PRECONDITIONED SEQUENTIAL AND PARALLEL CONJUGATE GRADIENT ALGORITHMS FOR HOMOTOPY CURVE TRACKING

by

Kashmira M. Irani

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

for partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Layne T. Watson

Dr. Layne T. Watson, Chairman

Calvin J. Ribbens

Dr. Calvin J. Ribbens

Terry L Herdman

Dr. Terry L. Herdman

October, 1990

Blacksburg, Virginia

PRECONDITIONED SEQUENTIAL AND PARALLEL CONJUGATE GRADIENT ALGORITHMS FOR HOMOTOPY CURVE TRACKING

by

Kashmira M. Irani

Dr. Layne T. Watson, Chairman

Computer Science

(ABSTRACT)

There are algorithms for finding zeros or fixed points of nonlinear systems of equations that are globally convergent for almost all starting points, i.e., with probability one. The essence of all such algorithms is the construction of an appropriate homotopy map and then tracking some smooth curve in the zero set of this homotopy map. HOMPACK is a mathematical software package implementing globally convergent homotopy algorithms with three different techniques for tracking a homotopy zero curve, and has separate routines for dense and sparse Jacobian matrices. The HOMPACK algorithms for sparse Jacobian matrices use a preconditioned conjugate gradient algorithm for the computation of the kernel of the homotopy Jacobian matrix, a required linear algebra step for homotopy curve tracking. Variants of the conjugate gradient algorithm along with different preconditioners are implemented in the context of homotopy curve tracking and compared with Craig's preconditioned conjugate gradient method used in HOMPACK. In addition, a parallel version of Craig's method with incomplete LU factorization preconditioning is implemented on a shared memory parallel computer with various levels and degrees of parallelism (e.g., linear algebra, function and Jacobian matrix evaluation, etc.). An in-depth study is presented for each of these levels with respect to the speedup in execution time obtained with the parallelism, the time spent implementing the parallel code and the extra memory allocated by the parallel algorithm.

Acknowledgements

I would like to express my appreciation to my advisor and committee chairman, Dr. L. T. Watson for his support, guidance and patience. His insight was very helpful at crucial points in my research. I would also like to thank my committee members Dr. C. J. Ribbens and Dr. T. L. Herdman for their time and effort. Special thanks are due to my parents; their unconditional support and encouragement throughout my life has been invaluable to me.

CONTENTS

List of Figures

List of Tables

1. Introduction

The fundamental problem motivating this thesis is to solve a nonlinear system of equations $F(x) = 0$, where $F : E^n \rightarrow E^n$ is a C^2 map defined on real n-dimensional Euclidean space E^n. The homotopy approach to solving $F(x) = 0$ is to construct a continuous map $H(\lambda, x)$, the "homotopy," deforming a simple function $s(x)$ to the given function $F(x)$ as λ varies from 0 to 1. Starting from the easily obtained solution to $H(0, x) = s(x) = 0$, the essence of a homotopy algorithm is to track solutions of $H(\lambda, x) = 0$ until a solution of $H(1, x) = F(x) = 0$ is obtained. The theoretical and implementational details of such algorithms are nontrivial, and significant progress on both aspects has been made recently [39], [54].

Homotopies are a traditional part of topology, and only recently have begun to be used for practical numerical computation. The (globally convergent probability-one) homotopies considered here are sometimes called "artificial-parameter generic homotopies", in contrast to natural-parameter homotopies, where the homotopy variable is a physically meaningful parameter. In the latter case, which is frequently of interest, the resulting homotopy zero curves must be dealt with as they are, bifurcations, ill-conditioning, etc. The homotopy zero curves for artificial-parameter generic homotopies obey strict smoothness conditions, which generally will not hold if the homotopy parameter represents a physically meaningful quantity, but they can always be obtained via certain generic constructions using an artificial (i.e., nonphysical) homotopy parameter. Not just any random perturbation will suffice to create a globally convergent probability-one (generic) homotopy, e.g., the perturbation implied by discretization is generally not sufficient to produce a probability-one homotopy map.

If the objective is to solve a "parameter-free" system of equations, $F(x) = 0$, then ex-

tra attention can be devoted to constructing the homotopy, and the curve-tracking algorithm can be limited to a well-behaved class of curves. The goal of using these globally convergent probability-one homotopies is to solve fixed-point and zero-finding problems with homotopies whose zero curves do not have bifurcations and other singular and ill-conditioned behavior. The mathematical software package HOMPACK used here for comparative purposes is designed for globally convergent probability-one homotopies.

The theory and algorithms for functions $F(x)$ with small dense Jacobian matrices $DF(x)$ are well developed, which is not the case for large sparse $DF(x)$. Solving large sparse nonlinear systems of equations via homotopy methods involves sparse rectangular linear systems of equations and iterative methods for the solution of such sparse systems. Preconditioning techniques are used to make the iterative methods more efficient. The first part of this thesis compares the performance of several variants of the preconditioned conjugate gradient algorithm to that of Craig's method with Gill Murray preconditioning [13],[21],[54], the algorithm implemented in HOMPACK. The second part describes several parallel versions of a preconditioned Craig's algorithm and analyses the speedup obtained by each version vis-à-vis the time spent on the implementation and the extra memory required by the algorithm at each level. The test problems used include actual large scale, sparse structural mechanics problems.

Chapter 2 gives a brief description of homotopy algorithms for solving nonlinear systems of equations and discusses the zero-finding problem and the normal flow homotopy algorithm, which is the algorithm used for solving the problems in this thesis. Chapter 3 introduces iterative methods for solving invertible linear systems. Chapter 4 discusses the linear algebra details of homotopy curve tracking and various algorithmic possibilities for that. Chapter 5 gives a brief description of the three test problems that were used for the experiments. Chapter 6 presents the numerical results of the implementation of the various

sequential algorithms on the test problems. Chapter 7 describes the parallel work that was done and discusses the results obtained from the various parallel algorithms that were implemented. Some general conclusions from all the experiments are drawn in Chapter 8.

2. Globally Convergent Homotopy Algorithms

The philosophy of globally convergent probability-one homotopy algorithms is to create homotopies whose zero curves are well behaved with well-conditioned Jacobian matrices and that reach a solution for almost all choices of a parameter. These homotopies are used to solve fixed-point and zero-finding problems, with homotopies that avoid bifurcations and other singular and ill-conditioned behavior.

The fixed-point problem is to solve $x = f(x)$, where $f : B \to B$ is a C^2 map and B the closed unit ball in n-dimensional real Euclidean space E^n. Define the map $\rho_a : [0,1) \times B \to E^n$ by

$$\rho_a(\lambda, x) = \lambda(x - f(x)) + (1 - \lambda)(x - a). \tag{1}$$

The fundamental result [10] 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

$$F(x) = 0, \tag{2}$$

where $F : E^n \to E^n$ is a C^2 map, is more complicated. Suppose 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

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

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 [56], [57] 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. Large nonlinear systems of equations with sparse symmetric Jacobian matrices occur in many engineering disciplines (the symmetry in the problems of interest here is due to the fact that the Jacobian matrix is actually the Hessian of a potential energy function). In this thesis, we consider only the zero finding problem $F(x) = 0$, the normal flow curve tracking algorithm, and large sparse symmetric Jacobian matrices $DF(x)$ stored in a packed skyline data structure.

Consider the homotopy map

$$\rho_a(x,\lambda) = \lambda F(x) + (1-\lambda)(x-a). \tag{3}$$

The matrix $D_x\rho_a(x,\lambda) = \lambda DF(x) + (1-\lambda)I$ is symmetric and sparse with a "skyline" structure, such as

$$A = \begin{pmatrix} \bullet_1 & \bullet_3 & & \bullet_9 & \\ \bullet & \bullet_2 & \bullet_5 & \bullet_8 & \\ & \bullet & \bullet_4 & \bullet_7 & \bullet_{12} \\ \bullet & \bullet & \bullet & \bullet_6 & \bullet_{11} \\ & & \bullet & \bullet & \bullet_{10} \end{pmatrix}.$$

Such matrices are typically stored in packed skyline format, in which the upper triangle is stored in a one-dimensional indexed array as shown above. An auxiliary integer array $(1,2,4,6,10,13)$ of length $(n+1)$ of diagonal indices is also required with the $(n+1)th$ element containing the length of the packed array plus one. Assuming that $F(x)$ is C^2, a is such that the Jacobian matrix $D\rho_a(x,\lambda)$ has full rank along γ, and γ is bounded, the zero curve γ is C^1 and can be parameterized by arc length s. Thus $x = x(s), \lambda = \lambda(s)$ along γ, and

$$\rho_a(x(s),\lambda(s)) = 0$$

identically in s.

The zero curve γ given by $(x(s),\lambda(s))$ is the trajectory of the initial value problem

$$\frac{d}{ds}\rho_a(x(s),\lambda(s)) = [D_x\rho_a(x(s),\lambda(s)), D_\lambda\rho_a(x(s),\lambda(s))]\begin{pmatrix} dx/ds \\ d\lambda/ds \end{pmatrix} = 0, \tag{4}$$

$$\left\| \left(\frac{dx}{ds}, \frac{d\lambda}{ds} \right) \right\|_2 = 1, \tag{5}$$

$$x(0) = a, \quad \lambda(0) = 0. \tag{6}$$

Since the Jacobian matrix has rank n along γ, the derivative $(dx/ds, d\lambda/ds)$ is uniquely determined by (4), (5) and continuity, and the initial value problem (4–6) can be solved for $x(s),\lambda(s)$. From (4) it can be seen that the unit tangent $(dx/ds, d\lambda/ds)$ to γ is in the kernel of $D\rho_a$.

The normal flow curve tracking algorithm has four phases: prediction, correction, step size estimation, and computation of the solution at $\lambda = 1$. For the prediction phase, assume that two points $P^{(1)} = (x(s_1), \lambda(s_1))$, $P^{(2)} = (x(s_2), \lambda(s_2))$ on γ with corresponding tangent vectors $(dx/ds(s_1),\ d\lambda/ds(s_1))$, $(dx/ds(s_2),\ d\lambda/ds(s_2))$ have been found, and h is an estimate of the optimal step (in arc length) to take along γ. The prediction of the next point on γ is

$$Z^{(0)} = p(s_2 + h), \tag{7}$$

where $p(s)$ is the Hermite cubic interpolating $(x(s), \lambda(s))$ at s_1 and s_2. Precisely,

$$p(s_1) = (x(s_1), \lambda(s_1)), \quad p'(s_1) = (dx/ds(s_1), d\lambda/ds(s_1)),$$

$$p(s_2) = (x(s_2), \lambda(s_2)), \quad p'(s_2) = (dx/ds(s_2), d\lambda/ds(s_2)),$$

and each component of $p(s)$ is a polynomial in s of degree less than or equal to 3.

Starting at the predicted point $Z^{(0)}$, the corrector iteration is

$$Z^{(k+1)} = Z^{(k)} - \left[D\rho_a(Z^{(k)})\right]^+ \rho_a(Z^{(k)}), \qquad k = 0, 1, \ldots \tag{8}$$

where $\left[D\rho_a(Z^{(k)})\right]^+$ is the Moore-Penrose pseudoinverse of the $n \times (n+1)$ Jacobian matrix $D\rho_a$. Small perturbations of a produce small changes in the trajectory γ, and the family of trajectories γ for varying a is known as the "Davidenko flow". Geometrically, the iterates given by (8) return to the zero curve along the flow normal to the Davidenko flow, hence the name "normal flow algorithm".

A corrector step ΔZ is the unique minimum norm solution of the equation

$$\left[D\rho_a\right]\Delta Z = -\rho_a. \tag{9}$$

Fortunately ΔZ can be calculated at the same time as the kernel of $\left[D\rho_a\right]$, and with just a little more work. The numerical linear algebra details for solving (9), the optimal step size

estimation, and the endgame to obtain the solution at $\lambda = 1$ are in [56], [57].

The calculation of the implicitly defined derivative $(dx/ds, d\lambda/ds)$ is done by computing the one-dimensional kernel of $D\rho_a$, i.e., by solving the $n \times (n+1)$ linear system $[D\rho_a]y = 0$. This can be elegantly and efficiently done for small dense matrices [49], [50], but the large sparse Jacobian matrix presents special difficulties. The difficulty now is that the first n columns of the Jacobian matrix $D\rho_a(x, \lambda)$ involving $DF(x)$ are definitely special, and any attempt to treat all $n+1$ columns uniformly would be disastrous from the point of view of storage allocation. Hence, what is required is a good algorithm for solving nonsquare linear systems of equations (9) where the leading $n \times n$ submatrix $D_x\rho_a$ of $D\rho_a$ is symmetric and sparse. In the next chapter, we consider various iterative methods for solving such linear systems of equations.

3. Iterative Methods For Invertible Linear Systems

In the last chapter we observed that in the course of homotopy curve tracking, we need to solve nonsquare linear systems of equations for the tangent vector and the normal flow iteration calculations. These nonsquare systems of the form

$$(B \quad f)\, y' = b',$$

are converted to equivalent square linear systems of the form

$$Ay = \begin{pmatrix} B & f \\ c^t & d \end{pmatrix} y = b, \tag{10}$$

where the $n \times n$ matrix B is bordered by the vectors f and c to form a larger system of dimension $(n+1)\times(n+1)$. In the present context $B = D_x\rho_a(x,\lambda)$ is symmetric and sparse, but A is not necessarily symmetric. Techniques for solving linear systems are classified as either direct methods or iterative methods. In the absence of roundoff error, direct methods compute the exact solution $y := A^{-1}b$ with a finite number of numerical operations.

Direct methods generally use some form of Gaussian elimination [25]. When the rows of the matrix A are ordered appropriately, A is factored into the product

$$A = LU,$$

where L is a lower triangular matrix and U is an upper triangular matrix. The solution is then computed by solving successively

$$Lx = b \text{ and } Uy = x.$$

These methods are most suitable for solving dense systems and densely packed sparse systems (such as banded systems), but they have drawbacks that limit their usefulness for general sparse systems. The main difficulties stem from the fact that the factors L and

U tend to have many more nonzeros than the coefficient matrix A. Thus, more storage is required for the factors than for the original matrix. The number of arithmetic operations needed to compute the factorization can also become larger than is desirable. Hence it may not be practical to solve large sparse problems by direct methods.

Iterative methods are generally used for solving these linear systems. (If B has only a couple nonpositive eigenvalues, direct methods may be a viable alternative; this issue is addressed later in this chapter.) Iterative methods compute a sequence of approximate solutions $\{y_i\}$ which converge to the exact solution y by some algorithm of the form

$$y_{i+1} = F_i(y_0, y_1, \ldots, y_i),$$

where y_0 is an arbitrary initial guess and F_i may be linear or nonlinear. Iterative methods require the coefficient matrix A in the algorithm, generally only to compute matrix-vector products. Since matrix-vector computations are quite inexpensive for sparse problems, iterative methods have low computational cost per iteration. Iterative methods are also attractive because they have low storage requirements, due to the fact that at each iteration, only a small number of vectors of length $N = n + 1$ need to be computed and stored to calculate the next iterate y_{i+1}, and A itself can be generated or stored compactly. Thus iterative methods are sometimes more attractive than direct methods for solving large sparse linear systems of equations.

