Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems

Keith D. Gremban

October 1996
CMU-CS-96-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:

Gary Miller, Chair
Guy Blelloch
Paul Heckbert
Bruce Maggs
Omar Ghattas
Mike Heroux, Cray Research

This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, and the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330, and in part by NSF Grant CCR-9016641. Cray C-90 computing time was provided by the Pittsburgh Supercomputing Center under Grant ASC890018P. The U.S. government is authorized to reproduce and distribute reprints for government purposes, notwithstanding any copyright notation thereon.

Keywords: linear systems, iterative methods, preconditioners, sparse and very large systems, parallel algorithms

Abstract

Solution of large, sparse linear systems is an important problem in science and engineering. Such systems arise in many applications, such as electrical networks, stress analysis, and more generally, in the numerical solution of partial differential equations. When the coefficient matrices associated with these linear systems are symmetric and positive definite, the systems are often solved iteratively using the preconditioned conjugate gradient method. We have developed a new class of preconditioners, which we call *support tree* preconditioners, that are based on the combinatorial properties of the graphs corresponding to the coefficient matrices of the linear systems. We call the resulting iterative method *support tree conjugate gradient*, or STCG. These new preconditioners are applicable to the class of symmetric and diagonally dominant matrices, and have the advantage of being well-structured for parallel implementation, both in construction and in evaluation. In this thesis, we present the intuition, construction, implementation, and analysis of STCG.

STCG is based upon an interesting isomorphism between a certain class of matrices (which we call Laplacian matrices), edge-weighted undirected graphs, and resistive networks. Using this isomorphism, we show that an iterative method can be interpreted in terms of these discrete structures. Based on this interpretation, the STCG method for accelerating convergence is developed, which involves constructing other, more efficient discrete structures called support trees, and using their interpretation as matrices to apply them as preconditioners. Interestingly, the matrix preconditioners used in STCG are larger, but sparser than conventional preconditioners. Additionally, the construction of support trees is basically an application of recursive divide-and-conquer. Support trees have very regular structures and are very well-suited for parallel implementation.

Through theoretical analysis and numerical experiments, we show that STCG is practical and efficient for the parallel solution of large sparse linear systems with Laplacian coefficient matrices. STCG is an interesting example of combinatorial techniques being applied to solve an algebraic problem. These techniques have wider applicability than the acceleration of iterative techniques. We also demonstrate an application of these techniques to the more general problem of bounding eigenvalues.

Acknowledgments

My graduate work at CMU was a long process involving a number of cycles of elation, excitement, and frustration. I entered the program after several years of working in a robotics research group at an aerospace company and expected to find a thesis topic and graduate in record time. Was I in for a shock, or rather a series of shocks!

Without going into detail, suffice it to say that I encountered a number of setbacks. Fortunately, though, the structure of the Computer Science Ph.D. program at CMU is such that I was given time to overcome these difficulties. I am grateful to Chuck Thorpe, Takeo Kanade, Katsushi Ikeuchi, and Gary Miller for having the patience and the confidence in me to help me through this period. I am also grateful to Roger Schappell for creating a work environment that enabled me to finish this thesis after having physically left CMU.

It would not have been possible for me to have finished the program without lots of help from friends. The list is far too long to detail here, but I would especially like to thank Doug Reece for all the time spent listening to my troubles on long runs through Schenley Park. Tracy Ann Lewis was a wonderful listener over coffee throughout my years at CMU.

I couldn't have made it through my last several years there without the help of the Carnegie Mellon Shotokan Karate Club. Karate is an amazingly effective way to focus one's mind and relieve stress! I need to specifically acknowledge my immediate seniors, Mark Ciancutti and Bruce Schmidt, as well as the many junior members who helped me out, especially the Wednesday afternoon sparring club, which included Rahul Sukthankar, Ruth Chabay, Yoichi Sato, Sanjay Sachdev, and Tony Lattanze.

Of course, I have to thank my parents, Joe and June Gremban, who are responsible for my scientific curiousity. Special thanks go to my family, who put up with me for all this time: my wife Kathy, and my two daughters, Stephanie and Kelly.

My final thanks are to my late dachshund, Random Variable, who gave his life for this thesis, and again to my wife, Kathy, who gave up having a life because of this thesis.

Table of Contents

1 Introduction

Consider the solution of linear systems of the form

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1.1}$$

where A is an nxn matrix, and both $\boldsymbol{x}$ and $\boldsymbol{b}$ are nx1 vectors. Of special interest is the case where A is large and sparse. Systems of this sort arise frequently in many applications such as electrical networks, tomography, diffusion, and structural mechanics [Axelsson (1994)]. We are especially interested in linear systems that arise from the solution of elliptic boundary value problems by either the finite element or finite difference methods.

The term *sparse* above refers to the relative number of non-zeros in the matrix A. An nxn matrix A is considered to be *sparse* if A has only $O(n)$ non-zero entries. In this case, the majority of the entries in the matrix are zeros, which do not have to be explicitly stored. An nxn dense matrix has $\Omega(n^2)$ non-zeros. One of the goals of dealing with sparse matrices is to make efficient use of the sparsity in order to minimize storage throughout the computations, as well as to minimize the required number of operations. Sparse linear systems are often solved using different computational techniques than those employed to solve dense systems.

The term *large* varies with respect to the current generation of computers, and with the sparsity of the system. A dense nxn system requires $\Omega(n^2)$ storage vs. $O(n)$ for a sparse matrix of the same order. Nonetheless, some generalizations can be made with respect to the size of feasible linear systems. Where at one time (1960's) dense linear systems with $n > 100$ were considered large, such systems can be solved easily on the current (1995) generation of scientific workstations. In 1995, dense linear systems with $n > 10{,}000$, and sparse linear systems with $n > 100{,}000$ are considered large by most computational scientists, and n will continue to grow. As systems with $n = 10{,}000$ become the norm, the desire for more resolution and more accurate simulations will push n to become larger and larger.

Since large linear systems may require intensive computational resources to solve, it is important to take advantage of any special information about the coefficient matrices that is available. Consequently, a host of techniques exist that are applicable to specific kinds of matrices. The techniques presented in this thesis are no different: they are applicable to a specific class of matrices which we call Laplacian matrices. These matrices are characterized by being symmetric, diagonally dominant[1], and containing non-positive off-diagonals. Laplacian matrices frequently arise in the

1. An nxn matrix $A = [a_{ij}]$ is *diagonally dominant* if, for all $i \in \{1, 2, \ldots, n\}$, $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$.

solution of elliptic boundary value problems. For example, elliptic problems that are discretized with the finite element method using linear, triangular elements yield coefficient matrices that are Laplacian provided that all the angles of the elements are less than 90°.

There are two broad categories of methods for solving linear systems: *direct* and *iterative*.

A direct method for solving the system of equations (1.1) is any method that produces the solution $\boldsymbol{x}$ after a finite number of operations [Axelsson and Barker (1984)]. An example of a direct method is using Gaussian elimination to factor A into matrices L and U where L is lower triangular and U is upper triangular, then solving the triangular systems by forward and back substitution. Direct methods are typically preferred for dense linear systems. The problem with direct methods for sparse systems is that the amount of computational effort and storage required can be prohibitive. For example, consider the case of a nxn finite element mesh defined for Laplace's equation on the unit square. The coefficient matrix A in this case is n^2xn^2, and will have $O(n^2)$ non-zero elements. A naive direct solution may lead to storage of $O(n^4)$ and computational effort of $O(n^6)$ [Axelsson and Barker (1984), Golub and Ortega (1993)]. Nested dissection is a technique for reducing storage requirements that has been shown to be asymptotically optimal. However, even an efficient direct solution method utilizing nested dissection may require $O(n\log n)$ storage and $O(n^3)$ computations [Axelsson and Barker (1984), Hoffman, Martin, and Rose (1973)].

An alternative to direct methods of solution are iterative methods, which involve the construction of a sequence $\{\boldsymbol{x}^{(i)}\}$ of approximations to the solution $\boldsymbol{x}$, for which $\boldsymbol{x}^{(i)} \rightarrow \boldsymbol{x}$. Many different iterative methods have been developed. Possibly the most basic iterative method is the Jacobi method, defined by $\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - D^{-1}(A\boldsymbol{x}^{(n)} + \boldsymbol{b})$, where D is the matrix containing only the main diagonal of A. The basic first-order linear stationary iterative method involves repeating a *predict-test-correct* cycle until the result produced with the approximate solution is sufficiently accurate [Hageman and Young (1981)]. Every *test* step requires the computation of $A\boldsymbol{x}^{(i)}$. Iterative methods are storage efficient — the basic iterative method requires only $O(n^2)$ storage for an nxn coefficient matrix. Moreover, each step of an iterative method requires only $O(n^2)$ operations, as well. An iterative method is more efficient than the best direct method if the number of iterations required can be held to less than $O(n)$.

The *conjugate gradients* (CG) method is a popular and efficient iterative method that can be used whenever the coefficient matrix is symmetric and positive definite.[1] Applied in its basic form to an n^2xn^2 coefficient matrix, CG may require $O(n^2)$ iterations, which is not an improvement over the best direct method. However, through the use of a technique known as *preconditioning*, the convergence of CG can be accelerated. The resulting method is known as the method of *preconditioned conjugate gradients* (PCG). The method presented in this thesis is a variant of PCG that we call *support tree conjugate gradient*, or STCG.

Parallel performance of a solution method is an important issue. The size of the linear systems that are being solved has continued to grow over recent years. Linear systems can easily be formulated that exceed the capability of current serial computers to solve. Consequently, the use of parallel computers is required and it is necessary to design new algorithms that can simultaneously take advantage of the sparsity and parallel potential of a linear system.

Direct methods for sparse matrices have proven difficult to parallelize. The most common direct methods involve using Gaussian elimination to factor the coefficient matrix into two triangular matrices, and then performing two successive triangular solves. Some progress has been made in parallel factorization, but triangular solution is recognized as the major bottleneck to effective parallelization [Heath, *et al* (1990)].

In contrast, the primary operation in iterative methods for sparse matrices is matrix-vector multiplication, which can be fairly efficiently parallelized for dense matrices, as well as for sparse matrices with regular structure [Blelloch, *et al* (1993)]. Therefore, iterative methods are attractive for parallel implementation if the number of iterations can be kept small by accelerating the rate of convergence. Convergence acceleration requires the use of preconditioners, however, and these form a bottleneck to parallel implementation. This is because every iteration of a preconditioned iterative method requires the solution of a linear system involving the preconditioner. Typically, this is done using a direct method: the preconditioner is factored into triangular matrices, and two triangular solves must be performed.

1. Recall that A is symmetric if $A^t = A$. A is positive definite if, for any vector $x \neq 0$, $\boldsymbol{x}^t A\boldsymbol{x} > 0$.

The factorization is performed only once, and the factors are stored for use in every iteration. However, the two triangular solves must be performed at every iteration, and, as noted above, sparse triangular solves are difficult to parallelize efficiently. Several studies have shown that the use of preconditioners can actually decrease the parallel performance of iterative methods [Greenbaum, *et al* (1989), Heroux, *et al* (1991)].

A variant of iterative methods are the multigrid methods, which are primarily applicable to the solution of linear systems resulting from the discretization of partial differential equations. Multigrid methods can be efficiently parallelized, and require very few iterations to converge. However, in addition to being somewhat limited in applicability, multigrid methods typically require extensive knowledge of the meshing and discretization processes. Such information is often unavailable, and so multigrid methods lack generality.

We can briefly summarize the state-of-the-art in solving linear systems as follows:

- the size of the linear systems to be solved can be expected to increase indefinitely;
- parallel machines are required to solve the largest linear systems;
- iterative methods are promising for parallel implementation, if convergence is sufficiently rapid;
- convergence can be accelerated through the use of preconditioners, but the use of preconditioners can adversely affect the parallel performance of the iterative method.

On the basis of the summary above, we can conclude that the development of an efficient parallel preconditioner would be a significant advance in the state-of-the-art for solving sparse linear systems on parallel machines.

In this thesis, we present the design and analysis of a new parallel iterative method for the solution of certain types of large, sparse linear systems. The systems that we shall consider are large, sparse, symmetric, and diagonally dominant with non-positive off-diagonals. The new method is an extension of the PCG method, and is characterized by the form of the preconditioner. We call the new method *support tree conjugate gradients*, or STCG. STCG is unusual in that the preconditioner is derived from an analysis of the combinatorial properties of the linear system, rather than the algebraic properties. We shall show, both analytically and empirically, that STCG has the following characteristics:

- the rate of convergence of STCG is superior to most variants of PCG;
- the support tree preconditioners are straightforward to construct, given the coefficient matrix;
- support tree preconditioners are large, but very sparse, therefore requiring
    - relatively little storage (usually less than the original coefficient matrix);
    - relatively little processing per iteration (usually less than that required by a matrix-vector multiplication involving the original coefficient matrix);
- STCG is efficient to implement on parallel machines;
- STCG is nearly as efficient on serial machines as many standard variants of PCG.

In addition, a major contribution of this thesis is a new method for analyzing matrices. The new analytic method relies on the combinatorial properties of the graphs that are associated with matrices, and enables proofs of various matrix properties without explicitly referring to the underlying vector spaces. Instead, the proofs involve determining bounds on flows in networks, and bounds on various properties of network embeddings. Using this new method, we obtain bounds on the largest eigenvalues of matrices, and prove theoretical bounds on the convergence rate of STCG.

The format of this thesis is as follows. In the next chapter, we review the current state of the art in direct and iterative methods and show in more detail why a new method is necessary. Following the review, we present the implementation of STCG. This is followed by a theoretical analysis of the properties of STCG which is divided into two chapters. Next, numerical experiments are presented that compare the performance of STCG with that of other common variants of PCG. Finally, STCG is extended to a larger class of problems, and some interesting extensions and applications of the theory developed for the analysis of STCG are presented.

2
Background

This thesis involves the design and analysis of a new method for solving linear systems. A basic understanding of linear algebra and the techniques and tools used to analyze and solve linear systems is therefore necessary. In addition, the method developed in this thesis is based on graph theory, so a basic understanding of this topic is necessary as well. Below, we first establish some basic definitions and notation. Following that, we review some basic results in graph theory, direct methods for the solution of linear systems, and iterative methods for the solution of linear systems.The reader already familiar with this material is encouraged to skip this chapter.

2.1 Definitions and Notation

$\Re$ will be used to denote the set of real numbers. $\Re^+$ will denote positive real numbers, while $\Re^-$ will denote negative real numbers.

A *column vector* $\boldsymbol{x}$ is an nx1 array of numbers indexed by row. $\boldsymbol{x}_i$ denotes the element in row i of vector $\boldsymbol{x}$. An nx1 vector will sometimes be referred to as a n-vector. We will denote vectors with boldface italic lower-case letters. In general, a vector may have complex elements, but in this thesis, we shall only be concerned with vectors having real elements.

A collection of vectors $\{\boldsymbol{x}_1,...,\boldsymbol{x}_n\}$ is called *linearly independent* if, for any collection of scalars $\{\alpha_1,...,\alpha_n\}$ not all equal to zero, $\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i \neq \boldsymbol{0}$. Conversely, if there exists a collection $\{\alpha_1,...,\alpha_n\}$ with some $\alpha_i \neq 0$ and $\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i = \boldsymbol{0}$, then the collection $\{\boldsymbol{x}_1,...,\boldsymbol{x}_n\}$ is said to be *linearly dependent*.

A *matrix A* is an mxn array of numbers indexed by row and column. Thus, $A(i,j)$ is the element in row i and column j of the matrix A. An n-vector is a special case of a matrix with a single column. In this thesis, we will be considering *square* matrices in which $m = n$. An nxn matrix A is said to be of order n. We will denote matrices with italic capital letters. The element $A(i,j)$ will often be denoted by a_{ij}. In general, the elements of a matrix may be complex numbers, although in this thesis, we will only be concerned with matrices having real elements.

The *rank* of an *mxn* matrix *A*, rank(*A*), is the number of linearly independent rows/columns of *A*. An *nxn* matrix *A* is said to be of *full rank* if $\mathrm{rank}(A) = n$.

The *main diagonal* of an *nxn* matrix *A* is the set of elements indexed by a_{ii} for $i = 1,...,n$, and is denoted by *diag*(*A*). A matrix *D* is said to be *diagonal* if all elements off the main diagonal are zero.

A *nxn* matrix *A* is *lower (upper) triangular* if all elements above (below) the main diagonal are zero. *A* is *unit lower (upper) triangular* if *A* is lower (upper) triangular and all the elements on the main diagonal are 1. *A* is *strictly lower (upper) triangular* if *A* is lower (upper) triangular and all elements of the main diagonal are zero.

The *transpose* of an *mxn* matrix *A* is denoted by A^t, and is the *nxm* matrix defined by $A^t(i, j) = A(j, i)$. The transpose of a column vector is a row vector.

The *inner product* of two *n*-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is denoted $(\boldsymbol{x},\boldsymbol{y})$, and is defined by $(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} x_i y_i$. Note that $(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{y}, \boldsymbol{x})$.

The *magnitude* of a vector $\boldsymbol{x}$ is denoted $\|\boldsymbol{x}\|$, and is given by $\|\boldsymbol{x}\| = (\boldsymbol{x}, \boldsymbol{x})^{1/2}$. A vector with magnitude 1 is called a *unit vector.*

A matrix is *sparse* if the ratio of zero to non-zero elements is large. An *nxn* matrix *A* is sparse if *A* contains only *O*(*n*) non-zeros. The location of the non-zeros in a sparse matrix *A* is called the *sparsity pattern* of *A*.

The *scalar product* of a matrix/vector with a scalar α is the component-wise product of α with the elements of the matrix/vector: $(\alpha * A)(i,j) = \alpha * A(i,j)$, and $(\alpha * x)_i = \alpha * x_i$.

The *matrix-vector product* of an *mxn* matrix *A* with a *p*-vector $\boldsymbol{x}$ is only defined for *n*=*p*. The result is an *m*-vector $\boldsymbol{y}=A\boldsymbol{x}$ given by $y_i = \sum_{j=1}^{n} a_{ij} x_j$.

The *matrix-matrix product* of an *mxn* matrix *A* with a *pxq* matrix *B* is only defined for *n*=*p*. The result is an *mxq* matrix *C*=*AB* given by $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

The *identity matrix* of order *n* is the *nxn* matrix *I* for which $a_{ii}=1$, $i=1,...,n$, and $a_{ij}=0$ for $i \neq j$. The identity matrix *I* has the property that for any vector $\boldsymbol{x}$, $I\boldsymbol{x}=\boldsymbol{x}$.

A *permutation matrix* is an identity matrix with rows (columns) reordered. Thus, a permutation matrix *P* has exactly one 1 in each row and column, and is zero elsewhere. Left multiplication by a permutation matrix interchanges rows, while right multiplication interchanges columns.

The inverse of an *nxn* matrix *A* is denoted by A^{-1}, and is the unique *nxn* matrix defined by $AA^{-1} = A^{-1}A = I$. The inverse of a matrix only exists when there is a unique solution to the equation $A\boldsymbol{x} = \boldsymbol{b}$. When *A* has an inverse, *A* is said to be *non-singular.* Conversely, a matrix without an inverse is *singular.* An *nxn* matrix *A* is non-singular if and only if *A* is of full rank.

An *nxn* matrix *A* is *symmetric* if $A^t = A$.

An *nxn* matrix A is *diagonally dominant* if, $\forall i = 1 \ldots n$, $a_{ii} \geq \sum_{j=1, j \neq i}^{n} |a_{ij}|$; that is, *A* is diagonally dominant if, for each row, the sum of the absolute values of the off-diagonal elements is less than or equal to the value of the diagonal element. *A* is *strictly diagonally dominant* if $\exists i$ such that $a_{ii} > \sum_{j=1,\, j \neq i}^{n} |a_{ij}|$; that is, *A* is strictly diagonally dominant if *A* is diagonally dominant and at least one of the diagonal elements is strictly larger than the corresponding off-diagonal absolute sum.

Let *A* be an *nxn* matrix. *A* scalar λ is an *eigenvalue* of *A* if there exists an *n*-vector $\boldsymbol{x}$, with $\boldsymbol{x} \neq 0$, such that $A\boldsymbol{x} = \lambda\boldsymbol{x}$. The vector $\boldsymbol{x}$ is said to be the *eigenvector* corresponding to the eigenvalue λ. The collection of eigenvalues of A is called the *spectrum* of *A*, and is denoted $\lambda(A)$. The *spectral radius* of *A* is denoted $\rho(A)$, and is given by the largest magnitude of any eigenvalue of *A*. That is, $\rho(A) = \max\{|\lambda| : \lambda \in \lambda(A)\}$. In general, an eigenvalue may be real or complex. However, in this thesis, we are only concerned with real symmetric matrices, and these matrices have only real eigenvalues.

Let *A* and *B* be *nxn* matrices. A scalar λ is a *generalized eigenvalue* of the ordered pair of matrices (*A*, *B*) if there exists an *n*-vector $\boldsymbol{x}$, with $\boldsymbol{x} \neq 0$, such that $A\boldsymbol{x} = \lambda B\boldsymbol{x}$. The vector $\boldsymbol{x}$ is said to be the *generalized eigenvector* corresponding to the generalized eigenvalue λ. We denote the collection of generalized eigenvalues of (*A*,*B*) by $\lambda(A,B)$.

An *nxn* matrix *A* is *positive* (*negative*) *definite* if, $\forall \boldsymbol{x}$, $\boldsymbol{x}^t A\boldsymbol{x} > 0$ (< 0). Equivalently, *A* is *positive (negative) definite* if, $\forall \lambda \in \lambda(A)$, $\lambda > 0$ $(\lambda < 0)$.

An *nxn* matrix *A* is *positive* (*negative*) *semi-definite* if, $\forall \boldsymbol{x}$, $\boldsymbol{x}^t A\boldsymbol{x} \geq 0$ (≤ 0). Equivalently, *A* is *positive (negative) semi-definite* if, $\forall \lambda \in \lambda(A)$, $\lambda \geq 0$ $(\lambda \leq 0)$.

Let *A* be an *nxn* positive definite matrix. The *spectral condition number* of *A* is denoted by $\kappa(A)$, and is given by the ratio of the largest to smallest eigenvalues of *A*. That is, let $\lambda(A) = \{\lambda_1 \leq \ldots \leq \lambda_n\}$. Then $\kappa(A) = \lambda_n / \lambda_1$. Let *B* also be *nxn* and positive definite. The g*eneralized condition number* of (*A*,*B*) is denoted by $\kappa(A,B)$, and is given by the ratio of the largest to smallest generalized eigenvalues of (*A*,*B*). Note that $\kappa(A,B) = \kappa(B^{-1}A)$.

An *nxn* matrix *L* is a *Laplacian matrix*, or *Laplacian*, if *L* is real, symmetric, and diagonally dominant with non-positive off-diagonals.

An *nxn* matrix *L* is a *generalized Laplacian matrix* (*generalized Laplacian*) if *L* is real, symmetric, and diagonally dominant.

2.2 Graph Theory

In this section, we will review some basic, relevant results in graph theory. Further details can be obtained from books on graph theory, such as the texts by Chartrand (1977) or Harary (1969). First, we start with the following basic definitions.

An *undirected graph G* = (*V*,*E*) is a collection *V* of *nodes* or *vertices*, together with a set *E* of *edges* or *arcs* where each edge in *E* is an unordered pair of nodes. A *self-loop* is an edge in which the vertices are identical. We denote the cardinality of a set of vertices *S* by |*S*|, and the cardinality of a set of edges *E* by |*E*|. An undirected graph is depicted as a set of points connected by lines.

A graph *G* = (*V*,*E*) is said to be *ordered* if each of the *n* vertices in *V* is assigned a unique number in the range 1,...,*n*; such an assignment is called an *ordering*. Given an ordered graph G, we will denote vertices by $V = \{v_1, \ldots, v_n\}$, and edges by $E = \{e_1, \ldots, e_m\}$, where $e_i = (v_j, v_k) = (v_k, v_j)$ for some *j*,*k*.We will assume that all graphs are ordered.

Let *G* = (*V*,*E*) be a graph. If $e_i = (v_j, v_k) \in E$, then vertices v_j and v_k are called *adjacent*, denoted v_j adj v_k. Let $v \in V$; the *degree* of *v*, deg(*v*), is the number of distinct vertices adjacent to *v*.

A *complete graph* is a graph in which all vertices are pairwise adjacent. We denote by K_n the complete graph on n vertices.

A *walk* is an alternating sequence of vertices and edges that begins and ends with a vertex, such that any edge in the sequence connects the vertex preceding it to the vertex following it.

A *path* is a walk in which all the vertices are distinct.

A *cycle* is a walk in which the first and last vertex are the same.

A graph is *connected* if there exists a path between every pair of vertices. Let $G_1,...,G_m$ be subgraphs of G such that each G_j is connected and there exist no edges between G_j and G_k for $j \neq k$; then the G_j are called the c*onnected components* of G.

A *tree* is a connected graph with no cycles. A *forest* is a graph with no cycles, and is therefore a collection of trees.

A *directed graph G* is a graph in which the edges are ordered pairs; that is, $(v_j,v_k) \neq (v_k,v_j)$. For an edge $e = (v_j,v_k)$, v_j is termed the tail of the edge, and v_k is the head. A directed graph G is depicted as a set of points connected by lines with arrowheads denoting the orientation from tail to head.

A *weighted graph G* (directed or undirected) is a graph together with a function $w{:}E \rightarrow \Re$, which assigns weights to edges.

Let $G = (V(G),E(G))$ and $H = (V(H),E(H))$ be graphs. H is a *subgraph* of G if $V(H) \subseteq V(G)$, and $E(H) \subseteq E(G)$. G is then a *supergraph* of H.

Let $S \subseteq V(G)$. Let H be the subgraph of G given by $V(H) = S$, and $(v_i,v_j) \in E(H)$ iff $v_i \in S$ and $v_j \in S$. Then H is the subgraph of G *induced* by the set S.

Let $G = (V,E)$ be a graph and $S \subseteq V$. Let H be the subgraph of G induced by S. Then the *frontier*, or *boundary*, of H is the set of edges (v_i, v_j) such that either $v_i \in S$ and $v_j \notin S$, or $v_i \notin S$ and $v_j \in S$.

Let G and H be graphs. An *embedding* of H into G is a mapping of vertices of H onto vertices of G, and edges of H onto paths in G. The *dilation* of the embedding is the length of the longest path in G onto which an edge of H is mapped; we denote the dilation of the embedding by $\delta(G,H)$. The *congestion* of an edge e in G is the number of paths of the embedding that contain e. The *congestion* of the embedding is the maximum congestion of the edges in G. We denote the congestion of the embedding by $\gamma(G,H)$.

2.2.1 Graphs and matrices

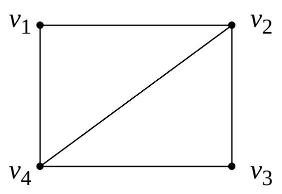
Graphs and matrices are related in a variety of interesting ways. In this section, we present an overview of the relationships that are important to understanding this thesis. Further details can be found in the books by Varga (1962) and George and Liu (1981). In the succeeding chapters of this thesis, we will use some of the relationships between graphs and matrices to develop a new approach to accelerating the convergence of the preconditioned conjugate gradient method. The preconditioned conjugate gradient method was derived in terms of the algebraic properties of vectors and matrices; in contrast, the approach developed in this thesis is derived in terms of the combinatorial, or graph-theoretic properties of the matrices. In this section, we review some of the basic properties that relate graphs and matrices.

Let $G = (V,E)$ be an ordered, undirected graph with n vertices and no self-loops. Then the *adjacency matrix A* corresponding to G can be constructed as follows:

- $a_{ij} = a_{ji} = 1$ if $e = (v_i,v_j) \in E$;

- $a_{ij} = 0$ otherwise.

Figure 2.1 illustrates an undirected graph and its corresponding adjacency matrix. Note that the adjacency matrix is symmetric, but has all zeros on the diagonal.



$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 2.1: ***An Undirected Graph and Corresponding Adjacency Matrix.***

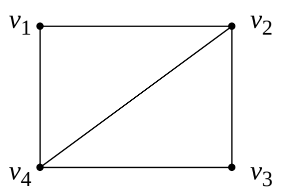
Let $\boldsymbol{p}$ be an ordering of the numbers $1,...,n$. Then $\boldsymbol{p}$ is a list of length n, $\boldsymbol{p}(i) \in \{1,...,n\}$, and $\boldsymbol{p}(i) \neq \boldsymbol{p}(j)$ for $i \neq j$. $\boldsymbol{p}$ can be used to construct a permutation matrix $P = P(\boldsymbol{p})$ by setting $P(i,\boldsymbol{p}(i)) = 1$, for $i = 1,...,n$, and $P(i,j) = 0$ elsewhere. $\boldsymbol{p}$ defines a new labeling of the vertices of G. The corresponding adjacency matrix B that reflects the new ordering is given by $B = PAP^t$. The topology of the graph underlying A and B is the same, only the labeling of the vertices has changed.

It is easy to show that if a graph G consists of exactly m connected components $G_1,...,G_m$, then there exists an ordering of G such that the adjacency matrix of G is block diagonal with exactly m diagonal blocks [see George and Liu (1981)]. Equivalently, let A be the adjacency matrix of G. Then there exists a permutation matrix P such that $B = PAP^t$ is block diagonal with m diagonal blocks.

A generalization of the adjacency matrix is the *unit Laplacian matrix.* Given an undirected graph $G = (V,E)$, the unit Laplacian, L, is defined as follows:

- $l_{ij} = l_{ji} = -1$ if $e = (v_i,v_j) \in E$;
- $l_{ii} = \deg(v_i)$;
- $l_{ij} = 0$ otherwise.

(Some authors refer to the unit Laplacian as simply the Laplacian of a graph [for example, see Pothen, Simon, and Liou (1990)]. We prefer to reserve the term *Laplacian* for a generalization to weighted undirected graphs that will be presented in Chapter 4.) Figure 2.2 illustrates the graph from Figure 2.1, and the corresponding unit Laplacian matrix. Note the differences between the adjacency matrix and the unit Laplacian: the Laplacian has a non-zero diagonal and non-positive off-diagonals. The sparsity pattern of the off-diagonal elements is the same for the Laplacian and the adjacency matrix.



$$\begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$

Figure 2.2: ***An Undirected Graph and Corresponding Unit Laplacian Matrix.***

The unit Laplacian has some interesting properties related to the structure of the graph. Let G be a graph and L the corresponding unit Laplacian. It is easy to show that all the rows and columns of L sum to zero. Therefore, the vector $\mathbf{1}$, which consists of all ones, is an eigenvector of L with eigenvalue 0; that is, $L\mathbf{1} = \mathbf{0}$. Moreover, if G consists of exactly m connected components $G_1,...,G_m$, then there are m linearly independent eigenvectors $\mathbf{x}_1,...,\mathbf{x}_m$ corresponding to the eigenvalue 0, and $\mathbf{x}_i(j) = 1$ if $v_j \in G_i$, and $\mathbf{x}_i(j) = 0$ otherwise.

Any symmetric matrix defines an unweighted undirected graph. (Symmetric matrices can also define weighted undirected graphs, but the details of this are postponed until Chapter 4.) Let A be an nxn symmetric matrix. We define G corresponding to A as follows:

- G has vertex set $V = \{v_1,...,v_n\}$. That is, G has a vertex for every row/column in A.

- G has edge set $E = \{(v_i,v_j)\text{: } A(i,j) = A(j,i) \neq 0\}$.

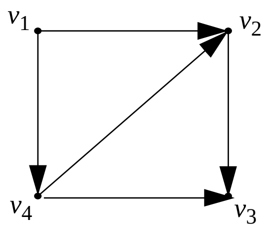
In section 2.3 on direct methods for solving linear systems, we will show that analysis of the graph corresponding to a symmetric matrix can reduce the amount of storage required to factor the matrix.

Just as undirected graphs correspond to symmetric matrices, directed graphs correspond to nonsymmetric matrices. Let $G = (V,E)$ be a directed graph. Then the adjacency matrix A corresponding to G can be constructed as follows:

- $a_{ij} = 1$ if $e = (v_j,v_i) \in E$;

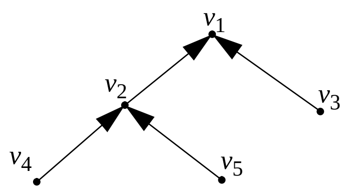
- $a_{ij} = 0$ otherwise.

a)



$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

b)



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.3: ***Directed Graphs and Adjacency Matrices.***
a) *A directed graph similar to Figure 2.1, and corresponding adjacency matrix.*
b) *A directed tree and corresponding adjacency matrix.*

Clearly, a directed graph in which every directed edge is paired with an edge oriented in the opposite direction defines a symmetric matrix.

Figure 2.3 illustrates two directed graphs and corresponding adjacency matrices. Figure 2.3a presents a graph similar to that of Figure 2.1, but directed. Note the difference between the adjacency matrix of Figure 2.1 and that of Figure 2.3a. The directed graph has a much sparser adjacency matrix, and is not symmetric. Figure 2.3b illustrates a directed tree which is ordered from root to leaves and directed from leaves to root. Note that the adjacency matrix is upper triangular. This fact will be used in the implementation of support tree conjugate gradient in Chapter 3.

Given any arbitrary nxn matrix A, symmetric or nonsymmetric, we can also define an unweighted directed graph G =

(V,E):

- G has vertex set $V = \{v_1,...,v_n\}$. That is, G has a vertex for every row/column in A.

- G has edge set $E = \{(v_i,v_j): A(i,j) \neq 0\}$.

2.2.2 Network flow and multicommodity flow

Network flow problems constitute an important area of research in graph theory and combinatorial optimization. Results from the area of multicommodity flow will be used in Chapter 5 to analyze the theoretical properties of the support tree conjugate gradient method.

The most basic network flow problem is that of maximum flow. Following Tarjan (1983), the single-commodity maximum flow problem can be described as follows:

2.1 ***Definition***: (single commodity maximum flow) *Let $G = (V,E)$ be a directed graph with two distinguished vertices, a source s and a sink t, and a positive capacity* $c(v,w)$ *on every edge* $(v,w) \in E$. ($c(v,w) = 0$ *if* $(v,w) \notin E$.) *A flow on G is a function* $f: E \rightarrow \Re$ *having the following three properties*:

1. $f(v,w) = -f(w,v)$.

2. $f(v,w) \leq c(v,w)$

3. *If* $v \in V$, $v \neq s,t$, then $\sum_{w} f(v, w) = 0$.

The value $|f|$ of a flow f is the net flow out of the source s. The maximum flow problem is that of finding a flow of maximum value.

2.2 ***Definition***: *A cut $(S,\bar{S})$ is a partition of the vertex set V into two sets S and $\bar{S}=V-S$ such that $s \in S$, and $t \in \bar{S}$. The capacity of a cut,* $c(S,\bar{S})$, *is defined by* $c(S, \bar{S}) = \sum_{x \in S, y \in \bar{S}} c(x, y)$. *A minimum cut is a cut for which the capacity of the cut is a minimum over all cuts.*

The maximum flow problem has been studied extensively. The main result of interest to us is Ford and Fulkerson's Max-Flow Min-Cut Theorem [Ford and Fulkerson (1956), (1962)].

2.3 ***Theorem*** (Max-Flow Min-Cut): *Let $G = (V,E)$ be a directed graph, and let* $c: E \rightarrow \Re^+$ *be a capacity function. A flow f is a maximum flow iff there exists a cut $(S,\bar{S})$ such that* $|f| = c(S,\bar{S})$.

That is, the value of the maximum flow can be determined without explicitly determining the flow itself simply by finding the value of a minimum cut.

The multicommodity flow problem generalizes the single commodity problem. Following Leighton and Rao (1988):

2.4 ***Definition***: *A multicommodity flow problem consists of*

- *a graph $G = (V,E)$;*

- *a set of commodities,* Γ;

- *a capacity function on the edges* $c: E \rightarrow \Re^+$;

- *a supply function for vertices* s: $V \times \Gamma \rightarrow \Re^{+}$;
- *a demand function for vertices* d: $V \times \Gamma \rightarrow \Re^{+}$;

The objective of a multicommodity flow problem is to construct flows for each commodity that satisfy the demands without violating the constraints of supply and capacity. In contrast to single-commodity flow, there is no Max-Flow Min-Cut Theorem for multicommodity flow.

Shahrokhi and Matula (1990) define the ratio of flow supplied between pairs to the demand as the *throughput*, and studied multicommodity flow for the case in which the throughput must be the same for all pairs. They called this the *maximum concurrent flow problem* (MCFP). They showed that the dual of MCFP is the problem of assigning distances to the edges to maximize the minimum cost of routing the flow. They developed a polynomial time approximation algorithm for the MCFP for the case of arbitrary demands but uniform capacity.

Leighton and Rao (1988) considered a special case of multicommodity flow called the *uniform multicommodity flow problem* (UMFP). In a UMFP, every ordered pair of vertices defines a commodity, and the demands are the same for each commodity. While it is clear that the flow in a UMFP cannot exceed the capacity of a minimum cut, Leighton and Rao first showed that the converse is not true. That is, there are cases of UMFP where the value of a maximum flow is less than the value of a minimum cut.

Let $(S,\bar{S})$ be a cut. For a UMFP with unit demands, the demand across $(S,\bar{S})$ is given by $|S| \cdot |\bar{S}|$. Define the *ratio cost* of a cut to be

$$\gamma(S, \bar{S}) = \frac{\sum_{x \in S, y \in \bar{S}} c(x, y)}{|S| \cdot |\bar{S}|} \tag{2.1}$$

Leighton-Rao define a *minimum cut* to be a cut $(S,\bar{S})$ for which the ratio cost is the minimum over all cuts. That is, for a minimum cut,

$$\gamma(S, \bar{S}) = \min_{S \subseteq V} \frac{\sum_{x \in S, y \in \bar{S}} c(x, y)}{|S| \cdot |\bar{S}|}$$

If the edge capacities are all unity, then the minimum cut corresponds to the concept of a *sparsest cut* from graph theory. Finding a sparsest cut is NP-hard [Garey, Johnson, and Stockmeyer (1976)].

A concept related to that of ratio cost is the *flux*, or *minimum edge expansion*, defined by

$$\alpha = \min_{S \subseteq V} \frac{\sum_{x \in S, y \in \bar{S}} c(x, y)}{\min(|S| \cdot |\bar{S}|)} \tag{2.2}$$

A *flux cut* is a cut for which the value of flux is the minimum over all cuts. The flux is an important concept because it is a measure of the connectedness of a graph. Any component S of a graph G such that $|S| \leq |G|/2$ is connected to the rest of the graph with at least $\alpha|S|$ edges.

The key theorem that Leighton and Rao proved is the relationship between throughput and minimum cuts for the UMFP. In particular, they proved the following theorem for a graph with n vertices:

2.5 ***Theorem***: *There is a feasible flow with* $|f| = \Omega(\gamma/\log n)$.

Theorem 2.5, coupled with the fact that the value of a maximum flow cannot exceed the value of a minimum cut yields the following:

$$\Omega\left(\frac{\gamma}{\log n}\right) \le |f| \le \gamma \tag{2.3}$$

Another interesting result from Leighton and Rao deals with the area of *graph embedding*. In Chapters 4 and 5, we will show that the theoretical properties of the support tree conjugate gradients method are related to the congestion and dilation of certain graph embeddings. In particular, it will be important to minimize the congestion and dilation of the embeddings.

Leighton and Rao proved the following theorem which yields bounds on the congestion and dilation of embeddings without explicitly constructing the embeddings. This result will be used extensively in Chapter 5:

2.6 ***Theorem***: *Consider any n-node bounded degree graph H, and any 1-1 embedding of the nodes of H onto the nodes of an n-node bounded degree graph G with flux* α*. The edges of H can be routed as paths in G with congestion and dilation* $O(\frac{\log n}{\alpha})$.

2.2.3 Graph partitioning

We have discussed above the basic relationship between symmetric matrices and undirected graphs. The linear systems that arise in many applications have interesting interpretations in terms of graphs. In addition, many application problems in computational science and engineering are based on meshes, which are special cases of graphs.

Graph partitioning is a process which is fundamental to the generation of support trees in Chapter 3. Graph partitioning is the process of decomposing a graph into two or more pieces of roughly equal size by removing a collection of either edges or vertices called a *separator* (an edge or vertex separator, respectively). Graph partitioning has a number of applications. For example, finding good partitions is useful in determining orderings for linear systems that minimize the fill that occurs as a result of factorization [George and Liu (1981), Gilbert (1980), Lipton, Rose, and Tarjan (1979)]. Optimization of the physical layout of a VLSI circuit can be performed by using graph partitioning [Donath (1988), Leighton (1983), Leiserson (1983)]. Finally, solving a finite element problem on a distributed processor requires partitioning the finite element mesh and distributing the pieces among the processor elements [Blelloch, *et al* (1992), Farhat and Lesoinne (1993), Nour-Omid, *et al* (1987),Williams (1991)].

In each of the applications above, it is desirable to have the pieces (subgraphs) contain roughly the same number of nodes, with as few edges as possible connecting the pieces (since edges imply communication). This observation motivates the following definition:

2.7 ***Definition*** (edge separator): *An f(n) edge separator that* δ*-splits is a subset of edges, F, of a graph G with n vertices if* $|F| \le f(n)$ *and the vertices of G-F can be partitioned into two sets S and* $\bar{S}$ *such that there are no edges from S to* $\bar{S}$, *and* $|S|, |\bar{S}| \le \delta n$, *where f is a function and* $0 \le \delta \le 1$.

A similar definition can be formulated for vertex separators.

With respect to the previous discussion of network and multicommodity flow, an edge separator defines a cut, and the capacity of a cut is the sum of the weights of the edges in the separator. Similarly, a cut defines a partition of the graph into two sets.

The goal of graph partitioning is to find small separators. Not all graphs have small separators. Consider, for example, K_n, the complete graph on n vertices, where $n = 2k$. If $(S, \bar{S})$ is a cut of K_n such that $|S| = |\bar{S}| = k$, then the separator contains k^2 edges. Compare this with a $\sqrt{n} \times \sqrt{n}$ rectangular mesh, which has the same number of nodes as K_n, but fewer edges; partitioning the mesh into two pieces of equal size can be done with a separator having only $\sqrt{n} < k$ edges.

An interesting and important research problem is characterizing families of graphs by separator size. For example, an early result is that every tree has a single vertex separator that 2/3-splits [Jordan (1869)]. More recently, Lipton and Tarjan (1979) proved that every planar graph has an $O(\sqrt{n})$-separator that 2/3-splits; the constant they obtained for the asymptotic bound was $\sqrt{8}$. Djidjev (1982) improved their result by reducing the constant to $\sqrt{6}$. Other extensions have been made as well [Gazit and Miller (1987), Miller (1986)]. Gilbert, Hutchinson, and Tarjan (1984) proved that all graphs with genus bounded by g have $O(\sqrt{gn})$ separators. Alon, Seymour, and Thomas (1990) proved an $O(\sqrt{h^{3/2}n})$ bound on the separator size for graphs with an excluded minor isomorphic to the complete graph on h vertices. (Roughly speaking, a *minor* is a subgraph that can be obtained by shrinking edges to identify vertices; an *excluded minor* is one which cannot be obtained by such reduction operations.)

Many different approaches to graph partitioning have been taken. In general, graph partitioning algorithms can be classified as being either *combinatorial* or *geometric.*

Combinatorial algorithms only make use of graph connectivity information. Combinatorial algorithms include:

- *iterative improvement algorithms*

  The first iterative improvement algorithm was proposed by Kernighan and Lin (1970). Roughly speaking, this algorithm is implemented by starting with an initial cut, and then iteratively swapping pairs of vertices across the cut if doing so improves the size of the separator. The algorithm continues iterating until no further improvements are possible. Because of the dependence on starting condition, this algorithm is not guaranteed to achieve the best cut.

  Fiduccia and Mattheyses (1982) improved on the Kernighan and Lin algorithm. In their extension, only one vertex is moved at a time, and the method extends to unbalanced cuts, as well as graphs for which the vertex weights may vary.

- s*imulated annealing* [Nour-Omid, *et al* (1987), Williams (1991)]

  Simulated annealing is a general purpose optimization method that is modeled on the process of slow cooling that allows liquids to crystallized. The idea, by analogy to liquid crystallization, is that the energy of a physical system is distributed among its components, the distribution is a probabilistic function of the temperature, and crystallization is a minimum energy state. By cooling the system, there is less freedom in the energy distribution, and the system will tend towards an energy minimum. Local minima are avoided by cooling slowly enough that there is enough energy in the system for components to “bounce” out of local minima. [Press, et al (1988)].

  Simulated annealing often produces good results, but the only guarantee of optimality is the following: if the temperature decreases sufficiently slowly, then the probability of ending in a global optimum tends to certainty [Hajek (1988)].

- *spectral partitioning* [Donath and Hoffman (1972), Pothen, Simon, and Liou (1990), Hendrickson and Leland (1992)]

  Let L be the unit Laplacian matrix of a graph. Then the smallest eigenvalue of L is zero, and the second smallest eigenvalue, λ_2, is related to the connectivity of the graph. Fiedler (1973) was among the first to make this observation, and called λ_2 the algebraic connectivity of the graph. υ_2, the eigenvector corresponding to λ_2, contains information about the relative distances between vertices.

  Spectral partitioning is implemented by determining υ_2, sorting the entries, and partitioning the corresponding vertices about the median value. Higher order eigenvectors can be used to obtain cuts resulting in more than two subgraphs. While this method seems to perform well in practice, there are no guarantees of the quality of the cut. Recently, Guattery and Miller (1994, 1995) have shown examples where the spectral algorithm performs poorly.

- *greedy method* [Dagum (1993), Farhat and Lesoinne (1993)]

  The greedy method is a region growing procedure. The best way to think about the greedy method is as wave propagation. Wavefronts propagate outward from two or more starting nodes, traveling at the same speed in terms of number of edges per step. The places where wavefronts collide define the separator. This method has the advantage that partitioned pieces are connected. However, there are no performance guarantees.

- *multicommodity flow* [Leighton and Rao (1988), Lang and Rao (1994)].

  Multicommodity flow was discussed in §2.2.2. One side effect of the proof relating the value of a maximum uniform multicommodity flow to the value of a minimum cut, was an algorithm guaranteed to find the minimum ratio cut to within a factor of $O(\log n)$ [Leighton and Rao (1988)]. The multicommodity flow method is one of the few graph partitioning methods to provide a performance guarantee.

In contrast to combinatorial methods, geometric methods require the spatial coordinates of the nodes of the graph. For many finite element and finite difference problems, this geometric information is a by-product of mesh construction. Geometric methods include:

- *coordinate bisection* [Simon (1991), Williams (1991)]

  This method is the simplest of the geometric methods. The method is implemented by sorting the nodes according to their coordinates, then bisecting with a hyperplane orthogonal to one the coordinate axes. The coordinate axis that yields the smallest separator is chosen. Again, there are no performance guarantees with this method, and the performance can vary with the orientation of the graph. Pathological cases exist for which the separator obtained with coordinate bisection is among the worst possible.

- *inertia-based slicing* [Farhat and Lesoinne (1993)]

  Inertia-based slicing is a generalization of coordinate bisection. Instead of bisecting orthogonal to one of the coordinate axes, the inertia matrix of the mesh is computed, and the principal axes of the mesh are determined. Bisection is then performed with respect to the principal axes, rather than the coordinate axes. Again, there are no performance guarantees, and pathological cases exist for which inertia-based slicing yields very large separators.

- *sphere separators* [Miller, et al (1992)].

  It can be shown that there exist pathological cases for which the planar cuts used in coordinate bisection and inertia-based slicing cannot yield good separators. In contrast, sphere separators do not suffer these inadequacies [Teng (1991)]. The idea behind sphere separators is to conformally map the mesh points onto a sphere, rotate the points on the surface so that the mass of the mesh is more or less evenly spread out over the sphere surface, then partition using a plane through the center of the sphere (which yields a great circle on the sphere surface). Biasing the cut using the inertia matrix of the mesh improves the quality of the separators [Gremban, Miller, and Teng (1994)]. The sphere separator algorithm is also one of the few separator algorithms for which there are bounds on the expected performance.

2.3 Direct Methods for the Solution of Sparse Linear Systems

This subsection is not intended to be a thorough review of the state-of-the-art in direct methods for the solution of sparse linear systems. Rather, it is intended to be an overview that presents the problems and the solutions that have relevance to STCG. For more detailed information, the reader is referred to Dongarra, *et al* (1991), Duff, Erisman, and Reid (1986)], and Heath, *et al* (1990).

Consider the solution of linear systems of the form

$$A\boldsymbol{x} = \boldsymbol{b} \tag{2.4}$$

where A is nxn large, sparse, symmetric, and positive definite. Direct methods for solution usually involve some variation of computing the Cholesky factorization of A:

$$A = C \cdot C^t \tag{2.5}$$

where C is lower triangular. The solution $\boldsymbol{x}$ is then obtained by the successive solution of the two constituent triangular systems $C\boldsymbol{y} = \boldsymbol{b}$, and $C^t\boldsymbol{x} = \boldsymbol{y}$ by forward and backward substitution, respectively.

2.3.1 Cholesky factorization and the problem of fill

Cholesky factorization is most easily explained by the use of Gaussian elimination to obtain the root-free Cholesky factorization:

$$A = LDL^t \tag{2.6}$$

where L is unit lower triangular, and D is diagonal. Following Khaira, Miller, and Sheffler (1992), the process of factoring A can be described as a recursive series of steps. Let $A_0 = A$, and let I_n denote the nxn identity matrix. Then,

$$A_0 = \begin{bmatrix} d_1 & \boldsymbol{v}_1^t \\ \boldsymbol{v}_1 & B_1 \end{bmatrix} \tag{2.7}$$

$$= \begin{bmatrix} 1 & 0 \\ \boldsymbol{v}_1/d_1 & I_{n-1} \end{bmatrix} \begin{bmatrix} d_1 & \boldsymbol{0}^t \\ \boldsymbol{0} & B_1 - (\boldsymbol{v}_1\boldsymbol{v}_1^t)/d_1 \end{bmatrix} \begin{bmatrix} 1 & \boldsymbol{v}_1^t/d_1 \\ \boldsymbol{0} & I_{n-1} \end{bmatrix} \tag{2.8}$$

$$= L_1 A_1 L_1^t \tag{2.9}$$

$$A_1 = \begin{bmatrix} d_1 & 0 & \boldsymbol{0}^t \\ 0 & d_2 & \boldsymbol{v}_2^t \\ \boldsymbol{0} & \boldsymbol{v}_2 & B_2 \end{bmatrix} \tag{2.10}$$

$$= \begin{bmatrix} 1 & 0 & \boldsymbol{0}^t \\ 0 & 1 & \boldsymbol{0}^t \\ \boldsymbol{0} & \boldsymbol{v}_2/d_2 & I_{n-2} \end{bmatrix} \begin{bmatrix} d_1 & 0 & \boldsymbol{0}^t \\ 0 & d_2 & \boldsymbol{0}^t \\ \boldsymbol{0} & \boldsymbol{0} & B_2 - (\boldsymbol{v}_2\boldsymbol{v}_2^t)/d_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & \boldsymbol{0}^t \\ 0 & 1 & \boldsymbol{v}_2^t/d_2 \\ \boldsymbol{0} & \boldsymbol{0} & I_{n-2} \end{bmatrix} \tag{2.11}$$

This process proceeds recursively until $A_{n\text{-}1} = D$, where D is diagonal. The final result is

$$A = L_1 \dots L_{n-1} D L_{n-1}^t \dots L_1^t \tag{2.12}$$

Since $L = L_1 \dots L_{n-1}$ is unit lower triangular, we have the result sought. For A positive definite, all the elements of D are positive, so the Cholesky factorization can be easily obtained from the *root-free* form:

$$A = LDL^t = (LD^{1/2})(D^{1/2}L^t) = CC^t \tag{2.13}$$

Each step in the sequence of factorizations shown above is a single step of Gaussian elimination. We will occasionally refer to such a step as a *reduction* step. For *A* non-singular, we also have

$$D = L^{-1}AL^{-t}, \tag{2.14}$$

which shows that Gaussian elimination can be performed as a pair of matrix multiplications: a pre-multiplication by a lower triangular matrix, and a post-multiplication by an upper triangular matrix.

