Multigrid Principles

John Millar and John C. Bancroft

ABSTRACT

Multigrid methods are used to numerically solve discrete differential equations. The method is far faster than any other direct or iterative solving method ($\mathcal{O}(N)$ vs. $\mathcal{O}(N^3)$). Multigrid methods work by decomposing a problem into separate length scales, and using an iterative solver method that optimizes error reduction for that length scale, rather than globally. For multigrid to work, several sub-routines must be developed to pass the data from coarse grid to fine grid (interpolation), from fine grid to coarse grid (restriction), and correction of the error at each grid interval (smoothing). While used frequently in areas such as fluid dynamics, it has not yet been seen frequently in literature concerning seismic inversion or modelling. The approach used here is to develop the basic ideas that form the basis of multigrid applications.

INTRODUCTION

In geophysics, there often arises the need to solve partial differential equations (PDE's). It is possible to express these systems of differential equations as a system of linear equations, $\mathbf{Ax}=\mathbf{b}$. As the size of the matrix $\mathbf{A}$ increases, direct matrix solvers tend to take far too long to be practical. Quadratic-form iterative methods (steepest descent, conjugate gradient, etc.) can converge very quickly, however for non-linear problems there is a possibility that the solution converged to is not the absolute minimum of the quadratic form (Shewchuk, 2002). Even for linear applications, the quadratic form iterative methods require strict and specific matrix forms.

Multigrid algorithms are a fast and flexible way to solve the matrix representation of PDE's. Instead of trying to solve the system at full resolution, they typically solve the equations on a coarse grid, and refine the solution to the desired accuracy. The general

principles of multigrid do not change significantly between applications, and its flexibility comes from modularity. For new problems, only the components need to be adjusted, leaving the larger framework intact.

Multigrid methods operate on the concept of decomposition of scale. This scheme is most familiar in the fast Fourier transform. In a discrete Fourier transform, for a signal length containing N samples, it requires $\mathcal{O}(N^2)$ operations to calculate the frequency domain. By dividing the signal in two, and adding the Fourier transform of each half, we can reduce the number of operations to $\mathcal{O}(N^2/2)$. By repeatedly subdividing the original signal, the fast Fourier transform yields a computation time of $\mathcal{O}(N \log N)$, or a saving of $\log N/N$ in operations.

As of yet, very few attempts have been made to use multigrid for seismic problems in the literature. (Wang and Zhou, 1992) use a multigrid algorithm to model seismic waveforms in teleseismic and tomographic exploration of subduction zones. (Shih and Levander, 1985) use a multigrid framework to solve for the spacial derivatives in reverse time migration.

Iterative methods in general do not handle multiple minimums associated with the quadratic form of non-linear equations. Multigrid avoids this problem by first finding the global minimum on a coarse grid (Bunks et al., 1995).

(Gray and Epton, 1990) decompose a recorded wavefield into steeply and shallow dip components, and reserve an expensive wide aperture migration for use only on steeply dipping features. The speed of the migration comes from restricting the number of traces used in the high order algorithms. While not strictly a multigrid method, it is an interesting application of manipulating the scale of a problem.

A detailed description of the various tools that can be used for specific applications is not the goal of this paper. Instead, its purpose is to outline the general concept of multigrid schemes, and provide an intuitive idea of how the method works. For how to use multigrid in more specific applications, see (Trottenberg et al., 2001) or (Wesseling, 1992).

The following section introduces the use of linear algebra to solve a PDE. A brief review of some simple iterative methods for solving linear systems is also given.

The next section starts with an examination of *coarse grid correction*, the most basic

multigrid technique. *Multigrid corrections* and the *full multigrid method* are extended from the theory of coarse grid correction.

Lastly, the individual functions that make up a multigrid code are looked at in slightly more detail. Some general advice on the appropriate choices for individual components is given. Excerpts of a functional multigrid code are included in the appendix.

PDE'S AS LINEAR SYSTEMS

A linear system is often expressed in the forward form as

$$\mathcal{L}\mathbf{u} = \mathbf{b}. \tag{1}$$

The *source term*, $\mathbf{b}$, is a known vector of length N. The known matrix $\mathcal{L}$ is of size $N \times N$. The goal is to find the vector $\mathbf{u}$, in the simpler form,

$$\mathbf{u} = \mathcal{L}^{-1}\mathbf{u} \tag{2}$$

Equations of the form of equation (1) are well understood, and there are a variety of ways to solve them, depending on the exact form of the matrix. Discrete approximations to partial differential equations can be written out in this form as well (Trefethen, 1996, eg.).

Consider the simple approximation for a first derivative of a function $u(x)$,

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x) - u(x-\Delta x)}{2\Delta x}. \tag{3}$$

Lets say that u has been discretized along the length of the domain, at regular intervals. We store the values of u in a vector, $\mathbf{u}$, such that $u_i = u(i\Delta x)$. To access the i^{th} entry of $\mathbf{u}$, we can multiply the column vector $\mathbf{u}$ by a row vector of the form

$$\begin{pmatrix} 0 & \cdots & 1 & \cdots & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_i \\ \vdots \\ u_N \end{pmatrix} = u_i, \tag{4}$$

where the 1 is the i^{th} entry in the row vector. Likewise, we can subtract the $(i-1)^{th}$ value from the $(i+1)^{th}$ value using

$$\begin{pmatrix} 0 & \cdots & -1 & 0 & 1 & \cdots & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_{i-1} \\ u_i \\ u_{i+1} \\ \vdots \\ u_N \end{pmatrix} = u_{i+1} - u_{i-1}. \tag{5}$$

Taking this idea further, we can express the finite difference approximation across all of $\mathbf{u}$ using a sparse matrix equation of the form

$$\frac{1}{2\Delta x} \begin{pmatrix} 0 & 1 & & & & & \\ -1 & 0 & 1 & & & & \\ & -1 & 0 & 1 & & & \\ & & -1 & 0 & 1 & & \\ & & & -1 & 0 & 1 & \\ & & & & -1 & 0 & 1 \\ & & & & & -1 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} \approx \frac{\partial u}{\partial x}. \tag{6}$$

For brevity, we will identify the operators with only one row, and the operator is always centered on the main diagonal. This representation is often called the *kernel*. For instance, the second derivative of $\mathbf{u}$ with respect to x is

$$\frac{\partial^2 \mathbf{u}}{\partial x^2} = \frac{1}{\Delta x^2} \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \mathbf{u}. \tag{7}$$

Often the scalar $1/\Delta x^2$ in front of the kernel is omitted in kernel form for brevity.

To make use of equation (1) when the problem is two dimensional, the solution $u(x, y)$ must still be expressed as a vector $\mathbf{u}$. This requires a re-sorting of the points. The simplest method for discretizing a function of two dimensions into a vector is depicted in Figure 1. Diagrammed is a function discretized into 6 x points and 4 y points. The numbers in the box correspond to the index that the value of $u(x, y)$ will be stored at in $\mathbf{u}$. Notice that to

access the value of $u(x, y + \Delta y)$ relative to $u_i = u(x, y)$ in index notation, use u_{i+n}, where n is simply the number of x grid points.

