PRECONDITIONED ITERATIVE METHODS FOR HIGHLY SPARSE, NONSYMMETRIC, UNSTRUCTURED LINEAR ALGEBRA PROBLEMS

by

William D. McQuain

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Layne T. Watson

Christopher A. Beattie

Calvin J. Ribbens

August, 1992

Blacksburg, Virginia

PRECONDITIONED ITERATIVE METHODS FOR HIGHLY SPARSE, NONSYMMETRIC, UNSTRUCTURED LINEAR ALGEBRA PROBLEMS

by

William D. McQuain

Committee Chairman: Layne T. Watson

Computer Science

(ABSTRACT)

A number of significant problems require the solution of a system of linear equations $Ax = b$ in which A is large, highly sparse, nonsymmetric, and unstructured. Several iterative methods which are applicable to nonsymmetric and indefinite problems are applied to a suite of test problems derived from simulations of actual bipolar circuits and to a viscous flow problem.

Methods tested include Craig's method, GMRES(k), BiCGSTAB, QMR, KACZ (a row-projection method) and LSQR. The convergence rates of these methods may be improved by use of a suitable preconditioner. Several such techniques are considered, including incomplete LU factorization (ILU), sparse submatrix ILU, and ILU allowing restricted fill in bands or blocks. Timings and convergence statistics are given for each iterative method and preconditioner.

ACKNOWLEDGEMENTS.

Dr. Layne Watson and Dr. Calvin Ribbens gave generously of their time and knowledge, both in guiding the research reported in this thesis and in criticizing the presentation of that work. Dr. Chris Beattie provided an able introduction to the field of numerical analysis, and is largely responsible for my association with the Parallel Computation Laboratory.

In addition to the members of my committee, I must thank Dr. Robert Melville of AT & T Bell Labs for his participation in this work. The suite of circuit simulation test problems could not have been obtained without his efforts. Moreover, he suggested several fruitful lines of attack on those problems.

Also, I would like to acknowledge the assistance of Dr. Henk van der Vorst, not only for providing both a preprint of his article and a copy of his code, but also for correspondence that materially aided the completion of this work. Dr. Roland Freund and Dr. Noel Nachtigal supplied their implementation of QMR, and numerous, invaluable technical reports. Dr. Homer Walker shared useful insights based on his experience with GMRES(k).

Finally I must thank my wife, Margaret, and my daughters for their understanding during this work. Worthwhile research frequently carries personal costs, and cannot easily be done without the support of ones family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION.

A number of significant problems require the solution of a consistent system of linear equations $Ax = b$ where A is large, highly sparse, nonsymmetric, and unstructured. This thesis considers the effectiveness of a variety of iterative methods, in combination with several preconditioning schemes, when applied to two such problems.

The need to solve large, highly sparse systems of linear equations led to an early recognition of the desirability of fast, efficient iterative algorithms for that purpose. Many such algorithms guarantee convergence only under restrictive hypotheses, such as that A be positive real or symmetric and positive definite. The iterative methods considered here impose no such restrictions, although the convergence of each method may improve if such conditions are met.

The use of preconditioning can accelerate the convergence of an iterative method significantly. The use of the incomplete LU factorization of A as a preconditioner has been examined in a variety of contexts [19]. The experiments reported in this thesis consider the ILU preconditioner and a number of its variations.

Theoretical results concerning the convergence of these iterative methods are often poorly understood or simply unhelpful. It is, therefore, useful to examine the performance of the methods in question on real problems. Two such examples are considered here. This thesis reports considerable experimentation with iterative methods on linear systems arising in circuit simulation, and a more limited examination of the performance of selected iterative methods on a problem in computational fluid dynamics.

The cost and difficulty of producing a prototype of a proposed design for an integrated circuit provide substantial motivation for the development of accurate, efficient computer simulations of such designs. The mathematical models for common components of such circuits are nonlinear, and so the simulation of such integrated circuits requires solving large systems of nonlinear equations $F(x) = 0$, where $F : E^n \rightarrow E^n$ is C^2. Algorithms that solve such nonlinear systems through the use of an artificial-parameter homotopy mapping have been studied for some time [46]. Under reasonable hypotheses these algorithms are guaranteed to be convergent for almost all starting points [35].

The curve tracking that is inherent to homotopy algorithms requires solving a rectangular linear system whose coefficient matrix is the Jacobian matrix of the homotopy. For many applications, the Jacobian matrix of F is symmetric and positive definite, or nearly so. It is possible to take advantage of these properties when implementing the homotopy algorithm. However, for the DC operating point problem in circuit simulation, the Jacobian matrix of F will usually be nonsymmetric, indefinite, sparse, and unstructured. A detailed description of the DC operating point problem, and the justification of the application of homotopy methods to it, are given by Melville et al. [24], [35], [36].

The analysis of a steady recirculating flow induced within a cavity provides another source of linear systems for which A is large, sparse and unstructured. The numerical computation of such a flow within an equilateral triangular cavity is considered. Such a flow is modeled by a fourth-order nonlinear partial differential equation, which may be solved by a Newton-like iteration. That, in turn, requires the solution of a linear system for which A is large, sparse, indefinite, and nonsymmetric, although symmetrically structured. A detailed description of the problem may be found in [27], and is summarized in Chapter 8.

Since the linear systems derived from circuit simulation problems arise in the context of homotopy curve tracking, Chapter 2 gives a brief summary of globally convergent homotopy theory. The various iterative methods are described in detail in Chapter 3, followed by the preconditioning strategies in Chapter 4. Chapter 5 presents a discussion of the numerical analysis of circuit simulation, and Chapter 6 describes the suite of circuit-simulation-based test problems to which the iterative methods were applied. Numerical results are given and interpreted in Chapter 7. The problem of viscous flow in a triangular cavity is described in Chapter 8, and numerical results obtained by applying selected iterative methods to such a problem are presented in Chapter 9. Conclusions follow in Chapter 10.

2. GLOBALLY CONVERGENT HOMOTOPY ALGORITHMS.

Consider the problem of solving a nonlinear system of equations

$$F(x) = 0, \tag{1}$$

where $F : E^n \to E^n$ is a C^2 map defined on real n-dimensional Euclidean space E^n. Equation (1) may be solved by use of a suitable *homotopy*, a continuous mapping $H(\lambda, x)$ such that $H(0, x) = 0$ is easily solved and $H(1, x) = F(x)$. Given a solution of $H(0, x)$ as a starting point, a homotopy algorithm will attempt to track a curve in the zero set of $H(\lambda, x)$, terminating at a solution of $F(x) = 0$ when $\lambda = 1$. A globally convergent probability-one homotopy algorithm is based upon the construction of a homotopy whose zero curves are well behaved and reach a solution for almost all starting points. Such homotopies are easy to construct for Brouwer fixed point problems.

Let B be the closed unit ball in E^n, and let $f : B \to B$ be a C^2 map. The Brouwer fixed point problem is to solve $x = f(x)$. Define $\rho_a : [0, 1) \times B \to E^n$ by

$$\rho_a(\lambda, x) = \lambda(x - f(x)) + (1 - \lambda)(x - a),$$

where a is some point in the interior of B. The fundamental result [4] is that for almost all a in the interior of B, there is a zero curve $\gamma \subset [0, 1) \times B$ of ρ_a, along which the Jacobian matrix $D\rho_a(\lambda, x)$ has rank n, emanating from $(0, a)$, and reaching a point $(1, \bar{x})$, where $\bar{x}$ is a fixed point of f. Thus with probability one, picking a starting point $a \in \text{int}B$ and following γ leads to a fixed point $\bar{x}$ of f. An important distinction between standard continuation and modern probability-one homotopy algorithms is that, for the latter, λ is not necessarily monotonically increasing along γ. Indeed, part of the power of probability-one homotopy algorithms derives from the lack of a monotonicity requirement for λ.

The zero-finding problem (1) is more complicated. Suppose that there exists a C^2 map

$$\rho : E^m \times [0, 1) \times E^n \to E^n$$

such that

(a) the $n \times (m+1+n)$ Jacobian matrix $D\rho(a,\lambda,x)$ has rank n on the set

$$\rho^{-1}(0) = \{\,(a,\lambda,x) \mid a \in E^m, 0 \le \lambda < 1, x \in E^n, \rho(a,\lambda,x) = 0\,\},$$

and for any fixed $a \in E^m$, letting $\rho_a(\lambda,x) = \rho(a,\lambda,x)$,

(b) $\rho_a(0,x) = \rho(a,0,x) = 0$ has a unique solution x_0,

(c) $\rho_a(1,x) = F(x)$,

(d) $\rho_a^{-1}(0)$ is bounded.

Then for almost all $a \in E^m$ there exists a zero curve γ of ρ_a along which the Jacobian matrix $D\rho_a$ has rank n, emanating from $(0,x_0)$, and reaching a zero $\bar{x}$ of F at $\lambda = 1$. γ does not intersect itself and is disjoint from any other zeros of ρ_a. The globally convergent homotopy algorithm is to pick $a \in E^m$ (which uniquely determines x_0), and then track the homotopy zero curve γ starting at $(0,x_0)$ until the point $(1,\bar{x})$ is reached.

There are many different algorithms for tracking the zero curve γ; the mathematical software package HOMPACK [43], [45] supports three such algorithms: ordinary differential equation-based, normal flow, and augmented Jacobian matrix. Small dense and large sparse Jacobian matrices require substantially different algorithms. For the circuit simulation problems considered in this thesis, the Jacobian matrix $DF(x)$ is an invertible, nonsymmetric $n \times n$ matrix, possibly with diagonal blocks, but with largely unstructured nonzeros outside those blocks.

For practical computational reasons, it is convenient to reverse the order of λ and x, and write $\rho_a(x,\lambda)$. In this case the $n \times (n+1)$ Jacobian matrix $D\rho_a(x,\lambda)$ has a dense last column and the sparse nonsymmetric $DF(x)$ as its leading $n \times n$ submatrix. If $F(x)$ is C^2, a is such that the Jacobian matrix $D\rho_a(x,\lambda)$ has full rank along γ, and γ is bounded, then the zero curve γ is C^1 and can be parameterized with respect to arc length s. Moreover, the unit tangent vector $(dx/ds, d\lambda/ds)$ to γ lies in the kernel of $D\rho_a$. Since curve tracking algorithms require tangent vectors to γ, it is necessary to compute the one-dimensional kernel of $D\rho_a$ at points along γ.

3. ITERATIVE METHODS FOR NONSYMMETRIC INVERTIBLE SYSTEMS.

Since the Jacobian matrix is $n \times (n+1)$, the rectangular linear system $[D\rho_a(x,\lambda)]z = 0$ is converted to an equivalent square linear system of rank $N = n+1$:

$$Ax = \begin{pmatrix} D_x\rho_a & D_\lambda\rho_a \\ w^t & d \end{pmatrix} x = \begin{pmatrix} 0 \\ c \end{pmatrix}, \tag{2}$$

by augmenting $D\rho_a$ with an additional row. The row $(w^t \quad d)$ was taken to be a standard basis vector e_k^t, where k was the index of the largest magnitude component of the unit tangent vector $\bar{y}$ to γ found at the previous step along γ. Other choices are possible, such as $(w^t \quad d) = \bar{y}^t$ or using $w = (D_\lambda\rho_a)$ and choosing d so as to make A nonsingular. All these choices were tested. Taking e_k^t for the augmenting row maintained the high sparsity of the Jacobian matrix, and yielded lower execution times, although slightly higher iteration counts. In the numerical experiments reported below, e_k^t was used as the augmenting row, and c was taken to be the max norm of the previous unit tangent vector to γ.

This thesis is primarily concerned with solving the linear system (2). The coefficient matrix A will be invertible, nonsymmetric, unstructured, and highly sparse. Available iterative methods offer a number of advantages when dealing with a large, sparse, unstructured, linear system. The coefficient matrix A is usually needed only to perform a few matrix-vector multiplications at each iteration, and that is generally cheap for a sparse problem. In addition, the iterative methods considered avoid generating any matrix fill-in, and thus require the storage of only a small number of auxiliary vectors at each iteration. Some of the leading iterative methods applicable to our class of problems are described in the following sections.

Let Q be an invertible matrix of the same size as A; then the linear system $Ax = b$ is equivalent to

$$\tilde{A}x = Q^{-1}Ax = Q^{-1}b = \tilde{b}. \tag{3}$$

The use of such an auxiliary matrix is known as *preconditioning*. Ideally, the preconditioned linear system (3) requires less computational effort to solve than does $Ax = b$, due to a reduction in the number of iterations required to achieve convergence. The choice of an effective preconditioning matrix Q is often tied to some special structure or other property of the coefficient matrix A. Since the Jacobian matrices arising in the circuit simulation problems considered here lack symmetry and are almost certainly not positive definite, the selection of an effective preconditioner is nontrivial. Several choices are discussed in Chapter 4.

3.1 Craig's method.

Craig's method [18] applies the conjugate gradient algorithm to the problem

$$AA^t z = b, \qquad x = A^t z,$$

without using either z or AA^t directly. Given a starting point x_0, at the k-th iteration Craig's method determines x_k so that $\|e_k\|_2 = \|x - x_k\|_2$ is minimized over the translated space

$$x_0 + \langle A^t r_0, A^t(AA^t) r_0, \cdots, A^t(AA^t)^{k-1} r_0 \rangle,$$

or equivalently

$$e_k \perp \langle A^t r_0, A^t(AA^t) r_0, \cdots, A^t(AA^t)^{k-1} r_0 \rangle.$$

The error norm $\|e_k\|_2$ is guaranteed to decrease at each iteration, so progress is assured. However, the rate of convergence of Craig's method depends upon the condition number of the matrix AA^t, and for that reason an effective preconditioner is desirable.

Craig's method (with preconditioning matrix Q) is:

choose x_0, Q

set $r_0 = b - Ax_0$;

$\tilde{r}_0 = Q^{-1} r_0$;

$p_0 = A^t Q^{-t} \tilde{r}_0$;

for $i = 0, 1, \ldots$ **until** convergence **do**

begin

$a_i = \dfrac{(\tilde{r}_i, \tilde{r}_i)}{(p_i, p_i)}$;

$x_{i+1} = x_i + a_i p_i$;

$\tilde{r}_{i+1} = \tilde{r}_i - a_i Q^{-1} A p_i$;

$b_i = \dfrac{(\tilde{r}_{i+1}, \tilde{r}_{i+1})}{(\tilde{r}_i, \tilde{r}_i)}$;

$p_{i+1} = A^t Q^{-t} \tilde{r}_{i+1} + b_i p_i$;

end

With preconditioning, Craig's method requires, at minimum, storage of 5 vectors of length N, and each iteration requires two preconditioning solves, two matrix-vector products, two inner products and three SAXPY operations.

3.2 GMRES.

At the k-th iteration, GMRES [30] computes x_k in the translated space

$$x_0 + \langle r_0, Ar_0, \cdots, A^{k-1}r_0 \rangle$$

to minimize the residual norm $\|r_k\|_2 = \|b - Ax_k\|_2$, or equivalently to guarantee

$$r_k \perp \langle Ar_0, A^2r_0, \cdots, A^kr_0 \rangle.$$

In contrast to Craig's method, the rate of convergence of GMRES does not depend on the condition number of A. Let P_k be the space of complex polynomials of degree k or less, and for any bounded set S of complex numbers and $p_k \in P_k$, define $\|p_k(z)\|_S = \sup_{z \in S} |p_k(z)|$. Let Λ_ϵ be the set of *ϵ-pseudo-eigenvalues* of A: all complex numbers z which are eigenvalues of some matrix $A + E$ with $\|E\| \leq \epsilon$. Let L be the arc length of the boundary of Λ_ϵ. For arbitrary invertible A and $\epsilon > 0$, the residual norms must satisfy [25]:

$$\frac{\|r_k\|_2}{\|r_0\|_2} \leq \frac{L}{2\pi\epsilon} \inf_{\substack{p_k \in P_k \\ p_k(0)=1}} \|p_k(z)\|_{\Lambda_\epsilon}.$$

Speaking loosely then, the rate of convergence of GMRES depends on the spectrum of A. Also in contrast to Craig's method, the reduction of the residual norms is not necessarily strictly monotonic.

GMRES (with preconditioning matrix Q) is:

choose x_0, Q

set $r_0 = b - Ax_0$;

$\tilde{r}_0 = Q^{-1}r_0$;

$v_1 = \dfrac{\tilde{r}_0}{\|\tilde{r}_0\|_2}$;

for $m = 1, 2, \ldots$ **until** convergence **do**

begin

for $j = 1$ **to** m **do**

$h_{j,m} = (Q^{-1}Av_m, v_j)$;

$\tilde{v}_{m+1} = Q^{-1}Av_m - \sum_{j=1}^{m} h_{j,m}v_j;$

$h_{m+1,m} = \|\tilde{v}_{m+1}\|_2;$

$v_{m+1} = \tilde{v}_{m+1} \ / \ h_{m+1,m};$

Find y_m to minimize $\left\| \|\tilde{r}_0\|_2 e_1 - \bar{H}_m y \right\|_2$ where $\bar{H}_m$ is described in [30];

$x_m = x_0 + \sum_{j=1}^{m} (y_m)_j v_j;$

end

The m-th iteration of GMRES requires one preconditioning solve, one matrix-vector product, a least-squares solution and $\mathcal{O}(m)$ SAXPY operations and inner products. Unfortunately, GMRES also requires the retention of a potentially large number of vectors of length N. In order to control the storage requirements, the algorithm is frequently used in a truncated or restarted form, GMRES(k), in which the inner loop is limited to k iterations, for some $k \ll N$, and the algorithm is restarted unless the residual norm has been reduced to an acceptable size. The price of this restarting is that the residual norms may not converge to zero, but may stagnate at some positive number.

3.3 BiCGSTAB.

At the k-th iteration, the biconjugate gradient method BiCG [10], [25] computes x_k in the same translated space as GMRES, but does not attempt to choose x_k so as to minimize the residual norm. Instead, x_k is chosen to satisfy the orthogonality condition

$$r_k \perp \langle \tilde{r}_0, A^t\tilde{r}_0, \cdots, (A^t)^{k-1}\tilde{r}_0 \rangle, \tag{4}$$

where $\tilde{r}_0$ is chosen so that $(r_0, \tilde{r}_0) \neq 0$. BiCG cannot achieve convergence in fewer iterations than GMRES, but may require less time since BiCG iterations are significantly cheaper than those of GMRES. The BiCG iterates are computed using three-term recurrences, and each iteration of BiCG requires a constant number of vector operations and a fixed amount of storage, in contrast to GMRES. The conjugate gradients squared (CGS) algorithm [34] reorganizes the BiCG algorithm to eliminate multiplications involving A^t and increase the rate of convergence by up to a factor of 2.

Since BiCG and CGS do not minimize the residual norm on each iteration, their convergence can be irregular [25], [37]. BiCGSTAB [37] is a recent modification of the BiCG algorithm that attempts to achieve both faster and smoother convergence than BiCG. The iterates in the BiCGSTAB method are also obtained with three-term recurrences, so the storage and complexity advantages of BiCG are not lost.

BiCGSTAB is:

choose x_0

set $r_0 = b - Ax_0$;

$\rho_0 = \alpha = \omega_0 = 1$;

$v_0 = p_0 = 0$;

choose $\tilde{r}_0$ such that $(r_0, \tilde{r}_0) \neq 0$;

for $i = 1, 2, 3, \ldots$ **until** convergence **do**

begin

$\rho_i = (\tilde{r}_0, r_{i-1})$; $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$;

$p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$;

$v_i = Ap_i$; $\alpha = \rho_i/(\tilde{r}_0, v_i)$;

$s = r_{i-1} - \alpha v_i$; $t = As$;

$\omega_i = (t, s)/(t, t)$;

$x_i = x_{i-1} + \alpha p_i + \omega_i s$;

if x_i is accurate enough then quit;

$r_i = s - \omega_i t$;

end

Each iteration of BiCGSTAB requires two matrix-vector products, four inner products and five SAXPY operations. If preconditioning is used, then two preconditioning solves are also required.

3.4 QMR.

The *quasi-minimal residual* method (QMR) [13] is based on a modification of the classical nonsymmetric Lanczos algorithm [22]. Given two starting vectors, v_1 and w_1, the Lanczos algorithm uses three-term recurrences to generate sequences $\{v_i\}_{i=1}^{L}$ and $\{w_i\}_{i=1}^{L}$ of vectors satisfying, for $m = 1, \ldots, L$,

$$\text{span}\{v_1, \ldots, v_m\} = \langle v_1, Av_1, \cdots, A^{m-1}v_1 \rangle,$$

$$\text{span}\{w_1, \ldots, w_m\} = \langle w_1, A^t w_1, \cdots, (A^t)^{m-1} w_1 \rangle,$$

and

$$w_i^t v_j = d_i \delta_{ij}, \text{with } d_i \neq 0, \text{ for all } i, j = 1, \ldots, L, \tag{5}$$

where δ_{ij} is the Kronecker delta. It is possible that the classical Lanczos algorithm can break down, where v_m and w_m are orthogonal and nonzero. The look-ahead Lanczos [11] algorithm attempts to avoid such failure by generating different sequences of vectors v_i and w_i, imposing a block bi-orthogonality condition in place of (5). These look-ahead Lanczos vectors also satisfy three-term recurrences, so the computational cost is comparable to that of the classical Lanczos algorithm.

