Editor: Dianne P. O'Leary, oleary@cs.umd.edu

MULTIGRID METHODS

Managing Massive Meshes

By Dianne P. O'Leary

IN OUR LAST HOMEWORK ASSIGNMENT, WE INVESTIGATED ITERATIVE METHODS FOR SOLVING LARGE, SPARSE, LINEAR SYSTEMS OF EQUATIONS. WE SAW THAT THE GAUSS-SEIDEL (GS) method was intolerably slow, but various forms of preconditioned conjugate gradient (CG) algorithms gave us reasonable results.

The test problems we used were discretizations of elliptic partial differential equations, but for these problems, we can use a faster class of methods called *multigrid* algorithms. Surprisingly, the GS method (or some variant) is one of the two main ingredients in these algorithms!

To introduce the ideas, let's drop back to a somewhat simpler problem, one that we considered in May/June 2005 ("Finite Differences and Finite Elements: Getting to Know You," vol. 7, no. 3, pp. 72–79).

A Simple Example

Suppose we want to solve the differential equation

$$-u_{xx}(x) = f(x)$$

on the domain $x \in [0, 1]$, with $u(0) = u(1) = 0$. We know from the earlier homework assignment that we can approximate the solution by defining a mesh $x_j = jh$, where $h = 1/(n + 1)$ for some integer n. Then we can determine approximate values $u_j \approx u(x_j), j = 1, \ldots, n$ using *finite difference* or *finite element* approximations. If we choose finite differences, we have

$$-u_{xx}(x_j) \approx \frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2},$$

so we obtain a system of equations $\boldsymbol{A}\mathbf{u} = \mathbf{f}$ with $\mathbf{u} = [u_1, \ldots, u_n]^T$, $\mathbf{f} = [f(x_1), \ldots, f(x_n)]^T$, and $\boldsymbol{A}$ equal to the tridiagonal matrix

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}.$$

Recall that in the GS method, we take an initial guess $\mathbf{u}^{(0)}$ for the solution and then update the guess by cycling through the equations, solving equation i for the ith variable u_i, so that given $\mathbf{u}^{(k)}$, our next guess $\mathbf{u}^{(k+1)}$ becomes

$$\mathbf{u}_i^{(k+1)} = \left(f_i - \sum_{j=1}^{i-1} a_{ij}\mathbf{u}_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\mathbf{u}_j^{(k)} \right) / a_{ii}, \; i = 1, \ldots, n.$$

In our case, this reduces to

$$\mathbf{u}_i^{(k+1)} = h^2\left(f_i + u_{i-1}^{(k+1)} + u_{i+1}^{(k)} \right) / 2, \; i = 1, \ldots, n,$$

where we define

$$u_0^{(k)} = u_{n+1}^{(k)} = 0$$

for all k.

It's easy to see how GS can be so slow on a problem like this. Suppose, for example, that we take $\mathbf{u}^{(0)} = \mathbf{0}$, and that $\mathbf{f}$ is zero except for a one in its last position. Then, $\mathbf{u}^{(1)}$ is zero except for its last entry, $\mathbf{u}^{(2)}$ is zero except for its last two entries, and it takes n iterations to get a guess that has a nonzero first entry. Because the true solution has nonzeros everywhere, this isn't good!

The real problem is that although GS is good at fixing the solution locally, the information is propagated much too slowly *globally*, across the entire solution vector. So if we're going to use GS effectively, we need to couple it with a method that has good global properties.

A Multigrid Algorithm

When we set up our problem, we chose a value of n, probably guided by the knowledge that the error in the finite dif-

Tools

The multigrid idea dates back to R.P. Fedorenko in 1964. A good introduction is given in a tutorial by William Briggs, Van Emden Henson, and Steve McCormick.[1] Multigrid ideas are useful even when there is no natural geometric "grid" underlying the problem; the resulting method is called *algebraic multigrid* and is briefly discussed in the previously mentioned tutorial.[1]

It's also useful to use multigrid if only a portion of the grid is refined from one level to the next; for example, we might want to refine only in regions in which the solution is so rapidly changing that the current grid can't capture its behavior accurately enough. These adaptive methods are also discussed in the previously mentioned tutorial.[1]

A multigrid approach to solving the Helmholtz equation with negative κ appears in the *SIAM Journal on Scientific Computing*.[2] Watch for articles in the next issue of *CiSE* (November/December 2006), which will cover multigrid.