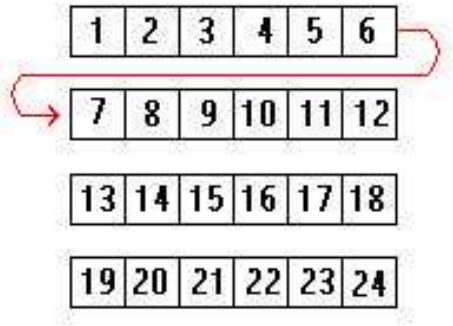


Figure 1: The sorting of a discrete two dimensional function into a vector

The kernel representation of the Laplacian operator in 2 dimensions is of the form

$$\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \Rightarrow \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}. \tag{8}$$

Each row of the matrix representation of the operator will look like

$$\begin{bmatrix} \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \cdots \\ & (i-n) & & (i-1) & (i) & (i+1) & & (i+n) & \end{bmatrix}. \tag{9}$$

where all entries not referred to will be zero. This representation is only accurate for calculating the interior points. Boundary conditions are handled separately and will be discussed later.

Relaxation Methods

For extremely large matrices of these sorts, it is sometimes faster to refine an initial approximation of the solution than it is to solve the equation exactly (Riley et al., 2002, eg.). Relaxation methods are a simple way to reduce the error of the solution. They work by solving for the central value of the operator, and only considering the other values in the kernel.

We now solve Poisson's equation,

$$\nabla^2 u(x,y) = f(x,y). \tag{10}$$

Expressing this problem as a matrix equation of the form in equation 1, we derive the Jacobi relaxation method.

Solving the finite difference approximation to Poisson's equation for $u(x,y)$,

$$u^{k+1}(x,y) = \frac{1}{4}[u^k(x+\Delta x,y) + u^k(x-\Delta x,y) + u^k(x,y+\Delta y) + u^k(x,y-\Delta y)] \tag{11}$$
$$-\frac{\Delta x^2}{4} f(x,y)$$

Where $f(x,y)$ is the source term, and k corresponds to the iteration number. In index notation, using the convention for discretizing a 2d function mentioned earlier,

$$u_i^{k+1} = \frac{1}{4}[u_{i+1}^k + u_{i-1}^k + u_{i+n}^k + u_{i-n}^k] - \frac{\Delta x^2}{4} f_i \tag{12}$$

To get an improved estimate of $u(x,y)$, cycle through all values of i, and update each point in u^{k+1} by the values in u^k. Adapting the Jacobi method (11), so that the new, updated values u^{k+1} are used as they become available, results in the Gauss-Seidel method. It is fairly similar,

$$u_i^{k+1} = \frac{1}{4}[u_{i+1}^k + u_{i-1}^{k+1} + u_{i+n}^k + u_{i-n}^{k+1}] - \frac{\Delta x^2}{4} f_i \tag{13}$$

but yields a slightly faster rate of convergence. The difference between equations (12) and (13) lies in the use of updated values u^{k+1} on the right hand side of equation (13).

There are more advanced iterative methods that converge faster, but are usually not appropriate in multigrid schemes. The reasons for this will be looked at later.

Consider Laplaces equation on a square cartesian grid, with homogeneous boundary conditions and no source terms.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0. \tag{14}$$

The solution to this problem is trivial, $u(x,y) = 0$. This is convenient, because the current estimate of the function will also be the error. Suppose that we didn't know this, and begin with the estimate depicted in the top left of Figure 2 (we are terrible guessers). The three

following axes in Figure 2 show the error after a number of iterations of the Gauss-Seidel relaxation operator (In appendix). It is clear that we are converging to the correct answer of $u(x, y) = 0$. However the rate of convergence seems to be slowing down. The maximum value of the error after 25 iterations is 3.96. After 50 iterations this improves to 2.65. To reduce the norm of the error to beneath 0.1, it takes 1378 iterations. Clearly this is nowhere near fast enough to be used by itself to rapidly produce a reasonable solution. However, it should be noted, that while the long wavelength portion of the error is very difficult to reduce, the high frequency error is dissipated almost instantly.

Instead of specifying a fixed number of iterations, it is preferable to set up a tolerance for the maximum allowable error. An easy way to quantify the error in general, is to take the norm of the change in error, or

$$\Theta^k = \sum_{i=1}^{n} (\varepsilon_i^k - \varepsilon_i^{k-1})^2, \tag{15}$$

where ε is the error, the superscript k is the iteration number, and i is the variable index. The calculation will stop when Θ^k falls below a specified value, usually near the truncation error. This ensures that the calculation will not take longer than necessary, and will also provide an approximate solution within a predetermined limit.

THE MULTIGRID METHOD

At the very root of multigrid methods is the high frequency dampening effect of relaxation methods. If the error has a high frequency component, as compared with the grid spacing, then the relaxation will attenuate it quickly. For the long wavelength errors, the relaxation methods will not work as well. What the multigrid method does is re-define the problem on a more coarse grid. The relaxation methods will be more effective on the long wavelength error at the coarse grid spacing. Once the coarse grid solution has been reasonably approximated, the solution is interpolated to a finer grid. The interpolated values do introduce an error of about the same wavelength of the grid. Further passes of the relaxation method correct the interpolated points.

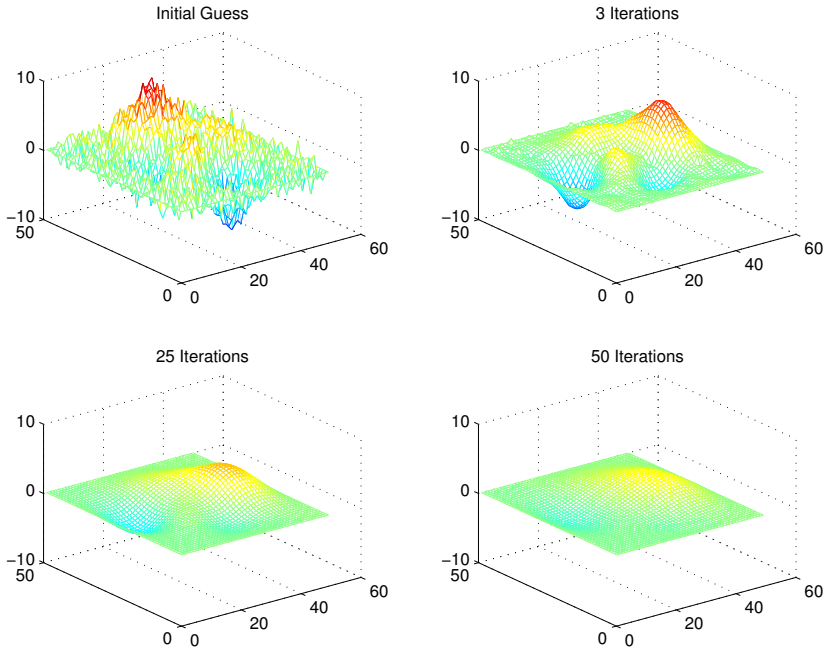


Figure 2: The Error performance of a Gauss-Seidel Iterative method on Laplaces equation.