At the k-th iteration, QMR computes an iterate x_k in the same translated space as GMRES. If the initial residual r_0 is taken as one of the starting vectors v_1 for the look-ahead Lanczos method, then the residual at the k-th iteration satisfies

$$r_k = V^{(k+1)} \left(\|r_0\|_2 e_1 - H_e^{(k)} z \right), \tag{6}$$

where the columns of the matrix $V^{(k+1)}$ are the right (look-ahead) Lanczos vectors $v_1, v_2, \ldots, v_{k+1}$, and $H_e^{(k)}$ is a $k \times k$ block tridiagonal matrix augmented with a row of the form ρe_k^t. In equation (6), the vector z is the unique minimizer of

$$\left\| \|r_0\|_2 e_1 - H_e^{(k)} z \right\|_2, \tag{12}$$

which can be found with considerably less work than would be needed to minimize the residual norm, since the matrix $V^{(k+1)}$ will not usually be unitary. This choice of z gives

an r_k minimal with respect to the norm $\|r\| = \|\alpha\|_2$, $r = V^{(k+1)}\alpha$, and satisfying (4) in a generalized sense. The next iterate is then given by

$$x_{k+1} = x_0 + V^{(k+1)}z.$$

3.5 LSQR.

The algorithm LSQR [26] uses a bidiagonalization scheme due to Golub and Kahan [15] to solve both least squares problems and consistent systems of linear equations. At the k-th iteration, LSQR computes x_k in the translated space

$$x_0 + \langle A^t r_0, (A^t A)A^t r_0, \cdots, (A^t A)^{k-1} A^t r_0 \rangle$$

to minimize the residual norm $\|r_k\|_2 = \|b - Ax_k\|_2$. The iterates x_k are analytically the same as those produced when the conjugate gradient iteration is applied to the normal equations, and the residual norm is guaranteed to decrease monotonically.

At each iteration, the residual satisfies the equation

$$r_k = U^{(k+1)} \left(\|r_0\|_2 e_1 - B^{(k)} z \right),$$

where $U^{(k+1)}$ is orthogonal and $B^{(k)}$ is bidiagonal. Thus the residual norm may be minimized by finding z to minimize

$$\left\| \|r_0\|_2 e_1 - B^{(k)} z \right\|_2 .$$

The special form of $B^{(k)}$ allows the computation of the next iterate x_{k+1} as an update of x_k, requiring only two matrix-vector products, four SAXPY operations, and two vector norm calculations.

A description of the LSQR algorithm follows. The scalars α_i and β_i are chosen to normalize the corresponding vectors v_i and u_i. So the assignment $\beta_1 u_1 = b$ implies the calculations $\beta_1 = \|b\|_2$ and $u_1 = (1/\beta_1)b$.

set $x_0 = 0$; $\quad \beta_1 u_1 = b$; $\quad \alpha_1 v_1 = A^t u_1$;

$w_1 = v_1$; $\quad \bar{\phi}_1 = \beta_1$; $\quad \bar{\rho}_1 = \alpha_1$;

for $i = 1, 2, 3, \ldots$ **until** convergence **do**

begin

$$\beta_{i+1}u_{i+1} = Av_i - \alpha_i u_i; \qquad \alpha_{i+1}v_{i+1} = A^t u_{i+1} - \beta_{i+1}v_i;$$

$$\rho_i = (\bar{\rho}_i^2 + \beta_{i+1}^2)^{1/2}; \qquad c_i = \bar{\rho}_i/\rho_i; \qquad s_i = \beta_{i+1}/\rho_i;$$

$$\theta_{i+1} = s_i\alpha_{i+1}; \qquad \bar{\rho}_{i+1} = -c_i\alpha_{i+1};$$

$$\phi_i = c_i\bar{\phi}_i; \qquad \bar{\phi}_{i+1} = s_i\bar{\phi}_i;$$

$$x_i = x_{i-1} + (\phi_i/\rho_i)w_i; \qquad w_{i+1} = v_{i+1} - (\theta_{i+1}/\rho_i)w_i;$$

end

3.6 KACZ.

There are a number of iterative projection methods which are suitable for large, sparse, nonsymmetric linear systems [2]. Kaczmarz [20] proposed an iterative method in which each equation is viewed as a hyperplane so that the solution is simply the point of intersection of the hyperplanes. The initial guess is then projected onto the first hyperplane, the resulting point is projected onto the second hyperplane, and so on. This approach applies equally well if the hyperplanes are defined by blocks of equations. Partition A into m blocks of rows as

$$A^t = [A_1, A_2, \ldots, A_m],$$

and partition b conformally. The orthogonal projection of a vector x onto the range of A_i is given by $P_i x = A_i(A_i^t A_i)^{-1}A_i^t x$. Let $Q_u = (I - P_m)\cdots(I - P_1)$, $\hat{b}_i = A_i(A_i^t A_i)^{-1}b_i$ for $1 \leq i \leq m$, and

$$b_u = \hat{b}_m + (I - P_m)\hat{b}_{m-1} + \cdots + (I - P_m)\cdots(I - P_2)\hat{b}_1.$$

Then for $k \geq 0$ the Kaczmarz iterates are given by

$$x_{k+1} = Q_u x_k + b_u.$$

This method will converge for any system with nonzero rows, even if the system is rectangular, singular or inconsistent. Unfortunately the rate of convergence depends on the spectral radius of the iteration matrix Q_u, and may be arbitrarily slow. In order to improve the rate of

convergence, Björck and Elfving [1] proposed the row projection method KACZ, in which the forward sweep through the rows of A is followed with a backward sweep, effectively symmetrizing the iteration matrix. The general formulation of the KACZ method involves a relaxation parameter ω; for a number of reasons the best choice for ω is generally 1, and that value was used in the experiments reported in this thesis. A full discussion of this method may be found in [2].

Let $AA^t = L + D + L^t$, where L is block lower triangular and D is block diagonal. Let $\bar{b} = A^t(D+L)^{-t}D(D+L)^{-1}b$ and $Q = (I-P_1)(I-P_2)\cdots(I-P_m)^2\cdots(I-P_2)(I-P_1)$. Then for $k \geq 0$ the KACZ iterates are given by

$$x_{k+1} = Qx_k + \bar{b}.$$

3.7 Storage Requirements.

The suitability of an iterative method for a particular problem depends not only upon the number of iterations required for convergence but also upon the amount of storage the method requires. Each of the iterative methods described in this thesis requires storage for the coefficient matrix A and the right hand side b. If the sparse storage scheme described in Chapter 6 is used, this amounts to $R+N$ reals and $R+N+1$ integer indices, where N is the dimension of A and R is the number of nonzeros in A.

There is considerable variation in the amount of additional storage required by these iterative methods. Since storage of arrays dominates total storage, the discussion here is limited to the need for local array storage. Table 1 shows the storage requirement for each method. For GMRES(k), total storage depends not only upon N but also upon k. For QMR, the block-size limit m and the iteration limit *itlim* are used to dimension internal arrays, and the storage needed by QMR is approximately $(3m+6)N + 5m^2 + 19m + itlim$. The value shown in Table 1 for QMR is based on the values $m = 5$ and $itlim = 5N$, which were used in the experiments reported in this thesis. For KACZ, the storage requirement depends in part upon the size of the block rows; in Table 1, it is assumed that each block row of A contains N/m rows.

A comparison indicates that if limited storage is a major concern then Craig's method, LSQR and BiCGSTAB have an advantage. Both GMRES(k) and QMR may require far larger amounts of storage. Moreover, the storage requirements of the latter methods are difficult to anticipate in practical use.

TABLE 1. COMPARISON OF STORAGE COSTS

Method	Storage
Craig's	$4N$
LSQR	$5N$
GMRES(k)	$(k+2)N+k^2+4k$
BiCGSTAB	$7N$
QMR	$26N$
KACZ	$4N+N^2/m$

4. PRECONDITIONING STRATEGIES.

The rate of convergence of the iterative methods considered here depends upon the spectrum and condition number of the coefficient matrix. For the problems in the test suite, many of the Jacobian matrices along the homotopy zero curve γ have condition numbers on the order of 10^6 to 10^9. Moreover, many of the Jacobian matrices have a significant number of eigenvalues far from one. As a result, the performance of each of the methods would be expected to suffer. It may be possible to improve the spectrum or conditioning through use of a judiciously chosen preconditioning matrix Q.

The preconditioned matrix $Q^{-1}A$ in (3) is not usually formed explicitly; the sparsity of A provides no guarantee that $Q^{-1}A$ will not be relatively dense. Thus the extra work required for preconditioning lies in the computation of matrix-vector products involving Q^{-1}, and preconditioning will be effective when the cost of these matrix-vector products is outweighed by the reduction in the number of iterations required to achieve convergence. Use of preconditioning also requires additional storage. For a sparse problem, this typically amounts to one extra array to store the elements of Q and another to store the associated indices, or roughly the same amount of storage needed for the matrix A. The preconditioning schemes that were examined were selected in an attempt to balance the density of the matrix Q against the desirable property that Q^{-1} approximate A^{-1}. Throughout the rest of this thesis it is assumed that the diagonal of A contains no zeros. This assumption is always valid for the circuit models under consideration.

4.1 ILU preconditioning.

The first preconditioner considered is the incomplete LU factorization (ILU) [23] of the coefficient matrix A. Let Z be a subset of the set of indices $\{(i,j) \mid 1 \leq i,j \leq N, i \neq j\}$, typically where A has structural zeros. Let $\bar{Z}$ be the complement of Z. Then the ILU factorization of A is given by $Q = LU$ where L and U are lower triangular and unit upper triangular matrices, respectively, satisfying

$$\begin{cases} L_{ij} = U_{ij} = 0, & \text{if } (i,j) \in Z; \\ Q_{ij} = A_{ij}, & \text{if } (i,j) \in \bar{Z}. \end{cases}$$

Taking the ILU factorization of A as the preconditioning matrix allows the preservation of the exact sparsity pattern in A, and so permits the arrays storing Q and A to share the same array of indices.

4.2 Sparse submatrix preconditioning.

The Jacobian matrices considered in this thesis have substantial numbers of off-diagonal entries that are very small in magnitude relative to the majority of nonzeros. This suggests constructing an inexpensive preconditioner by ignoring some of these relatively small values.

Let S be a matrix formed from A by taking the diagonal of A and a percentage of the largest off-diagonal elements of A, and zero elsewhere. The ILU factorization, Q, of S may be used as a preconditioner for A. We refer to this as ILUS preconditioning. This is similar to the ILUT(k) preconditioner [13], [29] but does not allow any fill-in. Taking the percentage of off-diagonal entries retained to be 100 yields the usual ILU preconditioner. Counterintuitively, it is possible that retaining a smaller percentage of the off-diagonal entries may produce a superior preconditioner; i.e., the condition number or spectrum of $Q^{-1}A$ may be better than if ILU preconditioning were used. In any case, the increased sparsity of the ILUS matrix Q will reduce the amount of work necessary to apply the preconditioner. If the ILUS preconditioner yields an iteration matrix whose conditioning or spectrum is only slightly worse than with ILU preconditioning, then use of ILUS preconditioning may reduce execution time, even though more iterations are required to achieve convergence.

Assuming that the percentage of off-diagonal entries to be retained is specified, the matrix S may be extracted with work proportional to $|\bar{Z}|$. The off-diagonal entries of A must be examined to determine a threshold value to serve as a lower bound for the retained off-diagonal entries. This may be done by using an order statistics algorithm [33], rather than a full sort of the off-diagonal entries. The nonzero part of the matrix S then consists of the diagonal of A and all off-diagonal entries of A whose magnitudes equal or exceed the threshold. The actual density of S may differ from the specified value, since there may be many equal elements in A. The Jacobian matrices to which this scheme was applied typically had $5N$ to $6N$ nonzeros, so the cost of computing the threshold was not large, so long as the value of the threshold did not need to be recomputed too often. The ILU factorization of S may be computed in the usual manner.

If we proceed naively, there is little difference between the use of ILU and ILUS preconditioning. Consider the application of an iterative method with ILUS preconditioning

to a sequence of Jacobian matrices along a homotopy zero curve. The value of the threshold may be determined from the first Jacobian matrix. For each Jacobian matrix A, we extract a sparse matrix S from A in the manner described above, compute the ILU factorization Q of S, and perform preconditioned iterations of the method until it converges to a solution of (3). ILUS preconditioning potentially requires increased storage, since A and Q cannot share array indices.

However, the entries of A may vary considerably in magnitude as the zero curve γ is tracked. If the value of the threshold obtained from the initial Jacobian matrix is used along the entire zero curve, the actual percentage of off-diagonal entries retained will also vary. It is therefore desirable to consider the actual percentage of off-diagonal entries retained at each step, and recompute the threshold whenever the actual percentage retained differs excessively from the target percentage. In the following discussion, this variant will be called *threshold-adaptive* ILUS or simply ILUSt.

Care must be taken to avoid recomputing the threshold too often—experiments indicate that allowing the actual percentage retained to vary 10% or 15% from the target percentage produces good performance. In the experiments reported here, the threshold was not recomputed unless the actual percentage retained differed from the target by at least 15%.

Experiments with threshold-adaptive ILUS indicate that using a denser matrix S will generally produce lower iteration counts. However, use of a relatively sparser matrix S may result in smaller execution times, so long as the increase in the iteration count is modest. This suggests adjusting the target percentage upward or downward according to the number of iterations required for convergence in the previous step along the zero curve. In this way, a sparse preconditioner can automatically be used for relatively easy problems and a denser one for relatively hard problems. In the following discussion, this variant will be called *hybrid-adaptive* ILUS or simply ILUSh. Given a desired iteration count, a current target percentage, a corresponding threshold value, and the actual percentage of off-diagonal entries retained at the previous step along γ, ILUSh preconditioning is implemented as follows:

if $|actual\_pct - target\_pct| > tol_1$ **then**

recompute the threshold.

Extract sparse matrix S from A.

Compute ILU factorization Q of S.

for $iteration\_count := 1$ **step** 1 until convergence **do**

perform preconditioned iteration and increment $iteration\_count$.

$iteration\_ratio := iteration\_count/iteration\_limit$.

if $(iteration\_ratio < tol_2)$ **or** $(iteration\_ratio > tol_3)$ **then**

reset $target\_pct$.

The values of tol_2 and tol_3, which determine when the target percentage will be reset, should not be too close to 1. Experiments indicate that taking $tol_2 = 0.75$ and $tol_3 = 1.5$ produces good results, and those values were used in the numerical experiments reported below. The experiments also indicate that the target percentage is reset far more often than the threshold is recomputed, i.e., when the target percentage is reset, the actual percentage of off-diagonal elements retained still falls within an acceptable range. A reset of the target percentage requires only a few flops in each step along γ, so these unnecessary resets are acceptable.

The experiments with threshold-adaptive ILUS indicate that setting the target percentage too low can result in a dramatic increase in the iteration count. Therefore a lower bound of 50% was established for the target percentage in the experiments reported below. Two schemes for adjusting the target percentage were examined. The first was a simple averaging scheme:

if $(iteration\_ratio < tol_2)$ **then**

$target\_pct := (target\_pct + minimum\_pct)/2$;

else if $(iteration\_ratio > tol_3)$ **then**

$target\_pct := (target\_pct + 100)/2$;

while the second used a proportional adjustment:

if $(iteration\_ratio < tol_2)$ **then**

$$target\_pct := target\_pct - (target\_pct - minimum\_pct) * (1 - iteration\_ratio)$$

else if $(iteration\_ratio > tol_3)$ **then**

$$target\_pct := target\_pct + (100 - target\_pct) * (1 - 1/iteration\_ratio)$$

Tests indicated that the proportional adjustment scheme held a slight advantage, and the experiments reported here used that approach.

The iteration limit represents a target for the iteration count on each step along the zero curve. In order to adjust expectations to experience, it is necessary to recompute the iteration limit in order to reflect the number of iterations actually required for convergence. In the codes tested, whenever the target percentage was reset, the iteration limit was reset by averaging it with the iteration count from the previous step. The effect of choosing different initial values for the iteration limit was examined, and the choice was found to not influence results significantly. In the experiments reported here, the iteration limit was initialized to $N/2$.

Finally, an initial target percentage must be specified. The hybrid-adaptive ILUS codes were tested using starting percentages ranging from 100% down to 40%, in increments of 5%. While there was some variation in the results, taking an initial value of 100% produced results that were generally as good as for any other initial value. In the absence of a reason to do otherwise, it seems reasonable to initialize the target percentage to 100%, and that was done in the experiments reported below.

4.3 ILU preconditioning with limited fill.

The Jacobian matrices for some of the test problems have a pattern of 6×6 blocks along a portion of the diagonal and relatively unstructured nonzeros elsewhere. The diagonal blocks (which correspond to the internal variables in a particular transistor model) are structured, nearly symmetric, and approximately 78% full. Most of the nonzeros in the Jacobian matrices occur within these blocks, so it is possible that allowing fill to occur within these blocks when the ILU factorization Q is computed will result in a Q^{-1} that is a better approximation

to A^{-1}. In the following discussion, this scheme is referred to as *block-fill* ILU or simply ILUB.

The implementation of ILUB preconditioning only requires the insertion into the sparse data structure of additional cells corresponding to the empty cells in the 6×6 blocks. The ILU factorization may then be computed in the usual manner. Since allowing fill within the diagonal blocks changes the structure of the matrix, the ILUB preconditioning matrix Q cannot share an index array with A. Thus the ILUB scheme will require more storage than either ILU or ILUS. Moreover, the increased density of Q implies an increase in the cost of applying the preconditioner.

A number of alternative preconditioning schemes that also use an ILU factorization with limited fill were considered. The presence of the diagonal blocks suggests that allowing fill to occur within bands encompassing the blocks might produce a more effective preconditioner. This scheme, called *band-fill* ILU, was implemented in a manner similar to that used for ILUB. Tests with Craig's method and GMRES(k) using band-fill ILU preconditioning demonstrated occasional slight reductions in the average number of iterations needed for convergence (compared to ILUB preconditioning). However, the considerable increase in density led to higher execution times in every case. Numerical results are given in Section 7.3.

Limited experiments were also conducted with a hybrid-adaptive variation of the band-fill ILU preconditioner, in which small entries outside the bands were discarded. The results indicated that this approach was likely to increase iteration counts and execution times substantially. A variation of ILUB preconditioning, in which the smallest off-diagonal entries lying outside the diagonal blocks are discarded, is also possible. In view of the performance of the hybrid-adaptive band-fill ILU preconditioner, that ILUB variation was not pursued. One could also use as a preconditioner the LU factorization of the matrix consisting of the entries of the Jacobian matrix that lie on the diagonal or within the diagonal blocks. Again, limited experiments showed these preconditioners to be relatively ineffective. All these minor variations add no substantially different information from the ILUB preconditioner, and results for them are not included in Chapter 7.

5. NUMERICAL ANALYSIS OF ELECTRONIC CIRCUIT EQUATIONS.

Computer simulation is a necessary adjunct to the design of integrated circuits. The high cost of design iterations, combined with market pressure in the chip business place a high premium on design methodologies which produce working silicon as quickly as possible. Computer simulation techniques provide important performance information to a designer without the expense and delay of building a prototype circuit.

A necessary component of almost any simulation task is the computation of a DC operating point. This point is used as the initial value for an ODE solver, or can be used to compute an approximate linearized model of the circuit for so-called "small signal" analysis.

The sharply nonlinear behavior of certain semiconductor elements makes the computation of an operating point difficult. This chapter describes a formulation of the DC operating point problem using globally convergent homotopy methods. Chapter 6 presents performance data for a suite of bipolar integrated circuits taken from current industrial designs.

5.1 Equation formulation and device models.

The DC operating point problem is typically formulated as a system of n equations in n unknowns to be solved for zero, i.e., $F(x) = 0$, where $F : E^n \to E^n$. The unknowns are node voltages and branch currents and the equations represent the application of Kirchoff's current and voltage laws at various points in the circuit. Different circuit components (resistor, transistor, diode, etc.) impose linear and nonlinear constraints among the unknowns.

Values of n range up to 100,000. For large n, the Jacobian matrix of F is very sparse, with only a few nonzero elements per row. In present fabrication technologies, circuit components are arranged in the form of a graph which is almost planar, which puts a limit on the density of interconnections between circuit elements, leading to a sparse Jacobian matrix.

If a circuit contains only the familiar two-terminal elements—voltage source, current source, resistor and diode—it is possible to formulate equations so that the resulting Jacobian matrix is symmetric positive definite. However, the introduction of any *coupling* element like a transistor or controlled source destroys the symmetry of the matrix. Thus, except for

rather special cases, circuit equations lead to large, sparse nonsymmetric matrix problems. In some cases, a circuit element like a transistor is represented as a small "subcircuit", which is installed in place of every transistor in the network. This policy results in a replication of structurally identical submatrices throughout the overall system Jacobian matrix. This structure can be used to advantage during the solution of linear systems involving the Jacobian matrix.

Of course, electronic devices operate in a fashion which is mathematically smooth, although device modeling subroutines might not always reflect this. Semiconductor physics often analyzes the behavior of a circuit element by considering qualitatively different "regions" of behavior. For example, the operation of a diode is classified as "reverse biased" or "forward biased". Equations describing the operation of the device within one region of operation are naturally C^∞ smooth, however, transitions between regions are problematic. A simple method of achieving smooth modeling is to sample the operation of the device at various voltages and currents, then fit a spline through the sample points. Such *table models* are popular and allow a model for a device to be built before the detailed physics of the element is known. However, each different setting of the parameters of the device requires a new table. Also, in some cases it is useful to compute *sensitivity information*, which shows how the response of the circuit element changes with a perturbation in one or more of its parameters. This kind of analysis is easier with so-called *analytical models*, which represent the behavior of the device with equations derived from an understanding of the semiconductor physics of the device. These equations are symbolic expressions involving the device parameters, voltages and currents. Thus, it is possible to compute exact analytic derivatives of voltage and current with respect to device parameters. There is a difficulty with analytic models, however, in getting smooth transitions between regions of operation. In bad cases, the analytic equations are not even continuous across region boundaries. Even when formulated as smooth functions, device modeling functions can still present extremely sharp nonlinearities. This is particularly true for bipolar devices because of the exponential nature of the *pn*-junction equation.