Iterative methods such as the successive over-relaxation (SOR) algorithm [45] and the alternating direction implicit (ADI) algorithm [59] require the estimation of scalar parameters, which is a drawback. The conjugate gradient procedure [24] is an efficient algorithm for solving symmetric positive definite systems which requires no such estimates. For nonsymmetric problems, iterative methods like the conjugate gradient method can be applied but convergence to the solution is not guaranteed. For many years, the only iterative methods

known to converge for general nonsymmetric problems were the conjugate gradient method applied to the normal equations [24] and Lanczos' biconjugate gradient algorithm [31]. Other early conjugate gradient-like methods for nonsymmetric problems which avoided the use of the normal equations were the generalized conjugate gradient method of Concus and Golub [11] and Widlund [58], and Orthomin by Vinsome [46]. These methods only apply to matrices with positive definite symmetric part, although with preconditioning they can be used to solve more general problems [18]. Other conjugate gradient-like methods for more general problems were proposed by Axelsson [1], Eisenstat, Elman, and Schultz [17], Jea [26], Saad [41], Young and Jea [60]–[61], and Saad and Schultz [43]. Preconditioning techniques that have been effective for symmetric, positive definite systems include the incomplete LU factorization [31], [32], the modified incomplete LU factorization [15], [22], and the SSOR preconditioning [59]. Most of these extend naturally to nonsymmetric problems. A lot of work has also been done comparing these various iterative methods and the preconditioning techniques [9], [14], [18], [43]. Unfortunately very little of this existing theory is directly applicable to the sparse linear systems arising from homotopy curve tracking.

The rate of convergence of conjugate gradient-type methods depends on the symmetry, inertia, spectrum, and condition number of the coefficient matrix. There are efficient conjugate gradient algorithms for solving linear systems with symmetric positive definite coefficient matrices, whereas no comparable theory exists for general systems with nonsymmetric or indefinite A. The first part of the numerical experiments compares the relative performance of conjugate gradient-type algorithms for solving nonsymmetric or indefinite linear systems of the form $Ax = b$ arising from globally convergent homotopy algorithms, in terms of execution time, storage requirements, and the number of iterations required to converge.

Let Q be a $N \times N$ nonsingular matrix. The solution to $Ax = b$ can also be obtained

by solving the system:

$$\tilde{A}x = (Q^{-1}A)x = Q^{-1}b = \tilde{b}.$$

The use of such an auxiliary matrix is known as *preconditioning.* The goal of preconditioning is to decrease the computational effort required to solve linear systems of equations by increasing the rate of convergence of an iterative method. For preconditioning to be effective, the faster convergence must outweigh the costs of applying the preconditioning, so that the total cost of solving the linear system is lower. The preconditioned coefficient matrix $\tilde{A}$ is usually not explicitly computed or stored. The main reason for this is that although A is sparse, $\tilde{A}$ may not be. The extra work of preconditioning, then, occurs in the preconditioned matrix-vector products involving Q^{-1}. The main storage cost for preconditioning is usually for Q, which typically is stored so that one extra array is required to handle the preconditioning operation.

3.1. Alternatives for solving square linear systems.

There are three main approaches to solving (10):

(1) In the block factorization approach to the problem, a block elimination algorithm is used instead of working with the whole matrix A directly. Such an algorithm would take advantage of the special properties of the submatrix B.

(2) Here A is split into the sum of a symmetric matrix M and a low rank correction L. These methods also take advantage of the fact that the leading submatrix B is symmetric and can use conjugate gradient algorithms requiring a symmetric coefficient matrix.

(3) The general approach works directly with the whole matrix A without taking any special advantage of the fact that the submatrix B contained in A is symmetric.

3.1.1. Approach 1 – block factorization methods.

The linear system (10) can also be written as

$$Ay = \begin{pmatrix} B & f \\ c^t & d \end{pmatrix} \begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} b' \\ b'' \end{pmatrix}.$$

A block-elimination algorithm [5] would be:

factor B;

solve $Bv = f$;

solve $Bw = b'$;

compute $y'' = (b'' - c^t w)/(d - c^t v)$;

compute $y' = w - y''v$.

With such block factorization methods, the work consists mainly of one factorization of B (assuming that is possible) and two backsolves with the factors of B. Observe that block elimination will frequently fail in the homotopy context, because even though rank $A = n+1$ and rank $(B \quad f)$ = rank $D\rho_a = n$, it may very well happen that $B = D_x\rho_a$ is singular (rank $B = n - 1$). Singular B can be handled by deflation techniques [5]–[8], resulting in a direct algorithm very similar to other direct algorithms discussed below under matrix splittings. If the deflated systems were solved iteratively, this would constitute yet another iterative algorithm with no apparent advantage over the other iterative algorithms considered here. Deflation and block elimination will not be considered further.

3.1.2. Approach 2 – coefficient matrix splittings.

There are several ways of splitting the coefficient matrix A in (10) as the sum of a symmetric matrix M and a low rank matrix L. The choice (c^t, d) as the last row of M gives the splitting

$$M = \begin{pmatrix} B & c \\ c^t & d \end{pmatrix}, \qquad L = ue_{n+1}^t, \qquad u = \begin{pmatrix} f - c \\ 0 \end{pmatrix}, \tag{11}$$

where e_{n+1} is a vector with 1 in the $(n+1)$st component and zeros elsewhere. There are many reasonable choices for (c^t, d), discussed later (recall that (c^t, d) can be almost any, in the sense of Lebesgue measure, vector for which (10) produces a solution to the true problem (9) or $[D\rho_a]y = 0$). The linear system $Ay = b$ is then solved by applying iterative techniques to two linear systems with coefficient matrix M followed by the Sherman-Morrison formula; the algorithmic details of this are in the next section. Another possibility would be to compute a symmetric indefinite factorization of M, and not use iterative methods at all. However, this destroys the skyline data structure containing M, and a tacit assumption here is that the skyline data structures must be preserved. If it were acceptable to destroy the skyline data structure, this direct approach would likely be the most efficient of all for skyline sparsity patterns, but would not generalize to arbitrary sparsity patterns (which the iterative methods will). We describe two closely related methods for solving linear systems with a symmetric positive definite coefficient matrix A, namely the conjugate gradient method and the conjugate residual method.

The conjugate gradient procedure (CG) [24] is an efficient algorithm for solving symmetric positive definite systems. The CG method computes a sequence of iterates $\{y_i\}$ converging to a solution of (10), in at most N iterations starting with an initial arbitrary guess y_0. It is a polynomial-based algorithm [9] which has a strong minimization property. Let $r_i = b - Ay_i$, $i = 0, 1, \ldots$ denote the residuals of the conjugate gradient iterates and consider the i-dimensional Krylov space

$$S_i = \langle r_0, Ar_0, \ldots, A^{i-1}r_0 \rangle.$$

At each step, CG computes the point $y_i \in y_0 + S_i$ that minimizes the error functional

$$E_{CG}(y_i) = (y - y_i, A(y - y_i))^{1/2} = \|y - y_i\|_A.$$

The CG algorithm is:

choose y_0;

set $r_0 = b - Ay_0$;

set $p_0 = r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(r_i, r_i)}{(p_i, Ap_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

$$r_{i+1} = r_i - a_i A p_i;$$

$$b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)};$$

$$p_{i+1} = r_{i+1} + b_i p_i;$$

end

Note that for this algorithm and the others following this, (x, y) denotes the inner product of x and y. The work per iteration for this algorithm is 5N multiplications plus one matrix-vector product. In addition to A, storage is required only for four vectors of length N.

The error bound at the ith step is

$$E_{CG}(y_i) \leq 2\left[\frac{1 - 1/\sqrt{K(A)}}{1 + 1/\sqrt{K(A)}}\right]^i E_{CG}(y_0),$$

where $K(A) = \|A\|_2 \|A^{-1}\|_2$ is the condition number of the matrix A with respect to the 2-norm. Then an approximate upper bound on the number of iterations required to make the relative error $E_{CG}(y_i)/E_{CG}(y_0) \leq \epsilon$ is

$$i = \left(\frac{\log(2/\epsilon)}{2}\right)\sqrt{K(A)}$$

iterations.

The conjugate residual method (CR) is closely related to the conjugate gradient method, differing mainly in the inner product and the error functional associated with it. CR is also an iterative algorithm whose iterates $\{y_i\}$ minimize the error functional

$$E_{CR}(y_i) = (A(y - y_i), A(y - y_i))^{1/2} = \|b - Ay_i\|_2$$

over the translated i-dimensional Krylov space

$$y_0 + \langle r_0, Ar_0, \ldots, A^{i-1}r_0 \rangle.$$

Like the CG method, this method also computes y in at most N steps and requires no parameter estimates. The CR algorithm is:

choose y_0;

set $r_0 = b - Ay_0$;

set $p_0 = r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(r_i, Ar_i)}{(Ap_i, Ap_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

$$r_{i+1} = r_i - a_i Ap_i;$$

$$b_i = \frac{(r_{i+1}, Ar_{i+1})}{(r_i, Ar_i)};$$

$$p_{i+1} = r_{i+1} + b_i p_i;$$

$$Ap_{i+1} = Ar_{i+1} + b_i Ap_i;$$

end

Conjugate Residual is slightly more expensive than Conjugate Gradient. The work per iteration is 6N multiplications plus one matrix-vector product. Storage is required for the vectors y, r, p, Ap, Av in addition to A.

The error bound at the ith step is

$$E_{CR}(y_i) \leq 2\left[\frac{1-1/\sqrt{K(A)}}{1+1/\sqrt{K(A)}}\right]^i E_{CR}(y_0),$$

so that

$$\left(\frac{\log(2/\epsilon)}{2}\right)\sqrt{K(A)}$$

is an approximate upper bound on the number of iterations required to make the relative error $E_{CR}(y_i)/E_{CR}(y_0) \leq \epsilon$.

Another way of splitting up the coefficient matrix A is

$$A = D - A_L - A_U, \tag{12}$$

where D is the diagonal of A, A_L is the strict lower triangle of $-A$, and A_U is the strict upper triangle of $-A$. The symmetric successive over-relaxation (SSOR) iterative method [59] is the following two stage algorithm:

$$(D - \omega A_L)y_{i+1/2} = [(1-\omega)D + \omega A_U]y_i + \omega b,$$

$$(D - \omega A_U)y_{i+1} = [(1-\omega)D + \omega A_L]y_{i+1/2} + \omega b,$$

where ω is a real scalar parameter between 0 and 2. With

$$Q = \frac{1}{\omega(2-\omega)}(D - \omega A_L)D^{-1}(D - \omega A_U),$$

this method can be formulated as a one step algorithm

$$Qy_{i+1} = (Q - A)y_i + b.$$

In the homotopy context, D^{-1} frequently does not exist, and a diagonal matrix Σ such that $\left[\text{diag}\,(A+\Sigma)\right]^{-1}$ does exist may not be of low rank (meaning that the solution for A cannot be easily recovered from the solution for $A+\Sigma$). Consequently SSOR and methods based on similar splittings are of limited utility in the homotopy context; in fact, SSOR failed for all the test problems described in Chapter 5. A few experiments were also tried with SSOR ($\omega = 1$) as a preconditioner, but it was not competitive, and is not considered further here.

3.1.3. Approach 3 – general methods.

These algorithms work on the nonsymmetric A directly. If y_0 is an initial approximation of y, and r_0 the corresponding residual vector $r_0 = b - Ay_0$, then the Krylov subspace methods consist of finding an approximate solution belonging to the affine subspace $y_0 + K_j$, where K_j is the Krylov subspace generated by $r_0, Ar_0, \ldots A^{j-1}r_0$. There are several such methods like Craig's method[13], Orthomin(k) [46], Orthodir and Orthores [61], the Incomplete Orthogonalization Method [42], the GCR method [18], GMRES method[44]. The algorithms described here can also be used for approach 2, i.e., the splitting approach, but the conjugate gradient and the conjugate residual algorithms cannot be applied to nonsymmetric general matrices. We describe these algorithms in the following paragraphs.

One of the earliest iterative methods known to converge for general nonsymmetric problems is the conjugate gradient method applied to the normal equations. Consider solving system (10) where the matrix A is nonsingular but need not be symmetric or positive definite. This problem is equivalent to the normal equations

$$A^t A y = A^t b, \tag{13}$$

and to the similar system

$$AA^t x = b, \quad y = A^t x. \tag{14}$$

Since the coefficient matrices in (13) and (14) are symmetric and positive definite, a natural way to use CG to solve nonsymmetric problems is to apply it to either one of these two problems.

When CG is used to solve (13) the iterates y_i minimize the residual norm $\|r_i\|_2$ over the translated Krylov space

$$y_0 + \langle A^t r_0, (A^t A) A^t r_0, \ldots, (A^t A)^{i-1} A^t r_0 \rangle.$$

This method is called the Conjugate Gradient Applied To Normal Equations With Minimum Residual [CGNR]. The CGNR algorithm is:

choose y_0;

set $r_0 = b - Ay_0$;

set $p_0 = A^t r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(A^t r_i, A^t r_i)}{(Ap_i, Ap_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

$$r_{i+1} = r_i - a_i A p_i;$$

$$b_i = \frac{(A^t r_{i+1}, A^t r_{i+1})}{(A^t r_i, A^t r_i)};$$

$$p_{i+1} = A^t r_{i+1} + b_i p_i;$$

end

CG can also be used to solve (14) in which y is computed directly, without reference to x, any approximations of x, or AA^t. This method, due to Craig[13], is called the Conjugate

Gradient Applied to Normal Equations With Minimum Error (CGNE) and is described in [19] and [23].

The iterates y_i minimize the error norm $\|y - y_i\|_2$ over the translated Krylov space

$$y_0 + \langle A^t r_0, A^t(AA^t)r_0, \ldots, A^t(AA^t)^{i-1}r_0 \rangle.$$

The CGNE algorithm is:

choose y_0;

set $r_0 = b - Ay_0$;

set $p_0 = A^t r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(r_i, r_i)}{(p_i, p_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

$$r_{i+1} = r_i - a_i A p_i;$$

$$b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)};$$

$$p_{i+1} = A^t r_{i+1} + b_i p_i;$$

end

For this algorithm, a minimum of $5(n+1)$ storage locations is required (in addition to that for A). The vectors y, $\tilde{r}$, and p all require their own locations; $Q^{-t}\tilde{r}$ can share with Ap; $Q^{-1}Ap$ can share with $A^tQ^{-t}\tilde{r}$. The computational cost per iteration of this algorithm is:

(a) two preconditioning solves ($Q^{-1}v$ and $Q^{-t}v$);

(b) two matrix-vector products (Av and A^tv);

(c) $5(n+1)$ multiplications (the inner products (p,p) and $(\tilde{r},\tilde{r})$, ap, bp, and $aQ^{-1}Ap$.

For both versions of CGN without preconditioning, the work per loop is 5N multiplications plus two matrix-vector products. Besides A, storage is required for the vectors y, r, p and Ap. The upper bound for the error at the ith step is

$$E_{CGN}(y_i) \leq 2\left[\frac{1-1/K(A)}{1+1/K(A)}\right]^i E_{CGN}(y_0),$$

where

$$E_{CGN}(y_i) = \begin{cases} \|b - Ay_i\|_2, & \text{for CGNR;} \\ \|y - y_i\|_2, & \text{for CGNE.} \end{cases}$$

And so

$$\left(\frac{\log(2/\epsilon)}{2}\right) K(A)$$

is an approximate upper bound on the number of iterations required to make the relative error $E_{CGN}(y_i)/E_{CGN}(y_0) \leq \epsilon$.

These bounds illustrate the main drawback of *CGN*. The upper bound is larger by a factor of $\sqrt{K(A)}$ than the analogous number for *CG* applied directly to a problem with a symmetric positive definite matrix, see [18] for a precise statement. This suggests that if A is poorly conditioned, then the convergence of *CGN* could be slow.

Consider system (10) where A is nonsymmetric but has positive definite symmetric part $(A + A^T)/2$. We survey four descent methods that arise from CR by combining the CR solution update

$$y_{i+1} = y_i + a_i p_i$$

with some kind of modification of the direction vectors $\{p_i\}$ where

$$p_{i+1} = r_{i+1} + b_i p_i.$$

The approximate solution obtained at each step minimizes the residual norm over some subspace of a Krylov space based on A. We wish to compute new directions $\{p_i\}$ that

produce significant decreases in the residual norms $\{\|r_i\|\}$ at each iteration, with as little cost and storage as possible.

