CGCODE: SOFTWARE FOR SOLVING LINEAR SYSTEMS WITH CONJUGATE GRADIENT METHODS

BY

MICHAEL JAY HOLST

B.S., Colorado State University, 1987

THESIS

**Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990**

Urbana, Illinois

Acknowledgements

I would like to thank my advisor, Paul Saylor, for his generous financial support while I have been at the University of Illinois, and for his personal guidance and friendship during the last three years. His wonderful family has also provided emotional support, cheesecake, and lodging when I needed it most. I would like to thank Steve Ashby for all the time and effort he has put into this project, and for his advice and unlimited patience. And, a thank you to everyone in the NA group, who have helped make my life bearable these last few months.

I would also like to thank my family for their tireless support, both emotional and financial, over the last eight years. Finally, thank you Mai, for always being there for me when I needed you.

Contents

Chapter

Chapter 1

Introduction

In many scientific applications it is necessary to solve a system of linear algebraic equations, $Ax = b$. Depending on the application, these systems may be real or complex. They typically arise when a continuous problem, such as a system of differential equations, is discretized so that it may be solved approximately using a numerical technique on a digital computer. Linear systems arise in this way in applied mathematics, physics, chemistry, atmospheric science, computer science, and especially engineering. Direct methods, such as matrix factorization, are commonly used to solve these systems. However, in many important cases, such as the solution of partial differential equations, the linear systems are large and sparse. Since direct methods are too expensive for such systems, iterative methods are used.

Many scientific problems lead to linear systems with special properties. In the solution of partial differential equations, for example, it is common for the resulting linear system to be Hermitian positive definite (hpd). (The symmetry in the underlying differential operator gives rise to a symmetric system matrix when symmetry-preserving discretization techniques are used.) One of the most popular and effective iterative methods for solving hpd linear systems is the conjugate gradient (CG) method of Hestenes and Stiefel [19]. We will examine this method, and variants of it, in the pages that follow.

In many other scientific problems, large and sparse nonhermitian and/or indefinite linear systems arise. The traditional CG method of Hestenes and Stiefel cannot be used for these systems because convergence is guaranteed only for hpd linear systems. However, variants of the CG method, such as conjugate residuals and CG on the normal equations, can be used for Hermitian indefinite and nonhermitian linear systems, respectively. Unfortunately, there are disadvantages to

some of these methods. For example, CG on the normal equations has the property of squaring the condition number of the original matrix, resulting in slower convergence of the method. (However, CG on the preconditioned normal equations often works quite well.) There are several methods that avoid some of these difficulties, including adaptive Chebyshev [2, 24, 25] and GMRES [33].

1.1 What is CgCode?

In this report we describe a package called CgCode. It is a collection of conjugate gradient algorithms, written in FORTRAN 77, for solving linear systems. We consider only *3-term* CG methods (Chapter 3). CG-like methods, such as Orthomin(k) and GMRES(k), are neither considered here nor implemented in CgCode. See [15] for a discussion of some of these methods.

CgCode was originally developed at Los Alamos National Laboratory by Steven Ashby and Thomas Manteuffel. Their original implementation has been expanded to include several new methods, and the user interface has been modified to conform to a proposed standard for iterative linear solvers [7]. The resulting package, which is easy to use and well-documented, combines several well-known CG methods with some special features, including:

- dynamic eigenvalue estimation,
- a standard user interface,
- matrix data structure independence,
- a menu of stopping criteria, including one based on the true error,
- adaptive Chebyshev polynomial preconditioning for hpd A.

The package contains eleven user-callable subroutines, including the driver subroutine, which allows the user to easily call any of the implemented methods. The user simply provides CgCode with a few parameters, some work space, and two subroutines: one to compute a matrix-vector product and another to effect preconditioning. A single call to CgCode returns the solution and various messages. See Chapter 2 for a complete description of these required parameters and subroutines. While many users will solve only real linear systems, complex systems arise in many applications, and so CgCode is available in complex arithmetic, as well as single and double precision real arithmetic.

1.2 Outline of the Report

Chapter 2 is a self-contained user's guide to CgCode. It contains all the information needed to use CgCode, including a complete description of all required parameters and subroutines. If the user is primarily concerned with solving a linear system, Chapter 2 should be read carefully, with special attention paid to the examples presented. If the user is interested in the methods in CgCode, Chapters 3, 4, and 5 provide an introduction to CG methods, preconditioning, and stopping criteria, respectively. Chapter 6 discusses future releases of CgCode and gives information about obtaining a copy of CgCode. A sample program is given in the Appendix.

Chapter 2

How to use CgCode

This chapter describes CgCode in detail. We first describe our user interface philosophy and then examine its two key components: the matrix-vector multiplication routine MATVEC, and the preconditioning routine PCONDL. Finally, we describe each of the parameters required by the CgCode package interface routine, CGDRV[1]. After reading this chapter, and perhaps consulting the sample program in the Appendix, one should have all the information needed to use the package to solve a linear system $Ax = b$.

To use CgCode, one simply calls the driver routine, CGDRV, supplying it with certain parameters, subroutines, and work space, all passed through the CGDRV parameter list. On return, CgCode provides the solution of the linear system, perhaps with a few messages. If an error was encountered, CgCode will return prematurely with an error flag set to an appropriate value, telling what went wrong. The user is expected to provide the following:

- a matrix-vector multiplication routine, which we call MATVEC,
- a preconditioning routine, which we call PCONDL,
- integer and real work space,
- the right-hand side b and an initial guess x_0,
- a few control parameters, such as the error tolerance.

Each of these parameters will be described in full below. First, we will discuss the philosophy behind our approach.

[1] CGDRV is the generic name for the interface routine; the actual subroutine names are SCGDRV, DCGDRV, and CCGDRV for the single, double and complex arithmetic implementations, respectively.

CgCode conforms to the user interface proposed in [7], which has several advantages. To begin with, having some type of standardized user interface enables the user to more easily try different packages, without having to code a different interface for each package. This user interface also

- is easy to use and flexible,
- is problem-independent,
- is independent of the matrix storage format,
- provides a consistent way to use different preconditioners,
- allows the user to select from a menu of stopping criteria.

Although it may seem inconvenient to require the user to supply the matrix-vector multiplication routine MATVEC and the preconditioning routine PCONDL, we believe the benefits outweigh the disadvantages. All iterative methods require matrix-vector products. If a user has tried another package, he may have already written a subroutine to perform this operation. Moreover, he may have invested a great deal of time into optimizing this routine for a particular machine and problem. Thus, one advantage of our approach is that an existing matrix-vector multiplication subroutine can be reused by treating MATVEC as an *interface* to it (see below). A second advantage is this: since the matrix-vector multiplication routine is the only routine that needs to know how the matrix A is stored, the user is free to employ any data structure he desires. CgCode calls MATVEC, passing in the vector x, and MATVEC (usually through the user's existing routine) returns the product $y = Ax$. In other words, CgCode in no way depends on the data structure chosen for A. Moreover, one can easily experiment with different storage formats by modifying the matrix-vector multiplication routine. Future releases of CgCode will offer a menu of data structures and corresponding MATVEC routines. The user would then have the choice of providing his own MATVEC or using one of those provided by CgCode. Of course, if he chooses the latter, he must conform to a particular data structure.

Most iterative methods also need some sort of preconditioning. This can be thought of as the solution of another linear system, $My = x$, where x is a vector from the iteration and M approximates A, but is easier to invert[2]. Alternately, we can think of the preconditioning step as the product of M^{-1} and x, where the *preconditioner* M^{-1} approximates the *inverse* of A. We choose to view the preconditioning step as just another matrix-vector multiplication, $y = M^{-1}x$. Once

[2]This means the solution of $My = x$ can be easily computed. The inverse of M is seldom computed explicitly.

we adopt this view, it is clear that the routine PCONDL yields the same benefits as MATVEC: (1) the solver is kept independent of the storage format chosen for the preconditioner; and (2) an existing preconditioning subroutine can be reused by treating PCONDL as an interface to it. This is especially advantageous if the user has such a routine that exploits special knowledge of the problem. Moreover, this approach allows the use of another package's preconditioning software. Note that different data structures can be used for M^{-1} and A.

2.1 Matrix-Vector Multiplication

All iterative methods require the matrix-vector product $y = Ax$ at some point in the iteration. To use CgCode, one must provide an interface subroutine, MATVEC, that will return this product. The parameter list of MATVEC *must* have the form:

MATVEC(JOB,A,IA,W,X,Y,N)

All real work space needed to effect the matrix-vector multiplication must be passed through the work array A. It will usually contain the nonzeros of the matrix A. Similarly, all integer work space must be passed through IA, which usually contains additional information about how the matrix A is stored in the array A. Both A and IA must be initialized in the calling program. The parameter X is the input vector x, Y is the product vector $y = Ax$, and N is the order n of A, as well as the length of X and Y. The parameter JOB indicates the matrix-vector product to performed (see below). The input vector W is used in conjunction with JOB to compute some of these products. (Here we consider the products $y = Ax$ and $y = A^*x$, so W can be considered a dummy parameter.)

If the user already has a routine, USERMV, that computes a matrix-vector product, but does not have the specified parameter list, then MATVEC can be treated as an *interface* between CgCode and USERMV. CgCode will call MATVEC with the prescribed argument list, and MATVEC will call USERMV, which may have any argument list whatsoever. USERMV then computes the actual matrix-vector product, passing the result back through MATVEC to CgCode. Of course, MATVEC may be written to compute $y = Ax$ directly.

The short example given in Figure 2.1 may help make this clearer. In this example, the user has a pentadiagonal matrix A stored in the array A as 5 vectors. The starting index in A for each diagonal was previously stored in the array IA in the calling program. The user already has

```
      SUBROUTINE MATVEC(JOB,A,IA,W,X,Y,N)
      DIMENSION A(*),IA(*),W(*),X(*),Y(*)
C
C     MATVEC is an interface to USERMV, which expects the five diagonals
C     of the matrix A to be passed in separately.  The starting position of
C     each diagonal in the array A was stored in IA in the calling program.
C
      CALL USERMV(A(IA(5)),A(IA(3)),A(IA(1)),A(IA(2)),A(IA(4)),
     2            IA(6),IA(7),X,Y)
      RETURN
      END
C
C
C
      SUBROUTINE USERMV(CL,BL,A,BU,CU,NX,NY,X,Y)
      DIMENSION CL(*),BL(*),A(*),BU(*),CU(*),X(*),Y(*)
C
C     This subroutine computes the product of the pentadiagonal matrix
C     consisting of CL,BL,A,BU,CU and the vector X to produce the result Y,
C     where N=NX*NY is the length of each vector and each diagonal.
C
      RETURN
      END
```

Figure 2.1: Using MATVEC as an interface to USERMV.

a subroutine USERMV to form the product $y = Ax$, but it has a different parameter list than that required for MATVEC. To use USERMV, the user writes MATVEC as an interface routine. When CgCode needs a matrix-vector product, it calls MATVEC, which in turn calls USERMV. The matrix-vector product is then performed in USERMV and the result is passed back through MATVEC to CgCode.

2.2 Preconditioning

CgCode currently supports two types of preconditioning: (1) user-supplied preconditioning via PCONDL; and (2) adaptive Chebyshev polynomial preconditioning, available if A is Hermitian positive definite. Additional preconditioners will be offered in future releases of CgCode. We first

discuss user-supplied preconditioning, implemented as *left* preconditioning in CgCode. (As we will see in Chapter 4, left preconditioning is sufficient for CG methods.)

If a preconditioner C is employed, the iterative method needs the product $y = Cx$ at some point in the iteration. If the preconditioner is obtained from an incomplete factorization (see Chapter 4), this might be implemented via the solution of lower and upper triangular linear systems. The preconditioning might alternatively be implemented as several iterations of an iterative method such as SSOR, or something altogether different. To use this type of preconditioning in CgCode, the user must provide the routine PCONDL to CgCode. It must have the following parameter list, similar to that of MATVEC:

PCONDL(JOB,Q,IQ,W,X,Y,N)

The arrays Q and IQ usually contain information about the preconditioner C, such as the nonzeros and how they were stored by the calling program; c.f., the discussion of A and IA of MATVEC. The other parameters have the same meaning and function as those described for MATVEC. As with MATVEC, the PCONDL subroutine may be used as an interface to an existing subroutine, which we call USERPC.

CgCode also supports adaptive Chebyshev polynomial preconditioning for Hermitian positive definite systems. The theory of polynomial preconditioning is summarized in Chapter 4; here we will describe how to use it in CgCode. If the user wishes to use polynomial preconditioning, he must provide:

- NDEG: the degree of the Chebyshev preconditioning polynomial,
- AA: an initial estimate of the minimum eigenvalue of A,
- BB: an initial estimate of the maximum eigenvalue of A.

Numerical experiments [3, 4, 5] indicate that the optimum degree for the polynomial varies between 2 and 16, with the optimum degree increasing as the condition number of A, $\kappa(A)$, increases. In addition, for the adaptive procedures to work correctly, NDEG must be even. If the matrix has been scaled by its diagonal (Chapter 4), then setting AA=1.0 and BB=1.0 is recommended. Another acceptable initial value for AA and BB is $\langle Ab, b\rangle / \langle b, b\rangle$. Note that an *inner* preconditioning can be employed by way of the PCONDL routine.