5.2 Homotopies for circuit equations.

Homotopy (continuation) methods allow reliable solution of DC operating point equations. However, the performance of such methods is sensitive to the manner in which the homotopy parameter λ is introduced into the equations. The standard embedding $H(x,\lambda) = \lambda\, F(x) + (1-\lambda)(x-a)$ is widely used in continuation work, but our computational experience shows that this is a very poor embedding for circuit equations. Generally speaking, good performance can be obtained only by pushing the continuation parameter down *into the device model equations.* This requires minor modifications to the device model code, but the results are well worth the effort [24]. Consider, for example, the classic Ebers-Moll model for a bipolar transistor [14]. Suppose the forward and reverse current gains (α_F, α_R) are multiplied by λ. The result is a somewhat artificial *variable gain* transistor. The random perturbation required by the probability-one homotopy theory is accomplished by connecting a *leakage circuit* from each node of the circuit to ground. This circuit consists of a conductance in series with a random value voltage source. At $\lambda = 0$, each transistor reduces to a pair of diodes, and a damped Newton scheme is able to solve the start system $\rho_a(x,0) = 0$ quickly. As λ approaches one, the value of the leakage conductance goes to zero, thus disconnecting the leakage elements from the circuit. Moreover, the transistors are restored to a full-gain condition at $\lambda = 1$, so the solution to the homotopy equations at $\lambda = 1$ is an exact DC operating point of the network. The performance results reported below are based on this variable gain homotopy map.

6. CIRCUIT SIMULATION TEST PROBLEMS.

The test problems consist of real circuits studied and used by scientists at AT&T Bell Laboratories. Some of the problems are described in complete detail (with circuit diagrams) in [24], and the data for all of them is available from the author or *netlib*. The test data were obtained by solving each problem with HOMPACK, using a direct method to compute the kernel of the Jacobian matrix at each step along the zero curve, and writing the nonzero entries of the Jacobian matrices to a file. The iterative methods considered here were then applied to each sequence of Jacobian matrices. The sparse Jacobian matrices were stored in a variation of the compressed sparse row storage scheme [48]. The nonzero matrix entries were stored by rows in a linear array, with a parallel array of column indices. The starting indices for each row were also stored in a second integer array. The rows, and column indices for each row, were assumed to be in order within the linear arrays.

Table 2 gives the test problem names (matching those in [24]), dimension N, number of nonzeros NZ, and the number of Jacobian matrices NJ. The column labeled DB indicates whether the Jacobian matrices for each problem had diagonal blocks. Fig. 1 shows one of the circuits from which the test problems were derived. This circuit, known as the Brokaw voltage reference circuit [24], is a fairly well known circuit. The sparsity pattern for the Jacobian matrices corresponding to the circuits ups01a, vref and bgatt are shown in Figures 2, 3 and 4; note the diagonal blocks corresponding to the transistor subcircuits mentioned earlier. Fig. 5 illustrates the relatively unstructured sparsity pattern for is7a, obtained using a different circuit model. The sparsity pattern for is7b is similar to that for ups01a and bgatt.

TABLE 2. CIRCUIT SIMULATION TEST PROBLEMS

Problem	N	NZ	NJ	DB
rli13b	31	120	118	N
ups01a	59	342	25	Y
vref	67	411	13	Y
bgatt	125	782	14	Y
is7a	468	2871	5	N
is7b	1854	10849	5	Y

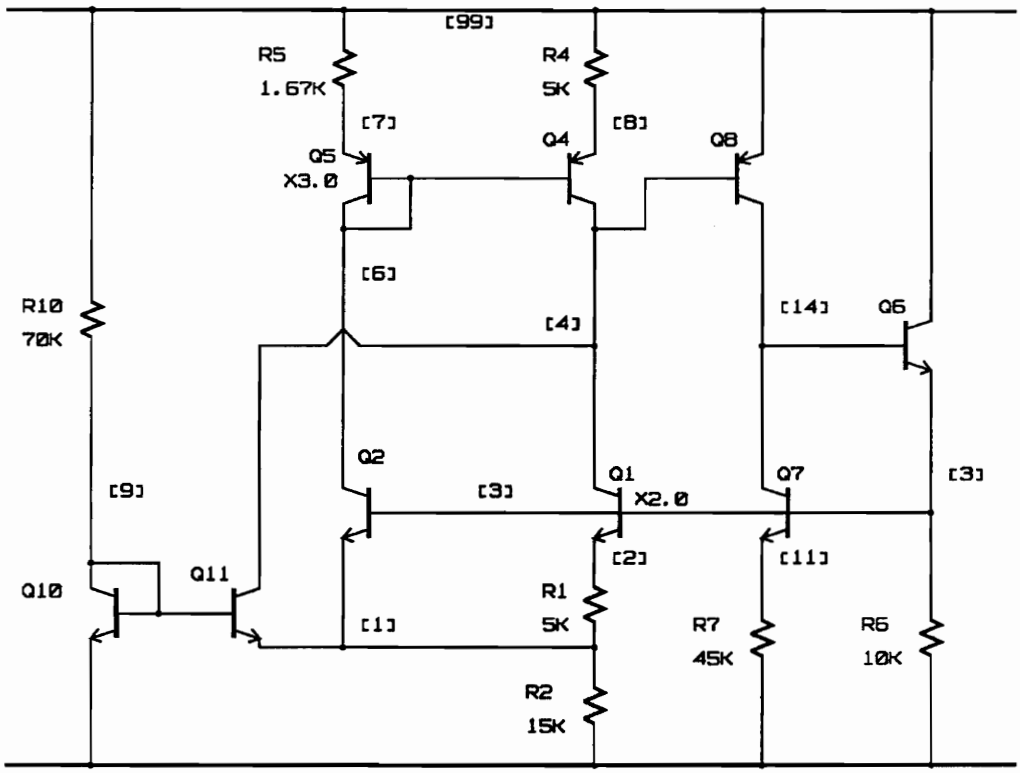

FIG. 1. *Circuit diagram for the circuit* vref.

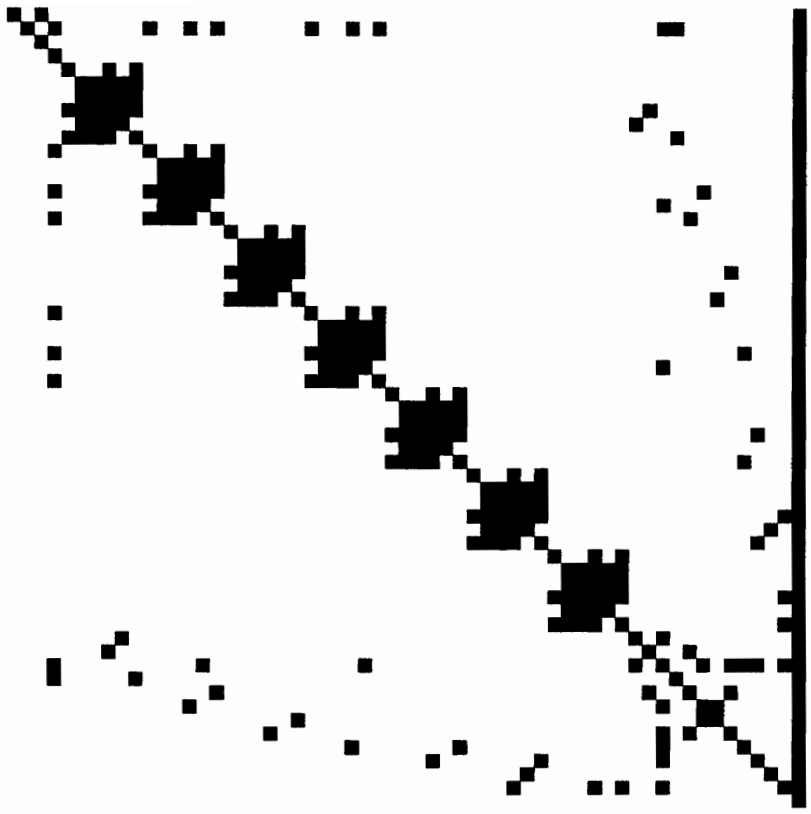

FIG. 2. *Jacobian matrix sparsity pattern corresponding to the circuit* ups01a.

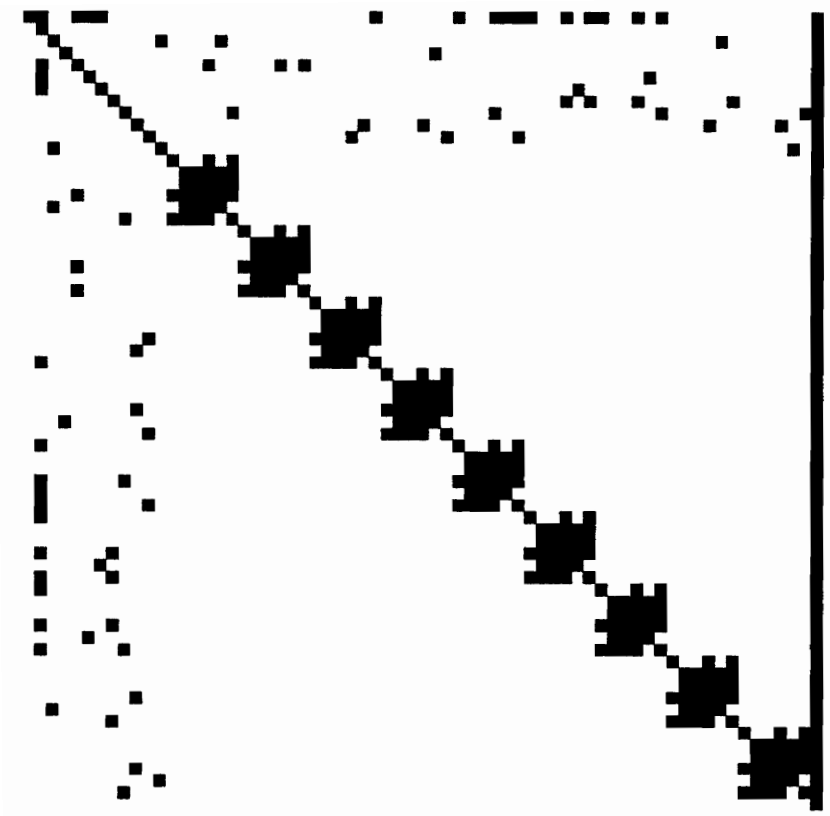

FIG. 3. *Jacobian matrix sparsity pattern corresponding to the circuit* vref.

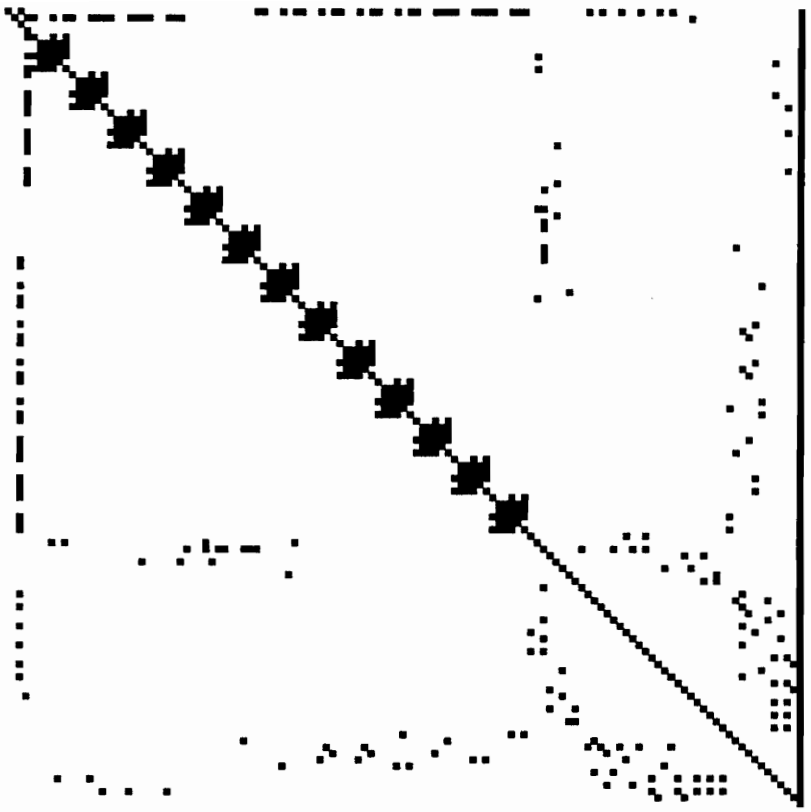

FIG. 4. *Jacobian matrix sparsity pattern corresponding to the circuit* bgatt.

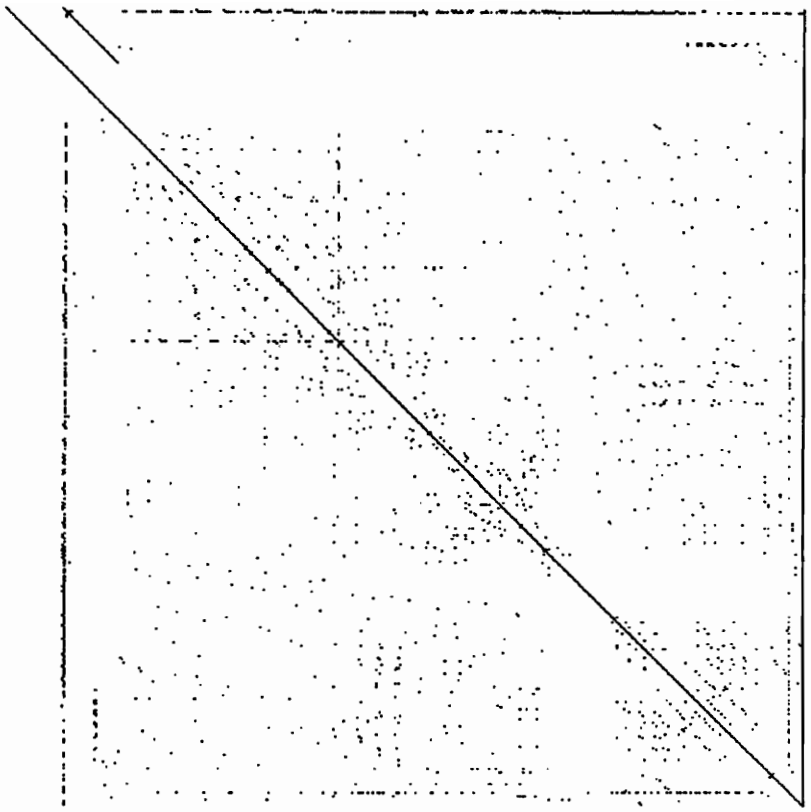

FIG. 5. *Jacobian matrix sparsity pattern corresponding to the circuit* is7a.

6.1 Conditioning.

The rate of convergence of many iterative methods depends upon the conditioning of the matrix A, or an associated matrix such as AA^t. While a smaller condition number may not necessarily imply faster convergence, the effect a preconditioning scheme has on the condition number may provide some measure of the usefulness of that preconditioner.

The condition number of an $n \times n$ matrix may be approximated by the LINPACK routine DGECO, which returns an estimate that is within a factor of n of the correct value. The effect of a preconditioning scheme on the conditioning of a matrix may be approximated in the same manner, if the preconditioner is explicitly applied to the matrix before DGECO is called. This approach was used to measure the conditioning of the smaller Jacobian matrices.

Since the Jacobian matrices were actually rectangular, it was necessary to augment them with one additional row in order to apply DGECO. In keeping with the discussion at the beginning of Chapter 3, each Jacobian matrix was augmented with the row e_n^t, where n was the number of columns in the Jacobian matrix.

Figures 6 through 10 show the results, plotting the base 10 logarithm of the condition number of each Jacobian matrix vx. Jacobian number, for the circuits rli13b, ups01a, vref, bgatt and is7a. Condition numbers without preconditioning, after ILU preconditioning and after ILUB preconditioning are plotted with solid, dashed, and dash-dotted lines, respectively. Since the rli13b and is7a Jacobian matrices do not have diagonal blocks, the corresponding figures show only condition numbers without preconditioning and after ILU preconditioning.

In most cases, application of ILU preconditioning reduced the condition number of the Jacobian matrix by between 2 and 3 orders of magnitude. There is, however, a chance that applying ILU preconditioning to an unsymmetric matrix will actually increase the condition number. That occurred for a small number of rli13b Jacobian matrices and for every is7a Jacobian matrix. It is worth noting that, even in such cases, the convergence rate of an iterative method may be improved. For example, the condition number of the 74-th rli13b Jacobian matrix is approximately 1.6×10^6 without preconditioning and approximately 1.0×10^7 with ILU preconditioning. However, Craig's method requires 78 iterations to solve

the corresponding linear system if no preconditioner is used, and only 25 iterations if ILU preconditioning is used.

There is little difference between the effect of ILU and ILUB preconditioning. For ups01a and vref, the plots are virtually identical; for bgatt, the condition numbers are slightly larger if ILUB preconditioning is used.

6.2 Spectra.

The rate of convergence of an iterative method may also depend upon the spectrum of A, or an associated matrix. As with the condition numbers, examination of the spectrum may not provide a foolproof prediction of the performance of a particular method. Meaningful quantitative assessments are, therefore, more difficult, but some general observations may be useful.

The eigenvalues of the augmented Jacobian matrices were computed by applying the EISPACK routine RG. Augmenting the Jacobian by e_n^t guaranteed that one eigenvalue would equal 1.0. In the discussion below, that somewhat artificial eigenvalue is disregarded. The effect of the ILU and ILUB preconditioners on the spectrum was assessed by applying the EISPACK routine RG to the matrix obtained by explicitly applying the preconditioner. These calculations were done for the rli13b, ups01a, vref and bgatt Jacobian matrices.

Generally, the eigenvalues of the Jacobian matrices exhibited little clustering. The magnitudes of the eigenvalues for typical Jacobian matrices ranged from 10^{-5} to 10^{-2} for rli13b, from 10^{-6} to 5.5 for ups01a, and from 10^{-5} to 9.2 for vref. Complex eigenvalues had positive real part, and frequently had imaginary parts of nearly equal magnitude.

After either preconditioner was applied the eigenvalues showed significant clustering near 1.0. For the typical Jacobian matrices considered above, 23 of 30 eigenvalues equaled 1.0 for rli13b, 46 of 58 equaled 1.0 for ups01a, and 42 of 66 equaled 1.0 for vref.

For the same Jacobian matrices considered above, if ILU preconditioning was used, the remaining eigenvalues ranged from 0.46 to 1.46 for rli13b, from 0.53 to 1.56 for ups01a, and from 8.1×10^{-4} to 1.997 for vref. For rli13b and ups01a, many of the eigenvalues not equaling 1.0 were near 1.0 in the complex plane. For vref, those eigenvalues included clusters with magnitudes near 10^{-4} and 10^{-2} and quite a few with magnitudes significantly greater than 1.0. The results with the ILUB preconditioner were almost identical, the only differences being slight variations in the ranges cited above.

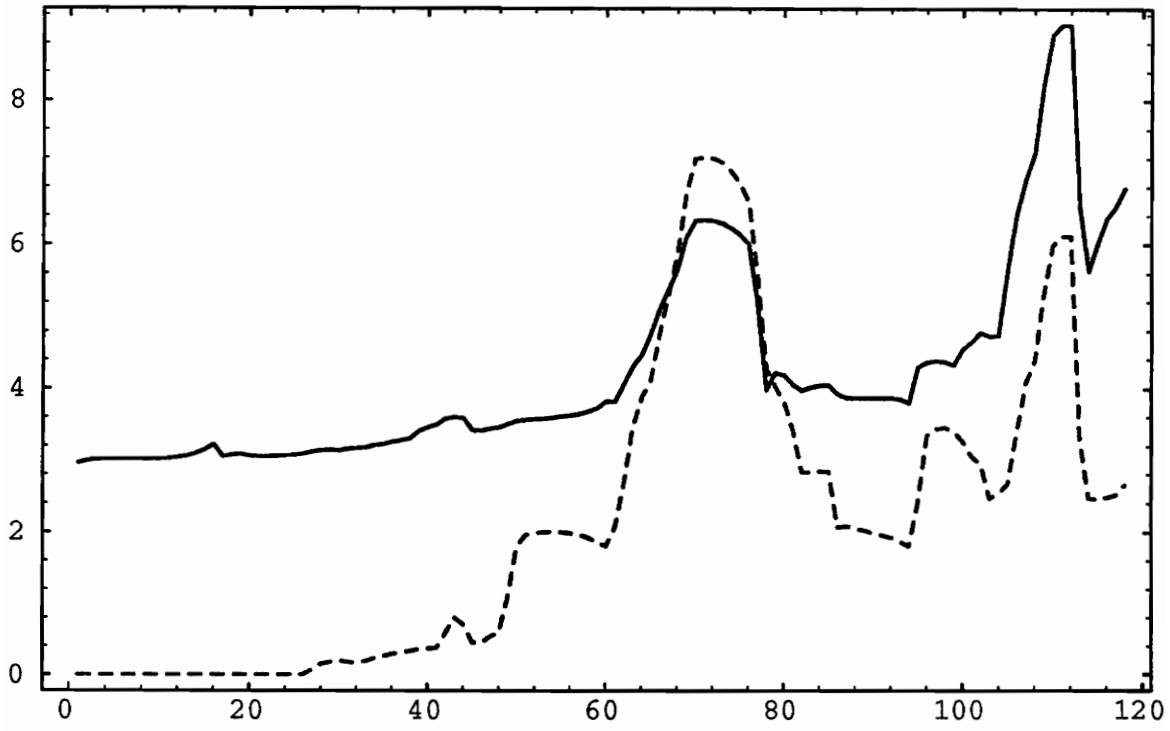

FIG. 6. *Conditioning of the Jacobian matrices for the circuit* rli13b.

