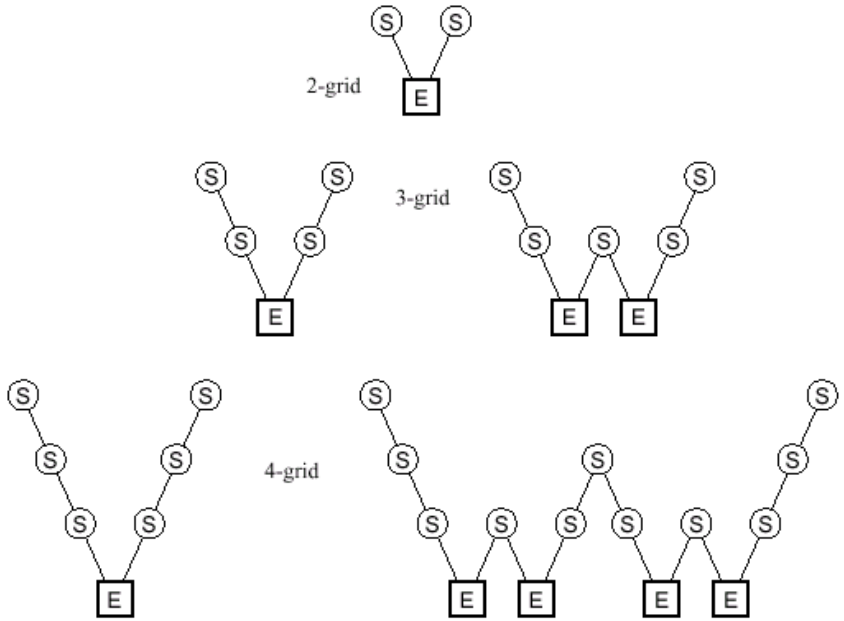
Lecture 21

- Multigrid methods: FAS and nonlinear equations.
- Truncation error.
- Non-linear Newton-Gauss-Seidel.

Multigrid methods:

Last class we introduced the multigrid method. It consisted in viewing a single Jacobi (or Gauss-Seidel) iteration as a “filter” that “swept away” high frequency errors of the initial guess. One would therefore jump up to a coarser grid to get rid fast of low frequency (long wavelength) errors.



Going back and forth between find and coarse meshes was achieved with “restriction” and “prolongation” operators.

One should not use SOR to “relax” in the multigrid method. Over-relaxation is not as good for killing the high frequency errors and this is at the root of the multigrid method.

The prolongation and restriction operators need to comply with the boundary conditions of the problem. If this is difficult, you can always write the problem as having Dirichlet=0 boundary conditions with extra source terms as we described.

Full multigrid method:

Instead of starting with a fine mesh and then iterate further with coarser meshes, the full multigrid method starts with the coarsest possible mesh. In that mesh you solve the problem exactly. You then prolong up to a finer mesh, iterate and either keep prolonging up or go back and forth. Instead of “V” and “W” schemes one ends with “N” schemes.

The most popular smoothing method is Gauss-Seidel, since it usually leads to a good convergence rate. If we order points from 1 to N, the GS scheme is,

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

We need to define the “prolongation” and “restriction” operators. One way characterize them is to give their “symbol”. One considers v_H to be one at a given lattice point and zero otherwise and then gives the values of $\mathfrak{P}v_H$.

The most popular prolongation operator is simple bilinear interpolation.

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

The symbol of R is defined by considering v_h to be defined everywhere on the fine grid and asking what is $\Re v_h$ at (x,y) as a linear combination of the values of v_h.

The simplest choice is “straight injection” where we copy the values at the corresponding points on the fine grid onto the coarse grid.