2.3 Calling CgCode

We now consider CGDRV, which is the user's primary interface to CgCode. The call to CGDRV has the following form:

```
      CALL CGDRV(MATVEC,PCONDL,PCONDR,A,IA,X,B,N,Q,IQ,P,IP,
     .           IPARAM,RPARAM,IWORK,RWORK,IERROR)
```

CGDRV is an interface to CgCode, which consists of 10 conjugate gradient algorithms. Each algorithm is targeted at linear systems with different characteristics, using various combinations of preconditioning strategies and conjugate gradient methods. The parameter IPARAM(33)=ICG selects the CG algorithm to be used. This and the other parameters required by CgCode will now be described in detail.

MATVEC External Subroutine MATVEC(JOB,A,IA,W,X,Y,N).
The user must provide a subroutine having the specified parameter list. MATVEC will usually serve as an interface to the user's own matrix-vector multiply subroutine. The subroutine must return the desired product, as specified by JOB:

JOB=0 $y = Ax$
JOB=1 $y = A^*x$
JOB=2 $y = w - Ax$
JOB=3 $y = w - A^*x$.

Here A^* denotes the Hermitian transpose of A. Note that only JOB=0,1 are required for CgCode. All the routines in CgCode require JOB=0; the routines CGNR, CGNE, PCGNR, and PCGNE also require JOB=1. (JOB=2,3 are not currently required by any of the routines in CgCode, but may be required by other iterative packages conforming to the proposed iterative standard.) The parameters W,X,Y are all vectors of length N. (If JOB=0,1 W may be a dummy argument.) A and IA are real and integer work arrays, usually containing the nonzeros of A and additional information about how A is stored in A. These arrays are provided solely for the user's convenience; they are simply passed as addresses to MATVEC. Note: MATVEC must be declared in an EXTERNAL statement in the calling program.

PCONDL External Subroutine PCONDL(JOB,Q,IQ,W,X,Y,N).
If preconditioning is desired, the user must provide a subroutine having the specified parameter list. PCONDL will usually serve as an interface to the user's own preconditioning routine. The integer parameter JOB specifies the product to be computed:

JOB=0 $y = Cx$
JOB=1 $y = C^*x$
JOB=2 $y = w - Cx$
JOB=3 $y = w - C^*x$.

Note that only JOB=0,1 are required for CgCode. The routines PCG, PCGNR, PCGNE, PPCG, and PCGCA in CgCode require JOB=0; the routines PCGNR and PCGNE also require JOB=1. (JOB=2,3 are not currently required by any of the routines in CgCode, but may be required by other iterative packages conforming to the proposed iterative standard.) The parameters W,X,Y are all vectors of length N. (If JOB=0,1 W may be a dummy argument.) Q and IQ are real and integer work arrays, analogous to A and IA of MATVEC. Note: PCONDL must be declared in an EXTERNAL statement in the calling program. If no preconditioning is being done, PCONDL is a dummy argument.

PCONDR Dummy argument.
This parameter is mandated by the proposed standard [7].

A Real array address.
The base address of the user's real work array, which usually contains the nonzeros of the matrix A. Since A is only accessed by calls to subroutine MATVEC, it may be a dummy address.

IA Integer array address.
The base address of the user's integer work array. This usually contains additional information about A needed by MATVEC. Since IA is only accessed by calls to MATVEC, it may be a dummy address.

X Real(N).
The initial guess vector, x_0. On return, X contains the approximate solution to $Ax = b$.

B Real(N).
The right-hand side vector of the linear system $Ax = b$. B is changed by the solver.

N Integer.
The order of the matrix A in the linear system $Ax = b$.

Q Real array address.
The base address of the user's left preconditioning array, Q. Since Q is accessed only by calls to PCONDL, it may be a dummy address. If no left preconditioning is done, this is a dummy argument.

IQ Integer array address.
The base address of an integer work array associated with Q. This provides the user with a way of passing integer information about Q to PCONDL. Since IQ is accessed only by calls to PCONDL, it may be a dummy address. If no left preconditioning is done, this is a dummy argument.

P Dummy argument.
This variable is mandated by the proposed standard [7].

IP Dummy argument.
This variable is mandated by the proposed standard [7].

IPARAM Integer(34).
An array of integer input parameters. IPARAM(1) through IPARAM(10) are mandated by the proposed standard; IPARAM(11) through IPARAM(30) are reserved for expansion of the proposed standard; IPARAM(31) through IPARAM(34) are additional parameters, specific to CgCode.

IPARAM(1) = NIPAR
Length of the IPARAM array. It should be set to 34.

IPARAM(2) = NRPAR
Length of the RPARAM array. It should be set to 34.

IPARAM(3) = NIWK
Length of the IWORK array. See the description of IWORK below.

IPARAM(4) = NRWK
Length of the RWORK array. See the description of RWORK below.

IPARAM(5) = IOUNIT
If IOUNIT > 0 then iteration information (as specified by IOLEVL) is written to unit=IOUNIT, which must be opened in the calling program. If IOUNIT ≤ 0, no output is generated.

IPARAM(6) = IOLEVL
Specifies the amount and type of information to be output if IOUNIT > 0:

If IOLEVL=0 Output error messages only.
If IOLEVL=1 Output input parameters and level 0 information.
If IOLEVL=2 Output STPTST (see below) and level 1 information.
If IOLEVL=3 Output level 2 information and more details.

For more information on the amount and type of information produced by CgCode for each of these values, see the documentation and [7].

IPARAM(7) = IPCOND
Preconditioning flag, specified as:

If IPCOND=0 No preconditioning.
If IPCOND=1 Left preconditioning.
If IPCOND=2 Right preconditioning.
If IPCOND=3 Both left and right preconditioning.

Note: right preconditioning is an option provided by the proposed standard [7], but not implemented in CgCode.

IPARAM(8) = ISTOP
Stopping criterion flag:

If ISTOP=0	then use:	$\|e_i\|/\|x\|$	$\leq \epsilon$ (DEFAULT)
If ISTOP=1	then use:	$\|r_i\|$	$\leq \epsilon$
If ISTOP=2	then use:	$\|r_i\|/\|b\|$	$\leq \epsilon$
If ISTOP=3	then use:	$\|Cr_i\|$	$\leq \epsilon$
If ISTOP=4	then use:	$\|Cr_i\|/\|Cb\|$	$\leq \epsilon$

where $e_i = x - x_i$, $r_i = b - Ax_i$, ϵ=ERRTOL, and C is the preconditioning matrix. (Here $\|\cdot\|$ denotes the Euclidean norm.) If ISTOP=0 is selected by the user, then ERRTOL is the amount by which the initial error is to be reduced. By estimating the condition number of the iteration matrix, the code attempts to guarantee that the final relative error is less than or equal to ERRTOL. See Chapter 5 for details.

IPARAM(9) = ITMAX
The maximum number of iterative steps to be taken. If the solver is unable to satisfy the stopping criterion within ITMAX iterations, it returns to the calling program with IERROR=-1000.

IPARAM(10) = ITERS
On return, the number of iterations actually taken. If IERROR=0, then x_{ITERS} satisfies the specified stopping criterion. If IERROR=-1000, CgCode was unable to converge within ITMAX iterations, and x_{ITERS} is CgCode's best approximation to the solution of $Ax = b$.

IPARAM(31) = ICYCLE
The frequency with which a condition number estimate is computed; this is used in the stopping criterion.

IPARAM(32) = NCE
The maximum number of condition number estimates to be computed. If NCE = 0 no estimates are computed. See ICYCLE and ISTOP above. Note: KMAX = ICYCLE*NCE is the order of the largest orthogonal section of CA used to compute a condition number estimate. This estimate is only used in the stopping criterion. As such, KMAX should be much less than N. Otherwise the code will have excessive storage and work requirements.

IPARAM(33) = ICG
A flag specifying the method to be used. In the table below, A is the system matrix, C is a preconditioning matrix, and $C(A)$ is a preconditioning polynomial. If (ICG < 1) or (ICG > 10), then ICG is set to 1.

ICG	Method	Restrictions	Comments
1	CGHS	A hpd	CGHS on A
2	CR	A hpd	CR on A
3	CRIND	A Hermitian	CR on A
4	PCG	A,C hpd	PCG on A
5	CGNR	none	CGHS on A^*A
6	CGNE	none	CGHS on AA^*
7	PCGNR	none	CGNR on AC
8	PCGNE	none	CGNE on CA
9	PPCG	A,C hpd	Polynomial PCG on A
10	PCGCA	A,C hpd	CGHS on $C(A)A$

IPARAM(34) = NDEG
When using PPCG and PCGCA, NDEG specifies the degree of the preconditioning polynomial to be used.

RPARAM Real(34).
An array of real input parameters. RPARAM(1) and RPARAM(2) are mandated by the proposed standard; RPARAM(3) through RPARAM(30) are reserved for expansion of the proposed standard; RPARAM(31) through RPARAM(34) are additional parameters, specific to CgCode.

RPARAM(1) = ERRTOL
User provided error tolerance; see ISTOP above.

RPARAM(2) = STPTST
On return, STPTST is the quantity used in the stopping criterion; see ISTOP above.

RPARAM(31) = CONDES
An initial estimate of the condition number of the iteration matrix. An acceptable initial value is 1.0. On return, CONDES is the final estimate used in the stopping criterion; see ISTOP above.

RPARAM(32) = AA
Required only when using PPCG or PCGCA. AA is an initial estimate of the smallest eigenvalue of A. On return, AA is the final estimate of the smallest eigenvalue of A.

RPARAM(33) = BB
Required only when using PPCG or PCGCA. BB is an initial estimate of the largest eigenvalue of A. On return, BB is the final estimate of the largest eigenvalue of A.

RPARAM(34) = SCRLRS
On return, SCRLRS is the final relative residual.

RWORK Real(N1+N2).
Work array, where N1 and N2 are integers satisfying:

routine to be used	N1 is at least	N2 is at least
CGHS	2*N	4*ICYCLE*NCE+2
CR	3*N	4*ICYCLE*NCE+2
CRIND	5*N	4*ICYCLE*NCE+2
PCG	2*N	4*ICYCLE*NCE+2
CGNR	2*N	4*ICYCLE*NCE+2
CGNE	2*N	4*ICYCLE*NCE+2
PCGNR	3*N	4*ICYCLE*NCE+2
PCGNE	3*N	4*ICYCLE*NCE+2
PPCG	6*N	2*ICYCLE*NCE+2
PCGCA	6*N	2*ICYCLE*NCE+2

The N2 space is for computing condition number estimates; the N1 space is for temporary vectors. To save storage and work, ICYCLE*NCE should be much less than N. If NCE = 0, N2 may be set to zero.

IWORK Integer(ICYCLE*NCE).
Integer work array for computing condition number estimates. If NCE = 0, this may be a dummy address.

IERROR Integer.
Returned error flag (negative errors are fatal):

If IERROR=0	Normal return: iteration converged.
If IERROR=−1000	Method failed to converge in ITMAX steps.
If IERROR=±2000	Error in user input.

This brings to a conclusion our discussion of the use of CgCode. We have presented the user interface philosophy employed in CgCode, and have explained how to use the CgCode package through use of the interface subroutine CGDRV. We have also described each of the required input parameters, as well as the two required subroutines, MATVEC and PCONDL. A sample program illustrating the use of CgCode to solve the linear system arising from the discretization of

a partial differential equation is given in the Appendix. For other examples illustrating the use of the MATVEC and PCONDL interface subroutines, see [7].

If the reader is interested in the basic theory of conjugate gradient methods and preconditioning, he should continue reading. Conjugate gradient methods are discussed in Chapter 3, a few preconditioning techniques are described in Chapter 4, and stopping criteria are discussed in Chapter 5. Instructions for obtaining a copy of CgCode are given in Chapter 6.

Chapter 3

Conjugate Gradient Methods

In this chapter we describe conjugate gradient methods. We begin with gradient methods and then discuss CG methods in the context of minimization and orthogonality. Following the taxonomy introduced in [6], we give necessary and sufficient conditions for the convergence of CG methods, and then present two algorithms for implementing CG methods. Finally, we review the methods implemented in CgCode. Much of what follows is summarized from [6], with permission of the authors.

The classical conjugate gradient method for Hermitian positive definite (hpd) matrices was proposed by Hestenes and Stiefel in [19]. Much work has since been done; for example, see [6, 12, 18]. There are several ways to introduce CG methods, such as the minimization of a quadratic functional [9], or as a variation of the Lanczos procedure [35]. Here, for the generality we will need later, we choose to begin with the Cayley-Hamilton theorem [36]. This theorem states that an $n \times n$ matrix A satisfies its *characteristic equation*:

$$P_n(A) = 0$$

where P_n is the *characteristic polynomial* of degree n. It can be shown that if A has only $m \leq n$ distinct eigenvalues, then there exists a polynomial Q_m of degree m such that

$$Q_m(A) = \sum_{i=0}^{m} c_i A^i = 0.$$

If m is the smallest degree for which this is true, then Q_m is called the *minimum polynomial* of A. The degree m of the minimum polynomial of A is denoted by $d(A)$. If we assume A is nonsingular,

then we know $c_0 \neq 0$. After some algebra, we find that

$$A^{-1} = \frac{1}{c_0} \sum_{i=0}^{m-1} c_{i+1} A^i = \hat{Q}_{m-1}(A).$$

In other words, if A has m distinct eigenvalues, then A^{-1} can be written as a polynomial of degree $m-1$ in A.