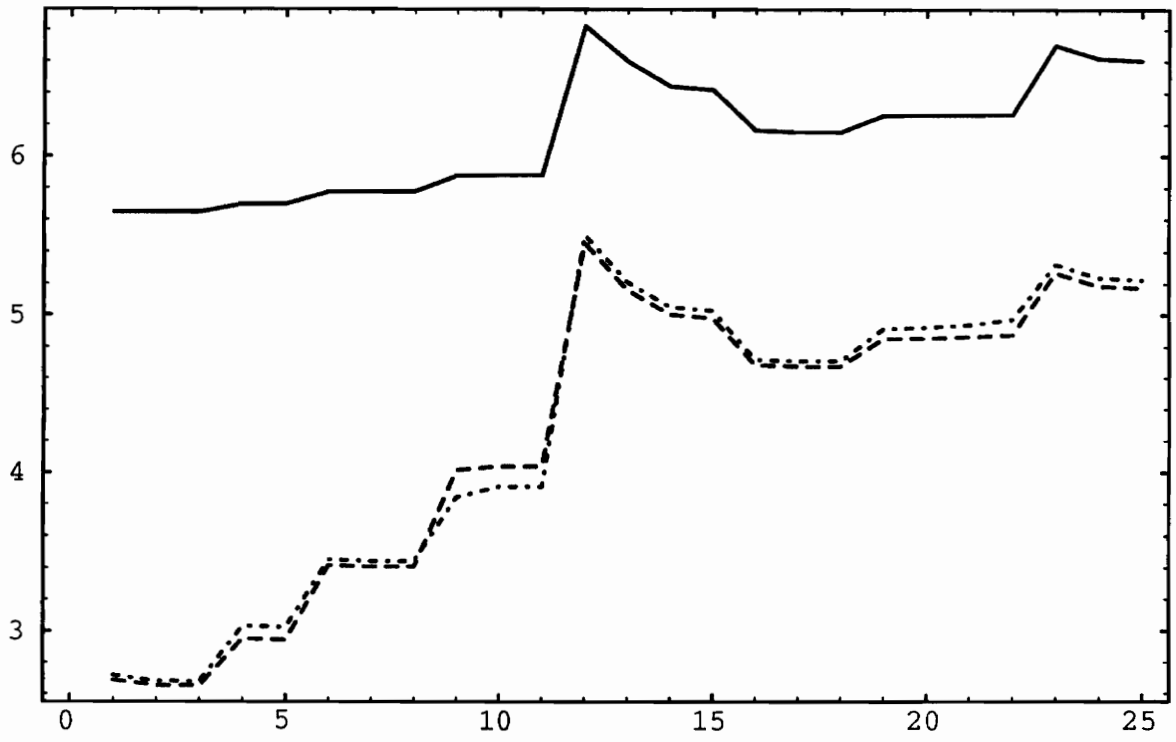

FIG. 7. *Conditioning of the Jacobian matrices for the circuit* ups01a.

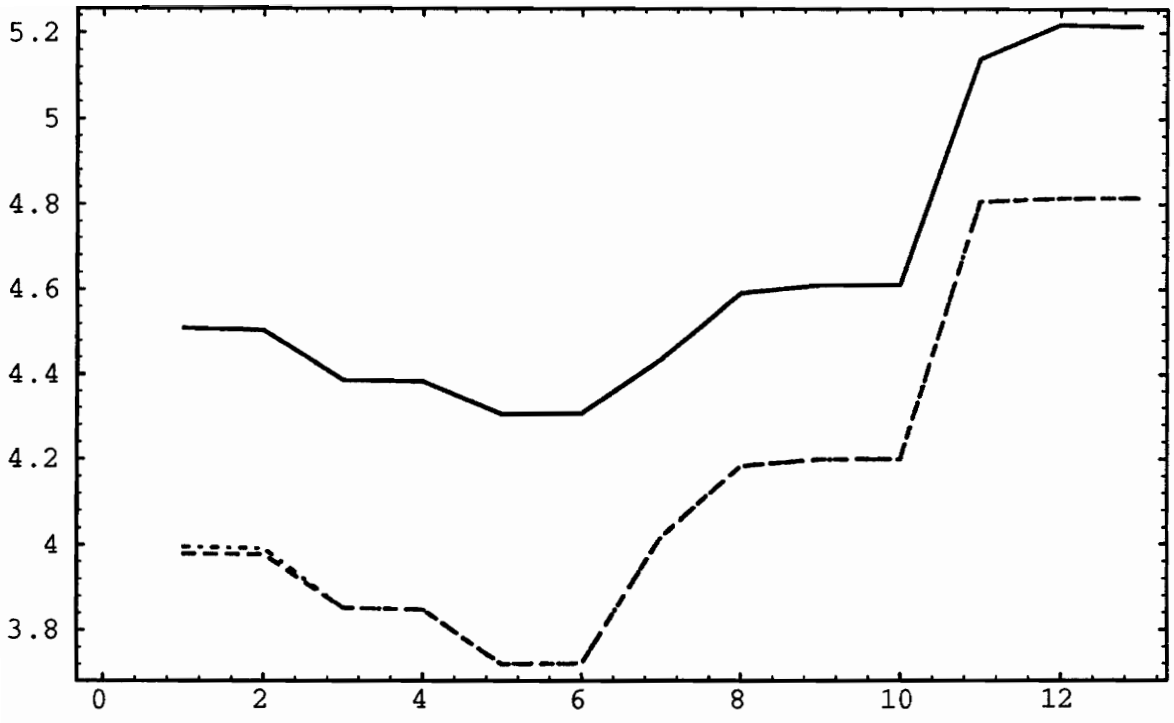

FIG. 8. *Conditioning of the Jacobian matrices for the circuit* vref.

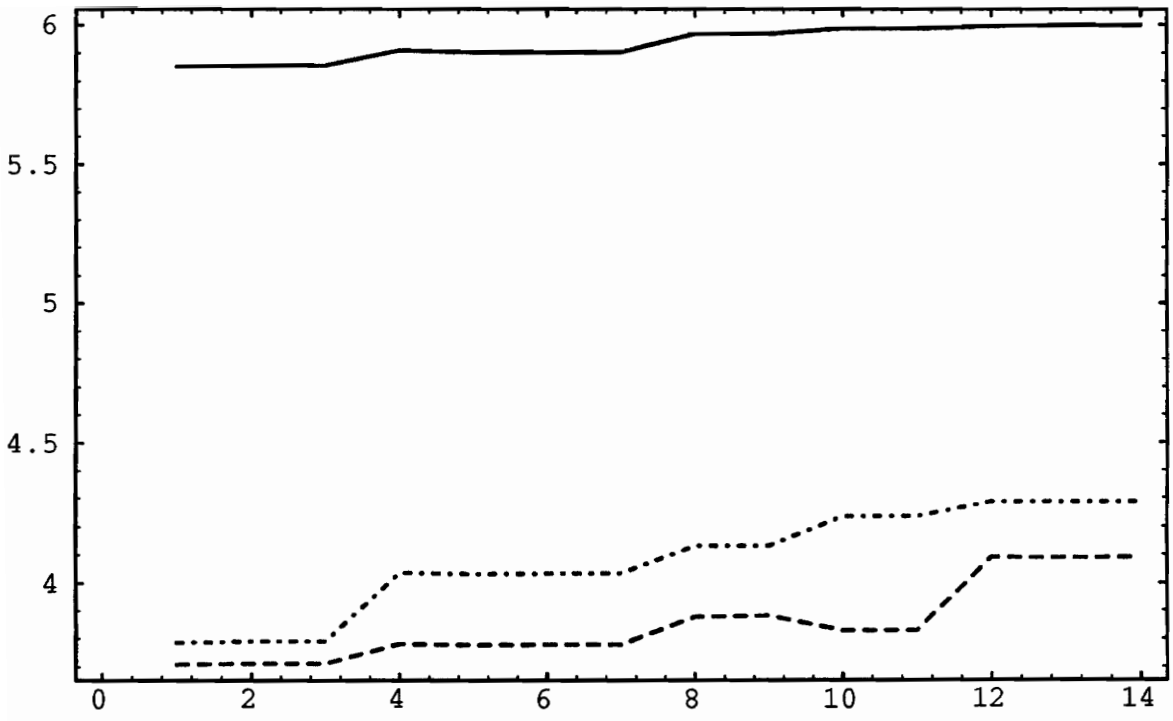

FIG. 9. *Conditioning of the Jacobian matrices for the circuit* bgatt.

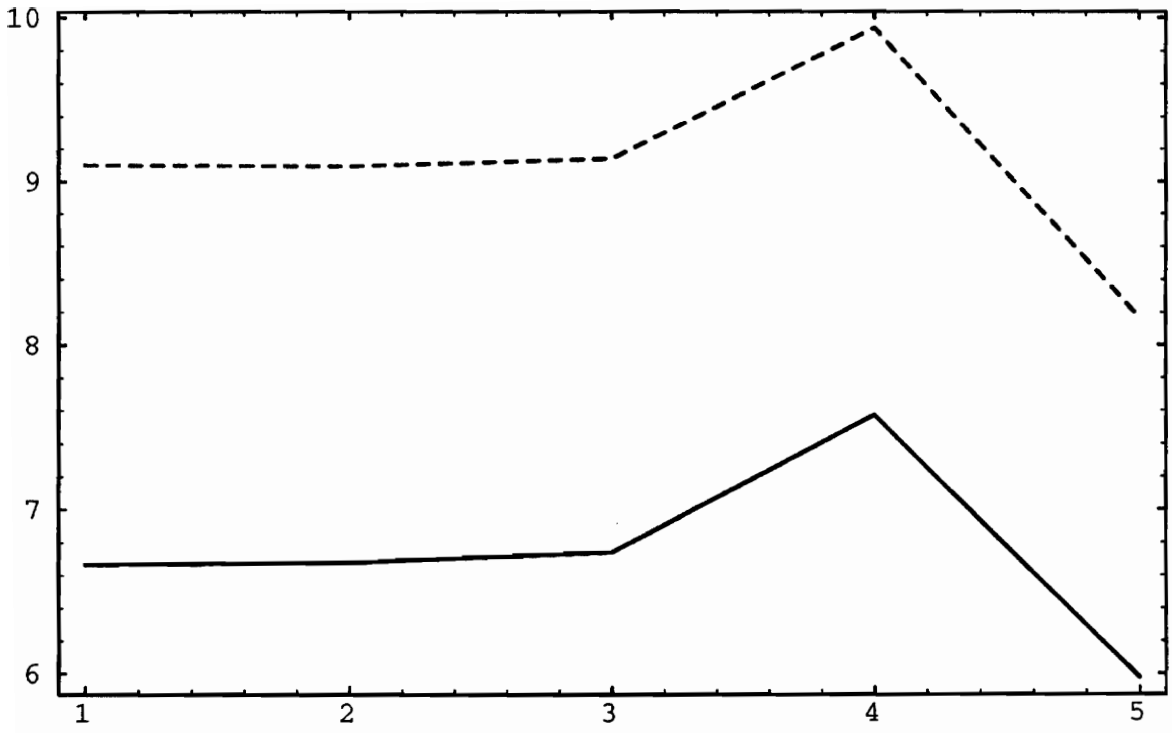

FIG. 10. *Conditioning of the Jacobian matrices for the circuit* is7a.

7. NUMERICAL RESULTS.

Each iterative method was applied to each of the test problems. When applied without preconditioning, none of the methods achieved convergence in $5N$ iterations except on the problem of order 31; and even for this problem, the iteration counts and execution times achieved without preconditioning were two or three times larger than those achieved with any of the preconditioning schemes.

The row projection method KACZ consistently required substantially more time than any of the other methods. For example, KACZ required about 8 times as long to solve the problem of order 31 as did Craig's method. Thus KACZ was not tested with the full range of preconditioners, nor are numerical results for KACZ given in the tables. However, it must be noted that a sequential implementation of KACZ was used, and the row projection methods are perhaps best suited to a parallel environment.

Each of the other iterative methods was applied in combination with each of the preconditioning schemes to each of the test problems, if appropriate. The remainder of this chapter presents and discusses the numerical results obtained. Section 7.1 discusses the performance of the threshold-adaptive preconditioner. Section 7.2 discusses the relative performance of ILU, ILUSh and ILUB preconditioning. Sections 7.3 and 7.4 discuss the effectiveness of some alternative preconditioners and the effect the choice of the parameter k had on the performance of GMRES(k). Section 7.5 compares the overall performance of the various methods.

In the tables that follow, avg% refers to the average percent of off-diagonal elements of the Jacobian matrices along γ retained in S. The minimum, maximum, and average number of iterations along the homotopy zero curve γ are shown, and the CPU time is in seconds on a DECstation 3100. The reported times are the medians of times obtained on from three to five runs. All code is double precision, since that is the default for the homotopy software HOMPACK. An asterisk denotes convergence failure at some point along the zero curve γ. In particular, a method was deemed to have failed if, at any step along γ, it required more than $5N$ iterations to converge. Convergence was construed to mean a relative residual less than 100 times machine epsilon; such high accuracy is frequently crucial for homotopy curve tracking [24]. The initial guess for the first system along γ was $x_0 = 0$, and thereafter the initial guess was the solution at the previous point along γ.

7.1 Performance of ILUSt preconditioning.

In order to begin to understand the potential of sparse submatrix preconditioners, the performance of the ILUSt preconditioner was tested with Craig's method and GMRES(k). Table 3 summarizes the results for the test problems rli13b ($N = 31$), ups01a ($N = 59$), vref ($N = 67$), and bgatt ($N = 125$), respectively. In testing GMRES(k), the value of k was chosen as the smallest for which convergence was achieved for relatively sparse preconditioners. Using ILU preconditioning, it was possible to solve each problem with a slightly smaller value of k.

For each method, using a slightly sparser preconditioner produced a slight reduction in execution time for the problem of order 31, even though the iteration count increased. For the problem of order 67, using a substantially sparser preconditioner with GMRES(k) actually reduced the iteration count. For each method, the execution time on that problem decreased significantly as the density of the preconditioner decreased, until the percentage of off-diagonal entries retained dropped below 50%. However, it is possible that the ILUS preconditioner is far less effective than ILU preconditioning on some problems, as the results on the second and fourth test problems show. Using the ILUSt preconditioner with a target percentage of 100% results in somewhat higher times than using the ILU preconditioner, since the ILUSt preconditioner requires separate storage for column indices and hence some additional data movement.

The results obtained using ILUSt preconditioning suggested that the use of a sparse preconditioner could improve performance in some cases. However, there was no obvious way to determine in advance what target percentage would produce the best performance. Moreover, comparing the performance of ILU and ILUSt preconditioning at each step along γ indicated that a denser preconditioner usually produced a lower iteration count when the Jacobian matrix was badly conditioned, but that a sparser preconditioner usually reduced execution time if the Jacobian matrix was not badly conditioned. The hybrid-adaptive preconditioning scheme was devised to provide a practical implementation of these ideas.

TABLE 3. NUMERICAL RESULTS WITH ILUST PRECONDITIONING

Craig's method with ILUSt preconditioning			
N	avg%	min, max, avg	time
31	100	1, 32, 17	4.83
	90	2, 32, 17	4.75
	80	2, 58, 19	5.14
	70	3, 64, 21	5.44
	60	3, 56, 21	5.22
	50	3, 60, 22	5.25
	41	4, 93, 30	6.88
	31	7, 95, 33	7.20
	25	7, 115, 35	7.53
59	100	28, 76, 42	5.93
	78	36, 97, 57	7.31
	70	36, 88, 57	7.05
	65	37, 87, 57	6.89
	59	37, 108, 58	6.91
	50	44, 116, 69	7.74
	41	107, 228, 163	17.94
67	100	75, 134, 89	7.50
	75	82, 132, 95	7.22
	65	81, 137, 96	7.07
	55	81, 133, 96	6.81
	46	82, 123, 96	6.57
	43	93, 142, 103	7.13
125	100	156, 359, 232	38.79
	80	221, 535, 331	51.24
	70	221, 556, 337	50.09
	64	221, 570, 340	49.39
	57	234, 619, 398	56.27

GMRES(k) with ILUSt preconditioning			
N	avg%	min, max, avg	time
31	100	1, 27, 11	2.41
(k=6)	90	1, 27, 11	2.32
	80	1, 35, 13	2.46
	70	1, 42, 13	2.55
	60	1, 39, 13	2.36
	50	1, 46, 14	2.53
	41	1, 75, 20	3.32
	31	5, 113, 28	4.43
	26	5, 136, 34	5.22
59	100	12, 26, 17	1.72
(k=7)	78	14, 35, 22	2.00
	70	14, 35, 22	1.93
	65	14, 35, 22	1.87
	59	14, 48, 24	1.94
	50	21, 61, 33	2.57
	41	*	
67	100	58, 115, 94	6.21
(k=20)	75	72, 111, 92	5.61
	65	70, 111, 92	5.41
	55	70, 108, 90	5.14
	50	59, 108, 88	4.96
	49	72, 120, 93	5.23
125	100	52, 81, 68	7.97
(k=11)	80	63, 175, 106	11.48
	70	64, 175, 106	11.01
	60	70, 220, 102	10.21
	55	83, 209, 114	11.16

7.2 ILU, ILUSh and ILUB preconditioning.

The results obtained using each iterative method in conjunction with ILU, ILUSh and ILUB preconditioning are presented in Table 4. The Jacobian matrices for the problems of order 31 and 468 do not have diagonal blocks, so the ILUB preconditioner is not appropriate for those problems. That is indicated in the table by the annotation NA.

The ILUB preconditioner was applicable to sixteen combinations of problem and method in which convergence was achieved. In terms of iteration counts, the ILUB preconditioner did as well as or better than either ILU or ILUSh preconditioning for every combination of problem and method, except when BiCGSTAB was applied to the problem of order 67. The increased cost of applying the denser ILUB preconditioner led to the highest execution time in five cases. In three cases the execution time using ILUB preconditioning was more than 10% greater than the best achieved with either of the other preconditioners. However, ILUB preconditioning reduced execution times by 10%, versus ILU, in two cases and by 75% in a third case. Execution times using ILUB preconditioning ranged from 25% to 119% of those achieved with the other preconditioners.

There were twenty-four combinations of problem and method in which convergence was achieved with ILUSh preconditioning. In eleven of those combinations, the number of iterations needed for convergence was so large that 100% of the off-diagonal entries of the Jacobian matrix were retained for the preconditioner, essentially reducing the ILUSh scheme to ILU preconditioning. The overhead in the ILUSh code is apparent in the timings for those cases. Overall, the ILUSh preconditioner was associated with the largest execution time in sixteen of the twenty-four problem/method combinations, including seven of the thirteen cases in which a sparse preconditioner was selected. ILUSh preconditioning produced the lowest execution time in only two cases, both involving the smallest test problem. Even in those cases the reduction in time was less than 4%, when compared to the best time achieved with the same method and ILU preconditioning. In six combinations of problem and method, the time required for convergence with ILUSh preconditioning was at least 10% greater than the best time achieved with the other preconditioners.

There were also twenty-four problem/method combinations in which convergence was achieved with ILU preconditioning. There were no cases in which the ILU preconditioner was either clearly best or worst among the three preconditioners tested. ILU preconditioning was associated with the lowest execution time in fourteen cases. In seven cases, the time required using the ILU preconditioner was second lowest, and within 5% of the best time for that problem/method combination. In only three cases did the ILU preconditioner correspond to an execution time more than 10% greater than the best time achieved with the other schemes.

7.3 A Band-fill ILU Preconditioner.

The development of the ILUB preconditioner was preceded by an examination of the effect of using an ILU factorization and allowing fill within b bands on either side of the diagonal. This *band-fill* ILU preconditioning scheme was implemented by inserting into the data structure for the Jacobian matrix A the new cells required to fill the desired number of bands. The usual ILU factorization code was applied to the resulting matrix. This approach had the virtue of simplicity, but created far more cells than were actually filled with nonzeros during the ILU factorization. This involved not only an excessive increase in the storage requirement but also in the number of operations required to apply the preconditioner.

It would have been possible to parse out the zeros after the ILU factorization was completed. That option was not pursued because the fill that occurred appeared to fall almost entirely within the existing diagonal blocks, and that motivated the ILUB preconditioner. Nonetheless, the results obtained using the band-fill ILU preconditioner are sufficiently interesting to report.

The first section of Table 5 shows the number of cells required for the band-fill preconditioner for various values of b. Note that $b = 0$ corresponds to ILU with no fill, and $b = 5$ exactly encompasses the diagonal blocks. The final section of Table 5 shows the number of cells that need to be stored if ILUB preconditioning is used. Letting $b = 5$ requires slightly more than twice as much storage as the ILU factorization, while the ILUB factorization increases storage cost by less than 20%.

