

DEF603-93ER25183

DOE/ER/25183--T4

PCG REFERENCE MANUAL

by

W.D. Joubert, G.F. Carey, N.A. Berner, A. Kalhan
H. Kohli, A. Lorber, R.T. McLay, Y. Shen

CNA-274

January 1995

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PCG Reference Manual

A Package for the Iterative Solution
of Large Sparse Linear Systems
on Parallel Computers

Version 1.0

W. D. Joubert[†] G. F. Carey[‡] N. A. Berner^{††} A. Kalhan^{†*}

H. Kohli[‡] A. Lorber[‡] R. T. McLay[‡] Y. Shen[‡]

[†] Los Alamos National Laboratory, Los Alamos, NM 87545

[‡] TICAM, WRW305, The University of Texas at Austin, Austin, TX 78712

* University of Tennessee, Knoxville, TN 37966

Disclaimer

The individuals and institutions listed on the cover of this document disclaim all warranties with regard to the PCG software package, including all warranties of merchantability and fitness, and any stated express warranties are in lieu of all obligations or liability on the part of these individuals and institutions for damages, including, but not limited to, special, indirect, or consequential damages arising out of or in connection with the use or performance of PCG. In no event will these individuals or institutions be liable for any direct, indirect, special, incidental, or consequential damages arising in connection with use of or inability to use PCG or the documentation.

The current version of PCG is preliminary, and the package is under development. Expansion of the package is in progress. Reports of difficulties encountered in using the system or comments and suggestions for improving the package are welcome. Please direct comments or inquiries to:

Wayne Joubert
Los Alamos National Laboratory
Group C-3, MS-B265
Los Alamos, NM 87545
e-mail: wdj@lanl.gov

G. F. Carey
TICAM, WRW305
The University of Texas at Austin
Austin, TX 78712
e-mail: carey@cfdlab.ae.utexas.edu

Contents

Disclaimer	i
Contents	ii
List of Tables	iv
Preface	v
1 Introduction	1
1.1 Design Objectives	1
1.2 Basic Features	1
1.3 Currently Supported Machines	2
1.4 Mathematical Notation	3
2 Usage	4
2.1 Top Level Usage	4
2.2 Iterative Method Level Usage	7
2.3 Reverse Communication Usage	9
2.4 Solving Multiple Linear Systems	11
3 Adaptations for Particular Machines	13
3.1 Use of Arrays	13
3.2 Processor Numbering	13
3.3 Communication on Message Passing machines	14
3.4 Input and Output	15
3.5 Memory Allocation	16
3.6 Subroutine Naming	16
3.7 Precisions and Arithmetics	16
4 Parameter Arrays IPARM and FPARM	17
4.1 The Use of Parameter Arrays	17
4.2 Integer Parameters, Reverse Communication Level	18
4.3 Integer Parameters, Iterative Method Level	22
4.4 Integer Parameters, Top Level	22
4.5 Floating Point Parameters, Reverse Communication Level	22
4.6 Floating Point Parameters, Iterative Method Level	23

4.7	Floating Point Parameters, Top Level	23
4.8	Which Parameters to Set	24
5	Matrix Storage Modes	25
5.1	The Regular Grid Stencil Format	25
5.2	Array Specification	26
5.3	Processor Embedding for Message Passing Versions	29
6	Matrix Preprocessing and Preconditioning	32
6.1	Matrix Preprocessing/Scaling	32
6.2	Matrix Preconditioning	33
7	Choices of Iterative Method	35
7.1	General Considerations	35
7.2	Available Iterative Methods	36
7.3	Guidelines for Choosing an Iterative Method	38
8	Stopping Tests and Error Conditions	40
8.1	Stopping Tests	40
8.2	Error Conditions	41
9	Package Requirements	44
9.1	Memory Requirements	44
9.2	Message Type Requirements	47
9.3	Counting Floating Point Operations	47
9.4	Package Timings	48
10	Sample Programs	49
10.1	The Model Problem	49
10.2	Cray YMP Example	49
10.3	CM Fortran Example	51
10.4	Intel IPSC/860 Example	52
11	Troubleshooting	55
	Acknowledgements	57
	References	58
	Index	62

List of Tables

1.1	Current Implementations	2
3.1	Default Values of MSGMIN and MSGMAX	14
9.1	Basic Memory Requirements	46
9.2	Additional Memory for Different Methods	46
9.3	Message Type Requirements	47
9.4	Operation Count Scale	48

Preface

What is PCG ?