For *A* positive definite, Cholesky factorization, as implemented above, is a stable numerical process. When sparse matrices are involved, however, the problem of *fill* becomes critical. Fill occurs when a zero entry of *A* become non-zero in one of the Cholesky factors. It is possible for the Cholesky factors of an *n*x*n* sparse matrix to be dense (with $\Omega(n^2)$ non-zero entries, instead of $O(n)$). Figure 2.4 illustrates an example for which two-thirds of the zero positions in the original matrix became non-zero in the Cholesky factors.

$$\begin{bmatrix} 3 & -1 & -1 & 0 & 0 \\ -1 & 3 & 0 & -1 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1.73 & 0 & 0 & 0 & 0 \\ -0.58 & 1.63 & 0 & 0 & 0 \\ -0.58 & \mathbf{-0.20} & 0.79 & 0 & 0 \\ 0 & -0.61 & \mathbf{-0.16} & 0.77 & 0 \\ 0 & -0.61 & \mathbf{-0.16} & \mathbf{-0.52} & 0.58 \end{bmatrix} \begin{bmatrix} 1.73 & -0.58 & -0.58 & 0 & 0 \\ 0 & 1.63 & \mathbf{-0.20} & -0.61 & -0.61 \\ 0 & 0 & 0.79 & \mathbf{-0.16} & \mathbf{-0.16} \\ 0 & 0 & 0 & 0.77 & \mathbf{-0.52} \\ 0 & 0 & 0 & 0 & 0.58 \end{bmatrix}$$

Figure 2.4: ***Cholesky Factorization with Fill.***
The sparse matrix was factored, with 2/3 of the zeros becoming nonzero.
Filled values are in bold font.

Fill is easiest to understand from a graph-theoretic point of view. As explained in §2.2.1, a correspondence exists between symmetric matrices and undirected graphs. Let *A* be an *n*x*n* symmetric matrix, and let $G = (V,E)$ be the graph defined by

- $V = \{v_1,...,v_n\}$
- $E = \{(v_i,v_j): A(i,j) \neq 0\}$

As explained above, Gaussian elimination proceeds in a matrix by selecting a diagonal element, and then zeroing out all the off-diagonal elements in the same row and column. Graph theoretically, the *i*th step of Gaussian elimination corresponds to selecting a vertex v_i, deleting all the edges between v_i and its neighbors, then adding edges between all the former neighbors of v_i. Figure 2.5 illustrates an example step of Gaussian elimination. The original matrix $A = A_0$ corresponds to the graph below it in the figure. The matrix A_1 that results from the first step of Gaussian elimination is shown with the corresponding graph below it. Note that node v_1 has been disconnected, and an edge has been added between nodes v_2 and v_3, corresponding to the fill in positions (2,3) and (3,2) of A_1. Each step of Gaussian elimination removes all the edges from one node; we call each such step a *node reduction.*

Fill is a property of the order in which nodes are eliminated. Fill can often be reduced by reordering the matrix; graph theoretically, this is equivalent to renumbering the nodes of the graph. Reordering is implemented by pre- and post-multiplication of *A* by a permutation matrix *P* and its transpose, respectively: $B = PAP^t$. The permutation matrix P_{ij} used to interchange row/column *i* and *j* of *A* has the form:

- $P_{ij}(i, j) = P_{ij}(j, i) = 1$
- $P_{ij}(k, l) = 0$, for $k \neq l$;
- $P_{ij}(k, k) = 1$, for $k \neq i$, j;

$$\begin{bmatrix} 3 & -1 & -1 & 0 & 0 \\ -1 & 3 & 0 & -1 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1/3 & 1 & 0 & 0 & 0 \\ -1/3 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 8/3 & \mathbf{-1/3} & -1 & -1 \\ 0 & \mathbf{-1/3} & 2/3 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1/3 & -1/3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

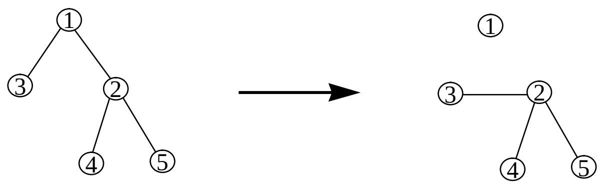


Figure 2.5: ***Graph Theoretic Interpretation of Gaussian Elimination.***
The original matrix $A = A_0$ corresponds to the graph at the left. The first step of Gaussian elimination yields $A_0 = L_1 A_1 L_1^t$. A_1 corresponds to the graph at the right, in which all edges to node v_1 have been eliminated, and all the neighbors of v_1 have been connected. Filled values are in bold font.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & -1 & 3 & -1 \\ -1 & 0 & 0 & -1 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1/3 & 0 \\ -1 & 0 & 0 & -1/3 & 1/3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1/3 & -1/3 \\ 0 & 0 & 0 & 0 & 1/3 \end{bmatrix}$$

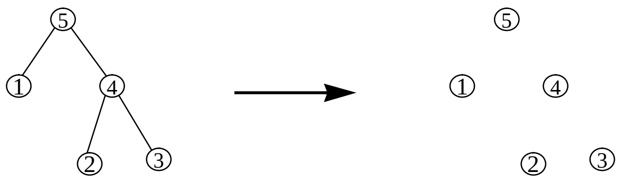


Figure 2.6: ***Gaussian Elimination with Zero Fill.***
The matrix/graph from Figure 2.5 has been reordered so that factorization does not lead to any fill.

Figure 2.6 illustrates an ordering of the matrix/graph from Figure 2.5 that leads to zero fill. Figure 2.6 is an example of a useful fact that will be exploited later in this thesis: *real symmetric matrices that correspond to trees have orderings that permit Cholesky factorization with zero fill.*

For most matrices encountered in real applications, factorization will result in fill. The problem, then, is to minimize fill. The problem of finding an ordering that will yield minimum fill has been shown to be NP-complete [Garey and Johnson (1979)]. Nonetheless, many different approaches to ordering have been suggested for reducing fill. For the purposes of this thesis, the most relevant ordering method is that of *generalized nested dissection* [Lipton, Rose, and Tarjan (1979), Gilbert (1980)], which is explained below.

Generalized nested dissection utilizes a recursive divide-and-conquer approach to produce an ordering. Consider a graph G with n nodes. Generalized nested dissection proceeds by finding a small node (vertex) separator S_0 of G, and ordering these nodes last, assigning them numbers n-$|S_0|$+1 to n. Suppose that removal of S_0 from G results in two connected components G_{00} and G_{01}. Separators S_{00} and S_{01} are then found for G_{00} and G_{01}, and the nodes of these separators are ordered from n-($|S_0|$+$|S_{00}|$+$|S_{01}|$)+1 to n-$|S_0|$. The process then continues recursively until all the nodes of G have been ordered.

Let the *fill-factor* be defined as the ratio of fill to the original number of non-zeros. Then generalized nested dissection yields a fill-factor of $O(\log n)$ for $n^2 \mathrm{x} n^2 \mathrm{x} n^2$ matrices corresponding to planar graphs, finite elements graphs, and several other classes of graphs [Gilbert and Tarjan (1987)]. Moreover, a fill-factor of $O(\log n)$ is asymptotically optimal [Hoffman, Martin, and Rose (1973)].

One of the interesting properties of the support tree preconditioners to be developed in the next chapter is that they can be easily ordered to have zero fill Cholesky factorizations. This property follows from the fact that the graphs of the preconditioners are trees.

2.3.2 Parallel performance

There are three stages in the direct solution of sparse linear systems which must be parallelized:

1. computation of the ordering
2. computation of the Cholesky factors
3. solution of the triangular systems

The ordering computations can be parallelized to some extent. In particular, generalized nested dissection has reasonable parallel potential: first, the processes for finding separators can be parallelized; second, the ordering computations at each level of recursion are independent and may be performed in parallel. Hence, an ordering can be determined using generalized nested dissection with logarithmic parallel speedup [Heath, *et al* (1990)].

Perhaps a bigger issue than that of parallelizing the ordering algorithm is that of selecting the ordering itself. On serial machines, the goal of ordering is to minimize fill, since fill affects both storage requirements and the work required to both factor and solve the resulting systems. The desire to achieve good parallel performance is often at odds with the desire to minimize fill and total work. For example, Heath, *et al* (1990) consider the case of factoring an $n \mathrm{x} n$ tridiagonal matrix. The associated graph is a simple path of length n. By factoring inward, starting at the ends, a no-fill factorization can be achieved (since a simple path is a tree), but at the cost of $O(n)$ parallel steps. On the other hand, if nested dissection were applied to the problem, the result would be a factorization with a fill factor of $O(\log n)$, but only $O(\log n)$ parallel steps in the factorization. As yet, the proper trade-off between parallel work and fill has not been determined.

Given an ordering and the resultant factorization, then the remaining problem is to parallelize the substitution algorithms used to solve the triangular systems.

Forward substitution is implemented straightforwardly in a serial manner by:

$$y_1 = \frac{b_1}{L(1,1)} \text{, and } y_i = \frac{1}{L(i,i)} \cdot \left[b_i - \sum_{j=1}^{i-1} L(i,j) \cdot y_j \right], i = 2, \ldots, n \ ,$$

while backward substitution is given by:

$$x_n = \frac{b_n}{U(n,n)} \text{, and } x_i = \frac{1}{U(i,i)} \cdot \left[y_i - \sum_{j=n}^{i+1} U(i,j) \cdot x_j \right], i = n-1, \ldots, 1 \ .$$

Unfortunately, it is very difficult to efficiently parallelize forward and backward substitution. The serial formulas above point out the inherent data dependencies - each new solution value may depend on one or more of the preceding solution values. High computational rates are even difficult to achieve when the triangular factors are dense, and the situation is made even worse in the sparse case. In the dense case, the products contained within each sum in the substitution formulas can be performed in parallel, and the sum computed in a sequence of parallel steps logarithmic in the number of terms. In the sparse case, however, there are very few products within each sum, so the parallel potential is greatly reduced. Furthermore, the speedup that is obtained in the dense case is largely done by exploiting the regularity of the structure of the dense factors; this regularity is largely lost in the sparse case [Heath, *et al*

(1990)].

Anderson and Saad (1989) studied the problem of triangular solution and proposed a method known as level scheduling for preprocessing sparse triangular matrices in order to maximize parallelism. Level scheduling is best understood as a graph reordering operation in which sets of independent nodes are numbered consecutively. For example, on a square mesh, one common ordering that can be obtained through level scheduling is by diagonal, starting at a corner. On an *n*x*n* mesh, this yields 2*n*-1 sets such that all the nodes in each set are independent and can be solved for in parallel once the previous set has been solved

Figure 2.7 illustrates the interpretation of level scheduling in terms of graph theory. Both graphs in the figure have 25 vertices. In Figure 2.7a, the independent sets correspond to vertices that lie along common diagonals. In Figure 2.7b, the independent sets correspond to levels in the tree. In general, an *n*x*n* mesh (corresponding to an n^2xn^2 matrix) will have $O(n)$ diagonals, while a tree with n^2 nodes will have $O(\log n)$ levels. Therefore, using simple level scheduling, more potential parallelism exists for tree structures than for mesh structures. This observation is important in understanding the parallelism inherent in the support tree conjugate gradient method developed in Chapter 3.

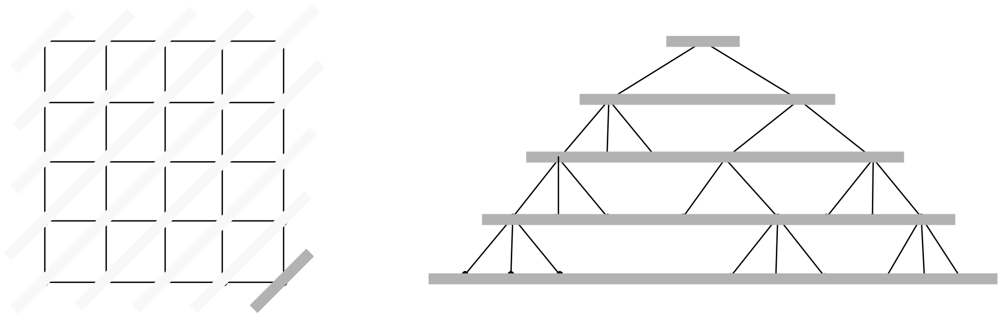


Figure 2.7: ***Graph-theoretic Interpretation of Level Scheduling.***
a) *Level scheduling of a mesh yields sets that lie along diagonals.*
b) *Level scheduling of a tree yields sets that correspond to levels of the tree.*

2.4 Iterative Methods for the Solution of Sparse Linear Systems

As was the case for the previous subsection, this subsection is not intended to be a complete review of the state-of-the-art in iterative methods. Rather, it is intended to bring the reader up to speed with those particular characteristics of iterative methods that will be needed to understand the remainder of this thesis. For more complete discussions of iterative methods, the reader is referred to Axelsson (1994), Hackbusch (1994), and Hageman and Young (1981).

Consider solving

$$A\boldsymbol{x} = \boldsymbol{b} \tag{2.15}$$

iteratively. Finding an iterative solution means finding a sequence $\{\boldsymbol{x}^{(n)}\}$ of approximations to the solution $\boldsymbol{x}$ such that $\boldsymbol{x}^{(n)} \rightarrow \boldsymbol{x}$ as *n* gets large. We shall call each $\boldsymbol{x}^{(n)}$ an *iterate*. Let $\boldsymbol{r}^{(n)} = A\boldsymbol{x}^{(n)} - \boldsymbol{b}$. $\boldsymbol{r}^{(n)}$ is called the *residual*, and is a measure of the accuracy of the *n*th iterate. Typically, an iterative method is halted when the size of the residual drops below a certain tolerance.

2.4.1 The classical iterative methods

Linear, stationary, first-order iterative methods have the form [Hageman and Young (1981)]:

$$\mathbf{x}^{(n+1)} = G\mathbf{x}^{(n)} + \mathbf{k} \tag{2.16}$$

The basic iterative method defined by (2.16) is *linear* because G and $\mathbf{k}$ do not depend on $\mathbf{x}^{(n)}$. The method is *stationary*, since G and $\mathbf{k}$ do not change. Finally, the method is *first-order* since $\mathbf{x}^{(n+1)}$ depends only on $\mathbf{x}^{(n)}$, and not on additional previous iterates. The matrix G is called the *iteration matrix* and is derived from the coefficient matrix A. The basic iterative method converges if and only if the *spectral radius* of G is strictly less than 1; that is, $\rho(G) < 1$, where $\rho(G) = \max\{|\lambda|: \lambda \in \lambda(G)\}$.

The first method we shall consider is called the RF method [Hageman and Young (1981)], and is given by

$$\mathbf{x}^{(n+1)} = (I-A)\mathbf{x}^{(n)} + \mathbf{b} \tag{2.17}$$

or, equivalently, as

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - (A\mathbf{x}^{(n)} - \mathbf{b}) \tag{2.18}$$

or

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \mathbf{r}^{(n)} \tag{2.19}$$

In the RF method, the iteration matrix is (I-A), and, from (2.18), can also be viewed as updating the current iterate with a correction vector that is given in this case by the residual. So, another way of viewing an iterative method is as a *predict-test-correct* loop: the current iterate is a *prediction* of the solution; this is *tested* by computing the residual and comparing the magnitude of the residual against a tolerance; if the residual is too large, the current iterate is *corrected* in some way to produce the next iterate (prediction). Convergence can be improved by finding a better iteration matrix, which is equivalent to finding a better way to correct each iterate.

Some intuition can be gained by observing that the coefficient matrix A is a mapping between two vector spaces, the domain and the range: $A: \mathcal{D} \rightarrow \mathcal{R}$. By examination of (2.19) with respect to A as a mapping, we see an obvious problem with the correction step: each iterate $\mathbf{x}^{(n)}$ is a vector in the domain, $\mathcal{D}$, while the correction vector $\mathbf{r}^{(n)}$ is in the range, $\mathcal{R}$. In general, $\mathcal{R}$ is rotated and scaled with respect to $\mathcal{D}$, so the residual $\mathbf{r}^{(n)}$ does not point directly from $\mathbf{x}^{(n)}$ to the solution $\mathbf{x}$. Figure 2.8 illustrates the mismatch of vector spaces in the predict-test-correct loop for the RF method.

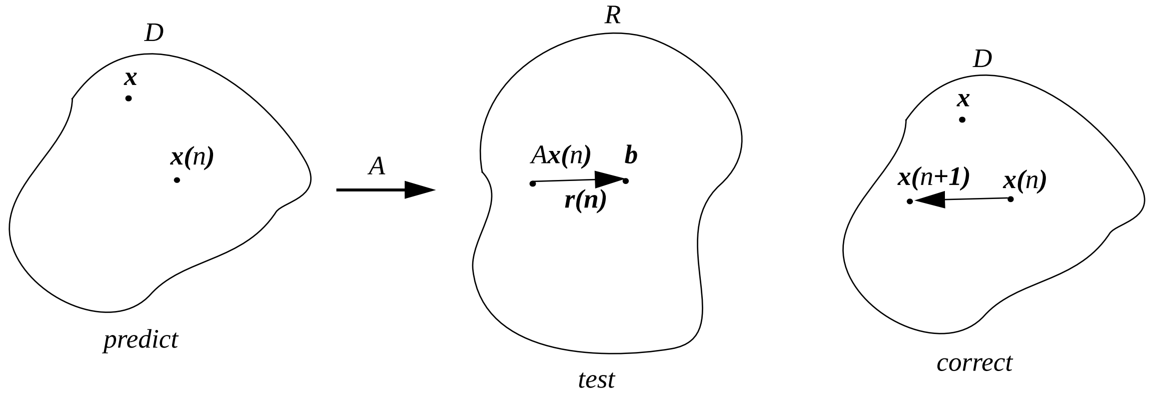


Figure 2.8: ***The Basic Predict-Test-Correct Step of the RF Method.***

Preconditioning is a technique for accelerating the convergence of an iterative method. In the context of the basic iterative method defined by (2.16), preconditioning is a way of determining a better iteration matrix. Consider again the original linear system to be solved, (2.15). Now, let B be some non-singular matrix, of the same size as A, and consider the solution of the linear system

$$B^{-1}A\boldsymbol{x} = B^{-1}\boldsymbol{b} \tag{2.20}$$

B is called a *preconditioner* for *A*. If *B* is chosen properly, then the preconditioned system will be easier to solve than the original system, and the iterative solution method will converge more quickly. The basic iterative method for the preconditioned system (2.20) is given by

$$\boldsymbol{x}^{(n+1)} = (I - B^{-1}A)\boldsymbol{x}^{(n)} + B^{-1}\boldsymbol{b} \tag{2.21}$$

or, in predict-test-correct form, as

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - B^{-1}(A\boldsymbol{x}^{(n)} - \boldsymbol{b}) \tag{2.22}$$

In equations (2.21) and (2.22) above, expressions involving B^{-1} are not meant to imply that B^{-1} is explicitly known. Instead, a term $B^{-1}\boldsymbol{z}$ should be understood as the vector $\boldsymbol{w}$ obtained by solving $B\boldsymbol{w} = \boldsymbol{z}$.

One way of viewing the preconditioner *B* is as an approximation to the mapping *A*, so that B^{-1} approximately transforms the residual from $\mathcal{R}$ into a vector space that is more similar to $\mathcal{D}$. This observation, along with examination of (2.20) shows that the best possible preconditioner for *A* is $B = A$. Of course, applying $B = A$ as a preconditioner means solving systems of the form $B\boldsymbol{w} = \boldsymbol{z}$, for $B = A$, which is the original problem to be solved. The key, then, is to find a preconditioner *B* that approximates *A* in some sense, but leads to systems that are easier to solve.

Many different preconditioners have been proposed. An interesting observation is that the classical iterative methods can all be viewed as instances of (2.21) with different preconditioners. For example, suppose *A* can be written as

$$A = L + D + U \tag{2.23}$$

where *L* is strictly lower diagonal, *D* is diagonal, and *U* is strictly upper diagonal. Then the classical iterative methods are as follows:

- The Jacobi method is defined when all diagonal elements are non-zero by [Hageman and Young (1981)]

$$\boldsymbol{x}^{(n+1)} = (I - D^{-1}A)\boldsymbol{x}^{(n)} + D^{-1}\boldsymbol{b} \tag{2.24}$$

  So, the Jacobi method can be defined by the use of *D* as the preconditioner.

- The Gauss-Seidel method is defined by [Hageman and Young (1981)]

$$\boldsymbol{x}^{(n+1)} = (I - (L + D)^{-1}A)\boldsymbol{x}^{(n)} + (L + D)^{-1}\boldsymbol{b} \tag{2.25}$$

  So, the Gauss-Seidel method can be defined by the use of (*L*+*D*), the lower triangular part of *A*, as the preconditioner.

- The Symmetric Successive OverRelaxation (SSOR) method is defined by [Hageman and Young (1981)]

$$\boldsymbol{x}^{(n+1)} = (I - Q^{-1}A)\boldsymbol{x}^{(n)} + Q^{-1}\boldsymbol{b} \tag{2.26}$$

$$\text{where} \quad Q = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + U\right) \quad \text{, and } 0 < \omega < 2 \text{ is a relaxation parameter.}$$

  So, the SSOR method can be defined by the use of *Q*, above, as the preconditioner.

2.4.2 Conjugate gradient-type methods

In this subsection, we present a brief overview of the method of conjugate gradients (CG) and the extension to the method of preconditioned conjugate gradients (PCG). CG was first developed as a direct solution method by Hestenes and Steifel (1956). CG can be shown, in the absence of rounding errors, to converge to the exact solution in *n* steps, where *A* is *n*x*n*. Rounding errors destroy this process, however, and CG was not considered as a viable direct solution method. Reid (1971) noted that CG converged well for large sparse problems, and sparked interest in the use of CG as an iterative method. An excellent derivation of CG and PCG can be found in Axelsson and Barker (1984). Golub and O'Leary (1989) contains an annotated bibliography that cover the development of CG between 1948 and 1976. In the paragraphs to follow, we essentially summarize the excellent treatment of Axelsson and Barker (1984).

Since the focus of this thesis is on the development of a new version of PCG, we will focus on the convergence rates of the varieties of PCG. In order to do this, we define a model problem, which is simply Poisson's equation defined on the unit square with Dirichlet boundary conditions:

$$\nabla^2 u(x, y) = f(x, y) \tag{2.27}$$

$$(x, y) \in (0, 1) \times (0, 1)$$

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0$$

We discretize the problem with the 5 point Laplacian, yielding an *n*x*n* rectangular mesh. The coefficient matrix corresponding to this problem is n^2xn^2. The choice of mesh and matrix size is made to provide consistency with later chapters.

2.4.2.1 Steepest descent

CG is best understood as an extension of the method of steepest descent, which in turn is derived by viewing the solution of a linear system as finding the minimum of a linear functional. In particular, when the coefficient matrix *A* is symmetric and positive definite, the solution to $A\boldsymbol{x} = \boldsymbol{b}$ can be formulated as a minimization problem for the quadratic functional $f(\boldsymbol{x})$ defined by

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^t A\boldsymbol{x} - \boldsymbol{b}^t\boldsymbol{x} + c \tag{2.28}$$

Equation (2.28) has a unique minimum, $\hat{\boldsymbol{x}}$, which is the solution to $A\boldsymbol{x} = \boldsymbol{b}$.

We wish to develop an iterative method of the form

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \tau_k \boldsymbol{d}^{(k)} \tag{2.29}$$

where $\boldsymbol{d}^{(k)}$ is a direction vector. τ_k is then a measure of how far along the new direction vector we wish to search. To specify the iterative method, we must specify how to choose $\boldsymbol{d}^{(k)}$ and τ_k at each step.

For fixed $f(\boldsymbol{x})$, equation (2.28) defines an ellipsoid in *n*-space, centered around $\hat{\boldsymbol{x}}$. The iterates $\boldsymbol{x}^{(k)}$ are points on ellipsoids centered around $\hat{\boldsymbol{x}}$. For a given $\boldsymbol{x}^{(k)}$, let $\boldsymbol{g}^{(k)} = \boldsymbol{g}(\boldsymbol{x}^{(k)})$ be the gradient at $\boldsymbol{x}^{(k)}$. $\boldsymbol{g}^{(k)}$ is perpendicular to the ellipsoid and points out, away from $\hat{\boldsymbol{x}}$. The gradient defines the direction of the maximum rate of change in the functional at a given point. The method of steepest descent is defined by taking $\boldsymbol{d}^{(k)} = -\boldsymbol{g}^{(k)}$ (hence the name *steepest descent*), and taking τ_k to minimize $f(\boldsymbol{x}^{(k)} + \tau_k\boldsymbol{d}^{(k)})$. It is easily shown that

$$\boldsymbol{g}^{(k)} = \boldsymbol{g}(\boldsymbol{x}^{(k)}) = A\boldsymbol{x}^{(k)} - \boldsymbol{b} \tag{2.30}$$

and

$$\tau_k = \frac{(\boldsymbol{d}^{(k)}, \boldsymbol{g}^{(k)})}{(\boldsymbol{d}^{(k)}, A\boldsymbol{d}^{(k)})} \tag{2.31}$$

where $(\boldsymbol{x},\boldsymbol{y})$ is the standard Euclidean inner product given by $(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{x}^t\boldsymbol{y}$.

Before presenting the steepest descent algorithm, it is useful to point out that, with $\boldsymbol{d}^{(k)} = -\boldsymbol{g}^{(k)}$, (2.29) becomes $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \tau_k \boldsymbol{g}^{(k)}$. Combining this with the definition of $\boldsymbol{g}^{(k)}$ in (2.30), a recursive formula for computing the gradient can be easily derived:

$$\boldsymbol{g}^{(k+1)} = \boldsymbol{g}^{(k)} - \tau_k A \boldsymbol{g}^{(k)} \tag{2.32}$$

A procedure for solving $A\boldsymbol{x} = \boldsymbol{b}$ by the method of steepest descent is given below. The procedure utilizes the recursive formula (2.32) to update the gradient, which saves one matrix-vector multiply.

```
2.8  Procedure steepest_descent ( $A$, $x^{(0)}$, $b$, $\varepsilon$ ) {
         $x = x^{(0)}$;
         $g = Ax - b$;
         $\delta = (g, g)$;
         while ( $\delta > \varepsilon$ ) do {
             $h = Ag$;
             $\tau = \delta / (g, h)$;
             $x = x - \tau g$;
             $g = g - \tau h$;
             $\delta = (g, g)$;
         }
         return ( $x$ );
     };
```

The catch with the method of steepest descent is that the gradient does not point directly towards $\hat{\boldsymbol{x}}$, and the geometry of the ellipsoids defined by the functional may be such that the gradient does not point very close to $\hat{\boldsymbol{x}}$ until the iterates are already very near convergence. To see this, consider a coefficient matrix A with $\lambda(A) = \{\lambda_1 < \lambda_2 < \ldots < \lambda_n\}$and a very large spectral condition number $\kappa(A) = \lambda_n / \lambda_1$. Then $\lambda_n >> \lambda_1$. The eigenvalues of A define the relative lengths of the axes of the ellipsoids defined by the quadratic functional (2.28). Therefore, a large condition number corresponds to ellipsoids that are relatively long and thin with respect to some pair of axes. For such an ellipsoid, the gradient is nearly perpendicular to the long axis, and so the method of steepest descent will march back and forth along the long axis, slowly moving inward towards $\hat{\boldsymbol{x}}$. Convergence in this case is quite slow.

The rate of convergence of steepest descent is given by the following theorem [after Axelsson and Barker (1984)]:

2.9 ***Theorem***: *The rate of convergence of the method of steepest descent is given by*

$$\|\boldsymbol{x}^{(k)} - \hat{\boldsymbol{x}}\|_A \le \left(\frac{\kappa(A) - 1}{\kappa(A) + 1}\right)^k \|\boldsymbol{x}^{(0)} - \hat{\boldsymbol{x}}\|_A$$

where $\kappa(A) = \lambda_n / \lambda_1$ *is the spectral condition number of A, and* $\|\boldsymbol{x}\|_A = (\boldsymbol{x}, A\boldsymbol{x})^{1/2}$.

Less formally, given $\varepsilon > 0$, the number of iterations required to reduce the initial error by a factor of ε is bounded above by $(1/2)\kappa(A)\ln(1/\varepsilon) + 1$ [Axelsson and Barker (1984)]. Thus, for constant ε, the convergence rate of steepest descent is $O(\kappa(A))$.

On the model problem (2.27), $\kappa(A) = O(n^2)$ [Axelsson and Barker (1984)]. Therefore, the rate of convergence of steepest descent is $O(n^2)$.

2.4.2.2 Conjugate gradients

Convergence can be accelerated with the use of conjugate directions. Let $\boldsymbol{d}^{(i)}$ and $\boldsymbol{d}^{(j)}$ be direction vectors. Then $\boldsymbol{d}^{(i)}$ and $\boldsymbol{d}^{(j)}$ are *A-conjugate* (or *A-orthogonal*) if $(\boldsymbol{d}^{(i)}, A\boldsymbol{d}^{(j)}) = 0$. The following theorem can be proved [Golub and Ortega (1993)]:

2.10 ***Theorem***: *If A is a real nxn symmetric positive definite matrix, and* $\{\boldsymbol{d}^{(1)},...,\boldsymbol{d}^{(n)}\}$ *is a set of nonzero A-conjugate direction vectors, then for any* $\boldsymbol{x}^{(0)}$, *with iterates defined by* (2.29), *and* τ_k *defined by* (2.31), *the iterates converge to the solution of A*$\boldsymbol{x}$ *=* $\boldsymbol{b}$ *in no more than n steps*.

The choice of

$$\beta_k = \frac{(\boldsymbol{g}^{(k+1)}, A\boldsymbol{d}^{(k)})}{(\boldsymbol{d}^{(k)}, A\boldsymbol{d}^{(k)})} \tag{2.33}$$

and

$$\boldsymbol{d}^{(k+1)} = -\boldsymbol{g}^{(k+1)} + \beta_k \boldsymbol{d}^{(k)} \tag{2.34}$$

defines the method of *conjugate gradients* (CG) [Golub and Ortega (1993), Axelsson and Barker (1984)].

Reid (1971) studied various implementations of CG with regard to computational labor, storage requirements, and accuracy. Following the algorithm in Axelsson and Barker (1984), which incorporates the implementation favored by Reid, a procedure for computing the solution of $A\boldsymbol{x} = \boldsymbol{b}$ using CG is given below:

```
2.11 Procedure conjugate_gradients ( $A, \boldsymbol{x}^{(0)}, \boldsymbol{b}, \varepsilon$ ) {
         $\boldsymbol{x} = \boldsymbol{x}^{(0)}$;
         $\boldsymbol{g} = A\boldsymbol{x} - \boldsymbol{b}$;
         $\delta = (\boldsymbol{g}, \boldsymbol{g})$;
         $\beta = 0$;
         $\boldsymbol{d} = \boldsymbol{0}$;
         while ( $\delta > \varepsilon$ ) do {
             $\boldsymbol{d} = -\boldsymbol{g} + \beta\boldsymbol{d}$;
             $\boldsymbol{h} = A\boldsymbol{d}$;
             $\tau = \delta / (\boldsymbol{d}, \boldsymbol{h})$;
             $\boldsymbol{x} = \boldsymbol{x} + \tau\boldsymbol{d}$;
             $\boldsymbol{g} = \boldsymbol{g} + \tau\boldsymbol{h}$;
             $\sigma = (\boldsymbol{g}, \boldsymbol{g})$;
             $\beta = \sigma / \delta$;
             $\delta = \sigma$;
         }
         return ( $\boldsymbol{x}$ );
     };
```

For CG, the rate of convergence is given by the following theorem [after Hackbusch (1994)]:

2.12 ***Theorem***: *The rate of convergence of the method of conjugate gradients is given by the following*:

$$\|\boldsymbol{x}^{(k)} - \hat{\boldsymbol{x}}\|_A \le \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \frac{2}{1 + \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^{2k}} \|\boldsymbol{x}^{(0)} - \hat{\boldsymbol{x}}\|_A$$

where $\kappa(A) = \lambda_n / \lambda_1$ *is the spectral condition number of A, and* $\|\boldsymbol{x}\|_A = (\boldsymbol{x}, A\boldsymbol{x})^{1/2}$.

Again, less formally, given $\varepsilon > 0$, the number of iterations required to reduce the initial error by a factor of ε is bounded above by $(1/2)\sqrt{\kappa(A)}\ln(2/\varepsilon) + 1$ [Axelsson and Barker (1984)]. Thus, for constant ε, the convergence rate of CG is $O(\sqrt{\kappa(A)})$.

On the model problem (2.27), $\kappa(A) = O(n^2)$ [Axelsson and Barker (1984)]. Therefore, the rate of convergence of CG is $O(n)$

2.4.2.3 Preconditioned conjugate gradients

As with steepest descent, preconditioning can accelerate the convergence of CG. The resulting method is called *preconditioned conjugate gradients* (PCG).

Because of the requirement that the iteration matrix be symmetric and positive definite, preconditioning for CG must be performed with a similarity transformation. Let $\tilde{A} = CAC^t$. Then $\tilde{A}$ is symmetric and positive definite, and CG can be applied to the system

$$\tilde{A}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}} \tag{2.35}$$

where $\tilde{\boldsymbol{x}} = C^{-t}\boldsymbol{x}$, and $\tilde{\boldsymbol{b}} = C\boldsymbol{b}$.

By algebraic manipulation, it is possible to rearrange the statements in the algorithm implementing CG on the preconditioned system (2.35), avoid all references to the preconditioning factors C and C^t, and instead implement preconditioning as a transformation of the residual with the preconditioner $B = (C^tC)^{-1}$ [see Golub and van Loan (1989), Golub and Ortega (1993)]. A procedure for computing the solution of $A\boldsymbol{x} = \boldsymbol{b}$ using PCG with preconditioner B is given below [after Axelsson and Barker (1984)].

```
2.13 Procedure preconditioned_conjugate_gradients ( $A$, $\boldsymbol{x}^{(0)}$, $\boldsymbol{b}$, $B$, $\varepsilon$ ) {
        $\boldsymbol{x} = \boldsymbol{x}^{(0)}$;
        $\boldsymbol{g} = A\boldsymbol{x} - \boldsymbol{b}$;
        solve $B\boldsymbol{h} = \boldsymbol{g}$;
        $\delta = (\boldsymbol{g}, \boldsymbol{h})$;
        $\beta = 0$;
        $\boldsymbol{d} = \boldsymbol{0}$;
        while ( $\delta > \varepsilon$ ) do {
            $\boldsymbol{d} = -\boldsymbol{h} + \beta\boldsymbol{d}$;
            $\boldsymbol{h} = A\boldsymbol{d}$;
            $\tau = \delta / (\boldsymbol{d}, \boldsymbol{h})$;
            $\boldsymbol{x} = \boldsymbol{x} + \tau\boldsymbol{d}$;
            $\boldsymbol{g} = \boldsymbol{g} + \tau\boldsymbol{h}$;
            solve $B\boldsymbol{h} = \boldsymbol{g}$;
            $\sigma = (\boldsymbol{g}, \boldsymbol{h})$;
            $\beta = \sigma / \delta$;
            $\delta = \sigma$;
        }
        return ( $\boldsymbol{x}$ );
    };
```

Since PCG is an implementation of CG, the convergence results of Theorem 2.12 for CG apply. In particular, given $\varepsilon > 0$, the number of iterations required to reduce the initial error by a factor of ε is bounded above by $\frac{1}{2}\sqrt{\kappa(\tilde{A})}\ln(2/\varepsilon) + 1$ [Axelsson and Barker (1984)]. Thus, for constant ε, the convergence rate of PCG is $O(\sqrt{\kappa(\tilde{A})})$.

A simple set of algebraic manipulations shows that

$$\lambda(\tilde{A}) = \lambda(CAC^t) = \lambda(B^{-1}A) = \lambda(A, B).$$

Therefore, the convergence rate of PCG is $O(\sqrt{\kappa(B^{-1}A)}) = O(\sqrt{\kappa(A, B)})$.

We call the expression $\kappa(A,B) = \kappa(B^{-1}A)$ the *generalized condition number* of the ordered pair of matrices (A, B).

Axelsson and Barker (1984) stated three criteria that define a good preconditioner B for a coefficient matrix A, which can be rephrased as follows:

1. $\kappa(B^{-1}A)$ should be significantly less than $\kappa(A)$;

2. B should be easy to factor, and the factors should not require much storage (in comparison to the storage requirements of A);

3. the system $B\boldsymbol{w} = \boldsymbol{z}$ should be substantially easier to solve than $A\boldsymbol{x} = \boldsymbol{b}$.

The most well-known preconditioners are diagonal scaling, the incomplete Cholesky factorization, the modified (and relaxed) incomplete Cholesky factorization, and the SSOR preconditioner. Each of these is discussed below with attention to the three points listed above.

- diagonal scaling

  The simplest preconditioner for CG is the preconditioner that defines the classical Jacobi method, $B = \text{diag}(A)$. The corresponding variant of PCG is often called *diagonal scaled conjugate gradients*, or DSCG.

  For the model problem (2.27), $\kappa(B^{-1}A) = O(n^2) = \kappa(A)$, so the asymptotic rate of convergence is not improved with diagonal scaling. B in this case does not need to be factored. The storage required for the preconditioner is $O(n^2)$. And, the preconditioned system is very easy to solve, since it simply requires dividing each vector entry by the corresponding diagonal value of B.

  Even though the asymptotic rate of convergence is not improved, diagonal scaling can sometimes make the difference between convergence and non-convergence for an ill-conditioned matrix A. Moreover, diagonal scaling generally achieves some reduction in the number of iterations, and is so cheap to apply that it might as well be done.

- incomplete Cholesky

  The incomplete Cholesky factorization as a preconditioner for CG was first proposed by Meijerink and van der Vorst (1977). The variant of PCG that utilizes incomplete Cholesky preconditioning is often called *incomplete Cholesky conjugate gradients*, or ICCG. The idea behind ICCG is to approximate the coefficient matrix A by performing the Cholesky factorization, but ignoring parts of the factors.

  Recall that for a sparse matrix A, the Cholesky factors C and C^t are often less sparse than A. Let J be the matrix that has a 1 wherever A has a nonzero, and is zero elsewhere. J defines the sparsity pattern of A,. Let K define the sparsity pattern of $(C+C^t)$. Then $J \subseteq K$. The zero-fill incomplete Cholesky factors of A, C_0 and C_0^t are obtained by performing all the steps in factoring A, except those that would change a zero to a nonzero. Thus, if K_0 defines the sparsity pattern of $(C_0+C_0^t)$, then $J = K_0$.

  The use of C_0 and C_0^t as preconditioner factors defines ICCG(0); the preconditioner is $B = C_0C_o^t$. Other variants of ICCG can be defined by specifying the *level* of fill that is allowed in the factors. For example, ICCG(1) allows one level of fill; the fill that results from non-zeros of A is allowed, but that fill is not allowed to propagate and create more fill. ICCG(i) allows i levels of fill.

For the model problem (2.27), $\kappa(B^{-1}A) = O(n^2) = \kappa(A)$, so the asymptotic rate of convergence is not improved with incomplete Cholesky preconditioning [Gustafsson (1978)]. In practice, the constant that accompanies the asymptotic rate is quite small, because ICCG usually requires far fewer iterations to converge than does DSCG or (unpreconditioned) CG. Because fill is not propagated, the factorization is fairly easy to perform, and the preconditioner requires the same amount of storage, $O(n^2)$, as does the coefficient matrix. Solving the preconditioned systems requires performing two sparse triangular solves, which are easy to perform serially, but are difficult to efficiently parallelize (see §2.3).

- modified incomplete Cholesky

  The modified incomplete Cholesky factorization was proposed by Gustafsson (1978). The corresponding variant of PCG is often called *modified incomplete Cholesky conjugate gradients*, or MICCG. The idea behind MICCG is to take the absolute value of the fill that was ignored in computing the IC preconditioner, and add it back to the diagonal. Therefore, MICCG can be viewed as a variant of ICCG with the approximation errors added back into the diagonal [van der Vorst (1989b)]. As with ICCG, various levels of fill can be allowed.

  Axelsson and Lindskog (1986) proposed a relaxed version of MICCG. Instead of adding all the approximation error back into the diagonal, a parameter α, with $0 \le \alpha < 1$ is defined, and α times the error is added in [van der Vorst (1989b)]. Therefore, $\alpha = 0$ defines ICCG (no error added to the diagonal), and $\alpha = 1$ defines MICCG. The relaxed version is often called *relaxed incomplete Cholesky conjugate gradients*, or RICCG.

  For the model problem (2.12), $\kappa(B^{-1}A) = O(n)$ [Gustafsson (1978)], so the asymptotic rate of convergence is significantly improved by MICCG. However, van der Vorst (1989b) reports that, while MICCG has been observed to converge much faster than ICCG on academic model problems, the situation is often reversed for real industrial problems. He explored the effects of varying the parameter α on the convergence rate, and found that the convergence was similar for $\alpha = 0$ and $\alpha = 1$. The convergence improved slowly as α increased from 0, achieving the best results for $\alpha = 0.95$. Between 0.95 and 1.0, the convergence rate sharply decreased again. Axelsson and Lindskog (1986) made similar observations.

  As for ICCG, the computation of the MICCG factors is fairly easy to perform, and the preconditioner requires the same amount of storage, $O(n^2)$, as does the coefficient matrix. Also, solving the preconditioned systems requires performing two sparse triangular solves, which are easy to perform serially, but are difficult to efficiently parallelize (see §2.3).

- symmetric successive over-relaxation (SSOR)

  The iteration matrix from the method of symmetric successive over-relaxation is positive definite for $0 < \omega < 2$, and can be used as a preconditioner in CG. Recall from §2.4.1 that, for $A = L + D + L^t$, the SSOR matrix for relaxation parameter ω is given by

$$B = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + U\right) \tag{2.36}$$

  For the model problem (2.27) with optimal relaxation parameter ω, $\kappa(B^{-1}A) = O(n)$ [Axelsson and Barker (1984)]. The factors of the SSOR preconditioner are generated as a result of the construction; equation (2.36) defines the LDL factorization of the preconditioner. The preconditioner requires the same amount of storage, $O(n^2)$, as does the coefficient matrix. As was the case for ICCG and MICCG, solving the preconditioned systems requires performing two sparse triangular solves, which are easy to perform serially, but are difficult to efficiently parallelize (see §2.3).

2.4.2.4 Parallel performance

In this section, we examine some of the details behind the parallel implementation of CG and PCG. As in §2.4.2, we assume an n^2xn^2 matrix for consistency with later chapters. van der Vorst (1989b) stated three performance requirements for a good preconditioner. The first of his performance requirements was essentially identical to the first of Axelsson and Barker (1984), and we do not bother to state it. The second two requirements are elaborations of requirement 3 of Axelsson and Barker (1984), above:

1. the amount of work per iteration step should be roughly the same as for unpreconditioned CG;

2. the computational speed for each iteration step should have the same order of magnitude as the unpreconditioned CG process.

To analyze the parallel performance of CG and the various versions of PCG, we use the parallel vector models of Blelloch (1990), which comprise a unifying framework for examining the parallel complexity of algorithms. Serial complexity is typically analyzed in terms of an ideal machine architecture, the random access machine, or RAM. Parallel complexity can be analyzed in terms of an ideal parallel architecture, the vector RAM, or V-RAM. A V-RAM is essentially a serial RAM with the addition of a vector processor and vector memory. Each location in vector memory can contain an arbitrarily long vector of scalar values. Each instruction executed by the vector processor can reference one or more vectors from vector memory, and one or more scalars from scalar memory. The reader interested in more details is referred to Blelloch (1990).

The complexity of an algorithm executing on a V-RAM can be characterized by two measures: the *step complexity*, and the *element complexity*. The step complexity is simply defined as the number of (parallel) steps executed sequentially; step complexity can be thought of as the parallel complexity. The element complexity is the sum over the steps of the lengths of the vectors operated on at each step; element complexity can be thought of as the serial complexity.

The basic CG algorithm as given by Procedure 2.11 requires, per iteration: one matrix vector multiply, two vector inner products, and three SAXPY operations (a$\boldsymbol{x}$+$\boldsymbol{y}$, see Dongarra, et al (1991)). van der Vorst (1989b) notes that all these operations can be implemented fairly efficiently on vector machines. The formulation of PCG given as Procedure 2.13 requires all these operations, plus an additional operation of solving the preconditioned system. This solution step is typically implemented as two triangular solves and/or a diagonal scaling.

A sparse matrix vector multiply of an n^2-vector with an n^2xn^2 sparse matrix, assuming that the number of non-zeros in any row of the matrix is bounded by a constant, has step complexity of $O(1)$, and element complexity of $O(n^2)$. This is because the operations to determine each output vector element are independent and may be performed in parallel. Each such operation involves a constant number of operations, and there are $O(n^2)$ of them.

A vector inner product operation has step complexity of $O(\log n)$, and element complexity of $O(n^2)$. The n multiplies required for an inner product can all be performed in parallel. Computing the sum of products requires $2\log n$ steps of pairwise sums: $n^2/2$ sums the first step, $n^2/4$ the next, and so on for a total of $O(n^2)$ operations.

A SAXPY operation takes a vector times a scalar and adds it to another vector. All elements of the scalar-vector multiply can be performed in parallel, as can each element in the vector-vector sum. Therefore, the step complexity is $O(1)$, and the element complexity is $O(n^2)$.

The vector inner product has the largest step complexity and is therefore the bound on the complexity of each iteration of CG. Thus, each iteration of (unpreconditioned) CG has step complexity $O(\log n)$, and element complexity $O(n^2)$.

Now, consider the complexity of applying a preconditioner. The simplest preconditioner is diagonal scaling. All the operations involved can be performed in parallel, yielding a step complexity of $O(1)$, and an element complexity of $O(n^2)$. The other preconditioners, IC, MIC, and SSOR, all require triangular solves.

The naive implementation of a triangular solve is purely serial — each element of the solution vector must be solved for one at a time. This yields a step complexity of $O(n^2)$ and an element complexity of $O(n^2)$, which is clearly a burden for a loop that would otherwise require only $O(\log n)$ steps.

As discussed in §2.3, it is possible to use level scheduling to find a good parallel ordering for triangular solution. For the model problem, an nxn mesh with an n^2xn^2 coefficient matrix, a good ordering is to walk along the diagonals, solving for elements on each diagonal in parallel. The asymptotic performance of this procedure is better, but is still fairly poor. Walking the diagonals has step complexity of $O(n)$, and element complexity of $O(n^2)$. Again, this is a burden on a loop with step complexity of $O(\log n)$.

With respect to van der Vorst's performance requirements for a good preconditioner, only diagonal scaling can be applied without significantly slowing each iteration. There are techniques that can be applied to speed up the preconditioning. For example, with rectangular meshes (such as our model problem) there is a technique developed by Eisenstat (1981) that solves the explicitly preconditioned system (2.35). This technique cannot be applied to meshes with triangles, and is therefore not completely general.

2.5 Domain Decomposition Methods

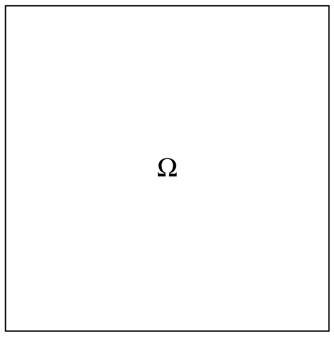
Divide-and-conquer is a powerful algorithmic principle. When applied to the solution of linear systems of equations, divide-and-conquer yields the class of methods called *domain decomposition*. The name arises because the methods involve partitioning the original domain of the problem into a number of smaller subdomains, for which solution can proceed independently in parallel. Domain decomposition methods involve both direct and iterative solvers. The article by Chan and Mathew (1994) is an excellent survey of domain decomposition methods. Additional treatments can be found in Golub and Ortega (1993) and Hackbusch (1994).

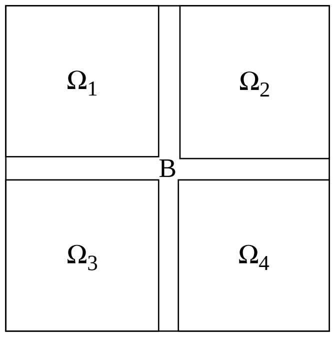
2.5.1 Non-Overlapping subdomains

The basic domain decomposition method involves partitioning a domain into non-overlapping subdomains separated by an interface subdomain. The idea behind domain decomposition is to solve the interface equations that connect the subdomains. This partial solution can then be used to compute the solution on each of the subdomains independently in parallel. Variations on the basic method include different methods used to solve the interface equations, and different methods that result when overlapping subdomains are used.

