CHAPTER 1

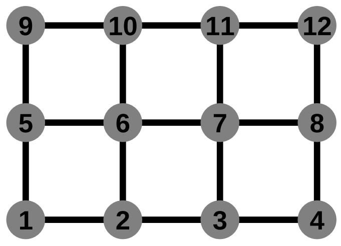
Motivation and Overview

In this chapter we examine solvers for two families of linear systems of equations. Our goal is motivate certain ideas and techniques, not to explain in detail how the solvers work. We focus on the measured behaviors of the solvers and on the fundamental questions that the solvers and their performance pose.

1. The Model Problems: Two- and Three Dimensional Meshes

The coefficient matrices of the linear systems that we study arise from two families of undirected graphs: two- and three-dimensional meshes. The vertices of the graphs, which correspond to rows and columns in the coefficient matrix A, are labeled 1 through n. The edges of the graph correspond to off-diagonal nonzeros in A, the value of which is always -1. If (i, j) is an edge in the graph, then $A_{ij} = A_{ji} = -1$. If (i, j) is not an edge in the mesh for some $i \neq j$, then $A_{ij} = A_{ji} = 0$. The value of the diagonal elements in A is selected to make the sums of elements in rows $2, 3, \ldots, n$ exactly 0, and to make the sum of the elements in the first row exactly 1. We always number meshes so that vertices that differ only in their x coordinates are contiguous and numbered from small x coordinates to large ones; vertices that have the same z coordinate are also contiguous, numbered from small y coordinates to high ones. Figure 1 shows an example of a two-dimensional mesh and the matrix that corresponds to it. Figure 2 shows an example of a small three-dimensional mesh.

Matrices similar to these do arise in applications, such as finite-difference discretizations of partial differential equations. These particular families of matrices are highly structured, and the structure allows specialized solvers to be used, such as multigrid solvers and spectral solvers. The methods that we present in this book are, however, applicable to a much wider class of matrices, not only to those arising from structured meshes. We use these matrices here only to illustrate and motivate various linear solvers.



$$A = \begin{bmatrix}
3 & -1 & & & -1 & & & & & & & \\
-1 & 3 & -1 & & & -1 & & & & & & \\
& -1 & 3 & -1 & & & -1 & & & & & \\
& & -1 & 2 & & & & -1 & & & & \\
-1 & & & & 3 & -1 & & & -1 & & & \\
& -1 & & & -1 & 4 & -1 & & & -1 & & \\
& & -1 & & & -1 & 4 & -1 & & & -1 & \\
& & & -1 & & & -1 & 3 & & & & -1 \\
& & & & -1 & & & & 2 & -1 & & \\
& & & & & -1 & & & -1 & 3 & -1 & \\
& & & & & & -1 & & & -1 & 3 & -1 \\
& & & & & & & & & & -1 & 2
\end{bmatrix}$$

FIGURE 1. A 4-by-3 two-dimentional mesh, with vertices labeled $1, \ldots, 12$, and the coefficient matrix A that corresponds to it.

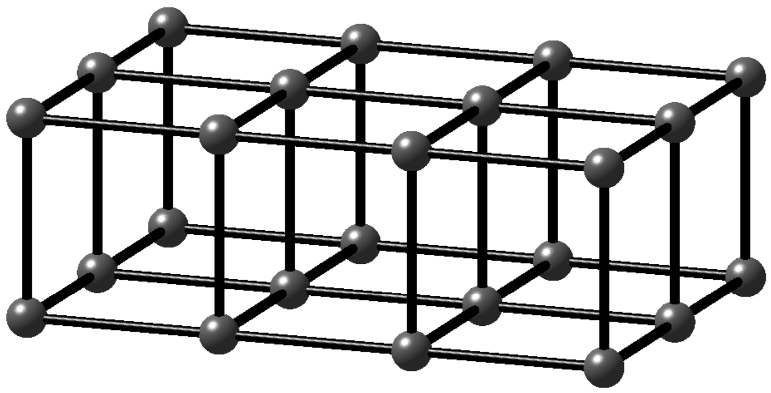


FIGURE 2. A 4-by-3-by-2 three-dimensional mesh. The indices of the vertices are not shown.

The experimental results that we present in the rest of the chapter all use a solution vector x with random uniformly-distributed elements between 0 and 1.

2. The Cholesky Factorization

Our matrices are symmetric (by definition) and positive definite. We do not show here that they are positive definite (all their eigenvalues are positive); this will wait until a later chapter. But they are positive definite.

The simplest way to solve a linear system $Ax = b$ with a positive definite coefficient matrix A is to factor A into a produce of a lower triangular factor L and its transpose, $A = LL^T$. When A is symmetric positive definite, such a factorization always exists. It is called the *Cholesky* factorization, and it is a variant of Gaussian Elimination. Once we compute the factorization, we can solve $Ax = b$ by solving two triangular linear systems,

$$\begin{aligned} Ly &= b \\ L^T x &= y \ . \end{aligned}$$

Solvers for linear systems of equations that factor the coefficient matrix into a product of simpler matrices are called *direct solvers.* Here the factorization is into triangular matrices, but orthogonal, diagonal, tridiagonal, and block diagonal matrices are also used as factors in direct solvers.

The factorization can be computed recursively. If A is 1-by-1, then $L = \sqrt{A_{11}}$. Otherwise, we can partition A and L into 4 blocks each, such that the diagonal blocks are square,

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & L_{22}^T \end{bmatrix} = LL^T \ .$$

This yields the following equations, which define the blocks of L,

$$A_{11} = L_{11}L_{11}^T \tag{1}$$
$$A_{21} = L_{21}L_{11}^T \tag{2}$$
$$A_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T \ . \tag{3}$$

We compute L by recursively solving Equation 1 for L_{11}. Once we have computed L_{11}, we solve Equation 2 for L_{21} by substitution. The final step in the computation of L is to subtract $L_{21}L_{21}^T$ from A_{22} and to factor the difference recursively.