PCG (Preconditioned Conjugate Gradient package) is a system for solving linear equations of the form $Au = b$, for A a given matrix and b and u vectors. The PCG package employs various gradient-type iterative methods coupled with preconditioners of various types. The PCG software library is designed to be applicable to general linear systems, with emphasis on sparse systems such as those arising from the discretization of partial differential equations arising from physical applications.

The PCG package can be used to solve linear equations efficiently on parallel computer architectures, such as the Intel Paragon, Connection Machine and Cray T3D computers. Fortran versions of the package are supported for computers which use either a data parallel programming environment or a message passing programming environment.

PCG is available in a generic Fortran 77 version, usable on single-processor computers. Platform-specific Fortran versions are available as well, such as a CM Fortran version for Connection Machines, a node-level Fortran version for the Intel Paragon, and a node-level Fortran version with PVM calls for systems or networks which implement the PVM (Parallel Virtual Machine) system, including the Cray T3D. The use of macros in a higher-level language for the package implies that Fortran versions for other machines can be constructed within the existing framework. Thus, much of the code is reusable across architectures and in this sense the package is portable across very different systems. A complete list of machines that are currently supported is found in Chapter 1 of this manual.

How to Use This Manual

This manual is intended to be the general-purpose reference describing all features of the package accessible to the user. Suggestions are also given regarding which methods to use for a given problem.

Users wishing to get started right away with the package should see the *Getting Started with PCG* document, which shows how to implement a single example from start to finish. A more extensive list of examples is found in the *PCG Examples Manual*. The user may wish to study the sections of the *Getting Started* manual, or the actual sample code, in conjunction with the associated chapters of this manual. A graphical user interface XPCG is also provided and is described in the *XPCG User's Manual*.

Section 2.1 of this Reference Manual describes the calling sequence for the simplest ("black box") level of accessing the package; lower-level means of access are described in Sections 2.2 and 2.3.

Other chapters describe in detail the choices of arguments and switch settings to be passed in to the package to control the iterative process in detail. Chapter 3 describes particular considerations for porting to specific parallel machines. Brief examples of usage are given in Chapter 10, and troubleshooting tips are given in Chapter 11.

Some chapters contain both a section on available methods or features, as well as a shorter section on guidelines for use. The user wishing to get started more quickly should begin with the guidelines section and later make use of the longer section when trying to fine-tune performance.

Related Documents

The following materials are included in the PCG documentation:

- *PCG Reference Manual* (this document).
- *Getting Started with PCG*, a short introductory document which describes in detail one example of using the package.
- *PCG Examples Manual*, a lengthy list of usage examples, used to illustrate a range of possible ways to use the package.
- *PCG Quick Reference Card*, a short reference card for package calling sequences and similar information.
- *XPCG Users' Manual*, a reference manual for the X-windows-based graphical interface for using the package.
- *PCG Programmer's Manual*, a manual describing in detail the internal construction of the package as an aid to adding features to the package.

Chapter 1

Introduction

1.1 Design Objectives

PCG is a computer package to solve large sparse systems of linear equations by iterative methods on different computer architectures. The package implements various iterative methods such as the conjugate gradient method and generalized conjugate gradient methods for nonsymmetric systems, in conjunction with various preconditioners. This document describes in detail the usage of the PCG package. For a briefer introduction, please consult the introductory document, *Getting Started with PCG*.

1.2 Basic Features

The goal of the PCG software is to provide high-performance state-of-the-art sparse iterative algorithms across a variety of architectures, including in particular shared and distributed memory parallel machines. Some of the specific features of the package are the following:

- (i) *Iterative Methods.* A wide variety of iterative methods are provided, such as the conjugate gradient (CG) method, the conjugate gradient squared (CGS) method, the biconjugate gradient (BCG) and QMR algorithms, and restarted GMRES.
- (ii) *Preconditioners.* Various parallelized preconditioners are available in PCG.
- (iii) *Matrix Formats.* PCG provides predefined matrix storage schemes for storage of the user's matrix. These storage schemes have been chosen to provide sufficient generality to implement a wide variety of sparse problems while at the same time taking greatest advantage of the given machine architectures.
- (iv) *Levels of Access.* The package features several layers of access. The simplest means of access is by a top-level or "black box" call, using one of the available predefined package sparse matrix formats. Alternatively, the package can be called at the level of the iterative method with a direct communication interface, for which the user provides subroutines to perform the matrix-vector product and preconditioner kernel operations for a user-defined format. Finally, a reverse communication layer allows greater user control of the iterative process by permitting more general forms of the matrix-vector product and preconditioning operations.