Let x_0 be our initial approximation to x, and define the *initial residual* to be $r_0 = b - Ax_0$. Multiplying by A^{-1} yields

$$x = A^{-1}b = x_0 + A^{-1}r_0 = x_0 + \hat{Q}_{m-1}(A)r_0.$$

If we define the *Krylov subspace of degree* $i+1$ of the matrix A with respect to the vector r_0 by

$$V_{i+1}(r_0, A) = \text{sp}\{r_0, Ar_0, A^2r_0, \ldots, A^i r_0\},$$

where $\text{sp}\{\ldots\}$ denotes the *span* of the enclosed vectors, then we see that the solution x to the linear system $Ax = b$ lies in the translated Krylov space, $x \in x_0 + V_m(r_0, A)$.

A *gradient* (or *polynomial*) *method* is an iteration that produces a sequence of approximations to $x = A^{-1}b$ by

$$x_{i+1} = x_i + d_i,$$

where $d_i \in V_{i+1}(r_0, A)$. By choosing the d_i in different ways, we define different gradient methods. Although the solution $x = A^{-1}b$ lies in the m-dimensional space $x_0 + V_m(r_0, A)$, gradient methods may not find this solution, since there is no guarantee that the d_i span the space. It is therefore desirable to have methods for which $\{d_i\}_{i=0}^m$ forms a *basis* for the space $V_m(r_0, A)$.

Let $e_{i+1} = x - x_{i+1}$ denote the error at step $i+1$ of a gradient method. We denote the Euclidean inner product as $\langle \cdot, \cdot \rangle$, and for an hpd inner product matrix B we define the B-inner product as $\langle B\cdot, \cdot \rangle$. The hpd matrix B also induces a norm, $\|v\|_B = \langle Bv, v \rangle^{\frac{1}{2}}$. A *conjugate gradient method* is a gradient method that chooses $d_i \in V_{i+1}(r_0, A)$ to minimize $\|e_{i+1}\|_B$. We will denote this method by CG(B,A). It can be shown [31] that choosing d_i in this way is equivalent to enforcing B-orthogonality between e_{i+1} and $V_{i+1}(r_0, A)$, written as $e_{i+1} \perp_B V_{i+1}$. That is, we require $\langle Be_{i+1}, v \rangle = 0$ for all $v \in V_{i+1}(r_0, A)$.

Assume we have enforced this condition at step i, i.e., $e_i \perp_B V_i$. At step $i+1$, we wish to impose $e_{i+1} = x - (x_i + d_i) = e_i - d_i \perp_B V_{i+1}$. Since $V_i \subset V_{i+1}$, we see that forcing $e_{i+1} \perp_B V_{i+1}$

implies $d_i \perp_B V_i$. The vector d_i is the new vector in V_{i+1} orthogonal to V_i. Hence, a CG method not only chooses the d_i as a basis for $V_m(r_0, A)$, but as an orthogonal basis for the space. Since $x \in x_0 + V_m(r_0, A)$ and V_m is m-dimensional, this implies that the CG method must terminate (i.e., find the true solution to $Ax = b$) in at most m steps (assuming exact arithmetic). More precisely, CG(B,A) converges in at most $d(r_0, A)$ steps, where $d(r_0, A) \leq d(A)$ is the degree of the polynomial $\tilde{Q}(A)$ of smallest degree such that $\tilde{Q}(A)r_0 = 0$. $\tilde{Q}(A)$ is called the minimum polynomial of A with respect to r_0.

How does one choose the vector d_i in V_{i+1} to be orthogonal to V_i? We can construct a B-orthogonal basis $\{p_k\}_{k=0}^{i}$ for $V_{i+1}(r_0, A)$ using the Gram-Schmidt process:

$$p_{k+1} = Ap_k - \sum_{j=0}^{k} \sigma_{kj} p_j, \qquad \sigma_{kj} = \frac{\langle BAp_k, p_j \rangle}{\langle Bp_j, p_j \rangle}, \qquad k = 0, 1, \ldots, i-1,$$

where $p_0 = r_0$. We call the p_k *direction vectors*. Since the set $\{p_k\}_{k=0}^{i}$ is constructed so that $p_i \in V_{i+1}$ and $p_i \perp V_i$, we have $d_i = \alpha_i p_i$ for some scalar α_i. Therefore,

$$x_{i+1} = x_i + \alpha_i p_i.$$

To determine α_i, recall that minimization of $\|e_{i+1}\|_B$ requires that $\langle Be_{i+1}, p_i \rangle = 0$. Since $e_{i+1} = e_i - \alpha_i p_i$, we have

$$\langle Be_i, p_i \rangle - \alpha_i \langle Bp_i, p_i \rangle = 0,$$

and thus

$$\alpha_i = \frac{\langle Be_i, p_i \rangle}{\langle Bp_i, p_i \rangle}.$$

Since this expression involves the unknown quantity e_i, we must choose B so as to obtain a *computable* method. We now have a method CG(B,A) of the form:

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i p_i, \\ p_{i+1} &= Ap_i - \sum_{j=0}^{i} \sigma_{kj} p_j. \end{aligned}$$

While this method is guaranteed to converge in $d(r_0, A)$ steps, it may converge rather slowly, depending on certain properties of A. In Chapter 4, we will explore ways of improving the convergence rate with *preconditioning*. The idea is this: rather than solve $Ax = b$, we instead solve the

equivalent preconditioned system

$$QAP\tilde{x} = Qb, \qquad P\tilde{x} = x,$$

where Q and P are nonsingular matrices. To solve this system, we apply the method CG($\tilde{B}$,$\tilde{A}$) for $\tilde{A} = QAP$ and some hpd $\tilde{B}$. The CG method above is now in terms of the quantities $\tilde{A}$, $e_i = P\tilde{e_i}$, $x_i = P\tilde{x_i}$, $r_i = Q^{-1}\tilde{r_i}$, and $p_i = P\tilde{p_i}$. If we define $B = P^{-*}\tilde{B}P^{-1}$ and $C = PQ$, we can rewrite the method in terms of the unscaled quantities:

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i p_i, & \alpha_i &= \frac{\langle Be_i, p_i\rangle}{\langle Bp_i, p_i\rangle}, \\ p_{i+1} &= CAp_i - \sum_{j=0}^{i} \sigma_{kj} p_j, & \sigma_{kj} &= \frac{\langle BCAp_k, p_j\rangle}{\langle Bp_j, p_j\rangle}. \end{aligned}$$

This method minimizes $\|e_{i+1}\|_B$ over $x_0 + V_{i+1}(Cr_0, CA)$, and we call it CG(B,C,A). It is equivalent to CG(B,CA), which implies that only left preconditioning need be considered, as right preconditioning can be absorbed into a composite left preconditioner and into the inner product [6]. We have followed this approach in CgCode, and consider only left preconditioning in the implementations.

3.1 Economical CG Methods

If all previous direction vectors $\{p_k\}_{k=0}^{i-1}$ were needed to build p_i, CG(B,C,A) would be prohibitively expensive. However, under certain conditions *economical* methods are possible, in which only a few past direction vectors are needed. Specifically, we are interested in methods that require only a *3-term recursion* for the direction vectors $\{p_k\}$:

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i p_i, & \alpha_i &= \frac{\langle Be_i, p_i\rangle}{\langle Bp_i, p_i\rangle}, \\ p_{i+1} &= CAp_i - \gamma_i p_i - \sigma_i p_{i-1}, & \gamma_i &= \frac{\langle BCAp_i, p_i\rangle}{\langle Bp_i, p_i\rangle}, \qquad \sigma_i = \frac{\langle BCAp_i, p_{i-1}\rangle}{\langle Bp_{i-1}, p_{i-1}\rangle}. \end{aligned}$$

To understand the conditions under which these 3-term methods will converge, we need a few definitions. Let A^+ be the *B-adjoint* of A, which is the unique matrix satisfying

$$\langle BAx, y\rangle = \langle Bx, A^+ y\rangle$$

for every x and y. This implies that $A^+ = (BAB^{-1})^* = B^{-1}A^*B$, where $A^* = \bar{A}^T$ is the adjoint of A in the standard inner product. A is called *self-adjoint* if $A^* = A$, and A is called *B-self-adjoint*

if $A^+ = A$. From this it is easy to see that A is B-self-adjoint if and only if BA is Hermitian. A is called *normal* if $AA^* = A^*A$, and A is called B-*normal* if $AA^+ = A^+A$. From [16] we know that $AA^+ = A^+A$ if and only if A^+ is a polynomial in A. If the polynomial of minimal degree has degree s, then we say A is B-normal(s). Therefore, if $A^+ = c_0 I + c_1 A$, we say that A is B-normal(1). It is easily shown [16] that this implies A is the translation and rotation of a B-self-adjoint matrix G:

$$A = e^{i\theta}\left(\frac{ir}{2}I + G\right), \quad r \geq 0, \quad 0 \leq \theta \leq 2\pi, \quad i = \sqrt{-1}.$$

Moreover, A is B-normal(1) for some B if and only if A is diagonalizable and the spectrum of A lies on a line segment in the complex plane [6]. We can now state the following theorem, which gives necessary and sufficient conditions for the convergence of a 3-term conjugate gradient method, CG(B,C,A).

Theorem 3.1.1 [16] *If B is hpd, then the 3-term conjugate gradient method CG(B,C,A) converges to $x = A^{-1}b$ in at most $d(Cr_0, CA)$ steps for every x_0 if and only if $d(CA) \leq 3$ or CA is B-normal(1).*

Given the linear system $Ax = b$, one must choose B and C so that CA is B-normal(1), in order to use a 3-term CG method. Therefore, preconditioning can be viewed as a way of either improving the condition number of A, or as a way to guarantee the convergence of a 3-term CG method. Solving the normal equations, $A^*Ax = A^*b$, is an example of choosing the preconditioner to guarantee convergence of the method.

We remark that any Hermitian or skew-Hermitian matrix is I-normal(1). For such matrices, the conjugate residual (CR) method, CG(A^2,I,A), may be used. Two algorithms implementing this method are included in CgCode.

3.2 The algorithms Odir and Omin

We now present two *algorithms* for implementing the method CG(B,C,A). The first of these is called Odir(B,C,A):

$$\begin{aligned} p_0 &= Cr_0 \\ \alpha_i &= \frac{\langle Be_i, p_i\rangle}{\langle Bp_i, p_i\rangle} \\ x_{i+1} &= x_i + \alpha_i p_i \end{aligned}$$

$$
\begin{aligned}
r_{i+1} &= r_i - \alpha_i A p_i \\
\gamma_i &= \frac{\langle BCAp_i, p_i \rangle}{\langle Bp_i, p_i \rangle} \\
\sigma_i &= \frac{\langle BCAp_i, p_{i-1} \rangle}{\langle Bp_{i-1}, p_{i-1} \rangle} \\
p_{i+1} &= CAp_i - \gamma_i p_i - \sigma_i p_{i-1}.
\end{aligned}
$$

If CA is B-normal(1) for some hpd inner product matrix B, or if $d(CA) \leq 3$, then the above iteration is guaranteed to converge. Moreover, $\|e_{i+1}\|_B$ is minimized over $x_0 + V_{i+1}(Cr_0, CA)$. Note that we have introduced the new quantity r_i, which is often used in stopping criteria (Chapter 5).

Let us now define the preconditioned residual $s_i = Cr_i$. When BCA is definite [6], the preconditioned residuals $\{s_j\}_{j=0}^{i}$ span V_{i+1}. When this occurs, a more efficient algorithm for the method CG(B,C,A) can be formulated. It is called Omin(B,C,A):

$$
\begin{aligned}
\hat{p}_0 &= Cr_0 \\
\hat{\alpha}_i &= \frac{\langle Be_i, p_i \rangle}{\langle Bp_i, p_i \rangle} \\
x_{i+1} &= x_i + \hat{\alpha}_i \hat{p}_i \\
r_{i+1} &= r_i - \hat{\alpha}_i A \hat{p}_i \\
s_{i+1} &= Cr_{i+1} \\
\beta_i &= -\frac{\langle BCAe_{i+1}, \hat{p}_i \rangle}{\langle B\hat{p}_i, \hat{p}_i \rangle} \\
\hat{p}_{i+1} &= s_{i+1} + \beta_i \hat{p}_i.
\end{aligned}
$$

It can be shown that $\hat{p}_i = c_i p_i$, where p_i is the direction vector from the Odir algorithm, and c_i is a scalar. Also note that Omin(A,I,A) is the classical conjugate gradient algorithm of Hestenes and Stiefel [19].

If $\hat{\alpha}_i = 0$ for some i, which is possible if BCA is indefinite [6], Omin may fail to converge. Therefore, the less expensive Omin(B,C,A) is also less robust than Odir(B,C,A). The following theorem, analogous to the earlier theorem for Odir(B,C,A), establishes necessary and sufficient conditions for $\hat{\alpha}_i \neq 0$ for all i, and hence gives necessary and sufficient conditions for convergence of Omin(B,C,A).