This algorithm is reliable and numerically stable, but if implemented naively, it is slow. Let $\phi(n)$ be the number of arithmetic operations that this algorithm performs on a matrix of size n. To estimate ϕ, we partition A so that A_{11} is 1-by-1. The partitioning does not affect $\phi(n)$, but this particular partitioning makes it easy to estimate $\phi(n)$. With this partitioning, computing L involves computing one square root, dividing an $(n-1)$-vector by the root, computing the symmetric outer product of the scaled $(n-1)$-vector, subtracting this outer product from a symmetric $(n-1)$-by-$(n-1)$ matrix, and recursively factoring the difference. We can set this up as a recurrence relation,

$$\begin{aligned} \phi(n) &= 1 + (n-1) + (n-1)^2 + \phi(n-1) \\ \phi(1) &= 1 \ . \end{aligned}$$

This yields

$$\begin{aligned}\phi(n) &= n+\sum_{j=2}^{n}(j-1)+\sum_{j=2}^{n}(n-1)^2\\ &= n+\sum_{j=1}^{n-1}j+\sum_{j=1}^{n-1}j^2\\ &= n+\frac{n(n-1)}{2}+\frac{n(n-\frac{1}{2})(n-1)}{3}\\ &= \frac{n^3}{3}+o\left(n^3\right)\ .\end{aligned}$$

The little-o notation means that we have neglected terms that grow slower than cn^3 for any constant c.

Cubic growth means that for large meshes, a Cholesky-based solver is slow. For a mesh with a million vertices, the solver needs to perform more than 0.3×10^{18} arithmetic operations. Even at a rate of 10^{12} operations per second, the factorization takes almost 4 days. On a personal computer (say a 2005 model), the factorization will take more than a year. A mesh with a million vertices may seem like a large mesh, but it is not. The mesh has less than 2 million edges in two dimensions and less than 3 million in three dimensions, so its representation does not take a lot of space in memory. We can do a lot better.

3. Sparse Cholesky

One way to speed up the factorization is to exploit sparsity in A. If we inspect the nonzero pattern of A, we see that most of its elements are zero. If A is derived from a 2-dimensional mesh, only 5 diagonals contain nonzero elements. If A is derived from a 3-dimensional mesh, only 7 diagonals are nonzero. Figure 3 shows the nonzero pattern of such matrices. Only $O(n)$ elements out of n^2 in A are nonzero.

Sparse factorization codes exploit the sparsity of A and avoid computations involving zero elements (except, sometimes, for zeros that are created by exact cancellations). Figure 4 compares the running times of a sparse Cholesky solver with that of a dense Cholesky solver, which does not exploit zeros. The sparse solver is clearly must faster, especially on 2-dimensional meshes.

To fully understand the performance of the sparse solver, it helps to inspect the nonzero pattern of the Cholesky factor L, shown in Figure 5. Exploiting sparsity pays off because the factor is also sparse: it fills, but not completely. It is easy to characterize exactly where the factor fills. It fills almost completely within the band structure of A, the

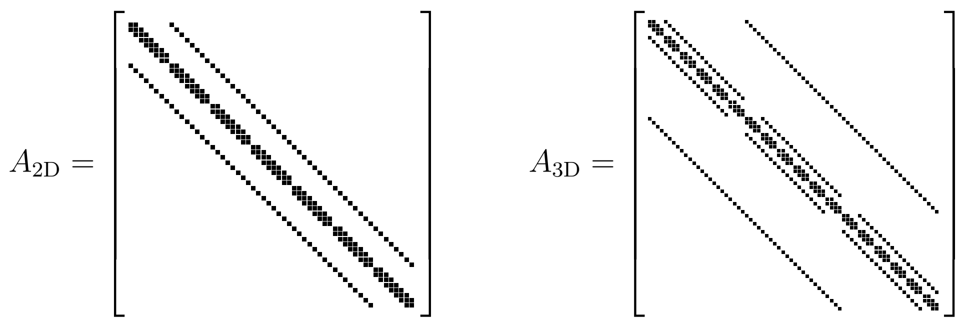


FIGURE 3. The nonzero pattern of matrices corresponding to an 8-by-7 mesh (left) and to a 4-by-6-by-3 mesh. Nonzero elements are denoted by small squares. The other elements are all zeros.

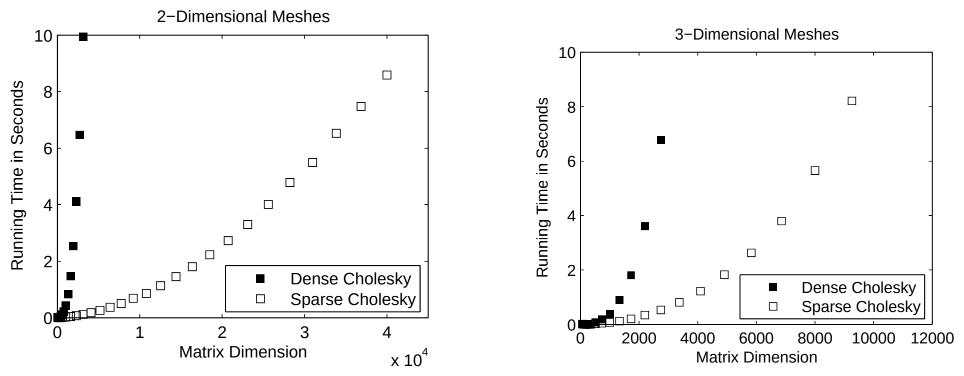


FIGURE 4. The performance of dense and sparse Cholesky on 2- and 3-dimensional meshes. The data in this graph, as well as in all the other graphs in this chapter, was generated using MATLAB 7.0.

set of diagonals d that are closer to the main diagonal than a nonzero diagonal. Outside this band structure, the elements of L are all zero. For a $\sqrt{n}$-by-$\sqrt{n}$ mesh, the outer diagonal is $\sqrt{n}$, so the total amount of arithmetic in a sparse factorization is $n \cdot 2\left(\sqrt{n}\right)^2 + o(n^2) = 2n^2 + o(n^2)$. For large n, this is a lot less than the dense $n^3/3 + o(n^2)$ bound. For three dimensional meshs, exploiting sparsity helps, but not as much. In a matrix corresponding to a $\sqrt[3]{n}$-by-$\sqrt[3]{n}$-by-$\sqrt[3]{n}$ mesh, the outer nonzero diagonal is $n^{2/3}$, so the amount of arithmetic is $2n^{7/3} + o(n^{7/3})$.