Figure 2.9 illustrates a region, Ω, its decomposition into non-overlapping regions Ω_1, Ω_2, Ω_3, Ω_4, and the interface region B. The linear systems corresponding to the region and its decomposition are shown in the figure as well. By ordering the nodes in the decomposition so that nodes in the subdomains are grouped together, and nodes in the interface region are ordered last, the linear system corresponding to the decomposition can be written in the block arrowhead form shown in the figure and below as equation (2.37). We assume that the original linear system $A\boldsymbol{x} = \boldsymbol{b}$ is symmetric and positive definite.

$$\begin{bmatrix} A_1 & 0 & 0 & 0 & B_1 \\ 0 & A_2 & 0 & 0 & B_2 \\ 0 & 0 & A_3 & 0 & B_3 \\ 0 & 0 & 0 & A_4 & B_4 \\ B_1^t & B_2^t & B_3^t & B_4^t & A_S \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \\ \boldsymbol{x}_S \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \boldsymbol{b}_3 \\ \boldsymbol{b}_4 \\ \boldsymbol{b}_S \end{bmatrix} \qquad (2.37)$$





$$Ax = \boldsymbol{b}$$

$$\begin{bmatrix} A_1 & 0 & 0 & 0 & B_1 \\ 0 & A_2 & 0 & 0 & B_2 \\ 0 & 0 & A_3 & 0 & B_3 \\ 0 & 0 & 0 & A_4 & B_4 \\ B_1^t & B_2^t & B_3^t & B_4^t & A_S \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \\ \boldsymbol{x}_S \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \boldsymbol{b}_3 \\ \boldsymbol{b}_4 \\ \boldsymbol{b}_S \end{bmatrix}$$

a) **b)**

Figure 2.9: ***Basic Domain Decomposition.***
a) *Original domain and corresponding linear system.*
b) *Subdomains and corresponding block-arrowhead system.*

In equation (2.37), the blocks A_i correspond to the linear system restricted to the subdomains Ω_i, respectively; A_S represents the linear system restricted to the interface nodes. Similarly, the $\boldsymbol{x}_i$ represent the vectors of unknowns corresponding to the subdomains and the interface region, while the $\boldsymbol{b}_i$ represent the reordering of the input vector to be consistent with the partitioning. The off-diagonal blocks B_i represent the interactions between the subdomains and the interface region.

Performing Gaussian elimination on the blocks in equation (2.37) yields:

$$\begin{bmatrix} I_1 & 0 & 0 & 0 & A_1^{-1}B_1 \\ 0 & I_2 & 0 & 0 & A_2^{-1}B_2 \\ 0 & 0 & I_3 & 0 & A_3^{-1}B_3 \\ 0 & 0 & 0 & I_4 & A_4^{-1}B_4 \\ 0 & 0 & 0 & 0 & S \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \\ \boldsymbol{x}_S \end{bmatrix} = \begin{bmatrix} A_1^{-1}\boldsymbol{b}_1 \\ A_2^{-1}\boldsymbol{b}_2 \\ A_3^{-1}\boldsymbol{b}_3 \\ A_4^{-1}\boldsymbol{b}_4 \\ \hat{\boldsymbol{b}}_S \end{bmatrix} \qquad (2.38)$$

where:

$$S = A_S - \sum_{i=1}^{4} B_i^t A_i^{-1} B_i \qquad (2.39)$$

and

$$\hat{\boldsymbol{b}}_S = \boldsymbol{b}_S - \sum_{i=1}^{4} B_i^t A_i^{-1} \boldsymbol{b}_i \tag{2.40}$$

The matrix S in (2.39) is called the *Schur complement*. The Schur complement embodies all the interactions between subdomains. Solving the subsystem of (2.38) that contains S solves the linear system at all the interface nodes. The Schur complement subsystem is given below:

$$S\boldsymbol{x}_S = \hat{\boldsymbol{b}}_S \tag{2.41}$$

The other subsystems of (2.38) for the subdomains Ω_i, i = 1,...,4 are given by:

$$\boldsymbol{x}_i = A_i^{-1}\boldsymbol{b}_i - A_i^{-1}B_i\hat{\boldsymbol{x}}_S \tag{2.42}$$

It is clear from the form of equation (2.42) that the subsystems for the subdomains are independent, involving only quantities belonging to the subdomain and to the interface region. Thus, once (2.41) has been solved for the unknowns in the interface region, the remaining subsystems can be solved in parallel. This coarse grain parallelism is a major benefit of domain decomposition. Golub and Ortega (1993) note that domain decomposition and its variants are the best known parallel algorithms for solving narrow banded systems.

For general linear systems, the Schur complement method of domain decomposition suffers from the fact that the Schur complement is dense and computationally intensive to solve directly. Typically, an iterative method such as preconditioned conjugate gradients is used to solve the Schur complement system. The condition number of the Schur complement system of a second order elliptic operator discretized on an *n*x*n* mesh is $O(n)$, which is a significant improvement over the condition number of $O(n^2)$ for the original system [Chan and Mathew (1994)]. A variety of preconditioners have been developed for use in the iterative solution of the Schur complement system. The reader is referred to Chan and Mathew (1994) for a survey.

2.5.2 Overlapping subdomains

The earliest method of domain decomposition dealing with overlapping subdomains was developed in the pioneering work of H. A. Schwarz over a century ago [Schwarz (1870)]. While variants of Schwarz's method have been developed, the based method for overlapping subdomains is still based on that work. For simplicity of exposition, we follow the treatment of Chan and Mathew (1994), and deal with a domain Ω partitioned into two overlapping subdomains, Ω_1 and Ω_2. Let Γ_1 denote that part of the boundary of Ω_1 that is contained in Ω_2, and let Γ_2 denote the part of the boundary of Ω_2 that is contained in Ω_1. Figure 2.10 illustrates the decomposition.

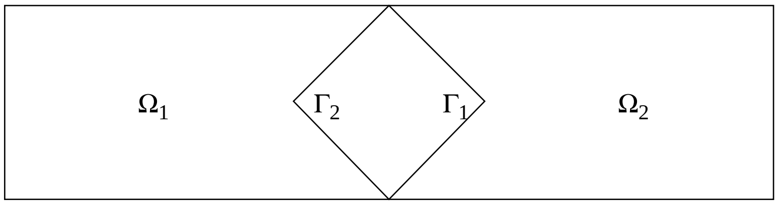


Figure 2.10: ***Decomposition of a domain into overlapping subdomains.***

Now, let $\boldsymbol{x}$ be the n-vector of unknowns in the domain Ω, and let $\boldsymbol{x}_i$ be the n_i-vector of unknowns in the subdomain Ω_i, i = 1, 2. Then $\boldsymbol{x} = \boldsymbol{x}_1 \cup \boldsymbol{x}_2$ and $\boldsymbol{x}_1 \cap \boldsymbol{x}_2 \neq \varnothing$. For i = 1, 2, let R_i be the rectangular matrix that restricts an n-vector to an n_i-vector by selecting the elements that belong to the domain Ω_i. Then, $\boldsymbol{x}_i = R_i\boldsymbol{x}$. Conversely, R_i^t extends an n_i-vector to an n-vector by filling with zeros in entries corresponding to elements of $\Omega - \Omega_i$. The coefficient matrices corresponding to the subdomains are therefore given by $A_i = R_iAR_i^t$.

The *multiplicative Schwarz method* generates a sequence of iterates starting with an initial estimate $\boldsymbol{x}^{(0)}$ by executing the following set of updates:

$$\boldsymbol{x}^{(k+1/2)} = \boldsymbol{x}^{(k)} + R_1^t A_1 R_1 (\boldsymbol{b} - A\boldsymbol{x}^{(k)}) \tag{2.43a}$$

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k+1/2)} + R_2^t A_2 R_2 (\boldsymbol{b} - A\boldsymbol{x}^{(k+1/2)}) \tag{2.43b}$$

The multiplicative Schwarz method is a generalization of the block Gauss-Seidel method. However, unlike the standard Gauss-Seidel method, with sufficient overlap, the convergence rate of the multiplicative Schwarz method is independent of the mesh size [Chan and Mathew (1994)].

The *additive Schwarz method* generates a sequence of iterates starting with an initial estimate $\boldsymbol{x}^{(0)}$ by executing the following set of updates:

$$\boldsymbol{x}^{(k+1/2)} = \boldsymbol{x}^{(k)} + R_1^t A_1 R_1 (\boldsymbol{b} - A\boldsymbol{x}^{(k)}) \tag{2.44a}$$

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k+1/2)} + R_2^t A_2 R_2 (\boldsymbol{b} - A\boldsymbol{x}^{(k)}) \tag{2.44b}$$

The additive Schwarz method is a generalization of the block Jacobi method. Again, with sufficient overlap, the additive Schwarz method has a convergence rate that is independent of the mesh size. Because the correction term in (2.44b) does not reference the update from (2.44a), the two updates in the additive method can be computed in parallel [Chan and Mathew (1994)].

2.6 Multilevel Methods

Multilevel methods refer to a family of methods designed to solve partial differential equations (PDEs), and are generalizations of a method known as multigrid. Multigrid and multilevel are often used interchangeably in the literature. These methods are fairly new; the multigrid paper by Brandt (1977) is often pointed to as the seminal paper in the field. Multilevel methods have received a great deal of attention from the scientific and engineering community because of their impressive theoretical properties: for a PDE defined on n^2 grid points, many multigrid methods have optimal ($O(1)$) or near-optimal ($O(\log^k n)$) convergence rates [Guo (1992)]. Moreover, the total number of operations required can be as low as $O(n^2)$. In this section, we present a very high level introduction to multigrid. More detailed presentations can be found in the paper by Press and Teukolsky (1991), the tutorial by Briggs (1987), or in the textbooks by Axelsson and Barker (1984), Golub and Ortega (1993), or Hackbusch (1994).

2.6.1 Basic multilevel concepts

We first consider a two-grid method. Consider a linear elliptic PDE in two dimensions defined on the unit square that has been discretized on a uniform square mesh n vertices on a side. The result is a linear system of equations

$$A_n \boldsymbol{x}_n = \boldsymbol{b}_n \tag{2.45}$$

where the subscript n serves to denote the resolution of the discretization.

(2.45) can be solved iteratively by one of the methods discussed above, say Gauss-Seidel, for example. Then each iteration produces an approximation to the solution, and we denote the kth approximation by $\boldsymbol{x}_n^{(k)}$. Recall from §2.4 that the kth residual, $\boldsymbol{r}_n^{(k)}$ is defined by $\boldsymbol{r}_n^{(k)}=A_n\boldsymbol{x}_n^{(k)}-\boldsymbol{b}_n$. The residual is often called the *defect*. If we now solve the equation

$$A_n \boldsymbol{z}_n^{(k)} = \boldsymbol{r}_n^{(k)} \tag{2.46}$$

then the solution $\boldsymbol{x}_n$ to (2.45) is given by

$$\boldsymbol{x}_n = \boldsymbol{x}_n^{(k)} + \boldsymbol{z}_n^{(k)} \tag{2.47}$$

The catch with this approach is, of course, that (2.46) is just as hard to solve as the original equation (2.45). However, many iterative methods will usually converge quickly given a good starting approximation. Therefore, a good approximation to (2.46) may be sufficient to accelerate the convergence of the iterative method used to solve (2.45). The question then is how to generate a good approximation.

Consider constructing a mesh over the unit square with only m=n/2 vertices on a side. The corresponding linear system is

$$A_m \boldsymbol{x}_m = \boldsymbol{b}_m \tag{2.48}$$

Since we have discretized the same equation over the same region, changing only the mesh size, we would expect that a solution to (2.48), expanded in some reasonable way, would be a good approximation to (2.45). Or, given the residual $\boldsymbol{r}_n^{(k)}$, we could generate a smaller residual vector $\bar{\boldsymbol{r}}_m^{(k)}$ (that is, *restrict* the residual), and solve the smaller system

$$A_m \bar{\boldsymbol{z}}_m^{(k)} = \bar{\boldsymbol{r}}_m^{(k)} \tag{2.49}$$

Then, we could expand $\bar{\boldsymbol{z}}_m^{(k)}$ to form $\bar{\boldsymbol{z}}_n^{(k)}$ (by interpolation, for example), and use $\bar{\boldsymbol{z}}_n^{(k)}$ to update $\boldsymbol{x}_n^{(k)}$:

$$\boldsymbol{x}_n^{(i+1)} = \boldsymbol{x}_n^{(k)} + \bar{\boldsymbol{z}}_n^{(k)} \tag{2.50}$$

The steps above describe the basic idea behind multigrid, and comprise one iteration of the *coarse-grid correction scheme* [Press and Teukolsky (1991)]:

i) compute the kth approximation;
ii) compute the kth defect;
iii) restrict the defect to a coarser grid;
iv) solve the defect equation on the coarse grid to find the correction;
v) interpolate the correction to the finer grid;
vi) use the correction to compute the (k+1)st approximation.

The coarse-grid correction scheme has intuitive appeal, and has formal justification as well. The way to analyze the coarse-grid correction scheme is to use Fourier analysis and look at the error in terms of frequency components. The residual at each iteration consists of a number of components of various frequencies. It can be shown that an iterative method such as Gauss-Seidel gradually reduces the amplitudes of the components of the residual. However, high frequency components are reduced more quickly than low frequency components; for this reason, many iterative methods are called *smoothers*. The key to the effectiveness of the coarse-grid correction scheme is that the low frequency components of the fine grid residual become high frequency components of the coarse grid residual produced by restriction. That is, the components whose amplitudes are reduced most slowly on the fine grid are reduced quickly on the coarse grid!

The basic multigrid method is a generalization of the coarse-grid correction method described above. Let $\{G_i\}$ be a sequence of successively finer grids, where G_0 is the coarsest, and G_n is the finest. At each iteration, we produce the residual $\boldsymbol{r}_n^{(k)}$, and restrict it to form $\boldsymbol{r}_{n\text{-}1}^{(k)}$. Now, instead of directly solving for the correction $\boldsymbol{z}_{n\text{-}1}^{(k)}$, we recursively apply the coarse-grid correction idea: we perform several iterations of smoothing on the equation $A_{n\text{-}1}\boldsymbol{z}_{n\text{-}1}^{(k)}$=$\boldsymbol{r}_{n\text{-}1}^{(k)}$, then restrict the resulting residual to the next coarser grid, $G_{n\text{-}2}$. This process of smoothing and restricting continues until the coarsest grid, G_0, is reached. Then, $A_0\boldsymbol{z}_0^{(k)}$=$\boldsymbol{r}_0^{(k)}$ is solved for the correction $\boldsymbol{z}_0^{(k)}$. Then, $\boldsymbol{z}_0^{(k)}$ is interpolated to form $\boldsymbol{z}_1^{(k)}$, and $A_1\boldsymbol{z}_1^{(k)}=\boldsymbol{r}_1^{(k)}$ is smoothed several times. The process of computing corrections then proceeds through ever finer grids until $\boldsymbol{z}_n^{(k)}$ is determined and used to compute $\boldsymbol{x}_n^{(k+1)}$. One multigrid cycle is now complete.

2.6.2 Multilevel preconditioners

Recall that preconditioners approximate the original coefficient matrix. Multilevel methods do exactly the same thing, approximate the original coefficient matrix, but with a matrix of lower resolution. Therefore, it is a natural extension to use multilevel methods in preconditioning.

Multilevel methods can be used to form preconditioners by considering all operations to take place in a vector space of dimension equal to the total number of nodes in the multilevel scheme. The expanded coefficient matrices at each level then consist of the original coefficient matrices augmented with diagonal blocks of identity matrices. Restriction and interpolation operators can likewise be defined by matrices.

Consider a basic two-level scheme with fine coefficient matrix A, coarse coefficient matrix C, and restriction and interpolation operators R and P, respectively. A preconditioner B for A can then be defined by $B=PCR$. Different multilevel methods then define different preconditioners which can all be analyzed in the same way as the preconditioners discussed in §2.4.2.3.

The basic multigrid method has a rate of convergence of $O(1)$ for a regular differential operator and a quasi-uniform discretization mesh [Bank and Dupont (1981), Braess and Hackbusch (1983)]. There are many variants of multigrid which achieve optimal or near-optimal convergence rates under more general conditions. For example, the multilevel preconditioner of Bramble, Pasciak, and Xu (1990) was proved by Oswald (1991) to have condition number $O(1)$. The multilevel preconditioner of Axelsson and Vassilevski (1989, 1990) also has optimal condition number $O(1)$.

2.6.3 The difficulty with multilevel methods

The difficulty in applying multilevel methods lies in the requirement for a nested set of meshes or matrices which discretize the problem at different levels of resolution. Constraints of geometry and physics often impose a certain minimum mesh resolution in order that the discrete problem be a good approximation to the continuous problem. Given a mesh at this minimum resolution, it is possible to further refine it in order to obtain finer meshes for application of a multigrid method. However, the mesh representing the minimum resolution may already be fine enough and further refinement may be unnecessary. Standard multigrid methods cannot be applied in this case, since a coarse mesh is unavailable. Moreover, in many applications, only the coefficient matrix is known, and no information about the meshing process is available. Again, standard multigrid methods cannot be applied.

Brandt, McCormick, and Ruge (1982) proposed a method for applying multigrid techniques given only a coefficient matrix. The method, known as Algebraic MultiGrid (AMG), was further refined by Stuben (1983). Essentially, AMG defines a method to construct coarser grids, given the matrix corresponding to the finest grid. AMG is applicable to the same class of matrices (non-singular Laplacian matrices) as the techniques described later in this thesis, and yields convergence rates similar to other multigrid methods [Stuben (1983)]. However, the algorithms involved are very complex and difficult to analyze. Judging from the lack of recent publications, AMG has apparently fallen out of favor within the numerical community.

In conclusion, multigrid methods are very efficient and practical when a nested sequence of matrices or meshes is available. However, when only a coefficient matrix is supplied, multigrid methods are difficult or impossible to apply.

3
Support Trees: Construction and Application

In this chapter, we introduce a new class of preconditioners for the preconditioned conjugate gradient (PCG) algorithm, which we call *support tree preconditioners*. We call the variant of PCG that utilizes these preconditioners *support tree conjugate gradient*, or STCG. Support tree preconditioners can be constructed for linear systems with coefficient matrices that are real, symmetric, and diagonally dominant. In this chapter, we show how to construct support trees for a more restricted set of matrices, those that are real, symmetric, and diagonally dominant with only non-positive off-diagonal elements. An extension to all symmetric and diagonally dominant matrices is presented in Chapter 7.

Support trees have the following advantages as preconditioners:

- They are easy to construct.
- They depend only on the coefficient matrix, and not on the differential equation or the meshing process.
- They are designed for efficient parallel evaluation.
- They are very sparse and therefore have relatively small resource requirements (both storage and work).
- They significantly improve convergence rates.

This chapter first presents the intuition behind the concept of support trees. Then, an algorithm for the construction of support trees is presented, including a discussion of how to implement STCG. Finally, the chapter closes with a discussion of the computational properties of support trees.

3.1 Communication and Mixing

First, some definitions.

3.1 ***Definition:*** *An nxn matrix L is a Laplacian matrix, or Laplacian, if L is real, symmetric, and diagonally dominant with non-positive off-diagonals.*

3.2 ***Definition:*** An *nxn* matrix *L* is a *generalized Laplacian matrix* (*generalized Laplacian*) if *L* is real, symmetric, and diagonally dominant.

Recall from Chapter 2 that a real symmetric matrix A corresponds to an unweighted, undirected graph in which every pair of nonzero off-diagonals $a_{ij}=a_{ji}$ corresponds to a edge between nodes v_i and v_j. For the case in which A is a Laplacian matrix, we can augment the graph by weighting the edges with the absolute values of the corresponding off-diagonal elements. More formally, let A be an *nxn* Laplacian matrix. Then A corresponds to an edge-weighted, undirected graph $G = G(A)$ defined by:

- G has vertex set $V = \{v_1,...v_n\}$, where n is the number of rows/columns in A, and node v_i corresponds to row/column i of A;

- G has edge set $E = \{(v_i,v_j) : A(i,j) \neq 0\}$;

- edge $(v_i,v_j) \in E$ has weight $wt((v_i,v_j)) = |A(i,j)|$.

An example of the correspondence between a Laplacian matrix and a graph is illustrated in Figure 3.1.

$$A = \begin{bmatrix} 9 & -2 & -3 & -4 \\ -2 & 2 & 0 & 0 \\ -3 & 0 & 3 & 0 \\ -4 & 0 & 0 & 4 \end{bmatrix}$$

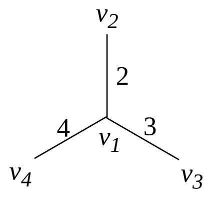


Figure 3.1: ***The correspondence between Laplacian matrices and weighted undirected graphs.***

Consider the multiplication of vector $\boldsymbol{x}$ with matrix A, $\boldsymbol{y} = A\boldsymbol{x}$. One view of the multiplication is as communication of information between adjacent nodes of the graph corresponding to the matrix: each node sends its value to its neighbors (multiplied by the weight of the connecting edge), and each node computes its new value as a weighted sum of its current value and the values from its neighbors. For example, Figure 3.2 illustrates the simple case of communication in a matrix corresponding to a path on 4 points.

Now, consider what happens in an iterative method like Conjugate Gradients (CG). The *i*th iterate in CG is the minimum error solution of the system $A\boldsymbol{x} = \boldsymbol{b}$ projected into $K_i(A; \boldsymbol{r}_0) = \text{span}\{\boldsymbol{r}_0, A\boldsymbol{r}_0, A^2\boldsymbol{r}_0,..., A^{i-1}\boldsymbol{r}_0\}$, where $\boldsymbol{r}_0 = \boldsymbol{b}\text{-}A\boldsymbol{x}_0$ [Dongarra, *et al* (1991), Golub and Ortega (1993)]. $K_i(A; \boldsymbol{r}_0)$ is called a *Krylov* subspace. Each multiplication by the matrix A increases the radius of information propagation by 1 mesh edge. The expression defining the Krylov subspace shows that after i iterations, information from the residual can only have propagated i mesh edges. Moreover, in addition to the limitation of the radius of propagation, the magnitude of the information may decrease as the radius increases.

Figure 3.3 illustrates an example for a 25x25 square mesh, in which the starting vector is an impulse function located at the center of the grid. The propagation of information outward from the impulse is clearly observable. Figure 3.3a shows the starting vector. Figure 3.3b shows the effect of a single matrix multiplication. Figure 3.3c shows the effect

$$\begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 3 & -2 & 0 \\ 0 & -2 & 5 & -3 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

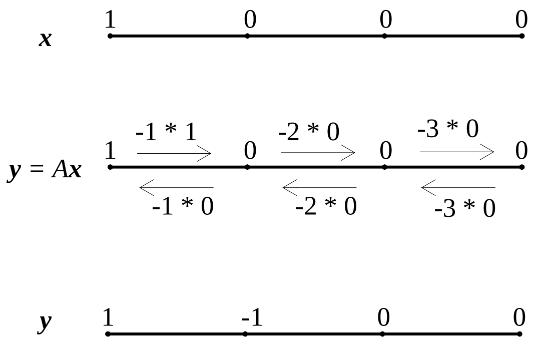


Figure 3.2: ***Matrix multiplication as communication.***
The matrix multiplication is shown at the left. The top right shows the values of ***x*** *superimposed on the graph of the matrix; edge weights have been omitted from the figure. The middle right shows the communication operations that make up the matrix multiplication. The bottom right shows the result of the matrix multiplication.*

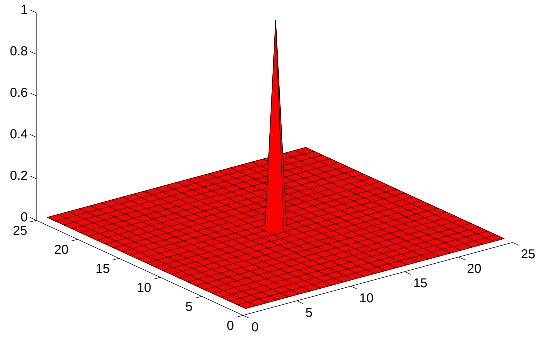
of 10 matrix multiplications; while the information from the impulse function has spread nearly to the edges of the mesh, the magnitude has decreased dramatically. Figure 3.3d show that, after 100 multiplications (corresponding to the 100th iteration of CG), the 100th Krylov subspace encompasses the entire mesh, but the magnitude of the information propagated is small.

Another way to look at the effect of a matrix multiplication is as one step of a mixing process. A matrix multiplication corresponds to having each node "mix" its value with the values from its neighbors. This viewpoint provides an insight into the convergence of iterative methods. The convergence rate of iterative methods in general, and CG in particular, can be related to the rate at which mixing takes place, which in turn is a function of both the rate at which information is propagated across the mesh and the rate at which magnitude is reduced with each multiplication. The solution to a linear system is the fixed point of the iteration, the point at which further information propagation produces no change to the value of the iterate; that is, the information from the initial vector has been completely "mixed".

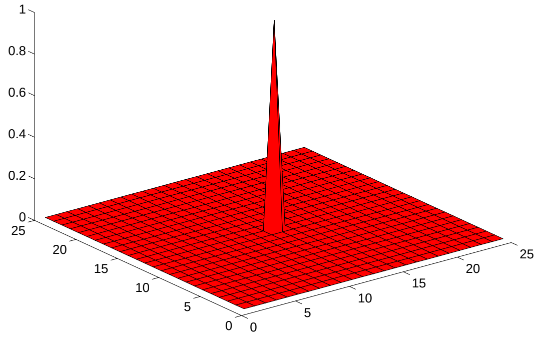
Figure 3.3 and the discussion above motivate the heuristic argument that convergence rate is a function of graph diameter; convergence requires complete mixing, and complete mixing requires information from every node to reach every other node. In fact, on an *n*x*n* mesh (which has n^2 nodes) this is a very good heuristic — the diameter of an *n*x*n* mesh is 2*n*, while the convergence rate of CG is O(*n*) [Guo (1990)].

The idea behind support trees is to accelerate the mixing process by increasing the rate of information propagation. This is implemented in a novel way — by providing an alternate communication network with a smaller diameter so that fewer steps are necessary to get information from one side of the graph to the opposite side. For an *n*x*n* planar graph with a diameter of *n*, the corresponding support tree has a diameter of only log*n*.

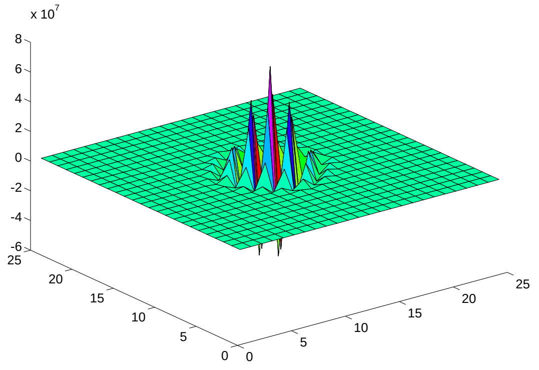
A support tree is constructed by recursively finding edge separators and adding a node for each separator with edges connecting nodes at different levels. Each support tree edge therefore defines a subgraph of the original graph, and the weight of each edge is equal to the total weight of the edges in the boundary of the corresponding subtree. The support tree is therefore able to carry roughly the same volume of communication in/out of a subtree as did the edges in the original graph. But, the support tree is constructed so that the communication distance is shorter. Hence, information mixes more rapidly.



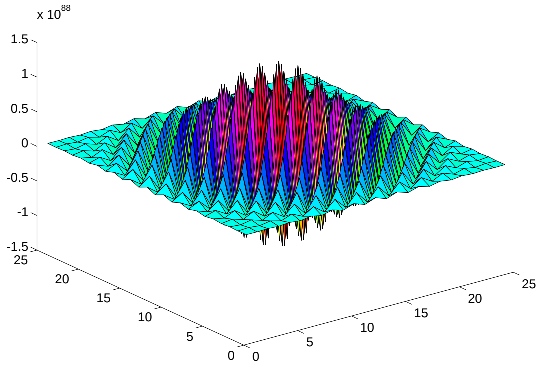
a) ***y*** = ***x***



b) ***y*** = A***x***



c) ***y*** = A^{10}***x***



d) ***y*** = A^{100} ***x***

Figure 3.3: ***Repeated matrix multiplication on a 25x25 grid.***
a) *the starting condition:* ***y*** *=* ***x***
b) *after 1 multiplication:* ***y*** *= A****x***
c) *after 10 multiplications:* ***y*** *=* A^{10}***x***
d) *after 100 multiplications:* ***y*** *=* A^{100}***x***

From the standpoint of preconditioners as approximate inverses, the support tree preconditioners are constructed to approximate the communication network represented by the coefficient matrix. In some sense, then, a support tree preconditioner is intended to approximate the topological properties of the coefficient matrix, rather than the algebraic properties.

3.2 Support Tree Construction

The procedures used in the construction of a support tree are presented as Procedures 3.3 through 3.5 below. In these procedures, the weights of the edges of the support tree are set equal to the total weight of the edges on the boundary of the subgraph induced by the tree edge. Support tree edge weights can be assigned in many other ways. In Chapter 5, we present a weighting based on the ratio of boundary edges to internal nodes; this weighting has some very nice theoretical properties. In all other chapters, however, we utilize boundary weighted support trees.

3.3 ***Procedure:*** *partition_fn*

```
set_of_graphs partition_fn (G) {

    % input: $G$ = an edge-weighted graph on $n$ nodes;
    % output: $\{G_i\}$ = a collection of subgraphs of $G$

    1. find $\mathcal{S}$, an edge separator of $G$;

    2. let $\{G_i\}$ be the connected components of $G - \mathcal{S}$;

    3. return($\{G_i\}$);

  } % end partition_fn
```

3.4 ***Procedure:*** *generate_support_tree*

```
tree generate_support_tree (G, partition_fn) {

    % input: $G$ = an edge-weighted graph on $n$ nodes;
    %   partition_fn = a function which returns a set of two or more subgraphs
    % output: $T$ = support tree for the graph $G$

    1. create a new node $S$;

    2. create a new tree $T$ containing only the node $S$;

    3. $T$ = generate_support_tree_sub ($T, S, G$, partition_fn);

    4. return($T$);

  } % end generate_support_tree
```

3.5 ***Procedure:*** *generate_support_tree_sub*

```
tree generate_support_tree_sub ($T, S_i, G_i$, partition_fn) {

    % input: $T$ = a tree;
    %   $S_i$ = a node of $T$, the root of the subtree to be created
    %   $G_i$ = a subgraph whose support tree is to be rooted at $S_i$
    %   partition_fn = a function which returns a set of two or more subgraphs
    % output: $T = T \cup T_i$, where $T_i$ is the support tree for $G_i$

    1. $\{H_i\}$ = partition_fn($G_i$);

    2. for each $H_i \in \{H_i\}$ {

    3.   create a node $R_i$ in $T$ corresponding to $H_i$;

    4.   compute $\beta_i$, the total weight of the edges on the frontier of $H_i$;

    5.   create an edge of weight $\beta_i$ in $T$ connecting $R_i$ to $S_i$;

    6.   if $|H_i| \geq 1$ then $T$ = generate_support_tree_sub ($T, R_i, H_i$, partition_fn);

    } %end for

    7. return($T$);

  } % end generate_support_tree_sub
```

The process of building a support tree is illustrated in Figure 3.4. The figure represents building a support tree for a finite element mesh that was derived for a cracked plate. The original mesh is presented in Figure 3.4a. Each additional illustration takes the support tree construction process down one more level of recursion. At each level, edges of the separators are drawn as dotted lines, while remaining edges are solid. Figure 3.4f is the final support tree. The crack is not visible in the illustration, since the points are separated by only a small amount, but runs from (0.0,0.5) to (0.5,0.5). A singularity exists at the inner end of the crack, which requires very fine meshing to resolve adequately; the fine meshing can be seen around the center of the mesh.

The reason for the name *support tree* is revealed by looking at the figure. The original mesh is planar, with the support tree sticking out in the third dimension. The mesh appears to be hanging from the support tree; that is, the mesh appears to be *supported* by the tree.

Each node of a support tree defines a subgraph of the graph; the root corresponds to the entire graph, while leaves correspond to individual nodes. Each non-leaf node of a support tree also defines a separator, in particular, the separator used to partition the associated subgraph. Finally, every edge in the support tree corresponds to the collection of edges that make up the frontier of the subgraph associated with the node on the leaf side of the edge.

3.2.1 Partitioning the graph

The process of graph partitioning is at the heart of the support tree construction procedure. Graph partitioning, reviewed in Chapter 2, is a process of deleting edges of a graph G in order to produce two or more subgraphs of roughly the same size that are disconnected from each other. The set of edges removed is called an *edge-separator.* Graph partitioning can also be performed by deleting vertices; the set of vertices which, upon removal, partitions the graph is called a *vertex separator.* In this thesis, we shall only be concerned with edge separators. The goal of graph partitioning is to find small separators.

Any graph partitioning algorithm is applicable to the construction of support trees, provided that the algorithm can be applied to the underlying graph. For example, if only the coefficient matrix is known, then one of the combinatorial algorithms must be used. When geometric information is available, then a geometric algorithm can be used as well.

One of the decisions that must be made in constructing a support tree is deciding upon the branching factor of each node. We have found that, for relatively regular problems, a good solution is to match the branching factor to the dimensionality of the space: a support tree for a mesh in d dimensions has a branching factor of 2^d. Thus, the support tree for a path is a binary tree, the support tree for a square mesh is a quadtree, and so on.

For exotic graphs that may be highly irregular, a somewhat irregular support tree is also desirable. In Chapter 5, we present a method for construction of irregular support trees in which the branching factor at each node depends upon the topology of the underlying subgraph.

3.2.2 Weighting the edges of a support tree

Recall that we have compared the convergence of an iterative method to a mixing process. Each of the constituent matrix/vector products propagates information one step, and the goal in constructing support trees is to accelerate mixing by reducing the distance across the graph. But just reducing the distance is not sufficient. The volume of communication between subsets must be maintained while distance is reduced.

Consider a subgraph G_i of a graph G that resulted from some number of recursive partitioning steps. The collection of edges (u,v) such that $u \in G_i$, and $v \notin G_i$ is called the *frontier* of G_i, which we denote $\mathit{frontier}(G_i)$. Each of the edges $(u,v) \in \mathit{frontier}(G_i)$ has a weight $wt(u,v)$ associated with it. The *frontier weight* of G_i, $\beta(G_i)$ is the sum of the weights of the frontier edges. Imagine that these edges are pipes with capacities given by the weights. Then $\beta(G_i)$ is the total capacity of the edges with which G_i communicates with the rest of the graph. Hence, the edge of the support tree that leads down to G_i must be able to accommodate the same capacity as the frontier of G_i. That is, the same volume of information must flow through the support edge as can flow through $\mathit{frontier}(G_i)$.

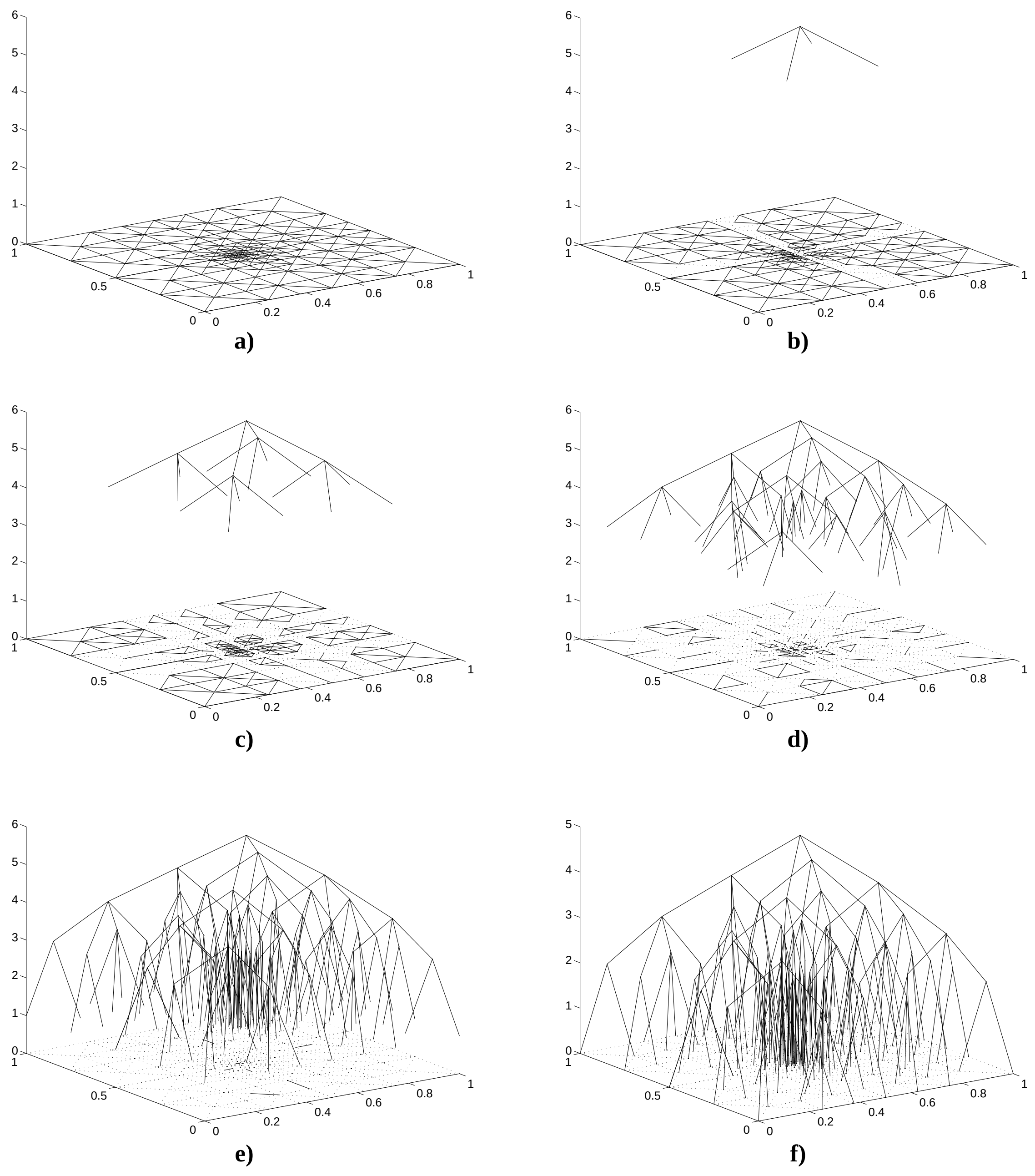


Figure 3.4: ***Support tree construction.***
The steps in the construction of a support tree are illustrated.
a) *the original mesh* ***b)*** *one level of partitioning*
c) *two levels of partitioning* ***d)*** *three levels of partitioning*
e) *four levels of partitioning* ***f)*** *five levels of partitioning: the final tree*

The reasoning above (which is formalized in the proofs of Chapter 4), provides the intuition behind weighting the support tree edges by the total weight of the edges on the frontier of the associated subgraph. In Chapter 5 we discuss a more elaborate way to weight the edges of both regular and irregular support trees that guarantees sufficient support of boundary edges, but also provides properties that are useful for theoretical analysis of general support tree performance. The boundary edge weighting used in the algorithms of this section are sufficient for regular graphs, however.

3.3 Implementation of Support Tree Conjugate Gradient

We use support tree preconditioners in a variant of PCG. We call our variant of PCG *support tree conjugate gradient* (STCG).

Support tree matrices are large and sparse, larger, in fact, than the original matrix. For a simple path on n nodes, the coefficient matrix A is nxn, while the corresponding binary support tree matrix T is $(2n-1)$x$(2n-1)$. How can we make use of a preconditioner that is larger than the original matrix? The answer is to consider T a computational implementation of a smaller matrix. This is explained below.

Let T be a support tree matrix. Then, if T is ordered from leaves to root, T has the form

$$T = \begin{bmatrix} D & R \\ R^t & S \end{bmatrix} \tag{3.1}$$

where D is nxn and diagonal. Rows/columns 1 through n of T correspond to the nodes of the original mesh, which are the leaves of the tree. The other rows/columns correspond to internal nodes of the tree, and constitute the additional variables that make the support tree matrix larger than the original matrix. In graph-theoretic terms, D represents the total connectivity at the leaves. R and R^t represent the connections between leaves and internal nodes of the tree, and S represents the internal nodes and connections in the tree. Figure 3.5 illustrates an example for a quadtree support tree constructed for a 2x4 mesh.

Let A be an nxn coefficient matrix and T be an mxm support tree matrix for A. Assume that A and T are ordered so that T has the block decomposition shown above. Consider performing Gaussian reduction to the internal nodes of T, which is the block S. This process reduces S to a diagonal matrix S_d. In addition, R and R^t are zeroed out, and it can be shown that D fills out to become a dense nxn matrix K. Denote the reduction of T by $\tilde{T}$. The Gaussian reduction can be implemented as matrix multiplication: pre-multiplication by a matrix G, and post-multiplication by G^t, as shown below.

$$GTG^t = \begin{bmatrix} K & 0 \\ 0 & S_d \end{bmatrix} = \tilde{T} \tag{3.2}$$

Let $\tilde{A}$ be the mxm matrix constructed from A by adding m-n rows and columns of zeros:

$$\tilde{A} = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \tag{3.3}$$

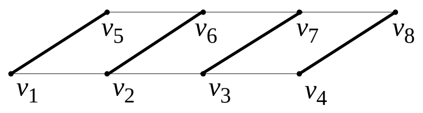
Similarly, for any n-vector $\boldsymbol{x}$, let $\tilde{\boldsymbol{x}}$ be the m-vector obtained by adding m-n zeros to $\boldsymbol{x}$:

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ 0 \end{bmatrix} \tag{3.4}$$

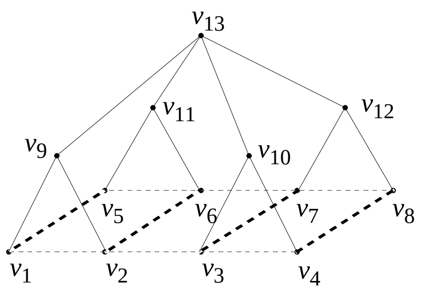
Consider the Gaussian reduction denoted by G and G^t to $\tilde{A}$. The elementary operations of G consist of adding multiples of row/column i, for $i \in \{n+1,...,m\}$ to row/column j, for $j \in \{1,...,n\}$. But, all the rows/columns of $\tilde{A}$ are zero for $i \in \{n+1,...,m\}$. Therefore, $\tilde{A}$ is unchanged by the given Gaussian reduction operations. We therefore have:

$$G\tilde{A}G^t = \tilde{A} \tag{3.5}$$

A similar argument shows that, for any m-vector $\tilde{\boldsymbol{x}}$ constructed by augmenting an n-vector $\boldsymbol{x}$ with zeros:



$$\begin{bmatrix} d_1+2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & d_2+3 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & d_3+3 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & d_4+2 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & d_5+2 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & d_6+3 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & d_7+3 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & d_8+2 \end{bmatrix}$$



$$\begin{array}{cccccccc|ccccc} \multicolumn{8}{c}{D} & \multicolumn{5}{c}{R} \\ d_1+2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & d_2+3 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 \\ 0 & 0 & d_3+3 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_4+2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d_5+2 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d_6+3 & 0 & 0 & 0 & 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d_7+3 & 0 & 0 & 0 & 0 & -3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d_8+2 & 0 & 0 & 0 & -2 & 0 \\ \hline -2 & -3 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & -3 \\ 0 & 0 & -3 & -2 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & -3 \\ 0 & 0 & 0 & 0 & -2 & -3 & 0 & 0 & 0 & 0 & 8 & 0 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & -3 & -2 & 0 & 0 & 0 & 8 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & -3 & -3 & -3 & 12 \\ \multicolumn{8}{c}{R^t} & \multicolumn{5}{c}{S} \end{array}$$

Figure 3.5: ***The structure of support tree matrices.***
At the top is a mesh and the corresponding matrix. At the bottom is a support tree constructed for the mesh, and the corresponding matrix. The block structure of the matrix is illustrated.

$$G\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{x}} \qquad (3.6)$$

We now have enough tools to show how an *m*x*m* support tree matrix *T* may be used as a preconditioner for an *n*x*n* matrix *A*, where $n < m$.

Consider solving $\tilde{A}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}}$, where $\tilde{A}$ is as defined in (3.3), and both $\tilde{\boldsymbol{x}}$ and $\tilde{\boldsymbol{b}}$ are constructed as in (3.4). This system is singular, so more than one solution exists. We need to show how to find a unique solution that can be easily converted into a solution to $A\boldsymbol{x} = \boldsymbol{b}$. Using *T* as a preconditioner is equivalent to solving the system

$$T^{-1}\tilde{A}\boldsymbol{z} = T^{-1}\tilde{\boldsymbol{b}} \qquad (3.7)$$

where $\boldsymbol{z} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}$, $\boldsymbol{x}$ is the solution to $A\boldsymbol{x} = \boldsymbol{b}$, and $\boldsymbol{y}$ is some unknown vector.

Applying Gaussian reduction to T^{-1}, where $GTG^t = \tilde{T}$ yields

$$G^{-t}T^{-1}G^{-1}G\tilde{A}\boldsymbol{z} = G^{-t}T^{-1}G^{-1}G\tilde{\boldsymbol{b}} \tag{3.8}$$

or, using (3.2),

$$\tilde{T}^{-1}G\tilde{A}\boldsymbol{z} = \tilde{T}^{-1}G\tilde{\boldsymbol{b}} \tag{3.9}$$

We now apply the same reduction to $\tilde{A}$:

$$\tilde{T}^{-1}G\tilde{A}G^{t}G^{-t}\boldsymbol{z} = \tilde{T}^{-1}G\tilde{\boldsymbol{b}} \tag{3.10}$$

Now, recalling from above the effect of the particular Gaussian reduction on $\tilde{A}$ and $\tilde{\boldsymbol{b}}$, as given by (3.5) and (3.6), we have:

$$\tilde{T}^{-1}\tilde{A}G^{-t}\boldsymbol{z} = \tilde{T}^{-1}\tilde{\boldsymbol{b}} \tag{3.11}$$

That is, we can use the reduced matrix $\tilde{T}$ as a preconditioner for a system involving a coefficient matrix $\tilde{A}G^{-t}$ instead of using T to precondition A. Moreover, the structure of this alternate system is useful. Equation (3.11) above, in block form, is

$$\begin{bmatrix} K^{-1} & 0 \\ 0 & S_d^{-1} \end{bmatrix}\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} I & 0 \\ H_{21} & H_{22} \end{bmatrix}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} K^{-1} & 0 \\ 0 & S_d^{-1} \end{bmatrix}\begin{bmatrix} \boldsymbol{b} \\ 0 \end{bmatrix} \tag{3.12}$$

Multiplying out the matrix terms yields:

$$\begin{bmatrix} K^{-1}A & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} K^{-1} & 0 \\ 0 & S_d^{-1} \end{bmatrix}\begin{bmatrix} \boldsymbol{b} \\ 0 \end{bmatrix} \tag{3.13}$$

In equation (3.13) above, the variables corresponding to the extra nodes of the support tree have no influence on the solution to $A\boldsymbol{x} = \boldsymbol{b}$. Therefore, we only need to use *K* as a preconditioner for *A*, and solve

$$K^{-1}A = K^{-1}\boldsymbol{b} \tag{3.14}$$

The catch is that *K* is dense, and requires too much work to use, both to compute the triangular factors of *K*, and to solve the resulting systems. However, since the added variables in ***y*** have no effect on the solution ***x***, we can use the sparse support tree matrix *T* as a preconditioner for *A* by simply augmenting the vectors ***x*** and ***b*** with zeros, solving (3.7). and throwing away the extra variables. The advantage in using (3.7) is that *T* is extremely sparse, and is also structured for efficient computation, meaning that solving the larger, sparser system (3.7) is more efficient than solving the smaller, denser system (3.14). The intuition is to think of *T* as a sparse, computationally efficient form of the dense preconditioner *K*.

We can now present the STCG algorithm, which is a variant of the PCG algorithm presented as Procedure 2.12 in Chapter 2. For convenience in exposition, Procedure 2.12 is reproduced here.

```
2.12 Procedure preconditioned_conjugate_gradients ( $A$, $\mathbf{x}^{(0)}$, $\mathbf{b}$, $B$, $\varepsilon$ ) {
        $\mathbf{x} = \mathbf{x}^{(0)}$;
        $\mathbf{g} = A\mathbf{x} - \mathbf{b}$;
        solve $B\mathbf{h} = \mathbf{g}$;
        $\delta = (\mathbf{g}, \mathbf{h})$;
        $\beta = 0$;
        $\mathbf{d} = \mathbf{0}$;
        while ( $\delta > \varepsilon$ ) do {
            $\mathbf{d} = -\mathbf{h} + \beta\mathbf{d}$;
            $\mathbf{h} = A\mathbf{d}$;
            $\tau = \delta / (\mathbf{d}, \mathbf{h})$;
            $\mathbf{x} = \mathbf{x} + \tau\mathbf{d}$;
            $\mathbf{g} = \mathbf{g} + \tau\mathbf{h}$;
            solve $B\mathbf{h} = \mathbf{g}$;
            $\sigma = (\mathbf{g}, \mathbf{h})$;
            $\beta = \sigma / \delta$;
            $\delta = \sigma$;
        }
        return ( $\mathbf{x}$ );
    };
```

We now state the STCG algorithm as Procedure 3.6:

```
3.6 Procedure support_tree_conjugate_gradients ( $A$, $\mathbf{x}^{(0)}$, $\mathbf{b}$, $T$, $\varepsilon$ ) {
        $n$ = dim($A$);
        $m$ = dim($T$);
        $\mathbf{x} = \mathbf{x}^{(0)}$;
        $\mathbf{g} = A\mathbf{x} - \mathbf{b}$;
        $\tilde{\mathbf{g}}$ = augment($\mathbf{g}$,$m$-$n$);
        solve $T\tilde{\mathbf{h}} = \tilde{\mathbf{g}}$;
        $\mathbf{h} = \tilde{\mathbf{h}}(1{:}n)$;
        $\delta = (\mathbf{g}, \mathbf{h})$;
        $\beta = 0$;
        $\mathbf{d} = \mathbf{0}$;
        while ( $\delta > \varepsilon$ ) do {
            $\mathbf{d} = -\mathbf{h} + \beta\mathbf{d}$;
            $\mathbf{h} = A\mathbf{d}$;
            $\tau = \delta / (\mathbf{d}, \mathbf{h})$;
            $\mathbf{x} = \mathbf{x} + \tau\mathbf{d}$;
            $\mathbf{g} = \mathbf{g} + \tau\mathbf{h}$;
            $\tilde{\mathbf{g}}$ = augment($\mathbf{g}$,$m$-$n$);
            solve $T\tilde{\mathbf{h}} = \tilde{\mathbf{g}}$;
            $\mathbf{h} = \tilde{\mathbf{h}}(1{:}n)$;
            $\sigma = (\mathbf{g}, \mathbf{h})$;
            $\beta = \sigma / \delta$;
            $\delta = \sigma$;
        }
        return ( $\mathbf{x}$ );
    };
```

The only differences between PCG (2.12) and STCG (3.6) are at the points where the preconditioner is applied. In

STCG, the residual must be augmented with zeros before the preconditioned system is solved, which is performed by the statement $\tilde{\boldsymbol{g}} = \boldsymbol{augment}(\boldsymbol{g},m\text{-}n)$. Then, following the solution step, performed by **solve** $T\tilde{\boldsymbol{h}} = \tilde{\boldsymbol{g}}$, the solution must be reduced in size by dropping elements corresponding to the added zeros, which is performed by $\boldsymbol{h} = \tilde{\boldsymbol{h}}(1{:}n)$ (where $\tilde{\boldsymbol{h}}(1{:}n)$ is the notation from Matlab [Mathworks, Inc., (1992)] for elements 1 through n of vector $\tilde{\boldsymbol{h}}$). Therefore, most of the STCG algorithm is executed with n-vectors and the nxn matrix A. The only step that involves the added variables is in solving the preconditioned system, which is done efficiently because of the structure of the support tree.