It is better not to have such an asymmetry in the two operators, but actually to define $\Re$ as the adjoint operator of $\Im$. To define an adjoint we need an inner product,

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

And the adjoint is defined as,

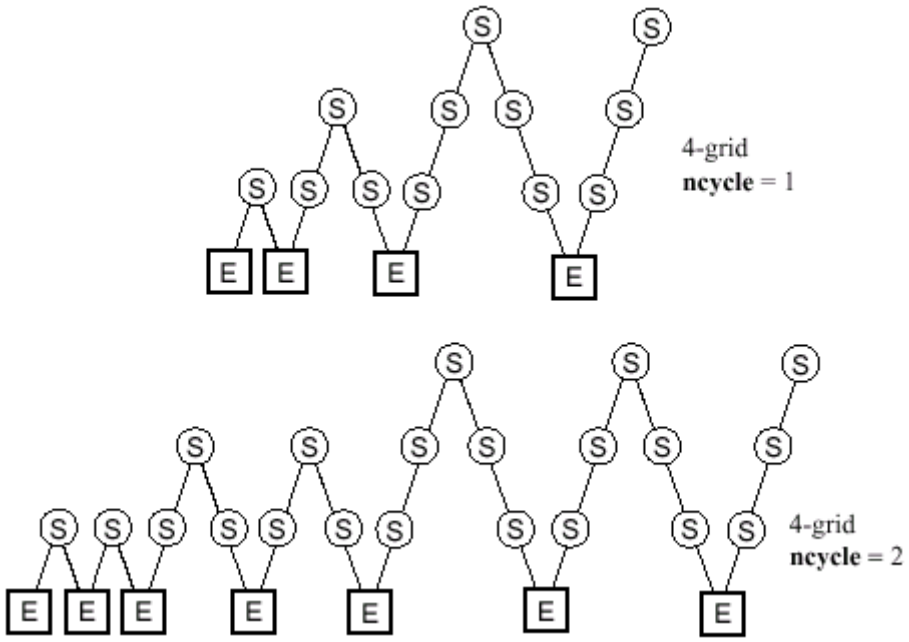
$$\langle u_H | \mathcal{P}^\dagger v_h \rangle_H = \langle \mathcal{P} u_H | v_h \rangle_h$$

Now choose u_H=1 at (x,y) zero elsewhere and H=2h. One gets that,

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

So the symbol is ¼ the transpose of $\mathcal{P}$

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



Multigrid codes are quite elaborate, in particular in terms of memory allocation, since one has to juggle many arrays of different sizes. It is the kind of elaborate programming where Fortran starts to be weak. For instance the Numerical Recipes implementation has to use a routine that mimics c's malloc routine.

Since one is obtaining the solution at various levels of meshes with different spacings, one can combine the multigrid method with Richardson extrapolation.

Multigrid methods for nonlinear equations:

Suppose you want to solve the discretized equation, $\mathcal{L}_h(u_h) = f_h$

Let us also assume that you have run a relaxation algorithm in such a way that you have an approximate solution and the residuals are smooth. That means that one can find a smooth v_h such that,

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

To find the v_h notice that, $\mathcal{L}_h(\widetilde{u}_h + v_h) - \mathcal{L}_h(\widetilde{u}_h) = f_h - \mathcal{L}_h(\widetilde{u}_h)$

$$= -d_h$$

Which means that the RHS of this equation is smooth. Therefore we can bump the LHS into the coarser grid,

$\mathcal{L}_H(u_H) - \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) = -\mathcal{R}d_h$ The coarse grid correction is,

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

And therefore we have $\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h)$

We mentioned that the main point of multigrid is to take advantage of the fast coarse grids to relax the long wavelength modes that are hard to control with fine grids. Let us look at a different aspect that illustrates why the method is so powerful.

Let us consider the truncation error (u is the solution to the continuum equation),

$$\tau \equiv \mathcal{L}_h(u) - f_h \qquad \text{or,} \qquad \mathcal{L}_h(u) = f_h + \tau$$

And define the relative truncation error as, $\tau_h \equiv \mathcal{L}_H(\mathcal{R}u_h) - \mathcal{R}\mathcal{L}_h(u_h)$

So, $\mathcal{L}_H(u_H) = f_H + \tau_h$

f_h

And we can view the relative truncation error as what we need to add to the source in order for the fine grain solution to solve the coarse-grain equation. Of course, we do not know such quantity, but we can estimate it as,

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

So making the replacement, we get,

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

But this is the formula we got for the nonlinear multigrid method.