4. Preordering for Sparsity

Sparse factorization codes exploit zero elements in the reduced matrices and the factors. It turns out that by reordering the rows and

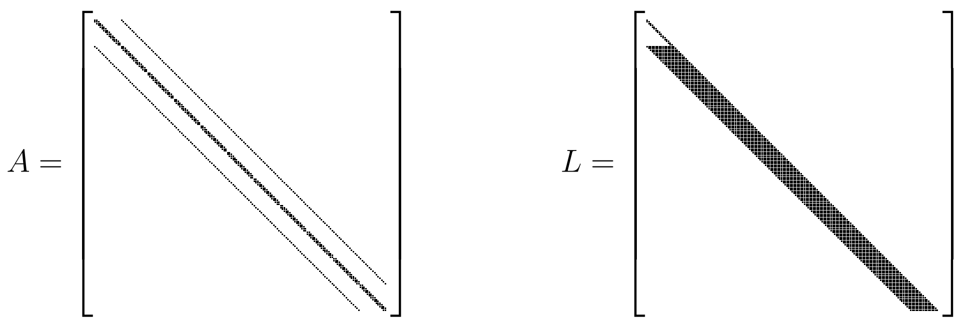


FIGURE 5. The matrix of a 15-by-11 mesh (left) and its Cholesky factor (right).

columns of a matrix we can reduce fill in the reduced matrices and the factors. The most striking example for this phenomenon is the arrow matrix,

$$A = \begin{bmatrix} n+1 & -1 & -1 & -1 & \cdots & -1 \\ -1 & 1 & & & & \\ -1 & & 1 & & & \\ -1 & & & 1 & & \\ \vdots & & & & \ddots & \\ -1 & & & & & 1 \end{bmatrix}.$$

When we factor this matrix, it fills completely after the elimination of the first column, since the outer product $L_{2:\,n,1}L^T_{2:\,n,1}$ is full (has no zeros). But if we reverse the order of the rows and columns, we have

$$\begin{aligned} PAP^T &= \begin{bmatrix} & & & 1 \\ & & 1 & \\ & 1 & & \\ 1 & \cdot^{\cdot^{\cdot}} & & \end{bmatrix} \begin{bmatrix} n+1 & -1 & -1 & -1 & \cdots & -1 \\ -1 & 1 & & & & \\ -1 & & 1 & & & \\ -1 & & & 1 & & \\ \vdots & & & & \ddots & \\ -1 & & & & & 1 \end{bmatrix} \begin{bmatrix} & & & 1 \\ & & 1 & \\ & 1 & & \\ 1 & \cdot^{\cdot^{\cdot}} & & \end{bmatrix} \\ &= \begin{bmatrix} 1 & & & & & -1 \\ & 1 & & & & -1 \\ & & 1 & & & -1 \\ & & & 1 & & -1 \\ & & & & \ddots & \vdots \\ -1 & -1 & -1 & -1 & -1 & n+1 \end{bmatrix}. \end{aligned}$$

The Cholesky factor of PAP^T is as sparse as PAP^T

$$\text{chol}(PAP^T) = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & \ddots & \\ -1 & -1 & -1 & -1 & \cdots & 1 \end{bmatrix}.$$

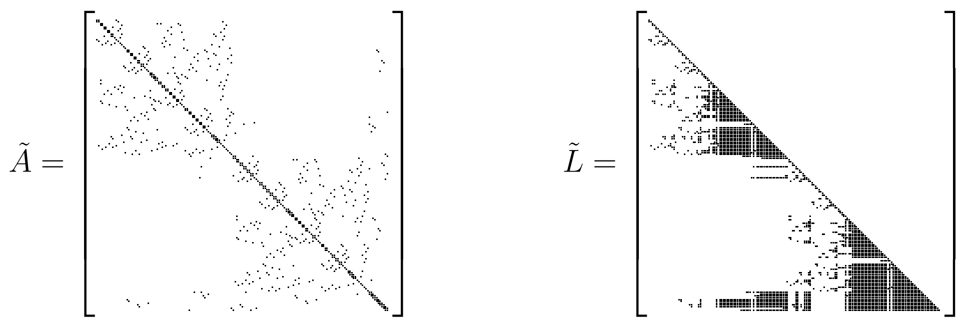


Figure 6. A symmetric nested-dissection permutation $\tilde{A}$ of the matrix shown in Figure 5 and the Cholesky factor $\tilde{L}$ of $\tilde{A}$.

Solving a linear system with a Cholesky factorization of a permuted coefficient matrix is trivial. We apply the same permutation to the right-hand-side b before the two triangular solves, and apply the inverse permutation after the triangular solves.

For matrices that correspond to meshes, there is no permutation that leads to a no-fill factorization, but there are permutations that reduce fill significantly. Figure 6 shows a theoretically-efficive fill-reducing permutation of the matrix of an 11-by-15 mesh and its Cholesky factor. This symmetric reordering of the rows and columns is called a *nested dissection ordering*. It reorders the rows and columns by finding a small set of vertices in the mesh whose removal breaks the mesh into two separate components of roughly the same size. This set is called a *vertex separator* and the corresponding rows and columns are ordered last. The rows and columns corresponding to each connected component of the mesh minus the separator are ordered using the same strategy recursively. Figure 7 illustrates this construction.