To discuss multigrid methods it is important to review some terms that will appear. *Restriction* refers to *coarsening* the grid of the solution. The most simple restriction operator is to reject every other grid point. As it turns out, this is not a good way to perform restriction, but for visualization of the concept it is adequate. It is often required in multigrid methods to interpolate the coarse grid onto a finer grid. This action is also known in much of the literature as *prolongation*. Again, there are many different prolongation methods, some of which will be discussed later. For now, what is illustrated by Figure 3 is adequate in terms of understanding the concepts discussed here. The application of a relaxation method is also called *smoothing*. The term smoothing is more accurate than relaxation, as not all relaxation operators are suitable for multigrid methods. These distinctions will be clarified later.

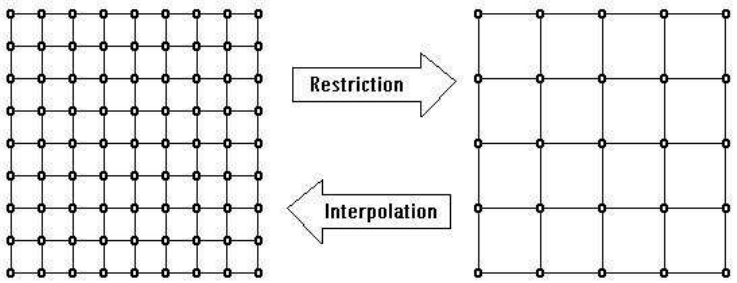


Figure 3: Diagrammatic explanation of restriction and interpolation.

Coarse Grid Correction

Coarse grid correction is an iterative 2-grid method, as well as being a good instructional tool to understand how more complicated multigrid methods work. We need to define more precisely some of the variables involved. We redefine equation 1,

$$\mathcal{L}u = f. \tag{16}$$

In general, $\mathcal{L}$ can be any linear partial differential operator. The functions u and f are the solution and the source terms respectively. The finite-difference approximation on a fine grid h, can be written as

$$\mathcal{L}_h u_h = f_h. \tag{17}$$

An estimate, $\tilde{u}_h$, of the solution is needed. The error in this estimate is simply

$$\varepsilon_h = u_h - \tilde{u}_h. \tag{18}$$

The *defect* is given by

$$\delta_h = \mathcal{L}_h \tilde{u}_h - f_h. \tag{19}$$

When $\mathcal{L}$ is linear, this satisfies

$$\delta_h = -\mathcal{L}_h \varepsilon_h. \tag{20}$$

Every one of these functions and operators has a corresponding function or operator on the coarser grid H. To change between the grid sizes, we define the prolongation (interpolation) operator $\mathcal{P}$ and restriction operator $\mathcal{R}$ such that

$$\mathcal{R} u_h \Rightarrow u_H, \tag{21}$$

$$\mathcal{P} u_H \Rightarrow u_h.$$

For simplicity of notation, we will also refer to the smoothing operator $\mathcal{S}$ in similar manner,

$$\mathcal{S} u_h \Rightarrow u_h^{new}.$$

The coarse grid correction starts with an initial guess on the finer grid, $\tilde{u}_h$. The first step is to compute the defect δ_h on the finest grid using equation (19). Next, the defect is restricted down to the coarse grid using $\delta_H = \mathcal{R}\delta_h$. On the coarse grid, we solve equation (20) for the error ε_H. This can either be done exactly, using a direct method such as Gaussian elimination, or using the same relaxation scheme that you are using as the smoother. Once the error ε on the coarsest grid is known, interpolate the error up to the finer grid spacing, smooth it to reduce the interpolation error, and apply it to the initial solution as a correction.

The 2-grid correction is more computationally efficient than standard relaxation for two reasons. First, the long wavelength error is reduced more quickly on the coarse grid, so that there is a lower number of iterations necessary to reduce the error. Secondly, the smoothing operations performed on the smaller vector take less time, as there are less points to update. Table 1 is a general 2-grid algorithm. The form of the smoother depends on the equation being solved. If using a smoother for step 5, the relaxation of ε uses the defect δ as the source term.

To evaluate the performance of the 2-grid method, we time how long it takes for convergence within a specified limit, and compare it to straight relaxation methods. For the same initial guess and smoother in the sample problem used in the section on relaxation methods, (depicted in figure 2), The Gauss-Seidel relaxation and a basic 2-grid algorithm were iterated over until convergence was met. Step 5 in the 2-grid algorithm was performed by a relaxation method, as the matrix operator $\mathcal{L}$ was nearly singular, and an exact solution was unstable.

The straight relaxation method for a 129×129 grid, (16641 equations in 16641 unknowns) took over 18 minutes of run time to reduce the error to beneath the specified tolerance. Using a 2-grid method, where the distance between the coarse grid points was twice that of the fine grid points, yielded a calculation time of 73 seconds. When the grid was further reduced to 4 times the original grid separation the computational cost was reduced to 23 seconds. At 8 times the separation the calculation time was 17 seconds. Further reductions in resolution did not realize much faster compute times, as the medium wavelength errors were left unrepresented on the coarse grid, and too smooth to be effiently reduced on the fine grid.

Multigrid Correction

The simplest way to view multigrid correction is to look at the algorithm for the 2-grid correction. The only difference is that in the algorithm step number 5 of the 2-grid correction (table 1), the equation to solve for the error, $\mathcal{L}\varepsilon = -d$, is itself solved by its own multigrid method. Instead of solving the initial problem, a solution to the error equation (20) is sought.

Any smooth error will be passed in the defect down to a coarser grid where it will have a reduced wavelength relative to the grid spacing.