Deriving the formula in this fashion allows us to understand the method in a different light. It appears as we are using the information from the fine mesh to improve the solution in the coarse mesh.

Which in turn was helping us speed up the solution of the fine mesh!

A piece of jargon: because one is solving for the full solution u (and not just the error as we did in the linear case) the algorithm we described is called Full Approximation Storage or FAS.

These algorithms, as you see, rely on pretty sophisticated numerical analysis. They were introduced in the late 70's by A. Brandt.

The truncation error is also useful as a guide as to when to stop the algorithms. Normally one would like to have residuals that are small, $|d_h|<\varepsilon$. The problem is what is ε?

A natural measure is to take it to be the truncation error. There's no point in requesting more accuracy than that. We don't quite know the truncation error, just an approximation to it, however, assuming things are smooth enough to warrant a Taylor expansion in the stepsize,

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

$$\tau_h = \mathcal{L}_H(u) - \mathcal{L}_h(u) = (H^2 - h^2)\tau_2 + O(h^4)$$

And if we are taking H=2h, then $\tau \simeq \frac{1}{3}\tau_h \simeq \frac{1}{3}\widetilde{\tau}_h$

So for our error estimate we would have $|d_h|<\tau_h/3$

The only thing we are lacking for fully implementing these methods is a non-linear relaxation routine. One possibility is to iterate at the level of the equation.

Example: $\nabla^2\phi = -\frac{1}{8}\frac{K^2}{\phi^7}$

Start with a guess for phi (perhaps=1), put it into the RHS and solve Poisson equation. This gives you a new guess.

Closely related is the non-linear Gauss-Seidel method. Suppose you have N equations,

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

The method consists in starting with a guess for N-1 variables and,

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

If as a consequence of this, we landed a linear equation for u_i^{new}, we're done.

If the equation is still non-linear then one goes Newton-Raphson,

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

As an example, consider the equation, $\nabla^2 u + u^2 = \rho$

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

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

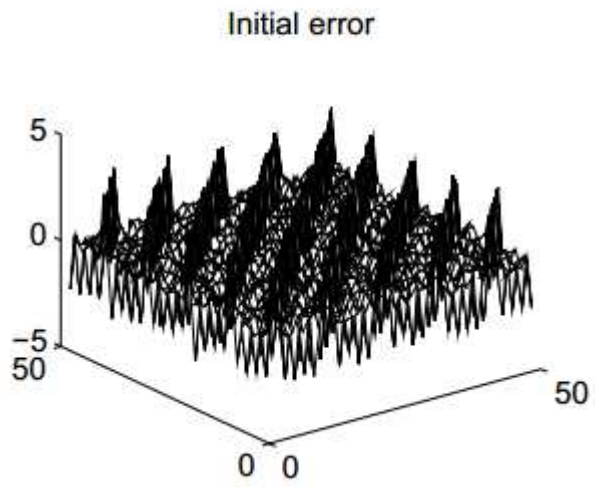
So the Newton-Gauss-Seidel iteration is $u_{i,j}^{\rm new} = u_{i,j} - \dfrac{\mathcal{L}(u_{i,j})}{-4/h^2 + 2u_{i,j}}$

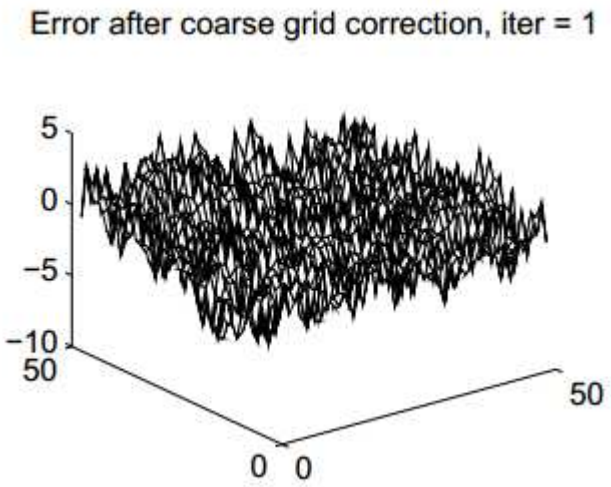
As we stressed last class, the routines that implement all this are quite elaborate. Studying them in detail goes well beyond the scope of this course, so we will leave the discussion at a theoretical level. Sorry!