References

1. W.L. Briggs, V.E. Henson, and S.F. McCormick, *A Multigrid Tutorial*, 2nd ed., SIAM Press, 2000.
2. H.C. Elman, O.G. Ernst, and D.P. O'Leary, "A Multigrid Method Enhanced by Krylov Subspace Iteration for Discrete Helmholtz Equations," *SIAM J. Scientific Computing*, vol. 23, 2001, pp. 1291–1315.

ference approximation is proportional to h^2. A whole family of finite difference approximations exists, defined by different choices of h, and we denote the system of equations obtained using a mesh length $h = 1/(n + 1)$ by

$$A_h \mathbf{u}_h = \mathbf{f}_h.$$

Figure 1 shows the meshes on the interval [0, 1] corresponding to $h = 1/2$, 1/4, 1/8, and 1/16.

A large value of h gives a *coarse grid*. The dimension n of the resulting linear system of equations is very small, though, so we can solve it fast using either a direct or an iterative method. Our computed solution $\mathbf{u}_h$ will have the same overall shape as the true solution u but will lose a lot of local detail.

In contrast, if we use a very *fine grid* with a small value of h, the linear system of equations is very large and much more expensive to solve, but our computed solution $\mathbf{u}_h$ will be very close to u.

To get the best of both worlds, we might use a coarse-grid solution as an initial guess for the GS iteration on a finer grid. To do this, we must set values for points in the finer grid that aren't in the coarse grid. If someone gave us a solution to the system corresponding to h, then we could obtain an approximate solution for the system corresponding to $h/2$ by *interpolating* those values:

- For points in the finer mesh that are common to the coarser mesh, we just take their values.
- For points in the finer mesh that are midpoints of two points in the coarser mesh, we take the average of these two values.

This defines an *interpolation operator* $\boldsymbol{P}_h$ that takes values in a grid with parameter h and produces values in the grid with parameter $h/2$. Because our boundary conditions are zero, for example,

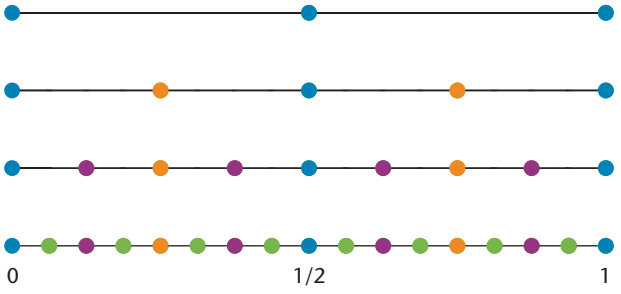

Figure 1. Four levels of nested grids on the interval [0, 1]. The coarsest grid, with $h = 1/2$, consists of blue points; adding the orange points gives $h = 1/4$. Including the purple points gives $h = 1/8$, and including all of the points gives the finest grid, with $h = 1/16$.

$$P_{1/8} = \begin{bmatrix} 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 \end{bmatrix}.$$

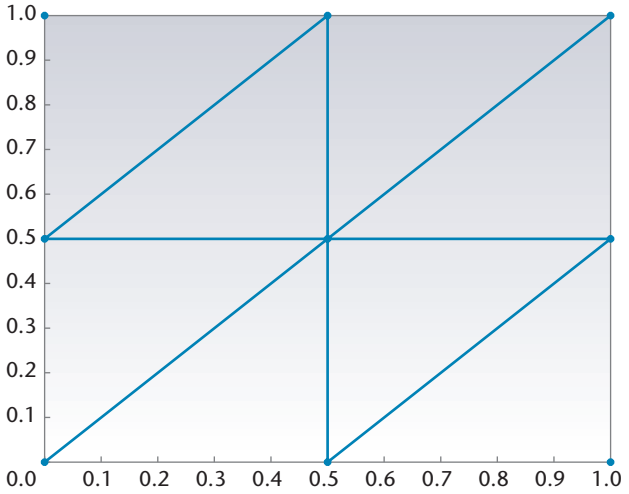

Figure 2. Blue grid points define a coarse mesh.

The process of solving the problems on the sequence of nested grids gives us a *nested iteration* algorithm for our sample problem.

