cell test problem on the unit square 128^2 grid (using a multi-grid method). (a) typical solution. (b) typical errors

to steady state (\mathcal{L} is an arbitrary elliptic or higher order differential operator). I.e. we start out with some initial guess for our solution u and then iterate in “time” until the solutions stops changing. Because the iterations are effectively like time steps, all the standard techniques for implementing boundary conditions in time dependent problems can be use directly in iterative schemes. The only real trick to iterative methods is to do the problem in much less time than it would take to solve the time-dependent problem.

Another way to look at iterative methods is from the point of view of solving the matrix problem

$$\mathbf{A}\mathbf{u} = \mathbf{f} \quad (9.5.2)$$

Suppose we could split the matrix \mathbf{A} into two pieces one that was easy to invert and one that was hard to invert, i.e. $\mathbf{A} = \mathbf{E} + \mathbf{H}$. Then we could rewrite Eq. (9.5.2) as

$$\mathbf{E}\mathbf{u} + \mathbf{H}\mathbf{u} = \mathbf{f} \quad (9.5.3)$$

and form the iterative scheme

$$\mathbf{u}^{n+1} = \mathbf{E}^{-1} [\mathbf{f} - \mathbf{H}\mathbf{u}^n] \quad (9.5.4)$$

where \mathbf{u}^n is our solution at iteration n and \mathbf{u}^{n+1} is our improved guess. The matrix $\mathbf{E}^{-1}\mathbf{H}$ is known as the *iteration matrix* and it must have the property that all of its eigenvalues must be less than one. The point is that the iteration matrix is trying to reduce the errors in our guess \mathbf{u}^n and those errors can always be decomposed into orthogonal eigenvectors with the property that

$$\mathbf{E}^{-1}\mathbf{H}\mathbf{x} = \lambda\mathbf{x} \quad (9.5.5)$$

where \mathbf{x} is the eigenvector and λ is the eigenvalue. If λ is not less than one, repeated iteration of Eq. (9.5.4) will cause the error to grow and blow up. Even for

non-exploding iteration schemes, however, the rate of convergence will be controlled by the largest eigenvalue. Unfortunately for most iteration matrices, the largest eigenvalue approaches 1 as the number of points increase and thus simple schemes tend to converge quite slowly. The amount of decay caused by this largest eigenvalue is called the *spectral radius* ρ_s which goes asymptotically to one as the grid-size is increased (we will see why shortly).

Before we go on to illustrating some of these issues with the simplest (and not very good) classical iteration schemes, it is worth rewriting Eq. (9.5.4) in a slightly different form. If we simply add and subtract $\mathbf{E}\mathbf{u}^n$ within the brackets of the right hand side. i.e.

$$\mathbf{u}^{n+1} = \mathbf{E}^{-1} [\mathbf{f} - (\mathbf{E} + \mathbf{H})\mathbf{u}^n + \mathbf{E}\mathbf{u}^n] \quad (9.5.6)$$

then we can show that Eq. (9.5.4) is equivalent to

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \mathbf{E}^{-1}\mathbf{r} \quad (9.5.7)$$

where

$$\mathbf{r} = \mathbf{f} - \mathbf{A}\mathbf{u}^n \quad (9.5.8)$$

is the *residual*, i.e. the difference between the right-hand side and the right-hand side that would produce our initial guess \mathbf{u}^n . When (and if) we have converged $\mathbf{r} \rightarrow 0$ and $\mathbf{u}^{n+1} \rightarrow \mathbf{u}^n$. Thus another way to look at iterative methods is that we start with a guess, calculate the residual and try to feed it back into our guess in a way that makes the residual smaller.

9.5.1 Classical Methods: Jacobi, Gauss-Seidel and SOR

Clear as mud? Well let's illustrate these concepts with the simplest iterative scheme called a *Jacobi* scheme. This is a very old and very bad scheme but it illustrates all the points and in conjunction with multi-grid methods can actually be a very powerful technique.

In the Jacobi scheme we split our matrix \mathbf{A} as

$$\mathbf{A} = [\mathbf{D} + \mathbf{L} + \mathbf{U}] \quad (9.5.9)$$

where \mathbf{D} is the diagonal of the matrix and \mathbf{L} and \mathbf{U} are the lower and upper triangular sections of the matrix respectively (see Fig. 9.1). Now the diagonal has the trivial inverse $\mathbf{D}^{-1} = 1/D_{ij}$, thus the Jacobi scheme can be written generally as

$$\mathbf{u}^{n+1} = \mathbf{D}^{-1} [\mathbf{f} - (\mathbf{L} + \mathbf{U})\mathbf{u}^n] \quad (9.5.10)$$

or for the simple regular Poisson stencil

$$A_{ij} = \frac{1}{\Delta^2} \begin{bmatrix} & & 1 & & \\ & 1 & -4 & 1 & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix} \quad (9.5.11)$$

we can write the Jacobi scheme in stencil notation as

$$u_{ij}^{n+1} = -\frac{1}{4} \left[\Delta^2 f_{ij} - \begin{bmatrix} & & 1 & & \\ 1 & 0 & 1 & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix} u_{ij}^n \right] \quad (9.5.12)$$

If that looks vaguely familiar it should because we have actually seen it before in our time dependent problems. A more physical way to get at (and understand) the Jacobi scheme is to start from the time dependent form of the diffusion equation

$$\frac{\partial u}{\partial t} = \nabla^2 u - f \quad (9.5.13)$$

and simply perform a FTCS differencing on it to produce

$$u_{ij}^{n+1} = -\beta \left[\Delta^2 f_{ij} - \begin{bmatrix} 1 & & 1 \\ & (1/\beta - 4) & \\ 1 & & 1 \end{bmatrix} u_{ij}^n \right] \quad (9.5.14)$$

where $\beta = \Delta t / \Delta x^2$ is the diffusion parameter (see Section 8, Eq. (8.5.6)). Thus the Jacobi scheme is identical to taking the maximum FTCS step allowable with $\beta = 1/4$, and that's the big problem. If you remember from the discussion on explicit diffusion schemes, then you will recall that β is effectively the time scale for the highest frequency components to decay. However, in general the amount of decay an arbitrary wavelength d will experience in one time step (for a scaled diffusivity $\kappa = 1$) is approximately

$$\rho_s \simeq \exp \left[-\frac{4\pi^2 \Delta t}{d^2} \right] \quad (9.5.15)$$

so if our grid has $N \times N$ points, our longest wavelength is approximately $d = N\Delta x$ and taking the maximum time step $\Delta t = \Delta x^2/4$ shows that within one time step the amount of decay we can expect for our longest wavelength error is about

$$\rho_s \simeq \exp \left[-\frac{\pi^2}{N^2} \right] \quad (9.5.16)$$

which for large N is

$$\rho_s \sim 1 - \frac{\pi^2}{N^2} \quad (9.5.17)$$

(the actual spectral radius for the Jacobi step is really $\rho_s = 1 - \pi^2/(2N^2)$) thus the physical meaning of the spectral radius is the amount of decay that our longest wavelength error component will decay in one iteration. The amount of decay we can expect in J iterations is therefore

$$\rho_j \simeq \exp \left[-\frac{\pi^2 J}{N^2} \right] \quad (9.5.18)$$

and thus to reduce the error by a factor of e^{-1} with this scheme will require on order N^2 iterations. This is not good.