- (v) *Preprocessing Options.* Several matrix preprocessing options are available with the top-level access mode. These options can be used to improve the linear solution process, by means such as diagonal matrix scaling.
- (vi) *Differing Precisions and Arithmetics.* All package routines are available in single and double precision and real and complex arithmetic versions. The LINPACK/LAPACK subroutine naming conventions are followed, by which the first letter of the routine denotes the desired arithmetic and precision (S=single real, D=double real, C=single complex, Z=double complex).
- (vii) *Modularity.* The user is allowed to utilize virtually any combination of preconditioner, iterative method, matrix format, arithmetic, precision and machine type provided in the package to solve the given problem.
- (viii) *Solution of Multiple Systems.* The package may be used to solve a series of linear systems efficiently by reuse of the calculated preconditioner. This is particularly effective for the solution of time-dependent partial differential equations (PDE's) and certain nonlinear problems, for example.
- (ix) *High Performance.* Considerable effort has been directed to optimizing the package to give high performance on particular parallel machines.
- (x) *Portability.* Besides a generic Fortran 77 version, optimized Fortran versions for specific parallel machines are provided, which have as nearly as possible the same user interface across machines. These may be generated from a single source written using higher-level macros. See the *PCG Programmer's Manual* for more details.

1.3 Currently Supported Machines

The machines and compilers in Table 1.1 are currently supported by the PCG package:

Table 1.1: Current Implementations

Machine	Compiler
Intel iPSC/860	if77
Intel Paragon	f77
Connection Machine 2/200	CM Fortran (slicewise)
Connection Machine 5, data parallel	CM Fortran
Connection Machine 5, message passing	f77
generic Fortran 77 + PVM	(Fortran)
generic Fortran 77 + MPI	(Fortran)
Cray YMP	cf77
Cray T3D, PVM interface	cf77

Due to inherent differences in programming models for the different machines, small variations in the usage of the package are necessary from machine to machine. These differences are described in detail in Chapter 3.

1.4 Mathematical Notation

The following notation is used throughout this manual to describe the solution techniques for solving system of linear equations. The system of equations to be solved is denoted by

$$Au = b \quad (1.1)$$

where A is a square matrix of size N . The corresponding preconditioned system $\hat{A}\hat{u} = \hat{b}$ may be obtained by defining the related quantities $\hat{A} = Q_L A Q_R$, $u = Q_R \hat{u}$ and $\hat{b} = Q_L b$, for some square nonsingular preconditioning matrices Q_L and Q_R , so that

$$(Q_L A Q_R)(Q_R^{-1} u) = (Q_L b) \quad (1.2)$$

or

$$\hat{A}\hat{u} = \hat{b} \quad (1.3)$$

The preconditioned linear system (1.3) is then passed to one of the iterative methods, such as the conjugate gradient method, GMRES, and so forth. It is common to utilize *left preconditioning*, for which $Q_L = Q$ is given and $Q_R = I$, but other arrangements of the preconditioner are also possible.

The user supplies the package with the vector b and an (optional) initial vector $u^{(0)}$. The package then generates iterates $u^{(i)}$, $i = 1, 2, \dots$. The error in $u^{(i)}$ is denoted in this document by $e^{(i)} = u - u^{(i)}$, and the residual associated with (1.1) by $r^{(i)} = b - Au^{(i)} = -Ae^{(i)}$. We also define the related quantities associated with (1.3): $Q_R \hat{u}^{(i)} = u^{(i)}$, $\hat{e}^{(i)} = \hat{u} - \hat{u}^{(i)}$, $Q_R \hat{e}^{(i)} = e^{(i)}$, and the preconditioned residual $\hat{r}^{(i)} = \hat{b} - \hat{A}\hat{u}^{(i)} = Q_L r^{(i)}$.

Throughout this manual, (\cdot, \cdot) and $\|\cdot\|$ denote the standard Euclidean ℓ^2 inner product and norm, $(u, v) = \sum \bar{u}_i v_i$, $\|v\| = \sqrt{(v, v)}$. A Hermitian matrix is a matrix which satisfies the property $M^* = M$, where $*$ denotes the conjugate transpose. When the matrix is real, this definition reduces to the definition of a symmetric matrix.

Chapter 2

Usage

Three usage levels are provided: (1) Top level usage; (2) Iterative method level usage; and (3) Reverse communication usage. Each of these levels is now briefly described. Finally in Section 2.4 we include some remarks on solving multiple systems.