The formula for passing the defect down to coarser grid is

$$d_H = \mathcal{R}(\mathcal{L}_h \varepsilon_h - \delta_h). \tag{22}$$

The first defect is found using formula (19), then passed down to the next coarsest grid using (22). The solution is improved by one or more passes of the smoother. The restricted defect is used as a source term, and the initial guess for ε should be zero at the new level. The defect of the error equation (20) is again restricted and moved down to the next coarser grid, and the process repeated until the coarsest grid is reached. One cycle of the coarsening looks like

$$d_H = \mathcal{R}(\mathcal{L}_h \varepsilon_h - \delta_h), \tag{23}$$

$$\varepsilon_H = \mathcal{S}\varepsilon_H$$

$$d_{H2} = \mathcal{R}(\mathcal{L}_H \varepsilon_H - \delta_H) \cdots$$

The exact solution of (20) is found on the coarsest grid using a direct method or repeated relaxations. Once an accurate estimate of the coarse error is found, the error is interpolated, and added to the error on the next finer grid. The interpolation operator itself introduces a small high frequency error that can easily be reduced using several passes again of the smoothing function. After the interpolation noise has been reduced, the process is repeated,

$$\varepsilon_h = \varepsilon_h^{old} + \mathcal{P}\varepsilon_H, \tag{24}$$

$$\varepsilon_h = \mathcal{S}\varepsilon_h$$

$$\varepsilon_{h2} = \varepsilon_{h2}^{old} + \mathcal{P}\varepsilon_h \cdots$$

The speed of convergence comes from running the smoother at each interpolation and restriction. The high frequency error is smoothed, leaving the low wave length parts of the error until later on the calculation, on a coarser grid. Unlike the 2-grid algorithm, multigrid relaxations provide the maximum error reduction at all wavelengths in the solution. However,

the cost of this is needing to keep track of data at multiple grid spacings. Tips on doing this will be dealt with later.

Using a multigrid correction, the error in the Laplacian sample problem, equation (14) that took over 18 minutes using Gauss-Seidel relaxation, was brought to beneath the same error tolerance in approximately 7 seconds.

Full Multigrid

Full multigrid is more of a direct solver than an iterative method. It uses the multigrid correction as one of its own subroutines, and is slightly more sophisticated.

Rather than starting with an approximate solution on the finest grid, the method starts with an exact solution on the coarsest grid. From there it interpolates the solution to a finer grid. The solution is then refined using the multigrid correction. The solution is interpolated, then corrected until it has reached the desired grid spacing and accuracy for the final answer.

Figure 4 depicts how the solution takes form. While it may sometimes be necessary to do two iterative corrections per grid step, it is probably sufficient in most cases to only perform 1 correction. If unsure, it is possible to use equation (15) to automatically stop the calculation once the norm of the error has been reduced sufficiently.

A full multigrid solver found the solution to Poisson's equation with properties similar to the previous test samples except that a non-zero source term was used, in 1.76 seconds. The results are in Figure 7. Unlike the multigrid correction, the source term f is required at all mesh sizes in full multigrid. If the source is not homogeneous, then $\mathcal{P}f_H$ may not properly represent f_h (likewise for the restriction of f_h). The safest precaution is to re-discretize the source term on each grid. In Figure 7, the importance in the difference between restricting and rediscretizing the source term is apparent.

PARTS OF MULTIGRID

Once the framework for a multigrid solver is built, in order to allow it to handle different types of problems, with complications such as strong anisotropy, non-linearity, and the

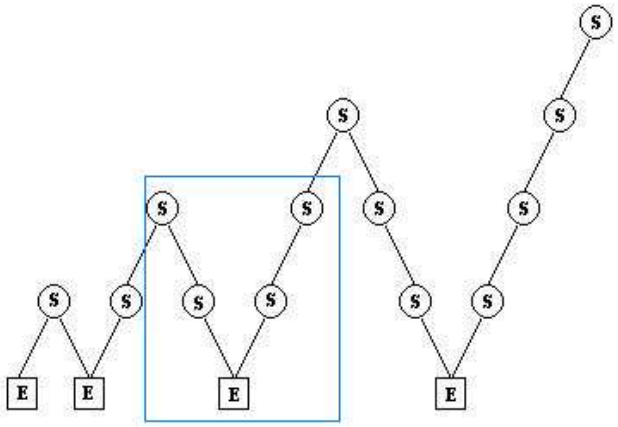


Figure 4: Schematic of data transfer in full multigrid method. S denotes smoothing, E exact solution of equation (20), /'s restriction, \'s interpolation. The defect is passed down while the error is passed upwards. The blue box outlines one application of the multigrid correction

presence of shock fronts, the main program itself may not need to be altered in any way. All that is needed is to update the functions that are called from the main program.

Interpolation

Interpolation is the adding of grid points in between existing grid points. The interpolation in Figure 3 has the effect of reducing Δx and Δy in the problem by half. For some problems, it may be advisable to derive interpolators that depend on direction, or to change the distance between the points by a factor other than 2. Complications such as these will not be dealt with, but the logic and methods are similar.

Bi-linear interpolation is the most common form of interpolation in 2 dimensions, and is depicted in Figure 5. It calculates the value of each new point based on the average of all

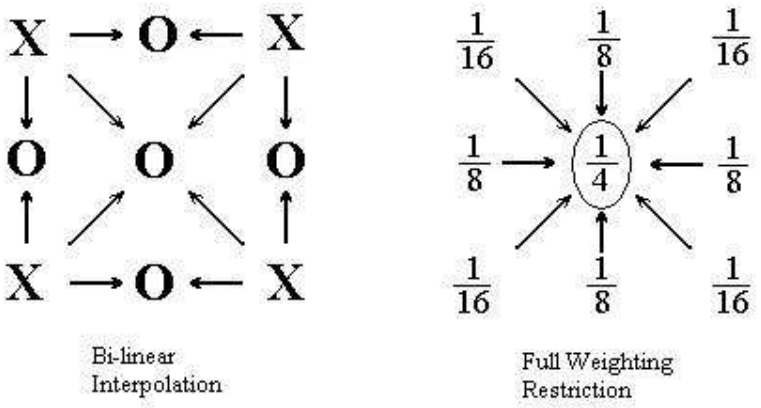


Figure 5: Schematic of Bi-linear Interpolation and full weighting restriction. For interpolation, The **x**'s represent previous values and **o**'s interpolated points, and the arrows shows the contribution from neighboring points. For restriction, only the central point remains, and is a weighted average of all 9 points, using the fractions as weights.

existing neighboring points. The following is an example of bilinear interpolation:

$$\mathcal{P}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & \frac{1}{2} & 1 & \frac{1}{2} & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{25}$$

The non-zero entries of the right hand side are called the *symbol* of the interpolation.

Restriction

Restriction is the opposite of interpolation, as pictured in Figure 3. The most straight forward restriction is called *straight injection*. All grid points that are not wanted in the coarser grid are simply omitted. While simple, it can cause problems in practice. The best choice for a restriction operator is the *adjoint* of the interpolator. A general method to find the adjoint is given in (Press et al., 1992). For bi-linear interpolation, the symbol for the adjoint restriction,

called full-weighting, is

$$\mathcal{R} \Rightarrow \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}. \tag{26}$$

To implement, the restriction consists of taking the weighted average of each point and all of its surrounding points, using the corresponding fractions in the symbol. Note that the restriction of the operators symbol equals 1, or

$$\mathcal{R} \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} = [1]. \tag{27}$$

Bi-linear interpolation and its adjoint, full weighting restriction, are both fairly simple to implement, and will do for most problems. A good rule for deciding if they will give stable solutions is if the following is satisfied:

$$\mathcal{O}(\mathcal{R}) + \mathcal{O}(\mathcal{P}) > \mathcal{O}(\mathcal{L}) \tag{28}$$

The orders of bi-linear interpolation and full weighting are both 2, so they should be sufficient providing the differential equation is of order 3 or less. If the $\mathcal{O}(\mathcal{L}) > 3$, cubic and, if necessary, quartic interpolators can be found in many finite element books (eg. (Hunter and Pullen, 1997 2003)).