The general form of the algorithm for these methods is as follows:

Choose y_0;

set $r_0 = b - Ay_0$;

set $p_0 = r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(r_i, Ap_i)}{(Ap_i, Ap_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

$$r_{i+1} = r_i - a_i Ap_i;$$

compute p_{i+1};

end

Generalization 1 – Generalized Conjugate Residual (GCR).

A set of directions can be computed for use in the CR algorithm as those given by the formula

$$p_{i+1} = r_{i+1} + \sum_{j=0}^{i} b_j^{(i)} p_j,$$

where

$$b_j^{(i)} = -\frac{(Ar_{i+1}, Ap_j)}{(Ap_j, Ap_j)} \qquad j \leq i.$$

With these directions, the algorithm works under the stated assumptions but the storage requirements and the work per step are exceedingly high when the matrix A is large, because all the previous directions p_j, $j = 1, 2, \ldots, i-1$ have to be stored in order to compute the

new direction p_i. This leads to the following generalization, which is an attempt to limit the cost and storage per iteration.

Generalization 2 – Orthomin(k).

A modification of GCR that is significantly less expensive per step is derived by limiting the number of direction vectors used to compute p_{i+1} allowing only k directions to be used in the computation of the new direction. Use the most recently computed directions $\{p_j\}_{j=i-k+1}^{i}$ with p_{i+1} chosen to be A^tA-orthogonal to these vectors :

$$p_{i+1} = r_{i+1} + \sum_{j=i-k+1}^{i} b_j^{(i)} p_j.$$

Since only k directions need to be stored, the storage requirements for Orthomin(k) make it feasible.

Generalization 3 – GCR(k).

Another alternative is to restart GCR periodically, i.e., after every $(k+1)$ iterations, the current iterate is taken as a new starting point. For this method, the storage requirements are the same as for Orthomin(k). However, the cost per iteration is lower, since on average $k/2$ direction vectors are used to compute the next direction p_{i+1}.

Generalization 4 – Minimal Residual Method.

For the special case $k = 0$, the new direction $p_{i+1} = r_{i+1}$. This alternative to the GCR has very modest work and storage requirements, and in the symmetric positive definite case, resembles the method of steepest descent.

Hence using the general form of the algorithm given above, typically, the preconditioned Orthomin(k) algorithm is given by:

choose y_0;

set $r_0 = b - Ay_0$;

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

set $p_0 = r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(\tilde{r_i}, Q^{-1}Ap_i)}{(Q^{-1}Ap_i, Q^{-1}Ap_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

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

$$b_j^{(i)} = -\frac{(Q^{-1}A\tilde{r}_{i+1}, Q^{-1}Ap_j)}{(Q^{-1}Ap_j, Q^{-1}Ap_j)}, \qquad j = \max\{0, i-k+1\}, \ldots i;$$

$$p_{i+1} = \tilde{r}_{i+1} + \sum_{j=(i-k+1)_+}^{i} b_j^i p_j;$$

end

where $(i-k+1)_+ \equiv \max\{0, i-k+1\}$. As for the storage costs, Ap_j is overwritten by $Q^{-1}Ap_j$ and $A\tilde{r}$ by $Q^{-1}A\tilde{r}$. Thus storage is required for y, $\tilde{r}$, $\{p_j\}_{(i-k+1)_+}^{i}$, $\{Q^{-1}Ap_j\}_{(i-k+1)_+}^{i}$, and $Q^{-1}A\tilde{r}$.

However, note that these generalizations of the conjugate residual method are guaranteed to converge only for positive definite coefficient matrices A (equivalently, A with positive definite symmetric part $(A + A^t)/2$.) (Some authors define *positive definite* only for symmetric matrices, while others say A is positive definite if $x^tAx > 0$ for all $x \neq 0$ in E^n, whether A is symmetric or not. This latter meaning is used here.) More general systems $Ax = b$, where A is not positive definite, can be solved by applying these algorithms to the transformed system $ZAx = Zb$, where Z is nonsingular and ZA is positive definite. The matrix Z must be known and used explicitly in the iteration, a major obstacle to the general applicability of these methods.

The GCR method may also break down if the coefficient matrix is not positive defi-

nite. Orthodir(k), introduced by Young and Jea [60],[61], is another generalization of the conjugate residual method which can be used to solve indefinite problems. It differs from the conjugate residual method in the computation of the direction vectors, the computation being more expensive in this case.

The preconditioned Orthodir(k) algorithm is given by:

choose y_0;

set $r_0 = b - Ay_0$;

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

set $p_0 = r_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(\tilde{r_i}, Q^{-1}Ap_i)}{(Q^{-1}Ap_i, Q^{-1}Ap_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

$$\tilde{r}_{i+1} = \tilde{r_i} - a_i Q^{-1}Ap_i;$$

$$b_j^{(i)} = -\frac{((Q^{-1}A)^2 p_i, Q^{-1}Ap_j)}{(Q^{-1}Ap_j, Q^{-1}Ap_j)}, \qquad j = \max\{0, i-k+1\}, \ldots i;$$

$$p_{i+1} = Q^{-1}Ap_i + \sum_{j=(i-k+1)_+}^{i} b_j^i p_j;$$

end

where $(i-k+1)_+ \equiv \max\{0, i-k+1\}$. The storage for Orthodir(k) is the same as that for Orthomin(k). Although the untruncated Orthodir algorithm is guaranteed to converge even for coefficient matrices which are not positive definite, Orthodir(k) may not necessarily converge. However, for indefinite problems, Orthodir, though convergent, is observed to have stability problems [42].

GMRES, which is equivalent to GCR for positive definite coefficient matrices, can be used to solve systems for which the coefficient matrix is not positive definite. However, GMRES requires storage of the order of the number of iterations performed for convergence. Hence, the algorithm is used iteratively, i.e., it is restarted every k steps, where k is a fixed parameter, leading to the following GMRES(k) algorithm [44]:

choose y_0, tol;

set $r_0 = b - Ay_0$;

while $\|r_0\| >$ tol **do**

begin

set $v_1 = r_0/\|r_0\|$;

for $j = 1$ **step** 1 **until** k **do**

begin

for $i = 1$ **step** 1 **until** j **do** $h_{i,j} = (Av_j, v_i)$;

$$\tilde{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i;$$

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

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

end

Solve $\min_x \| \|r_0\| e_1 - \bar{H}_k x \|$ for x_k where $\bar{H}_k$ is described in [44];

set $y_0 = y_0 + V_k x_k$; **set** $r_0 = b - Ay_0$

end

In practice the algorithm calculates $\|r_j\|$ ($\|r_j\|$ can be calculated without forming y_j or $r_j = b - Ay_j$ explicitly) at each iteration of the j loop, and breaks the j loop if $\|r_j\| <$ tol

[44], [47]. Also, it is important in practice that the classical Gram-Schmidt process in the inner j loop be replaced by the modified Gram-Schmidt process to ensure stability. GMRES(k), like Orthomin(k), is guaranteed to converge when the coefficient matrix is positive definite. However, for an indefinite coefficient matrix, GMRES(k), while it does not break down, may fail because the residual norms at each step, although nonincreasing, do not converge to zero.

The SYMMLQ algorithm of Paige and Saunders [36], like GMRES, does not require the coefficient matrix A to have any special properties. However, it can only be applied to linear systems with symmetric coefficient matrix A. It is based on a computational variant of the Lanczos procedure for tridiagonalizing a symmetric matrix. When A is symmetric positive definite, the SYMMLQ algorithm is equivalent to the conjugate gradient algorithm. Unlike CG, SYMMLQ is well defined and numerically stable even when A is indefinite.

SYMMLQ generates a sequence of iterates $\{y_i\}$, which minimize the error functional $(y - y_i, A(y - y_i))$, over the translated space $y_0 + \langle r_0, Ar_0, \ldots, A^{i-1}r_0 \rangle$. The SYMMLQ algorithm is:

choose y_0;

set $r_0 = b - Ay_0$; **set** $\beta_0 = \|b\|$; **set** $v_{-1} = 0$;

set $v_0 = \dfrac{b}{\|b\|}$; **set** $\bar{\xi}_0 = 1$; **set** $\bar{\gamma}_0 = \beta_0$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

while $\|r_i\| >$ tol **do**

begin

$\alpha_i = v_i^t A v_i$;

$v_{i+1} = Av_i - \alpha_i v_i - \beta_i v_{i-1}$;

$\beta_{i+1} = \|v_{i+1}\|$;

$\gamma_i = (\bar{\gamma}_i + \beta_{i+1}{}^2)^{1/2}$;

$c_i = \bar{\gamma}_i/\gamma_i$;

$\delta_i = \beta_{i+1}/\gamma_i$;

$\xi_i = c_i\bar{\xi}_i$;

compute $(w_i \quad \bar{w}_{i+1}) = (\bar{w}_i \quad v_{i+1}) \begin{pmatrix} c_i & \delta_i \\ \delta_i & -c_i \end{pmatrix}$ where $\bar{w}_0 = v_0$;

$y^l_{i+1} = y^l_i + \xi_i w_i$;

$r_{i+1} = b - Ay^l_{i+1}$;

$k = i + 1$;

end;

$y = y^l_k + \dfrac{\xi_{k-1}\delta_{k-1}}{c_{k-1}\bar{w}_k}$;

end

This method requires storage for A, y, w, Av and the two most recent v's. The cost per iteration is one matrix-vector product plus $9N$ multiplications.

3.2. Some preconditioning techniques.

This section considers some preconditioning techniques to be used in conjunction with the algorithms just described. Preconditioning matrices constructed from approximate factorizations of the coefficient matrix are considered first. A lower triangular matrix L and an upper triangular matrix U that are in some sense approximations of the factors in the LU factorization of A, but that are also sparse, are constructed. The preconditioning matrix is the product $Q = LU$. The heuristic used to insure that the preconditioning is inexpensive to implement is to force the factors to be sparse by allowing nonzeros only within a specified set of locations.

3.2.1. The incomplete LU factorization (ILU).

Let Z be a set of indices contained in $\{(i,j) \mid 1 \leq i,j \leq N, i \neq j\}$, typically where A is known to be zero. The incomplete LU factorization is given by $Q = LU$, where L and U are lower triangular and unit upper triangular matrices, respectively, that satisfy

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

The incomplete LU factorization algorithm is:

for $i = 1$ **step** 1 **until** N **do**

  for $j = 1$ **step** 1 **until** N **do**

    if $((i,j) \notin Z)$ **then**

      begin

$$s_{ij} = A_{ij} - \sum_{t=1}^{\min\{i,j\}-1} L_{it}U_{tj};$$

      if $(i \geq j)$ **then** $L_{ij} = s_{ij}$ **else** $U_{ij} = s_{ij}/L_{ii}$;

      end

It can happen that L_{ii} is zero in this algorithm. In this case L_{ii} is set to a small positive number, so that $Q_{ii} \neq A_{ii}$.

3.2.2. The modified incomplete LU factorization (MILU).

Let Z be the set of indices that determine the zero structure, and assume that $(i,i) \notin Z$, $1 \leq i \leq N$. The modified incomplete LU factorization is given by $Q = LU$, where L and U are lower triangular and unit upper triangular matrices, respectively, that satisfy

$$\begin{cases} L_{ij} = U_{ij} = 0, & (i,j) \in Z, \\ Q_{ij} = A_{ij}, & (i,j) \notin Z, i \neq j, \\ \sum_{j=1}^{N}(Q_{ij} - A_{ij}) = \alpha, & 1 \leq i \leq N, \end{cases}$$

where α is a scalar. The modified incomplete LU factorization algorithm is:

for $i = 1$ **step** 1 **until** N **do**

begin

$L_{ii} = \alpha$;

for $j = 1$ **step** 1 **until** N **do**

begin

$$s_{ij} = A_{ij} - \sum_{t=1}^{\min\{i,j\}-1} L_{it}U_{tj};$$

if $((i,j) \notin Z)$ **then**

begin

if $(i > j)$ **then** $L_{ij} = s_{ij}$;

if $(i = j)$ **then** $L_{ii} = L_{ii} + s_{ii}$;

if $(i < j)$ **then** $\tilde{U}_{ij} = s_{ij}$;

end

else $L_{ii} = L_{ii} + s_{ij}$;

end

for $j = i + 1$ **step** 1 **until** n **do**

$U_{ij} = \tilde{U}_{ij}/L_{ii}$;

end

Since LU factorizations preserve a skyline sparsity structure, the MILU factorization is the same as the ILU factorization for $\alpha = 0$. The motivation for the MILU factorization is to control the elements of Q where it does not match A, at least in an average sense. In the homotopy context here with skyline A and $\alpha > 0$, Q can be construed as an approximation to A that is closer to (or more) positive definite than A.

4. Solving Linear Systems In HOMPACK

As discussed in Chapter 2 for the normal flow algorithm, a corrector step ΔZ is the unique minimum norm solution of (9), which uses the solution of the rectangular linear system $[D\rho_a]y = 0$. In this chapter we describe various algorithms, including the existing algorithm in HOMPACK, for the solution of such linear systems.

Let $(\bar{x}, \bar{\lambda})$ be a point on the zero curve γ, and $\bar{y}$ the unit tangent vector to γ at $(\bar{x}, \bar{\lambda})$ in the direction of increasing arc length s. Then the matrix

$$A = \begin{pmatrix} D_x\rho_a(x,\lambda) & D_\lambda\rho_a(x,\lambda) \\ c^t & d \end{pmatrix}, \tag{15}$$

where $(c^t \quad d)$ is any vector outside a set of measure zero (a hyperplane), is invertible at $(\bar{x}, \bar{\lambda})$ and in a neighborhood of $(\bar{x}, \bar{\lambda})$. Thus the kernel of $D\rho_a$ can be found by solving the linear system of equations

$$Ay = \alpha e_{n+1} = b, \tag{16}$$

where $(c^t \quad d)\bar{y} = \alpha$. As discussed in Chapter 3, we consider only two main approaches to determining the solutions of the linear system, i.e., the *direct* approach wherein the algorithm is applied to the A matrix and the *splitting* approach in which the matrix is first split into the sum of a symmetric matrix and a low rank correction. The coefficient matrix A in the linear system of equations (16) has a very special structure which can be exploited in several ways.

4.1. The direct approach.

Here depending on the choice of $(c^t \quad d)$, the matrix A is either symmetric or non-symmetric and accordingly suitable algorithms can be used to solve equations (16). Note that the leading $n \times n$ submatrix of A is $D_x\rho_a$, which is symmetric and sparse, but possibly

indefinite. Since symmetry is advantageous for some algorithms, A can be made symmetric and invertible by choosing $c = D_\lambda \rho_a$. If rank $D_x \rho_a = n - 1$, then $D_\lambda \rho_a$ is not a linear combination of the columns of $D_x \rho_a$, because rank $\left[D_x \rho_a \; D_\lambda \rho_a \right] = n$ by the homotopy theory. Thus $c^t = \left(D_\lambda \rho_a \right)^t$ is not a linear combination of the rows of the symmetric matrix $D_x \rho_a$, and the row rank $\begin{bmatrix} D_x \rho_a \\ \left(D_\lambda \rho_a \right)^t \end{bmatrix} = n$. Finally $\begin{pmatrix} D_\lambda \rho_a \\ * \end{pmatrix}$ is not a linear combination of the first n columns of A, so the column rank $A = n + 1$ for *any* choice of d. Now suppose that rank $D_x \rho_a = n$. Then rank $\begin{bmatrix} D_x \rho_a \\ \left(D_\lambda \rho_a \right)^t \end{bmatrix} = n$, and it suffices to choose d to make the last column of A independent from the first n columns. $D_\lambda \rho_a$ is a unique linear combination of the columns of $D_x \rho_a$, and any choice of d other than this combination of the components of $\left(D_\lambda \rho_a \right)^t$ will make the $(n+1)$st column independent. Let $\bar{A}$ denote A at $(\bar{x}, \bar{\lambda})$. Since $\dim[\ker(\bar{A})] \leq 1$, $\bar{A}y = 0$ implies $y = \alpha \bar{y}$, and thus with $\bar{y}^t = \left(\hat{y}^t, \bar{y}_{n+1} \right)$, $\left(D_\lambda \rho_a(\bar{x}, \bar{\lambda}) \right)^t \hat{y} + d\, \bar{y}_{n+1} = 0$. Choosing any $\beta \neq 0$ and solving $\left(D_\lambda \rho_a(\bar{x}, \bar{\lambda}) \right)^t \hat{y} + d\, \bar{y}_{n+1} = \beta$ for d ($\bar{y}_{n+1} \neq 0$ since rank $D_x \rho_a(\bar{x}, \bar{\lambda}) = n$) gives a d such that rank$(A) = n + 1$ for (x, λ) near $(\bar{x}, \bar{\lambda})$.