TABLE 4. NUMERICAL RESULTS WITH ILUSH, ILU AND ILUB PRECONDITIONING

Craig's method								
	ILUSh			ILU			ILUB	
N	avg%	min, max, avg	time	min, max, avg	time		min, max, avg	time
31	73	1, 33, 17	4.60	1, 32, 17	4.78		NA	
59	100	28, 76, 42	5.84	28, 76, 42	5.46		28, 72, 37	5.24
67	100	75, 134, 89	7.40	75, 134, 89	6.87		68, 127, 82	6.80
125	100	156, 359, 232	38.49	156, 359, 232	35.32		134, 270, 187	31.79
468		*		*			NA	
1854		*		*			*	
LSQR								
31	87	1, 30, 16	5.65	1, 30, 16	4.93		NA	
59	100	24, 53, 32	4.91	24, 53, 32	4.49		24, 50, 29	4.73
67	100	69, 89, 77	6.96	69,89, 77	6.36		63, 83, 70	6.68
125	100	147, 276, 195	34.52	147, 276, 195	32.63		126, 216, 155	29.24
468		*		*			NA	
1854		*		*			*	
GMRES(k)								
31	84	1, 24, 11	2.42	1, 27, 11	2.36		NA	
59	71	12, 35, 22	1.96	12, 26, 17	1.70		7, 26, 16	1.81
67	100	58, 115, 94	6.36	58, 115, 94	6.19		20, 20, 20	1.55
125	100	52, 81, 68	8.11	52, 81, 68	7.93		53, 86, 67	8.37
468	84	107, 323, 162	33.60	106, 323, 161	33.20		NA	
1854	100	839, 1038, 926	1072.94	839, 1038, 926	1061.72		748, 1079, 898	1068.90
BiCGSTAB								
31	66	1, 21, 9	2.51	1, 18, 8	2.56		NA	
59	66	8, 19, 14	1.86	8, 16, 11	1.80		8, 17, 11	2.04
67	100	35, 97, 59	4.97	35, 97, 59	4.96		27, 114, 64	5.86
125	71	23, 42, 35	5.37	22, 39, 28	5.12		21, 35, 26	5.18
468	78	72, 124, 98	23.07	67, 124, 87	20.97		NA	
1854		*		*			*	
QMR								
31	78	2, 14, 9	4.59	2, 14, 9	4.72		NA	
59	69	10, 22, 16	3.30	10, 17, 13	3.15		10, 17, 12	3.38
67	100	27, 45, 33	3.66	27, 45, 33	3.68		24, 43, 32	4.12
125	75	28, 44, 34	6.86	26, 38, 29	6.34		25, 32, 27	6.32
468	80	77, 182, 113	32.62	77, 182, 104	30.69		NA	
1854		*		*			*	

TABLE 5. STORAGE COST OF BAND-FILL ILU PRECONDITIONING

	Problem Size		
	N=59	N=67	N=125
b=0	342	411	782
b=1	412	486	949
b=2	478	560	1113
b=3	540	631	1276
b=4	634	735	1489
b=5	725	838	1702
ILUB	398	483	886

Extensive experiments were done using the band-fill ILU preconditioner in conjunction with Craig's method and GMRES(k). Numerical results are given in Table 6. The value of b indicates the number of sub- and superdiagonals that were allowed to fill.

For all three problems, Craig's method with band-fill ILU preconditioning was slower than if ILU preconditioning were used. Moreover, the increase in execution time was considerable if the value of b was large enough to encompass the diagonal blocks. However, the iteration counts were consistently lower with band-fill ILU than for ILU, and larger values of b generally produced lower iteration counts.

The results for GMRES(k) were somewhat mixed. For the problem of order 59, band-fill ILU produced lower execution times than either ILU or ILUB for $b \leq 3$. For the problem of order 67, the execution time was lower with band-fill ILU than with ILU for $b > 1$. However, as might have been expected, the ILUB preconditioner produced an even lower execution time on that problem. Finally, for the problem of order 125, band-fill ILU produced higher execution times than either ILU or ILUB, and the times were much higher for large values of b. For the first two problems, the iteration count was much lower for $b = 5$ than if ILU preconditioning were used. However, for the problem of order 125, increasing the value of b actually increased the iteration count.

Very limited experiments were done using values of $b = 6$ and $b = 7$. That strategy produced even larger execution times and no further reduction in iteration counts.

These results suggested that allowing some fill in the ILU factorization could yield a superior preconditioner. However, the effect of the resulting reduction in iteration count could be overwhelmed by the increased work necessary to apply the denser preconditioner. Those observations provided substantial motivation for the ILUB preconditioner.

TABLE 6. NUMERICAL RESULTS WITH BAND-FILL ILU PRECONDITIONING

Craig's method			
N	b	min, max, avg	time
59	0	28, 76, 42	5.77
	1	28, 60, 36	5.76
	2	26, 63, 34	6.25
	3	23, 64, 33	6.90
	4	23, 62, 33	7.99
	5	22, 62, 33	8.15
67	0	75, 134, 89	7.30
	1	68, 116, 81	7.62
	2	68, 118, 81	8.57
	3	68, 118, 81	9.35
	4	71, 122, 86	11.44
	5	70, 119, 83	12.63
125	0	156, 359, 232	38.66
	1	155, 263, 203	40.22
	2	161, 295, 211	45.97
	3	154, 294, 205	50.02
	4	154, 255, 192	52.90
	5	153, 250, 190	55.85

GMRES(k)			
N	avg%	min, max, avg	time
59	0	12, 26, 17	1.71
(k=7)	1	7, 19, 13	1.43
	2	7, 14, 11	1.49
	3	7, 14, 10	1.65
	4	7, 14, 9	1.90
	5	7, 14, 9	2.30
67	0	58, 115, 94	6.17
(k=20)	1	20, 67, 24	1.66
	2	20, 67, 24	1.88
	3	20, 67, 24	2.05
	4	20, 67, 24	2.03
	5	*	
125	0	52, 81, 68	8.01
(k=11)	1	64, 359, 134	16.25
	2	72, 343, 143	17.72
	3	63, 241, 133	18.07
	4	62, 242, 131	20.09
	5	60, 231, 109	21.57

7.4 Effect of Varying k on GMRES(k).

The numerical results given in Section 7.2 show the performance of GMRES(k) for a single value of k, chosen to achieve convergence on the entire sequence of Jacobian matrices using a relatively sparse submatrix preconditioner. For each problem, GMRES(k) will fail for smaller k, if ILUSt preconditioning is used with a target percentage smaller than 60. However, if ILU or ILUB preconditioning is used, then GMRES(k) will converge for smaller values of k.

Moreover, it is known that the choice of k can have a significant effect on the convergence of GMRES(k). If the value of k is too small then GMRES(k) may not converge at all, and for k sufficiently large GMRES(k) will converge in no more than k iterations. Thus, in order to fairly assess the performance of GMRES(k), it was important to examine the convergence of GMRES(k) for various values of k.

Tables 7 through 10 show how the performance of GMRES(k) varies with k on the block-structured problems of order 59, 67, 125 and 1854. Except for the last problem, the values of k shown range from the smallest for which convergence was achieved to the smallest value of k for which the maximum number of iterations does not exceed k. Iteration counts and timings are given for both ILU and ILUB preconditioning.

The best execution times obtained with ILU preconditioning in Tables 7 through 10 range from 60% to 19% of the times reported in Table 4. However, these improvements require a significant increase in storage cost. For the problems of order 67 and 125, the optimal times correspond to $k = 25$ and $k = 24$. Since the number of nonzeros in these problems is roughly $5N$, the vectors stored by GMRES(k) require about five times as much storage as A itself.

Increasing the value of k had much the same effect when ILUB preconditioning was used. Except for the problem of order 67, the best times achieved with ILU preconditioning were somewhat better than those achieved with ILUB preconditioning.

For each of these test problems, it was possible to achieve dramatic improvements in execution times by increasing the value of k. For the three smaller problems it was possible to find an "optimal" value of k, for which the maximum number of iterations was bounded

by k. Moreover, it was generally possible to achieve significant reductions in execution time for smaller than optimal values of k. The importance of that is evident when the results for the largest problem are considered. In that case, the "optimal" k is larger than 100, and the corresponding increase in storage is unacceptable. However, increasing k from 40 to 50 reduces the execution time by more than 50%.

7.5 Comparison of the iterative methods.

Aside from KACZ, Craig's method and LSQR were clearly the slowest of the methods examined. Craig's method and LSQR were the only methods which failed on the problem of order 468. Craig's method did reduce the error norm at each iteration, as expected, but the rate of convergence was extremely slow, and the iteration limit of $5N$ was exceeded before convergence. LSQR managed a similar reduction of the residual norm, but again convergence was very slow. It is worth noting that LSQR, in combination with each preconditioner, solved the problems of order 59, 67, and 125 in significantly less time than Craig's method.

QMR did not produce the lowest overall execution time on any problem. However, with all three preconditioners, execution times with QMR were lower than for GMRES(k) on the problems of order 125 and 468. QMR with ILUSh and ILU preconditioning was also faster than GMRES(k) on the problem of order 67. QMR, with each preconditioner, was also faster than BiCGSTAB on the problem of order 67. The implementation of QMR that was used here requires the specification of an upper bound m on the number of look-ahead Lanczos vectors retained ($m = 5$ was used here). Changing the parameter m had little effect on the iteration count, and increasing m increased the execution time because of the overhead for the $m \times m$ blocks.

BiCGSTAB with ILU preconditioning achieved significantly better times on the problems of order 125 and 468 than any other combination of method and preconditioner. In addition, BiCGSTAB was only slightly slower than GMRES(k) on the problems of order 31 and 59. In almost every case, BiCGSTAB required fewer iterations to solve the problems of orders 31, 59, 125 and 468, when compared to the other methods using the same preconditioner.

TABLE 7. GMRES(k) ON UPS01A WITH VARYING k

GMRES(k) on ups01a (N=59)				
	ILU		ILUB	
k	iterations	time	iterations	time
6	13, 36, 24	2.33	12, 36, 24	2.60
7	12, 26, 17	1.70	7, 26, 16	1.83
8	8, 16, 12	1.33	7, 15, 11	1.37
9	8, 14, 9	1.05	7, 13, 9	1.17
10	8, 10, 9	1.02	7, 10, 9	1.14

TABLE 8. GMRES(k) ON VREF WITH VARYING k

GMRES(k) on vref (N=67)				
	ILU		ILUB	
k	iterations	time	iterations	time
17	*		56, 119, 80	5.46
18	71, 285, 139	8.79	47, 95, 74	5.13
19	64, 284, 106	6.87	35, 53, 41	2.94
20	58, 115, 94	6.19	20, 20, 20	1.57
21	45, 158, 83	5.52		
22	40, 74, 54	3.69		
23	23, 50, 43	2.98		
24	23, 47, 30	2.09		
25	23, 25, 24	1.79		

TABLE 9. GMRES(k) ON BGATT WITH VARYING k

GMRES(k) on bgatt (N=125)				
	ILU		ILUB	
k	iterations	time	iterations	time
7	90, 168, 110	12.39	77, 145, 114	13.58
8	62, 141, 98	11.18	63, 134, 96	11.55
9	61, 122, 84	9.65	61, 107, 83	10.12
10	56, 110, 77	8.88	57, 107, 77	9.46
11	52, 81, 68	7.93	53, 86, 67	8.33
12	48, 90, 62	7.31	54, 91, 64	8.06
13	46, 65, 55	6.61	46, 64, 59	7.52
14	41, 66, 54	6.59	42, 66, 53	6.96
15	41, 63, 51	6.29	38, 61, 49	6.45
16	38, 62, 47	5.90	39, 61, 47	6.23
17	34, 53, 44	5.63	34, 53, 44	5.94
18	34, 52, 39	5.04	33, 52, 37	5.21
19	33, 45, 36	4.71	31, 45, 35	4.96
20	32, 40, 35	4.61	27, 37, 33	4.65
21	26, 39, 25	3.40	21, 32, 25	3.77
22	22, 33, 25	3.43	21, 22, 22	3.36
23	22, 24, 22	3.21		
24	22, 24, 22	3.22		

TABLE 10. GMRES(k) ON IS7B WITH VARYING k

GMRES(k) on is7b (N=1854)				
	ILU		ILUB	
k	iterations	time	iterations	time
40	839, 1038, 926	1061.72	748, 1079, 898	1068.90
45	486, 709, 579	705.64	539, 668, 590	748.11
50	343, 444, 375	484.02	341, 448, 375	501.70
55	219, 275, 262	360.38	220, 275, 262	372.43
60	229, 287, 253	360.62	227, 289, 254	373.22
65	188, 237, 206	307.77	187, 237, 201	311.60
70	174, 194, 188	286.26	184, 194, 190	300.64
75	143, 195, 156	256.50	143, 196, 156	264.43
80	139, 156, 147	247.07	140, 156, 148	255.40
85	135, 155, 143	240.97	136, 155, 145	251.33
90	132, 155, 143	245.58	132, 155, 142	250.62
95	92, 152, 125	227.74	91, 151, 125	233.06
100	92, 146, 106	206.17	91, 147, 105	210.64

The results for the six problems shown in Table 4 for GMRES(k) were obtained using k = 6, 7, 20, 11, 25 and 40, respectively. For smaller values of k, GMRES(k) failed to converge at some point along the zero curve if a relatively sparse preconditioner was used. As noted in Section 7.4, larger values of k can make a dramatic difference, but not always. For example, GMRES(20) with ILU preconditioning takes 6.19 seconds on the problem of size 67, but GMRES(25) takes only 1.79 seconds. Similar sensitivity to k occurs for the structural mechanics problems in [19]. The problem of order 67 was solved in considerably less time using GMRES(k) with ILUB preconditioning than with any other combination of method and preconditioner. The problems of order 31 and 59 were solved in slightly less time with GMRES(k) than any other method. Moreover, GMRES(k) was the only method which solved the largest problem.

7.6 Low rank perturbations.

As suggested by Fig. 2, the Jacobian matrices in the test suite can be decomposed as $A = B + E$, where B is symmetrically structured (but not symmetric) and E is a matrix of low rank. Depending on the spectrum or conditioning of the matrix B compared to A, it may be advantageous to use B as the iteration matrix and apply the Sherman-Morrison formula to account for the low rank correction E. This approach has been used effectively in a number of applications [19], and was considered for the circuit simulation problems examined here. Unfortunately there was little difference between the condition numbers and spectra of the Jacobian matrices A and their symmetrically structured components B. Limited experiments with Craig's method and GMRES(k) did not show a significant advantage in overall performance and the approach was abandoned.

Whether an examination of the spectrum of the preconditioned matrix does, in fact, yield much useful information regarding the convergence rate of GMRES(k) is doubtful. For example, the eigenvalues of the eleventh Jacobian matrix for the vref circuit ($N = 67$) were computed after explicitly applying the ILU and ILUB preconditioners. The two spectra are only a Hausdorff distance of 0.18 apart, and have over fifty values in common. The performance of GMRES(k) on these two matrices is, however, strikingly different. Taking

$x_0 = 0$, GMRES(20) converges in only 20 iterations if the ILUB preconditioner is used, but requires 275 iterations when ILU preconditioning is used. Clearly, the spectrum of the preconditioned matrix alone neither predicts nor explains the performance of GMRES(k). This is unsurprising, considering the theory concerning the behavior of GMRES, but the spectrum of the preconditioned coefficient matrix is often cited when discussing the application of GMRES in preconditioned form.

8. VISCOUS FLOW IN A TRIANGULAR CAVITY.

Steady recirculating flows are a common phenomenon in fluid mechanics, occurring in the near wake of moving bluff bodies, in channel flows with abrupt constrictions, or inside partially bounded cavities, the latter commonly arising when there is a fluid flow across the open side of a groove in a surface. The most thoroughly examined example is that of a square cavity, for which both experimental observations and numerical analyses have been published. We consider here the numerical computation of an induced flow within a triangular cavity [27].

In [27] Ribbens et al. consider viscous flow induced within an equilateral triangular cavity by translating one side at a uniform speed. Let Ω' be the equilateral triangle with corners $(-\sqrt{3}a, a)$, $(\sqrt{3}a, a)$, and $(0, -2a)$, and let $\partial\Omega'$ be the boundary of Ω'. Let u' and v' be the velocity components in the Cartesian x' and y' directions, ρ be the density, p' be the pressure, and ν be the kinematic viscosity. Then the governing two-dimensional steady Navier-Stokes equations are

$$\begin{aligned} u'u'_{x'} + v'u'_{y'} &= -\frac{1}{\rho}p'_{x'} + \nu(u'_{x'x'} + u'_{y'y'}), \\ u'v'_{x'} + v'v'_{y'} &= -\frac{1}{\rho}p'_{y'} + \nu(v'_{x'x'} + v'_{y'y'}), \\ u'_{x'} + v'_{y'} &= 0. \end{aligned}$$

Let all moving sides of the triangle Ω' have a velocity of constant magnitude U. Assume boundary conditions such that no slip occurs on the moving sides, the velocity is zero on fixed sides, and that velocities are bounded inside Ω'. Normalize all velocities by U, the pressure by ρU^2, all lengths by a and drop primes. The governing equations in Ω then become

$$\nabla^4\psi = R(\psi_y\nabla^2\psi_x - \psi_x\nabla^2\psi_y),$$

where ∇^2 is the Laplacian operator, R is the Reynolds number Ua/ν, and ψ is a stream function defined by

$$u = \psi_y, \quad v = -\psi_x.$$

Assuming the top side of Ω' is moving (left to right), and the remaining sides are stationary, the boundary conditions are now

$$\psi = 0 \text{ on all three sides of } \Omega,$$

and

$$(\psi_y, -\psi_x) \cdot \mathbf{T} = \begin{cases} 1, & \text{for the moving side} \\ 0, & \text{for the fixed sides} \end{cases},$$

where $\mathbf{T}$ is a unit tangent vector to the boundary pointing in the direction of motion.

In [27] the governing equations in Ω are solved by means of a Newton-like iteration. A Newton-like linearization of the governing equations produces the following linear fourth order PDE to be solved at each iteration

$$\nabla^4 \psi - R\left(\psi_y^{(k)} \nabla^2 \psi_x^{(k)} \psi_y - \psi_x^{(k)} \nabla^2 \psi_y - \nabla^2 \psi_y^{(k)} \psi_x\right) = -R\left(\psi_y^{(k)} \nabla^2 \psi_x^{(k)} - \psi_x^{(k)} \nabla^2 \psi_y^{(k)}\right),$$

where $\psi^{(k)}$ is the approximate solution from the previous step. For a detailed derivation of this result, see Ribbens et al. [28].

As with the circuit simulation problem, the solution of the cavity flow problem requires the solution of a linear system $Ax = b$ at each Newton step. The discretization used in [27] yields a matrix A which is symmetrically structured, indefinite, and highly sparse. If an $m \times m$ grid is used, then the size of A is roughly $\frac{1}{2}m^2$, and there are about 17 nonzeros in each row of A. Figure 11 shows the sparsity pattern for A, using $m = 33$, which yields a linear system of 465 equations and unknowns. Due to the banded structure of A, solution by direct factorization will generate considerable fill and require far more storage than is needed for A. For that reason, it was natural to consider the use of iterative methods.

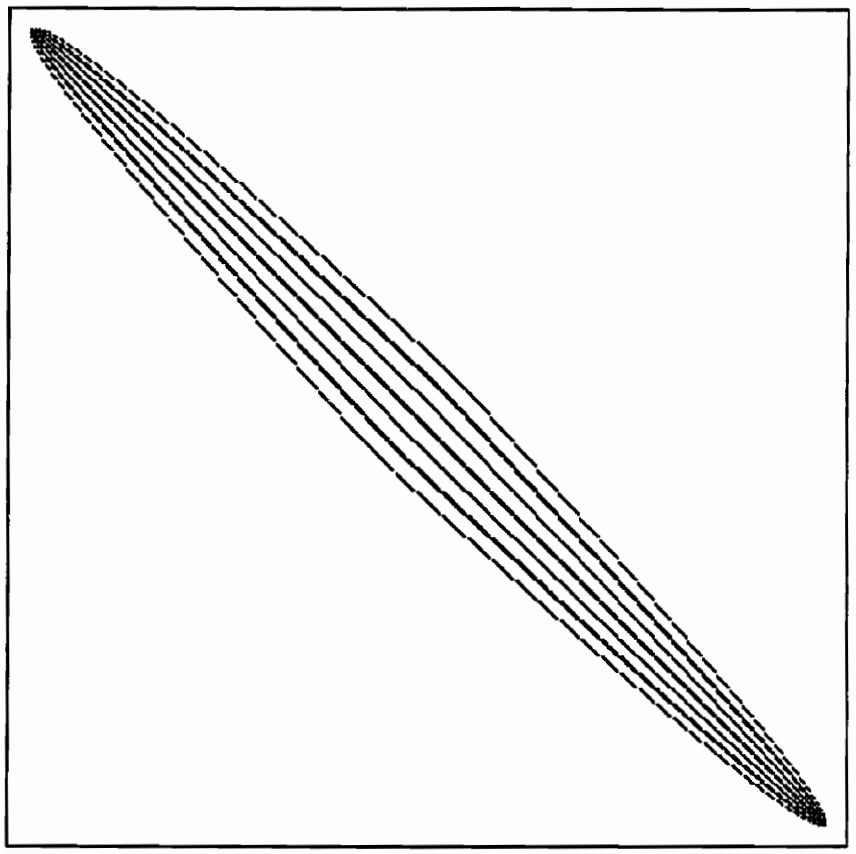

FIG. 11. *Sparsity pattern for the cavity flow problem*

9. NUMERICAL RESULTS FOR THE CAVITY FLOW PROBLEM.

Meaningful analysis of the cavity flow problem requires solving for relatively large Reynolds numbers, perhaps on the order of 1000 or more. In turn, that requires using an increasingly fine mesh for the discretization, in order to avoid computing spurious solutions [32]. However, the numerical results presented here are based on a relatively coarse grid, $m = 33$. That choice was made due to memory constraints and observed performance. The results in here were obtained on a DECstation 5000/200 with 24 megabytes of RAM, and that effectively limited the problem size to $m = 100$. Moreover, using a finer mesh did not have an appreciable effect on the relative performance of the iterative methods tested.