Can we do better? Well there are some additional simple schemes which are worth knowing although they are not particularly good either. The first variation on this theme are called *Gauss-Seidel* schemes which are easier to see in Fortran than in matrix notation. The Jacobi scheme in Fortran looks like

```

loop over i and j
  unew(i,j)=-.25*(f(i,j)-(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)))
end loop

```

where we do all the smoothing into a new array. The simplest Gauss-Seidel scheme could be written

```

loop over i and j
  u(i,j)=-.25*(f(i,j)-(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)))
end loop

```

i.e. we just do the smoothing in place using the new values of u as they come up. What do we gain from this? Not a whole lot but the spectral radius of a Gauss-Seidel scheme is the square of that for the Jacobi scheme so it converges in about a factor of 2 less time (still no good). The other advantage is that the Gauss-Seidel scheme can remove certain frequencies (like a checkerboard pattern of errors) that are invisible to the Jacobi scheme. Speaking of checkerboards, there is another variant of Gauss-Seidel called *red-black Gauss-Seidel* where instead of looping over all the points, we notice from the above algorithm that if we colored our grid points red and black like a checkerboard, the red points only depend on the black ones and vice-versa. So we could update all the red ones and then all the black ones which gains us about another factor of 2 and produces less biased errors. We will use the red-black schemes quite a bit.

Okay, even with the Gauss-Seidel schemes, these simplest iterative schemes are still impractical because they require N^2 iterations which makes these schemes order N_{tot}^2 schemes where $N_{tot} = N^2$ is the total number of points on the grid. Clearly for even moderately sized grids this is a disaster. The only way you can do better than this on a simple iterative scheme is to use *Successive Over Relaxation* or SOR. SOR is based on the notion that we can write either the Red-Black Gauss-Seidel or the Jacobi scheme in residual form as

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \mathbf{D}^{-1}\mathbf{r} \quad (9.5.19)$$

or in component notation as

$$u_{ij} = u_{ij} + r_{i,j}/D_{ij} \quad (9.5.20)$$

now we could generalize this by including a relaxation parameter ω such that

$$u_{ij} = u_{ij} + \omega r_{i,j}/D_{ij} \quad (9.5.21)$$

where $0 < \omega < 2$; i.e. we add in a variable amount of the residual (and hope that it improves the rate of convergence). If $\omega < 1$ its called *underrelaxation*; for $1 < \omega < 2$ it's *overrelaxation*. For $\omega > 2$ it blows up. It can be shown that only overrelaxation can produce better convergence than Gauss-Seidel and then only for an optimal value of ω such that

$$\omega = \frac{2}{1 + \sqrt{1 - \rho_{jacobi}^2}} \quad (9.5.22)$$

with this value, the spectral radius of SOR is

$$\rho_{SOR} \sim 1 - \frac{2\pi}{N} \quad (9.5.23)$$

for large N . This is a big improvement, but not really big enough because the solution time now scales as $N_{tot}^{1.5}$. A perfect scheme would have the solution time increase linearly with N_{tot} but for this we need multi-grid. Figure 9.5 shows that SOR does converge in about N iterations, however, it does so in a rather peculiar way where the residual increases dramatically before decreasing. By the way, SOR only works this well for a narrow region around the optimal value of ω . However, for more general problems than Poisson equations, finding the optimal value of ω can be problematic, particularly if the operator changes with time.

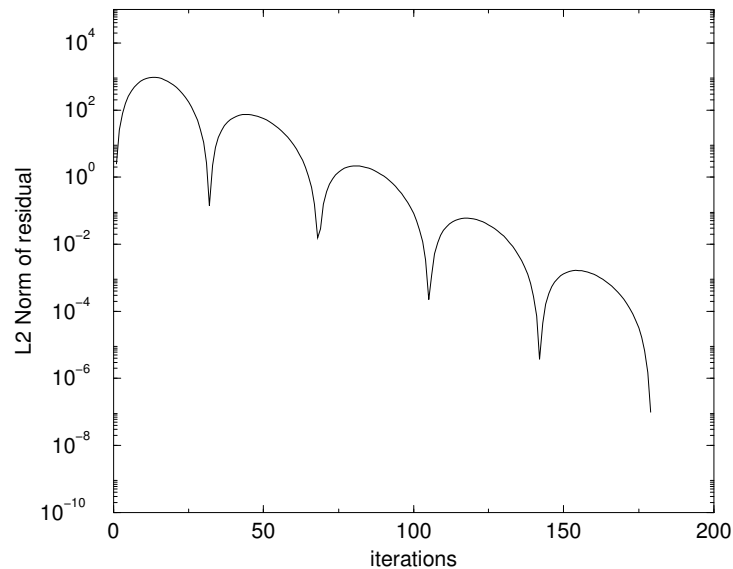


Figure 9.5: Convergence behaviour of optimal SOR (with chebyshev acceleration and red-black ordering) for a 128×128 square grid. Note that the residual increases wildly before decaying and still requires of order N iterations to converge.

9.5.2 Multigrid Methods

Multigrid techniques start from the realization that the simple iterative schemes of the previous section are not actually very good relaxers; however, they are very good smoothers, i.e. while they cannot reduce errors at all frequencies equally, they are very efficient at smoothing out the highest frequency errors. Moreover we note that the “highest frequencies” in our problem are defined only by the grid spacing. Thus it ought to be possible to use these simple smoothing schemes on a hierarchy of grids with different spacings to remove errors at all frequencies. This is the essence of multi-grid and the rest of this chapter will show how we can actually implement this idea to produce iterative schemes that are as efficient as

the best FACR scheme for both constant and variable coefficient matrices and even non-linear problems.

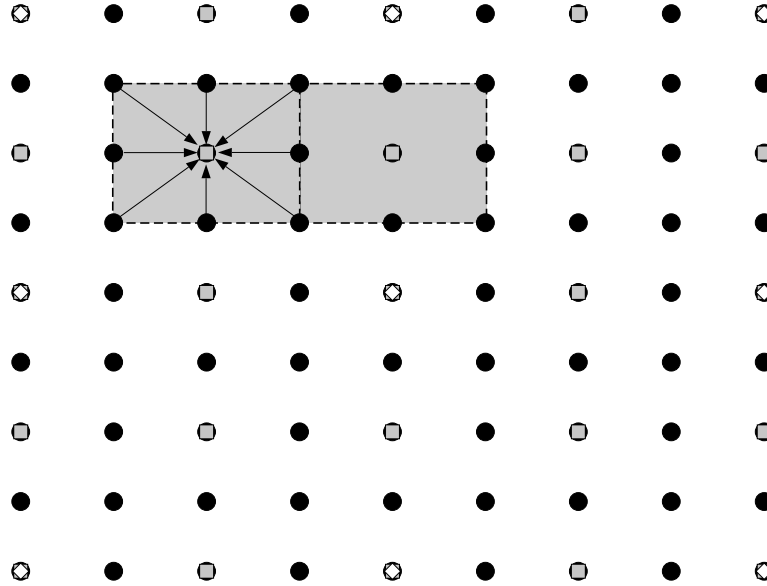


Figure 9.6: An example of a 3 level nested multi-level grid (or just a multi-grid). The *fine grid* or *level 1* is shown by the dots and has 9×9 grid points. Level 2 is shown by the grey squares and has been coarsened by a factor of 2 to have only 5×5 points. The coarsest grid is level 3 (diamonds) with only 3×3 points. Note that in 2-D each coarser grid has only $1/4$ of the points of the next finer grid. The grey boxes show a restriction operation from level 1 to level 2 where we set the value of the coarse grid point equal to the average value of the control volume defined on the fine grid. Interpolation takes information from coarse grids to fine grids