Observe also that if $D_x \rho_a$ is positive definite, choosing $d > 0$ sufficiently large guarantees that

$$A = \begin{pmatrix} D_x \rho_a & D_\lambda \rho_a \\ \left(D_\lambda \rho_a \right)^t & d \end{pmatrix}$$

is also positive definite. Proof: Since A is symmetric, by Sylvester's Theorem A is positive definite if and only if all its leading principal minors are positive. Since $D_x \rho_a$ is positive definite, the first n leading principal minors are positive, and it suffices to show $\det A > 0$. Expanding $\det A$ along the last column,

$$\det A = d \cdot \det D_x \rho_a + \text{terms not involving } d > 0$$

for $d > 0$ sufficiently large. □

Any other choice of c besides $D_\lambda \rho_a$ makes the matrix A nonsymmetric and suitable

algorithms for nonsymmetric coefficient matrices can be used.

4.2. The splitting approach.

Another approach is to attack (16) indirectly as follows. Write

$$A = M + L, \tag{19}$$

where

$$M = \begin{pmatrix} D_x\rho_a(\bar{x},\bar{\lambda}) & c \\ c^t & d \end{pmatrix}, \tag{20}$$

$$L = ue_{n+1}^t, \qquad u = \begin{pmatrix} D_\lambda\rho_a(\bar{x},\bar{\lambda}) - c \\ 0 \end{pmatrix}.$$

Observe that for almost all choices of $(c^t \quad d)$ the symmetric part M is also invertible. Then using the Sherman-Morrison formula, the solution y to the original system $Ay = b$ can be obtained from

$$y = \left[I - \frac{M^{-1}ue_{n+1}^t}{(M^{-1}u)^t e_{n+1} + 1} \right] M^{-1}b, \tag{21}$$

which requires the solution of two linear systems $Mz = u$ and $Mz = b$ with the sparse, symmetric, invertible matrix M. The scheme (19-21) was proposed in [28], and further investigated by Chan and Saad [9]. The HOMPACK algorithm, which uses this splitting approach to the solution of these linear systems, will be discussed next.

Let $|\bar{y}_k| = \max_i |\bar{y}_i|$ define the index k. In HOMPACK, $(c^t \quad d) = e_k^t$, where e_k is a vector with 1 in the kth component and zeros elsewhere. Hence (15) becomes

$$A = \begin{pmatrix} D\rho_a(x,\lambda) \\ e_k^t \end{pmatrix}.$$

The kernel of $D\rho_a$ can be found by solving the linear system of equations

$$Ay = \bar{y}_k e_{n+1} = b.$$

Again, splitting the coefficient matrix as

$$A = M + L$$

gives a symmetric

$$M = \begin{pmatrix} D_x \rho_a(\bar{x}, \bar{\lambda}) & * \\ e_k^t & \end{pmatrix},$$

$$L = u e_{n+1}^t, \qquad u = \begin{pmatrix} D_\lambda \rho_a(\bar{x}, \bar{\lambda}) \\ 0 \end{pmatrix} - e_k(1 - \delta_{k,n+1}).$$

(The Kronecker $\delta_{k,n+1}$ takes care of the special case $k = n+1$.) Then the Sherman-Morrison formula (21) is used for the solution y of the linear system (16). Craig's preconditioned algorithm is used for solving the systems $Mz = u$ and $Mz = b$. Craig's preconditioned algorithm is:

choose y_0, Q;

set $r_0 = b - Ay_0$;

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

set $p_0 = A^t Q^{-t} \tilde{r_0}$;

for $i = 0$ **step** 1 **until** convergence **do**

begin

$$a_i = \frac{(\tilde{r_i}, \tilde{r_i})}{(p_i, p_i)};$$

$$y_{i+1} = y_i + a_i p_i;$$

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

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

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

end

For this algorithm, a minimum of $5(n+1)$ storage locations is required (in addition to that for A). The vectors y, $\tilde{r}$, and p all require their own locations; $Q^{-t}\tilde{r}$ can share with Ap; $Q^{-1}Ap$ can share with $A^tQ^{-t}\tilde{r}$. The computational cost per iteration of this algorithm is:

(a) two preconditioning solves ($Q^{-1}v$ and $Q^{-t}v$);

(b) two matrix-vector products (Av and A^tv);

(c) $5(n+1)$ multiplications (the inner products (p,p) and $(\tilde{r},\tilde{r})$, ap, bp, and $aQ^{-1}Ap$.

The preconditioning matrix Q is taken as a positive definite approximation of M (the *Gill-Murray preconditioner*, described in detail in [21] and [54]).

If M had only one or two negative eigenvalues, then after several rank one updates making M positive definite, a direct Cholesky factorization could be obtained, and then the solution to (16) recovered after several more applications of (21). This direct algorithm for solving (16) would be effective for such M, but since only one or two negative eigenvalues for M cannot be assumed in general (the M for a large shallow dome problem has maany negative eigenvalues along the unloading portions of the equilibrium curve), a direct rank one update/Cholesky scheme would not be suitable for HOMPACK.

There are several other schemes which could be used instead of the one in HOMPACK for finding the kernel of $D\rho_a$, as we have discussed before. A brief summary of the schemes is as follows:

(i) using different last rows for the augmented coefficient matrix A of (15), i.e., other vectors $(c^t \quad d)$ instead of e_k^t;

(ii) using other preconditioners on M;

(iii) using other algorithms for the solution of the linear systems $Mz = u$ and $Mz = b$, e.g., Orthomin(k), SSOR, etc., instead of Craig's algorithm;

(iv) doing (i), (ii), or (iii) on the nonsymmetric A directly instead of on the symmetric M in the splitting $A = M + L$.

Combining the preconditioning techniques with the algorithms for solving linear systems with different last rows for A produces a large number of possible methods. Chapter 6 focuses on a subset of these possible methods and compares their efficiency.

5. Description Of The Test Problems

In this chapter, the test problems are described in detail, beginning with the shallow arch structural response problem.

5.1. Shallow arch problem.

The equations of equilibrium of the arch are obtained from the principle of the stationary value of the total potential energy, according to which, of all the kinematically admissible displacement fields, the one that makes the total potential energy of a structure stationary also satisfies its equations of equilibrium. The total potential energy π of a structure is given by the sum of its strain energy and the potential of external loads.

The shallow arch of Figure 1 is discretized by an assemblage of straight p-q frame elements such as those described in [27]. A frame element is a structural component that is initially straight and undergoes axial, bending, and torsional deformation resulting from finite displacements and rotations of its ends (nodes) p and q. The displacements of the end q relative to the end p are

$$\begin{pmatrix} \delta u \\ \delta v \\ \delta w \end{pmatrix} = [T]_p \begin{pmatrix} X_q - X_p \\ Y_q - Y_p \\ Z_q - Z_p \end{pmatrix} - \begin{pmatrix} L \\ 0 \\ 0 \end{pmatrix} + [T]_p \begin{pmatrix} U_q - U_p \\ V_q - V_p \\ W_q - W_p \end{pmatrix},$$

where L is the initial rigid body length, and U_i, V_i, W_i ($i = p$ or q) denote the global displacements of the nodes. The matrix $[T]_p$ can be shown to be [27] $[T]_p = [T_1(\phi_x, \phi_y, \phi_z)]$ $[T_1(\theta_{xp}, \theta_{yp}, \theta_{zp})]$ with

$$[T_1(\alpha_x, \alpha_y, \alpha_z)] = \begin{pmatrix} c_y c_z & c_y s_z & -s_y \\ -c_x s_z + s_x s_y c_z & c_x c_z + s_x s_y s_z & s_x c_y \\ s_x s_z + c_x s_y c_z & -s_x c_z + c_x s_y s_z & c_x c_y \end{pmatrix},$$

$c_i = \cos\alpha_i$ and $s_i = \sin\alpha_i$ for $i = x$, y, and z. Angles ϕ_x, ϕ_y, and ϕ_z are the initial orientation angles and angles θ_{xp}, θ_{yp}, and θ_{zp} are the rigid body rotations of the end p.

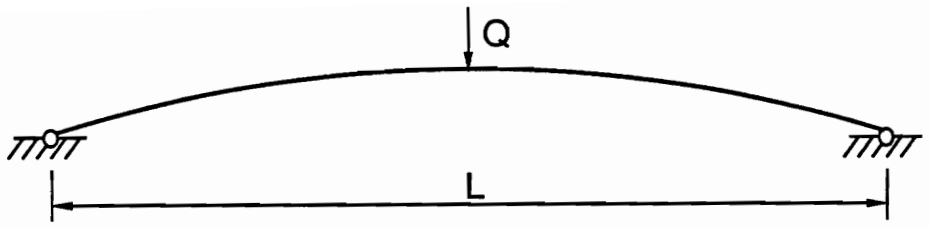

FIGURE 1. *Shallow arch.*

In the equation for $[T]_p$, Euler angle transformations are implied with the order of the rotations being α_z, α_y, and α_x.

Similarly, with the restriction of small relative rotations within the element, the rotations ψ_x, ψ_y, ψ_z of the end q relative to the end p are

$$\begin{pmatrix} \psi_x \\ \psi_y \\ \psi_z \end{pmatrix} = [T]_p \begin{pmatrix} \theta_{xq} - \theta_{xp} \\ \theta_{yq} - \theta_{yp} \\ \theta_{zq} - \theta_{zp} \end{pmatrix}.$$

With the relative generalized displacements $(\delta u, \delta v, \delta w)$ and (ψ_x, ψ_y, ψ_z) known, the usual deformation patterns of the reference axis of the beam element in the corotational coordinate system are assumed to be

$$u(\xi) = \xi \frac{\delta u}{L}, \qquad v(\xi) = \frac{1}{L}(3\xi^2 - 2\xi^3)(\delta v - z_s\psi_x) + (\xi^3 - \xi^2)\psi_z,$$
$$\beta = \xi\psi_x, \qquad w(\xi) = \frac{1}{L}(3\xi^2 - 2\xi^3)(\delta w + y_s\psi_x) - (\xi^3 - \xi^2)\psi_y,$$

where $\xi = x/L$ and y_s and z_s are the coordinates of the shear center of the cross section of the beam. The strain at any point (y, z) on the cross-section of the frame element can be shown to be

$$\epsilon = \frac{\delta u}{L} - \eta\left[\frac{6}{L}(1 - 2\xi)(\delta v - z_s\psi_x) + 2(3\xi - 1)\psi_z\right]$$
$$- \zeta\left[\frac{6}{L}(1 - 2\xi)(\delta w + y_s\psi_x) - 2(3\xi - 1)\psi_y\right],$$

with $\eta = y/L$ and $\zeta = z/L$. In these equations it is implicitly assumed that the lateral displacements and twists are referenced to a longitudinal axis through the shear center, while the axial displacements and rotations are referenced to the centroidal axis.

The total potential energy of such a discretized model of the arch can be expressed as

$$\pi = \sum_{e=1}^{m} U^e - q^t Q,$$

where U^e is the strain energy of the eth element, $e = 1, \ldots, m$, q is the vector of nodal displacement degrees of freedom of the entire model and Q is the vector of externally applied loads. The strain energy U^e of the eth frame element is given by

$$U^e = \frac{E}{2} \int_V \epsilon^2 dv = \frac{E}{2} \int_0^{L_e} \int_{A_e} \epsilon^2 dA\, dx,$$

where ϵ is the strain of a point (x, y, z) of the beam, which was derived above. Substituting for ϵ and doing the integration gives

$$U^e = U_{\mathrm{p-q}} = \frac{E}{2L_e} \Big\{ A_e(\delta u)^2 + \frac{12}{L_e^2} I_z \left[(\delta v)^2 + \frac{1}{3} L_e^2 \psi_z^2 - L_e \delta v\, \psi_z \right] + \frac{12}{L_e^2} I_y \left[(\delta w)^2 + \frac{1}{3} L_e^2 \psi_y^2 + L_e \delta w\, \psi_y \right] \Big\},$$

where A_e is the cross-sectional area, and I_y and I_z are the cross-sectional moments of inertia about the y and z axes respectively. It is evident that the potential energy π of the model is a highly nonlinear function of the nodal displacements. The equations of equilibrium of the model are obtained by setting the variation $\delta\pi$ to zero, or equivalently by

$$\nabla \pi = 0.$$

Closed form analytical expressions for $\nabla\pi$ can be obtained with some difficulty, but obtaining the Jacobian matrix of $\nabla\pi$ analytically seems out of the question. Hence the Jacobian matrix of the equilibrium equations is obtained by finite difference approximations.

By symmetry only half the arch need be modelled, and the results here are for the arch parameters used in [29], with a full arch load of 3000 lbs. This is just below the limit point. To go through the limit point and along the unloading portion of the equilibrium curve apparently requires very accurate Jacobian matrices and numerical linear algebra,

and none of these iterative linear equation solvers used in HOMPACK were able to go past the limit point without tweaking the HOMPACK stepsize control parameters.

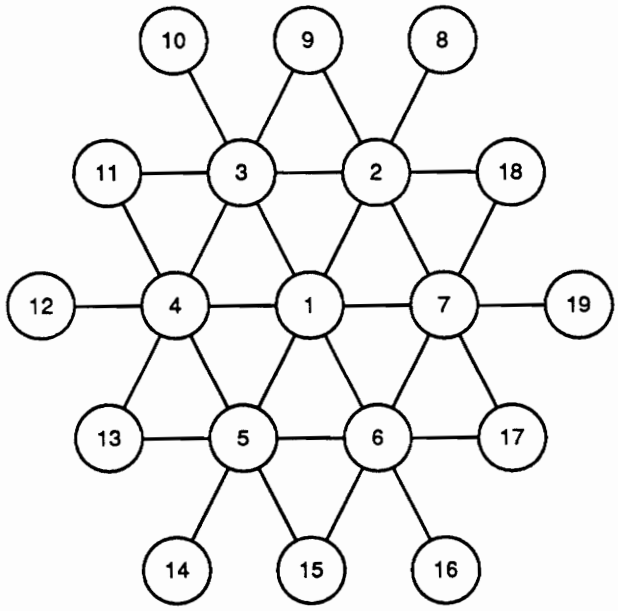

FIGURE 2. *Triangulation for 21 degree of freedom shallow dome.*

5.2. Shallow dome problem.

The shallow dome of Figure 2 is built up from space truss elements with three global displacement degrees of freedom (u_1, u_2, u_3) at each of the two nodes. For an element of original length L between its two nodes p and q, the change in length δL is given by

$$\delta L = \left[\sum_{i=1}^{3}(x_{qi} + u_{qi} - x_{pi} - u_{pi})^2\right]^{1/2} - \left[\sum_{i=1}^{3}(x_{qi} - x_{pi})^2\right]^{1/2},$$

where x_{ij}, u_{ij}, $i = p, q$; $j = 1, 2, 3$ are the global coordinates and displacements of the two nodes. This can be simplified to

$$\delta L = L\left[1 + \sum_{i=1}^{3}\left(\frac{2(\Delta x_i \Delta u_i)}{L^2} + \frac{(\Delta u_i)^2}{L^2}\right)\right]^{1/2} - L,$$

where Δ is the difference operator for the q and p values. Accordingly, the axial strain in the e-th element is

$$\epsilon^e = \frac{\delta L}{L} = \left[1 + \sum_{i=1}^{3} \left(\frac{2(\Delta x_i \Delta u_i)}{L^2} + \frac{(\Delta u_i)^2}{L^2} \right) \right]^{1/2} - 1.$$

The strain energy of the e-th element in a purely linearly elastic response is given by

$$U_s^e = \frac{E}{2} \int_V (\epsilon^e)^2 dV = \frac{EA^e L^e}{2} (\epsilon^e)^2,$$

where E and A are the Young's modulus and cross-sectional area, respectively, of the e-th element.