In Chapter 4, we show how an interpretation of Laplacian matrices as resistive networks also leads to the conclusion that a support tree can be used as a preconditioner. The demonstration in Chapter 4 is based on physical principles about current flow.

3.4 Computational Properties of Support Trees

A good preconditioner B for a coefficient matrix A should satisfy three criteria [Axelsson and Barker (1984), vanderVorst (1989)]:

1. Preconditioning with B should reduce the number of iterations required for PCG to converge.

2. B should be easy to construct. That is, the cost of constructing the preconditioner B should be small with respect to the total cost of solving the linear system.

3. The preconditioned system $B\boldsymbol{z} = \boldsymbol{r}$ should be easy to solve. On both serial and parallel machines, this means that the time required to solve $B\boldsymbol{z} = \boldsymbol{r}$ should be small with respect to the time required for an unpreconditioned iteration. On serial machines, a good preconditioner should require relatively little work to solve, and should be structured for efficient execution. On parallel machines, a good preconditioner should be well-structured for parallel execution.

In the subsections below, we address each of these criteria with respect to support tree preconditioners, with special emphasis on parallel implementations.

3.4.1 Reduction in the number of iterations

Recall from the review in Chapter 2 that the generalized condition number of an ordered pair of matrices (A,B) is given by the ratio of the maximum and minimum generalized eigenvalues:

$$\kappa(B^{-1}A) = \kappa(A, B) = \lambda_{max}(A, B)/\lambda_{min}(A, B)$$

The rate of convergence of PCG for coefficient matrix A and preconditioner B is $O(\sqrt{\kappa(B^{-1}A)})$ [Axelsson and Barker (1984)].

To compare convergence rates, we consider as our model problem the two-dimension Dirichlet problem on the unit square discretized by linear finite elements into an nxn square mesh. When the preconditioner is the identity (no preconditioning), the condition number is $O(n^2)$ [Johnson (1987)]. Diagonal scaling (DSCG) and the incomplete Cholesky (ICCG preconditioners are also $O(n^2)$ [Gustafsson (1978)]; although DSCG yields improvements over no preconditioning, and ICCG is known to be an improvement over diagonal scaling, neither actually improves the asymptotic convergence rate. The modified incomplete Cholesky (MICCG) preconditioner is $O(n)$, but requires determination of a relaxation parameter to achieve the optimal convergence rate [Gustafsson (1978), Axelsson and Lindskog (1986)]. The SSOR preconditioner with optimal relaxation parameter is also $O(n)$ [Axelsson and Barker (1984)].

In comparison, we show in Chapter 4 that support tree preconditioners have a generalized condition number of $O(n\log n)$. Thus, support tree preconditioners have better asymptotic properties than diagonal scaling and incomplete

Cholesky, but not as good as modified incomplete Cholesky or SSOR preconditioning. However, it should be noted that support tree preconditioners do not require the computation of any optimizing parameters, as in the case with modified incomplete Cholesky and SSOR. Furthermore, in this section we also show that the regular structure of support trees make their parallel performance much better than any of the others listed here except diagonal scaling. Support tree preconditioners are also more sparse than either modified incomplete Cholesky or SSOR preconditioners.

3.4.2 Ease of construction

The algorithm for support tree construction, presented in §3.2, is very straightforward and relies only on a subroutine to perform graph partitioning. As explained above, graph partitioning is a well-researched problem for which a number of efficient solutions have been proposed.

The algorithm for support tree construction contains two inherent levels of parallelism: parallelism within the partitioning code, and parallelism by subgraph. For example, consider partitioning a subgraph G_i. The partitioning code can be parallelized in various ways. Then, after partitioning G_i into G_{i1},...G_{i4}, the partitioning processes for each of the G_{ij} can be executed on separate processors.

The cost of construction of a support tree preconditioner T must include the cost of factoring T into triangular matrices, $T = VV^t$. The structure of a support tree yields an ordering for factoring the associated matrix T with zero fill. Moreover, the zero-fill ordering is a by-product of the support tree construction procedure. The tree is constructed from root to leaves, while the zero-fill factorization proceeds from leaves to root. Therefore, the order in which tree nodes are created is simply reversed to find the zero-fill ordering.

The zero-fill ordering can be used, with some modification, to perform Cholesky factorization in parallel. The key in parallel factorization is to find sets of independent nodes to factor in parallel. As discussed in the next section, a procedure called *leaf-raking* can be used to determine an ordering for parallel evaluation of independent nodes. When ordered properly, T can be factored in $O(\log n)$ parallel steps.

3.4.3 Ease of solution

Consider the process of solution on a serial processor. A support tree is large, but very sparse. A support tree matrix T for a coefficient matrix A will be stored, in application, in the form of its Cholesky factors V and V^t. The serial work is proportional to the total number of non-zeros in the Cholesky factors. Table 3.1 and Table 3.2 list the resource requirements for diagonal scaling (DSCG), incomplete Cholesky (ICCG), and support tree (STCG) preconditioners of square and cubic meshes, respectively. For the tables, lower order terms have been ignored. In 2D, a quadtree support tree was used for comparison, while in 3D an octtree was used. The tables make clear the sparsity of support trees: despite having more nodes than the original matrix, the support tree has fewer non-zeros than the incomplete Cholesky preconditioner, and even fewer non-zeros than the original matrix itself. Moreover, the advantages of support trees increase with increasing dimensionality: in 2D, the support tree has roughly 80% of the non-zeros that the incomplete Cholesky factors have; in 3D, the fraction drops to less than 50%.

Table 3.1: ***Preconditioner Resource Requirements for an nxn Mesh.***

2D (*nxn*)	DSCG	ICCG	STCG
storage	n^2	$5n^2$	$4n^2$
+	0	$3n^2$	$2n^2$
*	0	$4n^2$	$(8/3)n^2$
/	n^2	n^2	$(4/3)n^2$

Table 3.2: ***Preconditioner Resource Requirements for an nxnxn Mesh.***

3D(*nxnxn*)	DSCG	ICCG	STCG
storage	n^3	$7n^3$	$(24/7)n^3$
+	0	$5n^3$	$2n^3$
*	0	$6n^3$	$(16/7)n^3$
/	n^3	n^3	$(8/7)n^3$

Suppose that T is ordered to yield a zero-fill Cholesky factorization $T = VV^t$. The triangular factors V and V^t correspond to edge-weighted directed trees, one tree with all edges directed from leaves to root, and the other with edges directed from root to leaves. In the directed interpretation, row indices correspond to the heads of arcs and column indices to the tails. For example, $V(i,j) \neq 0$ corresponds to a directed edge (v_j,v_i) for which v_i is the head of the directed edge, and v_j is the tail. Figure 3.6 illustrates a simple example.

a)

$$\begin{bmatrix} 2 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ -1 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & -1 & 2 \end{bmatrix} = \begin{bmatrix} 1.4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -.7 & -1 & 0 & 1.2 & 0 \\ 0 & 0 & -1 & -.8 & .58 \end{bmatrix} * \begin{bmatrix} 1.4 & 0 & 0 & -.7 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1.2 & -.8 \\ 0 & 0 & 0 & 0 & .58 \end{bmatrix}$$

b)

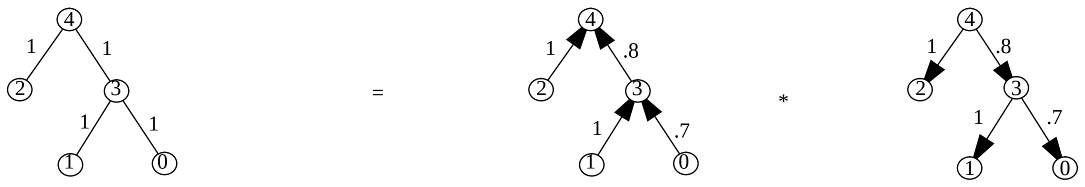


Figure 3.6: ***Graph-theoretic interpretation of Cholesky factorization.***
a) *Cholesky factorization of a Laplacian matrix.*
b) *Equivalent factorization of an undirected tree into two directed trees.*

Recall that, to solve $T\mathbf{z} = \mathbf{r}$, we solve two triangular systems: $V\mathbf{y} = \mathbf{r}$, and $V^t\mathbf{z} = \mathbf{y}$. By referring to Figure 3.6 and recalling the interpretation of triangular matrices as directed graphs, we see that the solution process consists of propagating weighted averages up the tree (solving $V\mathbf{y} = \mathbf{r}$), and then propagating corrections back down the tree (solving $V^t\mathbf{z} = \mathbf{y}$). We can make the process somewhat more efficient for parallel execution by using the root-free Cholesky factorization: $T = CDC^t$, where C is unit lower triangular, and D is diagonal. Then the solution process consists of propagating averages up (solving $C\mathbf{y} = \mathbf{r}$), scaling all values in parallel (computing $\mathbf{w} = D^{-1}\mathbf{y}$), then propagating corrections down the tree (solving $C^t\mathbf{z} = \mathbf{y}$).

The fact that the Cholesky factors correspond to directed trees can be used to optimize the solution procedure for parallel processors. The directed arcs represent dependencies: nodes at the heads of arrows are dependent on values from the nodes at the tails. Hence, leaves are entirely independent, and may be solved in parallel. If the leaves are then removed from the tree, a new, smaller tree results, with leaves that are again independent. We call this evaluation process *leaf-raking,* since all existing leaves are "raked" off the tree at each step. Parallel node evaluation by leaf-raking is a special case of a more general parallel algorithm known as *parallel tree contraction* [Reid-Miller, *et al* (1993)]. A complete binary tree with n leaves ($2n$-1 total nodes) can be evaluated using leaf raking in only $2\lceil \log n \rceil$ parallel steps.

As part of the initialization process for STCG, the order in which nodes should be evaluated is determined, and the linear systems are reordered to maximize data locality at each step. We call this ordering *rake-order.*

Figure 3.7 illustrates the process of leaf raking on a simple tree. An analogous process exists for the downward directed tree: expansions from parents to children can be performed independently in parallel. Leaf raking can result in impressive parallel performance. For example, consider the case of a quadtree support tree for an nxn mesh. The first and last steps in the evaluation of the preconditioned system can be performed by evaluating n^2 nodes in parallel.

Leaf raking is only one source of parallelism that can be exploited in the solution of the preconditioned system. The second source is evaluation by subtree. Each subtree of a support tree is independent of other subtrees. Hence, with a parallel-vector processor architecture such as the Cray C-90, separate subtrees can be assigned to separate processors, and only a single message from each processor to a central processor is necessary to combine results for complete evaluation of a single triangular factor. Each of the subtrees can be evaluated efficiently using the vector capabilities of each processor. Figure 3.8 illustrates the procedure for a simple quadtree support tree assuming p = 16 processors. On a massively parallel machine, clusters of processors can be assigned to subtrees. Interprocessor communication is

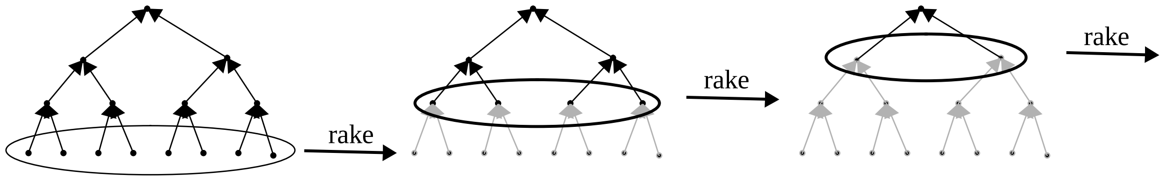


Figure 3.7: ***Leaf Raking and the Solution of a Lower Triangular System.***
***A** linear system corresponding to a tree directed from leaves to root is shown at the left. In the first parallel step, the solutions at the leaves are computed, and the right hand side values at the parents are updated. In succeeding parallel steps, the process is repeated at the leaves obtained when the previous set of leaves is removed. At the last step (not shown), the solution at the root is computed.*

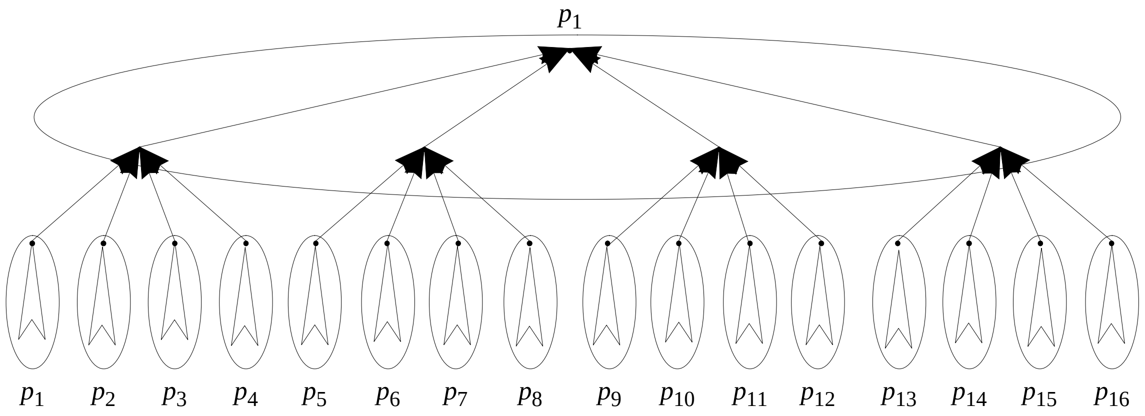


Figure 3.8: ***Parallel evaluation of subtrees.***
With 16 parallel-vector processors, the quadtree is partitioned up at level 3, with each subtree being assigned to a separate processor. By applying leaf raking, each of the subtrees can be efficiently evaluated using the vector capabilities of each processor. To combine the results, one processor is selected as the central processor, and the other processors each send a single number to it.

minimal since each processor must send/receive at most two messages during evaluation of a triangular factor, as long as every processor is assigned either a single node or a subtree.

The theoretical efficiency of STCG in solving the preconditioned system on a per-iteration basis can be compared with that of other methods using the parallel-vector models of Blelloch (1990), which were presented in §2.4.2.4. Consider solving the preconditioned systems for DSCG, ICCG, and STCG applied to an *n*x*n* mesh:

- DSCG is the most efficient of the methods. The preconditioned system can be solved in a single parallel step, yielding a step complexity of $O(1)$, and an element complexity of $O(n^2)$.

- For ICCG, a naive triangular solve method would have step complexity of $O(n^2)$, and asymptotic element complexity of $O(n^2)$ as well. Using the more efficient diagonal ordering (see §2.4.2.4), in which nodes that lie along a common diagonal are solved in parallel, yields step complexity of $O(n)$, and element complexity of $O(n^2)$.

- For STCG using leaf raking, step complexity is $O(\log n)$, and element complexity is $O(n^2)$.

3.5 Summary

In this section, we have presented the intuition that led to the development of support tree preconditioners. We presented an algorithm for the construction of a support tree preconditioner for an arbitrary Laplacian matrix, and showed how to implement the STCG algorithm. We stated that the condition number of STCG is $O(n \log n)$ for an nxn mesh, and showed how the implementation of STCG could be made very efficient for parallel processors.

In the next three chapters, we provide theoretical and practical demonstrations of the properties of STCG. In Chapters 4 and 5, we prove the convergence properties of STCG through analysis of generalized condition numbers, and show the applicability of STCG to Laplacian matrices. In Chapter 6, we present the results of numerical experiments that demonstrate the actual performance of the STCG algorithm. In Chapter 7, we show how to extend STCG to generalized Laplacian matrices.

4
Support Trees: Theory

In this chapter, we present a theoretical analysis of the convergence properties of support tree preconditioners. The analysis relies on interesting isomorphisms between Laplacian matrices, undirected graphs with self-loops, and grounded resistive networks. The theory interprets matrix multiplication as current flow in resistive networks and shows how condition numbers are based on a concept of support of one network for another.

4.1 Matrices and Graphs

Recall the definition of Laplacian matrices from Chapter 2:

> *An nxn matrix L is a* Laplacian matrix*, or* Laplacian*, if L is real, symmetric, and diagonally dominant with non-positive off-diagonals.*

A one-to-one correspondence exists between a subset of undirected graphs and Laplacian matrices.

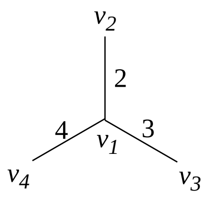
Given an undirected graph with positive edge weights, a corresponding Laplacian matrix can be constructed. Let $G = (V,E)$ be an undirected graph with vertex set $V = \{v_1 ... v_n\}$, edge set $E = \{(v_i,v_j): v_i$ is adjacent to $v_j\}$, and edge weights $w(v_i,v_j)$. The Laplacian matrix of G, $L = L(G)$, is an nxn real symmetric matrix such that:

- $L(i,j) = L(j,i) = -w(v_i,v_j)$, where $w(v_i,v_j)$ is the weight of the edge between v_i and v_j;
- $L(i,i) = \sum_{v_i \text{adj} v_j} w(v_i, v_j)$.

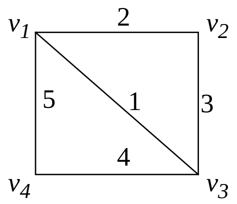
Figure 4.1 illustrates several edge-weighted graphs and their Laplacian matrices.

The Laplacian matrices illustrated in Figure 4.1 are all singular. In particular, they all have the zero row/column sum property: *the sum of all the elements in any row/column is zero*. The zero row/column sum property implies singularity because it means that the constant vector is an eigenvector corresponding to eigenvalue zero.

Given an nxn Laplacian matrix L with the zero row/column sum property, it is easy to construct the corresponding graph $G = G(L)$:



$$L = \begin{bmatrix} 9 & -2 & -3 & -4 \\ -2 & 2 & 0 & 0 \\ -3 & 0 & 3 & 0 \\ -4 & 0 & 0 & 4 \end{bmatrix}$$



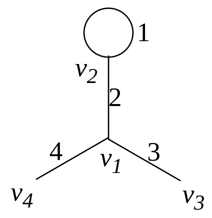
$$L = \begin{bmatrix} 8 & -2 & -1 & -5 \\ -2 & 5 & -3 & 0 \\ -1 & -3 & 8 & -4 \\ -5 & 0 & -4 & 9 \end{bmatrix}$$

Figure 4.1: ***Graphs and Laplacian Matrices.***

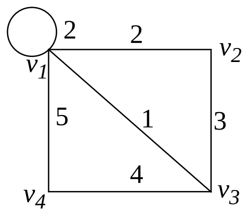
- G has vertex set $V = \{v_1,\ldots,v_n\}$, where n is the number of rows/columns in L, and v_i corresponds to row/column i of L;
- G has edge set $E = \{(v_i,v_j): L(i,j) \neq 0\}$;
- edge $(v_i,v_j) \in E$ has weight $wt(v_i,v_j) = |L(i,j)|$.

So there is an isomorphism between Laplacian matrices with the zero row/column sum property (that is, singular Laplacian matrices) and edge-weighted, undirected graphs. To deal with non-singular Laplacians, it is necessary to expand the class of graphs.

A Laplacian L can be made positive definite (and hence non-singular) by adding positive weight to one or more of the diagonal elements. That is, for L a Laplacian having the zero row/column sum property, L can be made non-singular by adding positive weight d_i to $L(i,i)$ for some i. In terms of the associated graph $G = G(L)$, we represent the added diagonal weight d_i as a self-loop on node v_i of weight d_i. Figure 4.2 illustrates diagonal weighting on Laplacians and their associated graphs.



$$L = \begin{bmatrix} 9 & -2 & -3 & -4 \\ -2 & 3 & 0 & 0 \\ -3 & 0 & 3 & 0 \\ -4 & 0 & 0 & 4 \end{bmatrix}$$



$$L = \begin{bmatrix} 10 & -2 & -1 & -5 \\ -2 & 5 & -3 & 0 \\ -1 & -3 & 3 & -4 \\ -5 & 0 & -4 & 9 \end{bmatrix}$$

Figure 4.2: ***Non-singular Laplacians and Self-loops.***

We have now demonstrated an isomorphism between all Laplacian matrices and undirected graphs with self-loops, in which all edges are weighted with positive weights.

4.2 Matrices and Resistive Networks

There is also an isomorphism between Laplacian matrices and resistive networks. Consider a simple circuit consisting of two nodes v_j and v_k separated by a resistor with resistance r_{jk}, as in Figure 4.3. The current between v_j and v_k, denoted i_{jk}, is a function of the voltage difference between the nodes and the resistance r_{jk}:

$$i_{jk} = \frac{u_j - u_k}{r_{jk}} \tag{4.1}$$

where u_j and u_k are the voltages at nodes v_j and v_k, respectively. Or, since conductance is the reciprocal of resistance, we have

$$i_{jk} = c_{jk}(u_j - u_k) \tag{4.2}$$

where c_{jk}, the conductance between v_j and v_k is given by

$$c_{jk} = 1/r_{jk} \tag{4.3}$$

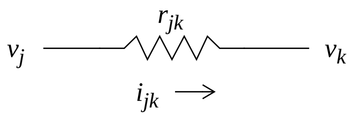


Figure 4.3: ***Simple resistive circuit.***

By symmetry in the circuit, we also have

$$i_{kj} = c_{kj}(u_k - u_j) \tag{4.4}$$

Let i_j and i_k be the net current flow out of node v_j and v_k, respectively. Since $c_{jk} = c_{kj}$, we can rewrite the equations above as a linear system:

$$\begin{bmatrix} i_j \\ i_k \end{bmatrix} = \begin{bmatrix} c_{jk} & -c_{jk} \\ -c_{jk} & c_{jk} \end{bmatrix} \begin{bmatrix} u_j \\ u_k \end{bmatrix} \tag{4.5}$$

or, in matrix notation,

$$\boldsymbol{i} = C\boldsymbol{u} \tag{4.6}$$

We call C the *conductance* matrix for the resistive circuit. The conductance matrix maps applied voltages at the nodes to net currents at the nodes. Because conductance is the property of interest to us, we will label resistors in figures by their conductances.

We can generalize these equations to resistive circuits with an arbitrary number of nodes. Consider a resistive network with n nodes. Suppose that nodes v_j and v_k are connected, as above, with a resistor having conductance c_{jk}. Let $\boldsymbol{e}_{jk}$, for $j < k$ be the n-vector with 1 in the jth component and -1 in the kth. Then let

$$E_{jk} = e_{jk}e_{jk}^t \tag{4.7}$$

E_{jk} is the nxn matrix having +1 in positions $E_{jk}(j,j)$ and $E_{jk}(k,k)$, -1 in positions $E_{jk}(j,k)$ and $E_{jk}(k,j)$, and zero elsewhere. Equation (4.5) can then be written as

$$\begin{bmatrix} i_j \\ i_k \end{bmatrix} = c_{jk}E_{jk} \begin{bmatrix} u_j \\ u_k \end{bmatrix} \tag{4.8}$$

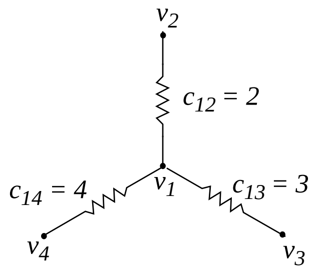
Since the net current flow at a node is given by the sum of the current flows between the node and its neighbors, we have for any node v_j:

$$i_j = \sum_{k \operatorname{adj} j} c_{jk}(u_j - u_k) \tag{4.9}$$

Or, for all nodes in the circuit:

$$\boldsymbol{i} = \left(\sum_{k \operatorname{adj} j,\, k > j} c_{jk}E_{jk} \right) \boldsymbol{u} = C\boldsymbol{u} \tag{4.10}$$

In the equation above, the conductance matrix *C* is now the sum of all the conductance matrices for the individual resistive connections. The sum is taken for adjacencies with nodes of higher index to avoid duplication. The conductance matrix *C* is exactly the Laplacian matrix of the weighted graph *G* that has the same topology as the resistive circuit, with edge weights given by the conductances of the individual connections. Figure 4.4 illustrates a resistive network and its associated conductance matrix. Compare this with the weighted graph and Laplacian in Figure 4.1.



$$C = \begin{bmatrix} 2 & -2 & 0 & 0 \\ -2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 0 & -3 & 0 \\ 0 & 0 & 0 & 0 \\ -3 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 4 & 0 & 0 & -4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -4 & 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 9 & -2 & -3 & -4 \\ -2 & 2 & 0 & 0 \\ -3 & 0 & 3 & 0 \\ -4 & 0 & 0 & 4 \end{bmatrix}$$

Figure 4.4: R*esistive network and conductance matrix.*

The conductance matrices for resistive circuits have the zero row/column sum property. This property makes sense from a physical standpoint: when all the nodes in a circuit have the same voltage, there is no net flow of current at any node. However, we need to extend the resistive circuit isomorphism to include non-singular Laplacians.

Since a conductance matrix is a Laplacian matrix, it can be made non-singular by adding positive weight to one or more of the diagonal elements. In the case of resistive circuits, this can be interpreted as adding a resistive connection to a ground node that is fixed at voltage zero. This is formalized below.

4.1 ***Definition*** (augmented matrix): *Let L be a Laplacian matrix. Then L = C + D, where C is the conductance matrix of a resistive circuit with n nodes, and D is an nxn diagonal matrix with non-negative entries. Let F be the conductance matrix corresponding to the circuit of C augmented with a ground node* v_{n+1} *such that F(i,n+1) = F(n+1,i) = -D(i,i), and* $F(n+1, n+1) = \sum_i D(i,i)$. *That is, each node in the circuit is connected to the ground node with a conductance equal to the additional diagonal weight at that node. We call F the* augmented matrix, *or* augmentation, *of L. Let* $\boldsymbol{d} = \mathrm{diag}(D) = [D(1,1), \ldots, D(n,n)]^t$, *and* $h = \sum_k d_k = tr(D)$. *Then F can be written in block form as*

$$F = \begin{bmatrix} L & -\boldsymbol{d} \\ -\boldsymbol{d}^t & h \end{bmatrix} \tag{4.11}$$

4.2 ***Lemma***: *Let L be a Laplacian matrix. Let F be the augmentation of L. For any vector of applied voltages* $\boldsymbol{u} = \begin{bmatrix} u_1 & \ldots & u_n \end{bmatrix}^t$, let $\boldsymbol{w} = \begin{bmatrix} u_1 & \ldots & u_n & 0 \end{bmatrix}^t$, $\boldsymbol{i} = L\boldsymbol{u}$, and $\boldsymbol{j} = F\boldsymbol{w}$. *Then we have* $i_k = j_k$, *for all* $k = 1 \ldots n$.

proof:

By construction, using equation (4.11), $F = \begin{bmatrix} L & \boldsymbol{d} \\ \boldsymbol{d}^t & h \end{bmatrix}$. Partitioning $\boldsymbol{w}$ in the same way yields $\boldsymbol{w} = \begin{bmatrix} \boldsymbol{u} \\ 0 \end{bmatrix}$. Substituting and multiplying, we obtain $F\boldsymbol{w} = \begin{bmatrix} L & \boldsymbol{d} \\ \boldsymbol{d}^t & h \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ 0 \end{bmatrix} = \begin{bmatrix} L\boldsymbol{u} \\ \boldsymbol{d}^t\boldsymbol{u} \end{bmatrix} = \begin{bmatrix} \boldsymbol{i} \\ \boldsymbol{d}^t\boldsymbol{u} \end{bmatrix}$.

■

Thus, any Laplacian matrix is isomorphic to a resistive network. Now we can move between three different, equivalent representations of the same object. In what follows, we will cease to distinguish between undirected weighted graphs and resistive networks, and will use both representations interchangeably.

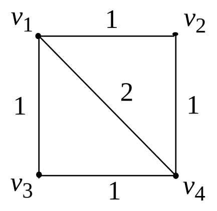
4.3 Support Trees and Resistive Networks

In Chapter 3, we presented the intuition for support trees in terms of making communication across the mesh more efficient. We now augment this intuition by presenting support trees in terms of resistive networks.

The previous subsection showed that a Laplacian matrix corresponds to a grounded resistive network; the Laplacian corresponds to the conductance matrix that defines the network. Furthermore, matrix-vector multiplication is a mapping from voltages at each node to net current flow at each node. For example, consider the simple example of the conductance matrix shown in Figure 4.5. The top of the figure shows the conductance matrix and the corresponding network in which edges are labeled with conductances. The bottom of the figure shows a matrix-vector multiply and its network interpretation. The input vector denotes the voltages assigned to each node of the network. The individual terms in the matrix-vector multiply give the current flow from one node to another: for example, $a_{12}u_2 = i_{12} = -1/2$, which is the current flow from v_1 to v_2. The output vector gives the net current flow at each node. A negative net flow indicates flow into the node — the node is a current sink. A positive net flow is flow out of the node — the node is a current source. In the figure, node v_1 is a sink, v_4 is a source, and nodes v_2 and v_3 have no net current flow.

a)

$$\begin{array}{c} A \\ \begin{bmatrix} 4 & -1 & -1 & -2 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -2 & -1 & -1 & 4 \end{bmatrix} \end{array}$$



b)

$$\begin{array}{ccccc} A & & \boldsymbol{u} & & \boldsymbol{i} \\ \begin{bmatrix} 4 & -1 & -1 & -2 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -2 & -1 & -1 & 4 \end{bmatrix} & & \begin{bmatrix} 0 \\ 1/2 \\ 1/2 \\ 1 \end{bmatrix} & = & \begin{bmatrix} -3 \\ 0 \\ 0 \\ 3 \end{bmatrix} \end{array}$$

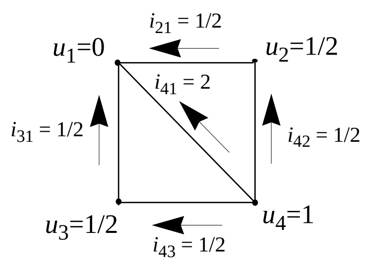


Figure 4.5: ***Matrix-vector multiplication maps voltages to currents.***
a) *A conductance matrix and the corresponding network.*
b) *A matrix-vector multiply, and the resulting interpretation in terms of current flow.*

The nodes v_2 and v_3 which have no net current flow are particularly interesting to us. Consider node v_2. As seen in the figure, v_2 has current flowing both in and out of it; the inward flow exactly balances the outward flow, however. The reason that the inward and outward flows balance is that the voltage at v_2 is the weighted average of the voltages at its neighbors. A function that satisfies the local averaging property is called *harmonic* [Doyle and Snell (1984)], and so we call nodes like v_2 and v_3 which have zero net current flow *harmonic nodes*. The concept of harmonic nodes are essential to the construction of support trees. All non-leaf nodes of support trees are harmonic nodes.

Consider the process of constructing a support tree T for the coefficient matrix A (see §3.2). Suppose that, at some step in the process, a subgraph G_i is partitioned into components G_{i0} and G_{i1}. Then, a support tree node is created with edges to each component, and each edge is weighted to support the total communication out of the connecting component. If the support tree node is harmonic, then its voltage is the weighted average of all the voltages in the tree, and it doesn't alter the net current flow through it. That is, the net communication between each subgraph and the rest of the graph is left unchanged. The *specific* communication effects are of course altered, but the *overall* communication with respect to each subgraph is unchanged. Therefore, a support tree effectively reduces the communication distance while accurately preserving global communication effects.

Internal support tree nodes are made harmonic by the simple process of setting their net current flow to zero. To be more specific, let T be a support tree for an nxn Laplacian matrix A. Then, as explained in §3.3, there is an ordering of the nodes in T such that

$$T = \begin{bmatrix} D & R \\ R^t & S \end{bmatrix} \tag{4.12}$$

where D is an nxn diagonal matrix and corresponds to the nodes of A, S is square and corresponds to the internal nodes of the tree and their interconnections, and R and R^t correspond to the connections between nodes of A and internal nodes of T. Now, if $\boldsymbol{r}^{(i)} = \boldsymbol{b} - A\boldsymbol{x}^{(i)}$ is the ith residual, then the usual preconditioned system that must be solved at the ith step is $B\boldsymbol{z}^{(i)} = \boldsymbol{r}^{(i)}$, where B is the preconditioner. The preconditioned system using support trees with harmonic internal nodes is $T\boldsymbol{w} = \boldsymbol{s}^{(i)}$, where, for T in the block diagonal form above, $\boldsymbol{s}^{(i)} = \begin{bmatrix} \boldsymbol{r}^{(i)} \\ 0 \end{bmatrix}$, and $\boldsymbol{w} = \begin{bmatrix} \boldsymbol{z}^{(i)} \\ \boldsymbol{y} \end{bmatrix}$.

Augmenting the residual with zeros ensures that the internal nodes of the support tree are harmonic so that the tree only propagates local averages. The quantities in the vector $\boldsymbol{y}$ represent weighted average voltages for different subgraphs, and can be ignored for the purposes of computing the next step in the PCG algorithm.

Recall from Chapter 3 that the leaves of the support tree correspond to nodes of the original mesh. By construction, a leaf of a support tree is grounded if and only if the corresponding mesh node is grounded. Since the coefficient matrix A is assumed to be non-singular, at least one node of A must be grounded; hence, at least one leaf node of T is grounded. By construction, T is Laplacian. Therefore, T is non-singular.

It is interesting to note that, in Chapter 3, we came to the same result about how to augment the residual vectors by a completely different route, and also concluded that the values in vector $\boldsymbol{y}$ could be ignored.

4.4 Generalized Eigenvalues and Support Numbers

To begin this section, recall from Chapter 2 that λ is an *eigenvalue* of a matrix A if there exists a vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = \lambda\boldsymbol{x}$. We denote the set of eigenvalues of A by $\lambda(A)$.

Further recall from Chapter 2 that λ is a *generalized eigenvalue* of the ordered pair of matrices (A,B) if there exists a vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = \lambda B\boldsymbol{x}$. We denote the set of generalized eigenvalues of (A,B) by $\lambda(A,B)$. Note that $\lambda(A) = \lambda(A,I)$, where I is the identity matrix. There exist n generalized eigenvalues if and only if the matrix B is non-singular. If B is singular, there may be fewer than n eigenvalues — this is the case when $A\boldsymbol{x} \neq 0$, but $B\boldsymbol{x} = 0$. In addition, there may be an infinite number of eigenvalues if A and B have a common null space — that is, there exists $\boldsymbol{x}$ such that $A\boldsymbol{x} = 0 = B\boldsymbol{x}$ [Golub and VanLoan (1989)].

When there exists a vector $\boldsymbol{x}$ and a unique λ such that $A\boldsymbol{x} = \lambda B\boldsymbol{x}$, we will call λ a *finite generalized eigenvalue*. In what follows, we will be concerned with only finite generalized eigenvalues. In application, the preconditioner B will be non-singular, and only finite generalized eigenvalues will exist. For simplicity of exposition, we make the following definition:

4.3 ***Definition***: *λ is a finite generalized eigenvalue of the ordered pair of matrices (A,B) if there exists a vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = \lambda B\boldsymbol{x}$, and λ is unique.* *We denote the collection of finite generalized eigenvalues by $\underline{\lambda}(A,B)$.*

The primary tool that we will use for bounding finite generalized eigenvalues is the *Support Lemma*. This lemma appears in a slightly different form as Corollary 2.1 in Axelsson (1992).

4.4 ***Lemma*** (Support Lemma): *Suppose that A and B are Laplacian matrices. If $\lambda \in \underline{\lambda}(A,B)$ is a finite generalized eigenvalue, and τB-A is positive semi-definite, then $\lambda \le \tau$.*

proof:

First, assume that both A and B are non-singular and hence positive definite. Suppose the lemma is false and there exists $\lambda \in \underline{\lambda}(A,B)$ such that $\lambda > \tau$. Let $\boldsymbol{y}$ be an eigenvector corresponding to λ. Then

$$\boldsymbol{y}^t(\lambda B - A)\boldsymbol{y} = 0$$

Now $\lambda > \tau$, so there is an $\varepsilon > 0$ such that $\lambda - \varepsilon = \tau$. This yields

$$\begin{aligned}\boldsymbol{y}^t[\tau B - A]\boldsymbol{y} &= \boldsymbol{y}^t[(\lambda-\varepsilon)B - A]\boldsymbol{y}\\ &= \boldsymbol{y}^t(\lambda B - A)\boldsymbol{y} - \varepsilon\boldsymbol{y}^t B\boldsymbol{y}\end{aligned}$$

which must be less than zero, since the first term is zero and B is positive definite. This contradicts the assumption of τB-A positive semi-definite. Therefore, we must have $\lambda \le \tau$.

Now, if either A or B is singular, we simply note that any $\boldsymbol{y}$ corresponding to $\lambda \in \underline{\lambda}(A,B)$, where λ is a finite generalized eigenvalue, may not lie in the null space of B. Hence the term $\varepsilon\boldsymbol{y}^t B\boldsymbol{y}$ above is always positive, and the result follows.

■

Based on the Support Lemma, we define the concept of *support* of one matrix for another.

4.5 ***Definition***: *The support, $\sigma = \sigma(A,B)$, of matrix B for A is the greatest lower bound over all τ satisfying the Support Lemma. That is, $\sigma = \liminf\{\tau$: τB-A is positive semi-definite$\}$.*

Note that the support number of an ordered pair of matrices (A,B) is an upper bound on the largest finite generalized eigenvalue of (A,B). That is, if $\lambda \in \underline{\lambda}(A,B)$, then $\lambda \le \sigma(A,B)$. Therefore, $\underline{\lambda}_{max}(A) = \underline{\lambda}_{max}(A,I) \le \sigma(A,I)$.

The concept of support has a physical meaning in terms of resistive networks. Consider the networks corresponding to matrices A and B. $\sigma(A,B)$ is the gain factor that must be applied to B to guarantee that, for a given set of voltages, at least as much current flows in B as in A. This concept is made clearer in the next lemma.

4.6 ***Lemma***: *Let A be a Laplacian matrix corresponding to the graph G = (V_G,E_G), consisting of a single simple cycle on n nodes, with edge weights given by w_{jk}. Let H = (V_H,E_H) be the graph corresponding to G, but missing edge e(i,j), and let B be the matrix corresponding to H. Then* $\sigma(A,B) \le \zeta$, *where*

$$\zeta = \left(w_{ij} \cdot \sum_{(k,l) \in E_H} \frac{1}{w_{kl}}\right) + 1 \tag{4.13}$$

In particular, if G is a unit weight graph (that is, $w_{jk} = 1, \forall j \neq k \in \{1...n\}$*), then* $\sigma(A,B) \le n$.

proof:

Assume without loss of generality that the cycle contain n nodes which are numbered in order around the cycle, and that the deleted edge is between nodes n and 1.

Let $\boldsymbol{e}_{ij}$, $i<j$, be the vector with a 1 in position i, -1 in position j, and 0 elsewhere. Then $E_{ij} = w_{ij}\boldsymbol{e}_{ij}\boldsymbol{e}_{ij}^t$ is the conductance matrix corresponding to a network with a single resistive connection between node i and node j of conductance w_{ij}. We call $\boldsymbol{e}_{ij}\boldsymbol{e}_{ij}^t$ a primitive matrix.

The matrix of any ungrounded resistive network can be formed as the weighted sum of primitive matrices. Hence,

$$A = \left(\sum_{i=1}^{n-1} w_{i,i+1}\boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{i,i+1}\right) + w_{1,n}\boldsymbol{e}_{1,n}\boldsymbol{e}^t_{1,n} \tag{4.14}$$

and

$$B = \sum_{i=1}^{n-1} w_{i,i+1}\boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{i,i+1} \tag{4.15}$$

We need to find τ such that τB-A is positive semi-definite. By substituting (4.14) and (4.15) into $\tau B - A$, then expanding and rearranging terms, we get

$$\tau B - A = (\tau - 1)\left(\sum_{i=1}^{n-1} w_{i,i+1}\boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{i,i+1}\right) - w_{1,n}\boldsymbol{e}_{1,n}\boldsymbol{e}^t_{1,n} \tag{4.16}$$

Now, it is easy to see that

$$\boldsymbol{e}_{1,n} = \boldsymbol{e}_{1,2} + \boldsymbol{e}_{2,3} + \ldots + \boldsymbol{e}_{n-1,n} \tag{4.17}$$

Substituting (4.17) into (4.16) yields

$$\tau B - A = (\tau - 1)\sum_{i=1}^{n-1} w_{i,i+1}\boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{i,i+1} - w_{1,n}\sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-1}\boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{j,j+1}\right) \tag{4.18}$$

Setting $(\tau - 1) = w_{1,n} \cdot \sum_{(k,l) \in H} \frac{1}{w_{k,l}}$, then expanding and rearranging terms:

$$B - A = w_{1,n}\sum_{i=1}^{n-2}\left(\sum_{j=i+1}^{n-1} \frac{w_{i,i+1}}{w_{j,j+1}}\boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{i,i+1} - \boldsymbol{e}_{i,i+1}\boldsymbol{e}^t_{j,j+1} - \boldsymbol{e}_{j,j+1}\boldsymbol{e}^t_{i,i+1} + \frac{w_{j,j+1}}{w_{i,i+1}}\boldsymbol{e}_{j,j+1}\boldsymbol{e}^t_{j,j+1}\right) \tag{4.19}$$

But the innermost sums are easily recognized as outer products, so (4.19) reduces to:

$$\tau B - A = w_{1,n} \sum_{i=1}^{n-2} \left(\sum_{j=i+1}^{n-1} \left(\sqrt{\frac{w_{i,i+1}}{w_{j,j+1}}} \boldsymbol{e}_{i,i+1} - \sqrt{\frac{w_{j,j+1}}{w_{i,i+1}}} \boldsymbol{e}_{j,j+1} \right) \left(\sqrt{\frac{w_{i,i+1}}{w_{j,j+1}}} \boldsymbol{e}_{i,i+1} - \sqrt{\frac{w_{j,j+1}}{w_{i,i+1}}} \boldsymbol{e}_{j,j+1} \right)^t \right) \tag{4.20}$$

Each of the terms on the right hand side above is an outer product, and hence is positive semi-definite. Therefore, the entire right hand side is positive semi-definite and by Lemma 3.1 we have the desired result.

Finally, for a cycle with unit edges, substitute $w_{ij} = 1$, $1 \le i,j \le n$ into $(\tau - 1) = w_{1,n} \cdot \sum_{(k,l) \in H} \frac{1}{w_{k,l}}$ and we get $(\tau\text{-}1) = n$ -1, or $\tau = n$.

■

Physics texts and texts on circuit theory generally discuss the rules that are used to reduce circuit complexity by combining multiple circuit elements into a single element. It is useful at this point to review some of these rules. In particular, the two rules which we shall employ deal with resistive (conductive) elements of the same type connected either in parallel or in series.

When two resistors of resistances r_1 and r_2 are connected in series, they may be replaced by a single resistor of resistance r, where

$$r = r_1 + r_2 \tag{4.21}$$

This law can be phrased in terms of conductors. Using the fact that conductance is the reciprocal of resistance, we have that two conductors of conductances c_1 and c_2 connected in series may be replace by a single conductor of conductance c, where

$$\frac{1}{c} = \frac{1}{c_1} + \frac{1}{c_2} \tag{4.22}$$

When resistors are connected in parallel, then the formulas above hold with the roles of resistance and conductance reversed. That is, two resistors of resistances r_1 and r_2 in parallel can be combined into a single resistor of resistance r, where

$$\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2} \tag{4.23}$$

And two conductors of conductances c_1 and c_2 connected in parallel can be combined into a single conductor of conductance c, where

$$c = c_1 + c_2 \tag{4.24}$$

Figure 4.6 illustrates the rules for circuit reduction.

Since we have demonstrated that Laplacian matrices and resistive networks are isomorphic, it should come as no surprise that there exist matrix operations which correspond to the rules for circuit reduction.

The rule for combining parallel conductors corresponds to matrix addition, and yields the important property of linearity for the networks that we are dealing with. If $A = A_1 + A_2$, then for every vector $\boldsymbol{x}$, $A\boldsymbol{x} = A_1\boldsymbol{x} + A_2\boldsymbol{x}$. Similarly, if we have two networks defined over the same set of nodes, and with corresponding matrices B_1 and B_2, then we can short together corresponding nodes of the two networks to form a combined network with matrix $B = B_1 + B_2$. An example is shown in Figure 4.7.

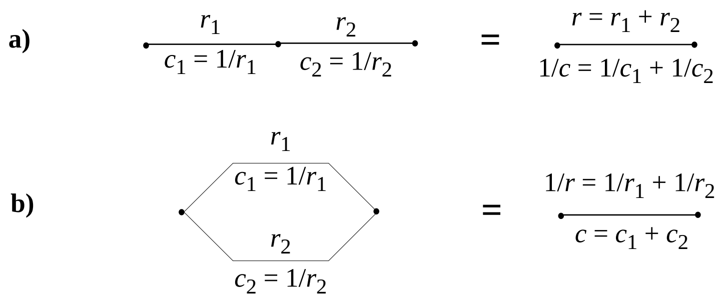
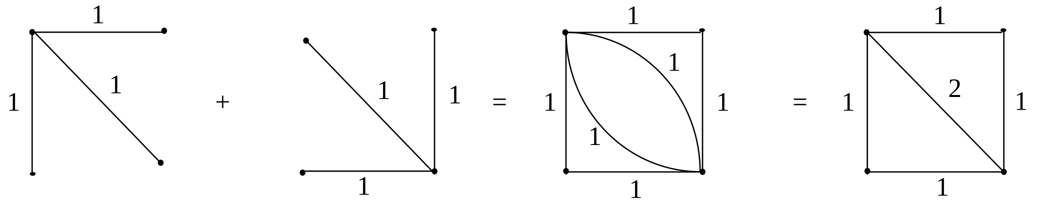


Figure 4.6: ***Rules for circuit reduction.***
a) *resistors/conductors in series*
b) *resistors/conductors in parallel*



$$\begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix} = \begin{bmatrix} 4 & -1 & -1 & (-1-1) \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ (-1-1) & -1 & -1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & -1 & -1 & -2 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -2 & -1 & -1 & 4 \end{bmatrix}$$

Figure 4.7: ***The linearity of resistive networks.***
The top of the figure illustrates combining resistive networks to form a single network.
The bottom figure illustrates the operation with the matrices corresponding to the networks.

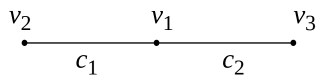
The rule for combining conductors in series corresponds to a single step of Gaussian elimination in which only a single row/column (a single node) is reduced. Consider the example in Figure 4.8, where we have replaced the series rule for conductors by the equivalent $c = c_1c_2/(c_1 + c_2)$. and ordered the nodes for convenient elimination. In Figure 4.8a, a simple path on 3 points is illustrated, along with its corresponding Laplacian matrix. To eliminate v_1, the central node, the off-diagonals in row and column 1 of the Laplacian must be eliminated. Elimination of $A(2,1)$ and $A(1,2)$ is performed by adding α_1 times row1 to row2, and α_1 times column 1 to column 2, where $\alpha_1 = c_1/(c_1 + c_2)$. This adds fill of $-c_1c_2/(c_1 + c_2)$ to entries $A(2,3)$ and $A(3,2)$, and subtracts $c_1^2/(c_1 + c_2)$ from entry $A(2,2)$, yielding $A_1(2,2) = c_1c_2/(c_1 + c_2)$. A similar analysis shows that the second step of Gaussian elimination changes only $A_1(3,3)$, yielding $A_2(3,3) = c_1c_2/(c_1 + c_2)$. From Figure 4.8c, the 2x2 block at the lower right of A_2 is easily recognized as the Laplacian matrix corresponding to a network with two nodes and a single connecting edge of conductance $c_1c_2/(c_1 + c_2)$.

Gaussian elimination also explains the more general *star-delta* transformation of circuit theory [Bollabas (1979)]. In fact, the reduction of conductors in series is simply the star-delta transformation with only two input conductors. Figure 4.9 illustrates the star-delta transformation with three input conductors; the configurations of the input and output networks for this case are the star and delta for which the transformation was named.

Figure 4.9 suggests a formula for determining the conductance of any wire obtained through star-delta transformation. Let v_0 be the central node with neighbors $v_1,...,v_n$ in a star configuration. Let c_i be the conductance of the wire connecting v_0 to v_i. Reduction of v_0 will connect all the other nodes. Examination of the figure suggests that c_{ij}, the

a)

$$\begin{bmatrix} c_1+c_2 & -c_1 & -c_2 \\ -c_1 & c_1 & 0 \\ -c_2 & 0 & c_2 \end{bmatrix}$$



b)

$$\begin{bmatrix} 1 & 0 & 0 \\ -c_1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} c_1+c_2 & -c_1 & -c_2 \\ -c_1 & c_1 & 0 \\ -c_2 & 0 & c_2 \end{bmatrix}\begin{bmatrix} 1 & -c_1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1+c_2 & 0 & -c_2 \\ 0 & \frac{c_1c_2}{c_1+c_2} & -\frac{c_1c_2}{c_1+c_2} \\ -c_2 & -\frac{c_1c_2}{c_1+c_2} & c_2 \end{bmatrix}$$

c)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -c_2 & 0 & 1 \end{bmatrix}\begin{bmatrix} c_1+c_2 & 0 & -c_2 \\ 0 & \frac{c_1c_2}{c_1+c_2} & -\frac{c_1c_2}{c_1+c_2} \\ -c_2 & -\frac{c_1c_2}{c_1+c_2} & c_2 \end{bmatrix}\begin{bmatrix} 1 & 0 & -c_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1+c_2 & 0 & 0 \\ 0 & \frac{c_1c_2}{c_1+c_2} & -\frac{c_1c_2}{c_1+c_2} \\ 0 & -\frac{c_1c_2}{c_1+c_2} & \frac{c_1c_2}{c_1+c_2} \end{bmatrix}$$

d)

$$\begin{bmatrix} \frac{c_1c_2}{c_1+c_2} & -\frac{c_1c_2}{c_1+c_2} \\ -\frac{c_1c_2}{c_1+c_2} & \frac{c_1c_2}{c_1+c_2} \end{bmatrix}$$

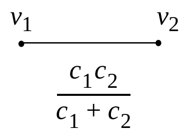


Figure 4.8: ***Gaussian elimination and circuit reduction.***
a) *Laplacian matrix corresponding to the path on three points shown*
b) *First step in reducing graph/matrix from* ***a***
c) S*econd step in reducing graph/matrix from* ***a***
d) *Laplacian matrix corresponding to the path on two points shown.*

conductance of the wire connecting v_i to v_j, is given by

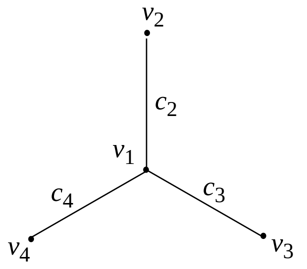
$$c_{ij} = \frac{c_i \cdot c_j}{\sum_k c_k} \qquad (4.25)$$

That is, the conductance of the wire connecting v_i to v_j upon reduction of v_0 is given by the product of the conductances of the wires from v_0 to v_i and v_j, divided by the sum of the conductances of all the wires connecting to v_0. This formula is easily proved by performing Gaussian elimination on v_0. This formula is useful because it allows us to easily determine conductances resulting from circuit reduction.