Theorem 3.2.1 [6] *If B is hpd, then Omin(B,C,A) converges to $x = A^{-1}b$ for every x_0 if and only if BCA is definite and either CA is B-normal(1) or $d(CA) \leq 3$.*

Although Omin(B,C,A) may converge for a given x_0 if BCA is indefinite, it is nevertheless necessary to monitor $\hat{\alpha}_i$ to ensure $\hat{\alpha}_i \neq 0$. If $\hat{\alpha}_i = 0$ for some i, then temporarily switching to Odir(B,C,A) will lift the iteration into the new space V_{i+1}, and the less expensive Omin(B,C,A) can be resumed. Such a *hybrid* Odir/Omin algorithm for $B = A^2$ can be found in [10].

The algorithms Omin(B,C,A) and Odir(B,C,A) are called Orthomin and Orthodir, respectively, by Young et al. [20, 22]. We have again followed [6] and used the names Omin and Odir for brevity. For a thorough discussion of the properties of these algorithms, as well as a third, see [6].

3.3 CG Methods Implemented in CgCode

CgCode provides algorithms for solving Hermitian and nonhermitian linear systems. Table 3.1, based on the CG(B,C,A) notation, summarizes the methods in CgCode, the algorithm chosen to implement each method, and any restrictions on the matrix A. To choose the appropriate algorithm for one's linear system, first determine the properties of the system matrix: Hermitian or nonhermitian, definite or indefinite. Once this has been done, the appropriate algorithm(s) in the table can be selected from the *restrictions* column. For details on the appropriate parameter settings to use CgCode, see Chapter 2.

ICG	Method	CG(B,C,A) Notation	Algorithm	Restrictions	Comments
1	CGHS	CG(A,I,A)	Omin	A hpd	CGHS on A
2	CR	CG(A^2,I,A)	Omin	A hpd	CR on A
3	CRIND	CG(A^2,I,A)	Odir/Omin	A Hermitian	CR on A
4	PCG	CG(A,C,A)	Omin	A,C hpd	PCG on A
5	CGNR	CG(A^*A,A^*,A)	Omin	None	CGHS on A^*A
6	CGNE	CG(I,A^*,A)	Omin	None	CGHS on AA^*
7	PCGNR	CG(A^*A,CC^*A^*,A)	Omin	None	CGNR on AC
8	PCGNE	CG(I,A^*CC^*,A)	Omin	None	CGNE on CA
9	PPCG	CG(A,$C(A)$,A)	Omin	A,C hpd	Polynomial PCG on A
10	PCGCA	CG($C(A)A$,$C(A)$,A)	Omin	A,C hpd	CGHS on $C(A)A$

Table 3.1: The methods implemented in CgCode

In Table 3.1, CGHS refers to the classical conjugate gradient algorithm of Hestenes and Stiefel. CR and CRIND are the Omin and Odir/Omin implementations of the conjugate residual method for hpd and Hermitian systems, respectively. PCG is the preconditioned conjugate gradient method

for hpd systems. CGNR and CGNE implement the conjugate gradient method on two different forms of the normal equations: (1) $A^*Ax = A^*b$, and (2) $AA^*y = b$, $x = Ay$. PCGNR and PCGNE are preconditioned versions of the previous two algorithms. PPCG is the standard preconditioned conjugate gradient algorithm using adaptive Chebyshev polynomial preconditioning. Finally, PCGCA is the classical CGHS algorithm applied to the polynomial preconditioned system $C(A)Ax = C(A)b$. For details about some of these algorithms, see [6, 15].

In this chapter we have summarized the taxonomy presented in [6]. We have introduced CG methods, presented two algorithms for implementing these methods, and given conditions guaranteeing convergence of the algorithms. We have also listed the methods and algorithms in CgCode. For a rigorous discussion of CG methods, see [6, 16]. An introduction to preconditioning CG methods follows in Chapter 4, and a discussion of stopping criteria for these methods appears in Chapter 5.

Chapter 4

Preconditioning

In this chapter, we review a few standard preconditioning techniques. But first, let us consider why preconditioning is needed in the conjugate gradient method. If the matrix A is Hermitian positive definite (hpd), and the classical conjugate gradient method is being used, then the following inequality from [23] gives a bound on the error at step k:

$$\|e_k\|_A \leq \min_{\lambda_i}[1 + \lambda_i P_{k-1}(\lambda_i)]^2 \|e_0\|_A \tag{4.1}$$

where P_{k-1} is any polynomial of degree $k-1$, and where the maximum above is taken over all eigenvalues λ_i of A. If the Chebyshev polynomial of degree $k-1$ is selected for P_{k-1}, then the following inequality [17] can be derived from (4.1), where $\kappa(A)$ is the condition number of A:

$$\|e_k\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|e_0\|_A. \tag{4.2}$$

We can also use (4.1) to obtain an upper bound on the number of iterations k required to reduce the relative error below a tolerance ϵ [15]:

$$k = \frac{1}{2}\ln\frac{2}{\epsilon}\sqrt{\kappa(A)}. \tag{4.3}$$

Therefore, if $\kappa(A)$ is large, the error reduction at each step may be poor, and the number of iterations required to reach the desired error tolerance may be large.

To decrease the number of iterations required for convergence, one uses preconditioning. Instead of solving the system $Ax = b$, we instead solve the equivalent system

$$QAP(P^{-1}x) = Qb.$$

If $\kappa(QAP) \ll \kappa(A)$, then the conjugate gradient method should converge faster for $QAP(P^{-1}x) = Qb$ than for $Ax = b$.

In the preconditioned system, the matrix Q is call a *left preconditioner* and the matrix P is called a *right preconditioner*. In the previous chapter, we remarked that only left preconditioning need be considered in CG methods because right preconditioning can be absorbed by a composite left preconditioner and the inner product. Therefore, given A such that $\kappa(A) \gg 1$, we seek C such that $\kappa(CA) \approx 1$, or at least $\kappa(CA) \ll \kappa(A)$. Since $\kappa(I) = 1$, this implies that if C approximates A^{-1}, then C should be a good preconditioner for A.

It should be noted, however, that the condition number of the matrix is really just an *indicator* of the rate of convergence. The *distribution* of the eigenvalues of A is far more important than $\kappa(A)$. It is well-known that the conjugate gradient method performs best if the eigenvalues of A are clustered. Thus, we want to choose C such that the preconditioned matrix CA has clustered eigenvalues. It is often (but not necessarily) the case that this choice of C reduces $\kappa(A)$. For a thorough analysis of the relationship between the clustering of eigenvalues and the rate of convergence of a CG method, see [8, 9, 34].

It is important to note that the matrix CA is never formed; rather, C is applied to a vector at an appropriate point in the algorithm. This is necessary because while A may be sparse, CA may be dense. Therefore, for the preconditioner to be practical, the cost of constructing C and applying it to a vector must be less than the cost of the number of iterations saved by preconditioning. For a detailed discussion of some of these issues, see [12, 14, 23, 27].

Now that we know what properties C must have to be an effective and practical preconditioner, we examine several well-known and widely-used preconditioners.

4.1 Diagonal and Block Diagonal Preconditioning

One simple, easy to implement, and often effective preconditioning is diagonal scaling. With this technique, the preconditioner C is taken to be a diagonal matrix. One obvious choice for C is the inverse of the diagonal of the matrix A. If the matrix A has property $\mathcal{A}$ [37], then this choice of C is the optimal diagonal scaling [28]. This scaling can be done prior to entering the CG iteration, which will reduce the number of multiplications by 1 for each row of the matrix during matrix-vector multiplication.

Diagonal preconditioning, also known as Jacobi preconditioning, is a special case of block diagonal preconditioning. When elliptic partial differential equations are discretized, it is common for the resulting linear system to have block tridiagonal form:

$$A = \begin{bmatrix} A_1 & B_1 & & & & & \\ C_2 & A_2 & B_2 & & & & \\ & C_3 & A_3 & B_3 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \end{bmatrix}. \tag{4.4}$$

If this is the case, a block diagonal preconditioner C would be of the form

$$C = \begin{bmatrix} A_1^{-1} & & & & & & \\ & A_2^{-1} & & & & & \\ & & A_3^{-1} & & & & \\ & & & \cdot & & & \\ & & & & \cdot & & \\ & & & & & \cdot & \end{bmatrix}. \tag{4.5}$$

Of course, one would not form the A_i^{-1} explicitly, but might instead work with their LU factorizations. If A is hpd, then $C_i = B_{i-1}^*$, in which case various block preconditioners can be implemented. An excellent study of block preconditioning for conjugate gradient methods is presented in [11].

4.2 Incomplete Factorizations

Another popular and effective preconditioning technique is based on the use of an *incomplete factorization* of the hpd matrix A. Recall that the standard *Cholesky* factorization of A,

$$A = LL^*,$$

may produce a dense lower triangular factor L as a result of fill-in during the factorization process, even if A is sparse. If we attempt to suppress this fill-in, then we are led to an *incomplete* Cholesky factorization:

$$A \approx LL^*,$$

where L is sparse. One technique for producing an incomplete Cholesky factorization is called the *no-fill factorization*, or IC(0). In this method, the sparsity structure of L is forced to follow that of A. (The factorizations known as IC(k), $k > 0$, are techniques for allowing more fill-in than IC(0) in order to construct better approximations to the true Cholesky factor.)

Once the approximate factorization has been obtained, the preconditioning matrix will be given by $C = (LL^*)^{-1}$ where L is the nonsingular incomplete Cholesky factor. Applying the preconditioner to a vector in a conjugate gradient algorithm requires forward and backward substitution rather than explicit inversion of L. Specifically, the preconditioner-vector product $w = Cv$ is implemented as two triangular solves: $Lz = v$ and $L^*w = z$. Of course, the factorization is computed once at the beginning of the iteration. *Block Cholesky factorizations* can be used as preconditioners as well [11]. For more information on incomplete factorizations see [26, 27, 30].

Unfortunately, there are difficulties with this technique. In particular, incomplete factorizations are difficult to vectorize. Both the forward and backward solves are highly recursive, and become a bottleneck for parallelization and vectorization of preconditioned CG algorithms. Much work has been done in the area of producing factorizations that have more satisfactory vectorization properties [27]. Incomplete factorizations are also data structure-dependent: if the data structure changes, then the factorization routine must be recoded to reflect the new format. A third difficulty arises when the matrix A is real. In this case, the incomplete Cholesky factorization of the symmetric positive definite matrix A may break down if the square root of a negative number is required. If A is a symmetric M-matrix the factorization is guaranteed to exist [26, 27]. Otherwise, other techniques are available to produce an incomplete factorization by avoiding the negative square root [30]. Also, by monitoring the factorization process, modifications of some elements may be made to insure that L will be nonsingular.

4.3 Polynomial Preconditioning

This section examines polynomial preconditioning for hpd linear systems, an idea that has been around for a while [31]. Here the preconditioner C is a polynomial in A, written $C(A)$. One chooses $C(A)$ using the same criteria as for other preconditioners: it should approximate the inverse of the system matrix A, and be easy to apply to a vector. The polynomial $C(\lambda)$ is referred to as the *preconditioning polynomial*, and $C(A)$ is the associated *polynomial preconditioner* [3, 5]. Since A is hpd, so are $C(A)$ and $C(A)A$; this property can be used to devise new conjugate gradient methods. (See the previous chapter, [3], and [6] for a discussion of the properties required of A and $C(A)$ to guarantee convergence of a 3-term CG method.)

There are several advantages to polynomial preconditioners. First, they can be implemented automatically: the user simply initializes a few parameters and provides the matrix-vector multiplication routine. Second, and most important, polynomial preconditioning is well-suited for parallel and/or vector architectures. Since the two intrinsic operations in polynomial preconditioning are matrix-vector multiplication and vector addition, these operations must be efficient on a given architecture for polynomial preconditioning to be effective [14]. The key to performance is a fast matrix-vector multiplication routine.

If C is to approximate A^{-1}, then $C(\lambda)$ should approximate λ^{-1} in some way. A simple choice is based on the *Neumann series* [1, 14]

$$A^{-1} = (M - N)^{-1} = (I + M^{-1}N + (M^{-1}N)^2 + (M^{-1}N)^3 + \cdots)M^{-1}.$$

This series converges if $\rho(M^{-1}N) < 1$. If we truncate the series, we obtain our polynomial preconditioner. However, this is not the most effective polynomial preconditioner. Moreover, the optimal degree seems to be 2 [1, 14].

To obtain a better preconditioner, one might consider the polynomial $C(\lambda)$ satisfying

$$\min_{C \in \pi_{m-1}} \|1 - C(\lambda)\lambda\|, \tag{4.6}$$

where π_{m-1} is the set of all polynomial of degree less than or equal to $m-1$. In general, the norm will depend on some set S containing the spectrum $\sigma(A)$ of A. Note that since A is hpd, we know $\sigma(A) \subset S = [c, d]$, $0 < c \leq d$. Ideally, c and d are the minimum and maximum eigenvalues of A.