Nested Iteration Algorithm

Set $k = 1$, $h = 1/2$, and $\tilde{\mathbf{u}}_h = 0$.
Until the approximation is good enough,
  Set $k = k + 1$, $n = 2^k - 1$, and $h = 1/(n + 1)$.
  Form the matrix A_h and the right-hand side $\mathbf{f}_h$, and then use the GS iteration, with the initial guess $P_{2h}\tilde{\mathbf{u}}_{2h}$, to compute an approximate solution $\tilde{\mathbf{u}}_h$ to $A_h\mathbf{u}_h = \mathbf{f}_h$.

The termination tolerance for the norm of the residual $\mathbf{f}_h - A_h\tilde{\mathbf{u}}_h$ on grid h should be proportional to h^2 because that matches the size of the local error.

This algorithm runs from coarse grid to finest and is useful (although rather silly for one-dimensional problems). But there's a better way.

The V-Cycle

We can do better if we run from finest grid to coarsest grid and then back to finest. This algorithm has three ingredients:

- An iterative method that converges quickly if most of the error is *high frequency*—oscillating rapidly—which happens when the solution's overall shape is already identified. GS generally works well.
- A way to transfer values from a coarse grid to a fine one—*interpolation* or *prolongation*.
- A way to transfer values from a fine grid to a coarse one—*restriction*. We let R_h be the operator that takes values on grid $h/2$ and produces values on grid h.

We already have matrices P_h for interpolation, and (for technical reasons related to preserving the *self-adjointness* of the problems considered here) we choose $R_h = P_h^T$. We'll define the V-Cycle idea recursively.

V-Cycle Algorithm

$\mathbf{v}_h$ = V-Cycle($\mathbf{v}_h$, A_h, $\mathbf{f}_h$, η_1, η_2):
1. Perform η_1 GS iterations on $A_h\mathbf{u}_h = \mathbf{f}_h$ using $\mathbf{v}_h$ as the initial guess, obtaining an approximate solution that we still call $\mathbf{v}_h$.
2. If h is the coarsest grid parameter, then compute $\mathbf{v}_h$ to solve $A_h\mathbf{v}_h = \mathbf{f}_h$ and return.
3. Otherwise,
   Let $\mathbf{v}_{2h}$ = V-Cycle($\mathbf{0}$, A_{2h}, $R_{2h}(\mathbf{f}_h - A_h\mathbf{v}_h)$, η_1, η_2).
   Set $\mathbf{v}_h = \mathbf{v}_h + P_{2h}\mathbf{v}_{2h}$.
4. Perform η_2 GS iterations on $A_h\mathbf{u}_h = \mathbf{f}_h$, using $\mathbf{v}_h$ as the initial guess, ultimately obtaining an approximate solution that we still call $\mathbf{v}_h$.

In using this algorithm, we can define $A_{2h} = R_{2h}A_hP_{2h}$. This definition is the key to extending the multigrid algorithm beyond problems that have a geometric grid; see the "Tools" sidebar for a reference on *algebraic multigrid methods*. But for now, let's see how it works on our original problem.

Problem 1.

Work through the V-Cycle algorithm to see exactly what computations it performs on our simple example for the grid sequence in Figure 1. Estimate the amount of work, measured by the number of floating-point multiplications performed.

The standard multigrid algorithm solves $A_h\mathbf{u}_h = \mathbf{f}_h$ by repeating the V-Cycle until convergence. We start the iteration by initializing $\mathbf{u}_h = \mathbf{0}$. Then, until convergence, we compute $\Delta\mathbf{u}_h$ = V-Cycle($\mathbf{0}$, A_h, $\mathbf{r}_h$, η_1, η_2), where $\mathbf{r}_h = \mathbf{f}_h - A_h\mathbf{u}_h$, and then update $\mathbf{u}_h = \mathbf{u}_h + \Delta\mathbf{u}_h$.

Cost of Multigrid

Guided by Problem 1, we can estimate the work for multigrid applied to a more general problem. One step of the GS iteration on a grid of size h costs about $nz(h)$ multiplications, where $nz(h)$ is the number of nonzeros in A_h. We'll call $nz(h)$ multiplications a *work unit*.