Before we discuss algorithms, we first need to define what we mean by a *multi-level* grid. Figure 9.6 shows a 3-level grid where each coarser grid has a grid spacing that is twice that of the next finer grid (and therefore ~ 4 times less points on each level). Note that all the grids share the same boundaries and the number of grid points in both directions must be repeatedly divisible by two. While we can't have completely arbitrary numbers of grid points in each direction, the restrictions are less severe than simple spectral methods which usually require power of 2 grids. In general, to guarantee the properties of the multi-grid we calculate the number of grid points in each direction by first specifying the *aspect ratio* of the coarsest grid in *grid cells* (i.e. $n_{ci} \times n_{cj}$) then the number of total levels in the grid n_g . Given these three numbers, the number of grid points in the fine grid is given by

$$N_i = n_{ci}2^{(n_g-1)} + 1 \quad (9.5.24)$$

$$N_j = n_{cj}2^{(n_g-1)} + 1 \quad (9.5.25)$$

For the example shown in Fig. 9.6 $n_{ci} = n_{cj} = 2$, $n_g = 3$ and therefore $N_i = 9$ and $N_j = 9$. Once the number of grid points on any level is known, the number of grid-points in the next coarser level is simply $N_i^{l+1} = N_i^l/2 + 1$ etc. where integer

math is assumed (we will also need to know how to relate coordinates between grids but we will get to that shortly).

Given the basic definition of the multi-level grid we'll start by just understanding the algorithm for a two-level problem and then show that you can create all other schemes by recursive calls to the two level scheme. Consider that we want to solve some linear elliptic problem

$$\mathcal{L}\mathbf{u} = \mathbf{f} \quad (9.5.26)$$

which is approximated by the discrete matrix problem

$$\mathbf{A}^h \mathbf{u}^h = \mathbf{f}^h \quad (9.5.27)$$

on a grid with grid spacing h (this is the fine grid, the next coarser grid on level 2 has grid spacing $2h$). Now let's apply a simple relaxation scheme like red-black gauss Seidel to our initial guess for just a few (e.g. 2–3) relaxation sweeps to give us an improved solution $\tilde{\mathbf{u}}^h$. $\tilde{\mathbf{u}}^h$ will still have a large amount of long wavelength error, but the high-frequency components will have been smoothed out⁶. We can now write our true discrete solution solution as

$$\mathbf{u}^h = \tilde{\mathbf{u}}^h + \mathbf{e}^h \quad (9.5.28)$$

where \mathbf{e}^h is the *correction* that we would need to add to our current guess to get the correct solution⁷. Substituting Eq. (9.5.28) into (9.5.27) and using the fact that \mathbf{A}^h is a linear operator, we can rewrite (9.5.27) as

$$\mathbf{A}^h \mathbf{e}^h = \mathbf{r}^h \quad (9.5.29)$$

where $\mathbf{r}^h = \mathbf{f}^h - \mathbf{A}^h \tilde{\mathbf{u}}^h$ is the residual. Equation (9.5.29) shows that for linear problems, one can solve for either the solution or the correction interchangeably (in linear multi-grid we will usually be solving for the correction). The point is that if we could solve (9.5.29) for the correction, then we could add it back to our improved guess and we would be done. Of course it is not any easier to solve for the correction than for the solution on the fine grid. However, we have already pre-smoothed our guess on the fine grid so the residual (and therefore the correction) should not contain any fine-grid scale errors. Thus if we could solve for the correction on a coarser grid we would not be losing any information but we would be gaining an enormous amount of time given the factor of 4 reduction in points at the next grid level. More specifically, we want to solve

$$\mathbf{A}^{2h} \mathbf{e}^{2h} = \mathbf{r}^{2h} \quad (9.5.30)$$

where \mathbf{A}^{2h} is our operator defined on the next coarser grid and

$$\mathbf{r}^{2h} = \mathcal{R}_h^{2h} \mathbf{r}^h \quad (9.5.31)$$

⁶While you have a lot of latitude in the relaxation scheme you use in multi-grid, don't use SOR because the over-relaxation destroys the smoothing behaviour of the relaxation and will cause things to blow up.

⁷the correction \mathbf{e} is also often called the *error* in the solution.

is the *restriction* of the fine grid residual onto the coarse grid. \mathcal{R}_h^{2h} is called the restriction operator and is some mechanism for transferring information from a fine grid to one coarser grid. There are many ways to do restriction but a standard approach is to simply use the average value of the control volume defined on the coarse grid (see Fig. 9.6). Thus for every point on the coarse grid (ic, jc) we can define the corresponding fine grid coordinates

$$if = 2ic + 1 \quad (9.5.32)$$

$$jf = 2jc + 1 \quad (9.5.33)$$

and the *full weighting restriction* for an interior point can be written in stencil form as

$$r_{ic,jc}^{2h} = \begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix} r_{if,jf}^h \quad (9.5.34)$$

where the stencil operates on the fine grid but is only evaluated for the number of points on the coarse grid. As for defining the coarse grid operator, this is probably the most difficult problem in multi-grid methods, however, the standard heuristic is to just define the operator as if you were simply differencing the problem on the coarse grid. Only start getting clever when that doesn't work. In the case of a Poisson problem the operator is just

$$A_{ic,jc}^{2h} = \frac{1}{(2h)^2} \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} \quad (9.5.35)$$

Given the operator and the right hand side, we could now solve the matrix problem Eq. (9.5.30) using any of the solvers we have already discussed but for much less computational cost on the coarse grid.

Just for argument sake, let's pretend that we can solve Eq. (9.5.30) exactly for the correction \mathbf{e}^{2h} on the coarse grid. Again, since \mathbf{e}^{2h} shouldn't have any high frequency components it can be related to the correction on the fine grid by *interpolation*, i.e.

$$\mathbf{e}^h = \mathcal{I}_{2h}^h \mathbf{e}^{2h} \quad (9.5.36)$$

where \mathcal{I}_{2h}^h is the *interpolation* or *projection operator* that moves information from the coarse grid to the fine grid. The simplest interpolation operator is just bilinear interpolation where fine-grid points that are coincident with coarse grid points are set to the coarse value, fine-grid points that lie on coarse grid line are just half their nearest coarse neighbors and fine points that are in the center of coarse cells are just 1/4 of the corner points.