The factorization of the nested-dissection-ordered matrix of a a $\sqrt{n}$-by-$\sqrt{n}$ mesh performs $\Theta(n^{3/2})$ arithmetic operations.[1] (We don't give here the detailed analysis of fill and work under nested-dissection orderings.) For large meshes, this is a significant improvement over the $\Theta(n^2)$ operations that the factorization performs without reordering. For a $\sqrt[3]{n}$ -by-$\sqrt[3]{n}$-by-$\sqrt[3]{n}$ mesh, the factorization performs $\Theta(n^2)$ operations. This is again an improvement, but the $\Theta(n^2)$ solution cost is too high for many applications.

Figure 8 compares the running times of sparse Choesky applied to the natural ordering of the meshes and to fill-reducing orderings. The

[1]explain asymptotic notation.

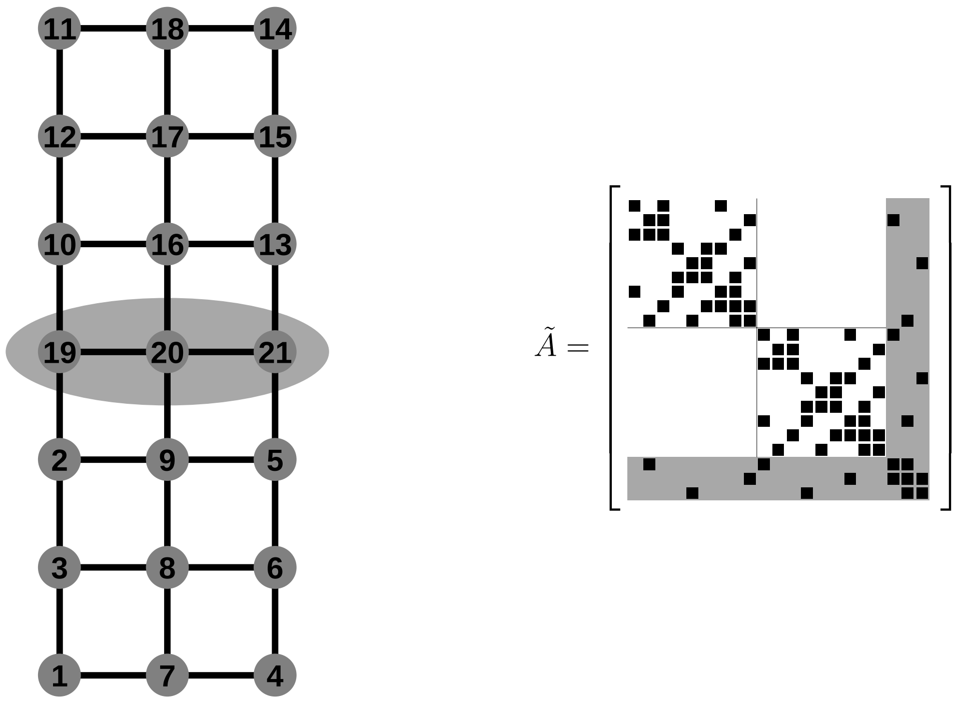


FIGURE 7. A nested-dissection ordering of a mesh (left) and the nested-dissection reordered matrix of the mesh (right). The shaded ellipse in the mesh shows the top-level vertex separators. The shading in the matrix shows the rows and columns that correspond to the separator; the thin lines separate the submatrices that correspond to the connected components of the mesh minus the separator..

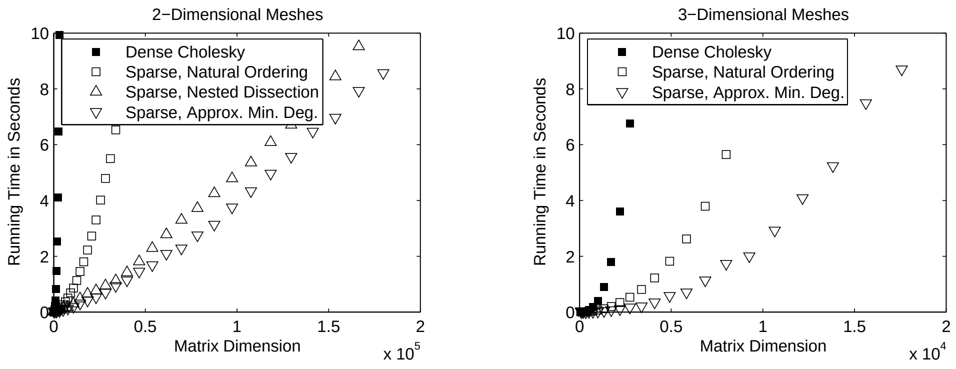


FIGURE 8. The effect of reordering the symmetrically rows and columns on the running time of sparse Cholesky factorizations.

nested-dissection and minimum-degree orderings speed up the factorization considerably.

Unfortunately, nested-dissection-type orderings are usually the best we can do in terms of fill and work in sparse factorization of meshes, both structured and unstructured ones. A class of ordering heuristics called *minimum-degree* heuristics can beat nested dissection on small and medium-size meshes. On larger meshes, various hybrids of nested-dissection and minimum-degree are often more effective than pure variants of either method, but these hybrids have the same asymptotic behavior as pure nested-dissection orderings.

To asymptotically beat sparse Cholesky factorizations, we something more sophisticated than a factorization of the coefficient matrix.