Note that $nz(h) \approx 2nz(2h)$ because A_{2h} has about half as

many rows as A_h. So performing one GS step on each grid h, $h/2$, …, 1 costs less than $nz(h)(1 + 1/2 + 1/4 + \ldots) \approx 2nz(h)$ multiplications ≡ 2 *work units*.

So the cost of a V-Cycle is at most two times the cost of $(\eta_1 + \eta_2)$ GS iterations on the finest mesh plus a modest amount of additional overhead.

PROBLEM 2.

Convince yourself that the storage necessary for all the matrices and vectors is also a modest multiple of the storage necessary for the finest grid.

We know that standard iterative methods such as GS are very slow (take many iterations), so multigrid's success relies on the fact that we need only a few iterations on each grid, thus the total amount of work to solve the full problem to a residual of size $O(h^2)$ is *a small number* of work units.

Because it's rather silly to use anything other than sparse Gauss elimination to solve a system involving a tridiagonal matrix, we won't implement the algorithm for our one-dimensional (1D) problem. Note, however, that our algorithm readily extends to higher dimensions: we just need to define $\boldsymbol{A}_h$ and $\boldsymbol{P}_h$ for a nested set of grids to use the multigrid V-Cycle algorithm.

Multigrid for Two-Dimensional Problems

Our first challenge in applying multigrid to 2D problems is to develop a sequence of *nested grids*. Because we discussed finite difference methods for the 1D problem, let's focus on finite element methods for the 2D problem, using a triangular mesh and piecewise-linear basis functions.

It's most convenient to start from a *coarse grid* and obtain our *finest grid* through successive refinements. Consider the initial grid in Figure 2, which divides the unit square into eight triangles with height $h = 1/2$. The grid points are marked in blue.

Consider taking the midpoints of each side of one of the triangles and then drawing the triangle with those points as vertices. If we do this for each triangle, we get the orange grid points and triangles in Figure 3. Each of the original blue triangles is replaced by four triangles, each having one or three orange sides (triangle height $h = 1/4$).

If we repeat this process, we get the purple grid points in Figure 4 and a mesh length $h = 1/8$.

Writing a program for mesh refinement on a general triangular grid takes a bit of effort; see `refine.m` on the Web site (www.computer.org/cise/homework/).

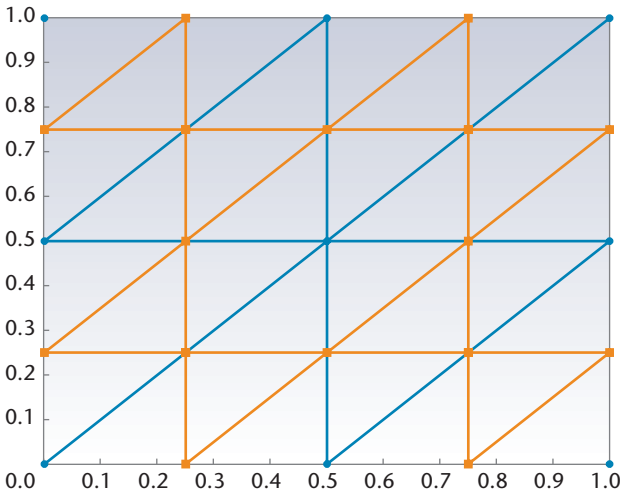

Figure 3. Blue and orange grid points define a finer mesh.

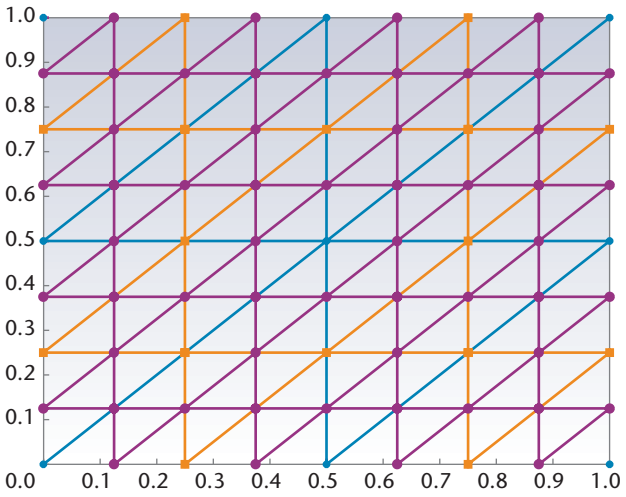

Figure 4. Blue, orange, and purple grid points define the finest mesh.