The total potential energy of the dome is then given by

$$\pi = \sum_{e=1}^{m} U_s^e - U^T Q,$$

where U_i, $i = 1, \ldots, 6$ are the six components u_{qk}, u_{pk}, $k = 1, 2, 3$, and Q is the generalized force vector. The equations of equilibrium of the model are then obtained by setting

$$\nabla \pi = \sum_{e=1}^{m} EA^e L^e \nabla \epsilon^e - Q = 0.$$

Both the gradient of π as well as its Hessian can be evaluated explicitly without resorting to finite differencing operations as in the case of the frame element used to model the shallow arch.

The effect of modelling the shallow dome with truss elements in concentric rings is that changing the number of truss elements changes the model and its behavior. Thus the dome problems with different degrees of freedom reported in the tables are qualitatively different, with different buckling loads and bifurcation points. The results reported here are for shallow domes with base radius 720 and sphere radius 3060, and a point load at the very top.

5.3. Artificial turning point problem.

The turning point problem is derived from the system of equations

$$F(\mathbf{x}) = \big(F_1(\mathbf{x}), F_2(\mathbf{x}), \ldots, F_N(\mathbf{x})\big)^t = 0$$

where

$$F_i(\mathbf{x}) = \tan^{-1}\big(\sin[x_i(i \bmod 100)]\big) - \frac{(x_{i-1} + x_i + x_{i+1})}{20}, \qquad i = 1, \ldots, N,$$

and $x_0 = x_{N+1} = 0$. The zero curve γ tracked from $\lambda = 0$ to $\lambda = 1$ corresponds to $\rho_a(x, \lambda) = (1 - .8\lambda)(x - a) + .8\lambda F(x)$, where a was chosen artificially to produce turning points in γ. HOMPACK had no difficulty going through numerous turning points using iterative linear equation solvers.

6. Numerical Experiments

In this chapter we present the numerical results of the experiments that were done with the serial algorithms and draw conclusions from them. Of the various algorithmic possibilities mentioned in Chapter 4, those considered further are given short names in the list below. Some possibilities do not make sense or are impractical in the homotopy context, and thus are not considered. Of the almost all mathematically valid choices for the last row $(c^t \quad d)$ of A, only e_k^t (the easiest to implement), $c = D_\lambda \rho_a(\bar{x}, \bar{\lambda})$ (the easiest symmetrization of A), and the tangent vector $\bar{y}^t$ at the previous point on the zero curve (the optimal choice for conditioning, since it is orthogonal to the top n rows of A at $(\bar{x}, \bar{\lambda})$) are used.

6.1. Algorithm acronyms.

SC – $A = M + L$ splitting, Craig's method with M, no preconditioning;

SCGM – $A = M + L$ splitting, Craig's method with M, Gill-Murray preconditioning from HOMPACK;

SCILU – $A = M + L$ splitting, Craig's method with M, incomplete LU preconditioning;

SCMILU – $A = M + L$ splitting, Craig's method with M, modified incomplete LU preconditioning;

C – no splitting, Craig's method with A, no preconditioning;

CGM – no splitting, Craig's method with A, Gill-Murray preconditioning from HOMPACK;

CILU – no splitting, Craig's method with A, incomplete LU preconditioning;

CMILU – no splitting, Craig's method with A, modified incomplete LU preconditioning.

SR – $A = M + L$ splitting, GMRES(2) with M, no preconditioning;

SRGM – $A = M + L$ splitting, GMRES(2) with M, Gill-Murray preconditioning from HOMPACK;

SRILU – $A = M + L$ splitting, GMRES(2) with M, incomplete LU preconditioning;

SRMILU – $A = M + L$ splitting, GMRES(2) with M, modified incomplete LU preconditioning;

R – no splitting, GMRES(2) with A, no preconditioning;

RGM – no splitting, GMRES(2) with A, Gill-Murray preconditioning from HOMPACK;

RILU – no splitting, GMRES(2) with A, incomplete LU preconditioning;

RMILU – no splitting, GMRES(2) with A, modified incomplete LU preconditioning.

6.2. Tables.

Tables 1–6 show some timing results for these three test problems. An asterisk indicates either that the iterative linear equation solver stalled, γ was lost because of inaccurate tangents from the linear equation solver, or the time was at least an order of magnitude larger than anything else in the table. The times are for tracking the entire zero curve γ and thus represent the solution of many linear systems of varying degrees of difficulty. The experiments were done in double precision using a single processor of a Sequent Symmetry S81 multiprocessor. The major headings are the acronyms for the algorithms, and the subheadings denote the choice $(c^t \quad d)$ for the last row of A. The MILU algorithms used $\alpha = 1$. There is asymmetry in the tables because some possibilities do not make sense. For instance, there is no CGM with e_k because the Gill-Murray preconditioner requires a symmetric matrix, and there are no S* with $D_\lambda \rho_a$ since the choice $c^t = \left(D_\lambda \rho_a\right)^t$ makes A symmetric so there is no need to split off a symmetric matrix M from A.

TABLE 1
Execution time in seconds for shallow arch problem.

	SC		SCGM		SCILU		SCMILU	
n	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$
29	1108	947	468	818	599	470	908	975
47	16904	17593	7322	10105	5674	5957	11538	12390
	SR		SRGM		SRILU		SRMILU	
29	*	*	461	*	559	443	*	*
47	*	*	5314	*	5796	6332	*	*

TABLE 2
Execution time in seconds for shallow arch problem.

	C			CILU			CMILU			CGM
n	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	$D_\lambda \rho_a$
29	856	884	919	533	458	443	841	845	900	464
47	14205	13591	14606	5794	5807	6776	9943	10968	10135	6921
	R			RILU			RMILU			RGM
29	*	*	*	443	431	506	*	*	*	429
47	*	*	*	5355	5304	5697	*	*	*	5260

TABLE 3
Execution time in seconds for shallow dome problem.

	SC		SCGM		SCILU		SCMILU	
n	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$
21	57	86	108	57	21	25	92	141
546	3127	4803	2710	1787	492	630	4892	6687
1050	5615	8553	5107	3177	887	1133	8259	11672
	SR		SRGM		SRILU		SRMILU	
21	*	*	*	*	14	15	*	*
546	*	*	*	*	299	335	*	*
1050	*	*	*	*	559	625	*	*

TABLE 4
Execution time in seconds for shallow dome problem.

	C			CILU			CMILU			CGM
n	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	$D_\lambda \rho_a$
21	46	47	47	16	16	16	68	79	69	89
546	2495	2545	2573	355	369	365	3037	3585	3094	2233
1050	4504	4691	4690	632	665	651	5536	6327	5570	4313
	R			RILU			RMILU			RGM
21	*	*	*	11	12	11	*	*	*	*
546	*	*	*	230	241	232	*	*	*	*
1050	*	*	*	425	446	430	*	*	*	*

TABLE 5

Execution time in seconds for turning point problem.

	SC		SCGM		SCILU		SCMILU	
n	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$
20	28	36	12	13	6	6	39	47
60	266	356	41	50	20	22	163	213
125	1635	2310	127	170	54	65	568	795
250	3026	3767	228	267	95	109	1032	1335
500	6279	7783	448	501	189	207	2130	2656
1000	14150	17768	1077	1174	434	490	4874	6052
	SR		SRGM		SRILU		SRMILU	
20	1301	*	13	18	4	4	120	*
60	*	*	47	*	13	14	*	*
125	*	*	*	*	36	40	*	*
250	*	*	*	*	60	69	*	*
500	*	*	*	*	119	131	*	*
1000	*	*	*	*	274	296	*	*

TABLE 6

Execution time in seconds for turning point problem.

	C			CILU			CMILU			CGM
n	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	$D_\lambda \rho_a$
20	17	19	21	4	7	4	24	26	27	5
60	167	176	186	13	22	13	109	112	118	22
125	1117	1132	1384	38	64	42	412	421	446	85
250	2296	1925	3873	66	110	74	765	699	726	134
500	4741	3899	8352	129	210	148	1573	1381	1465	260
1000	11577	9335	20375	323	493	353	3605	3031	3308	617
	R			RILU			RMILU			RGM
20	*	*	1263	3	3	3	43	70	75	6
60	*	*	*	9	10	9	*	*	2016	*
125	*	*	*	26	31	28	*	*	*	*
250	*	*	*	45	51	46	*	*	*	*
500	*	*	*	88	98	88	*	*	*	*
1000	*	*	*	199	225	202	*	*	*	*

TABLE 7

Average, maximum, and minimum number of conjugate gradient iterations per linear system along homotopy curve for shallow arch problem.

	SC		SCGM		SCILU		SCMILU	
n	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$
29	66,109,1	66,101,1	4,10,1	28,40,1	2,3,1	3,3,1	39,63,1	39,58,1
47	190,313,1	194,291,1	5,10,1	37,53,1	2,3,1	3,3,1	64,103,1	65,97,1
	SR		SRGM		SRILU		SRMILU	
29	*	*	4,92,1	*	1,2,1	1,2,1	*	*
47	*	*	2,140,1	*	1,2,1	1,2,1	*	*

TABLE 8

Average, maximum, and minimum number of conjugate gradient iterations per linear system along homotopy curve for shallow arch problem.

	C			CILU		
n	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$
29	99,127,51	91,107,38	98,120,52	3,3,2	4,5,2	3,3,2
47	265,360,109	239,305,133	265,355,105	3,3,2	4,4,2	3,3,2
	CMILU			CGM		
29	56,68,30	56,65,35	55,65,30	—	—	6,7,2
47	87,119,48	91,102,53	87,131,48	—	—	6,7,2
	R			RILU		
29	*	*	*	1,1,1	1,2,1	1,1,1
47	*	*	*	1,1,1	1,2,1	1,1,1
	RMILU			RGM		
29	*	*	*	—	—	2,2,1
47	*	*	*	—	—	2,2,1

TABLE 9

Average, maximum, and minimum number of conjugate gradient iterations per linear system along homotopy curve for shallow dome problem.

	SC		SCGM		SCILU		SCMILU	
n	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$
21	17,31,1	24,36,1	16,115,1	7,46,1	2,3,1	2,3,1	14,30,1	21,32,1
546	38,75,1	54,87,1	15,113,1	9,63,1	2,3,1	3,2,1	24,45,1	37,71,1
1050	38,76,1	53,91,1	16,114,1	8,101,1	2,3,1	3,3,1	24,47,1	36,61,1
	SR		SRGM		SRILU		SRMILU	
21	*	*	*	*	1,2,1	1,2,1	*	*
546	*	*	*	*	1,2,1	1,2,1	*	*
1050	*	*	*	*	1,2,1	1,2,1	*	*

TABLE 10

Average, maximum, and minimum number of conjugate gradient iterations per linear system along homotopy curve for shallow dome problem.

	C			CILU		
n	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$
21	26,36,14	26,36,14	26,36,14	2,3,2	2,3,2	2,3,2
546	58,81,17	57,82,17	58,82,18	2,3,2	2,3,2	2,3,2
1050	58,87,18	59,91,18	58,83,18	2,3,2	2,3,2	2,3,2
	CMILU			CGM		
21	19,25,8	22,28,8	19,24,8	—	—	23,113,2
546	34,47,12	38,52,12	34,45,11	—	—	22,111,2
1050	34,49,12	38,50,12	34,49,13	—	—	23,113,2
	R			RILU		
21	*	*	*	1,1,1	1,1,1	1,1,1
546	*	*	*	1,1,1	1,1,1	1,1,1
1050	*	*	*	1,1,1	1,1,1	1,1,1
	RMILU			RGM		
21	*	*	*	—	—	*
546	*	*	*	—	—	*
1050	*	*	*	—	—	*

TABLE 11

Average, maximum, and minimum number of conjugate gradient iterations per linear system along homotopy curve for turning point problem.

	SC		SCGM		SCILU		SCMILU	
n	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$	e_k^t	$\bar{y}^t$
20	21,28,1	24,29,1	4,6,1	5,7,1	2,2,1	2,2,1	17,27,1	19,26,1
60	60,100,1	69,87,1	4,8,1	5,8,1	2,3,1	2,3,1	21,37,1	25,39,1
125	127,261,1	154,264,1	5,9,1	6,11,1	2,3,1	2,3,1	26,51,1	31,51,1
250	139,302,1	150,246,1	5,11,1	5,10,1	2,2,1	2,3,1	27,60,1	30,55,1
500	149,314,1	164,281,1	5,11,1	5,10,1	2,2,1	2,3,1	28,62,1	31,53,1
1000	151,312,1	162,289,1	5,11,1	5,11,1	2,2,1	2,3,1	28,64,1	31,56,1
	SC		SCGM		SCILU		SCMILU	
20	732,4446,1	*	6,58,1	8,75,1	1,1,1	1,1,1	49,315,1	*
60	*	*	6,54,1	*	1,1,1	1,1,1	*	*
125	*	*	*	*	1,1,1	1,1,1	*	*
250	*	*	*	*	1,1,1	1,1,1	*	*
500	*	*	*	*	1,1,1	1,1,1	*	*
1000	*	*	*	*	1,1,1	1,1,1	*	*

TABLE 12

Average, maximum, and minimum number of conjugate gradient iterations per linear system along homotopy curve for turning point problem.

	C			CILU		
n	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$	e_k^t	$\bar{y}^t$	$D_\lambda \rho_a$
20	24,29,1	24,28,1	26,31,1	2,2,1	4,5,1	2,3,1
60	70,86,1	69,84,1	74,91,2	2,3,1	4,7,1	2,3,2
125	159,292,1	151,232,1	179,328,3	2,3,1	4,5,1	2,3,2
250	196,404,1	150,246,1	231,407,3	2,3,1	4,5,1	2,3,2
500	216,427,1	165,337,1	268,489,3	2,3,1	4,6,1	2,3,2
1000	224,446,1	164,323,1	285,536,3	2,3,1	4,5,1	3,3,2
	CMILU			CGM		
20	20,23,1	20,23,1	20,26,3	—	—	3,9,2
60	26,36,1	25,34,1	26,36,2	—	—	3,12,1
125	33,49,1	31,48,1	33,53,3	—	—	5,15,2
250	36,61,1	30,45,1	32,48,1	—	—	4,15,2
500	38,74,1	31,53,1	33,53,3	—	—	5,16,2
1000	39,75,1	31,50,1	33,54,3	—	—	5,16,2
	R			RILU		
20	*	*	1905,21000,2	1,1,1	1,2,1	1,1,1
60	*	*	*	1,1,1	1,2,1	1,1,1
125	*	*	*	1,1,1	1,2,1	1,1,1
250	*	*	*	1,1,1	1,2,1	1,1,1
500	*	*	*	1,1,1	1,2,1	1,1,1
1000	*	*	*	1,1,1	1,2,1	1,1,1
	RMILU			RGM		
20	47,110,2	61,200,2	79,226,2	—	—	4,92,1
60	*	*	568,12486,2	—	—	*
125	*	*	*	—	—	*
250	*	*	*	—	—	*
500	*	*	*	—	—	*
1000	*	*	*	—	—	*

6.3. Discussion of numerical results.

The convergence rate of conjugate gradient iterative methods for linear systems depends on the spectrum and the condition number of the coefficient matrix, and therefore one would predict $\bar{y}^t$ should be a better choice for the last row of A than e_k^t. Since $\bar{y}$ is orthogonal to the rows of $D\rho_a(\bar{x}, \bar{\lambda})$, a good approximation to the first n rows $D\rho_a(x, \lambda)$ of A, one expects A with $\bar{y}$ to be better conditioned than with e_k. Tables 1, 3, and 5 show that apparently this better conditioning does not compensate for the extra work involved in using $\bar{y}$. Although $\bar{y}$ is sometimes better than e_k, there seems to be no strong evidence that $\bar{y}$ is worth the trouble.