Smoothing

Choosing a smoother

It was alluded to before that some iterative solvers and relaxation schemes result in practical difficulties for implementing them in multigrid solvers. The use of the word *smoother* is a clue to the properties desirable for relaxation in multigrid. For a smoother to work properly in a multigrid algorithm, the necessary property is that it damps the high frequency error. Conjugate gradient methods typically reduce the norm of the error greatly with each iterative

pass. However, it is possible that, while the total error may be decreasing, individual frequencies of the error may actually be increasing. For details, see (Shewchuk, 2002). Successive overrelaxation (SOR) is another relaxation method that is closely related to Gauss-Seidel iterations. The difference is that the correction term derived from the classic Gauss-Seidel is amplified by a factor $1 \leq \omega \leq 2$. The convergence for SOR is far faster than that of straight Gauss-Seidel, but again the smoothing property is what is required for a multigrid application. It is possible to optimize the smoothing properties of SOR, using $0 \leq \omega \leq 1$. In (Trottenberg et al., 2001) the value $\omega = 0.8$ appears to be the preferred choice.

In general, Gauss-Seidel smoothers seem to be the most stable and reliable. While fancy smoothers may work more efficiently in specialized cases, these applications tend to be more specialized, and require far more effort for minimal gains in computational time.

Constructing a smoother

The form of the smoother can be found from the kernel representation of the linear operator. The best way to assemble a smoother is to break down the differential equations into its individual kernels, then add them together.

Consider a 2-D convection-diffusion problem,

$$\mathcal{L}u = -\alpha(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2})u + a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y}, \tag{29}$$

where α is the ratio of the effect of diffusion relative to the effect of convection. If $\alpha \gg 1$, the equation is elliptic, If $\alpha \ll 1$, the equation becomes hyperbolic. For now we also assume that

$$\frac{\Delta x}{\alpha}|a| \leq 2, \tag{30}$$

and similar conditions for Δy and b, for reasons that will be discussed later.

Writing each partial derivative in its kernel form,

$$\mathcal{L} = -\frac{\alpha}{\Delta x^2}\begin{bmatrix} 1 & -2 & 1 \end{bmatrix} - \frac{\alpha}{\Delta y^2}\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} + \frac{a}{2\Delta x}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} + \frac{b}{2\Delta y}\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}. \tag{31}$$

As a brief reminder on notation, the index k refers to the iteration number. The i index points to the position in the vectorized solution. The $i \pm n$ indices reference the $u(x, y \pm \Delta y)$ values.

To find the smoother, add all of the individual parts together,

$$\mathcal{L} = \begin{bmatrix} & \frac{-b\Delta y/2-\alpha}{\Delta y^2} & \\ \frac{-a\Delta x/2-\alpha}{\Delta x^2} & \frac{-2\alpha}{\Delta x^2}+\frac{-2\alpha}{\Delta y^2} & \frac{a\Delta x/2-\alpha}{\Delta x^2} \\ & \frac{b\Delta y/2-\alpha}{\Delta y^2} & \end{bmatrix}, \tag{32}$$

then solve for the central value in the kernel. The Gauss-Seidel smoother associated with equation (29) is

$$\Big(\frac{-2\alpha}{\Delta x^2}+\frac{-2\alpha}{\Delta y^2}\Big)u_i^{k+1} = \Big(\frac{-a\Delta x/2-\alpha}{\Delta x^2}\Big)u_{i-1}^{k+1} + \Big(\frac{a\Delta x/2-\alpha}{\Delta x^2}\Big)u_{i+1}^{k} + \Big(\frac{-b\Delta y/2-\alpha}{\Delta y^2}\Big)u_{i-n}^{k+1} + \Big(\frac{b\Delta y/2-\alpha}{\Delta y^2}\Big)u_{i+n}^{k} + f_i. \tag{33}$$

Equation (30) is called the *Peclet* condition. If this condition is not fulfilled, then the central difference operator will not provide a stable or accurate approximation to the first derivative. Unless a and b are both extremely small, the previous discretization will fail, as the condition must apply for all grids used in multigrid, including the most coarse grid. The way around this is to use an upwind scheme. The derivative is approximated using either forward or backward differencing, depending on the direction of convection. For positive convection, a backwards difference is used. The following prescription for an upwind first derivative will automatically use the upwind formula. In kernel form it reads

$$\frac{\partial}{\partial x} = \frac{1}{2\Delta x}\begin{bmatrix} -a-|a| & 2a & a-|a| \end{bmatrix}. \tag{34}$$

While straightforward to implement, it should be noted that this discretization is only $\mathcal{O}(\Delta x)$ accurate. In practice it is advisable to use a higher order upwind scheme for problems of this type.

Implementing a smoother

Often more important than the choice of smoother, is the order in which the grid points are updated. The smoothing properties are greatly improved with subtle changes to the

algorithms. This is analyzed in agonizing detail in (Trottenberg et al., 2001).

For the most general case, rather than proceeding uniformly along the vector, it is far better to update all the even points in the solution, then all of the odd indexes. This is called a *Red-Black* sweep. The name comes from the appearance that, were the all of grid points in the domain coloured like a checker board, all of the red points would be smoothed first, followed by all of the black points.

For convective problems, the convergence is improved if the smoother proceeds through the vector *down-wind*, or in the same direction as the convection.

In the case of strong anisotropy, variables are more intimately linked in one direction relative to another. In this case, it may be advisable to update entire lines at once. Line relaxation results in a tri-diagonal system of equations, which is a special form of sparse matrix system than can be solved very quickly using a modified LU decomposition and back substitution. The computational cost is no more than a pointwise relaxation. Convergence can be sped up using zebra-stripe alternation (analogous to red-black alternation).