Interpolating from one grid to the next finer one is easy. For example, given the blue grid values, we obtain values for the blue and orange grids by following two rules: blue grid points retain their values, and orange grid values are defined as the average of the nearest two values on the blue edge containing it. As before, we take the restriction operator to be the transpose of the interpolation operator.

So we have all the machinery necessary to apply multigrid to 2D problems; we'll experiment with it in the next problem.

PROBLEM 3.

Write a program that applies the multigrid V-Cycle iteration to the 2D problems used in the homework assignment

given in the previous issue of *CiSE*. The Matlab program `generateproblem.m` produces a structure called `mesh` that contains, in addition to the matrices and right-hand side information, the operators `p` and the coordinates `p` of the mesh points. The differential equation is

$$-u_{xx}(x, y) - u_{yy}(x, y) + \kappa u(x, y) = f(x, y)$$

for $(x, y) \in [-1, 1] \times [-1, 1]$ (`myproblem=1`) or for this domain with a hole cut out (`myproblem=2`). The boundary conditions are that u is zero on the square's boundary, and (for the second case) the normal derivative is zero at the hole's boundary.

Set $\kappa = 0$ and compare the time for solving the problem using multigrid to the methods defined in the last issue.

If the partial differential equation is elliptic, as it is for $\kappa = 0$, it isn't too hard to achieve convergence in a small number of work units. In fact, multigridders would say that if you don't achieve it, you've chosen either your iteration or your interpolation/restriction pair "incorrectly."

For problems that aren't elliptic, though, things get a bit more complicated, as we see in the next problem.

PROBLEM 4.

Repeat your experiment from Problem 3, but use $\kappa = 10$ and 100 and then $\kappa = -10$ and -100. (When $\kappa \neq 0$, the differential equation is called the *Helmholtz equation*.) The differential equation remains elliptic for positive κ but not for negative. How was convergence of multigrid affected?

We see that the problem is much harder to solve for negative values of κ. There are two reasons for this: the matrix A_h is no longer positive definite, so we lose a lot of good structure, and we need finer grids to represent the solution accurately. To restore convergence in a small number of work units for the nonelliptic problem, we must make the algorithm more complicated—for example, we might use multigrid as a preconditioner for a Krylov subspace method, bringing us back to the methods used in the last homework assignment. The "Tools" sidebar provides a reference on this.

Partial Solution to Last Issue's Homework Assignment

Iterative Methods for Linear Systems

Following the Meandering Way

By Dianne P. O'Leary

SOLVING A LINEAR SYSTEM OF EQUATIONS IS ONE OF THE EASIEST COMPUTATIONAL TASKS IMAGINABLE, BUT WHEN THE MATRIX IS VERY LARGE, THE ALGORITHMS WE LEARNED in linear algebra class take too long, so we need to use a different type of algorithm.

Problem 1.

Use Gauss-Seidel to solve the linear systems of equations generated by the Matlab function `generateproblem` found on the Web site (www.computer.org/cise/homework). (Set the input parameter `kappa` to zero, and note that you only need a part of the structure generated by this function: the matrices `mesh(k).A`, the right-hand side vectors `mesh(k).b`, and the convergence tolerance `mesh(k).tol`.) Use an initial guess $\mathbf{x}^{(0)} = \mathbf{0}$. The mathematical model of heat distribution isn't exact, so we can stop the iteration when the residual's norm is somewhat smaller than the error in the model. We approximate this by stopping when the residual's norm has been reduced by a factor of `mesh(k).tol`. As the number of unknowns in the linear system grows, this error decreases. Graph the number of iterations and the solution time as a function of the number of unknowns, choosing a range of problem sizes appropriate for your computer. Compare the time and storage with that for direct solution of the linear systems.

Problem 2.

Use the PCG algorithm to solve the linear systems of equations considered in Problem 1. Use an initial guess $\mathbf{x}^{(0)} = \mathbf{0}$ and set $\boldsymbol{M} = \boldsymbol{I}$. Graph the number of iterations and the solution time as a function of the number of unknowns and compare this and the storage with the results of Problem 1.

Problem 3.