Figure 3 shows the condition numbers of A and $Q^{-1}A$ along γ for the shallow arch problem with $n = 29$ (CGM). The shallow arch problem is indeed a hard problem, but Figure 3 alone would not suggest that–see the discussion of the shallow arch problem's spectra later. The Jacobian matrix $D_x\rho_a$ becomes indefinite near $\lambda = .88$, at which point the Gill-Murray preconditioner ceases being nearly perfect. This figure is typical for the Gill-Murray preconditioner. (During the course of the experiments it was observed that points far from γ frequently generate much worse conditioned problems. This has important implications for curve tracking strategy, because large steps along γ will be offset by expensive numerical linear algebra to return to γ.)

Tables 2, 4, 6 show that there is no clear winner between e_k, $\bar{y}$, and $D_\lambda\rho_a$, and further there is little correlation between the algorithm and the best choice for c. One is tempted to pick CGM with $c^t = \left(D_\lambda\rho_a\right)^t$ over SCGM with $c^t = e_k^t$, based on Tables 5, 6 and the fact that CGM only solves one linear system per tangent vector computation, as opposed to two linear systems with M for the splitting algorithms. However, from Tables 1 and 2, SCGM with e_k is substantially better than CGM with $D_\lambda\rho_a$. This demonstrates that only counting the number of linear system solves can be dangerously misleading. The results do indicate that for a given choice of c^t, when no preconditioning is used or when MILU preconditioning is used, it is slightly more efficient to use the no-splitting strategy than the splitting. However, with ILU preconditioning the differences between corresponding splitting and no-splitting cases are not at all significant.

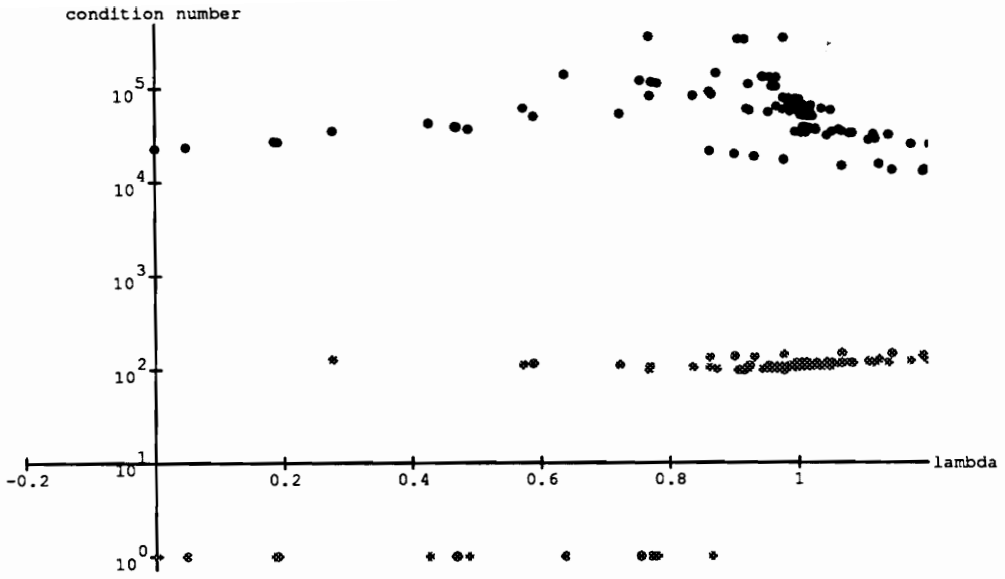

FIGURE 3. *Condition numbers for A (black dots) and $Q^{-1}A$ (grey dots) against λ along γ; shallow arch, $n = 29$, CGM.*

Tables 1–6 seem to strongly support an argument for CILU as the best Craig method, even though the ILU factorization fails to exist at turning points, and is unstable whenever A is indefinite. What is not indicated in the tables, though, are all the homotopy curve tracking runs which failed because the ILU preconditioner failed to exist or generated an overflow, or the difficulty caused HOMPACK by inaccurate tangents resulting from the ILU. Because of this potential catastrophic failure or instability, the ILU preconditioner would never be seriously considered for robust homotopy algorithms. Still, the tables do show why numerical analysts' paranoia about unstable algorithms is not shared by engineers.

The algorithms SSOR and Orthomin(k), discussed earlier, are not shown in the tables because they totally fail at turning points and along unloading portions of equilibrium curves (for reasons stated in §3). When these methods do work, they can be very efficient (e.g., Orthomin(1) on A with $c^t = \left(D_\lambda \rho_a\right)^t$ took 443 (6092) seconds for the shallow arch problem with $n = 29$ (47)), but that is no consolation for homotopy curve tracking.

GMRES(k) has a solid theoretical justification [44], and has been used very successfully in a variety of contexts [4], [44], [47], [48]. Nevertheless, GMRES(k) with $k < n$

performed unacceptably on the test problems here without preconditioning. For the shallow arch problem with $n = 29$ and tol $= 10^{-12}$, GMRES(29) on A with $c^t = \left(D_\lambda \rho_a\right)^t$ took 591 seconds, comparable to CGM and SCGM. For $k = 1, 3, 25$, GMRES(k) took over a day of CPU time. Relaxing the tolerance to 10^{-6}, GMRES(25) took 18,330 seconds. This is especially noteworthy because the A matrices are symmetric and positive definite up to $\lambda = .88$, and mildly indefinite from there to $\lambda = 1$. For the turning point problem with $n = 20$, tol $= 10^{-12}$, $c^t = \left(D_\lambda \rho_a\right)^t$, the performance degradation from the full GMRES to GMRES(k) was dramatic. With $k = 20, 19, 18, 15, 10, 8$, GMRES($k$) took, respectively, 19, 117, 154, 375, 338, 420 seconds. Thus for these problems, without preconditioning, only the full GMRES method is competitive.

The tables also show results for GMRES(2), implemented with all the same preconditioners and choices of $(c^t \quad d)$ as was Craig's method. $k = 2$ was chosen because a preconditioned GMRES(2) requires exactly the same amount of storage as the preconditioned Craig's method. Numerous other runs were made with $k = 1, 3, 5$, or 10, but there was no substantial difference from $k = 2$ on the larger problems. In virtually all cases the asterisks in the tables correspond to a stalled residual norm somewhere along γ. It was noted, though, that many of the linear systems along γ were solved efficiently by GMRES(2). Perhaps the most disappointing failure was that of RGM on the shallow dome problem even for $n = 21$, because the Gill-Murray preconditioner was fairly good there. It is evident from the tables that GMRES(2), without nearly perfect preconditioning (ILU), is unsuitable for use in a general, robust homotopy curve tracking code like HOMPACK.

There are some theoretical results concerning the convergence of GMRES(k) given by Saad and Schultz [44]. These results give worst-case bounds on the rate of residual norm reduction which are determined by the distribution of eigenvalues of A. For the shallow arch and turning point problems, the eigenvalues of A were determined numerically along the homotopy curve, and the resulting bounds were often (although not in every case) found to guarantee only hopelessly slow residual norm reduction, indeed often to guarantee no residual norm reduction at all even when $k = n$.

Tables 7–12 show the average, maximum, and minimum number of conjugate gradient iterations per linear system solution along the homotopy zero curve γ for the same algorithms as Tables 1–6. Such iteration statistics give an intuitive feel for how the algorithms behave, and are sometimes very revealing. Tables 7 and 8 show that symmetry does improve the algorithms' efficiency, and that all other things being equal, achieving symmetric coefficient matrices is worthwhile. (The S* algorithms based on symmetry are not uniformly better, because all other things are not equal.) Note that in all cases for Craig's method the maximum number of conjugate gradient iterations is less than or equal to eight times the average, which says that the convergence behavior is fairly consistent. On the other hand the range between the minimum and maximum (for the C* algorithms) is as great as 3 to 536 (C for $n = 1000$ in Table 12), showing that there is a wide variation in the difficulty of the linear systems encountered along γ. The convergence behavior of GMRES(2) is not as consistent as for Craig's method, with the maximum being as much as 70 times the average (SRGM for $n = 47$ in Table 7).

The Gill-Murray preconditioner is clearly excellent, as shown by the average number of iterations in Tables 7–12 and Figure 3. It is more robust than the ILU and MILU preconditioners in the presence of turning points and indefinite $D_x\rho_a$. However, the shallow dome problem (Tables 3, 4, 9, and 10) shows that the Gill-Murray preconditioner may do a very poor job indeed on strongly indefinite matrices (which occur on the unloading parts of the shallow dome equilibrium curve). While reducing the average number of iterations, the Gill-Murray preconditioner actually increases the maximum number of iterations compared to the unpreconditioned algorithm.

It would be possible to test separately each aspect of the iterative linear system solving algorithms, such as convergence rate, sensitivity to starting point, cost of preconditioning, storage cost, computational complexity per iteration, etc. What ultimately matters, however, is the combined performance of the total algorithm on a wide range of typical realistic problems. Measuring the performance along homotopy zero curves for nontrivial problems is an attempt to measure the overall performance *in situ*.

A succinct, albeit oversimplified summary of the discussion is that ILU preconditioning is the most efficient but it may completely fail for some cases, while the Gill-Murray preconditioner rarely fails but is somewhat slower, especially for very large or strongly indefinite problems. With somewhat imperfect preconditioning, Craig's method is more robust than GMRES(k) for $k \ll n$ for homotopy curve tracking.

7. Parallel HOMPACK

This chapter describes the second part of the research work undertaken, i.e., the parallelization of the HOMPACK code. More specifically, within the HOMPACK package, we look at different ways to parallelize the function and Jacobian matrix evaluations and the linear system solver code. We describe the various parallel algorithms that were implemented, and discuss relevant issues arising from the parallel execution of programs as compared to serial execution. The motivation behind the parallel work was to study and compare the effect that various degrees and levels of parallelism have in terms of speedup in execution time, the time/effort spent on implementing the parallel code and the extra memory allocated by the parallel algorithm. Amongst the various algorithms implemented for solving linear systems of equations that we described in the last chapter, we consider here only Craig's method with ILU preconditioning. Several parallel versions of the algorithm are implemented with various levels and degrees of parallelism. All the work was done on a Sequent Symmetry S81 with 10 processors.

7.1. HOMPACK algorithm.

To understand the levels and degrees of parallelism, we first describe briefly the sequential HOMPACK code used as the basis for the parallel implementation. First, we need to compute the function values and the Jacobian matrix for the coefficient matrix A. As discussed before, in the course of homotopy curve tracking, we also need to solve nonsquare linear systems of equations for the tangent vector and the normal flow iteration calculations. The subroutines PCGDS and PCGNS are called to solve the two nonsquare systems. Within PCGDS and PCGNS, these rectangular systems are converted to equivalent linear square systems. Now both these linear systems have a nonsymmetric coefficient matrix, and the *splitting* approach, discussed in Chapter 3, is used to obtain the solution. Using this approach, the final solution of the nonsymmetric linear system is obtained by solving two symmetric linear systems. Hence the combination of PCGDS and PCGNS involves the solution of four symmetric linear systems. These four systems have no data dependence between them and hence they can be solved in parallel. It is to these symmetric linear systems that Craig's preconditioned method is applied to obtain the solution. Since

a preconditioned method is used, the preconditioning matrix Q needs to be computed also. Before calls to PCGDS and PCGNS, the subroutine MFACDS is invoked to compute the ILU preconditioner. Note that there is only one preconditioning matrix to be computed since both nonsquare systems have the same coefficient matrix A with different right hand sides. With this background, we now describe the levels of parallelism.

7.2. Sequent facilities for parallel programming.

There are two facilities that were used to perform all the parallel programming, the system call m_fork and the compiler directive DOACROSS. When the system call m_fork is invoked, a new (child) process is created which is a duplicate copy of the old (parent) process, with the same data, register contents, and program counter. The child process has access to the same shared memory as the parent, along with its own private data segment. The division of the computing load adjusts automatically to the number of available processors. If there are any data dependencies, then one can synchronize parallel processes at critical points by using locks, etc.

The other parallel programming facility is the compiler directive DOACROSS and associated directives. These are a set of special directives for executing FORTRAN DO loops in parallel. With these directives, one marks the loop to be executed in parallel with the DOACROSS directive, and classifies the variables within the loop (whether shared, local, reduction, etc.). Accordingly data dependencies are identified and locks are used for synchronization.

7.3. The levels of parallelism.

There are five different levels that we considered for the parallel implementation:

7.3.1. Function and Jacobian matrix computations.

Unlike the other levels, the algorithms for this level vary from problem to problem because different problems have different computational structures for the function values and the Jacobian matrix. For the turning point problem and the shallow dome problem, the serial algorithm computes the function values and the Jacobian matrix entries with FORTRAN DO loops. Hence, for the parallel implementation, the DOACROSS directive was used to parallelize the loops and put locks on shared variables. The shallow arch problem has very complex function evaluation computations and, in fact, about 70% of the overall execution time is due to these function evaluations. There are two possible ways of implementing the parallelism at this level for this problem. One way of implementation is to analyze the FORTRAN DO loops and use the DOACROSS directive to implement the parallelism. We refer to this parallel implementation as M8 in Section 7.5, describing the algorithm acronyms. The second way (algorithm M1) is a higher level parallelism in which the columns of the Jacobian matrix are computed in parallel, with the function values still computed as in algorithm M8 above.

7.3.2. Low level linear algebra.

At this level, the lower level functions and linear algebra are implemented in parallel along with LINPACK functions and subroutines. These include , where x and y are vectors, A is a matrix, and k is a scalar:

1) DCOPY - $y = x$.

2) DSCAL - $y = k * x$.

3) DAXPY - $x = x + k * y$.

4) DDOT - the dot product $x \cdot y = x^t y$.

5) DNRM2 - $\|x\|_2$.

6) IDAMAX - finds the index of the element of a vector with maximum absolute value.

7) MULTDS - the matrix vector product $y = Ax$.

7.3.3. Computations with the preconditioner.

There are two subroutines which are candidates for parallelization at this level. The first one is MFACDS which computes the ILU preconditioner. The other one is QIMUDS which computes $Q^{-1}f$ by applying Gauss elimination to $Qx = f$. We have not shown the execution timings for this level in the tables because there was no speedup over the serial execution time. An explanation of this is given later in the results section.

7.3.4. The two linear solves within PCGDS and PCGNS in parallel.

At this level, PCGDS and PCGNS are executed serially one after the other. Within each, as explained earlier, two linear systems of equations need to be solved and these are done in parallel since they are independent of one another.

7.3.5. PCGDS and PCGNS done in parallel.

This is one level higher parallelism than the previous level. Here the subroutines PCGDS and PCGNS are executed in parallel. Note that this means that the two solves within each are still executed serially.