Resistive networks and linearity provide a clear physical intuition for Lemma 4.6 above. Consider supporting a cycle on n nodes with edges of unit weight by a simple path on n nodes, also with edges of unit weight. Figure 4.10 illustrates the example. The cycle contains two paths from node v_1 to node v_n: a path of length 1 and conductance 1, and a path of length n-1 and conductance 1/(n-1). The total conductance in the cycle from v_1 to v_n is therefore $1 + 1/(n-1) = n/(n-1)$. The total conductance in the path from v_1 to v_n is 1/(n-1). Therefore, for the path to support the cycle, the conductance of the path must be increased by a factor of *n*. This multiplication factor corresponds to the support number.

a)

$$\begin{bmatrix} c_2+c_3+c_4 & -c_2 & -c_3 & -c_4 \\ -c_2 & c_2 & 0 & 0 \\ -c_3 & 0 & c_3 & 0 \\ -c_4 & 0 & 0 & c_4 \end{bmatrix}$$



b)

$$\begin{bmatrix} c_2+c_3+c_4 & 0 & 0 & 0 \\ 0 & \frac{c_2c_3+c_2c_4}{c_2+c_3+c_4} & -\frac{c_2c_3}{c_2+c_3+c_4} & -\frac{c_2c_4}{c_2+c_3+c_4} \\ 0 & -\frac{c_2c_3}{c_2+c_3+c_4} & \frac{c_2c_3+c_3c_4}{c_2+c_3+c_4} & -\frac{c_3c_4}{c_2+c_3+c_4} \\ 0 & -\frac{c_2c_4}{c_2+c_3+c_4} & -\frac{c_3c_4}{c_2+c_3+c_4} & \frac{c_2c_4+c_3c_4}{c_2+c_3+c_4} \end{bmatrix}$$

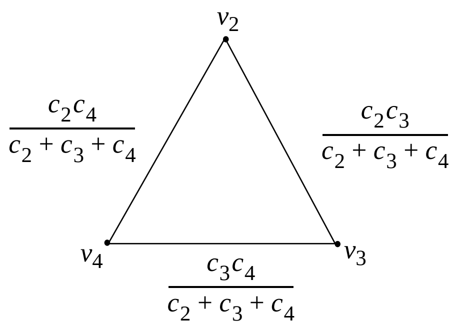

v_2
$\frac{c_2c_4}{c_2+c_3+c_4}$
$\frac{c_2c_3}{c_2+c_3+c_4}$
v_4
v_3
$\frac{c_3c_4}{c_2+c_3+c_4}$

Figure 4.9: ***Gaussian elimination and the star-delta transformation.***
a) *Laplacian matrix and corresponding graph*
b) *Matrix after Gaussian elimination of row/column 1, and corresponding graph.*

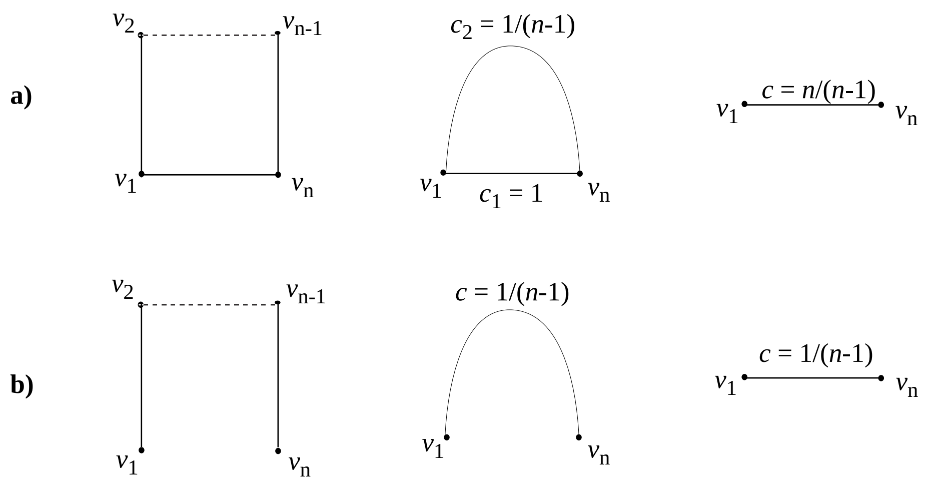


Figure 4.10: ***Support numbers and conductances.***
a) *In a simple cycle, the effective conductance from v_1 to v_n is n/(n-1)*
b) *In a simple path, the effective conductance from v_1 to v_n is 1/(n-1)*
For the path to support the cycle, its conductance must be increased by a factor of n.
This gain factor is equivalent to the support number.

We can compute the support number of any ordered pair of matrices (A,B), where A and B are Laplacian matrices, by applying the physical intuition above. Consider breaking up A into pieces $A_1+A_2+...+A_m$ in such a way that the pieces are particularly simple in structure. Next we break up B into corresponding pieces $B_1,...,B_m$, splitting conductors if necessary, so that B_i supports A_i. The following lemma shows how to analyze the support numbers of the pieces to determine the overall support number.

4.7 ***Lemma***: *Let A and B be Laplacian matrices corresponding to resistive networks G and H such that H is a subnetwork of G. Let* $\{G_1, G_2,..., G_m\}$ *be any partitioning of G such that* $G = \sum G_i$, *and let* $\{A_1, A_2,...,A_m\}$ *be the matrices corresponding to the* G_i. *Let* $\{H_1, H_2,..., H_m\}$ *be a corresponding partitioning of H, with associated matrices* $\{B_1,B_2,...,B_m\}$. *Then* $\sigma(A,B) \le max\{\sigma(A_i,B_i)\}$.

proof:

We want to find the minimum τ such that (τB-A) is positive semi-definite. Equivalently, this means finding the minimum τ such that for any vector $x \neq 0$, we have the following inequalities:

$$\boldsymbol{x}^t(\tau B - A)\boldsymbol{x} \ge 0$$

$$\boldsymbol{x}^t(\tau(B_1 + \ldots + B_m) - (A_1 + \ldots + A_m))\boldsymbol{x} \ge 0$$

$$\boldsymbol{x}^t(\tau B_1 - A_1)\boldsymbol{x} + \ldots + \boldsymbol{x}^t(\tau B_m - A_m)\boldsymbol{x} \ge 0$$

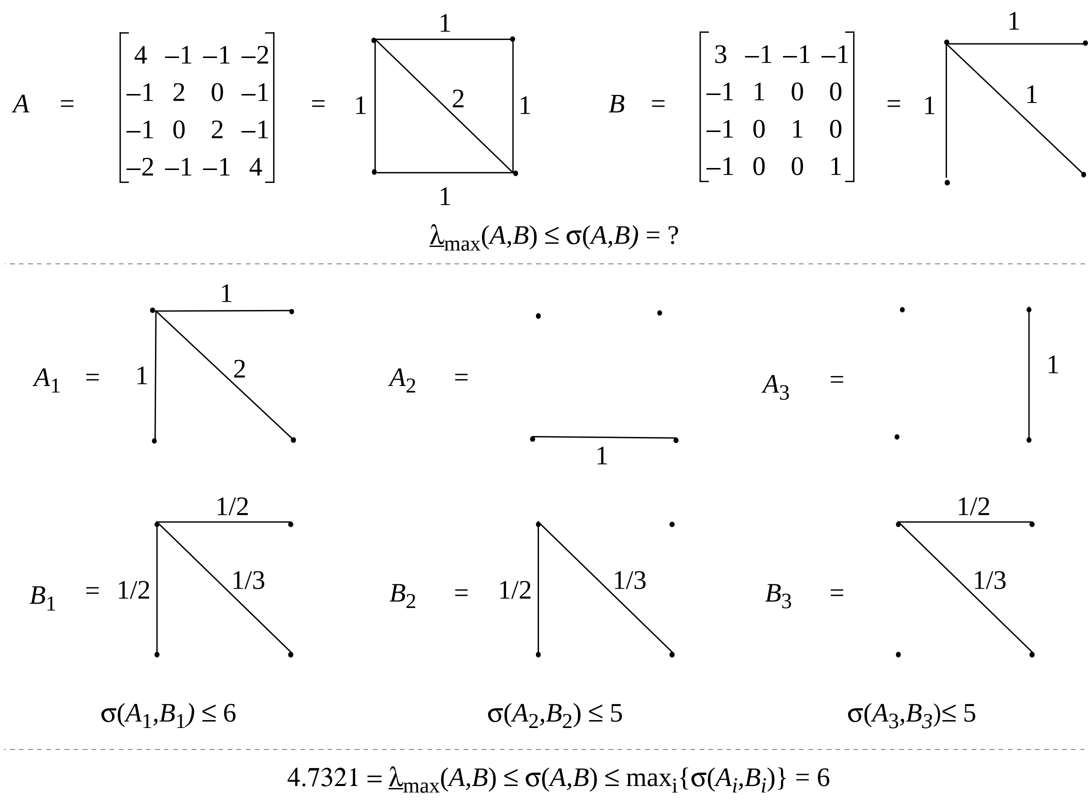
Clearly, if we can find any τ such that $\boldsymbol{x}^t(\tau B_i - A_i)\boldsymbol{x} \ge 0$ for all $\boldsymbol{x}$ and i, then $\boldsymbol{x}^t(\tau B - A)\boldsymbol{x} \ge 0$ for all $\boldsymbol{x}$**,** and so $\sigma(A, B) \le \tau$. Therefore, $\sigma(A,B) \le max\{\sigma(A_i,B_i)$: $i = 1,...,m\}$.

■

The lemmas and examples of this section provide some physical intuition for the concept of support. The key in bounding $\underline{\lambda}_{max}(A,B)$ is to determine a τ such that τB-A is positive semi-definite. The method which we have elaborated shows that τ may be considered to be the minimum number of copies of B that are necessary to replace each conductor (edge) of A by a path of conductors in B. We determine the number of copies needed by partitioning B so that every edge of A is supported by the edges in a single partition of B (although several edges of A may be supported by a single partition of B), and finding the partition with the greatest support requirement.

To see how to apply this general combinatorial procedure for bounding generalized eigenvalues, consider bounding the maximum finite generalized eigenvalue $\underline{\lambda}_{\max}(A,B)$ for the matrices/networks shown in Figure 4.11. The network B can be partitioned into three pieces as shown, one supporting the edges of A in common with B, and one each for the edges of A not in B. In the first partition, each edge of B_1 supports the corresponding edge of A_1. The minimum support is for the diagonal edge: the wire with conductance 1/3 must be multiplied by a gain factor of 6 in order to support the corresponding wire of conductance 2. Hence, $\sigma(A_1,B_1) \le 6$. In each of the next two partitions, a single wire of conductance 1 must be supported by a path of length 2 with conductances of 1/3 and 1/2. The total conductance of the path is 1/5, so a gain factor of 5 is required for the path to support the edge. For each partition i, $\sigma(A_i,B_i) \le 6$, so by the lemma above, $\sigma(A,B) \le 6$. By the Support Lemma, $\underline{\lambda}_{max}(A,B) \le \sigma(A,B) \le 6$. In fact, using Matlab [Mathworks, Inc. (1992)], we obtained $\underline{\lambda}_{max}(A,B) = 4.7321$.

It is useful to assign some terminology to associate with our combinatorial procedure for bounding generalized eigenvalues. Let the edges in B be called *support edges*. Then the edges in A are *supported edges*. For each partition i, we can compute the total conductance of the edges in A_i that are supported, at least in part, by $e \in B_i$. For edge e, we call the ratio of total supported conductance to support conductance the *congestion* of e in partition i of the support mapping, and denote it by $\gamma_e(A,B)$. Similarly, if f is an edge in A, then f is supported by a path in B; that is, the current flowing through f is routed via some path in B. We call the length of the support path the *dilation* of f in the support



$A = \begin{bmatrix} 4 & -1 & -1 & -2 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -2 & -1 & -1 & 4 \end{bmatrix} =$
1
1
2
1
1
$B = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} =$
1
1
1
$\lambda_{\max}(A,B) \le \sigma(A,B) = ?$
$A_1 =$ 1 1 2
$A_2 =$ 1
$A_3 =$ 1
$B_1 =$ 1/2 1/2 1/3
$B_2 =$ 1/2 1/3
$B_3 =$ 1/2 1/3
$\sigma(A_1,B_1) \le 6$
$\sigma(A_2,B_2) \le 5$
$\sigma(A_3,B_3) \le 5$
$4.7321 = \lambda_{\max}(A,B) \le \sigma(A,B) \le \max_i\{\sigma(A_i,B_i)\} = 6$

Figure 4.11: ***Partitioning matrices/networks for eigenvalue analysis.*** *At the top of the figure, networks A and B are illustrated. An upper bound on $\lambda_{max}(A,B)$ is obtained by partitioning A and B and determining the support number of the partitions.*

mapping, and denote it by $\delta_f(A,B)$. Note that congestion and dilation as used here are generalizations of the notions of congestion and dilation of graph embeddings (see the review in Chapter 2). For any support mapping of (A,B), the support number is bounded above by the maximum product of dilation and congestion, where dilation is determined for each edge in A, and congestion is determined over all the edges in B supporting the given edge in A. That is,

$$\sigma(A,B) \le \max\{\delta_f(A,B) \cdot \gamma_e(A,B) : f \in E_{A_i}, e \in E_{B_i} \ni e \text{ supports } f\}. \tag{4.26}$$

For example, consider again the two networks in Figure 4.11. The dilation for each of the edges of A_1 is 1, since each edge is supported by a single edge of B_1. The congestion for the horizontal edge of B_1 is 2, as is the congestion for the vertical edge. The congestion for the diagonal edge of B_1 is 6. The maximum product of congestion times dilation in partition 1 is therefore $\max\{2 \cdot 1, 2 \cdot 1, 6 \cdot 1\} = 6$. Similarly, the dilations for the edges in A_2 and A_3 are both 2, since both edges are routed over paths of length 2. The maximum congestion along either support path is 3, so the support numbers for partitions 2 and 3 are both 6. The support number obtained using the maximum product of congestion times dilation is 6, as was obtained when path conductances were computed, but the bounds for partitions 2 and 3 were not as tight. The congestion times dilation approach is, in general, simpler, but not as accurate as computing exact path conductances.

4.5 Condition Number Bounds for Support Trees: Preliminary Lemmas

The PCG method for coefficient matrix A and preconditioner B is known to have a convergence rate of $O(\sqrt{\kappa(B^{-1}A)})$, where the condition number of $B^{-1}A$, $\kappa(B^{-1}A)$, is given by $\kappa(B^{-1}A) = \lambda_{\max}(B^{-1}A)/\lambda_{\min}(B^{-1}A)$

(see Theorem 2.12 and the discussion of PCG in §2.4.2.3).

If $\lambda \in \lambda(B^{-1}A)$, then $B^{-1}A\boldsymbol{x} = \lambda\boldsymbol{x}$ for some vector $\boldsymbol{x}$. By multiplying both sides by B, this implies that $A\boldsymbol{x} = \lambda B\boldsymbol{x}$. That is, λ is a finite generalized eigenvalue of the ordered pair of matrices (A,B). Hence, bounding the condition number of a preconditioned system involves bounding generalized eigenvalues. The lemma below shows how to use the technology developed in the previous subsection to bound the condition number of a preconditioned system, which we will refer to as the *generalized condition number.*

4.8 ***Lemma:*** *Let A and B be non-singular Laplacian matrices, and suppose that B is a preconditioner for A. Then,*

$$\kappa(B^{-1}A) \le \sigma(A, B) \cdot \sigma(B, A)$$

proof:

$$\begin{aligned}\kappa(B^{-1}A) &= \lambda_{\max}(B^{-1}A)/\lambda_{\min}(B^{-1}A)\\ &= \lambda_{\max}(A, B)/\lambda_{\min}(A, B)\\ &= \lambda_{\max}(A, B) \cdot \lambda_{\max}(B, A)\\ &\le \sigma(A, B) \cdot \sigma(B, A)\end{aligned}$$

■

In Chapter 3, we addressed the problem of using a support tree matrix *T* as a preconditioner for a matrix *A*. The difficulty for application purposes was the difference in size between *T* and *A*. We proved that Gaussian elimination could be applied to the problem to yield an equivalent preconditioner *K* for A that was the same size as *A*. However, we did not use *K*, but rather showed how to use the more computationally efficient *T* by adding and then deleting variables. We now need to show how to analyze the condition number that results from preconditioning *A* with *T*, which means that once again we have to find a way to deal with the difference in sizes.

4.9 *Lemma*: *Let A and B be Laplacian matrices. Let G be a matrix representing 1 or more reduction steps of Gaussian elimination. Then* $\underline{\lambda}(A,B) = \underline{\lambda}(GAG^t,GBG^t)$.

proof:

Let $\lambda \in \underline{\lambda}(A,B)$, and $\boldsymbol{x}$ a corresponding generalized eigenvector. Then

$$A\boldsymbol{x} = \lambda B\boldsymbol{x}$$

$$G \cdot A\boldsymbol{x} = G \cdot \lambda B\boldsymbol{x}$$

$$GA\boldsymbol{x} = \lambda GB\boldsymbol{x}$$

Furthermore, *G* is non-singular, so there exists $\boldsymbol{y}$ such that $G^t\boldsymbol{y} = \boldsymbol{x}$. Making this substitution yields

$$GAG^t\boldsymbol{y} = \lambda GBG^t\boldsymbol{y}$$

So $\lambda \in \underline{\lambda}(GAG^t, GBG^t)$.

Since *G* is non-singular, we can perform the same sequence of operations in reverse for any

$\lambda \in \underline{\lambda}(GAG^t, GBG^t)$ to show that $\lambda \in \underline{\lambda}(A,B)$, which proves the lemma.

■

Lemma 4.9 allows us to apply Gaussian elimination in our analysis without fear of changing the generalized eigenvalues. From a physical standpoint, this was expected — since Gaussian elimination is equivalent to circuit reduction, it is natural to reduce the complexity of circuits before attempting to analyze them. However, we still haven't dealt explicitly with the fact that the coefficient matrix and the support tree matrix are different in size. The following lemma is useful for this.

4.10 ***Lemma***: *Let A be an nxn non-singular Laplacian matrix, and let T be the mxm matrix of a support tree for A, ordered consistently with A. Let $\tilde{A}$ be the mxm matrix which contains A in the first diagonal nxn block, and is zero elsewhere. Let $\tilde{T}$ be obtained from T by Gaussian elimination of the internal nodes of the support tree with diagonal blocks K and S_d, where K is nxn and corresponds to the nodes of A. Then,*

i $\underline{\lambda}(\tilde{A}, T) = \underline{\lambda}(\tilde{A}, \tilde{T})$

ii $\lambda = 0$ *is in* $\underline{\lambda}(\tilde{A}, T)$ *with multiplicity m-n*

iii $\{\lambda \in \underline{\lambda}(\tilde{A}, T) : \lambda \neq 0\} = \underline{\lambda}(A, K)$

proof:

We assume that the nodes of A, which are the leaves of T, are ordered v_1,...,v_n, so that any node v_j with $j > n$ is an internal node of the support tree.

To prove *i*), let G be the matrix that implements Gaussian elimination of the internal nodes of T. Gaussian elimination of the internal nodes adds multiples of the rows and columns n+1 through m to other rows and columns. Since rows/columns n+1 through m are identically zero in $\tilde{A}$, we have

$$G\tilde{A}G^t = G\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}G^t = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} = \tilde{A}$$

And, by Lemma 4.9,

$$\underline{\lambda}(\tilde{A}, T) = \underline{\lambda}(G\tilde{A}G^t, GTG^t) = \underline{\lambda}(\tilde{A}, \tilde{T})$$

To prove *ii*), let $\boldsymbol{y}_i$ be the (m-n)-vector that has a 1 in position i and is zero elsewhere. Let $\boldsymbol{z}_i$ be the m-vector given by $\boldsymbol{z}_i = [\boldsymbol{0}^t, \boldsymbol{y}_i^t]^t$; $\boldsymbol{z}_i$ has a 1 in position n+i, and is zero elsewhere. Then, for $i \in \{1, \ldots, m-n\}$, $\tilde{A}\boldsymbol{z}_i = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{y}_i^t \end{bmatrix} = \boldsymbol{0}$, and $\lambda T\boldsymbol{z}_i = \boldsymbol{0}$, so $\lambda \in \underline{\lambda}(\tilde{A}, T)$ with multiplicity m-n.

To prove *iii*), we have from *i*) and *ii*) that $\lambda = 0$ is a generalized eigenvalue of ($\tilde{A}$, $\tilde{T}$) with multiplicity m-n. Furthermore, as in *ii*), the m-vectors $\boldsymbol{z}_i$ are eigenvectors corresponding to $\lambda = 0$. Since $\tilde{A}$ is symmetric and $\tilde{T}$ is symmetric and non-singular, there are m orthogonal generalized eigenvectors of ($\tilde{A}$, $\tilde{T}$). The vectors $\boldsymbol{z}_i$ comprise m-n of the orthogonal generalized eigenvectors, so the remaining n must have the form $\boldsymbol{v} = [\boldsymbol{x}^t, \boldsymbol{0}^t]^t$, with $\tilde{A}\boldsymbol{v} = \lambda\tilde{T}\boldsymbol{v}$ for some $\lambda \in \underline{\lambda}(\tilde{A}, \tilde{T})$. But then we have

$$\tilde{A}\boldsymbol{v} = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} A\boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix}$$

and

$$\lambda\tilde{T}\boldsymbol{v} = \lambda\begin{bmatrix} K & 0 \\ 0 & S_d \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix} = \lambda\begin{bmatrix} K\boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix}$$

Therefore, we must have $A\boldsymbol{x} = \lambda K\boldsymbol{x}$, so $\lambda \in \underline{\lambda}(A, K)$.

■

4.11 ***Corollary***: *Let A be a non-singular Laplacian and T the matrix of a support tree constructed for A. Let* $\tilde{A}$, $\tilde{T}$, *and K be as defined in Lemma* 4.10. *Then* $\kappa(K^{-1}A) = \sigma(\tilde{A}, T) \cdot \sigma(K, A)$.

proof:

From Lemma 4.8, $\kappa(K^{-1}A) \le \sigma(A, K) \cdot \sigma(K, A)$.

From Lemma 4.10, $\underline{\lambda}_{\max}(A,K) = \underline{\lambda}_{\max}(\tilde{A}, \tilde{T}) = \underline{\lambda}_{\max}(\tilde{A},T)$, so $\sigma(A,K) = \sigma(\tilde{A},T)$

■

Lemma 4.10 and Corollary 4.11 accurately reflect our physical intuition that equivalent circuits should have equivalent effects. That is, the tree and the reduced tree have equivalent effects on the original network. We can now use the network obtained from the support tree by circuit reduction operations to analyze the effects of the support tree.

4.6 Condition Numbers for Regular Meshes

In this section, we prove condition number bounds for regular meshes in d-dimensions. The meshes are assumed to be d-dimensional cubes with n nodes on a side, where $n = 2^k$. Each mesh can be embedded in the unit d-dimensional cube with nodes located at all points given by $(i_1\Delta, i_2\Delta,...,i_d\Delta)$, where $\Delta = 1/(n\text{-}1)$. Each node v is connected to all other nodes w for which the distance between v and w is equal to Δ. Thus, for d=1, we have a unit weight path, for d=2 a square mesh, for d=3 a cubic mesh, and so on. We further assume that all edges have unit weight. The proof techniques demonstrated in these sections hold for regular meshes with non-unit weights, as long as the difference between maximum and minimum edge weights is bounded by a constant.

We assume that all support trees are constructed using recursive coordinate partitioning. Therefore, at each step, the current subgraph is partitioned into $2d$ pieces by coordinate bisection with respect to each coordinate axis. Since $n = 2k$, the support trees are all regular $2d$-ary trees of uniform depth $\log_2 n$. All edges are weighted by the total weight of the edges on the boundary of the induced subgraph.

To simplify the presentation, we will cease to distinguish between a matrix and its corresponding graph. Therefore, the coefficient matrix A will correspond to the mesh $A = (V_A,E_A)$. The support tree matrix T will correspond to a support tree $T = (V_T,E_T)$. $\tilde{A}$ will denote the mxm matrix which contains A in the first diagonal nxn block, and is zero elsewhere. $\tilde{T}$ will denote the matrix obtained from T by Gaussian elimination of the internal nodes of the support tree with diagonal blocks K and S_d, where K is nxn and corresponds to the nodes of A. We will refer to K as the *reduced tree*.

The basic proof technique demonstrated here is straightforward and based directly on the results from the previous section. We have, from Lemmas 4.9 and 4.10, that the effect of the tree on the mesh is the same as the effect of the reduced tree K on the mesh. Furthermore, from Corollary 4.11, $\kappa(K^{-1}A) = \sigma(\tilde{A}, T) \cdot \sigma(K, A)$. Therefore, we will prove a bound on the condition number by first embedding the mesh in the tree and determining $\sigma(\tilde{A},T)$ then embedding the reduced tree in the mesh and determining $\sigma(K,A)$.

In the proofs which follow, we only consider the support of edges between nodes of the mesh, and do not examine the support of the ground connections. Recall, however, that by construction, the support tree leaves have the same ground connections as the mesh. Therefore, the support of the ground connections is unity.

4.6.1 Regular meshes in 1D

In this section, we assume that the mesh corresponding to the coefficient matrix A is a path on n points with unit weight edges. A might be the matrix from a finite difference discretization of the Dirichlet problem in the unit interval, for example. For this example, we obtain $\kappa(K^{-1}\tilde{A}) = O(n\log n)$. While this is a simple example, it will serve to illustrate the flavor of the proofs to follow. Figure 4.12 illustrates the mesh and tree for $n = 8$.

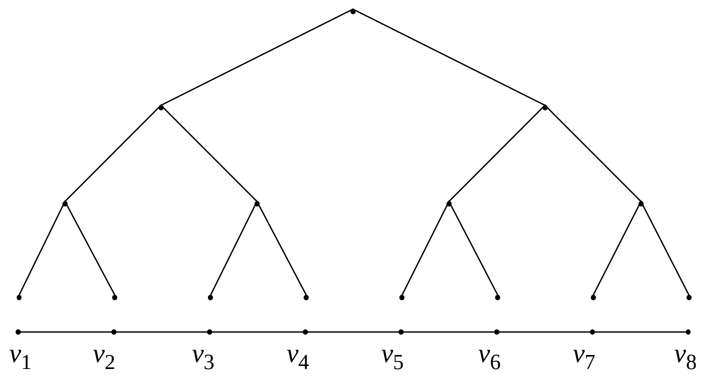


Figure 4.12: ***Support tree for a 1D mesh.***
The 1D mesh is at the bottom of the figure — a simple path on 8 points with unit edges. The support tree with unit edges is illustrated above the mesh. The leaves of the tree correspond to the nodes of the mesh.

In Chapter 3, we constructed support trees with edges weighted by the size of the frontier of the induced subgraph. In this section, such a support tree would have most of the edges of weight 2, instead of 1. This difference does not affect the asymptotic size of the condition number and simply complicates the analysis.

We are setting up the following methodology. First, to compute $\sigma(\tilde{A},T)$, we route all the wires of A along paths in T. This is fairly easy to do because of the way T is constructed. Next, to compute $\sigma(T,\tilde{A})$, we need to reduce T to K, and route all the wires of K through A and find the maximum support number; this gives us $\sigma(K,A)$, which, by Lemma 4.10, is an upper bound on $\lambda_{\max}(T, \tilde{A})$. To do this, we need an upper bound on the conductance values of the wires in K, which requires some analysis of the circuit reduction procedure.

Recall from §4.4 that, for graphs A and B of compatible sizes, and given a mapping of A into B, $\gamma_e(A,B)$ denotes the congestion of edge $e \in V_B$, and δ_f(A,B) denotes the dilation of edge $f \in V_A$. We further define

$$\gamma(A,B) = \max\{\gamma_e(A,B) : e \in V_B\}, \text{ and}$$

$$\delta(A,B) = \max\{\delta_f(A,B) : f \in V_A\}.$$

From section 4.4 we have that $\sigma(A, B) = \max\{\gamma_e(A, B) \cdot \delta_f(A, B)\} \le \gamma(A, B) \cdot \delta(A, B)$

4.12 ***Lemma***: *Let A be the Laplacian matrix of a 1d mesh with n vertices, and let T be the Laplacian matrix of the support tree with unit edges constructed for the mesh. Then* $\sigma(\tilde{A},T) = O(\log n)$.

proof:

Let $V = \{v_1,...,v_n\}$ be the nodes of A, and assume that they are ordered from left to right. Since the only edges of $\tilde{A}$ are edges of A, we only need to map A into T.

Any edge in T must support at most two edges of A. To see this, consider any node v_t of T. v_t implicitly defines a subgraph of A consisting of all the leaf nodes that have v_t as an ancestor; call this subgraph A_t. A_t has at most two boundary edges, so the edge from v_t to its parent must support at most two edges (since edges internal to A_t do not have to be mapped through the ancestor to v_t). But, the edges of T are weighted by the number of boundary edges in the induced subgraphs of A. Therefore, we can partition each edge in T into pieces of unit weight, each of which supports exactly one edge of A. Hence, the maximum congestion in T is 1, so $\gamma(A,T) = 1$.

The maximum dilation for any edge in A clearly comes from the edge that must be mapped through the root of T. The length of this path is $2\log_2 n$, so $\delta(A,T) = 2\log_2 n$.

Therefore, $\sigma(\tilde{A}, T) \le \gamma(\tilde{A}, T) \cdot \delta(\tilde{A}, T) = 1 \cdot 2\log_2 n = O(\log n)$.

■

In order to bound $\sigma(T,\tilde{A})$, we will compute $\sigma(K,A)$. To simplify this computation, we will consider sets of edges of K that all have the same conductance. We will then partition A so that each set of edges is supported by a separate partition. From Lemma 4.7, the maximum support number over all the partitions is an upper bound on $\sigma(K,A)$.

4.13 ***Lemma***: *Let T be a balanced binary tree with unit weight edges, n leaves, and height* $\log_2 n$. *Let K be the reduction of T to its leaves. Then the edges of K can be classified into* $\log_2 n$ *sets such that all the edges in a set have the same conductance.*

proof:

Since circuit reduction is equivalent to Gaussian elimination, the order in which nodes are reduced does not affect the final conductances. Consider reducing the tree from the root to the leaves one level at a time. Each reduction combines all pairs of input edges and creates a single edge for each pair. The output conductance is given by the product of the input conductances divided by the sum of the input conductances. Since we are reducing from the root down, every node has two edges of conductance 1, so the total conductance at an internal node is at least 2. Therefore, since all conductances are less than or equal to 1, the output conductance of a wire can be no greater than 1/2 times the smallest of the two input conductances.

We can state this observation another way. Let l_1 and l_2 be two leaves of the tree. Then, after reduction of all the internal nodes of the tree, the conductance between l_1 and l_2 will be bounded above by $1/2^m$, where m is the number of internal nodes on the path joining l_1 to l_2 in the original tree.

The root of the tree T divides the original graph A into two subgraphs A_0 and A_1. The conductances in K between nodes of A_0 and A_1 will all be the same, since T is perfectly balanced. Moreover, the conductances will be bounded above by $1/2^{2h-1}$, where h is the height of T.

Now, consider all the internal nodes $v_{l1},\ldots,v_{lm}$ at level l of T, where the root is at level 0, and $m = 2^l$. Each v_{lk} defines a subtree containing two disjoint subgraphs. As above, the conductances in K of all the wires between nodes in different subgraphs will be the same, and these conductances will be bounded above by $1/2^{2(h-l)-1}$.

Therefore, each set of internal nodes of the same height define a set of edges in K that all have the same conductances. There are $\log_2 n$ different heights, and the conductances are a function of the height of the subtrees.

■

4.14 ***Lemma***: *Let A and T be as defined above. Then* $\sigma(T,\tilde{A}) = O(n)$[1].

proof:

We reduce T to K and compute $\sigma(K,A)$. By Lemma 4.10, $\sigma(T,\tilde{A}) = \sigma(K,A)$.

Let $h = \log_2 n$. Let the root of T be at level 0, and the leaves at level h.

Using the results of the previous lemma, we partition K into subgraphs such that each subgraph contains edges of roughly the same conductance. The edge weights in T are all either 1 or 2. By assuming all edge weights are 2, we increase the total conductance of T, and make T harder to support. We therefore assume that all the edges in T are of weight 2; by computing the support number with uniform edge weights, we obtain an upper bound on the actual support number. Let K_i, $i = 1,...,h$ denote the subgraph made up of edges that cross subtree roots at level i; the edges of K_i have conductances bounded above by $2/2^{2(h-i)-1}$. We must therefore partition A similarly. Let $A_0, A_1,...,A_h$ be partitions of A constructed by splitting the edges of A such that all the edges of A_i, $i > 0$, have weight $1/2^i$. That is, $A_i = 1/2^i \cdot A$. A_0 contains the remaining conductance and will not be used further. Figure 4.13 illustrates the partitioning of K for the mesh and tree of Figure 4.12.

The mapping of K_i into A_i is the obvious one. Embed A_i in a line so that the nodes are ordered linearly from left to right. Embed K_i with the same ordering. Then an edge of K_i is mapped along the simple path in A_i that joins its endpoints.

Consider supporting the edges of K_i by the edges of A_i. There are $n/2^i$ nodes in each subgraph with a common ancestor at level i in T. (Equivalently, every internal node of T at level i is the root of a subtree with $n/2^i$ leaves and defines a subgraph of K_i with $n/2^i$ nodes.) There is one edge between every node in one subgraph to every node in the opposite subgraph of a pair; since $n = 2^h$, the total edge count per pair is

$$n^2/2^{2i} = 2^{2(h-i)} \tag{4.27}$$

A single edge of A_i only supports those edges of K_i which connect nodes that have a common ancestor at level i in the tree. Therefore,

$$\gamma(K_i, A_i) = 2^{2(h-i)} \tag{4.28}$$

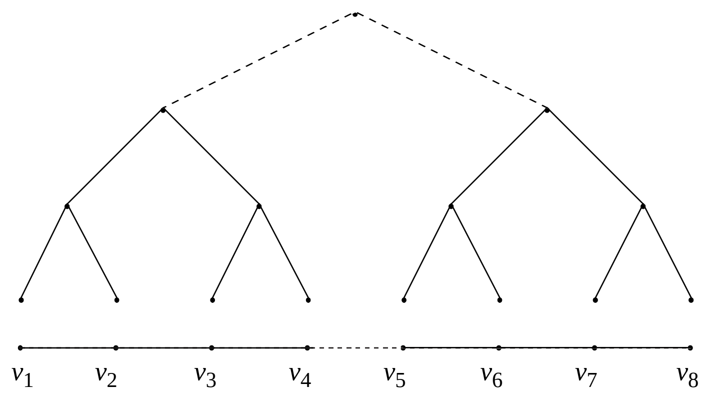
A_i is simply a linear path. So, by the construction of T, the edges of K_i have a dilation that is one less than the number of nodes in a connected subgraph. Therefore,

$$\delta(K_i, A_i) = 2(n/2^i) = 2^{h-i+1} \tag{4.29}$$

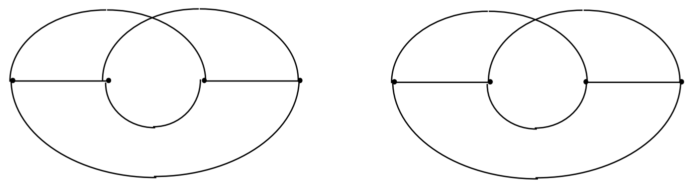
Now, $\sigma(A, B) = \gamma(A, B) \cdot \delta(A, B)$ when the edges of A and B are of unit conductance. In our case we must multiply the right hand side by the maximum conductance of the edges in A to determine the maximum conductance that must be supported by B. Similarly, the left hand side must therefore be multiplied by the minimum conductance in B. Therefore, the formula for the case of non-unit conductances is

1. In reading a preliminary version of this work, Dr. John Rief of Duke University found an error in the calculations and reduced the asymptotic bound to $O(n)$ instead of $O(n\log n)$.

a)



b)



c)

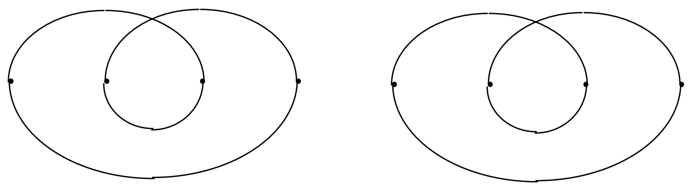


Figure 4.13: ***Partitioning of the reduced support tree.***
a) *Mesh and support tree. The subtrees rooted at level 1 induce subgraphs of the mesh.*
b) *Gaussian elimination of the subtrees yield complete graphs on the leaves.*
c) *K_1 consists of only the edges between nodes on opposite sides of the subtree root.*

$$\sigma(A, B) = \frac{\gamma(A, B) \cdot \delta(A, B) \cdot k_{max}(A)}{k_{min}(B)} \tag{4.30}$$

Applying (4.30) for $A = K_i$ and $B = A_i$, we obtain

$$\sigma(K_i, A_i) = \frac{2^{2(h-i)} \cdot 2^{h-i+1} \cdot 2^{-2(h-i)+2}}{1/2^i} = 2^{h+2} \tag{4.31}$$

This expression on the right of (4.31) is independent of *i*, so all the partitions have the same support. Therefore,

$$\sigma(K, A) = 2^{h+2} = 4n = O(n) \tag{4.32}$$

■

We now have a complete bound on the condition number.

4.15 ***Theorem***: *For A and T as defined,* $\kappa(T^{-1}\tilde{A}) = O(n\log n)$.

proof:

From Lemma 4.12, we have $\sigma(\tilde{A},T) = O(\log n)$. From Lemma 4.14, we have $\sigma(T,\tilde{A}) = O(n)$. Applying Lemma 4.8 yields the result.

■

4.6.2 Regular meshes in *d* dimensions (*d*>1)

Let A correspond to a cube in d dimensions, with $n=2^h$ nodes on a side. Then A contains $N = n^d = 2^{dh}$ total nodes. Assume that the edges of A have unit weight. The tree T is constructed by recursive coordinate bisection, and each edge of T is weighted by the number of edges in the frontier of the subgraph of A induced by the edge (see §3.2). T is $h = \log_2 n$ in height, and each internal node has 2^d children. The root of T is at level 0, and the leaves are at level h. We will show in this section that the condition number obtained by using T as a preconditioner for A is $O(d^2 n\log n)$

First, we show that $\sigma(\tilde{A},T)$, the support of T for $\tilde{A}$, is $O(\log n)$.

4.16 ***Lemma***: *Let A and T be as defined above. Then* $\sigma(\tilde{A},T) = O(\log n)$.

proof:

Any edge in T must support all the boundary edges of the induced subgraph of A. The edges of T are weighted by the number of boundary edges in the induced subgraphs of A. Therefore, we can partition each edge in T into pieces of unit weight, each of which supports exactly one edge of A. Hence, the maximum congestion in T is 1, so $\gamma(A,T) = 1$.

The maximum dilation for any edge in A clearly comes from an edge that must be mapped through the root of T. The length of this path is $2\log_2 n$, so $\delta(A,T) = 2\log_2 n$.

Therefore, $\sigma(\tilde{A}, T) \le \gamma(\tilde{A}, T) \cdot \delta(\tilde{A}, T) = 1 \cdot 2\log_2 n = O(\log n)$.

■

Let K be the complete graph formed by reducing the internal nodes of T. Next we show that $\sigma(K,A)$, the support of A for the reduced tree K, is $O(n)$. As in the case for $d=1$, we compute $\sigma(K,A)$ by mapping sets of edges of K into uniform partitions of A. The sets of edges of K are all those with subtree roots at the same level, and define subgraphs $K_0,\ldots,K_h$. K_i is the subgraph of K that contains only the edges between nodes in different components resulting from the ith partitioning step. Hence, if e is an edge in K_i connecting nodes u and v, then in T the highest internal node on the only path between u and v is at level i. K_i consists of 2^{di} components, each of which is a 2^d-partite graph.

To compute the support number, it is necessary to have an upper bound on the conductance of edges in K_i. Let v be an internal node of T at level i. Then v has 2^d children, but not all the children have the same boundary size, so the connecting edges have different weights. By assuming each edge has the weight of the maximum edge at level i, all the edges in K_i will have the same upper bound. Moreover, the conductance of K will only increase, making K harder to support. Therefore, the support number resulting from this assumption will be an upper bound on the true support number. We formalize this approach and determine the support number in the lemma below.

4.17 ***Lemma***: *Let A, T, and K be as defined above. Then* $\sigma(K, A) = O(d^2 n)$.

proof:

We prove this by performing conductance experiments. Let the subgraphs K_i be as defined above. We will show a support mapping such that the support for each K_i is independent of *i*. Consider a subtree of *T* rooted at node *u* at level *i*-1. Then, *u* has 2^d children. Each of the children of *u* is a node at level *i*. Each node at level *i* defines an induced subgraph of *A*, which is a cube characterized by the following parameters:

- the length of an edge is

$$e_i = n/2^i \tag{4.33}$$

- the size of a face is

$$f_i = e_i^{d-1} = (n/2^i)^{d-1} \tag{4.34}$$

- the size of the boundary is

$$\beta_i \le 2d f_i = 2d(n/2^i)^{d-1} \tag{4.35}$$

- the number of nodes in the subgraph is

$$n_i = e_i^d = (n/2^i)^d \tag{4.36}$$

- since the tree is boundary weighted, each edge between *u* and a child of *u* has weight $2d f_i$, since this is the maximum size of the frontier of any subgraph of *A* induced by a subtree of *T* rooted at level *i*.

Now we perform our electrical experiment. Apply the same voltage to every leaf node of every subtree rooted at a child of *u*, and let the other nodes in the tree be harmonic. Then the edges of each subtree can be combined from leaves to root so that the net effect of applying the same voltage to the leaves can be summed up as applying that voltage to a single conductor connected to *u*.

Figure 4.14 illustrates the electrical experiment for a binary tree. Leaf nodes l_1 and l_2 have a common parent v_1 and are each connected to v_1 by an edge of weight β_v. If the voltage applied to l_1 and l_2 is identical, then the effect is the same as if v_1 were connected to a single node l_{12} of the same voltage with a conductor of conductance 2β. v_1 is connected to its parent w_1 by an edge with conductance β_w. We now reduce *v* so that l_{12} is connected to w_1 by a single edge of conductance $\beta_w' = \dfrac{\beta_w \cdot 2\beta_v}{\beta_w + 2\beta_v}$, in accordance with the circuit reduction rules discussed in §4.4.

The edge weights in *T* reduce geometrically, so in the electrical experiment above, each subtree rooted at level *i* reduces to a single node connected to *u* by a conductor with conductance bounded above by $4d f_i$. (Doyle and Snell (1984) use this procedure to show that an infinite binary tree of unit resistors has a total root-to-leaf resistance of 1/2, or a total conductance of 2.)

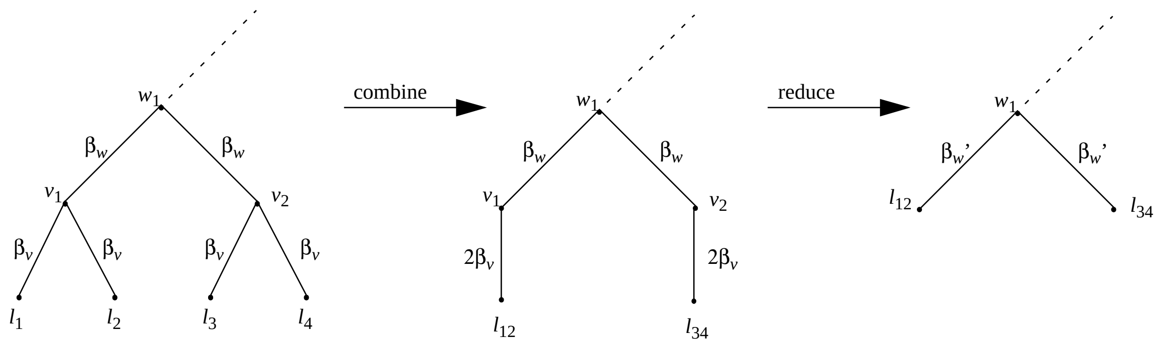


Figure 4.14: ***Reduction of a tree when leaves have the same voltage.***
A tree can be recursively reduced to a single edge when all leaves have the same voltage.

Thus, u effectively has 2^d children, each connected to u by a conductor of conductance $g_u \le 4df_i$. We are now interested in the effective conductance between each of these children of u. An upper bound on this conductance can be obtained by reducing u, while ignoring the connection between u and its parent (this connection can only decrease the effective conductance). Let g_i denote this upper bound, then g_i can be computed as in (4.37), below.

$$g_i = \frac{4df_i \cdot 4df_i}{2^d \cdot 4df_i} = \frac{4df_i}{2^d} \tag{4.37}$$

Now, each edge of conductance g represents the total conductance in all the edges in the bipartite subgraph of K_i that joins all the leaf nodes in two of the subtrees rooted at children of u at level i-1 in T. Each such bipartite subgraph contains n_i^2 identical edges. Each of these edges has conductance k_i given by (4.38).

$$k_i = \left(\frac{4df_i}{2^d}\right)/n_i^2 = \frac{4df_i}{2^d n_i^2} \tag{4.38}$$

Therefore, k_i is an upper bound on the conductance of every edge in K_i.

Now we need to look at supporting each edge of K_i by a path in A. Let $A_0,...,A_{h-1}$ be partitions of A obtained by dividing each edge of A into h=$\log_2 n$ pieces, with the pieces in A_i each conductance $1/2^i$. We embed K_i in A_i as follows. Let (u,w) be an edge of K_i, where u and w are nodes of the mesh with coordinates $(u_1,...,u_d)$ and $(w_1,...,w_d)$ respectively. We map (u,w) to a path in A_i by reducing the difference in each dimension in order. That is, we map (u,w) first to $v_1 = (w_1,u_2,...,u_d)$, then $v_2 = (w_1,w_2,u_3,...,u_d)$ and so on. This is not an optimal mapping, in that the boundary edges that cross the first dimension are more heavily congested than edges on other boundaries. However, it is sufficient for this proof. This is a standard construction, and additional details can be found in many books on algorithms, such as Leighton (1992).

Every edge in K_i arises from the reduction of a subtree of T rooted at some node v at level i. Let w be some child of v and $A_i(w)$ the induced subgraph of A_i. In the worst case, a boundary edge of $A_i(w)$ with must support an edge from every node in $A_i(w)$ with the same first coordinate to every edge in the half space of $A_i(v)$ on the opposite side of the first coordinate partition from $A_i(w)$. There are $e_i/2$ nodes that have the same first coordinate, and $n_{i-1}/2 = (2^d n_i)/2$ in the half space. Therefore, since this is the worst case,

$$\gamma(K_i, A_i) = \frac{2^d n_i e_i}{4} \tag{4.39}$$

The longest path in $A_i(v)$ is from some corner to the diagonally opposite corner. Given the mapping defined above, the length of this path is de_i. Therefore,

$$\delta(K_i, A_i) = de_i \tag{4.40}$$

Again, as in the d=1 case, we need to weight the congestion by the maximum conductance of the edges being mapped, and divide the product by the conductance of the edges in A_i, which is $1/2^i$. Taking (4.38), (4.39), and (4.40), this yields

$$\sigma(K_i, A_i) = \frac{\left(\frac{2^d n_i e_i}{4}\right) \cdot (de_i) \cdot \left(\frac{4df_i}{2^d n_i^2}\right)}{1/2^i} = \frac{d^2 f_i e_i^2 2^i}{n_i} = \frac{d^2 (n/2^i)^{d+1} 2^i}{(n/2^i)^d} = d^2 n \tag{4.41}$$

Applying Lemma 4.7 yields $\sigma(K, A) = \max_i \sigma(K_i, A_i) = O(d^2 n)$ for constant d.

■

With both support numbers determined, we can now bound the generalized condition number.

4.18 ***Theorem***: *For A, T, and K as defined,* $\kappa(K^{-1}A) = O(d^2 n \log n)$

proof:

From Lemma 4.12, we have

$$\sigma(A,K) = \sigma(\tilde{A},T) = O(\log n). \tag{4.42}$$

From Lemma 4.17, we have

$$\sigma(K, A) = O(d^2 n). \tag{4.43}$$

(4.42) and (4.43), together with Lemma 4.8, yield

$$\kappa(K^{-1}A) = \sigma(A, K) \cdot \sigma(K, A) = O(\log n) \cdot O(d^2 n) = O(d^2 n \log n)$$

■

4.6.3 Extensions to the regular case

In the preceding portion of this subsection, we have showed that, for regular rectangular meshes in d dimensions with $n = 2^k$ nodes on a side, the condition number obtained with a support tree preconditioner is $O(n \log n)$. The restriction that $n = 2^k$ was for ease of proof. Examination of the proof shows that the important conditions are that the depth of the tree be $O(\log n)$ and that the size of the boundary of each set be within a constant factor of the size of the separator of the set. Both of these conditions hold for arbitrary n.

Similarly, increasing the connectivity by adding diagonal edges does not affect the asymptotic size of the condition number. Increasing connectivity by a constant factor only increases the sizes of boundaries and separators by a constant factor, which does not affect the asymptotic bound on the condition number. On the other hand, the edge weights in the mesh must be bounded by a constant for the proofs to hold.

4.7 Summary

In this chapter, we demonstrated an isomorphism between Laplacian matrices, undirected edge-weighted graphs, and resistive networks. We used this isomorphism to prove that the convergence rate of STCG for a regular mesh in *d* dimensions was $O(\sqrt{n \log n})$, where *n* is the number of nodes on a side of the mesh. While an interesting result, a more general result, applicable to a wider variety of meshes, is desirable. In certain special cases, the results of this chapter can be generalized, but the proofs are quite intricate.

In the next chapter, we show how the results can be generalized to a wider variety of graphs using results from multi-commodity flow. The generalization comes at the cost of an additional factor of $\log^3 n$. That is, using multicommodity flow, we are able to prove that STCG converges as $O(\sqrt{n \log^4 n})$ for regular meshes. For more complex graphs, the convergence depends on a quantity known as the *flux* of the graph, which is a measure of the graph connectivity.

5
Extended Analysis

This chapter extends the analytical results of the previous chapter to more general cases. The work reported here was done jointly with Gary Miller and Bruce Maggs.

The analysis of the previous chapter is similar to analysis presented elsewhere for other techniques. For example, the textbooks of Hageman and Young (1981) and Hackbusch (1994) use the 2D regular rectangular mesh as the model grid for presentation of general iterative methods. In the research domain, Greenbaum, *et al* (1989) also used the 2D regular rectangular mesh in their comparative study of various CG methods. However, application problems rarely exhibit the kind of regular structure associated with the model grids of the previous chapter.