Finally, given our new solution for the correction on the fine grid we update our initial guess using Eq. (9.5.28). If the restriction, and interpolation processes added no high frequency error to our solution then we would be done; however, in general some error is always introduced moving between grids and the final step is to relax a few times starting with our improved guess to remove any high-frequency errors that have been introduced. To recap, the two-level correction scheme is

- Given an initial guess, **relax** on the fine grid N_{PRE} times for $\tilde{\mathbf{u}}^h$
- Form the fine grid **residual** $\mathbf{r}^h = \mathbf{f}^h - \mathbf{A}^h \tilde{\mathbf{u}}^h$
- **restrict** the residual to the coarse grid and form the new right hand side $\mathbf{r}^{2h} = \mathcal{R}_h^{2h} \mathbf{r}^h$
- **solve** $\mathbf{A}^{2h} \mathbf{e}^{2h} = \mathbf{r}^{2h}$ for \mathbf{e}^{2h}
- **interpolate** the correction to the fine grid via $\mathbf{e}^h = \mathcal{I}_{2h}^h \mathbf{e}^{2h}$
- **correct** $\tilde{\mathbf{u}}^h$ to form $\mathbf{u}^h = \tilde{\mathbf{u}}^h + \mathbf{e}^h$
- **relax** the new solution N_{POST} times

This two-level scheme can be repeated until convergence however it can also be applied recursively. Right now we have assumed that we have some technique (e.g. a direct solver) to solve exactly for the coarse grid correction. However, if the coarse grid is still fairly large it will also contain long-wavelength errors that will be costly to remove in a single pass. An alternative is to just continue to repeat the process iteratively. i.e. rather than solve Eq. (9.5.30) exactly for \mathbf{e}^{2h} we can apply our simple relaxation scheme a few times on the coarse grid to remove order $2h$ wavelength errors. Our new guess will still have longer wavelength errors so we need to do a coarse grid correction for the coarse grid correction (crystal clear eh? wait a bit and all will be revealed). We repeat the process on increasingly coarser grids, each time just removing the errors that are comparable to the grid spacing until we reach a sufficiently coarse level so that it really is cheap to solve exactly for the correction (to the correction to the correction to the ...). We then move back down again correcting each level. This particular iterative scheme is called a *V-cycle* and is illustrated schematically in Fig. 9.7. In pseudo-code we can define a V-cycle algorithm like

```
do levels from 1 to Ngrids-1 ! go up the V
  relax(npres times)
  find_residual(level)
  restrict(residual(level) to rhs(level+1))
enddo

solve correction on coarsest level

do levels from Ngrids-1,1,-1 ! go down the V
  interpolate(correction(level+1) to correction(level))
  add(correction(level)+solution(level))
  relax(nposts times)
enddo
```

Because you can implement different number of relaxations in the V-Cycle we will often specify them in the name of the V-cycle. For example, a useful scheme for Poisson problems is to use a (2,1) V-cycle where $N_{PRE}=2$ and $N_{POST}=1$. The V-cycle can then be called successively monitoring the norm of the fine-grid residual after every V-cycle. When the residual has been reduced to some predetermined tolerance, we can stop. Properly implemented, a standard V-cycle can reduce the

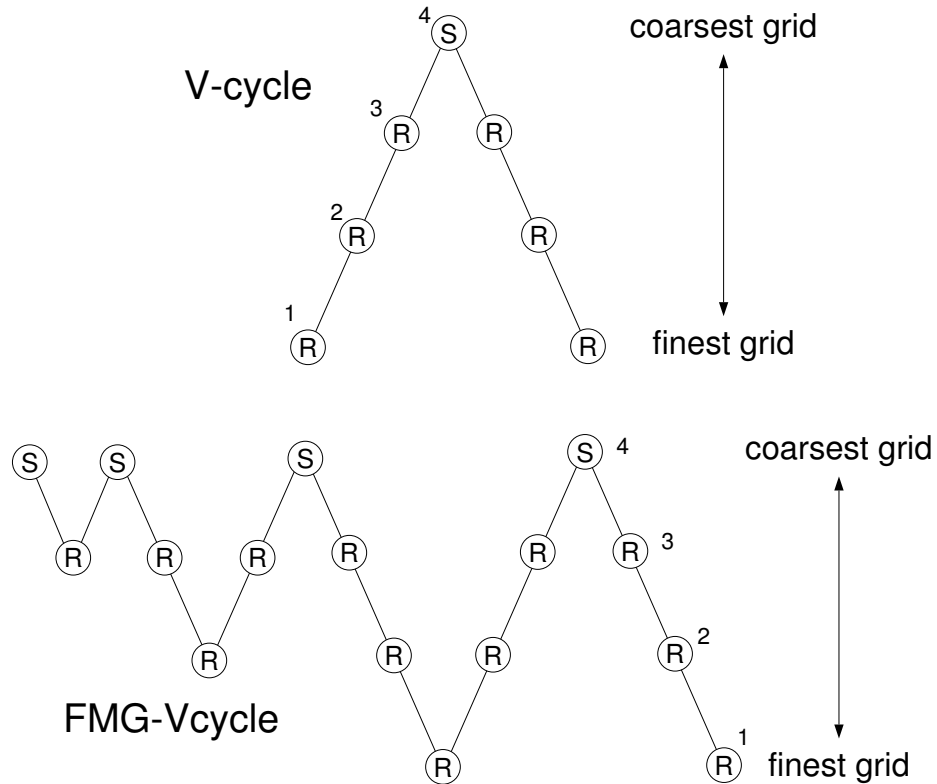


Figure 9.7: schematic illustrations of V-cycle and Full Multi-grid FMG-cycle schemes for a four level grid. Level 1 is the fine grid, level 4 is the coarse grid. Circles with R's denote relaxation (smoothing) steps, Circles with S's denotes exact solution on the coarse grid. Ascending lines denote restriction (fine-to-coarse) operations. Descending lines denote interpolation (coarse-to-fine) operations. In general a V-cycle is used when you have a good initial guess for your solution. Each V-cycle should reduce the fine-grid residual by about an order of magnitude. FMG-cycles (which are actually nested V-Cycles) are used when an initial guess is unknown. Properly coded, a FMG-cycle can behave like an order N direct solver.

norm of the residual by an order of magnitude per V-cycle, with only about as much work as it takes to smooth the finest grid ($N_{PRE}+N_{POST}+3$) times. Moreover, this convergence rate is independent of the size of the problem (because all frequencies are reduced efficiently) and the solution time scales only as the number of fine-grid points. Truly amazing.