7.4. Degrees of parallelism.

Levels 2–5 described above can be imbedded within each other giving rise to varying degrees of parallelism. For example, if we combine level 4 and level 5, then we are executing PCGDS and PCGNS in parallel as well as the two linear solver algorithms within each of the subroutines in parallel. So actually, all four linear solves are being executed in parallel. This gives a higher degree of parallelism than simply implementing level 4 or 5 individually. For the experiments we wanted to include all possible degrees of parallelism arising from the levels of parallelism, starting from combining levels 2 and 3 and eventually implementing levels 2, 3, 4 and 5 together. Combining levels, in order to obtain the degrees of parallelism, involves implementing a DOACROSS/m_fork within a DOACROSS/m_fork. For example, combining levels 2 and 3 together involves implementing a DOACROSS within a DOACROSS. Unfortunately, all these degrees of parallelism could not be implemented because of the limitations of the Sequent parallel programming directives, the limitation being that within a m_fork or a DOACROSS, we cannot insert another DOACROSS or

m_fork. Hence due to these constraints, the experiments that could be performed included the following:

1) Levels 4 and 5 together, i.e., all four solves in parallel.

2) Levels 1, 4 and 5 together.

3) Levels 1 and 2 together.

Note that combining level 4 and level 5 involves implementing a m_fork within a m_fork which cannot be done on the Sequent. However, we could get around this problem by using a different strategy. Combining levels 4 and 5 actually means implementing the four solves in parallel as mentioned above. So, since we could not insert a second m_fork at level 4 within the first m_fork at level 5, we modified the code to implement a single m_fork which forks out to four processors, with each processor assigned code for solving a single linear system. Also, note that we could have attempted several different combinations when combining the parallel function and Jacobian computations (level 1) with the linear solver parallelization (levels 2–5). We implemented two of the possible combinations for our experiments, the ones we thought would be most useful in drawing our conclusions.

7.5. Algorithm acronyms.

M1 – Function and Jacobian matrix evaluations in parallel.

M2 – Lower level linear algebra in parallel.

M3 – PCGDS and PCGNS in parallel.

M4 – Within PCGDS and PCGNS, the two linear solves in parallel.

M5 – Combining M3 and M4, all four linear solves in parallel.

M6 – Combining M1 and M5.

M7 – Combining M1 and M2.

M8 – *only* for arch problem, refer to §7.3.1 for a description.

7.6. Tables.

Next we give a set of tables summarizing the numerical experiments. The experiments were done for the shallow arch problem, the dome problem and the turning point problem. The tables contain the execution time and the efficiency for each problem with four processors and with eight processors. In addition, there is a table showing the amount of time spent on the programming effort for each method and another one with the extra storage required by the parallel implementation as compared to the serial one.

TABLE 13
Execution time in seconds for shallow arch problem with 8 processors.

n	Serial	M1	M2	M3	M4	M5	M6	M7	M8
29	440	86	435	432	433	425	69	82	133
47	5733	939	5590	5509	5558	5489	750	925	1210

TABLE 14
Efficiency with 8 processors for shallow arch problem.

n	M1	M2	M3	M4	M5	M6	M7	M8
29	0.640	0.126	0.127	0.127	0.129	0.797	0.671	0.414
47	0.763	0.128	0.130	0.129	0.131	0.955	0.775	0.592

TABLE 15
Execution time in seconds for shallow dome problem with 8 processors.

n	Serial	M1	M2	M3	M4	M5	M6	M7
21	18	16	16	12	13	9	7	14
126	95	90	66	61	62	43	41	62
252	185	180	125	119	120	83	79	120
525	403	394	263	255	255	175	163	254
1050	753	743	484	472	478	323	312	474

TABLE 16
Efficiency with 8 processors for shallow dome problem.

n	M1	M2	M3	M4	M5	M6	M7
21	0.141	0.141	0.188	0.173	0.250	0.321	0.161
126	0.132	0.180	0.195	0.192	0.276	0.290	0.192
252	0.128	0.185	0.194	0.193	0.279	0.293	0.193
525	0.128	0.191	0.198	0.198	0.288	0.309	0.198
1050	0.127	0.194	0.199	0.197	0.291	0.302	0.199

TABLE 17
Execution time in seconds for turning point problem with 8 processors.

n	Serial	M1	M2	M3	M4	M5	M6	M7
20	5	5	5	3	3	2	2	5
125	50	46	34	28	29	20	15	29
250	87	79	56	49	50	34	26	48
500	168	153	105	94	97	65	50	91
1000	392	355	238	219	224	151	101	205

TABLE 18
Efficiency with 8 processors for turning point problem.

n	M1	M2	M3	M4	M5	M6	M7
20	0.125	0.125	0.208	0.208	0.313	0.313	0.125
125	0.136	0.183	0.223	0.216	0.313	0.417	0.216
250	0.138	0.194	0.222	0.218	0.320	0.418	0.227
500	0.137	0.200	0.223	0.216	0.323	0.420	0.231
1000	0.138	0.206	0.224	0.219	0.325	0.485	0.239

TABLE 19
Execution time in seconds with 4 processors for higher dimension problems.

n	Serial	M1	M2	M3	M4	M5	M6	M7	M8
47	5733	1496	5623	5509	5558	5489	1467	1495	1818
1050	753	763	542	477	478	323	331	532	
1000	392	364	265	219	224	151	121	234	

TABLE 20
Efficiency with 4 processors for higher dimension problems.

n	M1	M2	M3	M4	M5	M6	M7	M8
47	0.958	0.255	0.260	0.258	0.261	0.977	0.958	0.788
1050	0.247	0.347	0.395	0.394	0.583	0.569	0.354	
1000	0.269	0.370	0.447	0.438	0.649	0.810	0.419	

TABLE 21
Actual and theoretical speedups for Algorithm M6.

n	Four processors	Eight processors
47	3.91/3.94	7.64/7.76
1050	2.28/2.71	2.41/3.79
1000	3.24/3.40	3.88/5.52

TABLE 22
Programming effort in man-hours.

Cost	M1	M2	M3	M4	M5	M6	M7	M8
Incremental	100,12,3	20	20	22	25	1	1	120
Total						126,38,29	121,33,24	

TABLE 23
Amount of extra memory allocated for each method.

	M1	M2	M3	M4	M5	M6	M7	M8
$(N+1)$ vectors			5	3	11	11		

7.7. Discussion of numerical results.

As can be observed from the tables, we have not included the timings for the second level, i.e., for the preconditioning matrix related computations in parallel. We did perform the experiments for this level but did not get any speedup with either four or eight processors. The coefficient matrix for all three test problems is sparse, which means that there are only a few nonzero entries in each row or column of the matrix. Also, as mentioned in Chapter 2, these matrices are stored in the packed skyline format, so that only the nonzero matrix entries are stored. Hence, for all DO loop computations involving the coefficient matrix, the number of computations to be performed per iteration is quite a small number. As discussed in §7.3.3, the two subroutines MFACDS and QIMUDS are possible candidates for parallelization at this level. The subroutine QIMUDS computes $Q^{-1}f$ by solving $Qx = f$ by Gauss elimination (forward and back substitution), with the preconditioning matrix Q also sparse and having the same skyline structure as the coefficient matrix A. Within the subroutine, there are two loops, one for doing the forward solve and the other one for the backsolve. Both these loops can be parallelized by using the DOACROSS parallel programming directive. However, the parallelization does nothing to speed up the execution. There are two main reasons for this.

The forward substitution solves $Ly = b$ for y and the back substitution solves $Ux = y$ for x where $Q = LU$. For the forward solve, once each new y_i is computed (from $i = 1, 2, \ldots,$), the updating for each y_k, $k = i + 1, \ldots, N$ is done in parallel. Similarly, for the back solve, once each new x_i is computed (from $i = N, N-1, \ldots,$), the updating for each x_k, $k = i-1, \ldots, 1$ is done in parallel. Now since the matrices L and U are sparse, with the same skyline structure as A, as explained above, the number of floating point operations in the DO loop to be executed in parallel is small. This results in each processor not getting enough work to do to overcome the overhead cost of executing a loop in parallel, which is always present immaterial of how much work each processor has to do.

The second reason for the poor speedup is the time spent doing the extra computations in the forward solve loop with the parallel implementation as compared to its serial implementation. For the forward solve, the parallel loop spans the remaining components of the vector y which are to be updated. Each iteration of the loop which is executed in

parallel requires a few extra calculations for checking whether that particular matrix entry is within the skyline pattern since consecutive matrix entries for the loop iteration are across the spikes of the matrix and not within a spike. For the backsolve this is not a problem because the parallel loop spans the nonzero column entries of the matrix within a spike and hence no extra calculations are required to check for valid matrix entries. The same points are valid for the MFACDS subroutine also, and since there is no speedup, we have not mentioned the timings for this level in the tables. This is very unfortunate, especially for the turning point and the dome problems, because a large percentage of the total execution time is spent in the execution of these subroutines. The percentages are about 39%, 57% and 2% for the turning point, dome and the shallow arch problems, respectively.

Tables 13–18 show the execution time and the efficiency for the three test problems with eight processors. For the linear solver code only, the most efficient algorithm was M5 amongst algorithms M2, M3, M4 and M5 for all three test problems. This is what one would expect since M5 has a much higher degree of parallelism than the others, being a combination of M3 and M4. Also note that the difference in timings for M3 and M4 is very small, the reason being that within each of PCGDS and PCGNS there are only a few computations to be done before executing the code for the two linear solves. Now if there were more code before the two linear solvers' code within each of PCGDS and PCGNS, then one would expect the timings for algorithm M3 to be smaller than those for algorithm M4.

Regarding algorithms M1 and M8 for the function and Jacobian matrix evaluations, a few issues need to be discussed. First note that we have included algorithm M8 only for the shallow arch problem, because unlike the turning point and dome problems, there are two different ways of parallelizing the code for the function values and the Jacobian matrix evaluations. For the shallow arch problem, M1 is better than M8 because M1 has a higher degree of parallelism than M8. Another very interesting point to observe is the efficiency we got with algorithm M1 for the shallow arch problem as compared to the turning point problem or the dome problems. This is because about 83% of the serial execution time for the arch problem is spent computing the function values and the Jacobian matrix whereas for the turning point or dome problem, the same number is less than 2%. For the arch

problem, each processor has a lot of work to do and is not idle for long, and so as the tables reflect, the parallel implementation is very efficient. There is one more point worth mentioning here. Notice from the tables that for the dome problem, with algorithm M1, as the dimension n of the problem increases, there is no increase in the efficiency as one would usually expect for higher n, since the processors then would have more work to do. For each of the larger dimension problems, $n = 126$, $n = 252$, etc. the function values and the Jacobian matrix are each computed for the smaller $n = 21$ dimension problem, and then these values are simply duplicated for the larger dimension problems. Hence, almost the same amount of work is assigned to each processor as for the smaller n , and we do not see an increase in the efficiency as n increases.

Overall, for all the three test problems, Algorithm M6 is the best algorithm in terms of timings and the speedup obtained by the parallel algorithms. This is what we expected since M6 combines the most efficient parallel algorithm for the function values and the Jacobian matrix evaluations with that for the linear solver code. We also ran the same experiments with four processors, only for the larger dimension n for each of the test problems. In terms of the most efficient algorithm, the same discussion holds as for eight processors. Comparing the efficiencies obtained with four processors to those obtained with eight processors, some very interesting observations can be made. First, for both the turning point and the dome problems, the efficiency obtained with four processors is almost twice as good as that with eight processors. The same holds true for the shallow arch problem, for algorithms M2, M3, M4 and M5, i.e., the linear solver parallel algorithms. However, this is not true for algorithms M1, M6, M7 and M8, i.e., all the algorithms involving the parallel function and Jacobian matrix evaluations for the shallow arch problem. The reason for this can be attributed to the fact that about 83% of the total execution time is spent executing the function and Jacobian matrix evaluation code. Hence even with eight processors, there is a lot of work for each processor to do and using four processors for the same job does not increase the efficiency considerably.

Amdahl's law provides a useful way of comparing the speedup attained by a parallel implementation to the maximum speedup that can be attained taking into consideration the fraction of the total execution time that is spent on sequential code. Amdahl's law

states that if a program consists of two parts, one that is inherently sequential and one that is fully parallelizable, and if the inherently sequential part consumes a fraction f of the total computation, then the speedup is limited by

$$S_p(n) \leq \frac{1}{f + (1-f)/p},$$

where p is the number of processors used by the parallel implementation. Table 21 gives the speedup that we obtained with our best parallel implementation (Algorithm M6) along with the maximum theoretical speedup that can be obtained according to Amdahl's law for all the three test problems. Note that the fraction f of the sequential part is not very accurate and is a lower bound on the exact serial execution fraction for the algorithms so that the numbers appearing in Table 21 for the theoretical speedup could be slightly lower than shown in the table. As can be observed from the table, for all three problems, with four processors, the actual speedup obtained is quite close to the theoretical speedup. This explains why we got an overall poor speedup for the dome and the turning point problems; for these problems, the fraction f of serial execution is high, and so we cannot do any better, being limited by the theoretical speedup as the upper bound. However, note that for eight processors, the actual speedup obtained is not close to the theoretical speedup for the dome and turning point problems. Note that algorithm M6 is a combination of algorithms M1 and M5, and for the parallel implementation, algorithm M5 uses only four processors always, even if it is given eight processors. Also as observed from the table, there is only a slight increase in speedup from using eight processors as compared to four processors. The slight increase in speedup is present only because the M1 algorithm does make use of eight processors, hence the wide gap between the theoretical speedup and the actual speedup. This also explains why the timings for algorithm M5 (as well as M3 and M4) are the same for eight processors and four processors. Note that for the arch problem, the scenario is completely different since unlike the dome and turning point problems, very little time is spent in the linear solver code.

Regarding the amount of extra memory allocated for each method, as observed from Table 23, algorithms M3, M4, M5 and M6 require some extra $(N+1)$-vectors for the parallel implementation. The algorithms which don't require any extra memory are at the lowest

level of parallelization. Algorithms M5 or M6, which require the maximum amount of extra memory, have a greater degree of parallelization than the others. So, the conclusion to be drawn from this table is that the higher the level or the degree of parallelism, the more memory the parallel implementation requires. As we have already observed, in terms of efficiency, M6 is the best parallel implementation. Hence, usually there is a tradeoff between memory and speedup. One has to pay for extra memory if speedup is the final goal. However, usually the amount of extra memory required by the parallel algorithm is not very significant in relation to the speedup achieved by the parallel algorithm. For example, algorithm M6, which has the best speedup, requires 11 extra $(N+1)$-vectors for the parallel algorithm implementation, which is just a small fraction extra compared to the overall memory space required by the serial algorithm.

Another interesting issue related to parallel computing is the tradeoff between the speedup and the amount of time spent on the implementation of the parallel algorithm. Table 22 gives the number of man-hours spent for each of the parallel implementations, along with the cumulative man-hours for algorithms M6 and M7, since each of them are combinations of other algorithms. Columns for M1, M6 and M7 have three entries each since these algorithms involve computing function values and Jacobian matrices in parallel, and for each of the three test problems, varying amounts of programming effort were required. The three entries are for, respectively, the shallow arch problem, the dome problem and the turning point problem. For algorithm M1, comparing the speedup obtained with the programming effort required for each of the test problems, it is seen that the arch problem, which has the best speedup, also has the maximum amount of man-hours required to do the parallel implementation. In general, comparing the efficiencies with the programming effort required, one can draw the conclusion that they are directly proportional to each other, i.e., the algorithms which require more time and effort to be implemented in parallel usually have a higher efficiency compared to those which require fewer man-hours for the parallel implementation. We have already concluded that the higher the level/degree of the parallel implementation, the better the efficiency. Hence, we can also conclude that the higher the degree desired of the parallel algorithm, the more time needs to be spent implementing it;

as one moves up in degree or level, it gets more and more time consuming to parallelize the serial code.

Another question which arises in the same context is whether it was worthwhile spending those many hours for the parallel implementation considering the efficiency that was obtained. For example, for the arch problem, we spent 120 hours implementing Algorithm M8 obtaining an efficiency of 0.788. It was not worth attempting the parallel implementation. Similarly for the arch problem, implementing algorithms M2, M3, M4 and M5 was not worth the effort put in because the efficiency obtained was not very good. Note that for the arch problem, the amount of time spent within the linear solver code is very small, around only 10%, and each of the algorithms M2, M3, M4 and M5 attempt to parallelize sections of the linear solver code. Hence, it is not worth the effort to parallelize some part of the program if just a small fraction of the total execution time is spent within that part of the program.