Moreover, several assumptions were implicit in the model grids of the previous chapter. Some assumptions are easily dealt with by straightforward modifications of the proofs. The assumptions that cannot be handled by modifications of previous results are the following:

1. irregular graph structure

   Despite the utility of the rectangular mesh as a model problem, many application problems are defined on irregular grids. Convergence proofs for support trees require embedding the reduced tree into the mesh. For a regular mesh, this is difficult but possible to do explicitly. For an irregular mesh, explicit mapping is so difficult as to be effectively impossible. An approach that permits implicit mapping is required.

2. unequal partitions

   In the earlier proofs of convergence rate, we relied on the fact that the support tree had depth $\log n$. This depth ensured that the tree could support the mesh with a $\log n$ gain factor, and that the conductance of any reduced tree edges was bounded above by $(1/2)^{2i}$, where i was the depth of the root of the subtree being considered. However, these bounds imply that all partitioning steps produce subgraphs that are approximately equal in size. Not all separators produce equal-sized subgraphs. For example, the sphere separator of Miller, *et al* (1992) may produce subgraphs of sizes $n/3$ and $2n/3$, where n is the number of nodes in the original graph. The Leighton-Rao separator [Leighton and Rao (1988)] has no guarantee on the relative size of the resultant subgraphs; the separator simply minimizes the ratio of boundary edges to internal nodes of the smallest subgraph.

Rao (1987) defined a *k-ratio edge separator* as a binary edge separator for which the ratio of subgraph sizes is (k-1)/k. From the discussion above, a 3-ratio separator is sufficient to make the convergence analysis of the previous chapter fail (the conductance of the edges in the reduced tree could be too large). Many separators (Leighton-Rao, for example) cannot even guarantee a bounded k for every step of the construction of a support tree.

3. unbounded frontier sizes

A close look at the convergence analysis of the previous chapter shows that a key factor in determining the support of the mesh for the tree is the ratio of the size of the frontier of a subgraph to the size of the subgraph's own separator. Recall that the frontier of a subgraph is the set of edges with exactly one endpoint in the subgraph. Intuitively, the importance of this ratio can be justified. The size of the separator is a measure of the connectivity of a subgraph, and a subgraph must be sufficiently well connected to support paths between all of the frontier edges.

The separators used in the convergence analysis were nice in the sense that they always produced subgraphs for which the frontier/separator ratio was bounded by a constant. For example, for a rectangular mesh in d dimensions, the frontier/separator ratio is bounded above by 2d. In general, for arbitrary graphs and arbitrary separators, no such bound may be possible.

In the subsections that follow, these problems will be addressed.

5.1 Irregular Graphs and Implicit Embedding

The solution to dealing with the problems of irregular graphs and explicit embeddings comes from the area of combinatorial optimization — specifically from results in multicommodity flow. A brief review of multicommodity flow was presented in §2.2.2. Recall the definition of the flux of a graph $G = (V,E)$, denoted by $\alpha = \alpha(G)$:

$$\alpha = \min_{S \subseteq V} \frac{\sum_{x \in S,\, y \in \bar{S}} c(x, y)}{\min(|S| \cdot |\bar{S}|)} \tag{5.1}$$

The flux is a measure of the connectivity of a graph. If G is a graph with n nodes and flux α, and S is any subset of G with $|S| \le n/2$, then the frontier of S has at least $\alpha|S|$ edges; that is, S is connected to the rest of G by at least $\alpha|S|$ edges.

Now, suppose that H and G are both graphs with n nodes and that we wish to embed H into G. Recall from §2.1, that embedding H into G means that for every edge in H, we must specify a path in G. Any edge in G may lie on one or more paths of the embedding, and the number of paths that include an edge e is the *congestion* of e. The *congestion* of the embedding is the maximum congestion of any edge in G. The *dilation* of the embedding is the maximum length of any path of the embedding.

Intuitively, the congestion and dilation of an embedding is related to the flux, $\alpha(G)$, of the target graph G. Let G_1 and G_2 be graphs with $|G_1| = |G_2| = n$, and $\alpha(G_1) < \alpha(G_2)$. Consider the embeddings of another n-node graph H into each of G_1 and G_2. Any subset S of nodes in H corresponds to subsets S_1 and S_2 in G_1 and G_2 respectively. Since $\alpha(G_1)<\alpha(G_2)$, there are fewer edges out of S_1 than out of S_2, so the congestion of the embedding into G_1 is larger than the congestion of the embedding into G_2. Similarly, let v be a vertex in H with corresponding nodes v_1 and v_2 in G_1 and G_2, respectively. Because $\alpha(G_1) < \alpha(G_2)$, the number of nodes of G_1 within edge radius r of v_1 is less than the number of nodes of G_2 within edge radius r of v_2. Thus, it takes longer paths in G_1 to reach all nodes from v_1 than it does in G_2 to reach all nodes from v_2. Therefore, the dilation of the embedding of H is larger in G_1 than in G_2.

Leighton and Rao (1988) bounded the congestion and dilation of embeddings with respect to the flux of the target graph. We repeat here the key result, Theorem 2.5 of §2.2.2.

2.5 ***Theorem***: (Leighton-Rao) *Consider any n-node bounded degree graph H, and any 1-1 embedding of the nodes of H onto the nodes of an n-node bounded degree graph G with flux* α. *The edges of H can be routed as paths in G with congestion and dilation* $O(\frac{\log n}{\alpha})$.

Equation 4.26 bounds support using congestion and dilation. Theorem 2.5 supplies us with bounds on congestion and dilation, *without explicitly mapping edges onto paths*! This implicit mapping comes at the cost of a complexity factor of $\log^2 n$, as we show below. We can now prove the following:

5.1 **Theorem**: *Let G = G(A) be a graph in d dimensions with unit edges corresponding to a Laplacian matrix A. Let T be a boundary-weighted support tree constructed for G by some process of recursive bisection. Suppose that there exists a constant k such that, for any subgraph* G_i *of G constructed during the partitioning process with* G_{i0} *and* G_{i1} *obtained by partitioning* G_i*, the following conditions hold:*

i. $\alpha(G_{i0}), \alpha(G_{i1}) \geq \alpha(G_i)$;

ii. $k \cdot \alpha(G_i) \geq \beta(G_i)$, *where* $\beta(G_i)$ *denotes the weight of edges on the boundary of* G_i.

Then $\kappa(A, S) = O(\frac{\log^4 n}{\alpha(G)})$, *where S is the Laplacian matrix obtained by reducing the interior nodes of T.*

proof:

$\sigma(A, S)$: Recall from the previous chapter that we determine this quantity by mapping each edge of G onto a path in T and examining the support. Since T is boundary weighted, each edge of T can be partitioned to provide unit support for each edge of G. The maximum length path in T has length $2\lceil \log n \rceil + 1$, so the conductance of the path is $O(\log n)$. Hence, $\sigma(A, S) = O(\log n)$

$\sigma(S, A)$: Let K be the reduction (Schur complement) of T obtained by applying Gaussian elimination to T, eliminating all the internal nodes and stopping at the leaves. Then S is the Laplacian matrix corresponding to K.

T is a support tree of depth $h = \lceil \log n \rceil$, where the root is at level 0, and the leaves are at levels h-1 and h. As before, we partition K, the reduced tree, into subgraphs $K_0,...,K_{h-1}$ such that K_i consists of all the edges in K between nodes u and w such that the highest node on the path in T from u to w is at level i. Similarly, we partition G into $G_0,...,G_{h-1}$ by dividing each edge of G into $\log n$ pieces, each of equal conductance. We will map K into G by mapping each K_i into the corresponding G_i.

Let v be an internal node of T at level i. Let G_v be the subgraph of G induced by v. There are $n_v \leq n \cdot (1/2)^i$ nodes in G_v, and hence in K_v, the subgraph of K_i induced by G_v. The conductances of the edges of K_v are bounded above by $\beta(G_v) \cdot (1/2)^{h-i} \leq k\alpha(G_v) \cdot (1/2)^{h-i}$. Consider embedding K_v into G_v.

By the Leighton-Rao Theorem (Theorem 2.5), the dilation of the embedding is $O(\log n_v / \alpha(G_v))$. Since we are embedding the complete graph on n_v points, rather than a bounded-degree graph, the congestion of the embedding is $O(n_v \cdot \log n_v / \alpha(G_v))$.

We now have that

$$\sigma(S, A) \sim k\alpha(G_v) \cdot (1/2)^{h-i} \cdot \frac{\log n_v}{\alpha(G_v)} \cdot \frac{n_v \cdot \log n_v}{\alpha(G_v)} \cdot \log n$$

$$\le k \cdot (1/2)^{h-i} \cdot \frac{\log^3 n}{\alpha(G_v)} \cdot n \cdot (1/2)^i$$

$$= O(\frac{\log^3 n}{\alpha(G)})$$

Therefore,

$$\kappa(A, S) = \sigma(A, S) \cdot \sigma(S, A) = O(\frac{\log^4 n}{\alpha(G)})$$

■

This immediately yields the following.

5.2 Corollary: *Suppose A is a Laplacian matrix with corresponding graph G that is a regular rectangular mesh in d dimensions with n nodes on a side. Suppose T is a support tree for G constructed as in* 5.1 *with Laplacian matrix S corresponding to the graph obtained by reducing the interior nodes of T. Then,*

$$\kappa(A, S) = O(n\log^4 n).$$

proof:

It is easy to verify that G satisfies the conditions of Theorem 5.1. Furthermore, $\alpha(G) = O(1/n)$. Therefore,

$$\kappa(A, S) = O(\frac{\log^4 n}{\alpha(G)}) = O(\frac{\log^4 n}{1/n}) = O(n\log^4 n)$$

■

Compare Corollary 5.2 with Theorem 4.19. Using Leighton-Rao, we were able to avoid explicitly embedding graphs, yet obtained nearly the same result. Implicit embedding with Leighton-Rao essentially incurs a cost of $O(\log^3 n)$, which is a small price to pay for the additional simplicity and generality. Theorem 5.1 applies to a much larger class of graphs than the rectangular grids considered in 4.19.

Theorem 5.1 addresses the problem of irregular graphs, but does not completely solve it. The requirement for a highly regular graph structure has been replaced by a looser requirement that the flux of the subgraphs is monotonically increasing with the partitioning process. For regular graphs such as rectangular meshes, this requirement can be met. However, another requirement is that the partition is a bisection, and it is possible to construct a graph for which bisection yields decreasing flux. Figure 5.1 illustrates a planar graph for which a bisection yields decreasing flux.

Finding a separator that will guarantee increasing flux is a difficult problem. Leighton and Rao (1988) proposed a *flux cut* that would partition a graph using the separator that defined the flux. This would seem to be a good candidate for a partitioning algorithm that would guarantee increasing flux. However, even a series of flux cuts does not provide such a guarantee. The graph of Figure 5.1 provides a counterexample. The bisection illustrated is a flux cut, yet the flux decreases after the cut.

For most graphs with a moderately homogeneous structure, the flux will generally increase with partitioning. Thus, while we can deal with a larger class of graphs at this point, we still lack the analytic tools to handle all graphs.

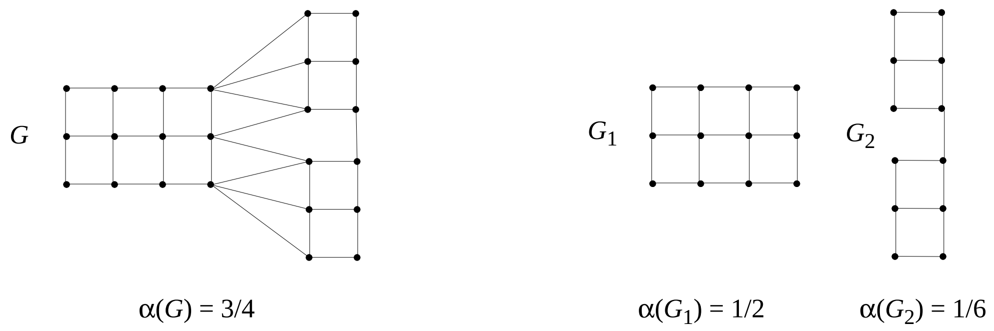


Figure 5.1: ***A bisection that yields decreasing flux.***

5.2 Dealing with Unequal Partitions

Another unrealistic assumption that we have been making is that the partitioning process yields subgraphs of roughly equal size. However, many of the best performing graph partitioning algorithms do not conform to this requirement. The problem with unequal partitions is that some subtree may be relatively shallow, and yield conductances in the Schur complement that are too large for the original graph to support.

To be able to build support trees using real separators, we need to be able to weight the tree in such a way as to yield a common upper bound on the conductances in the reduced graph. To solve this problem, we work backwards from the goal. Consider weighting a tree such that, when interior nodes are reduced starting with the parents of the leaves and working upwards, all the conductances from the leaves to the lowest unreduced ancestor are proportional to the number of leaves in the subtree. Figure 5.2 illustrates this property.

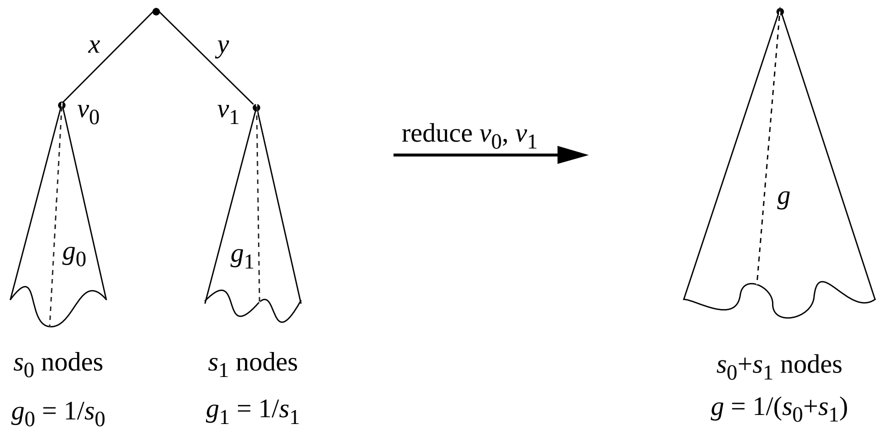


Figure 5.2: ***Reducing a tree for which leaf-to-subtree-root conductances are proportional to the number of nodes in the subtree***

Referring to Figure 5.2, we need to determine the values of x and y that will yield the desired conductances when v_0 and v_1 are reduced. Node v_0 has s_0 connections of conductance g_0, and 1 connection of conductance x. Therefore, applying Gaussian reduction to v_0 yields s_0 connections to the root of conductance g, given by

$$g = \frac{g_0 \cdot x}{s_0 \cdot g_0 + x} = \frac{x}{s_0 + s_0 \cdot x} \tag{5.2}$$

where the last step follow from the fact that $g_0 = 1/s_0$.

Now, setting the value for g obtained above (by reducing v_0) equal to the desired value in Figure 5.2, and solving for x yields

$$x = \frac{s_0}{s_1} \tag{5.3}$$

A similar derivation yields

$$y = \frac{s_1}{s_0} \tag{5.4}$$

That is, by weighting the tree edges by the ratio of the sizes of the corresponding subtrees, the root-to-leaf conductances are all equal and given by the reciprocal of the number of nodes in the subtree. Now we can determine conductances in the reduced tree independently of the shape of the tree.

Unfortunately, the derivation above solves one problem, but causes another. We now can easily compute conductances in the reduced tree, but have lost the capability to support the mesh with the tree. To support the mesh, the edges of the tree must be at least as large as the weight of the boundary edges in the induced subgraph of the mesh. Weighting with x and y as defined in (5.3) and (5.4) above means that at least one branch of the tree will have an edge of weight less than one, which is insufficient to support the induced subgraph. Equivalently, the leaf-to-root conductances, while identical, are too small to support the mesh.

However, the general idea has merit, and can be extended to provide the answer we need. To do so, we need to define a new separator.

5.3 ***Definition***: *Let $G = (V,E)$ be a graph. For any subgraph H induced by a set of nodes $S \subseteq V$, let $\beta(H)$ denote the weight of the edges on the boundary of H. Suppose $\mathcal{E}$ is an edge separator that partitions G into subgraphs G_0 and G_1. Then the* boundary ratio of $\mathcal{E}$, $\tau(\mathcal{E}) = \tau(G_0,G_1)$ *is given by*

$$\tau(\mathcal{E}) = \max\left\{\frac{\beta(G_0)}{|G_0|}, \frac{\beta(G_1)}{|G_1|}\right\} \tag{5.5}$$

5.4 ***Definition***: *Let $G = (V,E)$ be a graph. An* optimal boundary ratio separator for *G is an edge separator $\mathcal{E}$ that minimizes $\tau(\mathcal{E})$. That is, if $\mathcal{E}$ is any edge separator of G, then*

$$\tau(\mathcal{E}) = \min_{\mathcal{E}: G_0 \cup G_1 = G}\left\{\max\left\{\frac{\beta(G_0)}{|G_0|}, \frac{\beta(G_1)}{|G_1|}\right\}\right\}. \tag{5.6}$$

Since the value of the optimal boundary ratio separator is unique, we can unambiguously refer to the value without reference to the separator, and denote the value by $\tau(G)$.

Now, consider constructing and weighting a support tree using optimal boundary ratio separators. We want the conductances in the reduced tree to be large enough to support the mesh, which means that the conductances of the edges in a subtree T_0 with s_0 nodes should be at least $\beta(T_0)/s_0$. We achieve this if the conductances are given by $\tau(G_0)$, where G_0 is the subgraph induced by T_0.

Now, what should the edge weights be in the tree? The situation is illustrated in Figure 5.3.

Following the same procedure as before, reducing v_0 and v_1 to find the values of x and y yields:

$$x = \frac{s_0 \tau_0 \tau}{\tau_0 - \tau} \tag{5.7}$$

$$y = \frac{s_1 \tau_1 \tau}{\tau_1 - \tau} \tag{5.8}$$

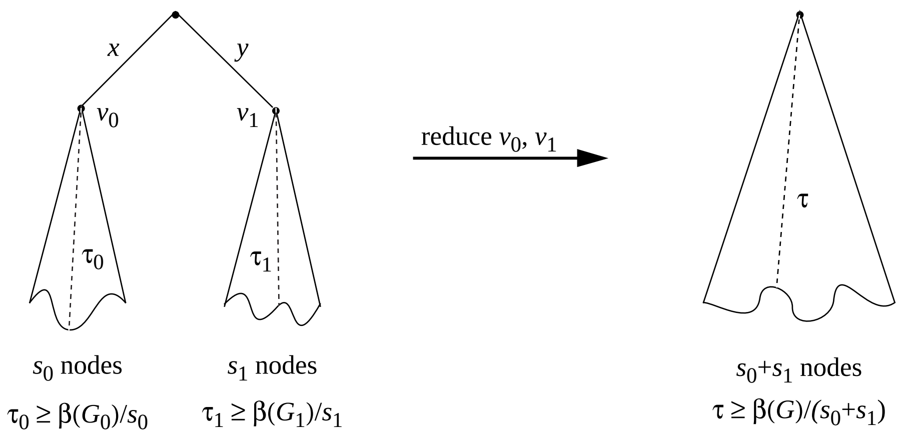


Figure 5.3: ***Reducing a tree for which leaf-to-subtree-root conductances are proportional to the boundary ratio of the subtree***

Note first that the values for x and y are dependent only on information from the induced subgraph. This means that the formulas (5.7) and (5.8) can be applied when more than two subgraphs result from a partitioning. Therefore, we are no longer limited to binary partitions. However, the values for x and y in (5.7) only make sense as edge weights if both x>0, and y>0. This implies that we must have both τ_0, $\tau_1 > \tau$. That is, τ must be monotonically increasing with optimal boundary ratio cuts. That this is true is shown in the next lemma. First, we need a basic proposition.

5.5 ***Proposition***: *If* $\frac{a}{b} > \frac{c}{d}$, *and a, b, c, d are all positive, then* $\frac{a}{b} > \frac{a+c}{b+d}$.

proof:

$$\frac{a}{b} > \frac{c}{d} \Rightarrow ad > bc \Rightarrow ab + ad > ab + bc \Rightarrow a(b+d) > b(a+c) \Rightarrow \frac{a}{b} > \frac{a+c}{b+d}$$

■

5.6 ***Lemma***: *Let G = (V,E) be a graph, and* $\tau(\mathcal{E})$ *be the boundary ratio of an edge separator* $\mathcal{E}$. *Suppose that G is recursively partitioned using optimal boundary ratio separators. Then* τ *is monotonically increasing with each partitioning. That is, if* G_i *is a subgraph produced at some stage of the partitioning process, and* G_i *is partitioned into subgraphs* G_{i0} *and* G_{i1}, *then* $\tau(G_i) < \tau(G_{i0})$ *and* $\tau(G_i) < \tau(G_{i1})$.

proof:

Let H be a subgraph produced at some stage of recursive partitioning, and consider the partitioning of H with an optimal boundary ratio separator. Referring to Figure 5.4, H has a boundary made up of boundary segments of weight a and b, and is partitioned by an optimal boundary ratio separator of weight e_0 into subgraphs A and B. Thus,

$$\tau(H) = \max\left\{\frac{\beta(A)}{|A|}, \frac{\beta(B)}{|B|}\right\} = \max\left\{\frac{a+e_0}{|A|}, \frac{b+e_0}{|B|}\right\}$$

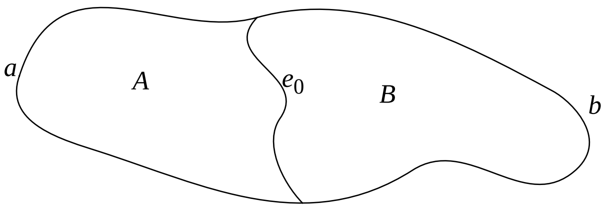


Figure 5.4: ***Partitioning of subgraph H with an optimal boundary ratio separator into subgraph A and B. a, b, and e0 are the weights of boundary segments.***

Now, suppose that one of the two subgraphs is recursively partitioned with an optimal boundary ratio separator. Without loss of generality, assume A is partitioned into subgraphs C and D. Referring to Figure 5.5 for the definition of the terms, we have:

$$\tau(A) = \max\left\{\frac{\beta(C)}{|C|}, \frac{\beta(D)}{|D|}\right\} = \max\left\{\frac{c+e_1+e_3}{|C|}, \frac{d+e_2+e_3}{|D|}\right\}$$

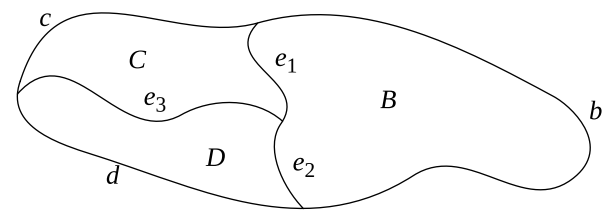


Figure 5.5: ***Partitioning of subgraph A from* Figure 5.4 *with an optimal boundary ratio separator.***

To complete the proof, we need to show that $\tau(A) \geq \tau(H)$. We consider four cases.

1. $\tau(H) = \dfrac{\beta(A)}{|A|} = \dfrac{c+d+e_1+e_2}{|C|+|D|}$ and $\tau(A) = \dfrac{\beta(D)}{|D|} = \dfrac{d+e_2+e_3}{|D|}$

   By the definition of the case being considered,

$$\tau(A) = \frac{d+e_2+e_3}{|D|} \geq \frac{c+e_1+e_3}{|C|} = \frac{\beta(C)}{|C|} .$$

   Applying 5.5, we obtain

$$\tau(A) = \frac{d+e_2+e_3}{|D|} \geq \frac{c+d+e_1+e_2+2e_3}{|C|+|D|} \geq \frac{c+d+e_1+e_2}{|C|+|D|} = \tau(H)$$

2. $\tau(H) = \frac{\beta(A)}{|A|} = \frac{c + d + e_1 + e_2}{|C| + |D|}$ and $\tau(A) = \frac{\beta(C)}{|C|} = \frac{c + e_1 + e_3}{|C|}$

By the definition of the case being considered,

$$\tau(A) = \frac{c + e_1 + e_3}{|C|} \geq \frac{d + e_2 + e_3}{|D|} = \frac{\beta(D)}{|D|} .$$

Applying 5.5, we again obtain

$$\tau(A) = \frac{c + e_1 + e_3}{|C|} \geq \frac{c + d + e_1 + e_2 + 2e_3}{|C| + |D|} \geq \frac{c + d + e_1 + e_2}{|C| + |D|} = \tau(H)$$

3. $\tau(H) = \frac{\beta(B)}{|B|} = \frac{b + e_1 + e_2}{|B|}$ and $\tau(A) = \frac{\beta(D)}{|D|} = \frac{d + e_2 + e_3}{|D|}$

We prove this by contradiction. Suppose that $\tau(H) > \tau(A)$. Then,

$$\tau(H) = \frac{b + e_1 + e_2}{|B|} > \frac{d + e_2 + e_3}{|D|} \geq \frac{c + e_1 + e_3}{|C|} .$$

Applying 5.5 using the first and third fractions above, we obtain

$$\tau(H) = \frac{b + e_1 + e_2}{|B|} > \frac{b + c + 2e_1 + e_2 + e_3}{|B| + |C|} \geq \frac{b + c + e_2 + e_3}{|B| + |C|} = \frac{\beta(B \cup C)}{|B \cup C|} .$$

From this inequality and the initial conditions defining this case, we have that

$$\tau(H) > \max\left\{\frac{\beta(D)}{|D|}, \frac{\beta(B \cup C)}{|B \cup C|}\right\}$$

That is, partitioning *D* from *B*∪*C* yields a smaller boundary ratio that partitioning *A*=*C*∪*D* from *B*. This contradicts the optimality of the (*A*,*C*∪*D*) partition. Therefore, we must have $\tau(H) \leq \tau(A)$.

4. $\tau(H) = \frac{\beta(B)}{|B|} = \frac{b + e_1 + e_2}{|B|}$ and $\tau(A) = \frac{\beta(C)}{|C|} = \frac{c + e_1 + e_3}{|C|}$

We again prove this by contradiction. Suppose that $\tau(H) > \tau(A)$. Then,

$$\tau(H) = \frac{b + e_1 + e_2}{|B|} > \frac{c + e_1 + e_3}{|C|} \geq \frac{d + e_2 + e_3}{|D|} .$$

Applying 5.5 using the first and third fractions above, we obtain

$$\tau(H) = \frac{b + e_1 + e_2}{|B|} > \frac{b + d + e_1 + 2e_2 + e_3}{|B| + |D|} \geq \frac{b + d + e_1 + e_3}{|B| + |D|} = \frac{\beta(B \cup D)}{|B \cup D|} .$$

From this inequality and the initial conditions defining this case, we have that

$$\tau(H) > \max\left\{\frac{\beta(C)}{|C|}, \frac{\beta(B \cup D)}{|B \cup D|}\right\}$$

That is, partitioning *C* from *B*∪*D* yields a smaller boundary ratio that partitioning *A*=*C*∪*D* from *B*. This contradicts the optimality of the (*A*,*C*∪*D*) partition. Therefore, we must have $\tau(H) \leq \tau(A)$.

The four cases above prove the lemma for the partitioning illustrated in Figure 5.4 and Figure

5.5. These cases and the associated figures also represent the general case. For any other partitioning of H and A into two pieces, no other boundary segments are added. Instead, some segments may go to zero. However, the conditions required by Proposition 5.5 still hold, and the lemma follows.

■

Now, we build support trees using recursive optimal boundary ratio partitioning using a two-phase process. In the first phase, the partitioning is performed, and the tree structure is built up. In the second phase, the edge weights of the support tree are computed level-by-level from leaves to roots: edges that connect leaf nodes are weighted with the value of the optimal boundary ratio separator that yielded the leaf; internal edges are weighted using (5.7).

We can now prove the following theorem, which is similar to Theorem 5.1, but relaxes two restrictions: that the partitioning is binary, and that the subgraphs formed by partitioning are equal in size.

5.7 ***Theorem***: *Let $G = G(A)$ be a graph in d dimensions with unit edges corresponding to a Laplacian matrix A. Let T be a boundary-weighted support tree constructed for G by recursive partitioning using optimal boundary ratio separators, and weighted as discussed above. Suppose that there exists a constant k such that, for any subgraph G_i of G constructed during the partitioning process with G_{i0} and G_{i1} obtained by partitioning G_i, the following condition holds*: $\alpha(G_{i0}), \alpha(G_{i1}) \geq \alpha(G_i)$.

Then $\kappa(A, S) = O(\frac{\log^4 n}{\alpha(G)})$, *where S is the Laplacian matrix obtained by reducing the interior nodes of T.*

proof:

$\sigma(A, S)$: Recall from Theorem 5.1 that we determine this quantity by mapping each edge of G onto a path in T and examining the support. In Theorem 5.1, since T was boundary weighted, each edge of T was partitioned to provide unit support for each edge of G. All we need to do in this case is to show that the support of each edge in T for each edge in G is $O(1)$.

Consider an edge e at level i in T, which connects nodes u and v, where u is the node on the root side of e. Let the subgraphs of G induced by u and v be denoted U and V, with optimal boundary ratios τ_{i-1}, and τ_i, respectively. Let $|V|$ denote the number of nodes in V. Let w be the weight of edge e. Then w is given by

$$w = \frac{|V|\tau_i\tau_{i-1}}{\tau_i - \tau_{i-1}}$$

We need only show that w is at least as large as $\beta(V)$, the size of the boundary of V. Let $V_1,...,V_k$ be the other subgraphs of U produced by the partitioning process. Then we have the following:

$$w = \frac{|V|\tau_i\tau_{i-1}}{\tau_i - \tau_{i-1}} > \frac{|V|\tau_i\tau_{i-1}}{\tau_i} = |V|\tau_{i-1} = |V| \cdot \max\left\{\frac{\beta(V)}{|V|}, \frac{\beta(V_1)}{|V_1|}, \ldots, \frac{\beta(V_k)}{|V_k|}\right\} \geq |V| \cdot \frac{\beta(V)}{|V|} = \beta(V)$$

That is, w is larger than the size of the boundary of V.

Since w is larger than the boundary size and V was chosen arbitrarily, each edge of T can provide unit support to every boundary edge in its induced subgraph. The maximum length path in T has length $2\lceil \log n \rceil + 1$, so the conductance of the path is $O(\log n)$. Hence, $\sigma(A, S) = O(\log n)$

$\sigma(S, A)$: As before, let K be the reduction (Schur complement) of T obtained by applying Gaussian elimination to T, eliminating all the internal nodes and stopping at the leaves. S is the Laplacian matrix corresponding to K.

T is a support tree of depth $h = \lceil \log n \rceil$, where the root is at level 0, and the leaves are at levels h-1 and h. As before, we partition K, the reduced tree, into subgraphs $K_0,\ldots,K_{h-1}$ such that K_i consists of all the edges in K between nodes u and w such that the highest node on the path in T from u to w is at level i. Similarly, we partition G into $G_0,\ldots,G_{h-1}$ by dividing each edge of G into log$n$ pieces, each of equal conductance. We will map K into G by mapping each K_i into the corresponding G_i.

Let v be an internal node of T at level i. Let G_v be the subgraph of G induced by v. There are $n_v \le n \cdot (1/2)^i$ nodes in G_v, and hence in K_v, the subgraph of K_i induced by G_v. By the way in which T was constructed, the conductances of the edges of K_v are identical, and equal to $\frac{\tau(G)}{n} = \frac{\alpha(G)}{n}$.Consider embedding K_v into G_v.

By Leighton-Rao (Theorem 2.5), the dilation of the embedding is $O(\log n_v / \alpha(G_v))$. Since we are embedding the complete graph on n_v points, rather than a bounded-degree graph, the congestion of the embedding is $O(n_v \cdot \log n_v / \alpha(G_v))$.

We now have that

$$\begin{aligned} \sigma(S, A) &= \left(\frac{\alpha(G)}{n} \cdot \frac{\log n_v}{\alpha(G_v)} \cdot \frac{n_v \cdot \log n_v}{\alpha(G_v)} \cdot \log n\right) \\ &\le \frac{\log^3 n}{\alpha(G_v)} \\ &= O\left(\frac{\log^3 n}{\alpha(G)}\right) \end{aligned}$$

Therefore,

$$\kappa(A, S) = \sigma(A, S) \cdot \sigma(S, A) = O\left(\frac{\log^4 n}{\alpha(G)}\right)$$

■

6
Support Trees: Evaluation

Previous sections presented the construction of support trees, the implementation of the support tree conjugate gradient method (STCG), and the theoretical analysis of the convergence properties of STCG. The theoretical analysis showed that the condition number for an *nxn* mesh was $O(n\log^2 n)$ using STCG, while the presentation on implementation of STCG showed that the tree structure of support tree preconditioners should lead to efficient execution. To examine the efficiency of the implementation, and to investigate the size of the constant in the condition number, we performed an empirical evaluation of the performance of STCG, by comparison with the performance using diagonal scaling (DSCG) and incomplete Cholesky preconditioning (ICCG) using a single vector processor of a Cray C-90. This evaluation was performed in collaboration with Marco Zagha, who was responsible for the Cray implementation. Cray C-90 time was provided by the Pittsburgh Supercomputing Center. A version of this section appeared as a CMU technical report [Gremban, Miller, and Zagha, (1994)], and in abbreviated form as a conference paper [Gremban, Miller, and Zagha, (1995)].

Similar performance evaluations of other conjugate gradient (CG) methods have been performed. Greenbaum, Li, and Chow (1989) compared the performance of four different variations of the preconditioned conjugate gradient (PCG) methods: diagonal scaling, incomplete Cholesky, hierarchical basis function, and Neumann-Dirichlet domain decomposition. The latter two are instances of multi-level preconditioning, and will not be discussed further here. All their tests were performed on a prototype of the NYU Ultracomputer and varied the number of processors used from 1 to 8. Their model problem was a time-independent version of the diffusion equation defined on the unit square with Dirichlet boundary conditions:

$$\nabla \cdot \rho(x, y)\nabla u(x, y) = f(x, y) \tag{6.1a}$$

$$(x, y) \in (0, 1) \times (0, 1) \tag{6.1b}$$

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0 \tag{6.1c}$$

The main conclusions to be drawn from the Greenbaum, *et al* (1989) study are the following:

1. In serial mode, ICCG requires nearly twice as much processing time per iteration as does DSCG. However, the reduction in the number of iterations is sufficiently large (more than a factor of three for the test

cases reported) that ICCG is, overall, faster than DSCG.

2. As the number of processors grows, the advantage in using ICCG decreases. In the experiments reported, on as few as 8 processors, DSCG outperformed ICCG in terms of total execution time. The reason for this is the difficulty in parallelizing the triangular solves required by the ICCG preconditioning (see the next subsection below for a more complete discussion).

A similar study was performed by Heroux, Vu, and Yang (1991). They compared DSCG, ICCG, least-squares polynomial preconditioning, and the multifrontal sparse Cholesky method. We shall discuss only the results for DSCG and ICCG. All experiments were performed on a Cray Y-MP using either 1 or 8 of the vector processors. The problems were obtained by assigning artificial values to various Harwell-Boeing matrices (most of which are pattern-only).

The main conclusions to be drawn from the Heroux, *et al* (1991) study are the following:

1. ICCG produced significant reductions in the number of iterations for most problems as compared to DSCG (at times by more than a factor of 1/4). However, on even a single vector processor, the total execution time of ICCG was greater than that of DSCG. On 8 processors, the ICCG:DSCG execution time ratio typically increased. This is consistent with the results of the Greenbaum, *et al* (1989) study, since a single vector processor of a Cray Y-MP can be viewed as collection of 128 parallel processors.

2. A breakdown of the solution time indicated that the primary reason for the increased execution time of ICCG was in the application of the preconditioner. The breakdown further showed that little improvement was gained in applying the preconditioner using 8 vector processors. Recall that application of the preconditioner requires the solution of two triangular systems. In general, it is difficult to parallelize triangular solves (see the discussion in Chapter 2, and Heath, *et al*, (1990)).

The Greenbaum, *et al* (1989) and Heroux, *et al* (1991) studies both point out the need for an effective parallel preconditioner — one that significantly reduces the number of iterations, but is also efficient to execute on parallel architectures. In the remainder of this section, we present an evaluation of STCG that suggests that both criteria, effectiveness and efficiency, are met by support tree preconditioners.

We used the work of Greenbaum, *et al* (1989) as a guide in our evaluation procedure. We took a typical problem, discretized it at various levels of resolution to obtain problems of various sizes, and compared the performance of the three PCG methods as a function of problem size. We took the model problem used by Greenbaum, *et al* (1989). as our two-dimensional model problem, but also compared the results for more complicated right hand sides. Since the model problem used regular meshes, we also compared the performance of the preconditioners on a sequence of irregular two-dimensional meshes. Finally, we extended the study to three dimensions, and compared results for two different sequences of three dimensional regular meshes. All experiments were performed on a Cray C-90, using a single vector processor.

6.1 Empirical Evaluation of STCG

Greenbaum, *et al* (1989) and Heroux, *et al* (1991) both conducted empirical evaluations of preconditioner performance with respect to convergence rates, and execution time (per iteration and total) on multiple processors. Greenbaum, *et al* conducted their research on a simple analytically defined PDE, which allowed them to scale the problem by varying the mesh size, and examine the performance as a function of problem size. Heroux et al. used various matrices from the Harwell-Boeing set with artificial values. Several conclusions were common to both studies. In particular, both studies found that ICCG significantly improved the convergence rate on even fairly small matrices. However, because ICCG lacks significant potential parallelism, both studies also found that the advantages of ICCG essentially vanish on vector and parallel machines.

In this section, we will demonstrate that STCG is superior to ICCG for solving large problems using serial machines, and is easily and effectively parallelized. Consequently, STCG vastly outperforms ICCG and DSCG for solving large problems on vector and parallel machines.

Because we are interested in the effects on performance as the scale of the problem changes, we primarily follow the methodology used by Greenbaum, *et al* in their study of preconditioners. We limit ourselves, to only comparing DSCG (diagonally scaled conjugate gradient), ICCG (incomplete Cholesky conjugate gradient), and STCG (support tree conjugate gradient). We compare the performance of the three solution methods versus problem size with respect to number of iterations and total running time over all iterations. In separate sections, we present the results for problems defined on a 2D regular mesh, a 2D irregular mesh, and two kinds of 3D regular meshes.

In all the results reported below, we report only the time utilized by the iterative process, and do not include the time required for formation of the preconditioners. While total time is important, in many instances the linear system will be solved many times, and the cost of forming the preconditioner can be amortized over the number of times the system is solved. Additionally, we are currently investigating the performance of various partitioning methods as one step towards constructing a version of STCG that is optimized from end to end. Currently, the code used to generate support tree preconditioners is written in NESL, an experimental data-parallel language [Blelloch (1993)]. The various implementations of PCG were written in FORTRAN.

We made no attempt to go beyond the obvious optimizations to improve the performance of ICCG. Numerous other authors have reported on the effects of ordering on ICCG [see, for example, Duff and Meurant (1989)], and on parallel implementations of ICCG [see Dongarra, *et al* (1991), van der Vorst (1989a), and van der Vorst (1989b)]. Rather than reproduce their work, we decided to extrapolate values for an optimistic implementation of ICCG.

We applied the results of other researchers discussed above in order to determine an optimistic execution time for ICCG. First, we assumed that a good node ordering could be computed and that solving the preconditioned system could be performed at the same Mflop rate as the sparse matrix-vector multiply performed at each iteration. We further assumed that the relative amount of work per iteration of ICCG was roughly twice that of DSCG. These assumptions yielded an optimistic time per iteration of ICCG to be a little more than twice that of DSCG. To be generous, we assigned a time per iteration for an optimistic ICCG to be exactly twice that of DSCG. We used this factor of two in all comparisons reported in this paper. We refer to the extrapolated optimistic ICCG as ICCG_OPT.

All results were obtained using a single processor on the Cray C-90 at the Pittsburgh Supercomputing Center.

In the discussions of the experiments that follow, all experimental results are presented as graphs. The raw results in tabular form can be found in §6.2.

6.1.1 Two-dimensional problem on regular *n*x*n* meshes

In their work, Greenbaum, *et al* (1989) considered the discretization of a time-independent version of the diffusion equation defined on the unit square with Dirichlet boundary conditions (see equations 6.1). For our experiments on regular meshes, we used the same equation with $\rho(x,y) = 1.0$, which reduces (6.1a) to Poisson's equation:

$$\nabla^2 u(x, y) = f(x, y) \tag{6.2}$$

We discretized the equation using the 5-point finite difference operator for the Laplacian, and varied the size of the *n*x*n* mesh using *n* ranging from 8 to 512 in powers of 2. In graph-theoretic terms, the resulting coefficient matrices correspond to graphs that are *n*x*n* meshes with unit weight edges and self-loops on all boundary nodes.

The support tree preconditioners for this problem were constructed using recursive coordinate partitioning in which, for each subset of points, the subset was split into four parts by bisecting first with respect to the x-coordinates and then with respect to the y-coordinates. Hence, each support tree had the form of a quadtree.

6.1.1.1 Smooth input data

For our initial experiments, we used the same forcing function as Greenbaum, *et al*:

$$f(x, y) = -2x(1-x) - 2y(1-y)$$

Our starting vector was $x^0 = 0$. We used as our stopping criterion the condition reported to be superior by Arioli, *et al* (1992):

$$\omega_2 = \frac{\|b - A \cdot \hat{x}\|_\infty}{\|A\|_\infty \cdot \|\hat{x}\|_1 + \|b\|_\infty} \tag{6.3}$$

We halted when $\omega_2 \le 1.0 \times 10^{-10}$.

Figure 6.1a shows the results in terms of number of iterations for convergence. The figure clearly shows that, while ICCG outperforms STCG in terms of number of iterations required for convergence on small meshes, the curves cross, and STCG is superior as the meshes get fairly large.

The total execution times are plotted in Figure 6.1b, with the extrapolated optimistic ICCG plotted as ICCG_OPT. Both STCG and ICCG_OPT outperform DSCG in total time, although STCG is the fastest method overall. Moreover, as the problem size increases, the difference between STCG and ICCG_OPT is increasing.

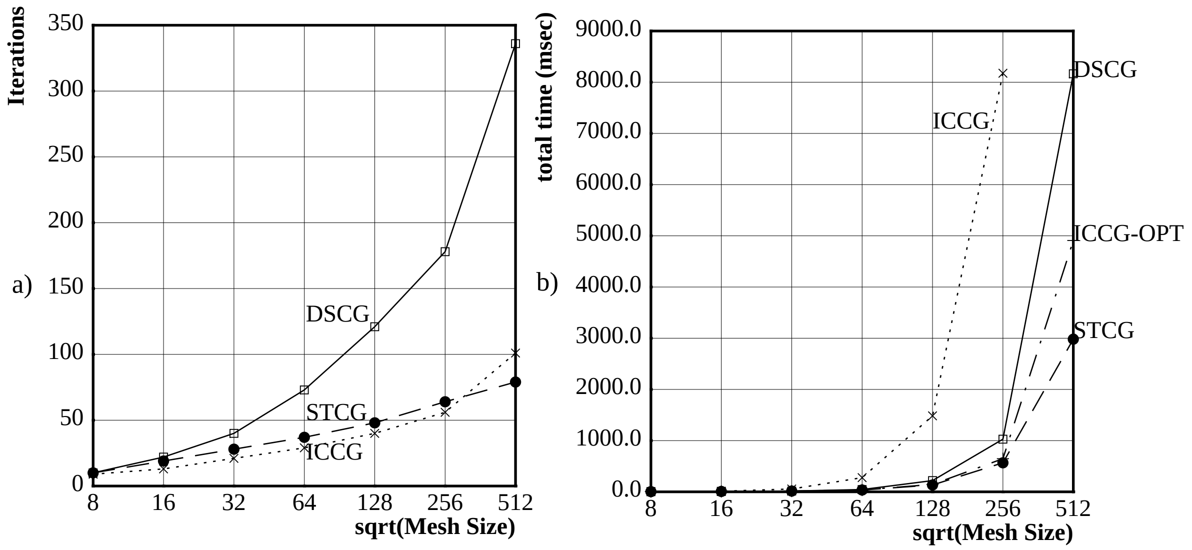


Figure 6.1: ***Results for 2D Regular Meshes, Smooth Input.***
a) *Iterations to convergence.*
b) T*otal time for iterative process on Cray C-90 (msecs).*

6.1.1.2 Random input

The forcing function used in the previous subsection was very smooth, and the problem converged to the solution fairly quickly. In a second set of experiments, we selected a more difficult right hand side. We used a random vector in which each component was independently selected from the uniform distribution on [0,1].

We used the same stopping criterion as before, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was again $x^0 = 0$.

Figure 6.2a shows the results in terms of number of iterations for convergence. In this set of experiments, convergence required as many as three times the number of iterations for the same size mesh as did the smooth input, and differences between the preconditioners became more pronounced. STCG started out performing similarly to ICCG, but improved rapidly, outperforming ICCG on the largest meshes.

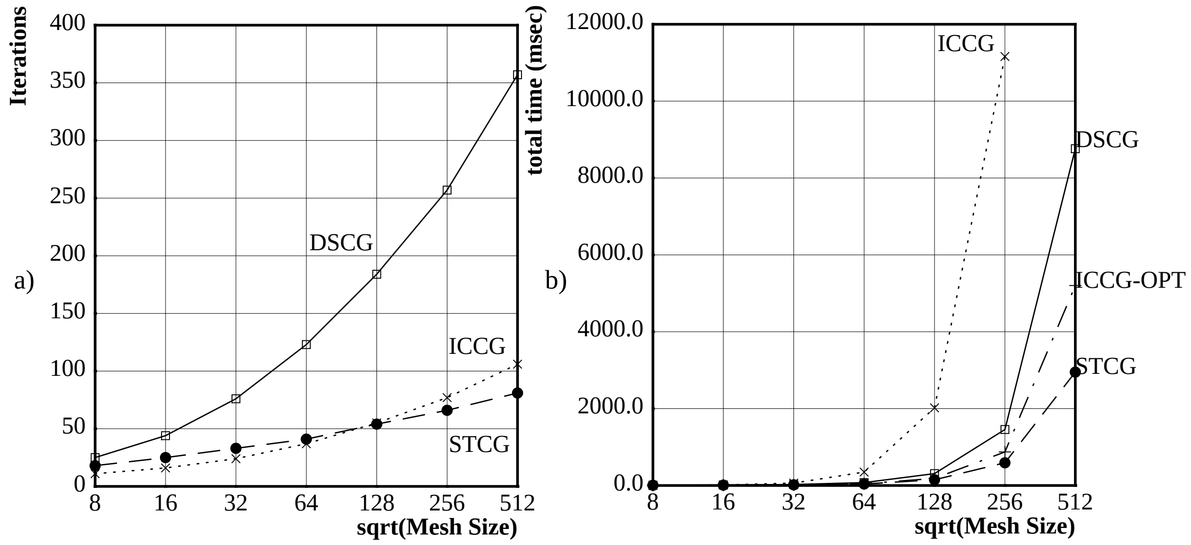


Figure 6.2: ***Results for 2D Regular Meshes, Random Input.***
a) *Iterations to convergence.*
b) *Total time for iterative process on Cray C-90 (msecs).*

Figure 6.2b shows total time for the iterative process. As above, we also show the results for an extrapolated optimistic ICCG_OPT. STCG clearly had the best total execution time. Moreover, the difference between STCG and the other methods increased with increasing mesh size.

6.1.1.3 Impulse function input

In a third set of experiments, we selected an additional difficult right hand side. We used an impulse function for b, defined by $b_0 = 1.0$, $b_i = 0.0$ for $i > 0$. In our node ordering, node 0 is the lower left hand corner of the mesh.

We used the same stopping criterion as before, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was again $x^0 = 0$.

Figure 6.3a shows the results in terms of number of iterations for convergence. In this set of experiments, convergence required even more iterations for the same size mesh as did the random input, and differences between the preconditioners became even more pronounced. Again, STCG started out with performance similar to that of ICCG, but significantly outperformed ICCG on the largest meshes

Figure 6.3b shows total time for the iterative process. Since STCG requires less work per iteration than does ICCG_OPT, and because STCG is highly vectorizable, STCG was the clear winner in terms of execution time.

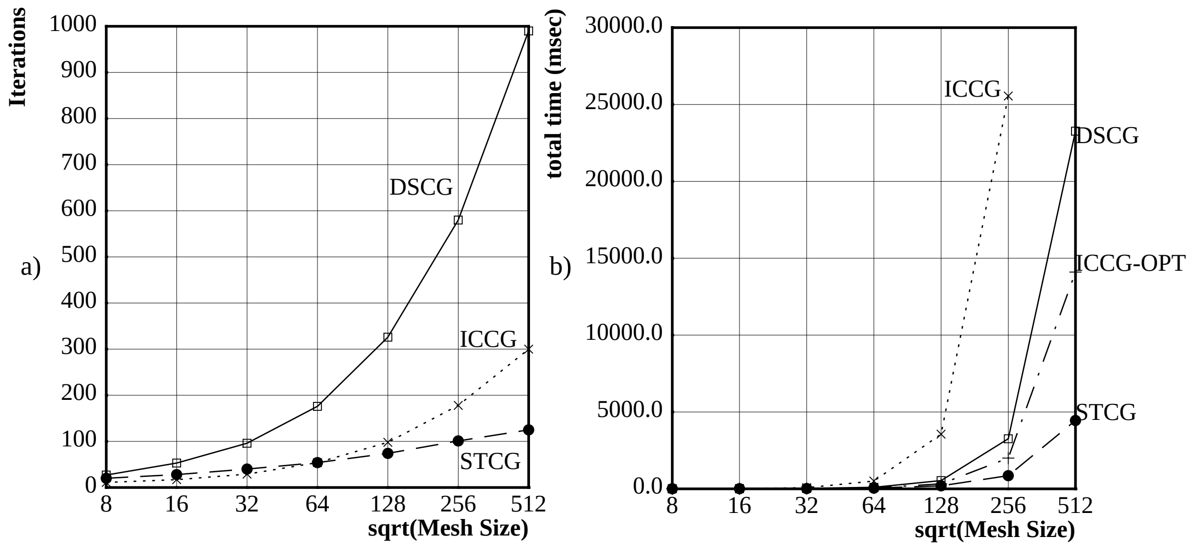


Figure 6.3: ***Results for 2D Regular Meshes, Impulse Function Input.***
a) *Iterations to convergence.*
b) T*otal time for iterative process on Cray C-90 (msecs).*

6.1.2 Two-dimensional problem on irregular meshes

The results for the PDE on a 2D regular mesh are one indication of the utility of STCG. Most application problems are not defined on regular meshes, however, so it is worthwhile to investigate the relative performance of STCG on an irregular case. We were fortunate to have available to us a nested sequence of meshes developed for an application problem — the computation of stress on a two-dimensional cracked plate. There are 9 meshes in all, with $10x2^i$ nodes in each mesh, i = 0,1,2,3,4,5,6,8,9,10. (The data for the mesh with i = 7 was unavailable.) Each mesh is a refinement of the next smaller (coarser) mesh. This sequence enabled us to investigate the performance of STCG as a function of grid size for an irregular mesh.