5. Krylov-Subspace Iterative Methods

A radically different way to solve $Ax = b$ is to to find the vector $x^{(k)}$ that minimizes $\|Ax^{(k)} - b\|$ in a subspace of $\mathbb{R}^n$, the subspace that is spanned by $b, Ab, A^2b, \ldots, A^{k-1}b$. This subspace is called the *Krylov* subspace of A and b, and such solvers are called *Krylov-subspace solvers*. This idea is appealing because of two reasons, the first obvious and the other less so. First, we can create a basis for this subspace by repeatedly multiplying vectors by A. Since our matrices, as well as many of the matrices that arise in practice, are so sparse, multiplying A by a vector is cheap. Second, it turns out that when A is symmetric, finding the minimizer $x^{(k)}$ is also cheap. These considerations, taken together, imply that the work performed by such solvers is proportional to k. If $x^{(k)}$ converges quickly to x, the total solution cost is low. But for most real-world problems, as well as for our meshes, convergence is slow.

The *conjugate gradients* method (CG) and the *minimal-residual* method (MINRES) are two Krylov-subspace methods that are appropriate for symmetric positive-definite matrices. MINRES is more theoretically appealing, because it minimizes the residual $Ax^{(k)} - b$ in the 2-norm. It works even if A is indefinite. Its main drawback is that it suffers from a certain numerically instability when implemented in floating point. The instability is almost always mild, not catastrophic, but it can slow convergence and it can prevent the method from converging to very small residuals. Conjugate gradients is more reliable numerically, but it minimizes the residual in the A-norm, not in the 2-norm.

Figure 9 compares the performance of a sparse direct solver (with a fill-reducing ordering) with the performance of a MINRES solver. The

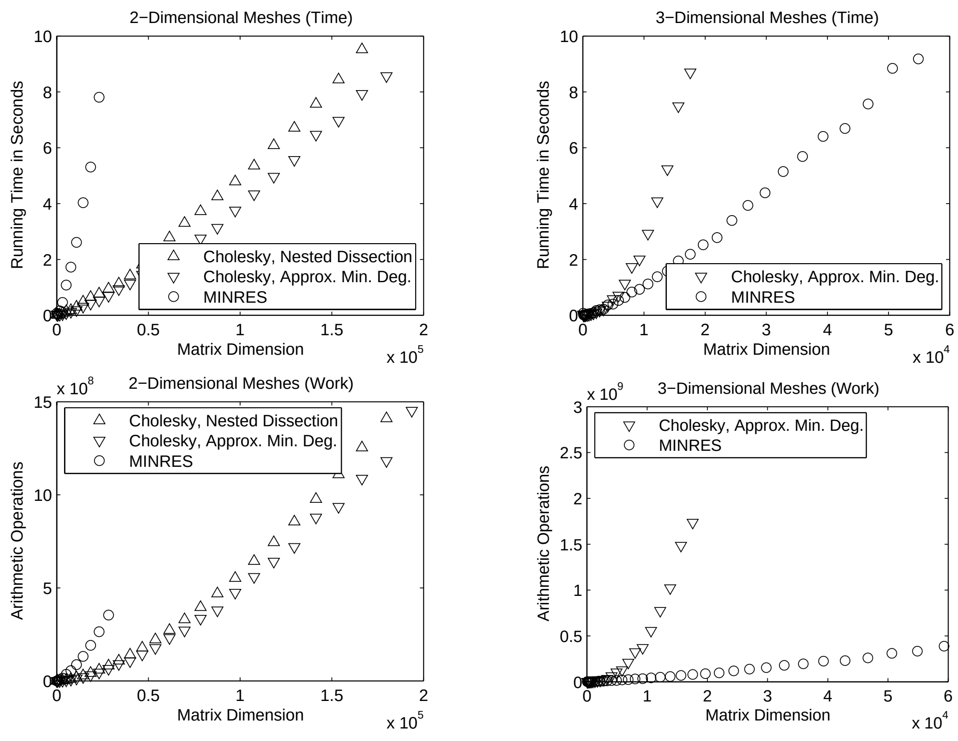
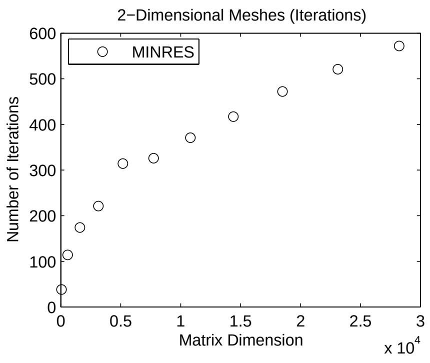


FIGURE 9. A comparison of a sparse direct Cholesky solver with an iterative MINRES solver.

performance of an iterative solver depends on the prescribed convergence threshold. Here we instructed the solver to halt once the relative norm of the residual

$$\frac{\|Ax^{(k)} - b\|}{\|b\|}$$

is 10^{-6} or smaller. On 2-dimensional meshes, the direct solver is faster. On 3-dimensional meshes, the iterative solver is faster. The fundamental reason for the difference is the size of the smallest balanced vertex separators in 2- and 3-dimensional meshes. In a 2-dimensional mesh with a bounded aspect ratio, the smallest balanced vertex separator has $\Theta(n^{1/2})$ vertices; in a 3-dimensional mesh, the size of the smallest separator is $\Theta(n^{2/3})$. The larger separators in 3-dimensional meshes cause the direct solver to be expensive, but they speed up the iterative solver. This is shown in Figure 10: MINRES converges after fewer iterations on 2-dimensional meshes than on 3-dimensional meshes. The full explanation of this behavior is beyond the scope of this chapter, but this phenomenon shows how different direct and iterative solvers are. Something that hurts a direct solver may help an iterative solver.