In the next section we will show how to specifically implement this algorithm. Before we do that, however it is also useful to mention another cycling scheme called *Full Multigrid* or *nested iteration*. In the V-cycle, we begin on the finest grid with what we hope is a reasonable first guess for our solution. If we are doing a time dependent problem, we could use the solution at the last time step which is often very close. However, if we really don't have a good guess for the fine grid solution, we can use FMG. The FMG scheme is actually a set of nested V-cycles of increasingly more levels. First we need to restrict the right-hand side f to all the levels and then, instead of starting on the fine grid we start on the coarse grid

and *solve* for $\mathbf{u}^H = \mathbf{A}^{-1H} \mathbf{f}^H$. Given this coarse grid solution, we interpolate it to one finer grid and use it as the first guess of a 2 level V-cycle. Given this improved guess we interpolate it down again to use as the initial guess of a 3-level V-cycle etc. until we reach the finest grid. The FMG-cycle is shown schematically in Fig. 9.7 and can be represented in pseudo-code as

```
do levels from 1 to Ngrids-1 ! first restrict f
  restrict(rhs(level) to rhs(level+1))
enddo

solve for u(ngrid)

do levels=Ngrids-1 to 1 ! nested vcycles
  interpolate(u(level+1) to u(level))
  vcycle(u(level),from level to Ngrids)
enddo
```

9.5.3 Practical computation issues: how to actually do it

The previous discussion is lovely and abstract however it doesn't really tell you how to write your own multi-grid solver. Actually, I will be giving you a working 2-D multi-grid solver with a fair numbers of bells and whistles but this section tells you how it works. Given a few initial pointers (particularly on storage), multi-grid schemes are not particularly hard to write because the individual pieces (relax, resid, restrict, solve, interpolate), are quite modular, represented by simple stencil operations and at most only have to deal with two-grids at a time. The most difficult part of these schemes is getting the storage right. Numerical Recipes (which is usually phenomenal) really screws this one up. The scheme I'll use is discussed in Briggs [1] and makes rather elegant use of all the features of Fortran array handling that I have been emphasizing.

The first issue is how to store all of these grids of different sizes in an efficient and orderly manner. As usual we will store the entire multi-level grid in one composite 1-D array; however, because the different grid levels have different numbers of grid points, the best way to pack them into memory is to use *index pointers*. Remember that in Fortran, all arrays are passed by address and it is possible to pass 1-D arrays to subroutines that treat them like $n - D$ arrays. Thus in Fortran it is legal to have a call like

```
real array(100)
integer ni,nj
ni=5
nj=5
call do_something(array(11),ni,nj)
...
subroutine do_something(arr,ni,nj)
integer ni,nj
real arr(ni,nj)
...
```

which will operate on the 25 consecutive points *starting at* `array(11)` (i.e. it will work on the memory in the subarray `array(11:35)` and treat it like a 2-D array

within the subroutine). How convenient. Thus to store an entire 4-level grid in a single 1-D array, we need to calculate the offset indices (or index pointers) so that $ip(1)$ is the index of the start of the fine grid, $ip(2)$ points to the beginning of the level 2 grid etc up to $ip(ngrids)$. Here's a little subroutine that does just that

```

c*****
c  subroutine to calculate index pointers for composite arrays
c  ip: is array of pointers
c  ng: total number of grid levels
c  ni,nj: number of grid points in i and j direction on finest grid
c  len: total length of composite array
c *****

      subroutine initpointer(ip,ng,ni,nj,len)
      implicit none
      integer ng,ni,nj,len
      integer ip(ng)

      integer n,nni,nnj

      ip(1)=1
      nni=ni
      nnj=nj
      do n=2,ng                ! increasing coarseness
         ip(n)=ip(n-1)+nni*nnj
         nnj=nnj/2+1
         nni=nni/2+1
      enddo
      len=ip(ng)-1+nni*nnj
      return
      end

```

Given $ip(1:ng)$ and len , the storage scheme of Briggs needs 3 composite arrays of length len : $u(len)$ to hold the solution (and corrections), $rhs(len)$ to hold the the right-hand sides, and $res(len)$ and the residual and interpolations. In addition we may also need to pass in a variable coefficient operator which in the most general case could be a full 5-point stencil operator $aop(5, len)$. These four arrays and their storage layout are shown in Figure 9.8 for a 4-level grid

With this storage scheme it is straightforward to efficiently implement a $vcycle$ in Fortran. Here is an example for a simple version with hard-wired boundary conditions

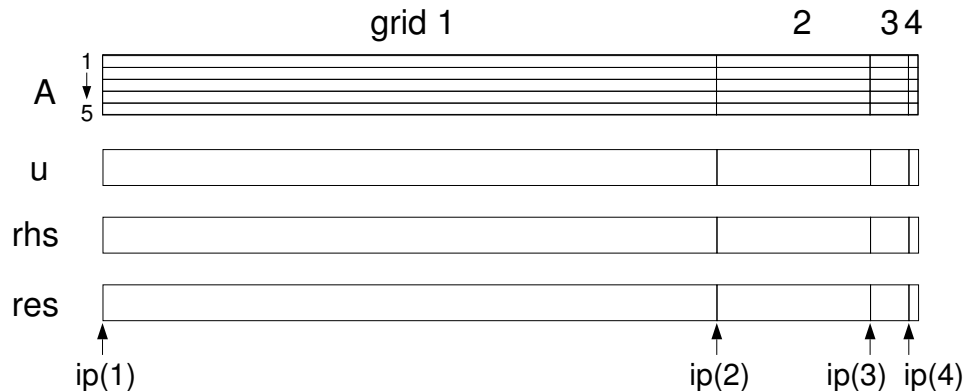


Figure 9.8: Multi-grid storage scheme ala Briggs [1] illustrated for a 4 level grid. All the composite grids for the operator, solutions, right-hand side, and residual are stored as 1-D arrays with index pointers $ip(1:ng)$. $u(ip(1))$ is the beginning of the fine grid, $u(ip(2))$ is the beginning of the next coarser grid (which has $\sim 1/4$ the points in 2-D), up to $u(ip(ng))$ which is the coarsest grid. In general the total length of the composite grid will always be less than $2^D/(2^D - 1)$ times the size of the fine grid (For series fans, $\sum_{n=0}^{\infty} 1/2^{nD} = 2^D/(2^D - 1)$) Thus in 2-D the total length is less than $4/3N_{fine}$ and in 3-D it's less than $8/7N_{fine}$.

```

c*****
c subroutine mg2d_vcyc
c one V cycle, of height ng
c NB: vcyc has no knowledge of the total number of grids, thus
c   ip must be passed such that cmp(ip(1)) is the beginning
c   of the solution array at height'th grid!!!!
c   Also, dz must be dz for the height'th grid!!!!
c   important variables
c aa: composite array for 5 point stencil operator
c uu: composite array for solution and corrections
c rhs: rhs
c res: composite array for residual and interpolants
c ip: 1-D,array of pointers to sub grids in uu,rhs,res
c len: length of composite grids,uu,rhs etc.
c ng: maximum number of grids in vcycle
c ni: number of horizontal points in finest grid
c nj: number of vertical points in finest grid
c ncycle: number of vcycles before convergence check
c npre: number of coarsening relaxations
c npos: number of fining relaxations
c nsolve: number of relaxations at coarsets grid
c dz: true grid spacing on finest grid
c*****
  subroutine mg2d_vcyc(aa,uu,rhs,res,ip,len,ng,ni,nj,ncycle,npre,npos,nsolve)

  implicit none

  integer len,ng
  real aa(5,ng)
  real uu(len)
  real rhs(len)
  real res(len)
  integer ip(ng),step,ncycle,npre,npos,pre,pos
  integer nni,nnj,cycle,nsolve,solve
  integer ni,nj

  do cycle=1,ncycle          ! the number of times to trace the 'V'
    nni=ni
    nnj=nj