Figure 6.4 illustrates the coarsest and finest of the meshes. The crack in the plate runs from the center to the left side, parallel to the x-axis. The crack was defined by creating two nodes for each visible mesh point; the two nodes are not connected to each other; one connects only to nodes above the crack, while the other connects only to nodes below the crack.

The crack data consisted of pattern-only information and node coordinates. We used the pattern information to construct non-singular coefficient matrices by augmenting the Laplacian matrices of the meshes with additional diagonal weight $d_i = 1.0$ added to the nodes corresponding to the four corners. Mesh edges were given unit weights.

The support tree preconditioners for this set of problems had the form of quadtrees and were constructed using recursive coordinate partitioning.

We performed two sets of experiments. The first was conducted with a random vector (values selected uniformly between 0.0 and 1.0) as the input. The second was conducted with an impulse function as input (b_0 = 1.0, b_i = 0.0, for i > 0). For all the crack meshes, node 0 is the node at the lower left of the mesh.

For the experiments done with the crack meshes, we again used ω_2 as the stopping criterion, and halted when $\omega_2 \leq$ 1.0 x 10^{-10}. Our starting vector was again $x^0 = 0$.

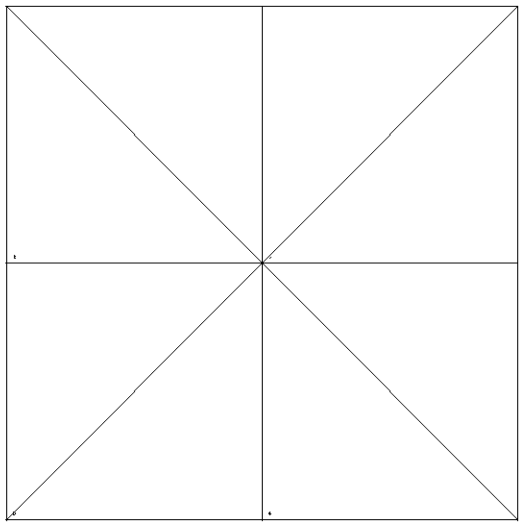
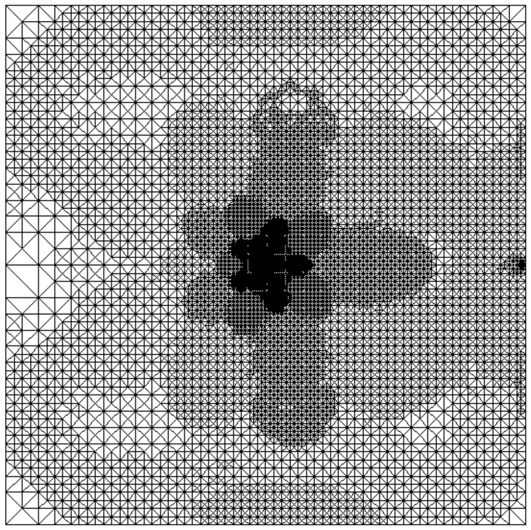
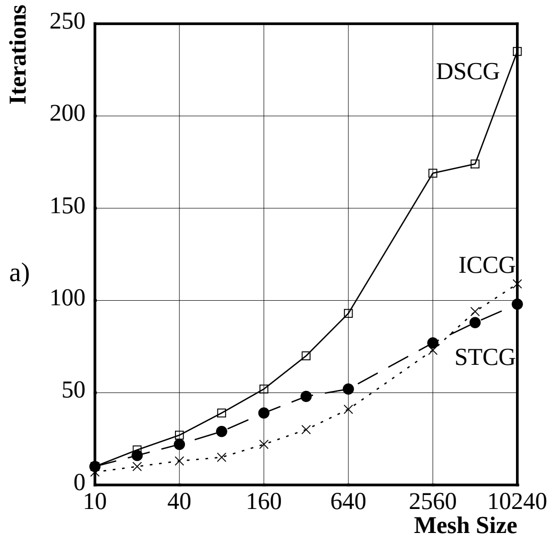
a)

b)


Figure 6.4: ***Crack Meshes.***
a) *crack00 with 10 nodes.*
b) *crack10 with 10240 nodes.*

Figure 6.5 illustrates the results of the experiments using the random vector as input. Figure 6.5a illustrates the number of iterations needed to converge to the specified tolerance as a function of the mesh size. The horizontal axis (mesh size) is plotted logarithmically. Both ICCG and STCG converge more rapidly than DSCG. While ICCG initially outperforms STCG, STCG converges more rapidly on the larger meshes. Figure 6.5b illustrates the total time taken to converge as a function of the mesh size. Again, we obtained a curve for ICCG_OPT by assuming that such an implementation would require only twice the time per iteration of DSCG. However, even this optimistic ICCG performed no better in overall time than DSCG; the two curves track each other almost perfectly. The advantage of STCG over the other methods is apparent, and the advantage is increasing with increasing mesh size. The largest crack mesh is only 10240 nodes, which is quite small for many applications.

Figure 6.6 illustrates the results of the experiments using the impulse function as input. Figure 6.6a illustrates the number of iterations needed to converge, while Figure 6.6b illustrates the total time taken for the iterative process. Again we see that STCG started off requiring more iterations than ICCG, but does not increase as fast as ICCG. On the largest meshes, STCG required fewer iterations than did ICCG. Again, because of the vectorizable nature of the support tree preconditioners, STCG was the clear winner in terms of execution time.

a)



b)

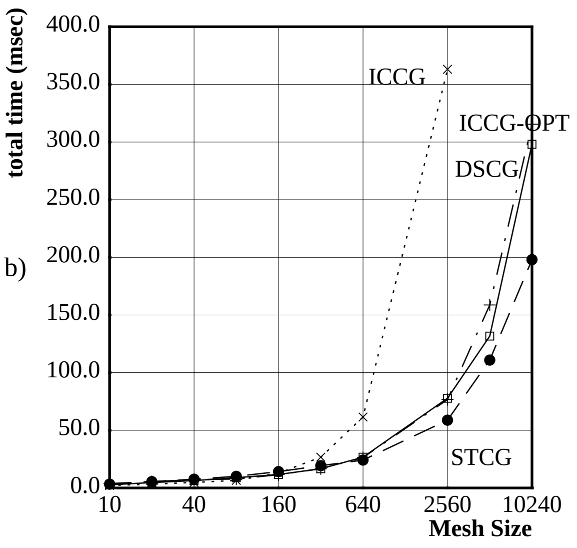
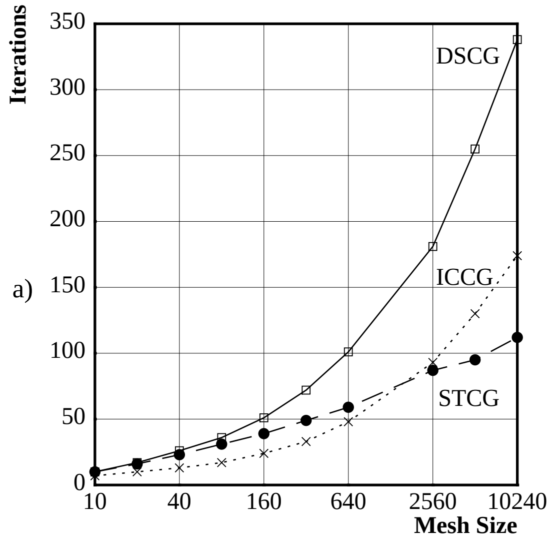


Figure 6.5: ***Results on 2D Irregular Meshes, Random Input.***
a) *Iterations to convergence.*
b) *Total execution time for iterative process on Cray C-90 (msecs).*

a)



b)

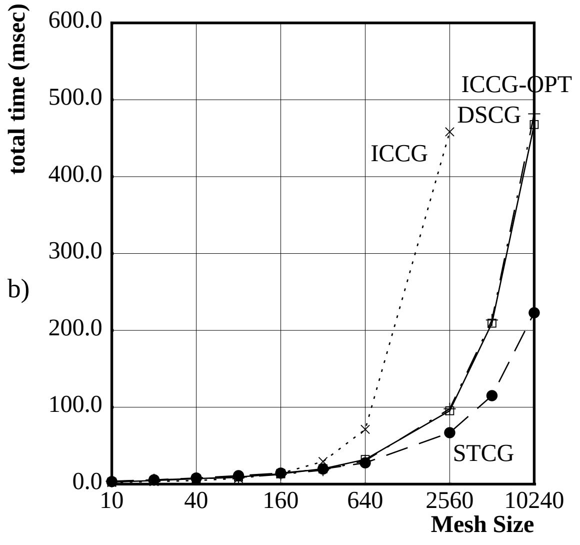


Figure 6.6: ***Results on 2D Irregular Meshes, Impulse Function Input.***
a) *Iterations to convergence.*
b) *Total execution time for iterative process on Cray C-90 (msecs).*

6.1.3 Three-Dimensional problem on regular *nxnxn* meshes

In Chapter 3, we showed that the advantage of STCG in terms of work required per iteration should increase with increasing graph dimensionality. To investigate this empirically, we performed a number of experiments in three dimensions using a regular *nxnxn* mesh.

In this set of experiments, we extended the two-dimensional problem from section 6.1.1 into three dimensions. That is, we considered the discretization of Poisson's equation defined on the unit cube with Dirichlet boundary conditions:

$$\nabla^2 u(x, y, z) = f(x, y, z) \tag{6.4a}$$

$$(x, y, z) \in (0, 1) \times (0, 1) \times (0, 1) \tag{6.4b}$$

$$u(0, y, z) = u(1, y, z) = u(x, 0, z) = u(x, 1, z) = u(x, y, 0) = u(x, y, 1) = 0 \tag{6.4c}$$

We discretized the equation using the 7-point finite difference operator for the Laplacian, and varied the size of the *nxnxn* mesh using *n* ranging from 8 to 512 in powers of 2. In graph-theoretic terms, the resulting coefficient matrices correspond to graphs that are *nxnxn* meshes with unit weight edges and self-loops on all boundary nodes.

The support tree preconditioners for this set of problems had the form of oct-trees and were constructed using recursive coordinate partitioning.

We ran two sets of experiments. The first was conducted with random vectors (values selected uniformly between 0.0 and 1.0) as the input. The second was conducted with impulse functions as input (b_0 = 1.0, b_i = 0.0, for i > 0). For the *nxnxn* mesh, node 0 is a corner node.

For these experiments, we again used ω_2 as the stopping criterion, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was $x^0 = 0$.

Figure 6.7 illustrates the results of the experiments conducted with random vectors as input. Figure 6.7a illustrates the number of iterations required for convergence, while Figure 6.7b illustrates the total execution time required for the iterative process.

Figure 6.8 illustrates the results of the experiments conducted with impulse functions as input. Figure 6.8a illustrates the number of iterations required for convergence, while Figure 6.8b illustrates the total execution time required for the iterative process.

For both random vectors and impulse functions, the problem converged extremely quickly, so it is difficult to draw definite conclusions. As for previous problems, in both of the *nxnxn* cases, STCG began by requiring more iterations for convergence than did ICCG. As observed in the previous problems, the rate of increase in the number of iterations for STCG appears to be less than that of ICCG. In terms of execution time, STCG is superior to STCG and roughly the same as ICCG_OPT.

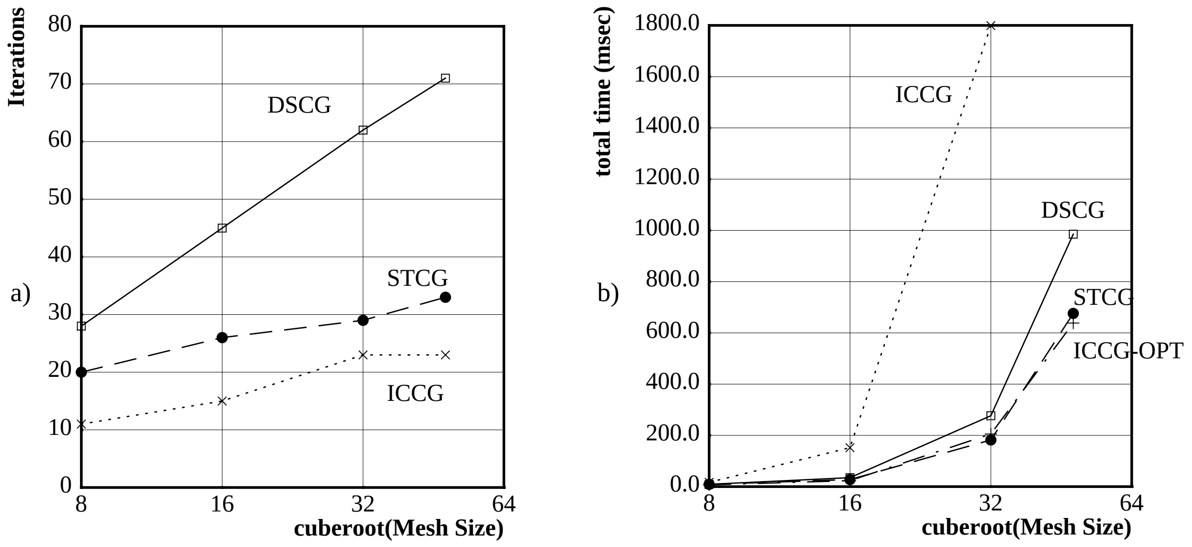


Figure 6.7: ***Results on 3D nxnxn Meshes, Random Input.***
a) *Iterations to convergence.*
b) *Total execution time for iterative process on Cray C-90 (msecs).*

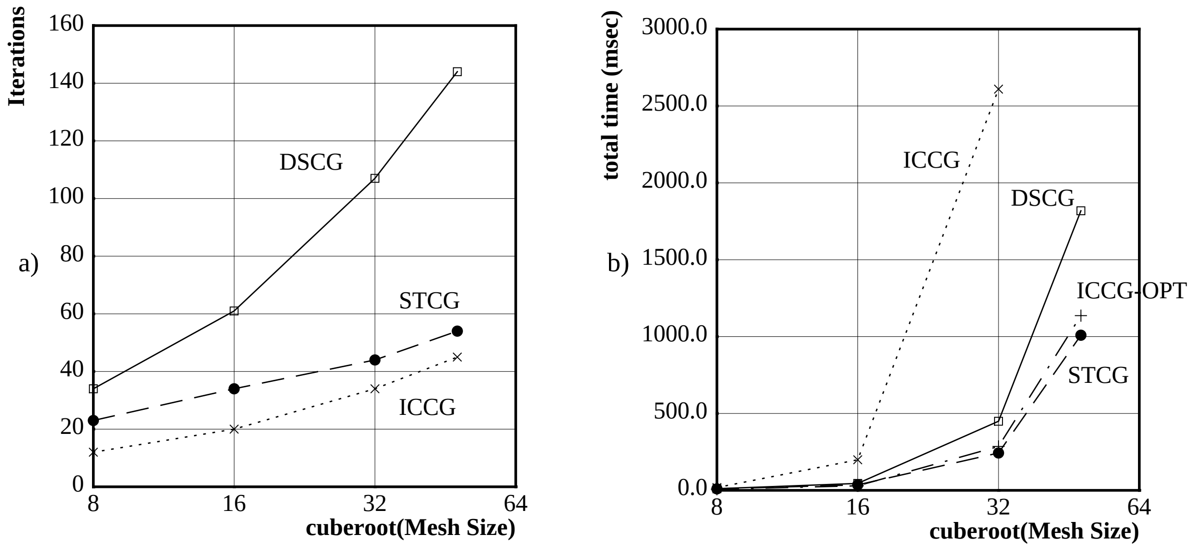


Figure 6.8: ***Results on 3D nxnxn Meshes, Impulse Function Input.***
a) *Iterations to convergence.*
b) *Total execution time for iterative process on Cray C-90 (msecs).*

6.1.4 Three-Dimensional problem on regular 8x8x*n* meshes

We stated in Chapter 3 that convergence is a function of the graph diameter. In three dimensions, the volume of a cube increases so rapidly with respect to diameter that it is difficult to construct a cubic 3D problem with a diameter large enough to require many iterations. Therefore, we defined an alternate 3D problem that would allow us to investigate convergence as a function of the graph diameter.

In this set of experiments, we modified the three-dimensional problem of equation (6.4) by extending it along one of the three dimensions. That is, we considered the discretization of Poisson's equation defined on a box:

$$\nabla^2 u(x, y, z) = f(x, y, z) \tag{6.5a}$$

$$(x, y, z) \in (0, 1) \times (0, 1) \times (0, 8n) \tag{6.5b}$$

Furthermore, we used mixed boundary conditions: Dirichlet conditions on the long ends of the box, and Neumann conditions on the sides:

$$u(x, y, 0) = u(x, y, 8n) = 0 \tag{6.5c}$$

$$\frac{\partial}{\partial x}u(x, 0, z) = \frac{\partial}{\partial x}u(x, 1, z) = 0 \tag{6.5e}$$

$$\frac{\partial}{\partial y}u(0, y, z) = \frac{\partial}{\partial y}u(1, y, z) = 0 \tag{6.5f}$$

For our experiments, we discretized the equation using the 7-point finite difference operator for the Laplacian, and varied the size of the 8x8x*n* mesh using *n* ranging from 8 to 1024 in powers of 2. In graph-theoretic terms, the resulting coefficient matrices correspond to graphs that are 8x8x*n* meshes with unit weight edges and self-loops on all boundary nodes of the 8x8 faces. The support tree preconditioners for this set of problems had the form of binary trees and were constructed using recursive coordinate partitioning.

We ran two sets of experiments. The first was conducted with random vectors (values selected uniformly between 0.0 and 1.0) as the input. The second was conducted with impulse functions as input ($b_0 = 1.0$, $b_i = 0.0$, for $i > 0$). For the 8x8x*n* mesh, node 0 is a corner node. For these experiments, we again used ω_2 as the stopping criterion, and halted when $\omega_2 \leq 1.0 \times 10^{-10}$. Our starting vector was $x^0 = 0$.

Figure 6.9 illustrates the results of the experiments with random vectors as inputs. Figure 6.9a illustrates iterations required to converge, while Figure 6.9b illustrates total time required for the iterative process.

Figure 6.10 illustrates the results of the experiments with impulse functions as inputs. Figure 6.10a illustrates iterations required to converge, while Figure 6.10b illustrates total time required for the iterative process.

The results of this set of experiments are particularly interesting. The large diameters of the graphs and the Neumann boundary conditions on the sides of the boxes led to problems that required very many iterations to converge. The differences between the preconditioners is now very apparent. While in all cases, STCG required more iterations than ICCG for small meshes, the number of iterations required for STCG is almost constant with respect to mesh diameter, while that of ICCG is clearly increasing. By the time the mesh diameter is over 512, STCG requires fewer than half the number of iterations of ICCG.

The difference in execution time is even more dramatic. As stated previously, STCG vectorizes extremely well. For problems of the size considered here, the vector lengths at the lowest level of the support tree range from thousands to hundreds of thousands.

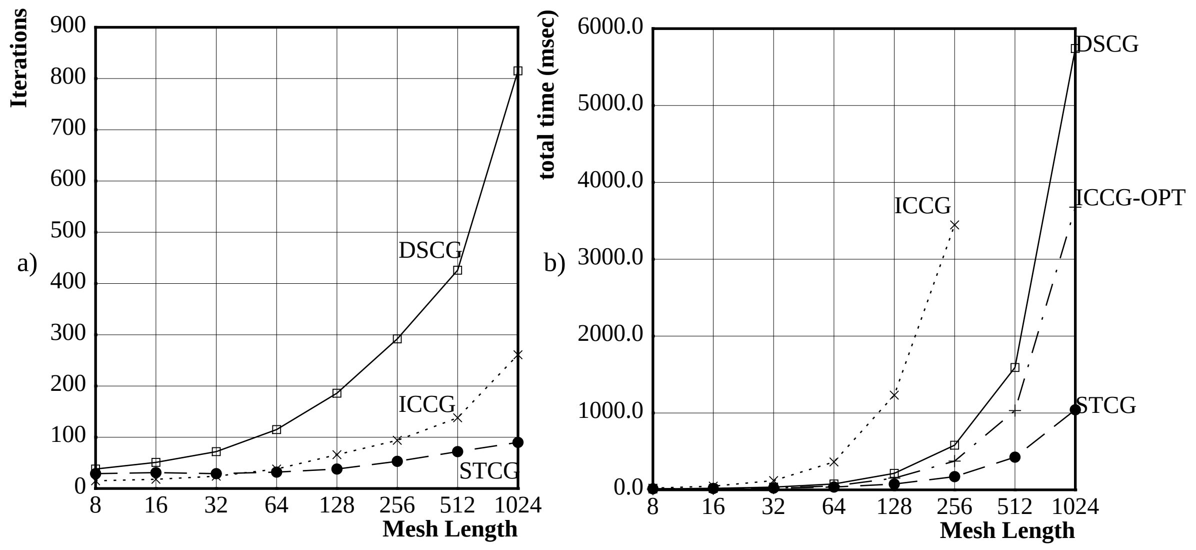


Figure 6.9: ***Results on 3D 8x8xn Mesh, Random Input.***
a) *Iterations to convergence.*
b) *Total execution time for iterative process on Cray C-90 (msecs).*

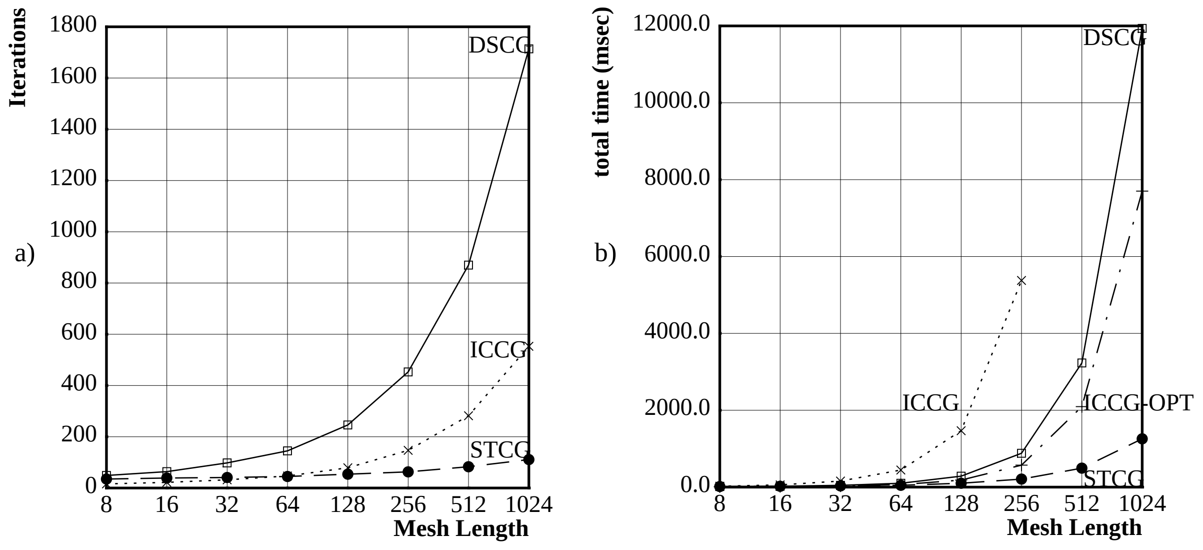


Figure 6.10: ***Results on 3D 8x8xn Mesh, Impulse Function Input.***
a) *Iterations to convergence.*
b) *Total execution time for iterative process on Cray C-90 (msecs).*

6.2 Tabulated Experimental Results

6.2.1 Results from 2D regular meshes

Table 6.1: ***Results of Experiments on 2D Square Meshes, Smooth Input***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	10	9	10	.29	.43	.39	2.9	3.9	3.9
16	22	13	19	.28	.88	.41	6.2	11.4	7.8
32	40	21	28	.36	2.61	.54	14.4	54.8	15.1
64	73	29	37	.66	9.56	1.03	48.0	280.0	38.0
128	121	40	48	1.83	37.00	2.90	222.0	1480.0	139.0
256	178	56	64	5.63	144.94	8.91	1002.5	8116.7	570.4
512	336	101	79	24.30	578.69	37.73	8164.9	58447.6	2981.0

Table 6.2: ***Results of Experiments on 2D Square Meshes, Random Input***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	25	11	18	.26	.43	.35	6.6	4.7	6.3
16	44	16	25	.27	.86	.38	11.8	13.7	9.4
32	76	24	33	.34	2.60	.50	25.8	62.5	16.4
64	123	37	41	.61	9.44	.94	75.3	349.1	38.7
128	184	55	54	1.68	36.73	2.76	308.9	2020.4	148.9
256	257	77	66	5.67	144.95	8.96	1456.4	11161.1	591.1
512	357	106	81	24.54	578.78	36.42	8761.8	61350.7	2950.0

Table 6.3: ***Results of Experiments on 2D Square Meshes, Impulse Input***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	27	11	20	.26	.43	.35	7.0	4.7	6.9
16	53	17	28	.27	.86	.38	14.2	14.6	10.5
32	96	29	40	.34	2.58	.49	32.2	74.7	19.4
64	176	54	54	.56	9.38	.87	99.2	506.3	46.9
128	326	98	74	1.65	36.28	2.66	537.6	3555.0	195.8
256	580	178	101	5.62	143.55	8.50	3259.2	25551.7	858.1
512	990	300	125	23.50	572.17	35.58	23266.5	171649.9	4447.0

6.2.2 Results from 2D irregular meshes

Table 6.4: ***Results of Experiments on 2D Irregular Meshes, Random Input.***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
10	10	7	10	.27	.31	.33	2.7	2.2	3.3
20	17	10	16	.28	.33	.34	4.7	3.3	5.5
40	23	13	22	.28	.37	.35	6.4	4.8	7.6
80	34	15	29	.26	.44	.35	8.9	6.6	10.1
160	45	22	39	.26	.59	.36	11.7	12.9	14.2
320	60	30	48	.28	.89	.40	16.7	26.7	19.3
640	83	41	52	.32	1.50	.47	26.8	61.6	24.2
2560	148	73	77	.53	4.97	.76	77.8	363.1	58.8
5120	156	94	88	.84	9.46	1.26	131.7	889.2	110.9
10240	206	109	98	1.45	18.35	2.02	298.1	1999.9	198.0

Table 6.5: ***Results of Experiments on 2D Irregular Meshes, Impulse Input.***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
10	10	7	10	.27	.31	.33	2.7	2.2	3.3
20	17	10	16	.28	.33	.35	4.7	3.3	5.6
40	26	13	23	.27	.36	.34	7.1	4.7	7.9
80	36	17	31	.26	.44	.35	9.3	7.4	11.0
160	51	24	39	.26	.58	.37	13.4	14.0	14.3
320	72	33	49	.28	.89	.40	20.0	29.4	19.7
640	101	48	59	.32	1.48	.47	32.1	71.2	27.7
2560	181	93	87	.53	4.93	.77	95.4	458.3	66.9
5120	255	130	95	.82	9.40	1.21	209.4	1222.2	115.1
10240	338	174	112	1.38	18.27	1.99	467.8	3178.4	222.9

6.2.3 Results from 3D regular meshes (*nxnxn*)

Table 6.6: ***Results of Experiments on nxnxn Regular Meshes, Random Input.***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	28	11	20	.32	1.54	.45	9.0	16.9	8.9
16	45	15	26	.78	10.13	1.07	35.0	152.0	27.7
32	62	23	29	4.46	78.23	6.28	276.7	1799.3	182.1
48	71	23	33	13.88	260.41	20.48	985.3	5989.4	675.9

Table 6.7: ***Results of Experiments on nxnxn Regular Meshes, Impulse Input.***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	34	12	23	.31	1.53	.43	10.7	18.3	9.8
16	61	20	34	.74	9.94	1.05	45.0	198.8	35.6
32	107	34	44	4.20	76.76	5.53	449.4	2609.8	243.1
48	144	45	54	12.62	251.93	18.69	1818.4	11336.8	1009.1

6.2.4 Results from 3D regular meshes (8x8x*n*)

Table 6.8: ***Results of Experiments on 8x8xn Regular Meshes, Random Input.***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	38	15	29	.30	1.49	.48	11.5	22.3	13.8
16	51	18	31	.38	2.68	.58	19.2	48.2	18.0
32	72	24	29	.49	5.01	.80	35.4	120.3	23.2
64	115	38	32	.66	9.54	1.13	76.3	362.7	36.3
128	186	66	38	1.15	18.66	1.98	214.6	1231.8	75.2
256	292	94	53	1.99	37.67	3.24	580.9	3446.7	171.9
512	426	138	72	3.74	72.95	5.91	1591.6	10066.6	425.2
1024	815	261	90	7.05	145.01	11.57	5741.8	37848.2	1041.7

Table 6.9: ***Results of Experiments on 8x8xn Regular Meshes, Impulse Input.***

size	iterations			time/iteration (msecs)			total time (msecs)		
n	DSCG	ICCG	STCG	DSCG	ICCG	STCG	DSCG	ICCG	STCG
8	49	16	35	.30	1.48	.46	14.8	23.6	16.2
16	64	22	39	.36	2.65	.57	23.2	58.4	22.2
32	98	32	41	.48	4.94	.77	46.7	158.2	31.7
64	145	47	45	.69	9.47	1.06	100.7	445.2	47.6
128	246	79	54	1.16	18.57	1.88	285.3	1467.1	101.7
256	453	147	63	1.95	36.58	3.31	881.3	5377.3	208.7
512	870	282	83	3.71	72.62	5.95	3230.7	20479.8	493.9
1024	1714	553	111	6.96	144.32	11.32	11935.6	79811.1	1256.7

6.3 Summary and Discussion

In this chapter, we presented an evaluation of support tree preconditioners. Through numerical experiments run on a single vector processor of a Cray C-90, we have demonstrated that on both irregular and regular meshes:

- the performance of STCG, in terms of iterations to converge, meets or exceeds the performance of ICCG, which in turn, outperforms DSCG.

  In all but one of the sets of experiments reported here, STCG began to outperform ICCG on fairly small matrices (2000 to 5000 nodes), with the difference in performance increasing with the size of the problem. In the experiments in which STCG did not outperform ICCG in terms of convergence rate, convergence was extremely rapid, so acceleration from preconditioning had a minimal effect, and STCG exhibited performance very close to that of ICCG. On problems that take many iterations to converge, STCG requires far fewer iterations than does ICCG.

- in terms of execution time, STCG outperforms both ICCG and DSCG on scalar processors, and far outperforms them on vector processors.

  On a scalar machine, execution time is the product of the number of iterations and the time/work per iteration. In comparison to DSCG, our analysis showed that STCG requires slightly less than twice the amount of work per iteration, and our experiments showed that STCG requires fewer than half the number of iterations. Hence, STCG is preferable to DSCG for large problems on a scalar processor. Analysis also showed that STCG requires less work per iteration than ICCG, and our experiments showed that, in most cases, STCG requires fewer iterations. Therefore, STCG is also preferable to ICCG on scalar processors.

  All our experiments were performed on a single vector processor of a Cray C-90. Without exception, for large meshes STCG outperformed both DSCG and ICCG, often by very wide margins.The reason for the performance advantage is that the STCG preconditioner has a tree structure, which allows all nodes at a given level to be evaluated in parallel.

  STCG preconditioners can be easily and efficiently level scheduled by leaf raking. The lower triangular matrices that appear in ICCG will not, in general, allow as many nodes to be evaluated in parallel as can be evaluated in STCG. For example, in the case of square meshes, moderate parallel efficiency can be obtained by ordering the nodes so that the incomplete Cholesky preconditioner is evaluated along diagonals of the mesh [Dongarra, *et al* (1991), Golub and Ortega (1993), van der Vorst (1989b)]; for an nxn mesh, this ordering requires $2n$ parallel steps with an average of $n/2$ nodes evaluated in parallel at each step. In contrast, the STCG preconditioner for an nxn mesh yields $2\lceil \log n \rceil$ parallel steps with an average of $n^2/\log n$ nodes evaluated at each step.

  For irregular graphs, the ordering problem is even more complicated. Figure 6.11 shows the graph structure of the incomplete Cholesky preconditioner for the fifth mesh in the crack series (160 nodes). An examination of the graph shows that it would be difficult to determine an optimal evaluation order for level scheduling. In contrast, Figure 6.12 shows the graph structure of the support tree preconditioner for the same graph. The simplicity of the support tree structure is apparent. Moreover, we believe that the regular structure of the support tree also makes implementation easier on distributed memory machines by reducing the amount of communication and synchronization required.

  Additional parallelism of STCG is possible due to the tree structure of the STCG preconditioner — separate subtrees may be evaluated in parallel on multiple vector processors.

.

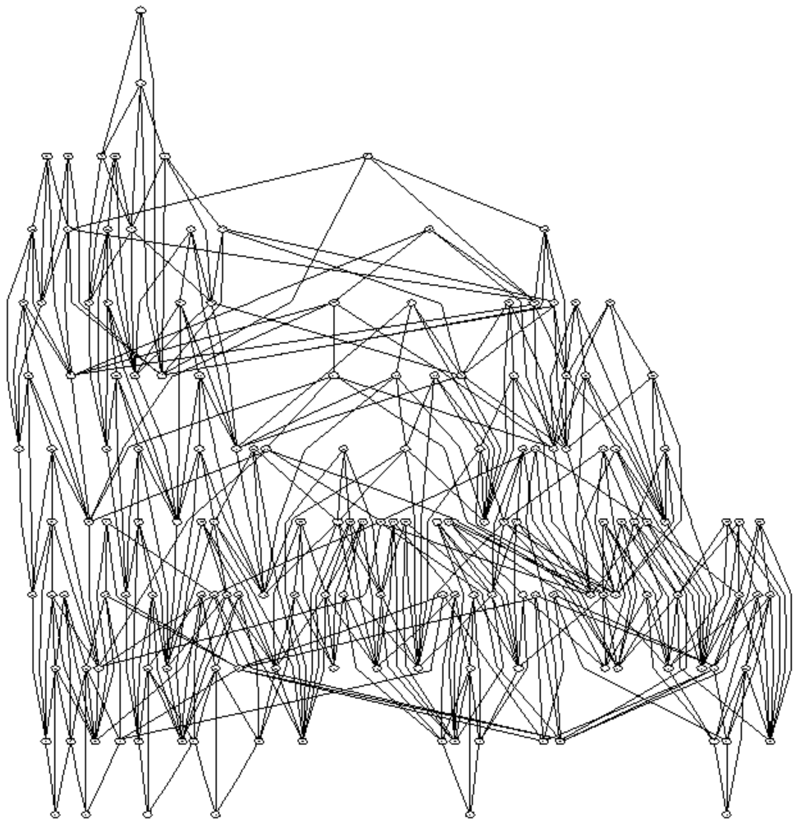


Figure 6.11: ***The graph structure of the incomplete Cholesky preconditioner for the 160 node crack mesh.***

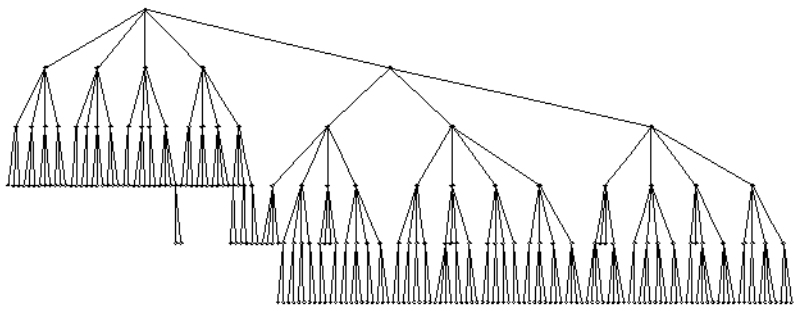


Figure 6.12: ***The graph structure of the support tree preconditioner for the 160 node crack mesh.***

7
Extensions and Applications of Combinatorial Analysis

Chapters 3-6 of this thesis presented a description of a new class of combinatorial preconditioners. The theoretical properties of these preconditioners were analyzed with a collection of tools based on the equivalence of Laplacian matrices, resistive networks, and edge-weighted, undirected graphs. In this chapter, we show how the same set of tools can be used to extend the domain of support trees to all symmetric diagonally dominant matrices. We also show how the techniques used to analyze support tree performance can be used in the analysis of a standard problem in linear algebra.

7.1 Symmetric and Diagonally Dominant Matrices

In the preceding chapters, support trees were defined and analyzed for coefficient matrices that were symmetric, diagonally dominant, and had non-positive off-diagonal elements. There is a very straightforward way to extend support trees to handle *all* symmetric and diagonally dominant matrices.

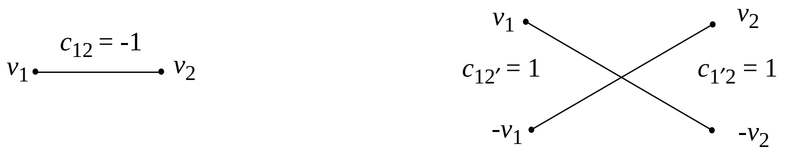
Consider the case of real symmetric diagonally dominant matrices with some positive off-diagonal elements. One of the simplest examples is

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{7.1}$$

Writing out the equation for the first component of $\boldsymbol{i} = M\boldsymbol{u}$ yields $i_1 = u_1 + u_2 = u_1 - (-u_2)$. That is, i_1 looks like the current resulting from a resistive connection between node n_1 and the *negative* of node n_2! Therefore, we get the should get the same result for i_1 and i_2 using M as we would from the matrix of a network twice as large in which node n_1 is connected to a node $n_{2'}$ and n_2 is connected to $n_{1'}$, where $u_{1'} = -u_1$ and $u_{2'} = -u_2$. The corresponding linear system is shown below.

$$\begin{bmatrix} i_1 \\ i_2 \\ -i_1 \\ -i_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ -u_1 \\ -u_2 \end{bmatrix} \tag{7.2}$$

A positive off-diagonal in some sense represents a negative conductance, but can be realized in a network of twice the size using only positive conductances. The networks corresponding to the matrices in (7.1) and (7.2) are illustrated in Figure 7.1.



$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}$$

Figure 7.1: *Resistive Networks with Negative Conductance.*
a) *A simple network with negative conductance.*
b) *The corresponding network with positive conductances.*

We now formalize these observations. First, we expand the notion of a Laplacian matrix to that of a *generalized Laplacian*, which may include positive off-diagonals:

7.1 ***Definition:*** (generalized Laplacian) *L is a generalized Laplacian matrix (generalized Laplacian) if L is real, symmetric, and diagonally dominant.*

Next, we define the notion of an *expanded matrix*, which formalizes the relationship between generalized Laplacian matrices like that in (7.1), and the larger related Laplacian matrices like that in (7.2).

7.2 ***Definition:*** (expanded matrix) *Let L be an nxn generalized Laplacian matrix. Then L = N + P, where N is a Laplacian matrix and P is symmetric and diagonally dominant with non-negative off-diagonals. Let M =* $M_1 + M_2$ *be a 2nx2n matrix constructed as follows*:

- $M_1(i,j) = M_1(n+i,n+j) = N(i,j)$, $1 \le i,j \le$ n
- *if* $P(i,j) = P(j,i) \neq 0$, *then*
  - $M_2(i,i) = M_2(n+i,n+i) = P(i,i)$;
  - $M_2(j,j) = M_2(n+j,n+j) = P(j,j)$;
  - $M_2(i,n+j) = M_2(n+j,i) = -P(i,j)$;
  - $M_2(j,n+i) = M_2(n+i,j) = -P(j,i)$.

M is a 2nx2n Laplacian matrix; we call M the expanded matrix, or expansion, of L.

Note that, if *M* is a standard Laplacian, then the expansion of *M* is simply $M = \begin{bmatrix} L & 0 \\ 0 & L \end{bmatrix}$.

7.3 ***Lemma:*** *Let L be an nxn generalized Laplacian matrix. Let M be the expansion of L. Let u be any vector of applied voltages* $\boldsymbol{u} = \begin{bmatrix} u_1 & \dots & u_n \end{bmatrix}^t$, *and let* $\boldsymbol{w} = \begin{bmatrix} u_1 & \dots & u_n & (-u_1) & \dots & (-u_n) \end{bmatrix}^t$, $\boldsymbol{i} = L\boldsymbol{u}$, *and* $\boldsymbol{j} = M\boldsymbol{w}$. *Then, for all* $k \in \{1 \dots n\}$, *we have* $i_k = j_k = -j_{n+k}$.

proof:

Let $k \in \{1,...,n\}$. From the construction of M, we have that L can be written as $L = N + P$, where N is a Laplacian and P is symmetric and diagonally dominant with non-negative off-diagonals. Let $R = \{r_1,...,r_m\}$ be the set of non-zero indices of the kth row of N, and $S = \{s_1,...s_q\}$ be the set of non-zero indices of the kth row of P. Assume that r_1 and s_1 are the indices of the diagonal elements: $r_1 = k = s_1$. Then,

$$i_k = \sum_R N(k, r_l)u_{r_l} + \sum_S P(k, s_t)u_{s_t}$$

and,

$$\begin{aligned} j_k &= \sum_R F(k, r_l)w_{r_l} + \sum_S F(k, s_t + n)w_{s_t} \\ &= \sum_R N(k, r_l)u_{r_l} + \sum_{S \backslash s_1} -P(k, s_t)(-u_{s_t}) + P(k, s_1)u_{s_1} \\ &= \sum_R N(k, r_l)u_{r_l} + \sum_S P(k, s_t)u_{s_t} \\ &= i_k \end{aligned}$$

And a similar derivation shows $i_k = -j_k+n$.

■

The definitions and lemma above show that a generalized Laplacian formally corresponds to a Laplacian of twice the size, the expansion. Since the expansion is a Laplacian, all the tools developed in the previous chapters can now be applied. We can now construct a support tree for any symmetric diagonally dominant matrix L by first constructing the expansion M, and then using the graph structure of M to construct a support tree for M. STCG can then be applied to the linear system defined by M, which is only a constant factor larger than that defined by L.

It may be the case that while L is non-singular with some positive off-diagonals, the graph corresponding to M consists of two unconnected components. This can be easily detected by running a connected components algorithm on the graph of M. Such algorithms have run times of order $O(m\log n)$, where m is the number of edges and n is the number of nodes. Since we are dealing with sparse graphs, m is $O(n)$, so connectivity can be determined in nearly linear time.

Let L be a generalized Laplacian with expansion M. It may be that the graph topology underlying L contains information that may be used to construct the support tree for M. This is an area of research that we have not pursued.

7.2 Bounding the Largest Eigenvalue

In this section, we show how the theoretical tools developed for support tree analysis can be applied to a standard problem in linear algebra — bounding the eigenvalues of a matrix. This is a problem of both theoretical and practical interest. For example, given a symmetric positive definite matrix A, for polynomial preconditioning and for Chebyshev acceleration, it is necessary to know an interval I_λ = [a,b] which contains the spectrum of A, $\lambda(A)$. [Ashby (1987), Hageman and Young (1981)].

A common method for finding an upper bound on the eigenvalues is the Gerschgorin Circle Theorem. Following the treatment of Golub and Ortega (1993), let *A* be a square matrix of order *n*, and define

$$r_i = \sum_{j \neq i} |a_{ij}|, \; i = 1, \ldots, n$$
$$\Lambda_i = \{z : |z - a_{ii}| \leq r_i\} \;, \; i = 1, \ldots, n$$

Then the Λ_i are disks in the complex plane centered at a_{ii} with radius r_i.

7.4 ***Gerschgorin's Theorem:*** *All the eigenvalues of A lie in the union of the disks* Λ_1*,...,* Λ_n*. Moreover, if S is a union of m disks such that S is disjoint from all the other disks, then S contains exactly m eigenvalues of A (counting multiplicities).*

Thus, using Gerschgorin's Theorem, the magnitude of $\lambda_{\max}$, the largest eigenvalue of *A*, must be less than or equal to the furthest extent of any of the Gerschgorin disks. That is, for a matrix with all real eigenvalues:

$$\lambda_{max} \leq max_i\{a_{ii} + r_i\}$$

Gerschgorin's Theorem provides a quick and easy way to estimate eigenvalues. For a real symmetric matrix, simply compute the sum of the absolute values of all the elements on a row/column of the matrix; the largest sum is an upper bound on $\lambda_{\max}$.

The combinatorial techniques developed for the analysis of support tree preconditioners can be used to quickly supply a bound on the maximum eigenvalue for real symmetric non-diagonal matrices that can be tighter than that obtained using Gerschgorin's Theorem. In particular, Lemma 7.7 will show that for matrices corresponding to connected graphs, the method is to simply find the largest sum of two diagonal elements that are connected by a non-zero off-diagonal; that is,

$$\lambda_{\max} \leq \max\{a_{ii} + a_{jj} : a_{ij} \neq 0\} \; . \tag{7.3}$$

For many graphs, this method often provides a tighter bound than does Gerschgorin's Theorem. A similar result for all Laplacian matrices is presented in Theorem 7.8.

To illustrate the application of Gerschgorin's Theorem and our combinatorial result, several example matrices are presented in Figure 7.2. All of the eigenvalues were computed using Matlab [MathWorks (1992)].

- The Dirichlet matrix of order 4 shown at the top of the figure is derived from a finite difference discretization of Laplace's equation in one dimension with Dirichlet boundary conditions: the actual value of $\lambda_{\max}$ is 3.6180, while the Gerschgorin and combinatorial bounds are both 4.0.

- The Poisson matrix of order 9 shown at the center of the figure is the Laplacian matrix of a 3x3 mesh: the actual value of $\lambda_{\max}$ is 6.0, the Gerschgorin bound is 8.0, and the combinatorial bound is 7.0. For the Dirichlet and Poisson matrices, the Gerschgorin bounds are quite good, and the combinatorial bound provides little improvement.

- The Laplacian matrix of a "wagon wheel" graph of order 9, is shown at the bottom of the figure: the actual value of $\lambda_{\max}$ is 9.0, the Gerschgorin bound is 16.0, and the combinatorial bound is 11.0.

The theory behind this new method of combinatorial eigenvalue estimation is only the newest of many results that combine the seemingly different areas of matrix theory, graph theory, and circuit theory. Fiedler (1973) was one of the first to establish an interesting relationship between a graph and its corresponding connectivity matrix. He showed that λ_2, the second-smallest eigenvalue of the connectivity matrix of a graph, is non-zero if and only if the graph is

$$D = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

Dirichlet matrix of order 4.
$\lambda_{\max}$ *= 3.6180, Gerschgorin bound = 4.0, combinatorial bound = 4.0*

$$P = \begin{bmatrix} 2 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 3 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix}$$

Poisson matrix of order 9.
λ_{max} *= 6.0, Gerschgorin bound = 8.0, combinatorial bound =7.0*

$$W = \begin{bmatrix} 3 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 8 \end{bmatrix}$$

“Wagon wheel” matrix of order 9.
λ_{max} *= 9.0, Gerschgorin bound = 16.0, combinatorial bound = 11.0*

Figure 7.2: ***Example matrices with maximum eigenvalues, Gerschgorin bounds, and combinatorial bounds.***

connected; he called λ_2 the *algebraic connectivity* of the graph. Pothen, *et al* (1990), and Hendrickson and Leland (1992) are among the proponents of the *spectral* approach to graph partitioning, which uses the eigenvectors of the connectivity matrix of a graph to partition the graph. Chandra, *et al* (1989), and Doyle and Snell (1984) discuss the relationship between the resistance of the electrical network corresponding to a graph and properties of random walks on the graphs.

Our theory builds primarily on these latter results dealing with the electrical resistance of a graph. In particular, we show that the largest eigenvalue of a matrix is bounded by the gain factor needed for one type of circuit to conduct the same amount of current as the circuit corresponding to the matrix.

Since $\lambda(A) = \lambda(A,I)$, we can apply the same methods used for bounding finite generalized eigenvalues of Laplacian matrices to bounding eigenvalues of any real symmetric matrices. We start by bounding eigenvalues of generalized Laplacian matrices. The trick is to use the augmented matrix instead of the generalized Laplacian. This is justified with the following lemma.

7.5 ***Lemma:*** *Let A and B be Laplacian matrices with augmentations E and F respectively. If λ is a finite generalized eigenvalue with corresponding eigenvector* $\boldsymbol{x}$ *such that*

$$A\boldsymbol{x} = \lambda B\boldsymbol{x}$$

then

$$E\boldsymbol{y} = \lambda F\boldsymbol{y}\text{, where } \boldsymbol{y} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix}.$$

proof:

A and B are Laplacian matrices, so there exist matrices M, N, R, and S such that $A = M + R$, $B = N + S$, where M and N have the zero row/column sum property, and both R and S are diagonal. let $\boldsymbol{r} = diag(R)$, and $\boldsymbol{s} = diag(S)$, where $diag(A) = [a_{11},...,a_{nn}]^t$. The augmentation process yields matrices E and F, where

$$E = \begin{bmatrix} A & -\boldsymbol{r} \\ -\boldsymbol{r}^t & \boldsymbol{r}^t\boldsymbol{1} \end{bmatrix}, F = \begin{bmatrix} B & -\boldsymbol{s} \\ -\boldsymbol{s}^t & \boldsymbol{s}^t\boldsymbol{1} \end{bmatrix}\text{, and } \boldsymbol{1} = \begin{bmatrix} 1 & \dots & 1 \end{bmatrix}^t.$$

Let $\lambda \in \underline{\lambda}(A,B)$, with associated unit eigenvector $\boldsymbol{x}$. Then,

$$A\boldsymbol{x} = \lambda B\boldsymbol{x}$$

$$\boldsymbol{1}^t A\boldsymbol{x} = \boldsymbol{1}^t\lambda B\boldsymbol{x}$$

$$\boldsymbol{1}^t(M+R)\boldsymbol{x} = \boldsymbol{1}^t\lambda(N+S)\boldsymbol{x}$$

$$\boldsymbol{1}^t M\boldsymbol{x} + \boldsymbol{1}^t R\boldsymbol{x} = \boldsymbol{1}^t\lambda N\boldsymbol{x} + \boldsymbol{1}^t\lambda S\boldsymbol{x}$$

M and N have the zero column sum property, $\boldsymbol{1}^t M\boldsymbol{x} = 0 = \boldsymbol{1}^t\lambda N\boldsymbol{x}$, so the equation above yields

$$\boldsymbol{1}^t R\boldsymbol{x} = \boldsymbol{1}^t\lambda S\boldsymbol{x}$$

$$\boldsymbol{r}^t\boldsymbol{x} = \lambda\boldsymbol{s}^t\boldsymbol{x}$$

Now, let $\boldsymbol{y} = [\boldsymbol{x}^t,\boldsymbol{0}]^t$, and consider $E\boldsymbol{y}$ and $\lambda F\boldsymbol{y}$:

$$F\boldsymbol{y} = \begin{bmatrix} B & -\boldsymbol{s} \\ -\boldsymbol{s}^t & \boldsymbol{s}^t\boldsymbol{1} \end{bmatrix}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} B\boldsymbol{x} \\ -\boldsymbol{s}^t\boldsymbol{x} \end{bmatrix}$$

$$E\boldsymbol{y} = \begin{bmatrix} A & -\boldsymbol{r} \\ -\boldsymbol{r}^t & \boldsymbol{r}^t\boldsymbol{1} \end{bmatrix}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} A\boldsymbol{x} \\ -\boldsymbol{r}^t\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \lambda B\boldsymbol{x} \\ -\lambda\boldsymbol{s}^t\boldsymbol{x} \end{bmatrix} = \lambda F\boldsymbol{y}$$

■

The lemma above holds for Laplacian matrices. To deal with generalized Laplacians (those with some positive off-diagonal values), we need to first compute the expansion, then the augmentation. That an upper bound on the eigenvalues of the expansion is also an upper bound of the eigenvalues of the original generalized Laplacian is shown in the next lemma.