Boundary conditions in the smoother

Handling boundaries in a multigrid solver is done in both the smoother and in the calculation of the residual. The behavior of the function on the boundaries must be specified in some manner such as

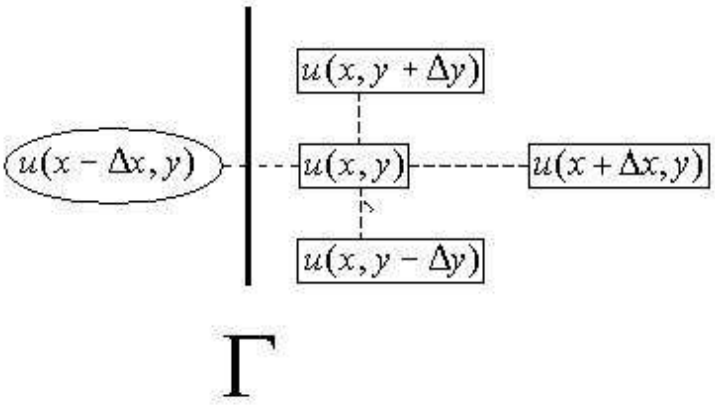
$$u^\Gamma = f^\Gamma, \tag{35}$$

or

$$\frac{\partial u^\Gamma}{\partial \mathbf{n}} = f^\Gamma,$$

or some combination of these two, where $\mathbf{n}$ is the outward normal of the boundary. Homogeneous boundary conditions ($u^\Gamma = 0$) are handled very naturally by most smoothers. When the kernel of the smoother extends beyond the boundary, the value at that point can be simply input as 0, or just left out of the calculation. For free surface conditions ($u'^\Gamma = 0$), the value of the function u on the first interior point (the center of the kernel) can be copied to the exterior point, and the calculation proceed as per usual. For more complicated boundaries

(1-way wave equations, periodicity, general functions) the exterior point can be calculated explicitly, then input into the calculation as if it were an interior point. See figure 6. If more accuracy is needed, the process of updating interior and boundary points can be iterative.



`Figure 6:` In smoothers and for calculating residual, the point outside the boundary Γ can be calculated explicitly and separately, then returned into the smoother/residual as if it were an interior point

Residuals

Once the kernel for the smoother is found, calculating the defect is simple. Move all the terms to one side, and the other side is the defect. For the convection diffusion problem (29), the residual is given by

$$\Big(\frac{-2\alpha}{\Delta x^2} + \frac{-2\alpha}{\Delta y^2}\Big)u_i - \Big(\frac{-a\Delta x/2 - \alpha}{\Delta x^2}\Big)u_{i-1} - \Big(\frac{a\Delta x/2 - \alpha}{\Delta x^2}\Big)u_{i+1} - \Big(\frac{-b\Delta y/2 - \alpha}{\Delta y^2}\Big)u_{i-n} - \Big(\frac{b\Delta y/2 - \alpha}{\Delta y^2}\Big)u_{i+n} - f_i = \delta_i. \tag{36}$$

Notice the similarity between this and the smoother (equation 32) for the same problem.

Moving Data around

The general theory of implementing multigrid solvers for PDE's is contained in the previous sections. However, some subtle tips on coding the solvers should be mentioned.

The first point is that modularity is key. Even if the method is not going to change much for different applications, it is far easier to de-bug smaller parts. If, for instance, it is deemed that a higher order interpolator is needed, it is more practical to write a new interpolator and change the function calls in a script file then to redo the whole method. For many problems, only the smoother and the residual need to change. Identical scripts with calls to different smoothers can solve many problems in the same coordinate system.

For the size of the grid, it is convenient to use $n = 2^m + 1$ number of grid points in each direction. When the interpolator discussed earlier is used, the new number of grid points is $n = 2^{m+1} + 1$ in each direction. When $m = 1$, the number of grid points is 3, which is a good choice for the coarsest grid to start. It is very convenient to use loops with m as an index.

The full multigrid method requires that residuals and errors be stored at all levels in the calculation. Rather than trying to keep track of a bunch of vectors of different size, it is easiest to have one vector store all the necessary information, and write a small subroutine that will return the index of the vector where the pertinent information is to be found. Details on how to do this are best left for an example. The Matlab code in the appendix demonstrates one way to store the data, another way is used in (Press et al., 1992).

CONCLUSION

Re-writing partial differential equations as linear matrix operators allows for a great flexibility in the methods available to solve them. Implicit and semi-implicit methods all require the solution of a matrix equation, and are inherently more stable and accurate than explicit methods. Unfortunately, in general, direct solutions of the matrix equation are computationally expensive and may be unstable due to near singular matrices generated by some boundary conditions.

Multigrid algorithms take $\mathcal{O}(N)$ operations to solve a matrix to within truncation error.

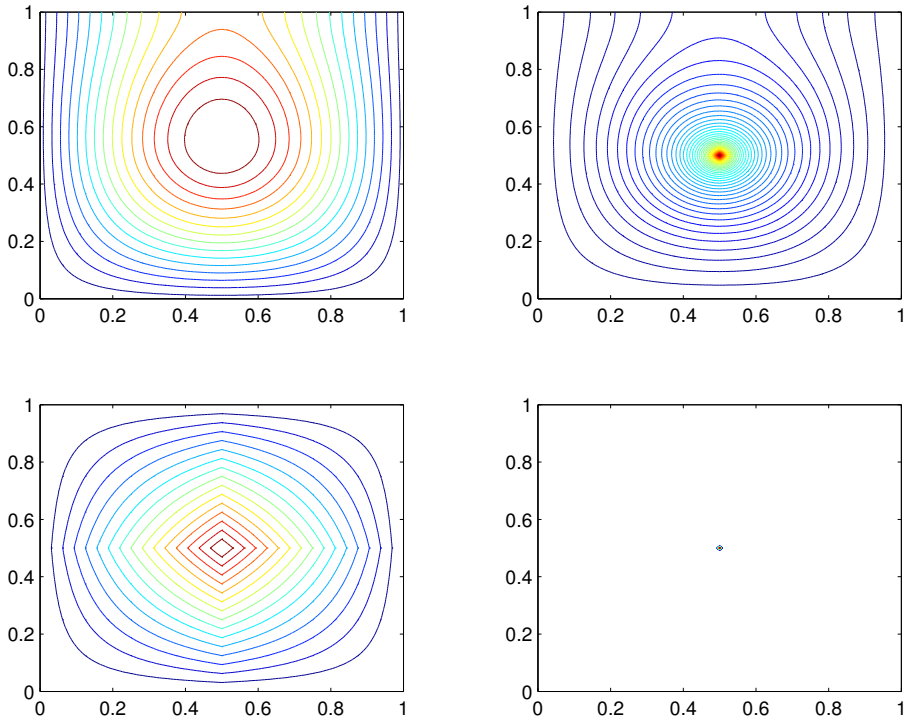


Figure 7: Full Multigrid Solutions to Poissons equation, with homogeneous boundaries, except at the top which has free surface type boundary. Top left is using the prolongation of the initial source (bottom left), where the top right uses rediscretizing of the source term (bottom right). The source in the bottom right is a barely visible single point at the center.

This is far faster than any other iterative or direct method. Coupled with this high speed, is a flexibility allowing multigrid to handle all sorts of problems that are out of the scope of many other fast iterative solvers. Once a general algorithm is written, only minor changes are needed to adapt it to new problems.