Use the PCG algorithm to solve the linear systems of equations considered in Problem 1. Use an initial guess $\mathbf{x}^{(0)} = \mathbf{0}$ and set $\boldsymbol{M}$ to be incomplete Cholesky preconditioners generated by `cholinc` with various choices for its parameters `droptol` and `opts`. Compare with the results of Problems 1 and 2.

Problem 5.

Use the PCG algorithm to solve the linear systems of equations considered in Problem 1. Use an initial guess $\mathbf{x}^{(0)} = \mathbf{0}$ and set $\boldsymbol{M}$ to be the symmetric Gauss-Seidel preconditioner. Compare with the results of Problems 1 and 2.

Problem 6.

Repeat your experiments after reordering the matrices using the approximate minimum degree (AMD) reordering and compare the results of all experiments.

Answers:

The solution to these five problems appears on the Web site (www.computer.org/cise/homework/) in `solution20.m`. Figures A and B illustrate the results for the square domain. Gauss-Seidel took too many iterations to be competitive, and the parameter `cut` is the drop-tolerance for the incomplete Cholesky factorization. The AMD-Cholesky factorization was the fastest algorithm for this problem, but it required 5.4 times the storage of the conjugate gradient (CG) algorithm and 2.6 times the storage of the ICCG-PCG algorithm for the problem of size 16,129. Without reordering, Cholesky was slow and very demanding of storage, requiring almost 30 million double-precision words for the largest problem (almost 70 times as much as for the AMD reordering).

Results for the domain with the circle cut out were similar; see Figures C and D.

Gauss-Seidel took a large amount of time per iteration. This is an artifact of the implementation because it's a bit tricky to get Matlab to avoid working with the zero elements when accessing a sparse matrix row-by-row. Challenge: look at the code in `gauss_seidel.m` and try to speed it up. A better version will be provided in the solution to the multigrid homework assignment in the next issue.

Copublished by the IEEE CS and the AIP 1521-9615/06/$20.00 © 2006 IEEE

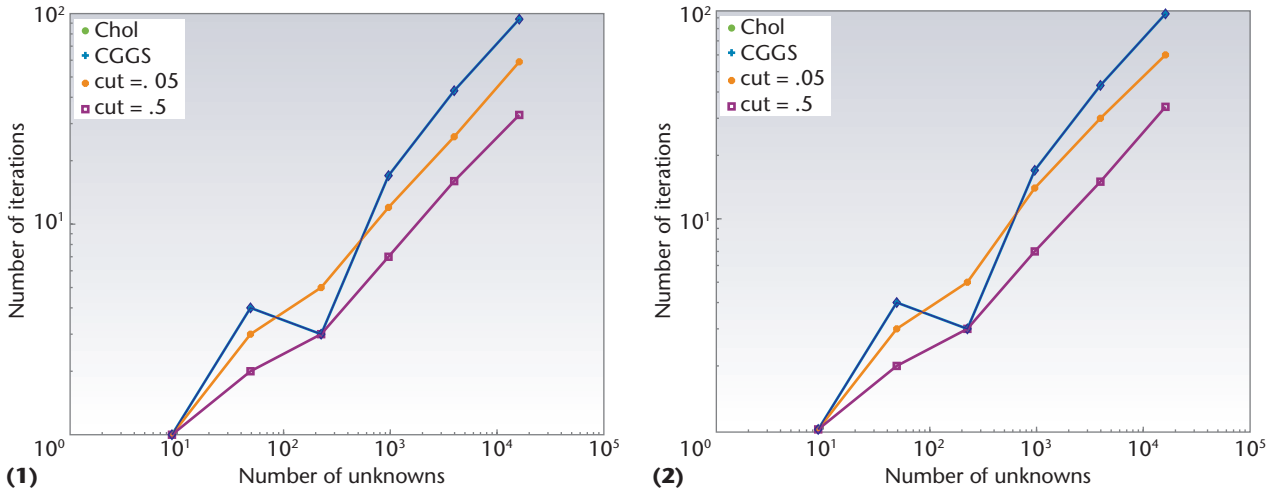

Figure A. The number of iterations for the various methods applied to the square domain. (1) The original ordering and (2) the approximate minimum degree (AMD) ordering.