If we solve the minimization problem (4.6) using the weighted least squares norm $\|f\|_\omega$, we obtain the *least squares preconditioned polynomial*. This norm is induced by the inner product $\langle f, g\rangle_\omega$, given by

$$\langle f, g\rangle = \int_c^d f(\lambda)\overline{g(\lambda)}\omega(\lambda)d\lambda,$$

for some positive weight function $\omega(\lambda)$ defined on S. It can be shown that the least squares polynomial can be computed using a 3-term recursion [5], yielding an efficient and stable procedure for generating these polynomials. Least squares preconditioning polynomials can exploit the eigenvalue distribution of A (if it is known) by appropriately choosing the weight function ω. For a thorough discussion of least squares polynomial preconditioning, see [5, 21, 32].

If the minimization problem (4.6) is solved in the uniform norm

$$\|f\|_\infty = \max_{\lambda \in S} |f(\lambda)|,$$

then the solution is obtained from a shifted and scaled Chebyshev polynomial [5]:

$$C(\lambda)\lambda = 1 - \frac{T_m(\frac{d+c-2\lambda}{d-c})}{T_m(\frac{d+c}{d-c})}, \tag{4.7}$$

where T_m is the Chebyshev polynomial of degree m. It can also be computed via a 3-term recursion. The Chebyshev polynomials have several other interesting properties, including the *minimax* property, which is a result of their equioscillation behavior [13]: of all m-th degree polynomials with leading coefficient 1, the polynomial $2^{1-m}T_m$ has the smallest maximum norm in the interval $[-1, 1]$. It can be shown that (4.7) is optimal in the sense that it minimizes a bound on the condition number of the preconditioned matrix [5, 21]:

$$\kappa(C(A)A) \leq \frac{1 + T_m^{-1}(\frac{d+c}{d-c})}{1 - T_m^{-1}(\frac{d+c}{d-c})}.$$

See [5] for a comparison of least squares and Chebyshev preconditioning polynomials.

Adaptive Chebyshev polynomial preconditioning is provided in CgCode. Two conjugate gradient algorithms with polynomial preconditioning are provided in CgCode: PPCG and PCGCA. The method PPCG is the standard preconditioned conjugate gradient method with $C(A)$ as the preconditioner; PCGCA is the classical CGHS algorithm applied to the system $C(A)Ax = C(A)b$. To use these methods, the user must provide the MATVEC routine and initialize a few parameters, such as initial eigenvalue estimates and the degree of the preconditioning polynomial. For a detailed description of how to use PPCG and PCGCA in CgCode, including suggestions for initial parameter settings, see Chapter 2.

We note that adaptive Chebyshev polynomial preconditioning has been implemented on parallel and vector machines. For a discussion of their performance on the Cray X-MP/48 and the Alliant FX/8, see [3, 5, 29]. Experiments indicate that the optimal degree for the preconditioning polynomial varies between 2 and 16, with the optimal degree increasing as the condition number of the system matrix increases. The performance of polynomial preconditioning is architecture-dependent: a small speedup over no preconditioning was evident on the Cray X-MP/48, while substantial speedups were observed on both the Alliant FX/8 and the Cray 2.

Chapter 5

Stopping Criteria

In any iterative method one must decide when to halt the iteration. For most methods, it is natural to stop when the error in some norm is below a specified tolerance

$$\|e_i\|_M \leq \epsilon,$$

where M is hpd and $\epsilon > 0$. It may be more desirable to stop when the relative error in this norm is sufficiently small:

$$\frac{\|e_i\|_M}{\|x\|_M} \leq \epsilon.$$

In either case, the choice of M will determine whether the stopping criterion is computable, and if so, whether it is efficient to implement. Ideally, M should be chosen so that quantities already present in the iteration are used to evaluate the stopping criterion, rather than requiring the computation of additional inner products and/or norms.

For example, if $M = A^*A$, then the norm used is called the *residual norm*, and the stopping criteria above become

$$\|e_i\|_M = \|r_i\| \leq \epsilon, \tag{5.1}$$

and

$$\frac{\|e_i\|_M}{\|x\|_M} = \frac{\|r_i\|}{\|b\|} \leq \epsilon, \tag{5.2}$$

where $\|\cdot\|$ denotes the Euclidean norm, $\|\cdot\|_2$. Note that r_i and $\|r_i\|$ are available in the classical conjugate gradient method CGHS. If $M = (CA)^*(CA)$, the *preconditioned residual norm* results, and two stopping criteria are

$$\|e_i\|_M = \|s_i\| \leq \epsilon, \tag{5.3}$$

and

$$\frac{\|e_i\|_M}{\|x\|_M} = \frac{\|s_i\|}{\|Cb\|} \leq \epsilon, \tag{5.4}$$

where $s_i = Cr_i$ is the *preconditioned residual*. Note that for PCG, s_i is available, but not its norm.

Different methods may use any of the above criteria to provide the most efficient stopping test. However, it has been proposed [7] that iterative linear solvers provide a *menu* of stopping criteria, consisting of the four tests above, plus an efficient default stopping criterion, as well as any additional stopping criteria. This would make it easier to compare various methods. The proposed standard mandates a flag ISTOP which indicates the stopping criterion to be used:

ISTOP = 0	efficient default stopping criterion
ISTOP = 1	stopping criterion (5.1)
ISTOP = 2	stopping criterion (5.2)
ISTOP = 3	stopping criterion (5.3)
ISTOP = 4	stopping criterion (5.4).

See [7] for more detailed information about the proposed standard and the stopping criteria menu. Note that CgCode conforms to the proposed standard, and provides the full menu of stopping criteria above for each of the ten methods in the package. In addition, CgCode uses condition number estimates of the system matrix to provide, for each of the ten conjugate gradient methods, a stopping criterion based on the true error e_i. This is discussed in the following section.

5.1 Stopping Criteria Based on the True Error

In many situations, it may be desirable to stop when the relative error has been bounded. That is, one may wish to halt when

$$\frac{\|e_i\|_M}{\|x\|_M} = \frac{\|x - x_i\|_M}{\|x\|_M} \leq \epsilon, \tag{5.5}$$

for some hpd M. Although the left-hand side of (5.5) cannot be evaluated, it can be bounded. To see how, let us first derive a useful inequality. If we take norms of both sides of the equation $CAx = Cb$, we obtain

$$\|x\|_M \geq \frac{\|Cb\|_M}{\|CA\|_M}. \tag{5.6}$$

Another useful inequality involves the error e_i:

$$e_i = x - x_i = (CA)^{-1}Cb - x_i = (CA)^{-1}(Cb - CAx_i) = (CA)^{-1}s_i,$$

which yields

$$\|e_i\|_M \leq \|(CA)^{-1}\|_M \|s_i\|_M. \tag{5.7}$$

Combining the inequalities (5.5), (5.6), and (5.7) above, we have:

$$\frac{\|e_i\|_M}{\|x\|_M} \leq \frac{\|(CA)^{-1}\|_M \|s_i\|_M}{\|x\|_M} \leq \frac{\|CA\|_M \|(CA)^{-1}\|_M \|s_i\|_M}{\|Cb\|_M},$$

or

$$\frac{\|e_i\|_M}{\|x\|_M} \leq \kappa_M(CA) \frac{\|s_i\|_M}{\|Cb\|_M}. \tag{5.8}$$

Therefore, if we have an estimate of $\kappa_M(CA)$, then we can approximate the stopping criterion given in (5.5) by

$$\kappa_M(CA) \frac{\|s_i\|_M}{\|Cb\|_M} \leq \epsilon. \tag{5.9}$$

This attempts to halt when the relative error is below a certain tolerance. While $\kappa_M(CA)$ is usually not known a priori, it can be shown [6] that $\kappa_B(CA)$ can be estimated *dynamically* (see below).

5.2 Default Stopping Criteria in CgCode

Let us now turn to the stopping criteria provided by CgCode, which are the following:

if ISTOP=0	then use:	Inequality (5.9)	$\leq \epsilon$ (DEFAULT)
if ISTOP=1	then use:	$\|r_i\|$	$\leq \epsilon$
if ISTOP=2	then use:	$\|r_i\|/\|b\|$	$\leq \epsilon$
if ISTOP=3	then use:	$\|Cr_i\|$	$\leq \epsilon$
if ISTOP=4	then use:	$\|Cr_i\|/\|Cb\|$	$\leq \epsilon$.

Each algorithm in CgCode attempts to use as a default stopping criterion

$$\frac{\|e_i\|}{\|x\|} \leq \epsilon,$$

by finding an estimate of $\kappa(CA) = \kappa_I(CA)$ and using inequality (5.9). If CA is definite, then $\kappa_B(CA)$ can be dynamically estimated from eigenvalue estimates available from the CG iteration itself, where B is the inner product matrix for the method. If CA is also Hermitian, then $\kappa(CA) = \kappa_B(CA)$, which is the quantity needed in (5.9). If CA is nonhermitian, then $\kappa_B(CA)$ and $\kappa(CA)$ need not be the same. However, in many cases CA is similar to a Hermitian matrix, and hence has real eigenvalues. As a result, inequalities involving $\kappa_B(CA)$ may be used to approximate (5.9). If CA is indefinite, it is impossible to estimate $\kappa_B(CA)$ from the iteration parameters. In this case, the user must provide an estimate of $\kappa(CA)$ if he wishes to use (5.9). See [6] for details.

Method	Default stopping criterion
CGHS	$\kappa(A)\ \frac{\|r_i\|}{\|b\|} \leq \epsilon$
CR	$\kappa(A)\ \frac{\|r_i\|}{\|b\|} \leq \epsilon$
CRIND	$\kappa(A)\ \frac{\|r_i\|}{\|b\|} \leq \epsilon$
PCG	$\kappa(CA)\ \frac{\|Cr_i\|}{\|Cb\|} \leq \epsilon$
CGNR	$\kappa(A^*A)\ \frac{\|A^*r_i\|}{\|A^*b\|} \leq \epsilon$
CGNE	$\kappa(A)\ \frac{\|r_i\|}{\|b\|} \leq \epsilon$
PCGNR	$\kappa(C^*CA^*A)\ \frac{\|CC^*A^*r_i\|}{\|CC^*A^*b\|} \leq \epsilon$
PCGNE	$\kappa(CA)\ \frac{\|Cr_i\|}{\|Cb\|} \leq \epsilon$
PPCG	$\kappa(C(A)A)\ \frac{\langle C(A)r_i, r_i\rangle^{\frac{1}{2}}}{\langle C(A)b, b\rangle^{\frac{1}{2}}} \leq \epsilon$
PCGCA	$\kappa(C(A)A)\ \frac{\|C(A)r_i\|}{\|C(A)b\|} \leq \epsilon$

Table 5.1: Default Stopping Criteria in CgCode

Table 5.1 lists the default stopping criteria, and the inequalities related to (5.9), for each of the algorithms in CgCode. Each inequality is derived in an attempt to approximate (5.5). Note that in PCG and PPCG, $\|Cr_i\|$ and $\|Cb\|$ are not available; instead $\langle Cr_i, r_i\rangle^{\frac{1}{2}}$ and $\langle Cb, b\rangle^{\frac{1}{2}}$ are available. Therefore, inequalities involving the available quantities were derived in order to implement an efficient stopping test. See [6] for a discussion of the inequalities and techniques used to implement a stopping criterion based on the true error.

In this chapter, we have discussed stopping criteria in iterative methods, techniques for approximating a stopping criteria based on the true error, and the default stopping criteria in CgCode. See Chapter 3 for a description of the CG methods listed in Table (5.1). For information on obtaining a copy of the CgCode software, turn to the next chapter.

Chapter 6

Future Directions

It is our hope that you will find CgCode both useful and easy to use. We believe that the variety of methods implemented in CgCode, and the simplicity and flexibility of the user interface, have combined to create a powerful tool for the solution of large and sparse linear systems arising in scientific applications. In Chapter 2 of this report, we have described how to use CgCode. Chapters 3, 4, and 5 provided introductions to conjugate gradient methods, preconditioning, and stopping criteria. After reading this material, you should have a basic understanding of the methods employed in CgCode, and perhaps a better understanding of how to solve your linear system.

6.1 Future Releases of CgCode

Future releases of CgCode are now in the planning stages, and these will incorporate the following:

- the addition of several new methods including PCR,
- a menu of data structures for the system matrix and their corresponding matvecs,
- a menu of preconditioners,
- extensive error checking,
- extension of adaptive polynomial preconditioning to Hermitian indefinite problems.

6.2 How to Get a Copy of CgCode

If you would like a copy of the CgCode software package, or if you have comments, suggestions, or questions about CgCode, please send electronic mail to:

cgwiz@martini.cs.uiuc.edu

Bibliography

[1] L. M. Adams, *m-step preconditioned conjugate gradient methods*, SIAM J. Sci. Stat. Comput., 6 (1985), pp. 452–463.

[2] S. F. Ashby, *CHEBYCODE: A FORTRAN implementation of Manteuffel's adaptive Chebyshev algorithm*, Master's thesis, Dept. of Computer Science, University of Illinois, Urbana, May 1985. Available as Technical Report 1203.

[3] ——, *Polynomial Preconditioning for Conjugate Gradient Methods*, PhD thesis, Dept. of Computer Science, University of Illinois, Urbana, December 1987. Available as Technical Report 1355.

[4] ——, *Private communication*, 1990.

[5] S. F. Ashby, T. A. Manteuffel, and J. S. Otto, *Adaptive polynomial preconditioning for HPD linear systems*, in Proceedings of the Ninth International Conference on Computing Methods in Applied Sciences and Engineering, 1990.