For those of you who use MATLAB, you can easily play with Multigrid with the MGLab package:
http://web.ics.purdue.edu/~fsaied/pubs/bordner_saied.pdf

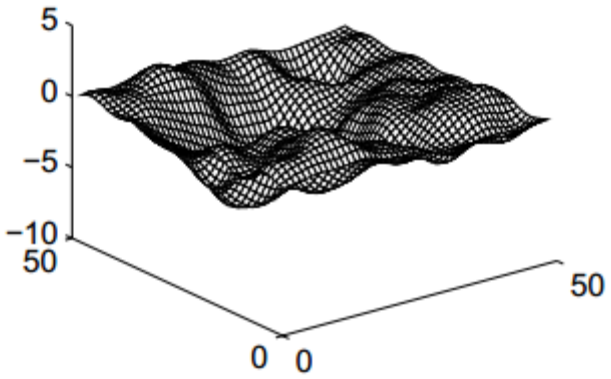
ftp://casper.cs.yale.edu:/mgnet/Codes/mglab

They have an example of a Poisson equation in a 49x49 grid

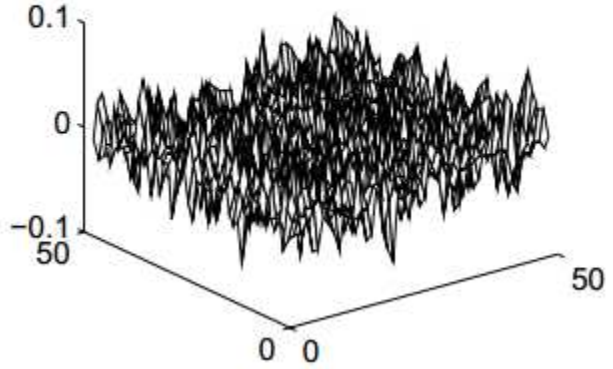




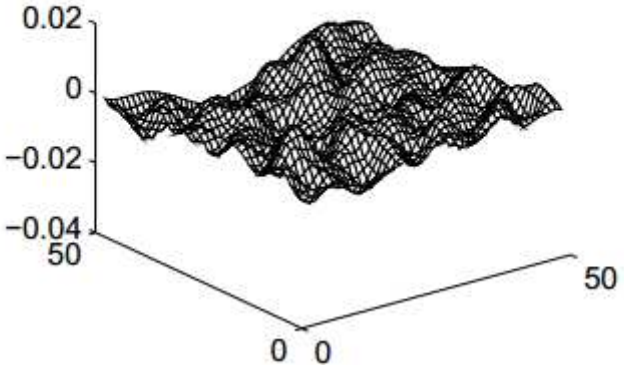
Error after post−smoothing, iter = 1



Error after coarse grid correction, iter = 2

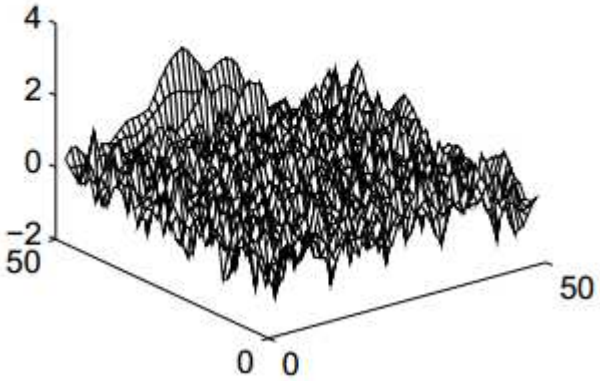


Error after post−smoothing, iter = 2



Error after coarse grid correction, iter = 3

x 10^{-3}



Summary

- Multigrid has various possible implementations, but the basic idea is the same.
- For non-linear systems one iterates.