```

```

        call arrfill0(uu(ip(2)),len-nni*nnj) ! zero out the correction
c-----Fine to coarse leg of the V
c
        do step=1,ng-1
            call mg2d_relax(npre,aa(1,step),uu(ip(step)),rhs(ip(step)),nni,nnj)
            call
            mg2d_resid(aa(1,step),res(ip(step)),uu(ip(step)),rhs(ip(step)),nni,nnj)
            nni=nni/2+1
            nnj=nnj/2+1
            call mg2d_rstrct(rhs(ip(step+1)),res(ip(step)),nni,nnj)
            call arrmult(rhs(ip(step+1)),nni*nnj,4.) ! why is this here?
        enddo
c
c-----Solve on coarsest grid by just relaxing nsolve times
c
        call mg2d_relax(nsolve,aa(1,ng),uu(ip(ng)),rhs(ip(ng)),nni,nnj)
c
c-----Coarse to fine leg the V
c
        do step=ng-1,1,-1
            nni=2*nni-1
            nnj=2*nnj-1
            call mg2d_addint(uu(ip(step)),uu(ip(step+1)),res(ip(step)),nni,nnj)
            call mg2d_relax(npos,aa(1,step),uu(ip(step)),rhs(ip(step)),nni,nnj)
        enddo
        enddo
        return
    end

```

Note that while all the storage is done in 1-D arrays, the actual stencil operations are implemented in subroutines that use 2-D arrays. As an example, here is the code for a red-black gauss-Seidel relaxation scheme on any level using dirichlet boundary conditions.

```

c*****
c  SUBROUTINE mg2d_relax(nits,a,u,rhs,ni,nj)
c  non-vectorised red black Gauss-Seidel relaxation
c  for a general constant 5 point stencil. Dirichlet
c  Boundary conditions
c  Variables:
c      nits: number of red-black iterations
c      a: 5 point operator stencil
c  u, rhs: solution and right hand side (assumes premult by h^2)
c  ni,nj: horizontal and vertical dimension of u, rhs
c*****
subroutine mg2d_relax(nits,a,u,rhs,ni,nj)
implicit none
integer ni,nj,nits
real a(5),rhs(ni,nj),u(ni,nj)
integer i,ipass,j
integer im,ip,jm,jp,rb,m,n

rb(m)=m-2*((m/2))      ! red-black toggle

c
c  do red-black gauss-seidel relaxation for nits
c
do n=1,nits
do ipass=0,1          !do red then black
do j=2,nj-1
do i=2+rb(ipass+1+j),ni-12
do im=i-1
do ip=i+1
u(i,j)=a(1)*(rhs(i,j)-(a(2)*u(im,j)+a(3)*u(ip,j)+a(4)*u(i,jm)+a(5)*u(i,jp)))
enddo
enddo
enddo
enddo

```

```

        enddo
      enddo
    enddo
  return
end

```

Figures 9.9 and 9.10 illustrate schematically how the storage scheme and local grid operators work in implementing a single vcycle on a 4-level grid. This scheme is efficient (and rather elegant) in use of storage etc. Additional composite arrays can be added easily.

9.5.4 A note on Boundary conditions in multi-grid

Boundary conditions are the curse of numerics but they are also often the most important part. For multi-level schemes implementation of boundary conditions may seem particularly confusing (and is often neglected in discussions of techniques) because boundary conditions must be enforced on all of the different levels as well as for the restriction operator.⁸ Personally, I've been through about 4 different ways of implementing boundary conditions but I think I have a fairly stable, easy to implement approach that will handle at least the most standard boundary conditions. Because iterative schemes are identical in operations to time-dependent problems, these boundary schemes can also be used in time-dependent update schemes. The basic idea is that when we are on an edge we have to specify what to do with the point that lies outside the grid. For example, if we want a reflection boundary on the left edge $i=1$ then if our stencil operation would want $im=i-1=0$ we would replace it with $im=2$. Likewise, for periodic boundaries we would set $im=ni-1$ and for dirichlet boundaries we would skip over the $i=1$ edge altogether and start with $i=2$. More generally, a standard 5-point star operation looks like

$$u(i, j) = a(1) * (rhs(i, j) - (a(2) * u(im, j) + a(3) * u(ip, j) + a(4) * u(i, jm) + a(5) * u(i, jp)))$$

where for an interior point $im=i-1$, $ip=i+1$, $jm=j-1$, $jp=j+1$. So all we really need to do is redefine im , ip , jm , jp appropriately when we are on an edge or corner. To do this in a general way, I pass in a small array $iout(2, 2, ngrids)$ which gives the value of the outside index for each of the sides and directions. The storage is

$iout(plus_minus, direction, grid)$ where $direction=1$ is the i direction of the grid $direction=2$ is the j direction (and so on for k in 3-D). and $plus_minus=1$ is the edge in the $direction-1$ direction (oy!) and $plus_minus=2$ is the edge in the $direction+1$ direction. To make this simpler the left edge on grid g is $iout(1, 1, g)$, the right edge is $iout(2, 1, g)$; bottom is $iout(1, 2, g)$ and top is $iout(2, 2, g)$. If $iout$ is negative we assume the edge is dirichlet and we skip the edge. Here is a general relaxation scheme that implements this type of boundary condition information. Study it and compare it to the hard-wired dirichlet relaxer.

```

c*****
c      SUBROUTINE mg2d_relaxc(nits,a,u,rhs,ni,nj,iout)
c non-vectorised red black Gauss-Sidel
c Variables:
c          nits: number of red-black iterations

```

⁸Boundary conditions are not required for interpolation because interpolation on the boundary only requires information on the boundary.