[6] S. F. Ashby, T. A. Manteuffel, and P. E. Saylor, *A taxonomy for conjugate gradient methods*, Tech. Rep. UCRL-98508, Lawrence Livermore National Laboratory, March 1988. To appear in SIAM J. Numer. Anal.

[7] S. F. Ashby and M. K. Seager, *A proposed standard for iterative linear solvers (version 1.0)*, Tech. Rep. UCRL-102860, Lawrence Livermore National Laboratory, 1990.

[8] O. Axelsson, *A class of iterative methods for finite element equations*, Comput. Methods Appl. Mech. Eng., 9 (1976), pp. 123–137.

[9] O. Axelsson and V. Barker, *Finite Element Solution of Boundary Value Problems*, Academic Press, Orlando, FL, 1984.

[10] R. Chandra, S. C. Eisenstat, and M. H. Schultz, *The modified conjugate residual method for partial differential equations*, in Advances in Computer Methods for Partial Differential Equations II, R. Vichnevetsky, ed., 1977, pp. 13–19.

[11] P. Concus, G. H. Golub, and G. Meurant, *Block preconditioning for the conjugate gradient method*, SIAM J. Sci. Stat. Comput., 6 (1985), pp. 220–252.

[12] P. Concus, G. H. Golub, and D. P. O'Leary, *A generalized 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, NY, 1976, pp. 309–332.

[13] G. Dahlquist and A. Bjorck, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[14] P. F. Dubois, A. Greenbaum, and G. H. Rodrigue, *Approximating the inverse of a matrix for use on iterative algorithms on vector processors*, Computing, 22 (1979), pp. 257–268.

[15] H. C. Elman, *Iterative Methods for Large, Sparse Nonsymmetric Systems of Linear Equations*, PhD thesis, Dept. of Computer Science, Yale University, 1982. Available as Technical Report 229.

[16] V. Faber and T. A. Manteuffel, *Necessary and sufficient conditions for the existence of a conjugate gradient method*, SIAM J. Numer. Anal., 21 (1984), pp. 352–362.

[17] G. H. Golub and C. F. V. Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 1989.

[18] G. H. Golub and D. P. O'Leary, *Some history of the conjugate gradient and Lanczos algorithms: 1948–1976*, Tech. Rep. CS-TR-1859, Computer Science Dept., University of Maryland, 1987.

[19] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Research of NBS, 49 (1952), pp. 409–435.

[20] K. C. Jea and D. M. Young, *On the simplification of generalized conjugate-gradient methods for nonsymmetrizable linear systems*, Linear Algebra Appl., 52/53 (1983), pp. 399–417.

[21] O. G. Johnson, C. A. Micchelli, and G. Paul, *Polynomial preconditioning for conjugate gradient calculations*, SIAM J. Numer. Anal., 20 (1983), pp. 362–376.

[22] W. D. Joubert and D. M. Young, *Necessary and sufficient conditions for the simplification of generalized conjugate gradient algorithms*, Tech. Rep. CNA-204, Center for Numerical Analysis, University of Texas, Austin, Austin, TX, 1986.

[23] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA, 1973.

[24] T. A. Manteuffel, *The Tchebyshev iteration for nonsymmetric linear systems*, Numer. Math., 28 (1977), pp. 307–327.

[25] ——, *Adaptive procedure for estimating parameters for the nonsymmetric Tchebyshev iteration*, Numer. Math., 31 (1978), pp. 183–208.

[26] ——, *An incomplete factorization technique for positive definite linear systems*, Math. Comp., 34 (1980), pp. 473–497.

[27] J. A. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[28] ——, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems*, J. Comp. Phys., 44 (1981), pp. 134–155.

[29] P. D. Meyer, A. J. Valocchi, S. F. Ashby, and P. E. Saylor, *A numerical investigation of the conjugate gradient method as applied to three-dimensional groundwater flow problems in randomly heterogeneous porous media*, Water Resources Research, 25 (1989), pp. 1440–1446.

[30] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, NY, 1988.

[31] H. Rutishauser, *Theory of gradient methods*, in Mitteilungen aus dem Institut fur Angewandte Mathematik Nr. 8, 1959, pp. 24–49.

[32] Y. Saad, *Practical use of polynomial preconditionings for the conjugate gradient method*, SIAM J. Sci. Stat. Comput., 6 (1985), pp. 865–881.

[33] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869.

[34] G. W. Stewart, *The convergence of the method of conjugate gradients at isolated extreme points of the spectrum*, Numer. Math., 24 (1975), pp. 85–93.

[35] G. Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, MA, 1986.

[36] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, New York, NY, 1966.

[37] D. M. Young, *Iterative Solution of Large Linear Systems*, Academic Press, New York, NY, 1971.