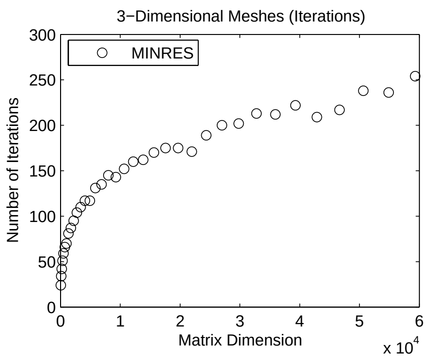


FIGURE 10. The number of iterations required to reduce the relative norm of the residual to 10^6 or less.

As long as A is symmetric and positive definite, the performance of a sparse Cholesky solver does not depend on the values of the nonzero elements of A, only on A's nonzero pattern. The performance of an iterative solver, on the other hand, depends on the values of the nonzero elements. The behavior of MINRES displayed in Figures 9 and 10 depend on our way of assigning values to nonzero elements of A.

Comparing the running times of sparse and iterative solvers does not reveal all the differences between them. The graphs that present the number of arithmetic operations, in Figure 9, reveal another aspect. The differences in the amount of work that the solvers perform are much greater than the differences in running times. This indicates that the direct solver performs arithmetic at a much higher rate than the iterative solver. The exact arithmetic-rate ratio is implementation dependent, but qualitatively, direct solvers usually run at higher computational rates than iterative solvers. Most of the difference is due to the fact that it is easier to exploit complex computer architectures, in particular cache memories, in direct solvers.

Another important difference between direct and iterative solvers is memory usage. A solver like MINRES only allocates a few n-vectors. A direct solver needs more memory, because, as we have seen, the factors fill. The Cholesky factor of a 3-dimensional mesh contains $\Omega(n^{4/3})$ nonzeros, so the memory-usage ratio between a direct and iterative solver worsens as the mesh grows.

6. Preconditioning

How can we improve the speed of Krylov-subspace iterative solvers? We cannot significantly reduce the cost of each iteration, because for matrices as sparse as ours, the cost of each iteration is only $\Theta(n)$.

There is little hope for finding a better approximation $x^{(k)}$ in the span of $b, Ab, \ldots, A^{k-1}b$, because MINRES already finds the approximation that minimizes the norm of the residual.

To understand what *can* be improved, we examine simpler iterative methods. Given an approximate solution $x^{(k)}$ to the system $Ax = b$, we define the error $e^{(k)} = x - x^{(k)}$ and the residual $r^{(k)} = b - Ax^{(k)}$. The error $e^{(k)}$ is the required *correction* to $x^{(k)}$; if we add $e^{(k)}$ to $x^{(k)}$, we obtain x. We obviously do not know what $e^{(k)}$ is, but we know what $Ae^{(k)}$ is: it is $r^{(k)}$, because

$$\begin{aligned} Ae^{(k)} &= A\left(x - x^{(k)}\right) \\ &= Ax - Ax^{(k)} \\ &= b - Ax^{(k)} \\ &= r^{(k)} \,. \end{aligned}$$

Computing the correction requires solving a linear system of equation whose coefficient matrix is A, which is what we are trying to do in the first place. The situation seems cyclic, but it is not. The key idea is to realize that even an inexact correction can bring us closer to x. Suppose that we had another matrix B that is close to A and such that linear systems of the form $Bz = r$ are much earsier to solve than systems whose coefficient matrix is A. Then we could solve $Bz^{(k)} = r^{(k)}$ for a correction $z^{(k)}$ and set $x^{(k+1)} = x^{(k)} + z^{(k)} = x^{(k)} + B^{-1}(b - Ax^{(k)})$. Since $Ae^{(k)} = b - Ax^{(k)}$, we can express $e^{(k)}$ directly in terms of $e^{(0)}$,

$$\begin{aligned} e^{(k+1)} &= x - x^{(k+1)} \\ &= x - x^{(k)} + B^{-1}\left(b - Ax^{(k)}\right) \\ &= e^{(k)} + B^{-1}Ae^{(k)} \\ &= \left(I - B^{-1}A\right)e^{(k)} \\ &= \left(I - B^{-1}A\right)^k e^{(0)} \,. \end{aligned}$$

Therefore, if $\left(I - B^{-1}A\right)^k$ converges quickly to a zero matrix, then $x^{(k)}$ quickly towards x. This is the precise sense in which B must be close to A: $\left(I - B^{-1}A\right)^k$ should converge quickly to zero. This happens if the spectral radius of $(I - B^{-1}A)$ is small (the spectral radius of a matrix is the maximal absolute value of its eigenvalues).

The convergence of more sophisticaded iterative methods, like MINRES and CG, can also be accelerated using this technique, called *preconditioning.* In every iteration, we solve a correction equation $Bz^{(k)} =$

$r^{(k)}$. The coefficient matrix B of the correction equation is called a *preconditioner*. The convergence of preconditioned MINRES and preconditioned CG also depends on spectral (eigenvalue) properties associated with A and B. For these methods, it is not the spectral radius that determines the convergence rate, but other spectral properties. But the principle is the same: B should be close to A in some spectral metric.

In this book, we mainly focus on one family of methods to construct and analyze preconditioners. There are many other ways. The constructions that we present are called *support preconditioners*, and our analysis method is called *support theory*.

We introduce support preconditioning here using one particular family of support preconditioners for our meshes. These preconditioners, which are only effective for regular meshes whose edges all have the same weights (off-diagonals in A all have the same value), are called *Joshi* preconditioners. We construct a Joshi preconditioner B for a mesh matrix A by dropping certain edges from the mesh, and then constructing a coefficient matrix B for the sparsified mesh using the procedure given in Section 1. Figures 11 and 12 present examples of such sparsified meshes.

To solve the correction equations during the iterations, we compute the sparse Cholesky factorization of PBP^T, where P is a fill-reducing permutation P for B.