9.1 Effect of Preconditioning.

In view of the results obtained for the circuit simulation problems, several preconditioning schemes were considered for the triangular cavity flow problem. These included ILU, ILUSh, ILU using only the diagonal bands, and several variants of ILU allowing limited fill. The last category included ILU allowing fill along one or two extra bands adjacent to the diagonal bands, and either retaining or discarding the other bands after factorization. The results were uniformly unsatisfying. In each case, the effect of preconditioning was to increase the condition number of A. For ILU and ILUSh the condition number typically increased by a factor of up to 10^3. Each of the schemes allowing limited fill also increased the condition number, although not as dramatically.

In short, the preconditioners used for the circuit simulation problems were not merely ineffective for the cavity flow problem, they were counterproductive. Given the very different nature of the matrices for the cavity flow problem, this is not surprising.

While solving circuit simulation problems, in some cases it was observed that preconditioning reduced the number of iterations a method required for convergence, even when worsening the conditioning. That was not the case for the cavity flow problems. In every test, the unpreconditioned method required fewer iterations, and less time, than if any of the preconditioners was used. Therefore, the numerical results reported in Section 9.2 are for unpreconditioned methods.

9.2 Performance of GMRES(k) and BiCGSTAB.

The experiments with circuit simulation problems indicated that GMRES(k) and BiCGSTAB were the two most effective methods, and only those methods were applied to the triangular cavity flow problem. Each method was applied using a 33×33 grid, which produced a coefficient matrix of order 465 with 7135 nonzeros. Table 11 gives numerical results for both methods. The iteration counts are the total number of iterations performed to solve the problem for the corresponding Reynolds number R, and time is in seconds.

The convergence tolerance was relaxed to approximately 10^{-8}, since that was accurate enough for the Newton iterates to converge. The iteration limit of $5N$ that was used for the circuit simulation problems proved to be too small for the triangular cavity flow problem. Since the primary concern with these problems was storage cost, the number of iterations needed to achieve convergence was of less importance. Therefore the iteration limit was raised to 10^6, allowing essentially an unlimited number of iterations.

In preliminary work, GMRES(k) was applied with various values of k. While increasing k did speed up convergence, increasing k to 100 only reduced execution times by a factor of 2 or 3, while increasing storage cost to an unacceptable level. The most complete set of results corresponded to $k = 20$, and those are reported below. While the results would be different for larger k, experience suggests that the conclusion regarding the relative effectiveness of the two methods would remain the same.

BiCGSTAB was consistently much faster than GMRES(20), by a factor of 10 in some cases. The number of iterations required generally increased with R, but the rate of growth was slower for BiCGSTAB. Finally, BiCGSTAB required far less storage than GMRES(k). The superiority of BiCGSTAB on this test problem seems clear.

TABLE 11. NUMERICAL RESULTS ON THE CAVITY PROBLEM

	GMRES(20)		BiCGSTAB	
R	iterations	time	iterations	time
1	4982	110.05	575	15.60
10	1899	41.35	372	11.20
20	3117	67.82	500	15.23
30	4733	102.18	568	17.00
40	6067	130.67	643	18.58
50	6668	143.53	753	21.32
60	8644	185.75	852	23.70
70	11585	248.37	1166	32.45
80	12477	267.27	1195	32.92
90	14976	320.78	1292	35.40
100	16809	360.03	1581	42.33
110	21405	457.83	1514	40.17
120	20996	449.02	1690	45.07

10. CONCLUSIONS.

Although the results are mixed and the interactions between the specific problem, the preconditioner, and the iterative method are complicated, some well supported conclusions can be drawn.

- When applied to circuit simulation problems, none of the preconditioning schemes appears to be substantially better in terms of robustness or efficiency. Convergence on each problem was either achieved with every applicable preconditioner or not achieved at all. Considering execution times, the ILUSh preconditioner was clearly the least effective. The fact that ILUB preconditioning produced a spectacularly low execution time with GMRES(k) on the problem of order 67 must be balanced against the fact the ILUB preconditioning was associated with the highest execution time for five problem/method combinations. In contrast, ILU preconditioning produced the highest time in only three cases involving the smallest test problem, which was not suitable for ILUB preconditioning. Moreover, in those three cases ILU preconditioning exhibited only a slight disadvantage when compared with ILUSh. If only the three fastest methods are considered, there was only one combination of problem and method in which ILUB preconditioning led to substantially faster convergence than ILU. In summary, ILU preconditioning should be preferred over either ILUSh or ILUB preconditioning; the fancier methods are not worth the extra complexity.

- The importance and difficulty of choosing a preconditioning scheme that is well-suited to the problem at hand are clear, considering the dismal performance of all the schemes on the cavity flow problem. Conventional wisdom suggested that a banded or band-fill preconditioner should be used for those problems. Although the variations considered in this thesis were not exhaustive, the results are far from encouraging. In particular, it seems very unlikely that a single preconditioning scheme can be effective for the general case of an unsymmetric, unstructured matrix.

- Of the iterative methods considered here, GMRES(k) and BiCGSTAB are the best for linear systems with large, sparse, nonsymmetric, indefinite, unstructured coefficient matrices, typified by the circuit simulation and cavity flow problems considered here. GMRES(k) achieved the lowest execution time on four of the six circuit simulation test problems, and

solved the largest problem on which all the other iterative methods failed. However, the speed of GMRES(k) requires a substantial increase in storage, compared to the other iterative methods considered here. Given the extreme sparseness of the Jacobian matrices, typically between $5N$ and $6N$ nonzeros, GMRES(k) required extra storage that was as much as six or seven times that required for storing the Jacobian matrix. If memory usage is a concern, then BiCGSTAB offers an attractive alternative to GMRES(k). Using substantially less storage, BiCGSTAB solved the problems of order 31 and 59 in only slightly more time than required by GMRES(k). BiCGSTAB also achieved the lowest execution times overall on the problems of order 125 and 468. And although GMRES(k) solved the problem of order 67 in roughly 31% of the time required by BiCGSTAB, that required the retention of 20 N-vectors. Despite the very good performance of GMRES(k) on the circuit simulation problems, it was entirely uncompetitive with BiCGSTAB on the cavity flow problem. These results illustrate the difficulty of general predictions regarding the performance of iterative methods for numerical linear algebra.

- Virtually all well known classes of applicable iterative methods have been considered here, and none have acceptable performance for linear systems with large, sparse, nonsymmetric, indefinite, unstructured coefficient matrices arising in circuit simulation. The best algorithms were BiCGSTAB and GMRES(k), but BiCGSTAB failed on the large problem is7b ($N = 1854$), and GMRES(k) requires an unpredictable and unacceptably large value of k to converge. To emphasize just how bad these iterative methods are (for circuit simulation problems, at least), using a proprietary ordering algorithm of AT&T, a direct stable LU factorization *never* generates more than $5N$ fill elements, and takes an order of magnitude less CPU time than GMRES(k) with $k \gg 5$. In addition, the direct solver in ELLPACK managed to solve the cavity flow problem described here in about 40% of the time required by the best iterative method, BiCGSTAB, and did not exhibit increasing execution time as the Reynolds number was increased. Improvements in preconditioning and algorithms are certain, but the gauntlet has been laid down for iterative methods on the type of linear systems considered here.

REFERENCES.

[1] A. BJÖRCK AND T. ELFVING, "Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations," *BIT*, 19 (1979), pp. 145–163.

[2] R. BRAMLEY AND A. SAMEH, "Row projection methods for large nonsymmetric linear systems," Tech. Report No. 957, Center for Supercomputing Research and Development, Univ. Illinois-Urbana, IL, January 1990.

[3] G. BRUSSINO AND V. SONNAD, "A comparison of direct and preconditioned iterative techniques for sparse unsymmetric systems of linear equations," *Int. J. Num. Meth. Eng.*, 28 (1989), pp. 801–815

[4] S. N. CHOW, J. MALLET-PARET, AND J. A. YORKE, "Finding zeros of maps: homotopy methods that are constructive with probability one," *Math. Comput.*, 32 (1978), pp. 887–899.

[5] E. J. CRAIG, "Iteration procedures for simultaneous equations," Ph.D. Thesis, MIT, 1954.

[6] J. E. DENNIS JR. AND K. TURNER, "Generalized Conjugate Directions," *Linear Algebra Applic.*, 88/89 (1987) pp. 187–209.

[7] C. DESA, K. M. IRANI, C. J. RIBBENS, L. T. WATSON, AND H. F. WALKER, "Preconditioned iterative methods for homotopy curve tracking," *SIAM J. Sci. Stat. Comput.*, 13 (1992), pp. 30–46.

[8] H. C. ELMAN, "Iterative methods for large, sparse, nonsymmetric systems of linear equations," Ph.D. Thesis, Computer Sci. Dept., Yale Univ., 1982.

[9] ———, "A stability analysis of incomplete LU factorization," *Math. Comput.*, 47 (1986), pp. 191–218.

[10] R. FLETCHER, "Conjugate gradient methods for indefinite systems," Proc. of the Dundee Biennial Conference on Numerical Analysis, Springer-Verlag, New York, 1975, pp. 73–89.

[11] R. W. FREUND, M. H. GUTKNECHT, AND N. M. NACHTIGAL, "An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, Part I," Tech.

Report 90.45, RIACS, NASA Ames Research Center, Moffett Field, CA, November 1990.

[12] ———, "An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, Part II," Tech. Report 91.09, RIACS, NASA Ames Research Center, Moffett Field, CA, April 1991.

[13] R. W. FREUND AND N. M. NACHTIGAL, "QMR: A quasi-minimal residual method for non-hermitian linear systems," *Numer. Math.*, to appear.

[14] I. GETREU, *Modeling the Bipolar Transistor*, Tektronix Inc., Beaverton, OR, 1976, pp. 9–23.

[15] G. H. GOLUB AND W. KAHAN, "Calculating the singular values and pseudoinverse of a matrix," *SIAM J. Numer. Anal.*, 2 (1965), pp. 205–224.

[16] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations, Second Ed.*, Johns Hopkins University Press, Baltimore, 1989.

[17] I. GUSTAFFSON, "A class of first order factorizations," *BIT*, 18 (1978), pp. 142–156.

[18] S. M. HADFIELD AND T. A. DAVIS, "Analysis of potential parallel implementations of the unsymmetric-pattern multifrontal method for sparse LU factorization," Tech. Report TR-92-017, Dept. of Computer and Information Sci., Univ. of Florida, Gainesville, FL, June 1992

[19] M. R. HESTENES AND E. STIEFEL, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Stand.*, 49 (1952), pp. 409–436.

[20] K. M. IRANI, M. P. KAMAT, C. J. RIBBENS, H. F. WALKER, AND L. T. WATSON, "Experiments with conjugate gradient algorithms for homotopy curve tracking," *SIAM J. Optim.*, 1 (1991), pp. 222–251.

[21] S. KACZMARZ, "Angenäherte auflösung von systemen linearer gleichungen," *Bull. Intern. Acad. Polon. Sci. Class A.*, (1939), pp. 355–357.

[22] C. KAMATH AND A. SAMEH, "A projection method for solving nonsymmetric linear systems on multiprocessors," *Parallel Computing*, 9 (1988/1989), pp. 291–312.

[23] C. LANCZOS, "Solution of systems of linear equations by minimized iterations," *J. Res. Nat. Bur. Stand.*, 49 (1952), pp. 33–53.

[24] J. A. MEIJERINK AND H. A. VAN DER VORST, "An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix," *Math. Comp.*, 31 (1977), pp. 148–162.

[25] R. C. MELVILLE, LJ. TRAJKOVIĆ, S.-C. FANG, AND L. T. WATSON, "Globally convergent homotopy methods for the DC operating point problem," Tech. Report TR-90-61, Dept. of Computer Sci., VPI&SU, Blacksburg, VA, 1990.

[26] N. M. NACHTIGAL, S. C. REDDY, AND L. N. TREFETHEN, "How fast are nonsymmetric matrix iterations?," Preliminary Proceedings of the Copper Mountain Conference on Iterative Methods, April 1990.

[27] C. C. PAIGE AND M. A. SAUNDERS, "LSQR: an algorithm for sparse linear equations and sparse least squares," *ACM Trans. Math. Soft.*, 8 (1982), pp. 43–71.

[28] C. J. RIBBENS, LAYNE T. WATSON, AND C.-Y. WANG, *Steady viscous flow in a triangular cavity*, to appear

[29] C. J. RIBBENS, C.-Y. WANG, LAYNE T. WATSON, AND K. A. ALEXANDER, "Vorticity induced by a moving elliptic belt," *Computer Fluids*, 20, 111 (1991)

[30] Y. SAAD, "SPARSKIT: a basic tool kit for sparse matrix computations," Tech. Report 90.20, RIACS, NASA Ames Research Center, Moffett Field, CA, May 1990.

[31] Y. SAAD AND M. H. SHULTZ, "GMRES: a generalized minimum residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, 7 (1986), pp. 856–869.

[32] R. SCHREIBER AND H. B. KELLER, "Driven cavity flows by efficient numerical techniques," *J. Comp. Phys.*, 49, 310 (1983b)

[33] R. SCHREIBER AND H. B. KELLER, "Spurious solutions in driven cavity calculations," *J. Comp. Phys.*, 49, 165 (1983a)

[34] R. SEDGEWICK, *Algorithms, Second Ed.*, Addison-Wesley, New York, 1988.

[35] P. SONNEVELD, "CGS, a fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, 10 (1989), pp.36–52.

[36] LJ. TRAJKOVIĆ, R. C. MELVILLE, AND S.-C. FANG, "Passivity and no-gain properties establish global convergence of a homotopy method for DC operating points,"

Proc. IEEE Int. Symp. on Circuits and Systems, New Orleans, LA, May, 1990, pp. 914–917.

[37] ———, "Finding DC operating points of transistor circuits using homotopy methods," Proc. IEEE Int. Symp. on Circuits and Systems, Singapore, 1991.

[38] H. F. WALKER, "Implementations of the GMRES method," *Comput. Phys. Comm.*, 53 (1989), pp. 311–320.

[39] ———, "Implementation of the GMRES method using Householder transformations," *SIAM J. Sci. Stat. Comput.*, 9 (1988), pp. 152–163.

[40] H. A. VAN DER VORST, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, 13 (1992), pp. 631–644.

[41] L. T. WATSON, "A globally convergent algorithm for computing fixed points of c^2 maps," *Appl. Math. Comput.*, 5 (1979), pp. 297–311.

[42] ———, "An algorithm that is globally convergent with probability one for a class of nonlinear two-point boundary value problems," *SIAM J. Numer. Anal.*, 16 (1979), pp. 394–401.

[43] ———, "Numerical linear algebra aspects of globally convergent homotopy methods," *SIAM Rev.*, 28 (1986), pp. 529–545.

[44] ———, "Globally convergent homotopy methods: a tutorial," *Appl. Math. Comput.*, 31BK (1989), pp. 529–545.

[45] ———, "A survey of probability-one homotopy methods for engineering optimization," Tech. Report TR-90-47, Dept. of Computer Sci., VPI&SU, Blacksburg, VA, 1990.

[46] L. T. WATSON, S. C. BILLUPS, AND A. P. MORGAN, "HOMPACK: A suite of codes for globally convergent homotopy algorithms," *ACM Trans. Math. Software*, 13 (1987), pp. 281–310.

[47] L. T. WATSON AND D. FENNER, "Chow-Yorke algorithm for fixed points or zeros of c^2 maps," *ACM Trans. Math. Software*, 6 (1980), pp. 252–260.

[48] S. WOLFRAM, *Mathematica: a system for doing mathematics by computer, Second Ed.*, Addison-Wesley Publishing Co., 1991.

[49] Z. ZLATEV, *Computational Methods for General Sparse Matrices*, Kluwer Acad. Pub., 1991.

Appendix A: FORTRAN Code.

The codes for the iterative methods were developed in accordance with the discussion at the beginning of Chapter 3. The coefficient matrix for the linear system to be solved is assumed to be an $N \times N+1$ Jacobian matrix augmented with a row equaling e_k^t for some $1 \leq k \leq N+1$. The codes for Craig's method, GMRES(k) and BiCGSTAB initialize the augmenting row internally, rather than requiring the calling program to perform that task. It is therefore assumed that the data structure for the coefficient matrix A will include exactly two cells for row $N+1$. The called subroutine assigns values and column indices to those cells.

In addition, the dimension passed to each of these subroutines should be the row dimension of the Jacobian matrix, rather than of A; that is, the passed dimension should be N, not $N+1$. The reasons for that are rooted in the development of the Craig's method code from an earlier implementation, and are of little importance or merit.

The iterative codes given here were used with ILU preconditioning. The ILUSh or ILUB preconditioners require separate index arrays, and so their use requires slight modifications to the parameter lists of the subroutines. Finally, the last section of this appendix contains FORTRAN code for the support routines. These include the various factorization routines, triangular solving routines, and matrix-vector produce routines.