2.1 Top Level Usage

The basic top-level calling sequence for the package is given by:

```
CALL _PCG (IJOB, _ppff, _meth, IA, JA, A, U, UEXACT, B,  
&          IWK, FWK, IPARM, FPARM, IER)
```

Here, the LINPACK/LAPACK convention is used for subroutine names: `_PCG` is one of `SPCG`, `DPCG`, `CPCG` or `ZPCG`, depending on whether single precision real, double real, single complex or double complex arithmetic is to be used. Throughout this manual, the prefix “_” for a subroutine name will refer to one of these four prefix letters. For all calls to the package, *all* floating point arrays passed into the package must conform in type (precision and arithmetic) to the type specified in the subroutine name. A value of this type will be referred to throughout this manual simply as a “floating point value”. Furthermore, subroutine names for different types may not be mixed in the same subroutine call, and calls associated with the same package initialization call must not mix types.

The parameters passed to the top-level routine `_PCG` are defined as follows:

IJOB (Input) An integer scalar which specifies the task to be performed by the package. The choices for this parameter are described in detail in Section 2.4. Mnemonics as defined in the system header file `pcg_fort.h` (see Chapter 4) may be used, as indicated below in parentheses. Allowable choices are:

IJOB=0 (=JIRT) Initialize, apply the iterative method, and terminate. This is the simplest and most common means of using the package.

- IJOB=1 (=JINIT)** Perform the setup for the initialization only (without solving the system), and save the preconditioner and other initializations internally for use with a future call to PCG. The IPARM variable IPTR is used to reference the matrix and preconditioner setup from this call.
- IJOB=2 (=JINITA)** Same as IJOB=JINIT, except that a copy of the matrix is also stored internally. This is important if the package is called later with a different matrix but with the same preconditioner as was generated by the setup call, and the preconditioner requires reference to the original matrix.
- IJOB=3 (=JRUN)** Apply the iterative method to the linear system based on the matrix and preconditioner from the initialization call. The arrays IA, A, IWK and FWK must be the same arrays as those passed into the package at the setup call and the terminate call. Furthermore, the preconditioner and iterative method choices and certain entries of IPARM and FPARM must be unchanged across such calls.
- IJOB=4 (=JRUNA)** Same as IJOB=JRUN, except that the entries of the matrix may have changed from those for the setup call. The new matrix but the old preconditioner are used. The structure of the new matrix must conform to that of the saved matrix. The initialization call for this case must necessarily have been IJOB=JINITA, not IJOB=JINIT. The saved matrix from the initialization call is not replaced by the new one from this call.
- IJOB=5 (=JRUNAQ)** Same as IJOB=JRUNA, except that the matrix is changed and the preconditioner is to be recalculated based on the new matrix. In this case the matrix and vectors need not agree in any way with the saved information from the setup call. In some cases, however, the previous setup information may be used to improve the speed of the new setup. The saved matrix and preconditioner from the initialization call is not replaced by the new one from this call.
- IJOB=-1 (=JTERM)** Termination: the saved matrix and preconditioner from a call with IJOB=JINIT or IJOB=JINITA are released from memory. It is not necessary to make this call if the package terminated with a fatal error.

_ppff (Input) An externally-defined name which refers to the user's choice of the preconditioning and also indicates the matrix storage format being used. The letters **pp** select the preconditioning, and the letters **ff** select the matrix format. The actual name must be chosen from the lists given in Chapter 5 and Section 6.2.

_meth (Input) An externally-defined name which refers to the user's choice for the iterative method. The actual name must be chosen from the list given in Chapter 7.

IA (Input, Output) An integer array used to store indexing information for the nonzero elements of the matrix stored in A. The exact specification of this array depends on the format chosen and is described in Chapter 5.

JA (Input, Output) A second integer array used to store indexing information for the nonzero elements of the matrix stored in A. The exact specification of this array depends on the format chosen and is described in Chapter 5.