Appendix

In this appendix, we illustrate the use of CgCode to solve a discretized partial differential equation.

```
      PROGRAM ILSDRV
C*********************************************************************
C
C PURPOSE:
C
C    THIS PROGRAM USES CGCODE TO SOLVE THE LINEAR SYSTEM AX=B, WHERE THE
C    SYSTEM ARISES FROM THE DISCRETIZATION OF A PARTIAL DIFFERENTIAL EQUATION.
C    CGCODE IS A PACKAGE OF FORTRAN 77 SUBROUTINES FOR THE SOLUTION OF LINEAR
C    SYSTEMS USING CONJUGATE GRADIENT METHODS.
C
C    THE INPUT PARAMETERS ARE READ FROM THE USER-PROVIDED FILE "IN".
C    THE OUTPUT FROM THIS DRIVER PROGRAM APPEARS IN THE FILE "OUT".
C    THE OUTPUT FROM THE SOLVER APPEARS IN THE FILE "OUTN".
C
C NOTE1:
C
C    THIS PROGRAM CALLS A SUBROUTINE TO SOLVE THE LINEAR SYSTEM ARISING FROM
C    THE DISCRETIZATION OF THE FOLLOWING ELLIPTIC BOUNDARY VALUE PROBLEM:
C    (POISSON'S EQUATION WITH DIRICHLET BOUNDARY CONDITIONS ON A RECTANGLE)
C
C           LU = F, U IN OMEGA
C            U = G, U ON BOUNDARY OF OMEGA
C
C    WHERE
C            OMEGA = [AX,BX]X[AY,BY]
C    AND
C            L = THE LAPLACEAN OPERATOR
C
C NOTE2:
C
C    WE DISCRETIZE THE ABOVE PROBLEM ON AN [NX BY NY] GRID, WHICH LEADS
C    TO A SPARSE SYSTEM OF LINEAR EQUATIONS OF THE FORM:
C
C            A*X = B
C
C    HERE, A IS OF ORDER N = NX*NY.
C
C PDE PROBLEM SPECIFIC INPUT FOR PDE DRIVER VIA USER SUPPLIED SUBPROGRAM
C   (1)  THE RIGHT HAND SIDE OF THE PROBLEM (CUOF(X,Y))
C   (2)  THE BOUDARY CONDITION OF THE PDE   (CUOG(X,Y))
C   (3)  ANALYTIC SOLN OF THE PDE FOR TESTS (CUOU(X,Y))
```

```
C
C INPUT FROM USER VIA UNIT IREAD:
C   (1)  LEVEL OF IO FROM SOLVER (IOLEVL)
C   (2)  PRECONDITIONING KEY (IPCOND)
C   (3)  STOPPING CRITERION KEY (ISTOP)
C   (4)  MAXIMUM NUMBER OF ITERATIONS ALLOWED (ITMAX)
C   (5)  TOLERANCE FOR THE SOLUTION (ERRTOL)
C   (6)  METHOD DEPENDENT INPUT PARAMETERS (ETC...)
C        INCLUDING:  ICYCLE,NCE,ICG,NDEG,COND,AA,BB
C        (SEE CGCODE REPORT AND DOCUMENTATION FOR DETAILED EXPLANATION)
C   (7)  WHETHER TO PRINT NUMERICAL RESULTS (KY)
C   (8)  NUMBER OF INTERIOR POINTS X DIRECTION (NX)
C   (9)  NUMBER OF INTERIOR POINTS Y DIRECTION (NY)
C   (10) LEFT ENDPOINT FOR X IN SPACE (AX)
C   (11) LEFT ENDPOINT FOR Y IN SPACE (AY)
C   (12) RIGHT ENDPOINT FOR X IN SPACE (BX)
C   (13) RIGHT ENDPOINT FOR Y IN SPACE (BY)
C
C OUTPUT FROM DRIVER:
C   (1)  THE STATISTICS CONCERNING THE RUN VIA OUTPUT TO UNIT IRITE
C   (2)  THE METHOD DEPENDENT OUTPUT MESSAGES TO UNIT IOUNIT
C   (3)  THE APPROXIMATED SOLUTION AT EACH GRID POINT VIA OUTPUT TO UNIT IOUNIT
C
C PARAMETERS:
C   IREAD           = UNIT TO READ ALL INPUT FROM
C   IRITE           = UNIT TO WRITE STATISTICS TO
C   IOUNIT          = UNIT TO WRITE NUMBERS TO
C   NGRID           = MAX DIMENSIONS IN EACH DIRECTION OF THE GRID
C   NXX             = MAX LENGTH OF ALL VECTORS
C   NONZK           = MAX NUMBER OF NONZEROS IN THE SYSTEM MATRIX
C   ETC...          = METHOD DEPENDENT PARAMETERS
C    INCLUDING: NIPAR,NRPAR,MAXICY,MAXNCE,NIWK,N1,N2,NRWK
C    (SEE CGCODE REPORT AND DOCUMENTATION FOR DETAILED EXPLANATION)
C
C VARIABLES:
C   A,IA            = DISCRETIZED OPERATOR MATRIX A
C   B               = RHS VECTOR
C   X               = SOLUTION VECTOR
C   TRUE,RHS        = TEMPORARY VECTORS USED BY DRIVER FOR STATISTICS
C   IPARAM,RPARAM   = INTEGER AND REAL PARAMETERS FOR THE SOLVER
C   IWORK,RWORK     = INTEGER AND REAL WORK ARRAYS FOR THE SOLVER
C   BEFORE,AFTER,
C   OVERHD,         = TIMING TEMPORARY VARIABLES
C   TITLE           = TITLE OF METHOD FOR OUTPUT
C   KY              = 0=NO OUTPUT,1=PRINT NUMERICAL SOLUTION
C   NX,NY           = THE NUMBER OF POINTS IN X AND Y DIRECTIONS IN GRID
C   N               = THE DIMENSION OF THE SYSTEM MATRIX, = NX*NY
C   AX,AY,BX,BY     = ENDPOINTS IN SPACE FOR X AND Y DIRECTIONS.
C   IRITE           = I/O UNIT FOR STATISTICS
C   IOUNIT          = I/O UNIT FOR NUMERICAL SOLUTION AT GRID POINTS
C   ETC...          = METHOD DEPENDENT VARIABLES
C    INCLUDING: IOLEVL,IPCOND,ISTOP,ITMAX,ERRTOL,ICYCLE,NCE,ICG,NDEG,COND,AA,BB
C    (SEE CGCODE REPORT AND DOCUMENTATION FOR DETAILED EXPLANATION)
C
C REQUIRED EXTERNAL ROUTINES:
```

```
C   MATMUL          = MATRIX-VECTOR PRODUCT ROUTINE
C   DIAGPC          = PRECONDITIONING ROUTINE
C    (SEE CGCODE REPORT AND DOCUMENTATION FOR DETAILED EXPLANATION)
C
C AUTHOR -->  MICHAEL JAY HOLST
C DATE   -->  18 MARCH 1990
C*************************************************************************
C
C     *** PARAMETERS ***
      PARAMETER    (IREAD=7,IRITE=8,IOUNIT=9)
      PARAMETER    (NGRID=63,NXX=NGRID*NGRID,NONZK=5*NXX)
      PARAMETER    (NIPAR=40,NRPAR=40)
      PARAMETER    (MAXICY=5,MAXNCE=5)
      PARAMETER    (NIWK=MAXICY*MAXNCE)
      PARAMETER    (N1=5*NXX,N2=4*MAXICY*MAXNCE+2,NRWK=N1+N2)
C
C     *** STORAGE AND EXTERNALS ***
      DIMENSION    A(NONZK+10),IA(10),Q(NXX),IQ(10)
      DIMENSION    X(NXX),TRUE(NXX),B(NXX),RHS(NXX)
      DIMENSION    IPARAM(NIPAR),RPARAM(NRPAR)
      DIMENSION    RWORK(NRWK),IWORK(NIWK)
      REAL         BEFORE,AFTER,OVERHD
      CHARACTER*50 TITLE
      EXTERNAL     MATMUL,DIAGPC
C
C     *** OPEN I/O FILES ***
      OPEN(UNIT=IREAD, FILE='in',  STATUS='UNKNOWN')
      OPEN(UNIT=IRITE, FILE='out', STATUS='UNKNOWN')
      OPEN(UNIT=IOUNIT,FILE='outn',STATUS='UNKNOWN')
      REWIND(IREAD)
      REWIND(IRITE)
      REWIND(IOUNIT)
C
C     *** SETUP SOME CGCODE PARAMETERS ***
      IPARAM(1) = NIPAR
      IPARAM(2) = NRPAR
      IPARAM(3) = NIWK
      IPARAM(4) = NRWK
      IPARAM(5) = IOUNIT
C
C     *** READ CONTROL PARAMETERS FROM USER AND CHECK STORAGE REQUIREMENT ***
      CALL SETUP (IPARAM,RPARAM,NX,NY,AX,AY,BX,BY,KY,IREAD,IRITE)
      IF ((NX .GT. NGRID) .OR. (NY .GT. NGRID)) GOTO 91
C
C     *** DISCRETIZE THE PDE TO BUILD THE LINEAR SYSTEM ***
      CALL BUILD (N,AX,AY,BX,BY,NX,NY,A,IA,B,TRUE,IRITE)
C
C     *** BUILD THE PRECONDITIONER ***
      CALL BLDPC (A,IA,Q,IQ,N)
C
C     *** SAVE RHS FOR LATER SINCE IT IS CHANGED BY CGCODE ***
      CALL SCOPY(N,B,1,RHS,1)
C
C     *** INITIAL GUESS ***
      CALL GETX0(X,N)
```

```
C
C     *** START THE TIMER ***
      CALL TSTART(BEFORE,OVERHD)
C
C     *** CALL THE ITERATIVE SOLVER ***
      CALL SCGDRV (MATMUL,DIAGPC,PCONDR,A,IA,X,B,N,Q,IQ,P,IP,
     2             IPARAM,RPARAM,IWORK,RWORK,IERROR)
C
C     *** STOP THE TIMER ***
      CALL TSTOP(AFTER)
C
C     *** CHECK FOR ERROR CONDITION ***
      IF (IERROR .NE. 0) GOTO 92
C
C     *** CALCULATE EXECUTION TIME (AND TRY TO APPROXIMATE MFLOP RATE) ***
      CPUTME = (AFTER - BEFORE) - OVERHD
C
C     *** CALCULATE THE TRUE ERROR IN APPROXIMATION ***
      CALL SAXPY(N,-1.0E0,X,1,TRUE,1)
      ERROR = SNRM2(N,TRUE,1)
C
C     *** CALCULATE THE RESIDUAL ERROR IN APPROXIMATION ***
      CALL MATMUL(0,A,IA,BDUMM,X,TRUE,N)
      CALL SAXPY(N,-1.0E0,RHS,1,TRUE,1)
      RESID = SNRM2(N,TRUE,1)
C
C     *** OUTPUT THE PARAMETER INFORMATION AND RESULT ***
      TITLE = ' SOLUTION OF POISSON EQUATION USING CGCODE '
      CALL SOUT(TITLE,KY,N,X,IPARAM,RPARAM,IERROR,CPUTME,
     2          ERROR,RESID,AX,AY,BX,BY,NX,NY,IRITE)
      GOTO 99
C
C     *** PROBLEMS ***
 91   CONTINUE
      WRITE (IRITE,*) ' NOT ENOUGH STORAGE DECLARED FOR THIS GRID '
      GOTO 99
 92   CONTINUE
      WRITE (IRITE,*) ' CGCODE RETURNED A NONZERO ERROR FLAG: ', IERROR
C
C     *** END IT ***
 99   CONTINUE
      CLOSE(IREAD)
      CLOSE(IRITE)
      CLOSE(IOUNIT)
      STOP 'ILSDRVOK'
      END
      SUBROUTINE TSTART(BEFORE,OVERHD)
C**********************************************************************
C THIS ROUTINE STARTS THE TIMER ON THE PARTICULAR MACHINE.
C**********************************************************************
      REAL      TARRAY(2),T0,BEFORE,OVERHD,GARBGE
C
C     *** FOR CONVEX: START TIMER ***
      CALL ETIME(TARRAY)
      T0 = TARRAY(1)
```

```
      CALL ETIME(TARRAY)
      OVERHD = TARRAY(1) - T0
      CALL ETIME(TARRAY)
      BEFORE = TARRAY(1)
C
C     *** FOR CRAY XMP: START TIMER ***
C     GARBGE = SECOND( )
C     T0 = SECOND( )
C     OVERHD = SECOND( ) - T0
C     BEFORE = SECOND( )
C
C     *** RETURN AND END ***
      RETURN
      END
      SUBROUTINE TSTOP(AFTER)
C*********************************************************************
C THIS ROUTINE STOPS THE TIMER ON THE PARTICULAR MACHINE.
C*********************************************************************
      REAL      TARRAY(2),AFTER
C
C     *** FOR CONVEX: STOP TIMER ***
      CALL ETIME(TARRAY)
      AFTER = TARRAY(1)
C
C     *** FOR CRAY XMP: STOP TIMER ***
C     AFTER = SECOND( )
C
C     *** RETURN AND END ***
      RETURN
      END
      SUBROUTINE SETUP (IPARAM,RPARAM,NX,NY,AX,AY,BX,BY,KY,
     2                  IREAD,IRITE)
C*********************************************************************
C THIS ROUTINE READS IN SOME INITIAL VALUES ABOUT THE PDE AND FOR THE SOLVER.
C*********************************************************************
      DIMENSION IPARAM(*),RPARAM(*)
C
C     *** INPUT THE CONTROLING PARAMETERS ***
      READ (IREAD,*) IOLEVL
      READ (IREAD,*) IPCOND
      READ (IREAD,*) ISTOP
      READ (IREAD,*) ITMAX
      READ (IREAD,*) ERRTOL
      IPARAM(6) = IOLEVL
      IPARAM(7) = IPCOND
      IPARAM(8) = ISTOP
      IPARAM(9) = ITMAX
      RPARAM(1) = ERRTOL
C
C     *** READ METHOD PARAMETERS FROM USER ***
      READ (IREAD,*)
      READ (IREAD,*) ICYCLE
      READ (IREAD,*) NCE
      READ (IREAD,*) ICG
      READ (IREAD,*)
```

```
      READ (IREAD,*) NDEG
      READ (IREAD,*) COND
      READ (IREAD,*) AA
      READ (IREAD,*) BB
      IPARAM(31) = ICYCLE
      IPARAM(32) = NCE
      IPARAM(33) = ICG
      IPARAM(34) = NDEG
      RPARAM(31) = COND
      RPARAM(32) = AA
      RPARAM(33) = BB
C
C     *** READ IN SOLUTION KEY ***
      READ (IREAD,*)
      READ (IREAD,*) KY
C
C     *** READ IN PDE PARAMETERS ***
      READ (IREAD,*)
      READ (IREAD,*) NX
      READ (IREAD,*) NY
      READ (IREAD,*) AX
      READ (IREAD,*) AY
      READ (IREAD,*) BX
      READ (IREAD,*) BY
C
C     *** RETURN AND END ***
      RETURN
      END
      SUBROUTINE GETX0(X,N)
C-----------------------------------------------------------------------
C MAKE THE INITIAL GUESS AT THE SOLUTION.
C-----------------------------------------------------------------------
      DIMENSION X(*)
      DO 10 I = 1, N
         X(I) = 0.0E0
 10   CONTINUE
      RETURN
      END
      SUBROUTINE SOUT(TITLE,KY,N,X,IPARAM,RPARAM,IERROR,CPUTME,
     2                ERROR,RESID,AX,AY,BX,BY,NX,NY,IRITE)
C-----------------------------------------------------------------------
C THIS ROUTINE PRINTS OUT THE CONTROLLING PARAMETERS, ITERATION INFORMATION,
C AND IF SPECIFIED, ALSO PRINTS OUT THE COMPUTED SOLUTION.
C-----------------------------------------------------------------------
      DIMENSION    X(*),IPARAM(*),RPARAM(*)
      CHARACTER*50 TITLE
C
C     *** DECODE IPARAM ARRAY ***
      IOUNIT = IPARAM(5)
      IOLEVL = IPARAM(6)
      IPCOND = IPARAM(7)
      ISTOP  = IPARAM(8)
      ITMAX  = IPARAM(9)
      ITERS  = IPARAM(10)
      ICYCLE = IPARAM(31)
```

```
      NCE    = IPARAM(32)
      ICG    = IPARAM(33)
      NDEG   = IPARAM(34)
C
C     *** DECODE RPARAM ARRAY ***
      ERRTOL = RPARAM(1)
      STPTST = RPARAM(2)
      CONDES = RPARAM(31)
      AA     = RPARAM(32)
      BB     = RPARAM(33)
      SCRLRS = RPARAM(34)
C
C     *** MAKE HEADER LISTING PARAMETERS FOR THE PROBLEM SOLVED ***
      ZHX = (BX-AX) / REAL(NX + 1)
      ZHY = (BY-AY) / REAL(NY + 1)
      WRITE(IRITE,600)
      WRITE(IRITE,610)
      WRITE(IRITE,600)
      WRITE(IRITE,801)
      WRITE(IRITE,530)' INTERIOR PTS IN X          (NX)=======> ',NX
      WRITE(IRITE,530)' INTERIOR PTS IN Y          (NY)=======> ',NY
      WRITE(IRITE,520)' MINIMUM X IN GRID          (AX)=======> ',AX
      WRITE(IRITE,520)' MINIMUM Y IN GRID          (AY)=======> ',AY
      WRITE(IRITE,520)' MAXIMUM X IN GRID          (BX)=======> ',BX
      WRITE(IRITE,520)' MAXIMUM Y IN GRID          (BY)=======> ',BY
      WRITE(IRITE,520)' STEPSIZE IN X              (ZHX)======> ',ZHX
      WRITE(IRITE,520)' STEPSIZE IN Y              (ZHY)======> ',ZHY
      WRITE(IRITE,801)
      WRITE(IRITE,600)
      WRITE(IRITE,801)
      WRITE(IRITE,500)' THE PROBLEM TITLE IS:   ',TITLE
      WRITE(IRITE,801)
      WRITE(IRITE,530)' DIMENSION OF LINEAR SYSTEM (N=NX*NY)==> ',N
      WRITE(IRITE,530)' INFORMATION LEVEL          (IOLEVL)===> ',IOLEVL
      WRITE(IRITE,530)' PRECONDITIONING KEY        (IPCOND)===> ',IPCOND
      WRITE(IRITE,530)' STOPPING CRITERION KEY     (ISTOP)====> ',ISTOP
      WRITE(IRITE,530)' MAXIMUM ALLOWED ITERATION  (ITMAX)====> ',ITMAX
      WRITE(IRITE,520)' ERROR TOLERANCE            (ERRTOL)===> ',ERRTOL
      WRITE(IRITE,530)' CONDITION ESTIMATE RATE    (ICYCLE)===> ',ICYCLE
      WRITE(IRITE,530)' CONDITION ESTIMATES        (NCE)======> ',NCE
      WRITE(IRITE,530)' CG METHOD USED             (ICG)======> ',ICG
      WRITE(IRITE,530)' DEGREE OF PREC POLY        (NDEG)=====> ',NDEG
      WRITE(IRITE,520)' INITIAL MIN EIG ESTIMATE   (AA)=======> ',AA
      WRITE(IRITE,520)' INITIAL MAX EIG ESTIMATE   (BB)=======> ',BB
      WRITE(IRITE,801)
      WRITE(IRITE,530)' OUTPUT KEY                 (KY)=======> ',KY
      WRITE(IRITE,801)
      WRITE(IRITE,530)' COMPLETION CODE            (IERROR)===> ',IERROR
      WRITE(IRITE,530)' ITERATIONS TAKEN           (ITERS)====> ',ITERS
      WRITE(IRITE,520)' FINAL STOPPING TEST        (STPTST)===> ',STPTST
      WRITE(IRITE,520)' FINAL CONDITION ESTIMATE   (CONDES)===> ',CONDES
      WRITE(IRITE,520)' SCALED RELATIVE RESIDUAL   (SCRLRS)===> ',SCRLRS
      WRITE(IRITE,520)' EXECUTION TIME             (CPUTME)===> ',CPUTME
      WRITE(IRITE,520)' RESIDUAL ERROR (B-A*XCG)   (RESID)====> ',RESID
      WRITE(IRITE,520)' PDE ANAL ERROR (XTRUE-XCG) (ERROR)====> ',ERROR
```

```
      WRITE(IRITE,801)
      WRITE(IRITE,600)
      WRITE(IRITE,801)
C
C     *** PRINT OUT THE SOLUTION VALUES ***
      IF (KY.EQ.1) THEN
         WRITE(IOUNIT,540) TITLE
         WRITE(IOUNIT,510) (NX+2)*(NY+2)
         WRITE(IOUNIT,801)
         X0   = AX
         Y0   = AY
         XNP1 = BX
         YNP1 = BY
         DO 10 I = 0, NX+1
            XI = AX + REAL(I) * ZHX
            WRITE (IOUNIT,550) XI,Y0,CUOG(XI,Y0)
 10      CONTINUE
         DO 30 J = 1, NY
            YJ = AY + REAL(J) * ZHY
            WRITE (IOUNIT,550) X0,YJ,CUOG(X0,YJ)
            DO 20 I = 1, NX
               XI = AX + REAL(I) * ZHX
               IDX = (J-1)*NX + I
               WRITE (IOUNIT,550) XI,YJ,X(IDX)
 20         CONTINUE
            WRITE (IOUNIT,550) XNP1,YJ,CUOG(XNP1,YJ)
 30      CONTINUE
         DO 40 I = 0, NX+1
            XI = AX + REAL(I) * ZHX
            WRITE (IOUNIT,550) XI,YNP1,CUOG(XI,YNP1)
 40      CONTINUE
         WRITE (IOUNIT,801)
      ENDIF
C
C     *** FORMAT STATEMENTS ***
 500  FORMAT (1X,A,A)
 510  FORMAT (I10)
 520  FORMAT (1X,A,1PE15.7)
 530  FORMAT (1X,A,I15)
 540  FORMAT (1X,A)
 550  FORMAT (1X,3(1PE15.7,10X))
 600  FORMAT (1X,'=======',
     2'============================================================',
     3           '=======')
 610  FORMAT (1X,'=======',
     2'============ PARTIAL DIFFERENTIAL EQUATION SOLVER ============',
     3           '=======')
 801  FORMAT (1X)
C
C     *** RETURN AND END ***
      RETURN
      END
      SUBROUTINE BUILD (N,AX,AY,BX,BY,NX,NY,A,IA,B,TRUE,IRITE)
C*****************************************************************************
C THIS ROUTINE BUILDS THE DISCRETE SYSTEM FROM THE USERS PDE SUBROUTINES.
```

```
C*********************************************************************
      DIMENSION A(*),IA(*),B(*),TRUE(*)
C
C     *** DEFINE N AND SETUP VARIOUS CONSTANTS FOR THE BUILD ***
      N     = NX * NY
      ZHX   = (BX-AX) / REAL(NX + 1)
      ZHY   = (BY-AY) / REAL(NY + 1)
      ZHXOY = ZHX / ZHY
      ZHYOX = ZHY / ZHX
      ZHPT  = ZHXOY / ZHYOX
      XO    = AX
      YO    = AY
      XNP1  = BX
      YNP1  = BY
      IA(1) = 1
      IA(2) = N+1
      IA(3) = 2*N+1
      IA(4) = 3*N+1
      IA(5) = 4*N+1
      IA(6) = NX
      IA(7) = NY
C
C     *** BUILD THE MESH POINTS, THE OPERATOR, THE RHS, AND TRUE ANAL SOLN ***
      IROW  = 0
      DO 10 J = 1, NY
         DO 10 I = 1, NX
            XI   = AX + REAL(I) * ZHX
            YJ   = AY + REAL(J) * ZHY
            IROW = IROW+1
            TRUE(IROW) = CUOU(XI,YJ)
            B(IROW) = - ZHX*ZHY*CUOF(XI,YJ)
C
C           *** SOUTH NEIGHBOR ***
            COEF = - ZHXOY
            IF (J .NE. 1) THEN
               A(IROW+IA(5)-1) = COEF
            ELSE
               A(IROW+IA(5)-1) = 0.0E0
               B(IROW) = B(IROW) - COEF*CUOG(XI,YO)
            ENDIF
C
C           *** WEST NEIGHBOR ***
            COEF = - ZHYOX
            IF (I .NE. 1) THEN
               A(IROW+IA(3)-1) = COEF
            ELSE
               A(IROW+IA(3)-1) = 0.0E0
               B(IROW) = B(IROW) - COEF*CUOG(XO,YJ)
            ENDIF
C
C           *** POINT ITSELF ***
            COEF = 4.0E0*ZHPT
            A(IROW+IA(1)-1) = COEF
C
C           *** EAST NEIGHBOR ***
```

```
            COEF = - ZHYOX
            IF (I .NE. NX) THEN
               A(IROW+IA(2)-1) = COEF
            ELSE
               A(IROW+IA(2)-1) = 0.0E0
               B(IROW) = B(IROW) - COEF*CUOG(XNP1,YJ)
            ENDIF
C
C           *** NORTH NEIGHBOR ***
            COEF = - ZHXOY
            IF (J .NE. NY) THEN
               A(IROW+IA(4)-1) = COEF
            ELSE
               A(IROW+IA(4)-1) = 0.0E0
               B(IROW) = B(IROW) - COEF*CUOG(XI,YNP1)
            ENDIF
 10   CONTINUE
C
C     *** RETURN AND END ***
      RETURN
      END
      SUBROUTINE MATMUL(JOB,A,IA,W,X,Y,N)
C**********************************************************************
C SPARSE MATRIX MULTIPLICATION.
C INTERFACE ROUTINE TO THE USERMV 5 DIAGONAL MATRIX-VECTOR PRODUCT ROUTINE.
C**********************************************************************
      DIMENSION A(*),IA(*),W(*),X(*),Y(*)
C
C     *** DO THE MULTIPLICATION ***
      CALL USERMV(A(IA(5)),A(IA(3)),A(IA(1)),A(IA(2)),A(IA(4)),
     2            IA(6),IA(7),X,Y)
C
C     *** GO HOME ***
      RETURN
      END
      SUBROUTINE USERMV(CL,BL,A,BU,CU,NX,NY,X,Y)
C**********************************************************************
C A FIVE DIAGONAL MATRIX-VECTOR PRODUCT ROUTINE.
C THE DIAGONALS ARE AS FOLLOWS:
C    THE COEFFICIENTS OF THE ITH EQUATION ARE STORED IN THE ITH COMPONENTS
C    OF THE ARRAYS CONTAINING THE DIAGONALS.
C    THE DIAGONALS OF THE OPERATOR ARE FROM LEFT TO RIGHT:   CL,BL,A,BU,CU
C    NX => THE NUMBER OF MESH POINTS IN THE X-DIRECTION.
C          WITH THE NATURAL ORDERING, NX IS THE ORDER OF EACH TRIDIAGONAL
C          BLOCK IN THE BLOCK-TRIDIAGONAL OPERATOR.
C    NY => THE NUMBER OF MESH POINTS IN THE Y-DIRECTION.
C          WITH THE NATURAL ORDERING, NY IS THE NUMBER OF TRIDIAGONAL
C          BLOCKS IN THE BLOCK-TRIDIAGONAL OPERATOR.
C**********************************************************************
      DIMENSION CL(*),BL(*),A(*),BU(*),CU(*),X(*),Y(*)
C
C     *** RECOVER MATRIX DIMENSION ***
      N = NX*NY
C
C     *** HANDLE FIRST BLOCK ***
```

```
      I = 1
      Y(I) = A(I)*X(I)+BU(I)*X(I+1)+CU(I)*X(I+NX)
      DO 10 I=2,NX
         Y(I) = BL(I)*X(I-1)+A(I)*X(I)+BU(I)*X(I+1)+CU(I)*X(I+NX)
 10   CONTINUE
C
C     *** HANDLE MIDDLE BLOCKS ***
      DO 20 I=NX+1,N-NX
         Y(I) = CL(I)*X(I-NX)+BL(I)*X(I-1)+A(I)*X(I)+BU(I)*X(I+1)
     2         +CU(I)*X(I+NX)
 20   CONTINUE
C
C     *** HANDLE LAST BLOCK ***
      DO 30 I=N-(NX-1),N-1
         Y(I) = CL(I)*X(I-NX)+BL(I)*X(I-1)+A(I)*X(I)+BU(I)*X(I+1)
 30   CONTINUE
      I = N
      Y(I) = CL(I)*X(I-NX)+BL(I)*X(I-1)+A(I)*X(I)
C
C     *** RETURN AND END ***
      RETURN
      END
      SUBROUTINE BLDPC(A,IA,Q,IQ,N)
C*****************************************************************************
C BUILD A SIMPLE DIAGONAL SCALING PRECONDITIONER FOR A FIVE-DIAGONAL MATRIX.
C*****************************************************************************
      DIMENSION A(*),IA(*),Q(*),IQ(*)
      DO 10 I = 1, N
         Q(I) = 1.0E0 / A(IA(1)+(I-1))
 10   CONTINUE
      RETURN
      END
      SUBROUTINE DIAGPC(JOB,Q,IQ,W,X,Y,N)
C*****************************************************************************
C A SIMPLE DIAGONAL SCALING PRECONDITIONER FOR A FIVE-DIAGONAL OPERATOR.
C*****************************************************************************
      DIMENSION Q(*),IQ(*),W(*),X(*),Y(*)
      DO 10 I = 1, N
         Y(I) = Q(I)*X(I)
 10   CONTINUE
      RETURN
      END
      FUNCTION CUOF (X,Y)
C*****************************************************************************
C THIS ROUTINE DEFINES THE RIGHT HAND SIDE OF THE PDE.
C*****************************************************************************
      PARAMETER (PI = 3.1415926535,NNX=3,NNY=1)
      CUOF = - PI*PI*REAL(NNX*NNX + NNY*NNY)
     2       * (SIN(REAL(NNX)*PI*X) * SIN(REAL(NNY)*PI*Y))
      RETURN
      END
      FUNCTION CUOG (X,Y)
C*****************************************************************************
C THIS ROUTINE IS THE BOUDARY CONDITION FOR THE PDE.
C*****************************************************************************
```

```
      CUOG = 0.0E0
      RETURN
      END
      FUNCTION CUOU (X,Y)
C**********************************************************************
C THIS ROUTINE IS THE ANALYTIC SOLUTION TO THE PDE (FOR TESTING PURPOSES).
C**********************************************************************
      PARAMETER (PI = 3.1415926535,NNX=3,NNY=1)
      CUOU = SIN(REAL(NNX)*PI*X) * SIN(REAL(NNY)*PI*Y)
      RETURN
      END
```