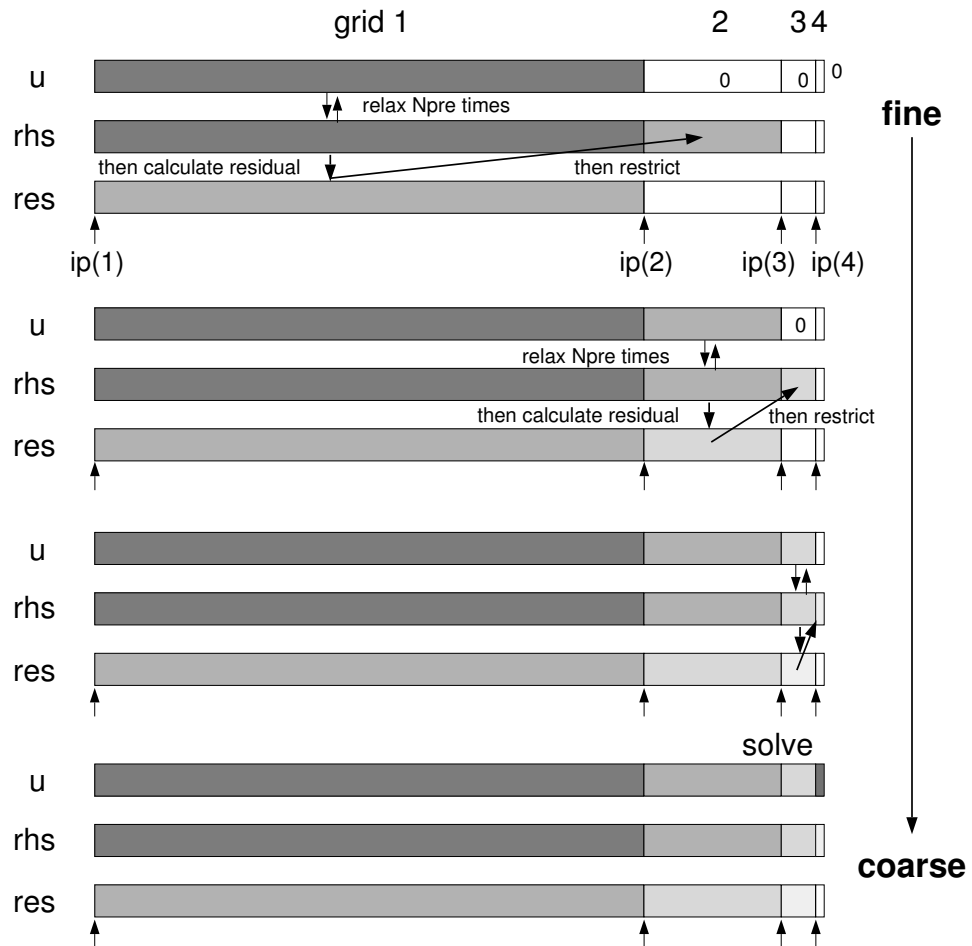


Figure 9.9: Schematic of how vcycle works in the Briggs storage scheme. This figure just shows the coarsening leg of the Vcycle and the coarse-grid solve. At the initial time, $u(ip(1))$ holds our initial guess and all the rest of the array is set to zero. At this time only $rhs(ip(1))$ contains the right-hand side for the finest grid. After a few relaxation steps, the fine grid residual is calculated in $res(ip(1))$ and restricted to $rhs(ip(2))$. The coarse grid correction e^{2h} is approximated by relaxation and stored in $u(ip(2))$ and the process is repeated up to the coarsest grid where the correction to the grid 3 correction is solved exactly (or by just lots of relaxations).

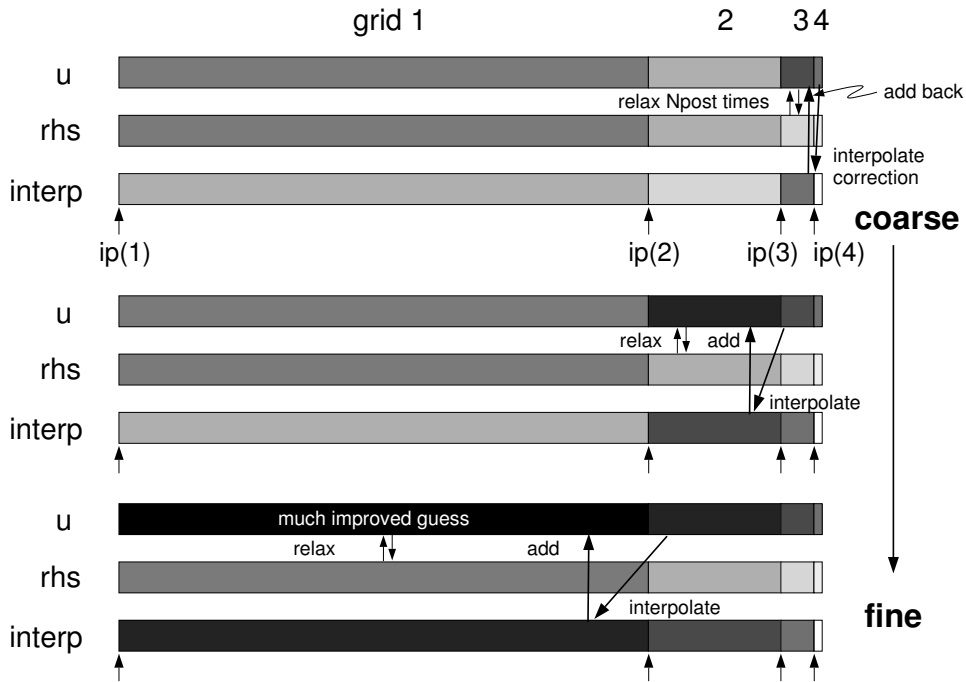


Figure 9.10: Coming back down. This figure shows the fine to coarse leg of the V-cycle. Given the solved coarsest correction in $u(ip(4))$, it is interpolated onto the storage of $res(ip(3))$ then added to $u(ip(3))$ which is then relaxed against $rhs(ip(3))$ which has not been changed from the up-stroke. The processes is repeated until we return to the finest grid where we can repeat the procedure by just zeroing out the corrections on $u(ip(2))$ to $u(ip(ng))$. Note that rhs is not changed on the down-stroke and that the interpolated correction just overwrites the residual array.

```

c          a: 5 point operator stencil
c u, rhs:  solution and right hand side (assumes premult by h^2)
c ni,nj:  horizontal and vertical dimension of u, rhs
c iout:   flags denoting boundary offsets
c*****
      subroutine mg2d_relax(nits,a,u,rhs,ni,nj,iout)
      implicit none
      integer ni,nj,nits
      real a(5),rhs(ni,nj),u(ni,nj)
      integer iout(2,2),ioff(2,2)
      integer i,ipass,j
      integer is,ie,js,je
      integer im,ip,jm,jp,rb,m,n

      rb(m)=m-2*((m/2))          ! red-black toggle

      do j=1,2                  !set up dirichlet boundaries if iout<0
        do i=1,2
          if (iout(i,j).lt.0) then
            ioff(i,j)=1
          else
            ioff(i,j)=0
          endif
        enddo
      enddo
      is=1+ioff(1,1)
      ie=ni-ioff(2,1)
      js=1+ioff(1,2)

```

```

        je=nj-ioff(2,2)
c
c----is,ie,js,je are the starting and ending bounds for the grid, is=1
c----if non-dirichlet, otherwise is=2 etc.
c
c
c      do red-black gauss-seidel relaxation for nits
c
c      do n=1,nits
c        do ipass=0,1          !do red then black
c          do j=js,je
c            jm=j-1
c            jp=j+1
c            if (j.eq.1) then
c              jm=iout(1,2)
c            elseif (j.eq.nj) then
c              jp=iout(2,2)
c            endif
c            do i=is+rb(ipass+1+j),ie,2
c              im=i-1
c              ip=i+1
c              if (i.eq.1) then
c                im=iout(1,1)
c              elseif (i.eq.ni) then
c                ip=iout(2,1)
c              endif
c              u(i,j)=a(1)*(rhs(i,j)-(a(2)*u(im,j)+a(3)*u(ip,j)+a(4)*u(i,jm)+a(5)*u(i,jp)))
c            enddo
c          enddo
c        enddo
c      enddo
c    enddo
c  return
c  end

```

Note, with a good compiler, the `if` statements in the `do` loops should be peeled although this may cause minor problems when `is,ie,js,je` are not specified before hand. Another approach is to use the same scheme but use pre-compilers to conditionally compile in specific boundary conditions for specific jobs. Remember the time you save in run time may not be worth the hassle.