Special mention needs to be made of the shallow arch problem with regard to algorithms involving the function values and Jacobian matrix evaluations, i.e., algorithms M1, M6, M7 and M8. As observed from the tables, all these algorithms require many more man-hours for the arch problem, as compared to the other test problems. The arch problem could have taken a little less programming effort had it not been for the fact that the serial code for the function values and Jacobian matrix evaluations was extremely difficult to parallelize. The code contained certain FORTRAN language constructs which caused a lot of problems when we tried to implement the code in parallel. For example, there were a lot of COMMON and EQUIVALENCE statements in the code, which were really unnecessary for certain subroutines but were simply put there for programming ease and uniformity. A lot of time was spent analyzing every one of the COMMON and EQUIVALENCE statement variables and eventually either eliminating them or for those which were required, either declaring them as local variables or passing them as parameters to the subroutines in order to make the parallel implementation work. The point is that to a certain extent, the programming effort also depends on how the serial code has been written and how easily the serial code can be modified for the parallel implementation, and not simply on the amount

of code to be implemented in parallel or what degree/level of parallelism one is attempting for the implementation.

To summarize our discussion above, there are several aspects one needs to take into consideration when attempting to implement a parallel version of a serial program. Memory and (programmer) time constraints adversely affect the efficiency of a parallel algorithm. If time constraints are an important factor, it would perhaps be wise for the programmer to analyze the serial code carefully first, and attempt to parallelize segments of the code which are easier to implement in parallel (as we saw with the shallow arch problem, because the function value and Jacobian matrix evaluation code was so difficult to parallelize, a lot of man-hours were spent implementing the code in parallel). With time constraints, better efficiency could also be acheived if one first concentrates on attempting to parallelize code segments which comparatively take up more execution time. Regarding memory, it is observed that a lower level of parallelism generally does not require any additional memory. As our experiments reflect, the higher the degree of parallelism attempted, the greater the amount of memory required by the parallel implementation. But memory is not a big issue in general, because the extra memory required by the parallel algorithm is not very significant. And of course, if there are no constraints about memory and time, then the most efficient parallel algorithm is obtained by attempting to maximize the degree of parallelism.

Most of the conclusions drawn above reaffirm existing parallel computing theory and are very general. Regarding specific conclusions to be drawn for parallel HOMPACK, some levels are simply not worth the time and effort required for the implementation, considering the speedup obtained. Algorithm M2 (lower linear algebra in parallel) was not worth the effort. It took 20 man-hours to obtain a maximum speedup of 1.65 using eight processors. Similarly, attempting to implement computations relating to the preconditioner in parallel is not worthwhile since we get no speedup at all. Regarding the other levels/degrees, the test problem used determines whether the level or degree implementation is worth the effort put in. For the dome and turning point problems, the implementation of the function values and the Jacobian matrices was not worth the effort whereas the degrees/levels relating to the linear solver code did give a good speedup considering the effort we put into the parallel implementation. For the arch problem it was exactly the opposite, although it could be debated that spending 100 hours to obtain a speedup of 6.1 with eight processors is just not worth it. Regarding extra memory allocation for the parallel implementation, the parallel algorithms require just a few extra $(N+1)$-vectors and hence memory is not an important issue for parallel HOMPACK.

8. Conclusions

In this chapter, we draw some general conclusions based on all the sequential and parallel algorithms that were implemented as part of the research work done for this thesis. We also discuss some future research work that could be done as an extension of the work done with this thesis.

8.1. General conclusions.

In the first part of the thesis, we attempted to study in detail iterative methods for solving large sparse linear systems of equations arising from homotopy curve tracking. We studied several iterative methods for the solution of such linear systems and the conclusions drawn from our experiments would provide a valuable reference for others who encounter such linear systems during their research work. There are two main conclusions drawn from the experiments. The ILU preconditioner is very efficient and as indicated by the tables, the best timings for both Craig's method and GMRES(k) were obtained with ILU preconditioning. However for homotopy curve tracking in the presence of turning points and indefinite coefficient matrices, the ILU preconditioner can create quite a few problems, as discussed in Chapter 6. The Gill-Murray preconditioner is excellent in such cases and should be the preferred preconditioner for robust homotopy algorithms. Also, Craig's method is more stable than GMRES(k), and is comparatively better whenever nearly perfect preconditioning is not possible.

As regards the experiments we performed with the parallel algorithms, we reaffirmed existing parallel computing theory regarding the factors affecting the efficiency of a parallel algorithm. For example, we performed experiments with various levels and degrees of parallelism and showed that the higher the degree or level of parallelism, the more speedup is obtained. We also studied in detail how memory and programmer time constraints adversely affect the speedup of a parallel algorithm. Specifically for the HOMPACK software that we worked on, attempting to parallelize the lower level linear algebra and preconditioner computations was not worth the effort. Attempting the function values and the Jacobian matrix evaluations in parallel is specific to the problem at hand and cannot be generalized. Hence a general purpose parallel HOMPACK, applicable to any problem, would implement the four linear solves in parallel. However, the user could use the parallel programming directives to parallelize the function and Jacobian matrix evaluation subroutines for a specific problem.

8.2. Future work.

1) For our serial experiments, the data regarding the average, maximum and minimum number of iterations was very helpful for the discussion of the results and drawing our conclusions. It would be interesting to study and test separately other aspects for the serial algorithms, too. For example, one could separately test other aspects of the iterative linear system solving algorithms such as convergence rate, sensitivity to starting point, cost of preconditioning, storage cost, computational complexity per iteration, etc.

2) As discussed in Chapter 7 for our parallel experiments, we did not get any speedup for the level involving computations with the preconditioner. The reasons have been explained in detail, and it would be worthwhile to try and get around these reasons and get some speedup for this level. One suggestion to improve the timings would be to look at other ways of storing a sparse matrix besides the one we used, i.e., the sparse skyline matrix storage scheme.

3) For our serial experiments, we did not implement all the algorithms that we mentioned in the literature survey in Chapter 3. It would be interesting to implement some other algorithms and compare their timings with those for our algorithms.

4) We mentioned in Chapter 7 that for our parallel experiments, we could not perform all the experiments relating to the degrees of parallelism because of the limitations of the Sequent parallel programming directives. One could look into this and maybe attempt the same experiments on some other parallel machine which does not have the same limitations.

1. REFERENCES

[1] O. AXELSSON, *Conjugate gradient type methods for unsymmetric and inconsistent systems of linear equations*, Linear Algebra Appl., 29 (1980), pp. 1–16.

[2] ———, *Solution of linear systems of equations: iterative methods*, in Sparse Matrix Techniques, Springer-Verlag, New York, 1976, pp. 1–51.

[3] O. AXELSSON, S. BRINKKEMPER AND V.P. ILIN, *On some versions of incomplete block-matrix factorization iterative methods*, Linear Algebra Appl., 58 (1984), pp. 3–15.

[4] P. N. BROWN AND A. C. HINDMARSH, *Reduced storage matrix methods in stiff ODE systems*, J. Appl. Math. Comp., 31 (1989), pp. 40–91.

[5] T. F. CHAN, *Deflation techniques and block-elimination algorithms for solving bordered singular systems*, Tech. Rep. 226, Dept. of Computer Sci., Yale Univ., New Haven, CT, 1982.

[6] ———, *Deflated decomposition of solutions of nearly singular systems*, Tech. Rep. 225, Dept. of Computer Sci., Yale Univ., New Haven, CT, 1982.

[7] ———, *On the existence and computation of LU-Factorizations with small pivots*, Tech. Rep. 227, Dept. of Computer Sci., Yale Univ., New Haven, CT, 1982.

[8] T. F. CHAN AND D. C. RESASCO, *Generalized deflated block-elimination*, Tech. Rep. 337, Dept. of Computer Sci., Yale Univ., New Haven, CT, 1985.

[9] T. F. CHAN AND Y. SAAD, *Iterative methods for solving bordered systems with applications to continuation methods*, SIAM J. Sci. Stat. Comput., 6 (1985), pp. 438–451.

[10] 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.

[11] P. CONCUS AND G. H. GOLUB, *A generalised conjugate gradient method for nonsymmetric systems of linear equations*, in Lecture Notes in Economics and Mathematical Systems, R. Glowinski and J. L. Lions, eds., Springer-Verlag, Berlin, 134, 1976, pp. 56–65.

[12] P. CONCUS, G. H. GOLUB, AND D. P. O'LEARY, *A generalised conjugate gradient method for the numerical solution of elliptic partial differential equations*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976, pp. 309–352.

[13] E. J. CRAIG, *Iteration procedures for simultaneous equations*, Ph.D. thesis, MIT, Cambridge, 1954.

[14] J. E. DENNIS, JR. AND K. TURNER, *Generalized conjugate directions*, Linear Algebra Appl., 88/89 (1987), pp. 187–209.

[15] T. DUPONT, R. P. KENDALL AND K. K. RACHFORD JR., *An approximate factorization procedure for solving self-adjoint elliptic difference equations*, SIAM J. Numer. Anal., 5 (1968), pp. 559–573.

[16] S. C. EISENSTAT, *Efficient implementation of a class of conjugate methods*, SIAM J. Sci. Stat. Comput., 2 (1981), pp. 1–4.

[17] S. C. EISENSTAT, H. C. ELMAN, AND M. H. SCHULTZ, *Variational iterative methods for non-symmetric systems of linear equations*, SIAM J. Numer. Anal., 5 (1983), pp. 345–357.

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

[19] D. K. FADEEV AND V. N. FADEEVA, *Computational Methods of Linear Algebra*, Freeman, London, 1963.

[20] R. FLETCHER, *Conjugate gradient methods for indefinite systems*, in Numerical Analysis Dundee 1975, G. A. Watson, ed., Springer-Verlag, New York, 1976, pp. 73–89.

[21] P. E. GILL AND W. MURRAY, *Newton-type methods for unconstrained and linearly constrained optimization*, Math. Prog., 28 (1974), pp. 311–350.

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

[23] M. R. HESTENES, *The conjugate gradient method for solving linear equations*, in Proc. Symp. Appl. Math., 6, Numer. Anal., AMS, New York, 1956, pp. 83–102.

[24] M. R. HESTENES AND E. STIEFEL, *Methods of conjugate gradients for solving linear systems*, J. Res. National Bureau of Standards, 49 (1952), pp. 409–435.

[25] A. S. HOUSEHOLDER, *The theory of matrices in numerical analysis*, Dover Publications, Inc., New York, 1964.

[26] K. C. JEA, *Generalised conjugate gradient acceleration of iterative methods*, Ph.D thesis, Math. Dept., Univ. of Texas at Austin, 1982.

[27] M. P. KAMAT, *Nonlinear transient analysis by energy minimization–theoretical basis for the ACTION computer code*, NASA Report CR-3287, 1980.

[28] M. P. KAMAT, L. T. WATSON AND J. L. JUNKINS, *A robust and efficient hybrid method for finding multiple equilibrium solutions*, Proc. Third Internat. Symposium on Numerical Methods in Engineering, Paris, France, 1983, pp. 799–808.

[29] H. H. KWOK, M. P. KAMAT, AND L. T. WATSON, *Location of stable and unstable equilibrium configurations using a model trust region quasi-Newton method and tunnelling*, Comput. & Structures, 21, (1985), pp. 909–916.

[30] C. LANCZOS, *Solution of systems of linear equations by minimized-iterations*, J. Res. National Bureau of Standards, 49 (1952), pp. 33–53.

[31] 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), 148–162.

[32] ———, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as occur in practical problems*, Tech. Rep. 550, Koninklijke/Shell Exploratie on Produktie Laboratorium, 1980.

[33] D. P. O'LEARY, *Hybrid conjugate gradient algorithms*, Ph.D. thesis, Computer Sci. Dept., Stanford Univ., 1976.

[34] ———, *The block conjugate gradient algorithm and related methods*, Linear Algebra Appl., 29 (1980), pp. 293–322.

[35] J. M. ORTEGA, *Efficient implementations of certain iterative methods*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 882–891.

[36] C. C. PAIGE AND M. A. SAUNDERS, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12 (1975), pp. 617–629.

[37] W. C. RHEINBOLDT, *Numerical analysis of continuation methods for nonlinear structural problems*, Comput. & Structures, 13 (1981), pp. 103–113.

[38] ———, *Numerical Analysis of Parametrized Nonlinear Equations*, Wiley-Interscience, New York, 1986.

[39] W. C. RHEINBOLDT AND J. V. BURKARDT, *Algorithm 596: A program for a locally parameterized continuation process*, ACM Trans. Math. Software, 9 (1983), pp. 236–241.

[40] J. K. REID, *On the method of conjugate gradients for the solution of sparse systems of linear equations*, in Large Sparse Sets of Linear Equations, J. K. Reid, ed., Academic Press, New York, 1971, pp. 231–254.

[41] Y. SAAD, *Krylov subspace methods for solving large unsymmetric linear systems*, Math. Comput., 37 (1981), pp. 105–126.

[42] ———, *Practical use of some Krylov subspace methods for solving indefinite and unsymmetric linear systems*, Tech. Rep. 214, Dept. of Computer Sci., Yale Univ., New Haven, CT, 1982.

[43] Y. SAAD AND M. H. SCHULTZ, *Conjugate gradient-like algorithm for solving nonsymmetric linear systems*, Math. Comp., 44/170 (1985), pp. 417–424

[44] ———, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869.

[45] R. S. VARGA, *Matrix iterative analysis*, Prentice-Hall, New York, 1962.

[46] P. K. W. VINSOME, *Orthomin, an iterative method for solving sparse sets of simultaneous linear equations*, in Proc. Fourth Symp. on Reservoir Simulation, Soc. of Petroleum Engineers of the AIME, 1976, pp. 149–159.

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

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

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

[50] ———, *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.

[51] ———, *Solving finite difference approximations to nonlinear two-point boundary value problems by a homotopy method*, SIAM J. Sci. Stat. Comput., 1 (1980), pp. 467–480.

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

[53] ———, *Globally convergent homotopy methods: A tutorial*, Tech. Rep. TR–87–13, Dept. of Industrial and Operations Eng., Univ. of Michigan, Ann Arbor, MI, 1985.

[54] 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.

[55] 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.

[56] L. T. WATSON AND M. R. SCOTT, *Solving spline-collocation approximations to nonlinear two-point boundary-value problems by a homotopy method*, Appl. Math. Comput., 24 (1987), pp. 333–357.

[57] L. T. WATSON AND L. R. SCOTT, *Solving Galerkin approximations to nonlinear two-point boundary value problems by a globally convergent homotopy method*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 768–789.

[58] O. WIDLUND, *A Lanczos method of a class of non-symmetric systems of linear equations*, SIAM J. Numer. Anal., 15 (1978), 801–812.

[59] D. M. YOUNG, *Iterative solution of large linear systems*, Academic Press, New York, 1971.

[60] D. M. YOUNG AND K. C. JEA, *Generalised conjugate gradient acceleration of iterative methods. Part 2: the nonsymmetrizable case*, Tech. Rep. CNA-163, Center for Numerical Analysis, Univ. of Texas at Austin, 1981.

[61] ———, *Generalised conjugate gradient acceleration of nonsymmetrizable iterative methods*, Linear Algebra Appl., 34 (1980), pp. 159–194.

Vita

Kashmira Irani was born in Bombay, India. She graduated from St. Xaviers College, India with a Bachelor of Science Degree in Mathematics and Statistics in 1985. She completed her Master of Science degree in Mathematics in 1987 from Virginia Polytechnic Institute and State University, and then joined the Computer Science Masters program at the same school. She is currently working with Pilot Research Associates in Vienna, Virginia.

Kashmira M. Irani