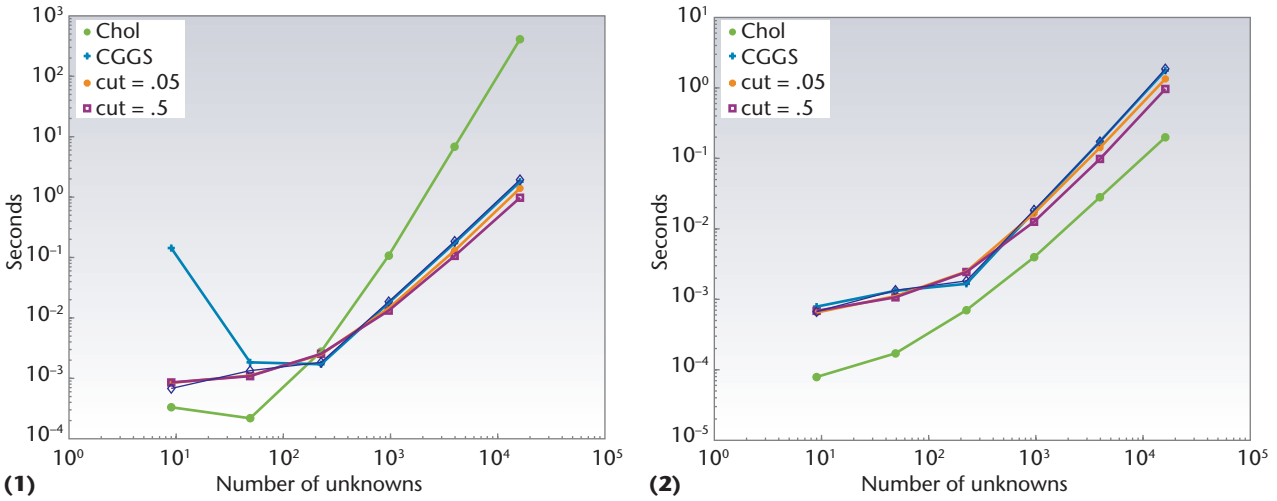

Figure B. Timings for the various methods applied to the square domain. (1) The original ordering and (2) the approximate minimum degree (AMD) ordering.

PROBLEM 4.

Consider our stationary iterative method (SIM)

$$M\mathbf{x}^{(k+1)} = N\mathbf{x}^{(k)} + \mathbf{b} \text{ or } \mathbf{x}^{(k+1)} = M^{-1}N\mathbf{x}^{(k)} + M^{-1}\mathbf{b}.$$

Show that $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + M^{-1}\mathbf{r}^{(k)}$, and therefore the step the SIM takes is

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = M^{-1}\mathbf{r}^{(k)}.$$

Answer:

Manipulating these equations a bit, we get

$$x^{(k+1)} = \mathbf{x}^{(k)} + (M^{-1}N - I)\mathbf{x}^{(k)} + M^{-1}\mathbf{b}$$

$$= \mathbf{x}^{(k)} + M^{-1}(N - M)\mathbf{x}^{(k)} + M^{-1}\mathbf{b}$$

$$= \mathbf{x}^{(k)} + M^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$$

$$= \mathbf{x}^{(k)} + M^{-1}\mathbf{r}^{(k)}.$$

Dianne P. O'Leary is a professor of computer science and a faculty member in the Institute for Advanced Computer Studies and the Applied Mathematics Program at the University of Maryland. She has a BS in mathematics from Purdue University and a PhD in computer science from Stanford. O'Leary is a member of SIAM, the ACM, and AWM. Contact her at oleary@cs.umd.edu; www.cs.umd.edu/users/oleary/.