- A** (Input, Output) A floating point array used to store nonzero entries of the matrix A . The exact specification of this array depends on the format chosen and is described in Chapter 5.
- U** (Input, Output) A floating point array. On input, this array contains an optional initial guess $u^{(0)}$ to the true solution u . By default, the IPARM variable IUINIT specifies that this vector is to be initialized by the package to an appropriate vector, so the user need not set it initially; however, the initial guess may be set by the user if desired. On output, it contains the final approximation $u^{(n)}$ to the solution u generated by the package. The exact specification of this array depends on the matrix format used and is described in Chapter 5.
- UEXACT** (Input, Output) A floating point array which optionally contains the true solution u to the linear system (1), used for testing purposes. If not used, then this reference to UEXACT should be replaced by a reference to the vector U , which functions as a dummy array. The IPARM variable IUEXAC indicates whether UEXACT in fact contains a valid value for the true solution u . The exact specification of this array depends on the matrix format used and is described in Chapter 5.
- B** (Input, Output) A floating point array which contains the right hand side b of the linear system (1.1). The exact specification of this array depends on the matrix format used and is described in Chapter 5.
- IWK** (Output) An integer array which contains integer workspace which may be used by the package. For CM Fortran versions, this is a front-end array. For machine-specific versions, the IPARM variable MALLOC by default indicates that workspace is obtained by the package by internal memory allocation, in which case this may be an array of length one; however, in any case this parameter *must* be an array, of type integer, and this single memory location is in fact used by the package. Furthermore, the memory location passed to the package in the form of this parameter must be identical across calls to the package related by the same value of the IPARM variable IPTR.
- FWK** (Output) A floating point array which contains floating point workspace which may be used by the package. For CM Fortran versions, this is a front-end array. For machine-specific versions, the IPARM variable MALLOC by default indicates that workspace is obtained by the package by internal memory allocation, in which case this may be an array of length one; however, in any case this parameter *must* be an array, of precision and arithmetic consistent with the called name of the PCG subroutine, and this single memory location is in fact used by the package. Furthermore, the memory location passed to the package in the form of this parameter must be identical across calls to the package related by the same value of the IPARM variable IPTR.
- IPARM** (Input, Output) An integer array which is used to pass integer parameters to and from the package. For CM Fortran versions, this is a front-end array. The size and entries of IPARM are described in Chapter 4.
- FPARM** (Input, Output) A floating point array which is used to pass floating point parameters to and from the package. For CM Fortran versions, this is a front-end array. The size and

entries of `FPARM` are described in Chapter 4.

IER (Output) Error code for the package. A value of zero indicates normal termination, while values greater than zero indicate warnings, and values less than zero indicate fatal errors. The possible error codes are described in Section 8.2 and are also listed in the `pcg_fort.h` header file (see Chapter 4).

Remark:

The arrays `IPARM` and `FPARM` contain many parameters which for most cases need not be altered from certain commonly-used default settings. To simplify setting these arrays, a package routine is supplied to initialize these arrays to default settings before calling `PCG`.

The defaults routine `_DFALT` of precision and arithmetic corresponding with that of the associated `PCG` call initializes the arrays `IPARM` and `FPARM`. The calling sequence for this routine is:

```
CALL _DFALT (IPARM,FPARM)
```

Sample programs to illustrate the use of the `_PCG` and `_DFALT` routines are given in Chapter 10 of this manual.

2.2 Iterative Method Level Usage

The `PCG` package allows the user the option of calling the iterative methods, such as `_CG`, directly, by specifying the matrix-vector product and preconditioning as user-defined subroutines. By calling the package at this level, the user can avoid using the builtin array storage formats and instead define a custom format for the desired matrix and vectors. An even greater amount of control of the matrix-vector product and preconditioning operations may be obtained by accessing `PCG` iterative methods at the reverse communication level as described in Section 2.3 following.

The basic calling sequence for the package called at the iterative method level is given by

```
CALL _meth (IJOB,mysuba,IA,JA,A,mysubq,IQ,JQ,Q,U,UEXACT,B,  
&          IWK,FWK,IPARM,FPARM,IER)
```

The allowable choices for `meth` (e.g., `CG`) are listed in Chapter 7. As with the top-level call, a call to the package at this level is typically preceded by a call to the routine `_DFALT`. The parameters for the call to `meth` are given as follows:

mysuba (Input) The subroutine name of an externally declared user-supplied routine which must compute the product of the matrix A (or its conjugate transpose, for some iterative methods) times a vector. The calling sequence for this routine must satisfy the specifications given below.

IA, JA, A (Input) Arbitrary arrays, used by the `mysuba` routine to accomplish the matrix-vector product, in whatever fashion the user wishes.

mysubq (Input) The subroutine name of an externally declared user-supplied routine which must compute the product of the matrices Q_L or Q_R (or the conjugate transposes, for some iterative methods) times a vector. The calling sequence for this routine must satisfy the specifications given below. It should be noted that if IQSIDE specifies that either Q_L or Q_R is the identity matrix, then the corresponding operation or its conjugate transpose will never be requested of this routine by the iterative method, so in such cases this routine need not perform any action.