The construction of Joshi preconditioners depends on a parameter k. For small k, the mesh of the preconditioner is similar to the mesh of A. In particular, for $k = 1$ we obtain $B = A$. As we shall see, Joshi preconditioners with a small k lead to fast convergence rates. But Joshi preconditioners with a small k do not have small balanced vertex separators, so their Cholesky factorization suffers from significant fill. Not as much as in the Cholesky factor of A, but still significant. This fill slows down the preconditioner-construction phase of the linear solver, in which we construct and factor B, and it also slows down each iteration. As k gets larger, the similarity between the meshes diminishes. Convergence rates are worse than for small k, but the factorization of B and the solution of the correction equation in each iteration become cheaper. In particular, when k is at least as large as the side of the mesh, the mesh of B is a tree, which has single-vertex balanced separators; factoring it costs only $\Theta(n)$ operations, and its factor contains only $\Theta(n)$ nonzeros.

Figure 13 presents the performance of preconditioned MINRES with Joshi preconditioners on two meshes. On the three dimensional mesh, the preconditioned iterative solver is much faster than the direct solver

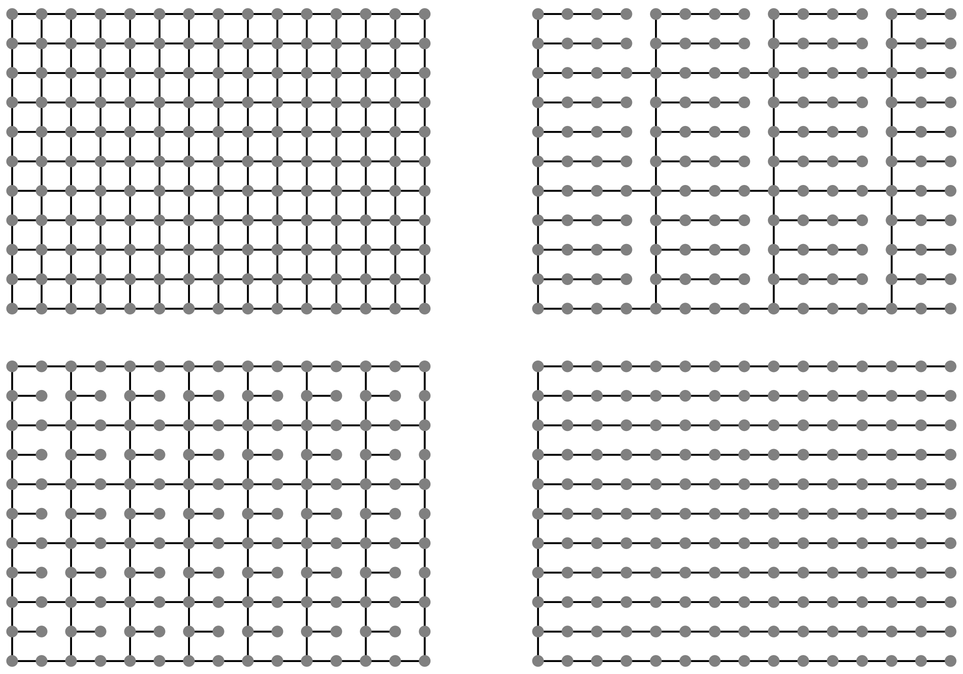


FIGURE 11. A 15-by-11 two-dimensional mesh (top left) with Joshi subgraphs for $k = 4$ (top right), $k = 2$ (bottom left), and $k = 15$ (bottom right). The subgraph for $k = 2$ is the densest nontrivial subgraph and the subgraph for $k = 15$ is the sparsest possible; it is a spanning tree of the mesh.

and about as fast as the unpreconditioned MINRES solver. The preconditioned solver performs fewer arithmetic operations than both the direct solver and the unpreconditioned solver. In terms of arithmetic operations, the Joshi preconditioner for $k = 3$ is the most efficient. For small k, the preconditioned solvers perform fewer iterations than the unpreconditioned solver. The number of iteration rises with k. Perhaps the most interesting aspect of these graphs is the fact that the cost of the factorization of B drops dramatically from $k = 1$, where $B = A$, to $k = 2$. Even slightly sparsifying the mesh reduces the cost of the factorization dramatically. The behavior on the 2-dimensional mesh is similar, except that in terms of time, the cost of the solver rises monotonically with k. In particular, the direct solver is fastest. But in terms of arithmetic operations, the preconditioned solver with $k = 2$ is the most efficient.

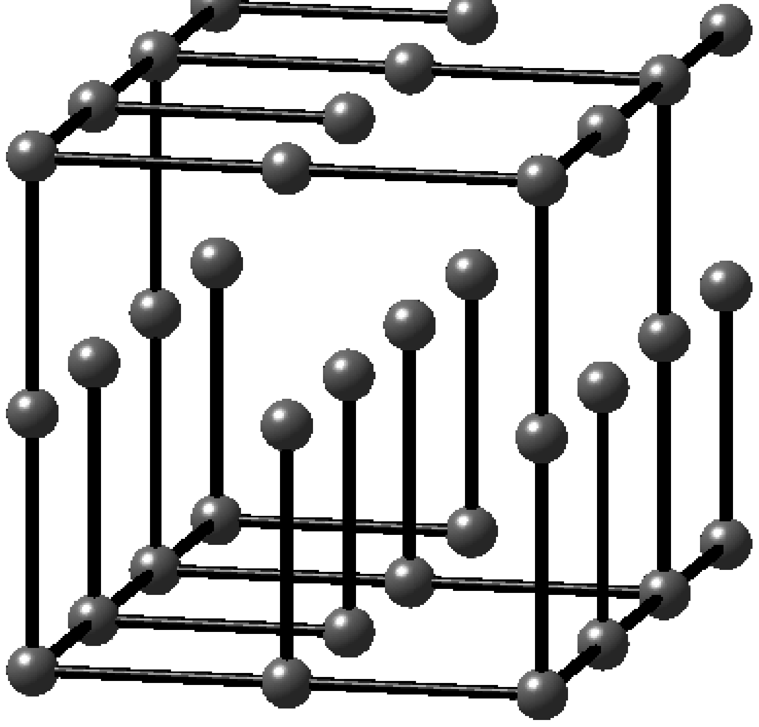


FIGURE 12. A 3-by-4-by-3 three-dimensional Joshi mesh.