A.1 Craig's method.

```
C-----------------------------------------------------------------------
C
      SUBROUTINE CRGNS(NN, AA, LENAA, MAXA, RHO, START, WORK,
     &                 IFLAG, NUMITS)
C
C     Solves a system of linear equations using Craig's method.
C-----------------------------------------------------------------------
C
C     Input variables:
C
C        NN -- dimension of the coefficient matrix minus 1.
C        AA -- one dimensional real array containing the NN+1
C           x NN+1 matrix AA in the general sparse form. The last
C           row of AA will be filled in with e_k^t; the data
C           structure should include two zero-initialized cells for
C           row NN+1. One will be reset to the value 0.0 and the
C           other to the value 1.0.
C        LENAA -- number of elements in the array AA.
C        MAXA -- integer array used to store information about AA. For
C           I = 1, .., NN+1, MAXA(I) is the index of AA in which the
C           the first nonzero of row I is held.  MAXA(NN+2) holds the
C           index of the cell in AA that follows the last nonzero
C           entry.  MAXA(NN+3) is unused.  For I .GE. NN+4, MAXA(I)
C           is the column index for the array entry stored in
C           AA(I - NN - 3).
C        RHO -- vector of length NN+1, negative of top part of right
C           hand side b .
C        START -- vector of length NN+1, initial guess at solution,
C           normally the solution to the previous linear system.
```

```
C       IFLAG -- exit condition flag, should be set to something
C         other than 4 on entry.
C       NUMITS -- iteration limit.
C
C    Output variables:
C
C       START -- solution vector x of  B x = b  (defined above).
C       IFLAG -- normally unchanged on output.  If convergence is
C         not achieved in IMAX iterations, the subroutine terminates
C         with IFLAG = 4 and does not compute x.
C
C    Working storage:
C
C       WORK -- array of length 4*(NN+1) + LENAA :
C            WORK(1..NN+1) = temporary working storage;
C            WORK(NN+2..2NN+2) = storage for current iterate x_k;
C            WORK(2NN+3..3NN+3) = storage for residual vector;
C            WORK(3NN+4..4NN+4) = storage for direction vector;
C            WORK(4NN+5..  *  ) = storage for the preconditioning
C                                 matrix Q.
C
C-----------------------------------------------------------------------
C
C    The following user-defined subroutines are required:
C
C      MUL(y,AA,x,MAXA,NN,LENAA)
C      -- computes y = AA x
C      TMUL(y,AA,x,MAXA,NN,LENAA)
C      -- computes y = AA^t x
C      ILUF(Q, LENAA, APIVOT, NN)
C      -- computes the preconditioning matrix Q based on M.  A copy
C         of AA is placed in Q before the call; after the call, it
C         is assumed that Q contains some factorization for the
C         preconditioning matrix Q.  If no preconditioning is
C         required, ILUF may be a dummy subroutine.
C      LUSOLV(Q, MAXA, NN, LENAA, b)
C      -- computes  INV(Q)*b for any vector b, given the factorization
C         of Q produced by subroutine ILUF.  Again, if no precon-
C         ditioning is required, LUSOLV may be a dummy subroutine.
C      TLUSOLV(Q, MAXA, NN, LENAA, b)
C      -- computes  INV(Q^t)*b for any vector b, given the
C         factorization of Q produced by subroutine ILUF.  Again,
C         if no preconditioning is required, TLUSOLV may be a
C         dummy subroutine.
C
C    Subroutines and functions called:
C
C       BLAS  -- DAXPY, DCOPY, DDOT, DNRM2, DSCAL, IDAMAX
C       USER  -- ILUF, LUSOLV, TLUSOLV, MUL, TMUL
C       OTHER -- D1MACH
C
C
      INTEGER IFLAG,IMAX,J,K,LENAA,NN,MAXA(1),NP1,NP2,N2P3,
     &   N3P4,N4P5,NUMITS, I, NP3
      DOUBLE PRECISION AA(LENAA),AU,BU,DZNRM,
     &   PUNPRD,RHO(1),RNPRD,RUNPRD,RUTOL,
```

```
     &   START(1),STARTK,WORK(1),
     &   ZLEN,ZTOL
      LOGICAL STILLU
C
      DOUBLE PRECISION D1MACH,DDOT,DNRM2
      INTEGER IDAMAX
C
C Set up bases for vectors stored in work array.
C
      NP1=NN+1
      NP2=NN+2
      NP3=NN+3
      N2P3=(2*NN)+3
      N3P4=(3*NN)+4
      N4P5=(4*NN)+5
C
C Find the element of largest magnitude in the initial vector, and
C record its position in k.
C
      K=IDAMAX(NP1,START,1)
      STARTK=START(K)
      CALL DCOPY(NP1,START,1,WORK(NP2),1)
C
C Construct e_k^t in the last row of AA.
C
      IF (K .EQ. NP1) THEN
         AA(MAXA(NP2)-1) = 1.0D0
         MAXA(NP3+LENAA) = NP1
         MAXA(NP2+LENAA) = 1
      ELSE
         AA(MAXA(NP1)) = 1.0D0
         MAXA(NP2+LENAA) = K
         MAXA(NP3+LENAA) = NP1
      ENDIF
      RHO(NP1) = -1.0D0*STARTK
C
C Compute ILU preconditioner.
C
      CALL DCOPY(LENAA, AA, 1, WORK(N4P5), 1)
      CALL ILUF(WORK(N4P5), LENAA, MAXA, NP1)
C
C Compute all tolerances needed for exit criteria.
C
      IMAX=5*NP1
      STILLU=.TRUE.
      ZTOL=D1MACH(4)*1.0D02
      RUTOL=ZTOL
C
C Compute initial residual vector for AAx = rhs.
C
      CALL MUL(WORK(N2P3),AA,WORK(NP2),MAXA,NP1,LENAA)
      CALL DSCAL(NP1,-1.0D0,WORK(N2P3),1)
      CALL DAXPY(NP1,-1.0D0,RHO,1,WORK(N2P3),1)
      CALL LUSOLV(WORK(N4P5),MAXA,NP1,LENAA,WORK(N2P3))
C
C Compute initial direction vector for AAx = rhs.
```

```
C
      CALL DCOPY(NP1,WORK(N2P3),1,WORK,1)
      CALL TLUSOLV(WORK(N4P5),MAXA,NP1,LENAA,WORK)
      CALL TMUL(WORK(N3P4),AA,WORK,MAXA,NP1,LENAA)
      RUNPRD=DDOT(NP1,WORK(N2P3),1,WORK(N2P3),1)
      PUNPRD=DDOT(NP1,WORK(N3P4),1,WORK(N3P4),1)
C
      J=1
C
C Do while ((STILLU) .AND. (J .LE. IMAX))
C
 100  IF (.NOT. ((STILLU) .AND. (J .LE. IMAX)) ) GO TO 200
C
C       IF ||RESIDUAL|| IS STILL NOT SMALL ENOUGH, CONTINUE.
        IF (DSQRT(RUNPRD) .GT. RUTOL) THEN
          IF (PUNPRD .EQ. 0.0) THEN
            CALL MUL(WORK(N2P3),AA,WORK(NP2),MAXA,NP1,LENAA)
            CALL DSCAL(NP1,-1.0D0,WORK(N2P3),1)
            CALL DAXPY(NP1,-1.0D0,RHO,1,WORK(N2P3),1)
            CALL LUSOLV(WORK(N4P5),MAXA,NP1,LENAA,WORK(N2P3))
            CALL DCOPY(NP1,WORK(N2P3),1,WORK,1)
            CALL TLUSOLV(WORK(N4P5),MAXA,NP1,LENAA,WORK)
            CALL TMUL(WORK(N3P4),AA,WORK,MAXA,NP1,LENAA)
            RUNPRD=DDOT(NP1,WORK(N2P3),1,WORK(N2P3),1)
            PUNPRD=DDOT(NP1,WORK(N3P4),1,WORK(N3P4),1)
            IF (DSQRT(RUNPRD) .LE. RUTOL) THEN
              STILLU=.FALSE.
            ENDIF
          ENDIF
          IF (STILLU) THEN
C           Update solution vector.
            AU=RUNPRD/PUNPRD
            CALL DCOPY(NP1,WORK(NP2),1,WORK,1)
            CALL DAXPY(NP1,AU,WORK(N3P4),1,WORK(NP2),1)
            CALL DAXPY(NP1,-1.0D0,WORK(NP2),1,WORK,1)
            ZLEN=DNRM2(NP1,WORK(NP2),1)
            DZNRM=DNRM2(NP1,WORK,1)
C           If relative change in solutions is small enough, exit.
            IF ( (DZNRM/ZLEN) .LT. ZTOL) THEN
              STILLU=.FALSE.
            ENDIF
          ENDIF
        ELSE
          STILLU=.FALSE.
        ENDIF
C
C If no exit criteria have been met, continue.
C
        IF (STILLU) THEN
C         Update residual vector.
          CALL MUL(WORK,AA,WORK(N3P4),MAXA,NP1,LENAA)
          CALL LUSOLV(WORK(N4P5),MAXA,NP1,LENAA,WORK)
          CALL DAXPY(NP1,-AU,WORK,1,WORK(N2P3),1)
          RNPRD=DDOT(NP1,WORK(N2P3),1,WORK(N2P3),1)
C         Update direction vector.
          BU=RNPRD/RUNPRD
```

```
            RUNPRD=RNPRD
            CALL DCOPY(NP1,WORK(N2P3),1,WORK,1)
            CALL TLUSOLV(WORK(N4P5),MAXA,NP1,LENAA,WORK)
            CALL TMUL(START,AA,WORK,MAXA,NP1,LENAA)
            CALL DAXPY(NP1,BU,WORK(N3P4),1,START,1)
            CALL DCOPY(NP1,START,1,WORK(N3P4),1)
            PUNPRD=DDOT(NP1,WORK(N3P4),1,WORK(N3P4),1)
            IF (DSQRT(RUNPRD) .LE. RUTOL) THEN
               STILLU = .FALSE.
            ENDIF
          ENDIF
C
          J=J+1
       GO TO 100
 200   CONTINUE
C
C End Do While
C
C Set error flag if the iteration did not converge.
C
       IF (J .GT. IMAX) THEN
          IFLAG=4
          WRITE (*,*) ' CRGNS LOOP ONE -- IFLAG = ', IFLAG
          RETURN
       ELSE
          NUMITS = J-1
       ENDIF
C
C Compute final solution vector X, return it in START.
C
       CALL DCOPY(NP1,WORK(NP2),1,START,1)
C
       RETURN
       END
```

A.2 GMRES(k).

```
C-----------------------------------------------------------------------
C
      SUBROUTINE GMRES (N, LENAA, A, MAXA, IM, RHS, SOL, SS, EPS,
     &                  MAXITS, IOUT, Q)
C
C     Solves a system of linear equations using the GMRES(k)
C     algorithm --- the value of the parameter IM specifies k.
C-----------------------------------------------------------------------
C     Input variables:
C
C        N -- dimension of the matrix A.
C        A -- one dimensional real array containing the nonzero entries
C           of the N+1 x N+1 sparse matrix.
C        LENAA -- number of nonzero entries in the array A.
C        IM -- value of GMRES(k) parameter k.
C        MAXA -- integer array used to store information about A. For
C           I = 1, .., N+1, MAXA(I) is the index of A in which the
C           first nonzero of row I is held.  MAXA(N+2) holds the
C           index of the cell in A that follows the last nonzero
C           entry.  MAXA(N+3) is unused.  For I .GE. N+4, MAXA(I)
C           is the column index for the array entry stored in
C           A(I - N - 3).
C        RHO -- vector of length N+1, right hand side of the linear
C           system.
C        SOL -- vector of length N+1, initial guess at solution,
C           normally the solution to the previous linear system.
C        Q -- one dimensional real array to hold the nonzero entries
C           of the preconditioning matrix.
C        MAXITS -- upper bound of number iterations.
C        IOUT -- flag for detailed output.  Set to 1 if output is
C           desired, any other value otherwise.
C
C     Output variables:
C
C        SOL -- solution vector x of linear system.
C        MAXITS -- number of iterations performed.
C        EPS -- 2-norm of residual vector on exit.
C
C     Working storage:
C
C        SS(N+1, IM) -- storage for the IM vectors that GMRES(k)
C           generates and saves.
C
C-----------------------------------------------------------------------
C
C     The following user-defined subroutines are required:
C
C       MUL(y,AA,x,MAXA,N,LENAA)
C       -- computes y = AA x
C       ILUF(Q, LENAA, APIVOT, N)
C       -- computes the preconditioning matrix Q based on M.  A copy
C          of AA is placed in Q before the call; after the call, it
C          is assumed that Q contains some factorization for the
C          preconditioning matrix Q.  If no preconditioning is
```

```
C         required, ILUF may be a dummy subroutine.
C       LUSOLV(Q, MAXA, N, LENAA, b)
C       -- computes  INV(Q)*b for any vector b, given the factorization
C          of Q produced by subroutine ILUF.  Again, if no precon-
C          ditioning is required, LUSOLV may be a dummy subroutine.
C
C     Subroutines and functions called:
C
C        BLAS  -- DAXPY, DCOPY, DDOT, DNRM2, DSCAL, IDAMAX
C        USER  -- ILUF, LUSOLV, MUL
C        OTHER -- D1MACH
C
C-----------------------------------------------------------------------
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DOUBLE PRECISION SS(N+1,1), RHS(1), SOL(1), HH(51,50), C(50)
     &       S(50), RS(51), A(1), Q(1), STARTK, EPSMAC
      INTEGER NP1, NP2, NP3, ITS, I, II, I1, IOUT, MAXITS
     &       LENAA, K, IDAMAX, MAXA(1)
C-----------------------------------------------------------------------
      EPSMAC = D1MACH(4)
      ZTOL = 100.0*EPSMAC
      EPS1 = ZTOL
C
C The Arnoldi size should not exceed 50 in this version.  To reset,
C modify the sizes of HH, C, S and RS.
C
      NP1 = N+1
      NP2 = N+2
      NP3 = N+3
      ITS = 0
C
C Compute k, set up last row of A and last entry of RHS.
C
      K = IDAMAX(NP1,SOL,1)
      STARTK = SOL(K)
      IF (K .EQ. NP1) THEN
         A(MAXA(NP2)-1) = 1.0D0
         MAXA(NP3+LENAA) = NP1
         MAXA(NP2+LENAA) = 1
      ELSE
         A(MAXA(NP1)) = 1.0D0
         MAXA(NP2+LENAA)= K
         MAXA(NP3+LENAA) = NP1
      ENDIF
      RHS(NP1) = STARTK
C
C Compute the ILU preconditioner Q.
C
      CALL DCOPY(LENAA, A, 1, Q, 1)
      CALL ILUF(Q, LENAA, MAXA, NP1)
C
C Outer loop starts here:
C
 10   CONTINUE
C
C Compute initial residual vector:
```

```
C
        CALL MUL(SS, A, SOL, MAXA, NP1, LENAA)
        DO 21 J=1,NP1
 21        SS(J,1) = RHS(J) - SS(J,1)
C
C Apply preconditioner to residual vector.
C
        CALL LUSOLV(Q, MAXA, NP1, LENAA, SS)
C
        RO = DSQRT(DDOT(NP1, SS, 1, SS, 1) )
        IF (RO .EQ. 0.0D0) THEN
          RETURN
        ENDIF
        DO 210 J=1, NP1
 210       SS(J,1) = SS(J,1)/RO
        IF (IOUT .GT. 0) THEN
          WRITE (IOUT,*)
          WRITE(IOUT, 199) ITS, RO
        ENDIF
C
C Initialize 1-st term of rhs of Hessenberg system.
C
        RS(1) = RO
        I = 0
 4      I=I+1
        ITS = ITS + 1
        I1 = I + 1
        CALL MUL(SS(1,I1),A,SS(1,I),MAXA,NP1,LENAA)
        CALL LUSOLV(Q,MAXA,NP1,LENAA,SS(1,I1))
C
C Modified Gram-Schmidt.
C
        DO 55 J=1, I
          T = DDOT(NP1, SS(1,J),1,SS(1,I1),1)
          HH(J,I) = T
          CALL DAXPY(NP1, -T, SS(1,J), 1, SS(1,I1), 1)
 55     CONTINUE
        T = DSQRT(DDOT(NP1, SS(1,I1), 1, SS(1,I1), 1))
        HH(I1,I) = T
        DO 57  K=1,NP1
 57        SS(K,I1) = SS(K,I1) / T
C
C Done with modified Gram-Schmidt and Arnoldi step, now update the
C factorization of HH.
C
        IF (I .EQ. 1) GOTO 121
C
C Perform previous transformations on I-th column of H:
C
        DO 66 K=2,I
          K1 = K-1
          T = HH(K1,I)
          HH(K1,I) = C(K1)*T + S(K1)*HH(K,I)
          HH(K,I) = -S(K1)*T + C(K1)*HH(K,I)
 66     CONTINUE
 121    GAM = DSQRT(HH(I,I)**2 + HH(I1,I)**2)
```

```
        IF (GAM .EQ. 0.0D0) GAM = EPSMAC
C
C Determine next plane rotation:
C
        C(I) = HH(I,I)/GAM
        S(I) = HH(I1,I)/GAM
        RS(I1) = -S(I)*RS(I)
        RS(I) =  C(I)*RS(I)
C
C Determine residual norm and test for convergence:
C
        HH(I,I) = C(I)*HH(I,I) + S(I)*HH(I1,I)
        RO = DABS(RS(I1))
        IF (IOUT .GT. 0)
     &     WRITE(IOUT, 199) ITS, RO
        IF ( (I .LT. IM) .AND. (RO .GT. EPS1) .AND.
     &     (ITS .LT. MAXITS) )  GOTO 4
C
C Now compute solution. First solve upper triangular system:
C
        RS(I) = RS(I)/HH(I,I)
        DO 30 II=2,I
           K=I-II+1
           K1 = K+1
           T=RS(K)
           DO 40 J=K1,I
 40           T = T-HH(K,J)*RS(J)
           RS(K) = T/HH(K,K)
 30     CONTINUE
C
C Done with back substitution, now form linear combination to get
C solution.
C
        DO 16 J=1, I
           T = RS(J)
           CALL DAXPY(NP1, T, SS(1,J), 1, SOL,1)
 16     CONTINUE
C
C Restart outer loop when necessary.
C
     IF (RO .GT. EPS1 .AND. ITS .LT. MAXITS) GOTO 10
C
     EPS = (RO / EPS1) * EPS
     MAXITS = ITS
C
     RETURN
 199  FORMAT('   ITS =', I4, '      RES. NORM =', D16.8)
     END
C-----------------------------------------------------------------------
```

A.3 BiCGSTAB.

```
C-----------------------------------------------------------------------
C
      SUBROUTINE CGSTAB(B, X, A, LENAA, MAXA, N, NIT, EPS, RES, R,
     &                              RR, P, S, V, T, Q)
C
C     Solves a system of linear equations using the BiCGSTAB
C     algorithm of H. van der Vorst, based on the pseudocode in:
C
C     "Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG",
C     SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631--644.
C-----------------------------------------------------------------------
C     Input variables:
C
C       B -- vector of length N+1, right hand side of the linear
C          system.
C       X -- vector of length N+1, initial guess at solution,
C          normally the solution to the previous linear system.
C       A -- one dimensional real array containing the nonzero entries
C          of the N+1 x N+1 sparse matrix.
C       LENAA -- number of nonzero entries in the array A.
C       MAXA -- integer array used to store information about A. For
C          I = 1, .., N+1, MAXA(I) is the index of A in which the
C          first nonzero of row I is held.  MAXA(N+2) holds the
C          index of the cell in A that follows the last nonzero
C          entry.  MAXA(N+3) is unused.  For I .GE. N+4, MAXA(I)
C          is the column index for the array entry stored in
C          A(I - N - 3).
C       N -- dimension of the matrix A.
C       NIT -- upper bound on the number of iterations.
C       Q -- one dimensional real array to hold the nonzero entries
C          of the preconditioning matrix.
C
C     Output variables:
C
C       X -- solution vector x of linear system.
C       NIT -- number of iterations performed.
C       EPS -- 2-norm of residual vector on exit.
C
C     Working storage:
C
C       RES -- vector of length at least NIT; holds residual norms
C          from each iteration.
C       R, RR -- vectors of length N+1, residual vector storage.
C       P, S, V, T -- vectors of length N+1.
C
C-----------------------------------------------------------------------
C
C     The following user-defined subroutines are required:
C
C       MUL(y,AA,x,MAXA,N,LENAA)
C       -- computes y = AA x
C       ILUF(Q, LENAA, APIVOT, N)
C       -- computes the preconditioning matrix Q based on M.  A copy
C          of AA is placed in Q before the call; after the call, it
```

```
C         is assumed that Q contains some factorization for the
C         preconditioning matrix Q.  If no preconditioning is
C         required, ILUF may be a dummy subroutine.
C       LUSOLV(Q, MAXA, N, LENAA, b)
C       -- computes  INV(Q)*b for any vector b, given the factorization
C         of Q produced by subroutine ILUF.  Again, if no precon-
C         ditioning is required, LUSOLV may be a dummy subroutine.
C
C     Subroutines and functions called:
C
C        BLAS  -- DAXPY, DCOPY, DDOT, DNRM2, DSCAL, IDAMAX
C        USER  -- ILUF, LUSOLV, MUL
C        OTHER -- D1MACH
C
C-----------------------------------------------------------------------
      INTEGER I, N, NP1, NIT, LENAA, MAXA(1), OUTFLAG
      INTEGER K, NP2, NP3
      DOUBLE PRECISION RES(1),R(1),RR(1),A(1),B(1),X(1),P(1),
     &       S(1),V(1),T(1),Q(1)
      DOUBLE PRECISION RZERO, ALPHA, OMEGA, OLDRHO, RHO, BETA,
     &       EPS, RESI, TEMP1, TEMP2, STARTK
C
      DOUBLE PRECISION DDOT, D1MACH
      INTEGER IDAMAX
C
      OUTFLAG = 1
      NP1 = N+1
      NP2 = N+2
      NP3 = N+3
      EPS = 100.0*D1MACH(4)
C
C Clear vector P.
C
      DO 7 I = 1, NP1
         P(I) = 0.0D0
 7    CONTINUE
C
C Set up last row of A.
C
      K = IDAMAX(NP1, X, 1)
      STARTK = X(K)
C      WRITE (*,*) ' K AND STARTK: ', K, X(K)
      IF (K .EQ. NP1) THEN
         A(MAXA(NP2) - 1) = 1.0D0
         MAXA(NP3 + LENAA) = NP1
         MAXA(NP2 + LENAA) = 1
      ELSE
         A(MAXA(NP1)) = 1.0D0
         MAXA(NP2 + LENAA) = K
         MAXA(NP3 + LENAA) = NP1
      ENDIF
      B(NP1) = X(K)
C
C Compute preconditioner Q.
C
      CALL DCOPY(LENAA, A, 1, Q, 1)
```

```
      CALL ILUF(Q, LENAA, MAXA, NP1)
C
C Compute initial residual r = b - Ax.
C
      CALL MUL(R, A, X, MAXA, NP1, LENAA)
      DO 10 I=1,NP1
         R(I) = B(I) - R(I)
         RR(I) = 1.0D0
 10   CONTINUE
      CALL LUSOLV(Q, MAXA, NP1, LENAA, R)
C
C Compute initial residual norm and initialize BiCG coefficients.
C
      RZERO=DDOT(NP1,R,1,R,1)
      RES(1)=DSQRT(RZERO)
      IF (OUTFLAG .EQ. 1) THEN
         PRINT 111,0,RES(1)
      ENDIF
      ALPHA  = 1.0D0
      OMEGA  = 1.0D0
      OLDRHO = 1.0D0
C
C Begin main loop.
C
      DO 100 K=1,NIT
         RHO = DDOT(NP1,RR,1,R,1)
         BETA = (RHO/OLDRHO)*ALPHA/OMEGA
         OLDRHO = RHO
C
         DO 30 I=1,NP1
            P(I)=R(I)+BETA*(P(I)-OMEGA*V(I))
 30      CONTINUE
C
         CALL MUL(V,A,P,MAXA,NP1,LENAA)
         CALL LUSOLV(Q, MAXA, NP1, LENAA, V)
C
         TEMP1 = DDOT(NP1,RR,1,V,1)
         IF (TEMP1 .EQ. 0.0) THEN
            PRINT 31
 31         FORMAT('   RR*V = 0.0 ')
            NIT=K
            RETURN
         ENDIF
         ALPHA=RHO/TEMP1
C
         DO 50 I=1,NP1
            S(I)=R(I)-ALPHA*V(I)
 50      CONTINUE
C
         CALL MUL(T,A,S,MAXA,NP1,LENAA)
         CALL LUSOLV(Q, MAXA, NP1, LENAA, T)
C
         TEMP2 = DDOT(NP1,T,1,T,1)
         IF (TEMP2 .EQ. 0.0) THEN
            PRINT 51
 51         FORMAT('  T*T = 0 ')
```

```
            NIT=K
            RETURN
          ENDIF
          OMEGA=DDOT(NP1,S,1,T,1)/TEMP2
C
          DO 70 I=1,NP1
            X(I)=X(I)+ALPHA*P(I)+OMEGA*S(I)
            R(I)=S(I)-OMEGA*T(I)
 70       CONTINUE
C
          RESI=DDOT(NP1,R,1,R,1)
          RES(K+1)=DSQRT(RESI)
          IF (OUTFLAG .EQ. 1) THEN
            PRINT 111,K,RES(K+1)
          ENDIF
 111      FORMAT(I5, D20.6)
C
C Test for convergence -- if satisfied, compute true residual.
C
          IF (DSQRT(RESI) .LE. EPS) THEN
            NIT=K
            CALL MUL(V,A,X,MAXA,NP1,LENAA)
            DO 112 I=1,NP1
 112            P(I)=V(I)-B(I)
            CALL LUSOLV(Q, MAXA, NP1, LENAA, P)
C
            RESI=DSQRT(DDOT(NP1,P,1,P,1))
            EPS = RESI
            IF (OUTFLAG .EQ. 1) THEN
              PRINT 113,RESI
            ENDIF
 113        FORMAT('  True residual = ',D13.6)
 115        CONTINUE
            RETURN
          ENDIF
 100  CONTINUE
C
      RETURN
      END
C-----------------------------------------------------------------------
```

A.4 LSQR.

The code for LSQR is not included here due to its considerable length and easy availability. FORTRAN source code for LSQR can be obtained from *netlib* by requesting Algorithm 583 from TOMS. The only modifications used here were to ensure that the convergence test considered the absolute rather than the relative residual norm, and to implement preconditioning. Both are straightforward.

A.5 QMR.