9.6 Krylov Subspace Schemes

Multi-grid schemes are currently some of the fastest, best scaling iterative schemes for solving elliptic Boundary Value problems. when implemented correctly they can show remarkable convergence properties. Their largest weakness however is that they require some care in defining coarse grid operators (or even course grid representations) that are faithful to the solution on a coarse grid and doesn't pollute the fine grid solutions. In the case of our model Poisson problem, this isn't an issue because the operators on all levels are effectively the same and the simple relaxation schemes act as filters on what are effectively orthogonal modes. I.e. there is very little interaction between levels, the different grids are just used to efficiently remove the highest frequency modes defined by each grid spacing. When the operator contains coefficients that vary in space (e.g. the permeability in a porous flow problem), it can become problematic to easily define coarse grid operators. One approach is *Algebraic Multi-Grid* which attempts to develop coarse grid accelerators directly from the find grid discretization $\mathbf{Ax} = \mathbf{b}$ without any notion of an underlying grid.

Another approach, however, is to give up and go to more general iterative schemes for sparse matrices such as the *Krylov Subspace Schemes*. These schemes are principally iterative schemes for solving large sparse linear systems for general (but sparse) A . They usually converge and scale more poorly than multi-grid but parallelize more easily, are reasonably general and are the basic solvers available in the PETSc package. Here we will describe the basics of Krylov Subspace schemes and two commonly used schemes GMRES (generalized minimum residual) and CG (Conjugate gradient).

To understand how these schemes work, however, it's useful to review the ideas of vector spaces and subspaces from linear algebra. blah, blah, blah.

9.7 Summary and timing results

Okay so we've now looked at most of the commonly used schemes for boundary value problems (well not conjugate gradient techniques but they're order $N^{1.5}$ schemes anyway), so it's time to put our money where our CPU's are and compare them against each other in terms of solution time and accuracy. For this problem we will compare a direct solver (Y12M), a FACR scheme from FISHPAK (hwsrct), SOR and a constant stencil multigrid scheme (from your's truly) in both V-cycle mode and FMG mode for the $2\text{-}2 \sin(k_x x) \sin(k_y y)$ cell test shown in Fig. 9.4. To compare behaviour for large problems, these tests have been run on a single node of an IBM SP2. Problems were done for square grids with dirichlet boundary conditions and $N = 65^2, 129^2, 257^2, 513^2, 1025^2$ points. Results are shown in Fig. 9.11 and show that for Poisson problems FACR and multi-grid are comparable in timing, scaling and errors. The generally larger times for plain V-cycle Multi-grid are also somewhat misleading because they are trying to solve the problem from an initial guess $\mathbf{u} = 0$ and thus require about 5 V-cycles to converge. If this solution were part of a time dependent problem and we had a much better initial guess, we could probably solve it in 1–2 V-cycles and the solution time would be proportionally smaller by a factor of 2 to 5.

Recommendations If you have a Poisson or Helmholtz problem that can be addressed using the boundary conditions available in FISHPAK, by all means use these routines (also if you have funny coordinate systems like polar coordinates). They are exceptionally fast and stable and are often a quick way to get going on a problem. For more general operators however Multi-Grid still maintains its remarkable convergence behaviour and it is not that difficult to modify the routines to handle more general problems and boundary conditions. In addition, Multi-grid is extremely modular so it is straightforward to replace any of the grid operators (for example changing the relaxation scheme to line relaxation) by just changing subroutines. In this way, Multi-grid is less of an algorithm and more of a strategy. Done correctly it can work wonderfully, done poorly... well what's new. Whatever you do, unless you really need the power of a direct solver, avoid them and for any problem where you would use SOR you should use multi-grid.

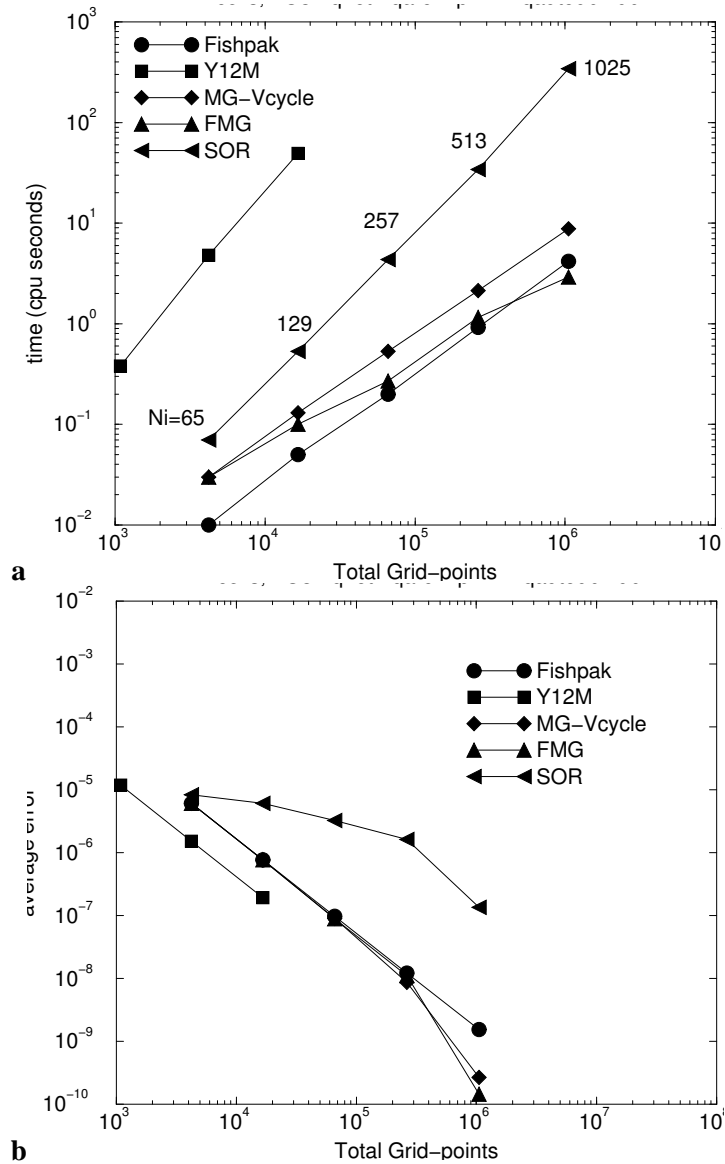


Figure 9.11: Comparison of solution time and true relative error for a bunch of elliptic solvers (direct, FACR, SOR, MG-Vcycle and FMG-Vcycle), Multi-grid techniques use a standard (2,1) V-cycle (i.e. 2 relaxations on the up-stroke and one on the down-stroke). (a) CPU times as measured on a RS6000 390H with 256Mb RAM (compiler flags are -O3 -qarch=pwr2 -qhot -qautodbl=dbl4) (b) average L_2 norm of the error from the true solution (this is actually the truncation error).

Bibliography

- [1] W. Briggs. A Multigrid Tutorial, SIAM, Philadelphia, PA, 1987.