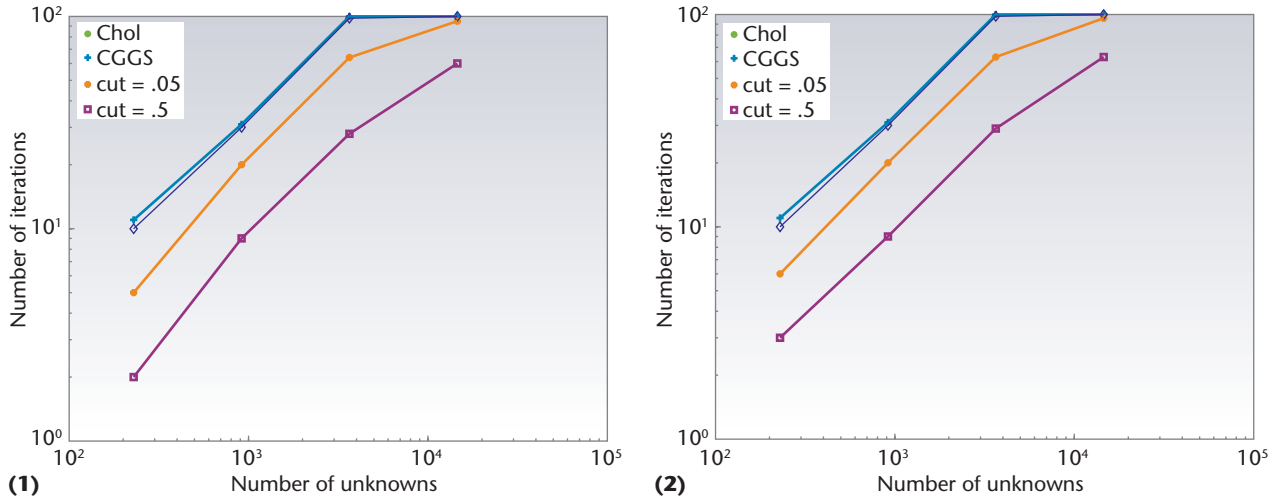

Figure C. Number of iterations for the various methods applied to the domain with the circle cut out. (1) The original ordering and (2) the approximate minimum degree (AMD) ordering.

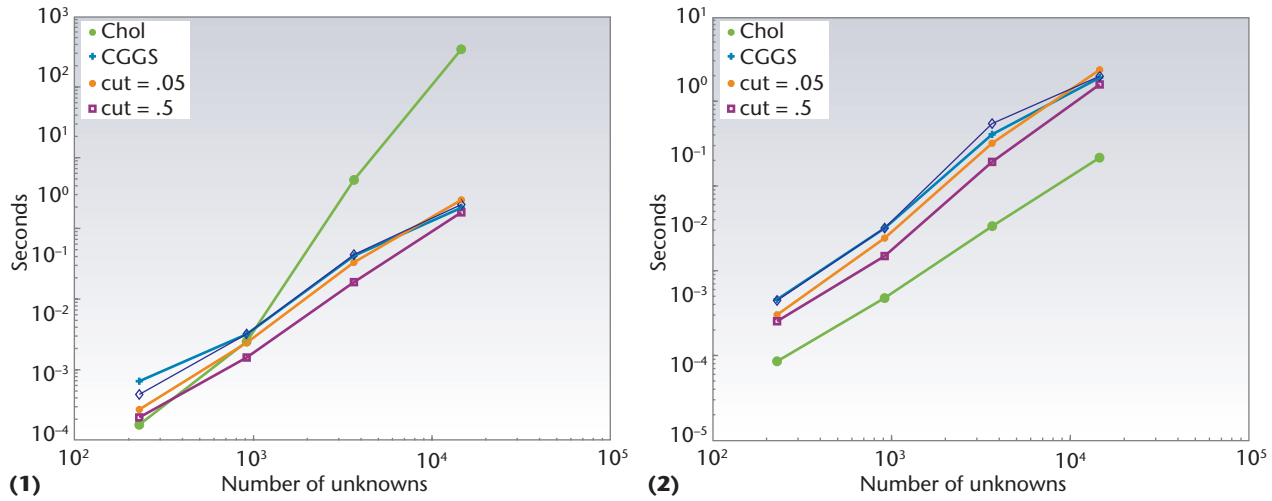

Figure D. Timings for the various methods applied to the domain with the circle cut out. (1) The original ordering and (2) the approximate minimum degree (AMD) ordering.

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. 2006 annual subscription rates: $43 for Computer Society members (print plus online), $73 (sister society), and $74 (individual nonmember). For AIP society members, 2006 annual subscription rates are $42 (print plus online). For more information on other subscription prices, see www.computer.org/subscribe/ or https://www.aip.org/forms/journal_catalog/order_form_fs.html. Computer Society back issues cost $20 for members, $96 for nonmembers; AIP back issues cost $22 for members.

Postmaster: Send undelivered copies and address changes to *Computing in Science & Engineering*, 445 Hoes Ln., Piscataway, NJ 08855. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Corporation (Canadian distribution) publications mail agreement number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8 Canada. Printed in the USA.