IQ, JQ, Q (Input) Arbitrary arrays, used by the **mysubq** routine to accomplish the preconditioning, in whatever fashion the user wishes.

All other arguments to the iterative method follow similar guidelines to those for the arguments to the top-level routine **_PCG** defined in Section 2.1. The argument **IJOB** must always be $0 = \text{JIRT}$. The specifications for the vectors **U**, **UEXACT** and **B**, as well as **VI** and **VO** described below, are arbitrary, but the elements of these vectors must be contiguous (except for CM Fortran versions), and the size of the vectors must be specified in the **IPARM** variable **NRU**. For CM Fortran versions, the arrays are assumed to be back-end arrays of matching rank and dimension on each axis, with **NRU** denoting the total number of elements.

The user-defined routines **mysuba** and **mysubq** are assumed to have calling sequences of the following form:

```
CALL mysuba (IJOB,IA,JA,A,VI,VO)
CALL mysubq (IJOB,IQ,JQ,Q,VI,VO)
```

Here, **VI** is the input vector to the routine and **VO** is the output vector. These vectors, as passed to the routines by the iterative method, are guaranteed not to overlap in any way. The parameter **IJOB** is defined as follows. For **mysuba**, when **IJOB=3=JAV** the routine should compute the matrix-vector product with **A**, and when **IJOB=4=JATV** the product with the conjugate transpose of **A** should be computed. Similarly, for **mysubq**, **IJOB=5=JQLV** specifies Q_L , **IJOB=6=JQLTV** specifies Q_L^* , **IJOB=7=JQRV** specifies Q_R , and **IJOB=8=JQRTV** specifies Q_R^* , all multiplied by the vector **VI**.

The conjugate transpose operations are only requested of **mysuba** and **mysubq** by those iterative methods as described in Chapter 7 which require them. Furthermore, the package will not call **mysubq** for preconditioning operations not specified by the variable **IQSIDE**. The **IPARM** variable **ICTRAN** is not used in a call to this level, but the user may choose for **mysuba** and **mysubq** to apply the transpose rather than the conjugate transpose in situations in which this is desired.

The use of user-defined routines allows a high degree of generality in the formats for the matrix and preconditioner storage. The iterative method performs low-level vector operations such as vector adds as well as the allocation of temporary vectors based on the specifications of the user-specified vectors **B** and so forth. The low-level operations assume that every element of each array **B** etc. corresponds to a distinct vector element, and that contiguity of the elements in memory is assured. Thus, duplication of points by overlapping of subdomains is not currently allowed.

For CM Fortran versions, the vectors supplied to **mysuba** and **mysubq** are back-end arrays which have a certain layout. For **mysuba** performing the matrix-vector product operation, **VI**'s rank, size and layout matches that of **U**, and **VO**'s matches that of **B**; when the conjugate transpose is requested,

VI is shaped like B, and VO is shaped like U. In `mysubq`, for an operation with Q_L , both vectors are shaped like B; similarly, for an operation with Q_R , both vectors are shaped like U.

For examples of the use of the package in matrix format free mode, see the *PCG Examples Manual*.

2.3 Reverse Communication Usage

The reverse communication layer of PCG is similar to the iterative method level of access but allows greater flexibility and control by the user. For this level of access, it is not necessary to encapsulate the matrix vector product and preconditioner into subroutines with a specific calling sequence, but instead, the package routine returns temporarily to the calling routine with a request to perform the desired vector operation on specified vectors.

The calling sequence for PCG at the reverse communication level is given by

```
CALL _methR (IJOB,IREQ,U,UEXACT,B,IVA,IVQL,IVQR,  
&           IWK,FWK,IPARM,FPARM,IER)
```

The allowable choices for `meth` (e.g. CG) are listed in Chapter 7. As with the other levels, a call to the package at this level is typically preceded by a call to the routine `_DFALT`. The parameters for the call to `methR` are given as follows:

- IJOB** (Input) An integer scalar which specifies the task to be performed by the package. Admissible choices are:
- IJOB=1 (JINIT)** This choice for IJOB should be used for the first call to `_methR`. On return from this call, the package makes its first request of the user via `IREQ`.
 - IJOB=3 (JRUN)** This choice should be used for every call following the initialization call, up to termination.
 - IJOB=-1 (JTERM)** This choice should be used to force the routine to terminate and yield the final value of U. This call is not necessary if the package terminates due to a fatal error.
- IREQ** (Output) An integer scalar by which the routine `_methR` requests of the caller to perform a certain operation and then make another call to `_methR`. Admissible choices are:
- IREQ=3 (JAV)** The caller should apply a matrix vector product to the vector in location `IVQR` and put the result in location `IVA`. The meaning of "vector location" is described below.
 - IREQ=4 (JATV)** The caller should apply a conjugate transpose matrix vector product to the vector in location `IVA` and put the result in location `IVQR`. This request is never made if the conjugate transpose operation is not required by the iterative method, as described in Chapter 7.

- IREQ=5 (JQLV) The caller should apply Q_L to the vector in location IVA and put the result in location IVQL. This request is never made if the IPARM variable IQSIDE denotes that the relevant preconditioner is the identity.
- IREQ=6 (JQLTV) The caller should apply Q_L^* to the vector in location IVQL and put the result in location IVA. This request is never made if the conjugate transpose operation is not required by the iterative method, as described in Chapter 7, or if the IPARM variable IQSIDE denotes that the relevant preconditioner is the identity.
- IREQ=7 (JQRV) The caller should apply Q_R to the vector in location IVQL and put the result in location IVQR. This request is never made if the IPARM variable IQSIDE denotes that the relevant preconditioner is the identity.
- IREQ=8 (JQRTV) The caller should apply Q_R^* to the vector in location IVQR and put the result in location IVQL. This request is never made if the conjugate transpose operation is not required by the iterative method, as described in Chapter 7, or if the IPARM variable IQSIDE denotes that the relevant preconditioner is the identity.
- IREQ=9 (JTEST) The caller should test for convergence, and then call `_methR` with either IJOB=JRUN or IJOB=JTERM, based on the result. The IPARM variable NEEDRC may be used to force the package to return $u^{(n)}$ (in U), $r^{(n)}$ (in location IVA), $Q_L r^{(n)}$ (in location IVQL), and $Q_R Q_L r^{(n)}$ (in location IVQR), for the calling routine to use for its stopping test. These vectors are only returned when IREQ=JTEST. This request is never made if the IPARM variable NTEST is not TST0.
- IREQ=-1 (JTERM) The routine `_methR` has terminated, either normally or abnormally, and should not be called again.

- IVA (Input, Output) For non-CMF versions, this is an output integer scalar which is the first location in the array FWK of the desired vector, which is of length IPARM(NRU) supplied by the caller. For CM Fortran versions, this is a back end array of rank and axis lengths matching those of U, UEXACT and B supplied by the calling routine and used for the requested operation.
- IVQL (Input, Output) For non-CMF versions, this is an output integer scalar which is the first location in the array FWK of the desired vector, which is of length IPARM(NRU) supplied by the caller. For CM Fortran versions, this is a back end array of rank and axis lengths matching those of U, UEXACT and B supplied by the calling routine and used for the requested operation.
- IVQR (Input, Output) For non-CMF versions, this is an output integer scalar which is the first location in the array FWK of the desired vector, which is of length IPARM(NRU) supplied by the caller. For CM Fortran versions, this is a back end array of rank and axis lengths matching those of U, UEXACT and B supplied by the calling routine and used for the requested operation.

All other arguments to the iterative method follow similar guidelines to those for the arguments to the top-level routine `_PCG` defined in Section 2.1. The specifications for the vectors U, UEXACT and B, as well as those at locations IVA, IVQL and IVQR, are arbitrary, but the elements of these vectors must be contiguous (except for CM Fortran versions), and the size of the vectors must be specified

in the IPARM variable NRU. For CM Fortran versions, the arrays are assumed to be back-end arrays of matching rank and dimension on each axis, with NRU denoting the total number of elements, but they may differ in layout. For non-CMF versions, vector locations supplied to the user guarantee that the relevant vectors do not overlap in memory in any way. For CMF versions, the vectors are only used for the requested operations, and the user may allow the vectors to overlap or be the same vector if desired.

For non-CMF versions, if the IPARM variable MALLOC is 1=YES, then the indices IVA, IVQL and IVQR may denote locations that are not within the declared extents of FWK, which may cause a subscriptrange error within the user's program. This may be remedied by turning off subscriptrange checking or setting MALLOC to 0=NO.