7. The Main Questions

We have now set the stage to pose the main questions in support theory and preconditioning. The main theme in support preconditioning is the construction of preconditioners using graph algorithms. Given a matrix A, we model it as a graph G_A. From the G_A we build another graph G_B, which approximates G_A in some graph-related metric. From G_B we build the preconditioner B. The preconditioner is usually factored as preparation for solving correction equations $Bz = r$, but sometimes the correction equation is also solved iteratively. This raises three questions

(1) How do we model matrices as graphs?
(2) Given two matrices A and B and their associated graphs G_A and G_B, how do we use the graphs to estimate or bound the

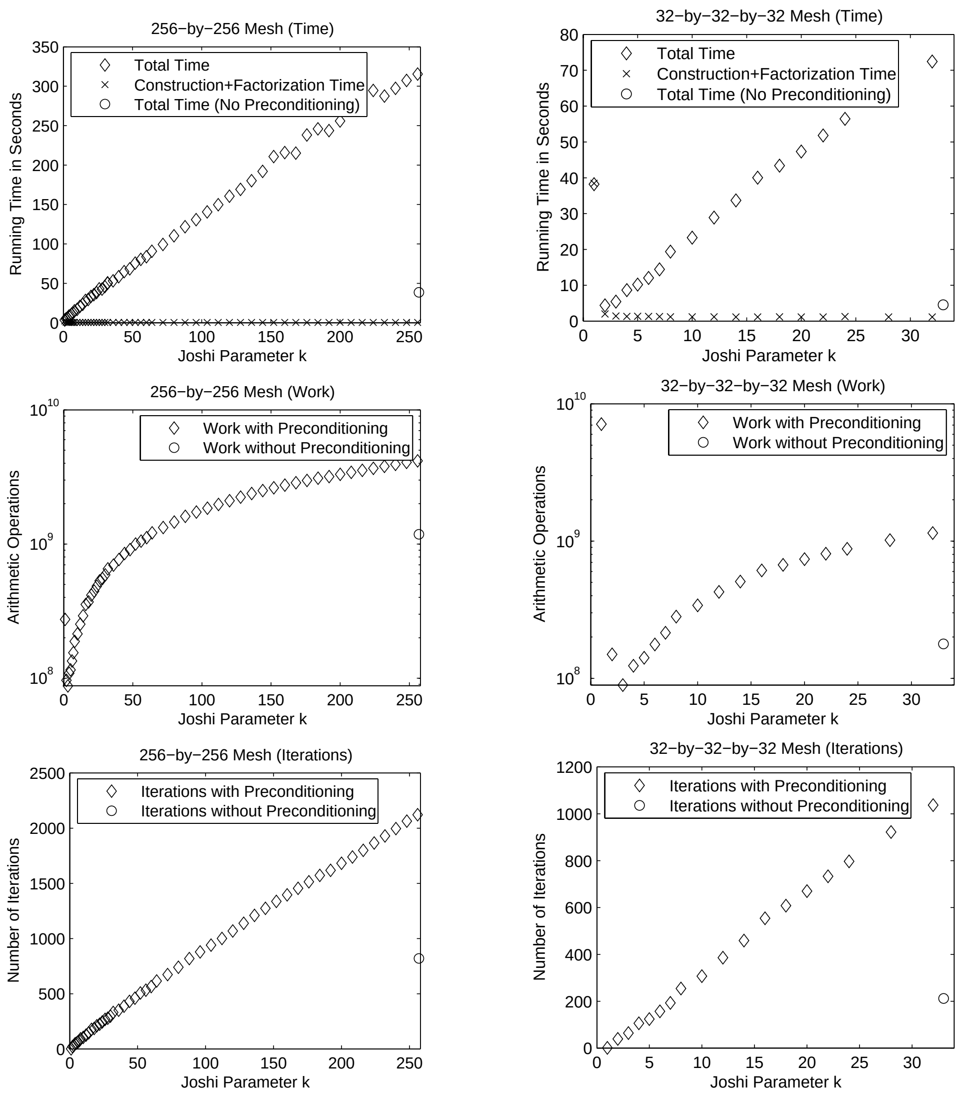


FIGURE 13. The behavior of preconditioned MINRES solvers with Joshi preconditioners. For $k = 1$, the performance is essentially the performance of the direct solver. The preconditioners were reordered using a minimum degree ordering and factored.

convergence of iterative linear solvers in which B preconditions A?

(3) Given the graph G_A of a matrix A, how do we construct a graph G_B in a metric that ensures fast convergence rates?

Support theory addresses all three questions: the modeling question, the analysis qustion, and the graph-approximation question.

The correspondence between graphs and matrices is usually an isomorphism, to allow the analysis to yield useful convergence-rate bounds. It may seem useless to transform one problem, the matrix-approximation probem, into a completely equivalent one, a graph-approximation question. It turns out, however, that many effective constructions only make sense when the problem is described in terms of graphs. They would not have been invented as matrix constructions. The Joshi preconditioners are a good example. They are highly structured when viewed as graph constructions, but would make little sense as direct sparse-matrix constructions. The analysis, on the other hand, is easier, more natural, and more general when carried out in terms of matrices. And so we transform the orignal matrix to a graph to allow for sophisticaed constructions, and then we transform the graphs back to matrices to analyze the convergence rate.

The preconditioning framework that we presented is the main theme of support theory, but there are other themes. The combinatorial and algebraic tools that were invented in order to develop preconditioners have also been used to analyze earlier preconditioners, to bound the extreme eigenvalues of matrices, to analyze the null spaces of matrices, and more.