7.6 ***Lemma:*** *Let A be a generalized Laplacian matrix, and let M be the expansion of A. Then* $\lambda_{max}(A) \leq \lambda_{max}(M)$.

proof:

If A contains no positive off-diagonals, then M is simply double the size of A, with a copy of A at the upper left and lower right corners (and 0 elsewhere). Therefore the eigenvalues of M are simply the eigenvalues of A with twice the multiplicity. Thus, $\lambda_{max}(A) \leq \lambda_{max}(M)$.

Suppose A contains at least one positive off-diagonal. Then M is constructed so that, given any vector $\boldsymbol{u}$,

$$M\begin{bmatrix} \boldsymbol{u} \\ -\boldsymbol{u} \end{bmatrix} = \begin{bmatrix} A\boldsymbol{u} \\ -A\boldsymbol{u} \end{bmatrix}$$

Let $\boldsymbol{x}$ be a unit eigenvector corresponding to $\lambda_m = \lambda_{max}(A)$. Then

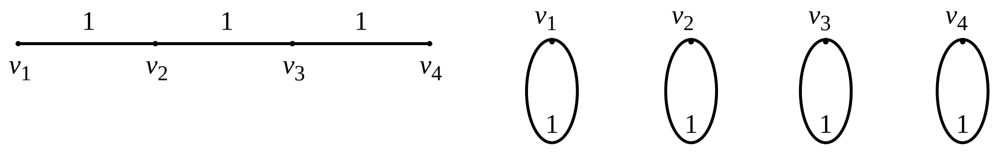
$$M\begin{bmatrix} \boldsymbol{x} \\ -\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} A\boldsymbol{x} \\ -A\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \lambda_m\boldsymbol{x} \\ -\lambda_m\boldsymbol{x} \end{bmatrix} = \lambda_m\begin{bmatrix} \boldsymbol{x} \\ -\boldsymbol{x} \end{bmatrix}$$

So λ_m is also an eigenvector of M. Therefore $\lambda_{max}(A) = \lambda_m \leq \lambda_{max}(M)$.

■

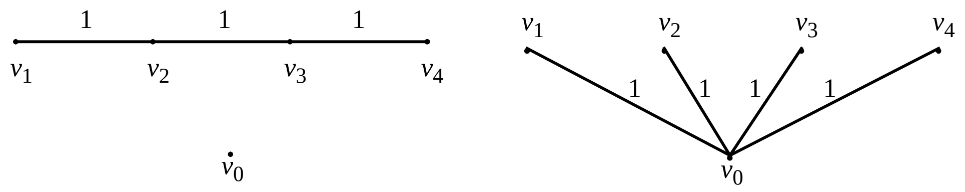
Consider bounding the largest eigenvalue of the matrix A corresponding to a simple path on 4 points with no grounding. That is, we are trying to bound $\lambda_{max}(A) = \lambda_{max}(A,I)$. The steps in determining a combinatorial bound are illustrated in Figure 7.3. The first step is to augment A to form E, and I to form J. Then, we use the combinatorial techniques developed in Chapter 4 to bound $\underline{\lambda}_{\text{max}}(E,J)$, the largest finite generalized eigenvalue of (E,J). To do this, the network corresponding to J is partitioned to support each of the edges in E. The worst support is for the central edge between nodes v_2 and v_3. This edge has conductance 1 and is supported by a path of length 2 with conductances of 1/2, yielding a support number of 4. Therefore $\lambda_{max}(A) \leq 4.0$, which in this case is identical to the Gerschgorin estimate. The actual eigenvalue determined using Matlab is 2.6180.

This example previews the technique that will be used to prove the main result in this chapter.



a)

$$A = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



b)

$$B = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad J = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix}$$

c)

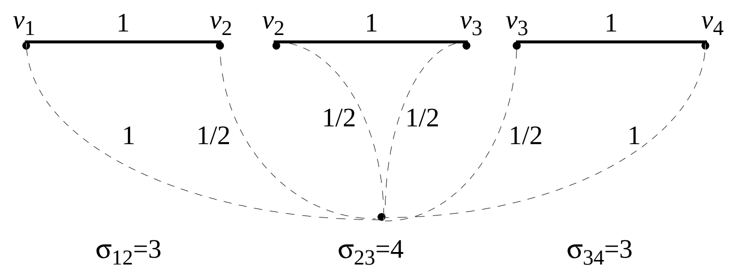


Figure 7.3: ***Bounding Eigenvalues with Augmentation and Support Analysis.***
a) *A is the matrix of a path, while I is the identity matrix.*
b) *B is the augmentation of A; J is the augmentation of I; v_0 is the ground node.*
c) *The edges of J have been partitioned to support the edges of B. The edges of B have been disconnected from each other in order to clarify the support relationships.*

7.7 ***Lemma:*** *Let A be a generalized Laplacian matrix corresponding to a connected graph. Let* $r_i = \sum_{j \neq i} |a_{ij}|$, $i = 1, \ldots, n$. *The largest eigenvalue of A,* λ_{max}, *is bounded above by*

$$\lambda_{\max} \leq \max\{a_{ii} + a_{jj} : a_{ij} \neq 0\}$$

proof:

Let A be a Laplacian matrix and B its augmentation. Let J be the augmentation of the identity matrix. Let v_i be any node in the resistive network corresponding to B. Let α_{i0} be the weight of the connection (if any) between v_i and the ground node. Let the weights of the connections from v_i to other nodes be denoted $\alpha_{i1},\ldots,\alpha_{im(i)}$. Since the graph of A is connected, every node is connected to at least one other node. Suppose that v_j is adjacent to v_i, and let $\alpha_{i1} = \alpha_{j1}$ be the weight of the edge between v_i and v_j. See Figure 7.4 for an illustration.

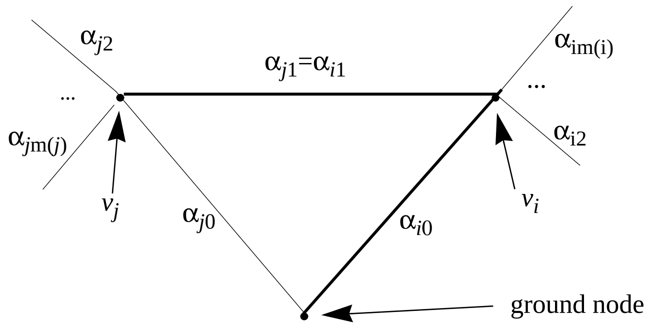


Figure 7.4: ***Labeling the weights of connections to a node.***

The network corresponding to J consists only of edges unit weight between the ground node and all the other nodes of B. Each of these connections to ground must be partitioned to provide support to the edges of B. Partition the edge from ground to node v_i into $m(i)$+1 pieces such that the piece supporting the edge of B with weight α_{ik} has itself weight γ_{ik} given by

$$\gamma_{ik} = \frac{\alpha_{ik}}{\sum_{l=0}^{m(i)} \alpha_{il}}$$

Similarly, partition the edge from ground to v_j such that each edge of weight α_{jk} is supported by an edge of weight δ_{jk} given by

$$\delta_{jk} = \frac{\alpha_{jk}}{\sum_{l=0}^{m(i)} \alpha_{jl}}$$

This weighting assigns support proportional to the product of the conductance of the edge being supported and the length of the support path.

Now, consider supporting the edge between v_i and v_j; this edge has weight $\alpha_{i1} = \alpha_{j1}$. This is supported by a path of length 2, with total path conductance of

$$c_{ij} = \frac{1}{\frac{\sum \alpha_{il}}{\alpha_{i1}} + \frac{\sum \alpha_{jl}}{\alpha_{j1}}} = \frac{\alpha_{i1}}{\sum \alpha_{il} + \sum \alpha_{jl}}$$

The last equality follows from the fact that $\alpha_{i1} = \alpha_{j1}$.

The support number is the gain factor needed to make the path conductance greater than or equal to the weight of the edge, and is therefore given by the ratio of the edge conductance to the path conductance. Therefore, the support of the edge being considered, σ_{ij}, is given by

$$\sigma_{ij} = \frac{\alpha_{i1}}{c_{ij}} = \alpha_{i1} \Big/ \left(\frac{\alpha_{i1}}{\sum \alpha_{il} + \sum \alpha_{jl}} \right) = \sum \alpha_{il} + \sum \alpha_{jl}$$

From the relationship between Laplacian matrices and resistive networks, for any node v_k,

$$a_{kk} = \sum_{l=0}^{m(i)} \alpha_{jl} .$$

Therefore, we obtain

$$\sigma_{ij} = a_{ii} + a_{jj} .$$

Similarly, from the definition of the support weighting, the support for the edge grounding node v_i, σ_{ii}, is

$$\sigma_{ii} = \alpha_{i0} / \gamma_{i0} = \alpha_{i0} \Big/ \left(\frac{\alpha_{i0}}{\sum_{l=0}^{m(i)} \alpha_{il}} \right) = \sum_{l=0}^{m(i)} \alpha_{il} = a_{ii}$$

Applying the techniques from Chapter 4, we find that $\sigma(B,J) \le \max\{\sigma_{ij}\}$. Then, by applying Lemma 4.4, we have the result for Laplacian matrices.

Finally, any generalized Laplacian can be expanded to a Laplacian and the same procedure applied. Expansion does not change any of the diagonal values, so the result holds on the original matrix.

■

Now we have the tools needed to prove the main result.[1]

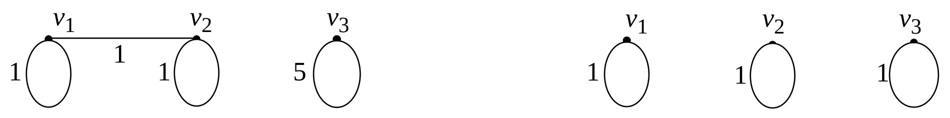
7.8 ***Theorem****: Let A be a real symmetric matrix. Let* $r_i = \sum_{j \ne i} |a_{ij}|$, $i = 1, \ldots, n$. *The largest eigenvalue of A,* λ_{max}*, is bounded above by*

$$\lambda_{\max} \le \max\{\{(a_{ii} + a_{jj}) : a_{ij} \ne 0\} \cup \{a_{kk} : a_{kl} = 0, \forall l \ne k\}\}$$

proof:

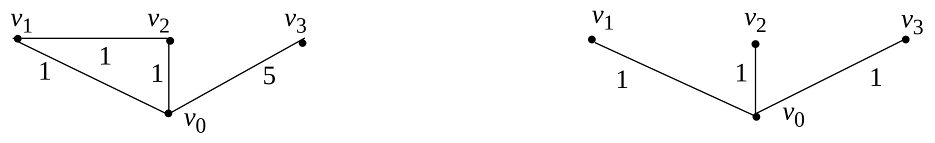
1. We would like to acknowledge and thank Dr. J. Demmel at the University of California at Berkeley who reviewed an early version of this chapter and found an error in a stronger version of this theorem.

Lemma 7.7 handled the case in which A was a generalized Laplacian corresponding to a connected graph. Therefore, consider the case when A is a generalized Laplacian, but corresponding to a graph with two or more connected components. Then A is called *reducible*, and there exists an ordering of the vertices in the graph corresponding to A such that A is block diagonal [Golub and Ortega (1993), Varga (1962)]. Each component with two or more vertices can be dealt with as in the proof of Lemma 7.7, and the same result holds. The catch comes when a component contains only a single, grounded node; this corresponds to a non-zero diagonal element a_{kk} for which all other entries in row/column k are zero. In such a case, only the grounded edge needs to be supported, and the support number is the conductance of that edge: a_{kk}. Figure 7.5 illustrates a simple example.



a)

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 5 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



b)

$$B = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 2 & 0 & -1 \\ 0 & 0 & 5 & -5 \\ -1 & -1 & -5 & 7 \end{bmatrix} \qquad J = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$

c)

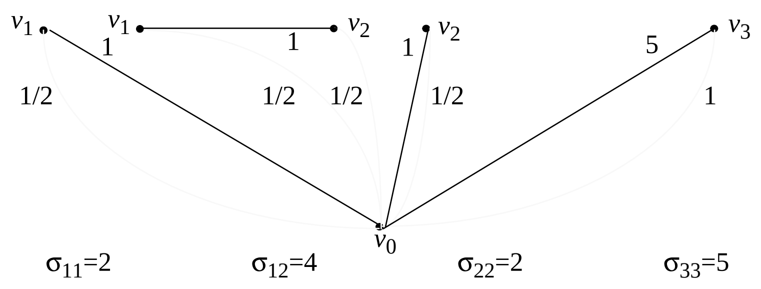


Figure 7.5: S*upport Analysis for a Two-Component Graph.*
a) *A is the matrix of a graph with two components; I is the identity matrix.*
b) *B is the augmentation of A; J is the augmentation of I.*
c) *The edges of J have been partitioned to support the edges of B. The edges of B have been disconnected to clarify the support relationships.*

Now, consider the case in which A is symmetric, but not a generalized Laplacian. Assume that A is *irreducible* (that is, the graph of A contains a single connected component). Then A must not be diagonally dominant. In this case, $A = L + D$, where L is a generalized Laplacian with the zero row/column sum property, and D is a diagonal matrix. At least one of the elements of D must be negative.

Let $\delta = max_i\{|d_{ii}|\}$, and let $B = A + \delta I$. Then B is a generalized Laplacian, and

$$\lambda_{max}(B) = \lambda_{max}(A) + \delta ,$$

By Lemma 7.7,

$$\lambda_{max}(B) \le \max\{(b_{ii} + b_{jj})/2 \ : \ b_{ij} \ne 0\} \ .$$

But, $b_{ii} = a_{ii} + \delta$, so

$$\begin{aligned}\lambda_{max}(A) &= \lambda_{max}(B) - \delta \\ &\le \max\{(b_{ii} + b_{jj})/2 \ : \ b_{ij} \ne 0\} - \delta \\ &= \max\{((a_{ii} + \delta + a_{jj} + \delta)/2) \ : \ b_{ij} \ne 0\} - \delta \\ &= \max\{((a_{ii} + a_{jj})/2 + \delta) \ : \ b_{ij} \ne 0\} - \delta \\ &= \max\{(a_{ii} + a_{jj})/2 \ : \ b_{ij} \ne 0\}\end{aligned}$$

Now, in the case that A is reducible, the argument above holds as long as all the components have more that one vertex. In the case in which one or more components has a single, grounded vertex, then essentially the same sequence of steps follows. B is a generalized Laplacian, so we have:

$$\lambda_{max}(B) \le \max\{\{(b_{ii} + b_{jj})/2 \ : \ b_{ij} \ne 0\} \cup \{b_{kk} \ : \ b_{kl} = 0, \forall l \ne k\}\}$$

As for the irreducible case, we then subtract δ from both sides, and the result follows.

■

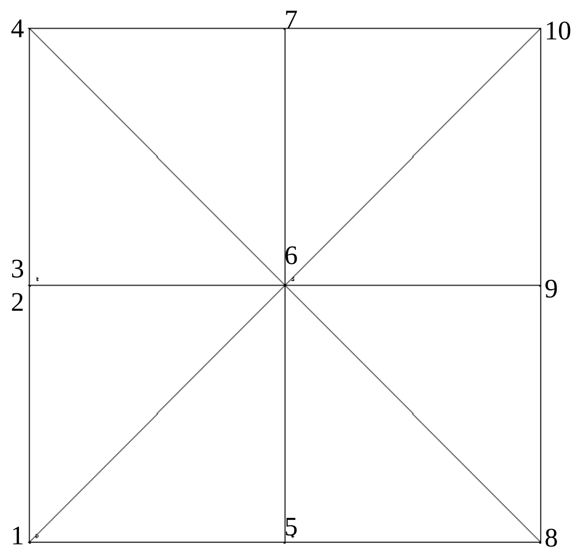
As an example of the improved bounds possible with this combinatorial estimate, consider the matrix/graph shown in Figure 7.6. The graph resulted from the finite element discretization of a cracked plate (the crack runs from the center of the left side to the center of the plate), and is the first in a sequence of mesh refinements. The matrix shown is the Laplacian of the graph in which all edges have been assigned unit weight.

- The Gerschgorin bound on the largest eigenvalue of the matrix is 18.0.
- The combinatorial bound using the formula from Theorem 7.8 is 12.0.
- The largest eigenvalue (obtained using Matlab) is 10.0.

7.3 Summary

In this chapter, we have shown how to extend the use of support trees as preconditioners to the class of generalized Laplacian matrices, which are matrices which are symmetric and diagonally dominant. The method is straightforward, involving creation of a graph of double the size of the original graph (corresponding to the expansion of the generalized Laplacian), and constructing a support tree for that graph. There may exist methods to construct support trees for generalized Laplacian that do not involve creation of the expanded graph; we have not yet researched this issue.

The class of generalized Laplacians are the largest class of matrices for which support trees can presently be constructed. The extension of support tree technology to the class of symmetric positive definite matrices is the next log-



$$\begin{bmatrix} 3 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 3 & 0 & -1 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 3 & -1 & 0 & -1 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 & 9 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 3 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & 3 \end{bmatrix}$$

Figure 7.6: ***Finite Element Mesh and Laplacian Matrix***
The graph was obtained from a finite element discretization of a cracked plate. The crack runs from the left side to the center. Nodes 2 and 3 are colocated in space, but lie on opposite sides of the crack, and are not connected. The matrix is the Laplacian of the graph with unit edges.

ical step, and remains a research issue.

In addition to the extension of the support tree technology to generalized Laplacians, we also demonstrated the application of combinatorial techniques to the problem of bounding the largest eigenvalue of a symmetric matrix. We showed that, in some cases, the combinatorial estimate can be better than an estimate made using Gerschgorin's Theorem. The combinatorial estimate has a particularly simple form, and can be easier to compute that the Gerschgorin estimate.

8
Discussion and Recommendations

This thesis presented a new approach to the parallel iterative solution of linear systems. In this chapter, we discuss the results presented in the previous chapters, and present recommendations for future work in this area.

8.1 Support Tree Conjugate Gradients

The support tree conjugate gradients (STCG) method is a variation of preconditioned conjugate gradients (PCG), and is characterized by the form of the preconditioner. Standard preconditioned conjugate gradients methods, like diagonal scaling (DSCG), incomplete Cholesky (ICCG), modified incomplete Cholesky (MICCG), or symmetric successive over-relaxation (SSOR-CG) are constructed based on the algebraic properties of the coefficient matrix. In fact, each of the standard preconditioners can be defined by a straightforward algebraic equation involving a decomposition of the coefficient matrix into additive terms. In contrast, a support tree preconditioner is defined by a construction procedure that is dependent upon the topological properties of the coefficient matrix.

STCG can only be applied when the coefficient matrix is a generalized Laplacian; that is, the coefficient matrix must be symmetric and diagonally dominant. The construction procedure reported in Chapter 3 only applies to Laplacian matrices (symmetric, diagonally dominant, with non-positive off-diagonals), although Chapter 7 reported a technique that can be used on a linear system with a coefficient matrix that is a generalized Laplacian to construct an equivalent linear system with a Laplacian coefficient matrix.

Unlike standard preconditioners, support trees were designed with parallel computation in mind. The construction of support trees is a straightforward implementation of recursive divide-and-conquer, which can be easily parallelized. The application of support trees takes advantage of the parallel efficiency obtainable with tree structures.

8.1.1 STCG performance

Three requirements for a "good" preconditioner B given a coefficient matrix A were stated in similar terms by both Axelsson and Barker (1984), and van der Vorst (1989). We can evaluate STCG based on these requirements.

1. $\kappa(B^{-1}A)$ should be significantly less than $\kappa(A)$.

In Chapter 4, we defined a model problem discretized onto an nxn mesh for which $\kappa(A) = O(n^2)$ [Guo(1990)]; we showed that $\kappa(B^{-1}A) = O(n\log n)$ for B the reduction of a support tree. In contrast, we have the following for the standard preconditioners [Guo (1990)]:

- DSCG is $O(n^2)$;
- ICCG is $O(n^2)$;
- MICCG is $O(n)$ with the optimal relaxation parameter;
- SSOR-CG is $O(n)$ with the optimal relaxation parameter.

Therefore, STCG has a better condition number than the simple DSCG and ICCG preconditioners, but not as good as the more elaborate MICCG and SSOR-CG preconditioners. However, these more elaborate preconditioners require optimal relaxation parameters to achieve the best results, and the computation of these parameters depends on algebraic properties of the coefficient matrices and can be difficult to compute.

2. The preconditioner should be easy to compute.

   Support trees are straightforward to compute using standard methods for graph partitioning. No elaborate, special purpose data structures are required.

3. The preconditioned system should be easy to solve; that is, the time required to solve the preconditioned system should be small with respect to the time required for an unpreconditioned iteration.

   Support trees are very sparse; as shown in Chapter 3, support trees can be more sparse (that is, have fewer non-zeros) than the original coefficient matrix. Moreover, the support tree provides an extremely regular data structure for efficient execution. Therefore, STCG can be implemented at least as efficiently as general versions of ICCG (recall from Chapter 3 that there is a particularly efficient implementation of ICCG applicable to matrices whose corresponding graphs have no triangles), MICCG, or SSOR-CG; additionally, the regular structure of STCG should provide more efficiency. Only DSCG is more efficient on a per iteration basis than STCG, but DSCG requires significantly more iterations.

Conditions 1 and 3 are not independent. In practice, the reduction in the number of iterations (condition 1) must be balanced against the time required to solve the preconditioned system (condition 3). A large reduction in condition number can lead to a significant reduction in the number of iterations, and justify the user of a preconditioner that requires a comparatively large amount of computation time to solve the preconditioned system. We have shown that support trees yield a significant reduction in the condition number while also requiring little time to solve the preconditioned system.

8.1.2 STCG parallel performance

STCG was designed with parallel performance in mind. The construction of a support tree is an application of a recursive divide-and-conquer process. When bisection is used, each step of construction yields two smaller, simpler subproblems. Therefore, a complete support tree for a mesh with n^2 nodes can be constructed in $O(\log n)$ parallel steps. Additionally, each partitioning step in support tree construction requires application of some graph partitioning algorithm. There is parallelism inherent in the partitioning algorithm which can also be taken advantage of.

During application of STCG, each iteration involves solving the preconditioned system. As discussed in Chapter 3, this can be done particularly efficiently in parallel by applying leaf raking and the parallel evaluation of subtrees. In general, for a mesh with n^2 nodes, STCG requires $O(\log n)$ parallel steps to solve the preconditioned system. In contrast, with diagonal ordering, ICCG, MICCG, and SSOR-CG require $O(n)$ parallel steps. Again, DSCG is the most efficient, requiring $O(1)$ diagonal steps, although requiring many more iterations.

8.2 STCG is not Multigrid

While similar in some respects to multilevel methods, STCG is not a multilevel method. The defining characteristic of a multilevel method or a multilevel preconditioner is the approximate solution of the underlying PDE at multiple levels of resolution. In contrast, STCG simply passes averaged information across the mesh using an alternate structure; no solution is computed at any level except the original.

For most multilevel methods, each grid is a discretization of the underlying PDE, and the method depends on having a nested sequence of grids. In contrast, STCG makes no claims about the underlying PDE, and multiple grids are not required. Most multilevel methods, construction is bottom up, from coarse grids to fine. That is, the development of a multilevel method requires grid refinement; an exception to this is AMG (Algebraic MultiGrid). In contrast, STCG starts with the finest grid and does not utilize a nested sequence.

The general statements above about multilevel methods do not apply to AMG. However, AMG is a complicated algorithm that applies local analysis to determine a coarse grid given an initial fine grid. AMG is applicable to the same class of matrices that STCG was initially constructed for - Laplacian matrices, but we have shown how STCG can be extended. Furthermore, we believe that STCG is a simpler algorithm.

An example helps to highlight the difference in behavior between full multigrid, STCG, and CG. All three algorithms were run on the same problem, so that the number of iterations and the behavior of the iterates could be compared.

- Figure 8.1 illustrates the qualitative convergence behavior of full multigrid on a simple example for which convergence took only 6 iterations of a complete V-cycle. The full multigrid used a single iteration of Gauss-Seidel for smoothing at each step. Notice that the shape and magnitude of the solution are nearly achieved at the first iteration. Succeeding iterations do little more than adjust the values of the iterates.

- Figure 8.2 illustrates the behavior of STCG on the same simple example. The support tree used was constructed using binary partitioning and was boundary weighted. STCG took 21 iterations to converge, but only the first 5 iterates are shown. In the case of STCG, little of the solution's final shape is exhibited in the first iterate. Interestingly, though, the mesh partitioning is visible in the rough shape of the first iterate. Succeeding iterations add both shape and value.

- illustrates the behavior of unpreconditioned CG on the same problem. CG took 28 iterations to converge, but only the first 5 iterates are shown. In the case of CG, the first iterate exhibits much of the shape of the final solution, but the values are too low. The figure shows that the iterates of CG change slowly, compared to those of the other two methods.

The experiment reported in Figures 8.1 through 8.3 help place STCG within the taxonomy of iterative methods. CG is the most local of the three methods compared. The behavior of CG shows that local information (e.g.: the shape of the curves in Figure 8.3) is accurately communicated with the first iterate, but global information such as the average magnitude of the solution values is slow to propagate. CG is easy to implement, and requires no special knowledge of the PDE or the underlying mesh. The behavior of full multigrid shows that both local information and global information is propagated rapidly. Full multigrid is the most complex of the methods to implement, and often requires information beyond simply the coefficient matrix and the forcing function. STCG lies somewhere in between full multigrid and CG, both in performance and difficulty of implementation. STCG propagates averaged global information, but smooths out local variation in the early stages of iteration. STCG is straightforward to implement, and requires only the coefficient matrix and forcing function for construction.

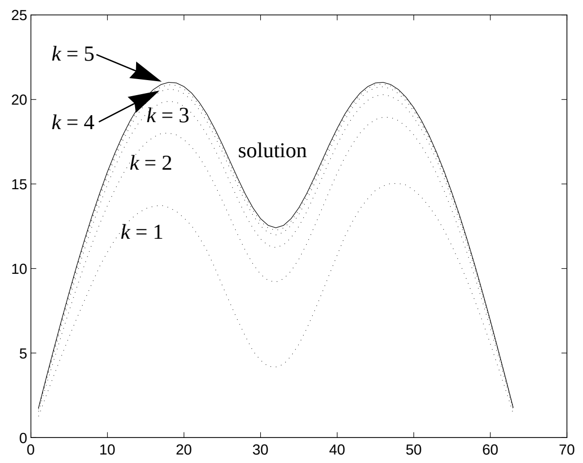


Figure 8.1: The convergence behavior of full multigrid.
The first five iterates of full multigrid are shown as dotted lines. k is the iteration number. The solution is the solid line at the top of the figure.

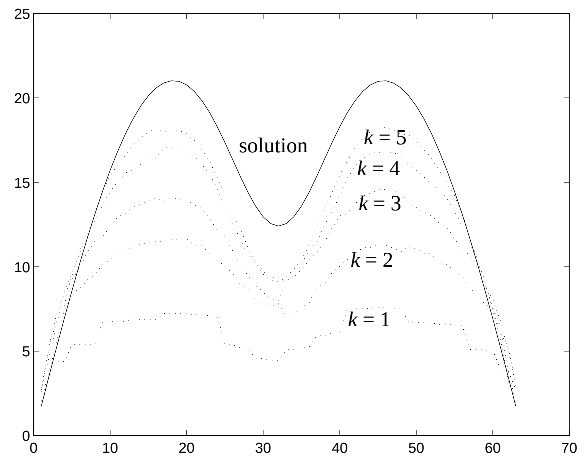


Figure 8.2: The convergence behavior of support tree conjugate gradient.
The first five iterates of support tree conjugate gradients are shown as dotted lines. The solution is the solid line at the top of the figure.

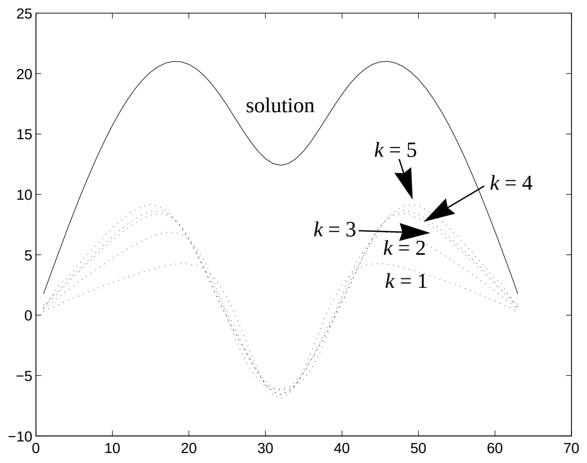


Figure 8.3: The convergence behavior of conjugate gradients.
The first five iterates of conjugate gradients are shown as dotted lines. The solution is the solid line at the top of the figure.

8.3 Recommendations for Future Work

8.3.1 Optimal support trees

In the preceding chapters, we chose simple rooted trees for the structure of our preconditioners. This topology was simply chosen because it naturally reflected the properties of the recursive divide-and-conquer approach that was used. The optimal shape for a preconditioner is an open question.

Reflecting on the similarities and differences between support tree preconditioners and multilevel preconditioners, it is likely that the optimal shape includes some cycles. Recall that support trees are primarily characterized by supporting efficient communication across a mesh. Multilevel preconditioners, on the other hand, involve solving the problem on a coarser mesh. Figure 8.4 illustrates both types of preconditioners for a simple 4x4 mesh. The support tree shown in a) is easy to construct and has excellent parallel properties, but does not converge as quickly as the multilevel preconditioner shown in b) that is difficult to construct and has poorer parallel properties.

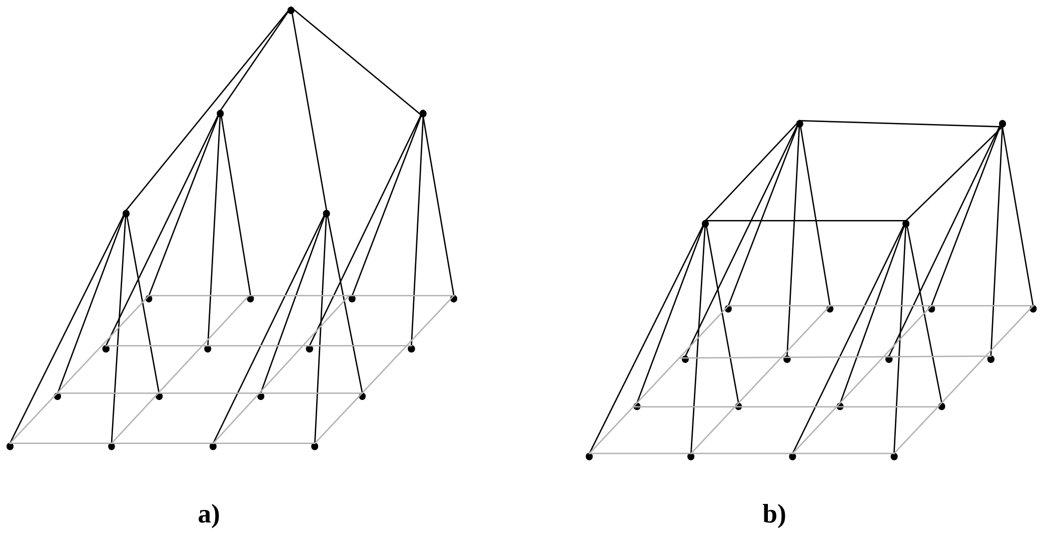


Figure 8.4: The topology of a support tree and a multilevel preconditioner.
a) Support tree preconditioner for a 4x4 mesh. b) Multilevel preconditioner for a 4x4 mesh.

Recall that, for an *n*x*n* square mesh, the condition number of the coefficient matrix is $O(n^2)$. With a support tree preconditioner, the generalized condition number drops to $O(n\log^4 n)$, while with a multilevel preconditioner, the generalized condition number is $O(1)$. Clearly, solving the equation at a coarser level (multilevel) is more powerful than just passing averaged information (support trees). On the other hand, for an arbitrary graph, it is easier to construct a support tree, given only a coefficient matrix, than it is to devise a multilevel solution.

A worthwhile research goal for the future would be investigate ways to combine the two techniques of STCG and multigrid to develop an easy to implement and powerful preconditioning technique.

8.3.2 Efficient implementation for generalized Laplacians

In Chapter 7, we showed how to construct a Laplacian matrix equivalent to a generalized Laplacian. This construction involved doubling the number of nodes in the graph, and then making connections based on the sign of the connection in the original matrix. While this is satisfactory from a theoretical point of view, it is less than satisfactory from a practical point of view to double the size of the problem.

The generalized Laplacian clearly contains all the information in the expanded Laplacian of twice the size. It should therefore be possible to analyze the generalized Laplacian and construct the support tree without ever making the transformation to the larger, equivalent, expanded Laplacian matrix.

A research goal for the future should be to focus on the mapping from a generalized Laplacian to its expansion and develop techniques to construct support trees for generalized Laplacians without ever performing the expansion step.

8.3.3 Extension to all symmetric positive definite matrices

We discussed support trees initially in the context of Laplacian matrices, which are symmetric, diagonally dominant, and have only non-positive off-diagonals. In Chapter 7, we showed how to extend the approach to generalized Laplacians which are symmetric and diagonally dominant, but may have otherwise arbitrary off-diagonals. The next logical step is to extend the methodology to all symmetric positive definite matrices.

8.3.4 Additional numerical experiments

Since STCG is a new method, more experimentation is needed to fully characterize it. The following additional numerical experiments should be performed:

- Test STCG over a larger range of problems. In particular, problems should be used that have parameters that vary across the mesh, and are defined on highly irregular meshes.

- Test STCG on multiple vector processors. In this thesis, we demonstrated the efficiency of STCG on a single vector processor of a Cray C-90. The performance improvement should be even greater if multiple vector processors are used and the support tree implementation takes advantage of both level scheduling and multiple independent subtrees.

- Test STCG on a massively parallel processor. The structure of the support tree is very regular and involves relatively little communication and should therefore be a good candidate for efficient implementation on an MPP.

- End-to-end timings of STCG should be performed. The performance analysis presented in this thesis only dealt with the performance of the iterative solution phase and ignored the timing for the construction phase. The construction of support trees requires repeated application of some graph partitioning algorithm, and the efficient parallel implementation of graph partitioning is a research topic in itself. A complete study of the performance of STCG should be conducted that includes the timings for construction using a variety of efficiently implemented graph partitioning algorithms.

8.3.5 Additional theory

The analysis of support trees involved an interesting application of graph theory to linear systems and should be continued to further explore the relationships between graph theory and linear systems.

9
References

[1] Alon, N., Seymour, P., and Thomas, R. (1990). *A separator theorem for graphs with an excluded minor and its applications*. **Proc. of the 22nd Annual ACM Symposium on Theory of Computing**, pp. 293-299.

[2] Alvarado, F. L., and Schreiber, R. (1993). *Optimal parallel solution of sparse triangular systems*. **SIAM J. Sci. Comput.** 14(2):446-460.

[3] Anderson, E., and Saad, Y. (1989). *Solving sparse triangular linear systems on parallel computers*. **Int. J. of High Speed Computing** 1(1):73-95.

[4] Arioli, M., Duff, I., and Ruiz, D. (1992). *Stopping criteria for iterative solvers*. **SIAM J. Matrix Anal. Appl.** 13(1):138-144.

[5] Ashby, S. F. (1987). *Polynomial preconditioning for conjugate gradient methods*. UIUCDCS-R-87-1355, Department of Computer Science, University of Illinois at Urbana-Champaign.

[6] Axelsson, O. (1992). *Bounds of eigenvalues of preconditioned matrices.* **SIAM J. Matrix Anal. Appl.**, 13(3):847-862.

[7] Axelsson, O. (1994). **Iterative Solution Methods**. Cambridge University Press.

[8] Axelsson, O., and Barker, V. A. (1984). **Finite Element Solution of Boundary Value Problems.** Academic Press.

[9] Axelsson, O., and Lindskog, G. (1986). *On the rate of convergence of the preconditioned conjugate gradient method*. **Numer. Math**. 48:499-523.

[10] Axelsson, O., and Vassilevski, P.S. (1989). *Algebraic multilevel preconditioning methods. I*. **Numer. Math**., 56:157-177.

[11] Axelsson, O., and Vassilevski, P.S. (1990). *Algebraic multilevel preconditioning methods. II*. **Numer. Math**., 56:157-177.

[12] Bank, R. E., and Dupont, T. F. (1981) *An optimal order process for solving finite element equations.* **Math. Comput.** 36:35-51.

[13] Blelloch, G. E. (1993). *NESL: A nested data-parallel language*. CMU-CS-93-129, School of Computer Science, Carnegie Mellon University.

[14] Blelloch, G. E. (1990). **Vector Models for Data-Parallel Computing.** MIT Press.

[15] Blelloch, G., Feldmann, A., Ghattas, O., Gilbert, J., Miller, G., O'Hallaron, D. R., Schwabe, E., Shewchuk, J., and Teng, S. -H. (1992). *Automated parallel solution of unstructured PDE problems*. **Proc. of the 1992 DAGS/PC Symposium**.

[16] Blelloch, G. E., Heroux, M. A., and Zagha, M. (1993). *Segmented operations for sparse matrix computation on vector multiprocessors.* CMU-CS-93-173, School of Computer Science, Carnegie Mellon University.

[17] Bollabas, B. (1979). **Graph Theory: An Introductory Course**. Springer-Verlag.

[18] Braess, D. and Hackbusch, W. (1983). *A new convergence proof for the multigrid method including the V-cycle*. **SIAM J. Numer. Anal.**, 20:986-975.

[19] Bramble, J. H., Pasciak, J. E., and Xu, J. (1990). *Parallel multilevel preconditioners*. **Math. Comp**. 55:1-22.

[20] Brandt, A. (1977). *Multi-level adaptive solutions to boundary-value problems*. **Math. Comp**. 31:333-390.

[21] Brandt, A., McCormick, S. F., and Ruge, J. (1982). *Algebraic multigrid (AMF) for automatic algorithm design and problem solution. A preliminary report*. Inst. Comp. Studies, Colorado State University.

[22] Briggs, W. L. (1987). **A Multigrid Tutorial.** SIAM.

[23] Chan, T. F., and Mathew, T. P. (1994). *Domain decomposition algorithms*. **Acta Numerica**, pp. 61-143.

[24] Chandra, A. K., Raghavan, P., Ruzzo, W. L., Smolensky, R., and Tiwari, P. (1989). *The electrical resistance of a graph captures its commute and cover times*. **Proceedings of the 21st Annual ACM Symposium on Theory of Computing**, pp. 574-586.

[25] Chartrand, G. (1977). **Introductory Graph Theory**. Dover.

[26] Dagum, L. (1993). *Automatic partitioning of unstructured grids into connected components.* **Proc. Supercomputing '93.**

[27] Djidjev, H. N. (1982). *On the problem of partitioning planar graphs*. **SIAM J. Alg. Disc. Meth.** 3(2):229-240.

[28] Donath, W. E. (1988). *Logic Partitioning*. in Preas, B. T., and Lorenzetti, M. J., eds., **Physical Design Automation of VLSI Systems**, pp. 65-86. Benjamin/Cummings.

[29] Donath, W. E., and Hoffman, A. J. (1972). *Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices.* **IBM Technical Disclosure Bulletin**, 15:938-944.

[30] Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. (1991). **Solving Linear Systems on Vector and Shared Memory Computers.** SIAM.

[31] Doyle, P. G., and Snell, J. L. (1984). **Random Walks and Electric Networks**. Carus Mathematical Monographs #22, Mathematical Association of America.

[32] Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). **Direct Methods for Sparse Matrices**. Clarendon Press.

[33] Duff, I. S., and Meurant, G. A. (1989). *The effect of ordering on preconditioned conjugate gradients*. **BIT** 29:635-657.

[34] Eisenstat, S. C. (1981). *Efficient implementation of a class of preconditioned conjugate gradient methods*. **SIAM J. Sci. Stat. Comput.** 2:1-4.

[35] Farhat, C., and Lesoinne, M. (1993). *Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics*. **Int. J. Num. Meth. Eng**. 36:745-764.

[36] Fiduccia, C. M., and Mattheyses, R. M. (1982). *A linear time heuristic for improving network partitions*. **Proc. 19th Design Automation Conference**, pp. 175-181.

[37] Fiedler, M. (1973). *Algebraic connectivity of graphs*. **Czechoslovak Math. J.** 23(98): 298-305.

[38] Ford, L. R., and Fulkerson, D. R. (1956). *Maximal flow through a network*. **Canad. J. Math.**, 8:399-404.

[39] Ford, L. R., and Fulkerson, D. R. (1962). **Flows in Networks**. Princton Univ. Press.

[40] Garey, M. R., and Johnson, D. S. (1979). **Computers and Intractability**. Freeman.

[41] Garey, M. R., Johnson, D. S., and Stockmeyer, L. (1976). *Some simplified NP-complete graph problems*. **Theoretical Computer Science**, pp. 237-267.

[42] Gazit, H. and Miller, G. L. (1987). *A parallel algorithm for finding a separator in planar graphs*. **Proc. of the 28th Annual Symposium on Foundations of Computer Science**, pp. 238-248.

[43] George, A., and Liu, J. W. (1981). **Computer Solution of Large Sparse Positive Definite Systems**. Prentice-Hall.

[44] Gilbert, J. R. (1980). *Graph Separator Theorems and Sparse Gaussian Elimination.* Ph. D. Thesis, Department of Computer Science, Stanford University.

[45] Gilbert, J. R., Hutchinson, J. P., and Tarjan, R. E. (1984). *A separator theorem for graphs of bounded genus*. **J. Algorithms** 5:391-407.

[46] Gilbert, J. R., and Tarjan, R. E. (1987). *The analysis of a nested dissection algorithm*. **Numer. Math**. 50(4):377-404.

[47] Golub, G. and O'Leary, D. (1989). *Some history of the conjugate gradient and Lanczos algorithms: 1948-1976*. **SIAM Rev**. 31:50-102.

[48] Golub, G., and Ortega, J. M. (1993). **Scientific Computing: An Introduction with Parallel Computing**. Academic Press.

[49] Golub, G. H., and Van Loan, C. F. (1989). **Matrix Computations**. Johns Hopkins University Press.

[50] Greenbaum, A., Li, C., and Chao, H. Z. (1989). *Comparison of linear system solvers applied to diffusion-type finite element equations*. **Numer. Math.** 56:529-546.

[51] Gremban, K. D., Miller, G. L, and Teng, S. -H. (1994). *Moments of inertia and graph separators*. **5th Annual ACM-SIAM Symposium on Discrete Algorithms**.

[52] Gremban, K. D., Miller, G. L., and Zagha, M. (1994). *Performance evaluation of a new parallel preconditioner.* CMU-CS-94-205, School of Computer Science, Carnegie Mellon University.

[53] Gremban, K. D., Miller, G. L., and Zagha, M. (1995). *Performance evaluation of a new parallel preconditioner.* **Proc. of the 9th International Parallel Processing Symposium**, pp. 65-69.

[54] Guattery, S. and Miller, G. L. (1994). *On the performance of spectral graph partitioning methods*. CMU-CS-94-228. School of Computer Science, Carnegie Mellon University.

[55] Guattery, S. and Miller, G. L. (1995). *On the performance of spectral graph partitioning methods*. **Proc. of the 6th Annual ACM/SIAM Symposium on Discrete Algorithms**.

[56] Guo, X. -Z. (1992). *Multilevel Preconditioners: Analysis, performance enhancements, and parallel algorithms.* CS-TR-2903, Department of Mathematics, University of Maryland.

[57] Gustafsson, I. (1978). *A class of first order factorizations*. **BIT**, 18:142-156.

[58] Hackbusch, W. (1994). **Iterative Solution of Large Sparse Systems of Equations**. Springer-Verlag.

[59] Hageman, L. A., and Young, D. M. (1981). **Applied Iterative Methods**. Academic Press.

[60] Hajek, B. (1988). *Cooling schedules for optimal annealing*. **Math. Oper. Res**. 13:311.

[61] Harary, F. (1969). **Graph Theory**. Addison-Wesley.

[62] Heath, M. T., Ng, E., and Peyton, B. W. (1990). *Parallel algorithms for sparse linear systems*. in **Parallel Algorithms for Matrix Computations**. SIAM.

[63] Hendrickson, B., and Leland, R. (1992). *An improved spectral graph partitioning algorithm for mapping parallel computations.* SAND92-1460, Sandia National Laboratories.

[64] Heroux, M. A., Vu, P., and Yang, C. (1991). *A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP.* **Appl. Num. Math.** 8:93-115.

[65] Hestenes, M. R., and Stiefel, E. (1952). *Methods of conjugate gradients for solving linear systems*. **J. Res. Nat. Bur. Standards B** 49:409-436.

[66] Hoffman, A. J., Martin, M. S., and Rose, D. J. (1973). *Complexity bounds for regular finite difference and finite element grids*. **SIAM J. Num. Anal**. 10:364-369.

[67] Johnson, C. (1987). **Numerical Solution of Partial Differential Equations by the Finite Element Method**. Cambridge University Press.

[68] Jordan, E. (1869). *Sur les assemblages de lignes*. **Journal Reine Angew. Math**, 70:185-190.

[69] Kernighan, B. W., and Lin, S. (1970). *An efficient heuristic procedure for partitioning graphs*. **Bell Sys. Tech. J.**, 49:291-307.

[70] Khaira, M. S., Miller, G. L., and Sheffler, T. J. (1992). *Nested Dissection: A survey and comparison of various nested dissection algorithms.* CMU-CS-92-106R, Computer Science Department, Carnegie Mellon University.

[71] Lang, K. and Rao, S. (1994). *Finding near-optimal cuts: an empirical evaluation.* **Proc. of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms.**

[72] Leighton, F. T. (1983). **Complexity Issues in VLSI**. Foundations of Computing, MIT Press.

[73] Leighton, F. T. (1992). **Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes**. Morgan Kaufmann.

[74] Leighton, F. T., and Rao, S. (1988). *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms*. **Proc. of the 29th Annual Symposium on Foundations of Computer Science**, pp. 422-431.

[75] Leiserson, C. E. (1983). **Area Efficient VLSI Computation**. Foundations of Computing, MIT Press.

[76] Lipton, R. J., Rose, D. J., and Tarjan, R. E. (1979). *Generalized nested dissection*. **SIAM J. Num. Anal**. 16:346-358.

[77] Lipton, R. J., and Tarjan, R. E. *A separator theorem for planar graphs*. **SIAM J. Appl. Math.** 36:177-189.

[78] MathWorks, Inc. (1992). **MATLAB Reference Guide**.

[79] Meijerink, J. A., and van der Vorst, H. A. (1977). *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix.* **Math. Comp.** 31:148-162.

[80] Miller, G. L. (1986). *Finding small simple cycle separators for 2-connected planar graphs*. **J. Comp. Sys. Sci.** 32(3):265-279.

[81] Miller, G. L., Teng, S. -H., Thurston, W. and Vavasis, S. A. (1992). *Automatic mesh partitioning*. **Proc. of the 1992 Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms.**

[82] Nour-Omid, B., Raefshy, A., and Lyzenga, G. (1987). *Solving finite element equations on concurrent computers*. in Noor, A. K., ed., **Parallel Computations and Their Impact on Mechanics**, pp. 209-228. The American Society of Mechanical Engineers, AMD-Vol. 86.

[83] Oswald, P. (1991). *On discrete norm estimates related to multilevel preconditioners in the finite element method*. **Proc. of the Int. Conf. Theory of Functions**.

[84] Pothen, A., Simon, H. D., and Liou, K. (1990). *Partitioning sparse matrices with eigenvectors of graphs*. **SIAM J. Matrix Anal. Appl**. 11(3):430-452.

[85] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1988). **Numerical Recipes in C: The Art of Scientific Computing**. Cambridge University Press.

[86] Press, W. H., and Teukolsky, S. A. (1991). *Multigrid methods for boundary value problems. 1.* **Computers in Physics**, Sep/Oct:514-519.

[87] Rao, S. (1987). Finding near optimal separators in planar graphs. **Proc. of the 28th Annual Symposium on Foundations of Computer Science**, pp. 225-237.

[88] Reid, J. K. (1971). *On the method of conjugate gradients for the solution of large sparse systems of equations*. in Reid, J. K. ed., **Large Sparse Sets of Linear Equations**, pp. 231-254. Academic Press.

[89] Reid-Miller, M., Miller, G. L., and Modugno, F. (1993). *List ranking and parallel tree contraction*. in Reif, J., ed., **Synthesis of Parallel Algorithms**, Morgan Kaufmann.

[90] Shahrokhi, F. and Matula, D. W. (1990). *The maximum concurrent flow problem*. **J. Association of Computing Machinery** 37(2):318-334.

[91] Simon, H. D. (1991). *Partitioning of unstructured problems for parallel processing*. **Comp. Sys. in Eng**. 2(2/3):135-148.

[92] Stuben, K. (1983). *Algebraic multigrid (AMG): Experiences and comparisons*. **Appl. Math. Comp**. 13:419-451.

[93] Tarjan, R. E. (1983). **Data Structures and Network Algorithms**. SIAM.

[94] Teng, S. -H. (1991). *Points, Spheres, and Separators: A unified geometric approach to graph partitioning*. CMU-CS-91-184, School of Computer Science, Carnegie Mellon University.

[95] van der Vorst, H. A. (1989a). *ICCG and related methods for 3D problems on vector computers.* **Comp. Physics Comm.** 53:223-235.

[96] van der Vorst, H. A. (1989b). *High performance preconditioning*. **SIAM J. Sci. Stat. Comput.** 10(6):1174-1185.

[97] Varga, R. (1962). **Matrix Iterative Analysis**. Prentice-Hall.

[98] Williams, R. D. (1991). *Performance of dynamic load balancing algorithms for unstructured mesh calculations*. **Concurrency: Practice and Experience**, 3(5):457-481.