Using a multigrid method to solve any PDE yields great savings in calculation time. The application of these methods to seismic problems is still in its infancy, but the relative saving in compute time, and the ease with which parallel computing methods may be incorporated, make multigrid an extremely attractive option for seismic modelling and inversion.

1 References

Bunks, C., Saleck, F., Zalesky, S., and Chavent, G., September-October 1995, Multiscale seismic waveform inversion: Geophysics, **60**, no. 5, 1457–1473.

Gray, S. H., and Epton, M., July 1990, Multigrid migration: Reducing the migration aperture but not the migrated dips: Geophysics, **55**, no. 7, 856–862.

Hunter, P., and Pullen, A., 1997-2003, The finite element/boudary element notes: Department of Engineering, University of Aukland.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., 1992, Numerical recipes in c: Cambridge University Press, 2nd edition.

Riley, K. F., Hobson, M. P., and Bence, S. J., 2002, Mathematical methods for physics and engineering: Cambridge University Press, 2nd edition.

Shewchuk, J. R., 2002, An introduction to the conjugate gradient method without the agonizing pain: unpublished.

Shih, R. C., and Levander, A. R., 1985, Multi-grid reverse-time migration: American Geophysical Union, fall meeting.

Trefethen, L. N., 1996, Finite difference and spectral methods for ordinary and partial differential equations: unpublished.

Trottenberg, U., Oosterlee, C., and Schűller, A., 2001, Multigrid: Academic Press.

Wang, H., and Zhou, H., 1992, Multi-grid inversion and noise suppression of isc traveltimes around west pacific subduction zones: American Geophysical Union, spring meeting.

Wesseling, P., 1992, An introduction to multigrid methods: John Wiley & Sons.

Full Multigrid Code, in Matlab

The following is a Matlab code of the full multigrid algorithm. The FMG and the multigrid correction are displayed verbatim in matlab code. Some of the subfunctions have been greatly abbreviated, and much of the error checking has been removed for clarity. Also,

- m is the grid level, ie. $m = 1$ is the coarsest grid.
- *u_calc* stores $[\varepsilon(m = 1), \varepsilon(m = 2)...\varepsilon(m = 6), u(m = 7)]$
- *b_calc* stores $[\delta(m = 1), \delta(m = 2)...\delta(m = 6), f(m = 7)]$ where f is the source term.
- Whenever possible, h values refer to fine grids, H to coarse
- ni is a vector that stores the starting index of u and b in *u_calc* and *b_calc*, as referenced by m.
- Subroutines starting with **mg** operate only on u and b at the one (input) level, and receive as input only a subset of *u_calc* and *b_calc*. Subroutines starting with **fmg** operate on the entire vectors *u_calc* and *b_calc* (exception **fmg_find_int**, which returns the index used in other **fmg** calls).
- **mg_restrict2** and **mg_interp2** were written for a general number of x points and y points, so don't follow the regular m pattern of the other functions.
- The boundaries are homogeneous, except for the top boundary which is a free surface type condition. The solution is shown in figure 7.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 function [u] = fmg_fullscript
%
%  Solves poissons Equation with a single source
%    at center of grid, using Full Multigrid Algorithm
%

u=zeros(9,1);    % initialize coarsest grid
b=zeros(9,1);b(5)=-1; % source term on coursest grid

for i =1:9
    u= mg_gsd_2(u,b,1,0.5)   %solve coarsest grid using gauss siedel relaxation.
end


u = mg_interp2(u,3);  %interpolate solution to next finest grid

b = zeros(length(u),1); % re discretize source term,
b(13)=-1;               %


for m = 2:6
    u = mg_corr2(u,b);      % send interpolated value to multigrid correction
    u = mg_interp2(u,2^m+1); % interpolate to next grid spacing
    b = zeros(length(u),1);  % rediscretize source term
    b(round(length(u)/2))=-1;% ^^^

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [u]=mg_corr2(u,b)
%
%   Returns Corrected vector in u.
%   Needs source vector b

nx=sqrt(length(u));  % Number of X points
m=log2(nx-1);        % m value

%%%%  Set up empty vectors for calculation
ind=fmg_find_ind(m,2); %length of calc vectors
u_calc= zeros(ind-1+nx^2,1); b_calc= zeros(ind-1+nx^2,1);

% insert finest grid guesses into calc vectors
u_calc([ind:length(u_calc)])=u; b_calc([ind:length(u_calc)])=b;

for m_ind = 1:m+1
    ni(m_ind)=fmg_find_ind(m_ind,2); %vector of first indexes referenced by m
end


for k = 1:1 % pre smoothing
   u_calc([ni(m):ni(m+1)-1]) = gsd_2(u_calc([ni(m):ni(m+1)-1]),b_calc([ni(m):ni(m+1)-1]),m,1/(2^m));
end                          %  gauss-seidel smooth fine grid solution
```

```
for k=m:-1:2   %downward V
    for kk=1:3
        u_calc([ni(k):ni(k+1)-1]) = ...   %pre smooth
            mg_gsd_2( u_calc([ni(k):ni(k+1)-1]) , b_calc([ni(k):ni(k+1)-1]) , k , 1/(2^k) );
    end       % pre smooth error on grid k

    b_calc = fmg_resid_2(u_calc,b_calc,k,1/(2^k));
              %calculate residual, store it in k-1 grid of b
end

for k = 1:20
    u_calc(1:9) = mg_gsd_2(u_calc(1:9),b_calc(1:9),1,0.5);
end       %exact solution on coarsest (3x3) grid


for k= 2:m    % upward V
    u_calc = fmg_adderr_2(u_calc,k); % add error to error on k+1

    for kk=1:10
        u_calc([ni(k):ni(k+1)-1]) = ...   % smooth error on k using defect on k as source
            mg_gsd_2( u_calc([ni(k):ni(k+1)-1]) , b_calc([ni(k):ni(k+1)-1]) , k , 1/(2^k) );
    end
end


for k = 1:1 % post smoothing
   u_calc([ni(m):ni(m+1)-1]) = gsd_2(u_calc([ni(m):ni(m+1)-1]),b_calc([ni(m):ni(m+1)-1]),m,1/(2^m));
end   %smooth final solution


u=u_calc([ni(m):ni(m+1)-1]);
% put solution in output vector


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [u]= mg_gsd_2(u,b,m,dx)
%
%  Performs 1 iteration of the Gauss-Siedel smoother on
%     u with b as source term
%%%%%%%%%
%  Homo Neumann BC inserted on y_max
%%%%%%%%%


nx=(2^m+1); N=nx^2; ind=1; b=0.25*dx*dx*b;

%top left
x(ind)= 0.25*(x(ind+1)+...
    x(ind+nx))-b(ind);

for i = 2:nx-1  %top mid
    ind=ind+1;
    x(ind)= 0.25*(x(ind-1)+x(ind+1)+...
        x(ind+nx))- b(ind);
```