The code for QMR is also not included, again due to its length and availability. FORTRAN source code for QMR can be obtained from the authors by sending e-mail to Noel M. Nachtigal (santa@riacs.edu). It should be noted that the experiments reported in this thesis used version 4 of the QMR code, and there has been a proliferation of updates. The original QMR source code was modified to use an absolute convergence test; preconditioning was already supported in the distritubed implementation.

One additional modification was required, and seemed critical to the successful use of the method. The distributed code set a tolerance to the fourth root of machine epsilon; with that value, QMR failed to converge at some step for most of the circuit simulation problems. Tightening that tolerance to the square root of machine epsilon eliminated the difficulty. It appeared that the looser tolerance was leading the code to conclude (incorrectly) that it had stagnated within an invariant subspace. The tolerance in question was `TOL(2)`, set on line 72 of the subroutine `DSYSLAL()`.

A.6 KACZ.

The code for the row-projection method KACZ was based on the description given in [2]. Aside from storing the matrix A, the primary storage cost related to the term $(A_i^t A_i)^{-1}$ appearing in the projection

$$P_i = A_i(A_i^t A_i)^{-1} A_i^t.$$

Let m be the row dimension of the largest block-row of A. A two-dimensional array, with m rows was allocated to hold the LU factorizations of the potentially full products $A_i^t A_i$, which were used in applying the projections. The computation of the LU factorizations and the backsolving needed to apply the inverse above were based on the code used by the LINPACK routine `DGECO`. Multiplication of a vector of suitable length by A_i or A_i^t was accomplished using a slight modification of the routines MUL and TMUL given in the following section. The modified subroutines accepted two additional arguments, a starting row index and an ending row index, to specify the extent of the block of A to be used. Due to the poor performance of this implementation, and the author's doubts concerning the viability of a purely sequential implementation of the KACZ method, the code is not reproduced here. A copy of the code used may be obtained from the author upon request.

A.7 Ancillary Routines.

```
C-----------------------------------------------------------------------
C
      SUBROUTINE ILUF(B, LENAA, APIVOT, NN)
C
C     Computes the incomplete LU factorization of the matrix B,
C     where B is NNxNN.  B is assumed to be stored in the general
C     sparse scheme described in Chapter 6.
C
C     The method used is that found in TR 89-41: 'Preconditioned
C     conjugate gradient algorithms for homotopy curve tracking',
C     page 10.
C-----------------------------------------------------------------------
C
C     Input variables:
C       B       matrix to be factorized.
C       LENAA   number of entries in B.
C       APIVOT  usual array of row-start and column indices for
C               sparse B.
C       NN      the dimension of B.
C
C     Output variables:
C       B       the ILU factors of input matrix B.
C-----------------------------------------------------------------
C
      DOUBLE PRECISION B(1), SIJ, LIT, LII
      INTEGER LENAA, NN, APIVOT(1)
      INTEGER I, J, COUNT, ISTRT, IFIN, TMAX, NP2, K, T, M
C
      NP2 = NN+2
C
      DO 100 I = 1, NN
         ISTRT = APIVOT(I)
         IFIN  = APIVOT(I+1) - 1
C-------------------------------------  For each element in row I,
C                                      compute the column number
C                                      and T.
         DO 90 COUNT = ISTRT, IFIN
            J = APIVOT(NP2+COUNT)
            TMAX = MIN0(I,J) - 1
            SIJ = B(COUNT)
C-------------------------------------  Compute the corresponding
C                                      sum of products of elements
C                                      of L and U.
            K = ISTRT
 42         T = APIVOT(K + NP2)
            IF (T .LE. TMAX) THEN
              LIT = B(K)
              M = APIVOT(T) + NP2
C-------------------------------------  Find value of U_tj.
 20           IF (APIVOT(M) .LT. J) THEN
                M = M + 1
                GOTO 20
              ENDIF
              IF (APIVOT(M) .EQ. J) SIJ = SIJ - LIT*B(M - NP2)
```

```
            K = K + 1
            GOTO 42
          ENDIF
C-------------------------------------  End of 'T' loop.
          K = NP2 + ISTRT
C-------------------------------------  Find value of L_ii.
 30        IF (APIVOT(K) .LT. I) THEN
            K = K+1
            GOTO 30
          ENDIF
          IF (APIVOT(K) .EQ. I) THEN
            LII = B(K - NP2)
            IF (DABS(LII) .EQ. 0.0) THEN
              LII = 0.00001
              B(K - NP2) = 0.00001
            ENDIF
          ELSE
            LII = 0.00001
          ENDIF
C-------------------------------------  Update L or U, as needed.
          IF (I .GE. J) THEN
            B(COUNT) = SIJ
          ELSE
            B(COUNT) = SIJ/LII
          ENDIF
 90     CONTINUE
C-------------------------------------  End of 'COUNT' loop.
 100  CONTINUE
C
      RETURN
      END
C-----------------------------------------------------------------------
```

```
C-----------------------------------------------------------------------
C
      SUBROUTINE ILUS(A, LENAA, APIVOT, NN, Q, LENQ, QPIVOT,
     &               PCTOD, THRESH)
C-----------------------------------------------------------------------
C
C    This routine creates a sparse submatrix preconditioner, for use
C    with either the threshold- or hybrid-adaptive preconditioning
C    schemes.
C-----------------------------------------------------------------------
C
C  Input variables:
C     A,
C     APIVOT -- matrix to be factorized, stored in the general
C       sparse scheme.
C     LENAA -- number of nonzeros in A.
C     NN -- dimension of A.
C     PCTOD -- target percentage used to determine if the
C       threshold THRESH should be recomputed.
C     THRESH -- threshold for discarding nonzero entries of A.
C       Entries will be discarded if they are less than the value
C       of THRESH. If the value of THRESH upon entry is negative,
C       then the threshold will be recomputed; this routine will
C       reset THRESH to -1.0 before exiting, if the difference
C       between PCTOD and the percentage of off-diagonal entries
C       actually retained exceeds the value of TRIGG.
C
C   Output variables:
C     Q,
C     QPIVOT -- ILUB factorization of A, stored in the general
C        sparse scheme.
C     LENQ   -- number of nonzeros in Q.
C     PCTOD  -- the actual percentage of off-diagonal entries
C        that were retained.
C
C   Subroutines and functions called: SELECT, ILUF
C
C-----------------------------------------------------------------------
C
      DOUBLE PRECISION A(1), Q(1), THRESH, PCTOD
      DOUBLE PRECISION PCTOLD, TRIGG
      INTEGER LENAA, LENQ, NN, APIVOT(1), QPIVOT(1)
      INTEGER NP1, NP2, K
      INTEGER ROW, QK, INDX
C
      NP1 = NN + 1
      NP2 = NN + 2
      TRIGG = 1.5D-1
      PCTOLD = PCTOD
C
C If necessary, (re)compute the value of the threshold.
C
      IF ((THRESH .LT. 0.0) .AND. (PCTOD .LT. 1.0)) THEN
        K = IDNINT((1.0 - PCTOD)*(LENAA - NN))
        QK = 1
        DO 12 ROW = 1, NN
```

```
            DO 13 INDX = APIVOT(ROW), APIVOT(ROW + 1) - 1
               IF (ROW .NE. APIVOT(NP2 + INDX)) THEN
                  Q(QK) = DABS(A(INDX))
                  QK = QK + 1
               ENDIF
 13          CONTINUE
 12       CONTINUE
          LENQ = QK - 1
          CALL SELECT(Q, LENQ, K)
          THRESH = DABS(Q(K))
       ENDIF
C
C Copy the entries of A that are greater than or equal to the
C threshold value into the structure for Q.
C
       QK = 1
       DO 17 ROW = 1, NN
          QPIVOT(ROW) = QK
          DO 18 INDX = APIVOT(ROW), APIVOT(ROW + 1) - 1
             IF ( (DABS(A(INDX)) .GE. THRESH) .OR.
     &            (ROW .EQ. APIVOT(NP2 + INDX)) ) THEN
                Q(QK) = A(INDX)
                QPIVOT(NP2 + QK) = APIVOT(NP2 + INDX)
                QK = QK + 1
             ENDIF
 18       CONTINUE
 17    CONTINUE
       LENQ = QK - 1
       QPIVOT(NN + 1) = LENQ + 1
C
C Compute % off-diagonal entries used.
C
       PCTOD = DBLE(LENQ - NN)/DBLE(LENAA - NN)
       IF (DABS(PCTOLD - PCTOD) .GT. TRIGG) THRESH = -1.0
C
C Call ILUF to factorize Q, then return.
C
       CALL ILUF(Q, LENQ, QPIVOT, NN)
C
       RETURN
       END
C-----------------------------------------------------------------------
```

```
C-----------------------------------------------------------------------
      SUBROUTINE ILUB(A, LENAA, APIVOT, NN, Q, LENQ, QPIVOT,
     &                BLKSTRT, BLKEND)
C-----------------------------------------------------------------------
C
C   This routine creates the ILUB preconditioner.  It takes the
C   matrix A and its associated index array as input, initializes
C   the values in the preconditioner array Q and its index array,
C   expanding the diagonal blocks of A to allow fill, and then calls
C   the ILU factorization code.
C-----------------------------------------------------------------------
C
C   Input variables:
C     A,
C     APIVOT -- matrix to be factorized, stored in the general
C       sparse scheme.
C     LENAA  -- number of nonzeros in A.
C     NN     -- dimension of A.
C     BLKSTRT,
C     BLKEND -- rows in which first diagonal block begins and
C       in which last diagonal block ends.
C
C   Output variables:
C     Q,
C     QPIVOT -- ILUB factorization of A, stored in the general
C       sparse scheme.
C     LENQ   -- number of nonzeros in Q.
C
C-----------------------------------------------------------------------
C
C   Subroutines and functions called: ILUF
C
C-----------------------------------------------------------------------
C
      DOUBLE PRECISION A(1), Q(1)
      INTEGER LENAA, LENQ, NN, APIVOT(1), QPIVOT(1)
      INTEGER NP1, NP2, K
      INTEGER BLKSTRT, BLKEND, BSIZE
      INTEGER BROW, CSTRT, CSTOP
      INTEGER ROW, COL, QK, INDX, TMPTR, MINDX
C
      NP1 = NN + 1
      NP2 = NN + 2
      BSIZE = 6
      BROW = 0
C
C Copy the entries of A into the structure for Q, inserting cells
C to fill out the diagonal blocks.
C
C This section handles the NW corner of A.
C
      QK = 1
      QPIVOT(1) = 1
      DO 10 ROW = 1, BLKSTRT - 1
        DO 5 INDX = APIVOT(ROW), APIVOT(ROW + 1) - 1
          Q(QK) = A(INDX)
```

```
            QPIVOT(NP2 + QK) = APIVOT(NP2 + INDX)
            QK = QK + 1
 5        CONTINUE
          QPIVOT(ROW + 1) = QK
 10     CONTINUE
C
C This section handles the part of A that has blocks.
C
       DO 20 ROW = BLKSTRT, BLKEND
          CSTRT = MAX0(ROW - BROW, 1)
          CSTOP = MIN0(ROW + BSIZE - BROW - 1, NN)
          MINDX = APIVOT(ROW + 1) - 1
          INDX = APIVOT(ROW)
 11       COL = APIVOT(NP2 + INDX)
          IF (COL .LT. CSTRT) THEN
             Q(QK) = A(INDX)
             QPIVOT(NP2 + QK) = COL
             QK = QK + 1
             INDX = INDX + 1
             GOTO 11
          ENDIF
          TMPTR = QK
          DO 21 K = CSTRT, CSTOP
             Q(QK) = 0.0D0
             QPIVOT(QK + NP2) = K
             QK = QK + 1
 21       CONTINUE
 12       IF ((COL .LE. CSTOP) .AND. (INDX .LE. MINDX)) THEN
 17          IF (QPIVOT(NP2 + TMPTR) .LT. COL) THEN
                TMPTR = TMPTR + 1
                GOTO 17
             ENDIF
             Q(TMPTR) = A(INDX)
             INDX = INDX + 1
             COL = APIVOT(NP2 + INDX)
             GOTO 12
          ENDIF
 13       IF (INDX .LE. MINDX) THEN
             Q(QK) = A(INDX)
             QPIVOT(NP2 + QK) = APIVOT(NP2 + INDX)
             QK = QK + 1
             INDX = INDX + 1
             GOTO 13
          ENDIF
          BROW = MOD(BROW + 1, BSIZE)
          QPIVOT(ROW + 1) = QK
 20     CONTINUE
C
C This section handles the SE corner of A.
C
       DO 30 ROW = BLKEND + 1, NN
          DO 25 INDX = APIVOT(ROW), APIVOT(ROW+1) - 1
             Q(QK) = A(INDX)
             QPIVOT(NP2 + QK) = APIVOT(NP2 + INDX)
             QK = QK + 1
 25       CONTINUE
```

```
        QPIVOT(ROW + 1) = QK
 30   CONTINUE
      LENQ = QK - 1
C
C Call ILUF to factorize Q, then return.
C
      CALL ILUF(Q, LENQ, QPIVOT, NN)
C
      RETURN
      END
C-----------------------------------------------------------------------
```

```
C-----------------------------------------------------------------
C
      SUBROUTINE LUSOLV(Q, PIVOT, NN, LENAA, B)
C
C     Computes INV(Q)*B -- returns result as B.  Note the dimension
C     of the local array DIAG restricts the size of Q and B.
C-----------------------------------------------------------------
C
C     Input variables:
C
C       Q       triangular factors of preconditioning matrix, stored
C               in the general sparse scheme described in chapter 6.
C       PIVOT    the usual integer array of row and column indices.
C       NN      dimension of Q -- note this is the dimension of the
C               whole matrix -- so NN here corresponds to NN+1 in
C               the subroutine CRGNS().
C       LENAA    number of data entries in Q.
C       B       right hand side -- should have dimension NN.
C
C     Output variables:
C
C       B       solution of Bx = b.
C
C-----------------------------------------------------------------
C
      INTEGER NN, LENAA, PIVOT(LENAA+NN+2)
      DOUBLE PRECISION Q(LENAA), B(NN)
      INTEGER DIAG(2000), I, K, NP2, J
C
      NP2 = NN + 2
C---------------------------------------------- compute B = INV(L)*B
      B(1) = B(1)/Q(1)
      DIAG(1) = 1
      DO 100 I = 2, NN
         K = PIVOT(I)
 42      J = PIVOT(K + NP2)
         IF (J .LT. I) THEN
            B(I) = B(I) - Q(K)*B(J)
            K = K + 1
            GOTO 42
         ELSE
            DIAG(I) = K
            B(I) = B(I)/Q(K)
         ENDIF
 100  CONTINUE
C---------------------------------------------- compute B = INV(U)*B
      DO 200 I = NN-1, 1, -1
         DO 43 K = DIAG(I)+1, PIVOT(I+1)-1
            B(I) = B(I) - Q(K)*B(PIVOT(K+NP2))
 43      CONTINUE
 200  CONTINUE
      RETURN
      END
C-----------------------------------------------------------------
```

```
C-----------------------------------------------------------------------
C
      SUBROUTINE TLUSOLV(Q, PIVOT, NN, LENAA, B)
C
C     Computes INV(Q^t)*B -- returns result as B.  See header for
C     LUSOLV() for additional details on data structures.
C-----------------------------------------------------------------------
C
C     Input variables:
C
C       Q       the LU factorization of A
C       PIVOT   the usual array of indices for sparse matrix
C       NN      the dimension of Q
C       LENAA   number of entries in Q
C       B       right hand side
C
C     Output variables:
C
C       B       solution of (Q^t)X = B
C
C-----------------------------------------------------------------------
C
      INTEGER NN, LENAA, PIVOT(LENAA+NN+2)
      DOUBLE PRECISION Q(LENAA), B(NN)
      INTEGER DIAG(2000), I, K, J, NP2
C
      NP2 = NN + 2
C------------------------------------------- compute b = INV(U^t)*b
      DO 100 J = 1, NN-1
         K = PIVOT(J)
 42      I = PIVOT(K+NP2)
         IF (I .LT. J) THEN
            K = K+1
            GOTO 42
         ELSE
            DIAG(J) = K
         ENDIF
         DO 43 I = K+1, PIVOT(J+1)-1
            B(PIVOT(I+NP2)) = B(PIVOT(I+NP2)) - Q(I)*B(J)
 43      CONTINUE
 100  CONTINUE
C--------------------------------------- find diagonal entry in row NN
      K = PIVOT(NN)
 44   I = PIVOT(K+NP2)
      IF (I .LT. NN) THEN
         K = K+1
         GOTO 44
      ENDIF
      DIAG(NN) = K
C------------------------------------------- compute b = INV(L^t)*b
      DO 200 J = NN, 2, -1
         B(J) = B(J)/Q(DIAG(J))
         DO 45 K = PIVOT(J), DIAG(J)-1
            B(PIVOT(K+NP2)) = B(PIVOT(K+NP2)) - Q(K)*B(J)
 45      CONTINUE
 200  CONTINUE
```

```
      B(1) = B(1)/Q(DIAG(1))
C
      RETURN
      END
C-----------------------------------------------------------------------
```

```
C-----------------------------------------------------------------------
C
      SUBROUTINE MUL(Y, B, X, MAXA, N, LENAA)
C
C     Returns B*X as Y.
C-----------------------------------------------------------------------
C
C     Input variables:
C
C       B,
C       MAXA    matrix stored in the usual sparse scheme.
C       NN      dimension of B -- note this is the dimension of the
C               whole matrix -- so NN here corresponds to NN+1 in
C               the subroutine CRGMS().
C       LENAA   number of data entries in B.
C       X       source vector -- should have dimension NN.
C       Y       target vector -- should have dimension NN.
C
C     Output variables:
C
C       Y       value of B*X.
C
C-----------------------------------------------------------------------
C
      INTEGER I,K,STRT,FIN,LENAA,NP2,N
      INTEGER MAXA(LENAA+N+2)
      DOUBLE PRECISION B(LENAA),X(N),Y(N),TMP
      NP2 = N+2
C
      DO 10 I = 1, N
         STRT = MAXA(I)
         FIN = MAXA(I+1)-1
         TMP = 0.0D0
         DO 20 K = STRT, FIN
            TMP = TMP + B(K)*X(MAXA(K+NP2))
 20      CONTINUE
         Y(I) = TMP
 10   CONTINUE
C
      RETURN
      END
C-----------------------------------------------------------------------
```

```
C-----------------------------------------------------------------------
C
      SUBROUTINE TMUL(Y, B, X, MAXA, N, LENAA)
C
C     Subroutine returns (B^t)*X as Y.
C-----------------------------------------------------------------------
C
C     Input variables:
C
C       B,
C       MAXA    matrix stored in the usual sparse scheme.
C       NN      dimension of B -- note this is the dimension of the
C               whole matrix -- so NN here corresponds to NN+1 in
C               the subroutine CRGNS().
C       LENAA   number of data entries in B.
C       X       source vector -- should have dimension NN.
C       Y       target vector -- should have dimension NN.
C
C     Output variables:
C
C       Y       value of (B^t)*X.
C
C-----------------------------------------------------------------------
C
      INTEGER I,K,STRT,FIN,LENAA,NP2,J,N
      INTEGER MAXA(LENAA+N+2)
      DOUBLE PRECISION B(LENAA),X(N),Y(N)
      NP2 = N+2
C
      DO 7 I = 1,N
 7       Y(I) = 0.0D0
C
      DO 10 I = 1, N
         STRT = MAXA(I)
         FIN = MAXA(I+1)-1
         DO 20 K = STRT, FIN
            J = MAXA(K+NP2)
            Y(J) = Y(J) + X(I)*B(K)
 20      CONTINUE
 10   CONTINUE
C
      RETURN
      END
C-----------------------------------------------------------------------
```

```
C-----------------------------------------------------------------------
      SUBROUTINE SELECT(ARR, LEN, K)
C
C     Order statistic routine to locate Kth largest element in the
C     array ARR.  On exit, ARR(K) is the Kth largest element in ARR.
C
C     Taken from 'Algorithms, 2nd Ed.' by Sedgewick.
C-----------------------------------------------------------------------
C
C     Input variables:
C       ARR -- array of length LEN
C       K   -- specifies order of element of ARR to be located
C
C     Output variables:
C       ARR(K) -- Kth largest element in ARR
C-----------------------------------------------------------------------
C
      INTEGER LEN, K, L, R, I, J
      DOUBLE PRECISION ARR(1), T, VAL
C
      L = 1
      R = LEN
C
 10   IF (R .GT. L) THEN
        VAL = ARR(R)
        I = L-1
        J = R
C
 1      CONTINUE
 5      I = I+1
        IF (ARR(I) .LT. VAL) GOTO 5
 7      J = J-1
        IF (ARR(J) .GT. VAL) GOTO 7
        T = ARR(I)
        ARR(I) = ARR(J)
        ARR(J) = T
        IF (J .GT. I) GOTO 1
C
        ARR(J) = ARR(I)
        ARR(I) = ARR(R)
        ARR(R) = T
        IF (I .GE. K) R = I-1
        IF (I .LE. K) L = I+1
        GOTO 10
      ENDIF
C
      RETURN
      END
C-----------------------------------------------------------------------
```

VITA.

William D. McQuain was born on September 1, 1954, in Staunton, Virginia. He earned a BA degree in philosophy in 1975, a BS degree in mathematics in 1976, and an MS degree in mathematics in 1978, all from Virginia Polytechnic Institute and State University, Blacksburg, Virginia. He was employed as an instructor of mathematics at James Madison University from 1979 to 1980, at Louisiana State University from 1980 to 1984 and at Virginia Polytechnic Institute and State University from 1984 to 1990. In August, 1992, he received the MS degree in computer science from Virginia Polytechnic Institute and State University.

William D McQuain