The following is a sample input file to the driver.

```
0                   iolevl    information output level
1                   ipcond    preconditioning flag
0                   istop     stopping criterion flag
500                 itmax     iteration maximum
1.0e-5              errtol    error tolerance

0                   icycle    condition number estimate cycle
0                   nce       number of condition estimates
4                   icg       cg method desired

0                   ndeg      degree of preconditioning polynomial
1.0e0               cond      initial condition number estimate
1.0e0               aa        estimate of smallest eigenvalue
1.0e0               bb        estimate of largest eigenvalue

0                   ky        whether to print solution

63                  nx        number of interior points in x direction (nx)
63                  ny        number of interior points in y direction (ny)
0.0e0               ax        minimum for x in grid (ax)
0.0e0               ay        minimum for x in grid (ay)
1.0e0               bx        maximum for x in grid (bx)
1.0e0               by        maximum for x in grid (by)
```

The following is a sample output file from the driver.

```
 ========================================================================
 ================== PARTIAL DIFFERENTIAL EQUATION SOLVER ================
 ========================================================================

  INTERIOR PTS IN X          (NX)=======>             63
  INTERIOR PTS IN Y          (NY)=======>             63
  MINIMUM X IN GRID          (AX)=======>   0.0000000E+00
  MINIMUM Y IN GRID          (AY)=======>   0.0000000E+00
  MAXIMUM X IN GRID          (BX)=======>   1.0000000E+00
  MAXIMUM Y IN GRID          (BY)=======>   1.0000000E+00
  STEPSIZE IN X              (ZHX)======>   1.5625000E-02
  STEPSIZE IN Y              (ZHY)======>   1.5625000E-02
```

```
================================================================================

 THE PROBLEM TITLE IS:    SOLUTION OF POISSON EQUATION USING CGCODE

 DIMENSION OF LINEAR SYSTEM (N=NX*NY)==>            3969
 INFORMATION LEVEL          (IOLEVL)===>               0
 PRECONDITIONING KEY        (IPCOND)===>               1
 STOPPING CRITERION KEY     (ISTOP)====>               0
 MAXIMUM ALLOWED ITERATION  (ITMAX)====>             500
 ERROR TOLERANCE            (ERRTOL)===>   9.9999990E-06
 CONDITION ESTIMATE RATE    (ICYCLE)===>               0
 CONDITION ESTIMATES        (NCE)======>               0
 CG METHOD USED             (ICG)======>               3
 DEGREE OF PREC POLY        (NDEG)=====>               0
 INITIAL MIN EIG ESTIMATE   (AA)=======>   1.0000000E+00
 INITIAL MAX EIG ESTIMATE   (BB)=======>   1.0000000E+00

 OUTPUT KEY                 (KY)=======>               0

 COMPLETION CODE            (IERROR)===>               0
 ITERATIONS TAKEN           (ITERS)====>               2
 FINAL STOPPING TEST        (STPTST)===>   5.7627392E-06
 FINAL CONDITION ESTIMATE   (CONDES)===>   1.0000000E+00
 SCALED RELATIVE RESIDUAL   (SCRLRS)===>   5.7627392E-06
 EXECUTION TIME             (CPUTME)===>   1.6622700E-01
 RESIDUAL ERROR (B-A*XCG)   (RESID)====>   8.2791452E-06
 PDE ANAL ERROR (XTRUE-XCG) (ERROR)====>   5.2695351E-02

================================================================================
```

The following is a sample output file from the solver.

```
THE METHOD IS PRECONDITIONED CG (PCG)

THE ITERATION PARAMETERS ARE
   N      =  3969
   ITMAX  =   500
   ICYCLE =     0
   NCE    =     0
   ERRTOL =  0.10000E-04
   CNDMIA =  0.10000E+01

RESID  = 2-NORM OF MINV*R
RELRSD = RESID / INITIAL RESID
COND(MINV*A) USED IN STOPPING CRITERION

INITIAL RESID =  0.19277E+00

ITERS =     1    RESID =  0.37110E-05    RELRSD =  0.19251E-04
ITERS =     2    RESID =  0.11109E-05    RELRSD =  0.57627E-05
```