```
end

ind=ind+1;  %top right
 x(ind)= 0.25*(x(ind-1)+...
    x(ind+nx))- b(ind);

for i=1:nx-2
    ind=ind+1;    %left side
    x(ind)= 0.25*(x(ind+1)+...
        x(ind-nx)+x(ind+nx))- b(ind);

    for j=1:nx-2
        ind=ind+1;  %middle
        x(ind)= 0.25*(x(ind-1)+x(ind+1)+...
            x(ind-nx)+x(ind+nx))- b(ind);

    end
    ind=ind+1;  %right side
    x(ind)= 0.25*(x(ind-1)+...
        x(ind-nx)+x(ind+nx))- b(ind);


end

ind=ind+1;% bottom left
 x(ind)= 0.25*(x(ind+1)+                x(ind)+        ...
    x(ind-nx))- b(ind);                 % For BC

for i = 2:nx-1
    ind=ind+1;  %bottom side
    x(ind)= 0.25*(x(ind-1)+x(ind+1)+    x(ind)+...
        x(ind-nx))- b(ind);             % For BC

end
ind=ind+1; %bottom right corner
x(ind)= 0.25*(x(ind-1)+                 x(ind)+...
    x(ind-nx))- b(ind);                 % For BC


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [b]= fmg_resid_2(u,b,m,dx)
%
%
%   copies the residual of the
%   of the estimate (u) into the right hand side (b)
%   at the next coarse grid location
%
%
%

nxh=(2^m+1); %number of x on fine grid
nxH=(2^(m-1)+1); %number of xs on course grid
Nh=nxh^2; NH=nxH^2; %total number of points
[ih]=fmg_find_ind(m,2); %starting index of fine grid
[iH]=fmg_find_ind(m-1,2); %starting index of course grid
ind=ih;
```

```
resid=zeros(Nh,1);%initialize resid

ii=1;

dx2i=1/dx^2;

%top left
resid(ii)= dx2i*(u(ind+1)+...
    u(ind+nx)-4*u(ind)) - b(ind);

....
%
%% proceeds just like gsd_2.m
%
....

%bottom right corner
ii=ii+1; ind=ind+1;
resid(ii)= dx2i*(u(ind-1)+...
    u(ind-nx) -3*u(ind)) - b(ind);
    %          ^ is -3 for homo neumann BC

resid=mg_restrict2(resid,nx); %restrict resid to coarse grid
b([iH:ih-1])=-resid; %store residual in b


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [u]= fmg_adderr_2(u,m)

%   adds the error of the
%   of the estimate (u) to the error
%   on the next finer grid

nxhh=(2^(m+1)); %nx of grid above fine
nxh=(2^m+1);     % nx of fine grid
 nxH=(2^(m-1)+1); %nx of coarse grid
 Nh=nxh^2; NH=nxH^2; %total grid points
 ih = fmg_find_ind(m,2); %starting index of fine grid
 iH = fmg_find_ind(m-1,2); %starting index of coarse grid
 ihh=fmg_find_ind(m+1,2);  % end of fine grid +1

err=x([iH:ih-1]); % copy error from coarse grid
err=mg_interp2(err,nxH); %interpolate coarse grid

% add interpolated error to error on next finer grid.
u([ih:ihh-1])=err+u([ih:ihh-1]);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ind]=fmg_find_ind(m,dim)
%
%   find first index of data at resolution m,
%   dim is the dimension of the data
%
ind=1;
```

```
if m~=1
    for i = 2:m
        ind=ind + (2^(i-1) +1)^dim;
    end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out] = mg_interp2(x_in,nx);
%  Second order (bi-linear) interpolation operator for multigrid

N=length(x_in);  %number of grid points in old
ny=N/nx;         %number of ys in old
nx_new=2*nx-1; % number of x's in new
ny_new=2*ny-1;  % number of y's in new
N_new=nx_new*ny_new;  % number of gpoints in new

x_out = zeros(N_new,1); % initialize output

n=0;   % counter for new matrix
m=0;   % counter for old matrix



for k=1:ny   % for each OLD y
    for j=1:nx-1   %for each old x
        n=n+1;               %\
        m=m+1;               % > copy 1st, 3rd 5th etc directly
        x_out(n)= x_in(m);   %/

        n=n+1;   % advance "new" one point
        x_out(n)= 0.5*(x_in(m) +x_in(m+1));% ave of left and right of old
    end
    n=n+1;               %\
    m=m+1;               % > copy last in line
    x_out(n)= x_in(m);   %/

    n=n+nx_new; %skip line where all points are new (do that next);
end


n=nx_new; m=0; for k=1:ny-1
    for i=1:nx-1
        n=n+1;
        m=m+1;
        x_out(n)=0.5*x_in(m)+0.5*x_in(m+nx);
        n=n+1;
        x_out(n)=0.25*(x_in(m) + x_in(m+1) + x_in(m+nx) + x_in(m+nx+1));
    end
    n=n+1;
    m=m+1;
    x_out(n)=0.5*x_in(m)+0.5*x_in(m+nx);
    n=n+nx_new;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_out] = mg_restrict2(x_in,nx);
%
%   restrict (uninterpolate?) input vector.
%   full weighting scheme

N=length(x_in);

ny=N/nx; nx_new=ceil(nx/2); ny_new=ceil(nx/2); N_new =
nx_new*ny_new;  % get sizes straightened out

x_out=zeros(N_new,1); %initialize output


%top left
m=1;n=1;
 x_out(m)= 0.25*x_in(n)...
            + 0.125*x_in(n+1) + 0.125*x_in(n+nx) ...
            + 0.0625*x_in(n+nx+1);
%top mid
for i = 2:nx_new-1
    m=m+1; n=n+2;
    x_out(m)= 0.25*x_in(n)...
            + 0.125*x_in(n-1) + 0.125*x_in(n+1) + 0.125*x_in(n+nx) ...
            + 0.0625*x_in(n+nx-1)+ 0.0625*x_in(n+nx+1);
end
%top right
m=m+1;n=n+2; x_out(m)= 0.25*x_in(n)...
          + 0.125*x_in(n-1) + 0.125*x_in(n+nx) ...
            + 0.0625*x_in(n+nx-1);

.............
% proceeds as mg_gsd_2
%
.............
```

Table 1: 2 grid algorithm in a box

1. *while* $\Theta < C$
2. $\tilde{u} = \mathcal{S}(\tilde{u})$
3. $d_h = \mathcal{L}\tilde{u} - f$
4. $d_H = \mathcal{R}d_h$
5. $\varepsilon_H = \mathcal{L}_H^{-1} d_H \qquad (or \quad \varepsilon_H = \mathcal{S}\varepsilon_H)$
6. $\varepsilon_h = \mathcal{P}\varepsilon_H$
7. $\tilde{u}^{new} = \tilde{u} + \varepsilon_h$
8. *end*

List of Figures