The user should not modify the arrays IPARM or FPARM within an associated sequence of calls, or modify the contents of IWK or FWK, except as requested by IREQ. Also, as mentioned earlier, the references IWK and FWK must be to the same memory locations across associated calls.

If the IPARM variable MALLOC is set to 0, then the user may attempt to save IWK, FWK, etc. on disk in the middle of a run in order to complete the solution process at a later time. This will not work however for CM Fortran versions since back-end memory will be lost.

For examples of the use of the package at the reverse communication layer, see the *PCG Examples Manual*.

2.4 Solving Multiple Linear Systems

In many applications it is desirable to solve a sequence of linear systems for which the matrix entries are unchanged or changed slightly but the right hand side vector b may have totally different entries. In such cases it may be desirable to reuse the matrix-vector product initialization or preconditioner calculation generated for the first solve in subsequent solves. This option is available for top-level calls to PCG.

The description of IJOB given in Section 2.1 gives a basic description of how to perform multiple solves. The following describes the general procedure:

- The package should first be called with IJOB = JINIT or JINITA. These calls apply matrix preprocessing options, perform precomputations required for the matrix-vector product, and calculate the preconditioner based on that matrix. If it is expected that the matrix entries will change for future calls but it will be desired to use the preconditioner based on the matrix supplied by this call, then IJOB=JINITA should be used, so that the package saves an internal copy of A . This is necessarily for preconditioners such as polynomial preconditionings which require A for their application. Otherwise, IJOB=JINIT can be used, since this value of A will be available for all future calls, for preconditioners which make use of its entries directly. In any case, the result of this initialization is referenced by the IPARM variable IPTR.
- A call using IJOB=JRUN following an initialization call simply applies the matrix and preconditioner from the initialization call to the given vector b . The values of IA, JA and A must match those of the initialization call. This call may be repeated for different values of U, UEXACT and B, but these vectors must not change in their specifications from those given in the initialization call.

- A call using IJOB=JRUNA following an initialization call solves the system with a different value of A from the initialization call but the same preconditioning. The arrays IA and JA must match those used for the initialization call, but A, U, UEXACT and B can have different entries though otherwise their specifications (size, layout, etc.) must match those from the initialization call. It should be noted that this call does not change the matrix vector product setup or preconditioner calculation generated by the previous initialization call, so calls with IJOB=JRUN may be mixed with calls with this option.
- A call using IJOB=JRUNAQ following an initialization call solves the system with a different value of A as well as a new value of the preconditioner based on this new A . The arrays IA, JA, A, U, UEXACT and B need not match those from the initialization call in any way. Currently this call is not much different than calling the package with a separate initialization call, but in future versions of the package the computation of the new matrix vector product setup and preconditioner may be increased in speed by making use of the information from the previous initialization. It should be noted that this call does not change the matrix vector product setup or preconditioner calculation generated by the previous initialization call.
- A call using IJOB=JTERM following an initialization call terminates the result of the initialization call by releasing any allocated memory. It should be noted that this call is not necessary if the package terminated before this with an error termination.

The user should not change any entries of the IPARM or FPARM arrays across associated calls. Also, as mentioned earlier, the references IWK and FWK must be to the same memory locations across associated calls.

It should be noted that matrix preprocessing operations are always undone before any return from the package to the calling routine.

If the IPARM variable MALLOC is set to 0, then the user may attempt to save IWK, FWK, etc. on disk to solve other systems at a future time with the given setup. This will not work however for CM Fortran versions since back-end memory will be lost.

For examples of using the package to solve multiple linear systems, see the *PCG Examples Manual*.

Chapter 3

Adaptations for Particular Machines

Since some parallel computers support node-level Fortran with message passing, while others support a data-parallel style of Fortran with global address space, the usage of the package will vary slightly across machines. The purpose of this chapter is to describe these differences.

3.1 Use of Arrays

For versions of Fortran with a global address space, a single global copy of each array exists, and in particular, arrays like **A**, **U** and **B** refer to the global array and vectors. On message passing machines under the SPMD (single program, multiple data) model of programming, arrays exist as multiple, possibly differing copies across processors. In this case, variables such as **IJOB**, **IPARM**, **FPARM** and **IER** exist as multiple copies across processors and, unless otherwise noted, must have identical values across processors. On the other hand, arrays such as **IA**, **JA**, **A**, **U**, **UEXACT** and **B** may have different values across processors and are typically interpreted to refer to the subvector or submatrix located on that processor. The means by which the vectors and matrix are mapped to the processors is described in Chapter 5.

The user must insure that the values provided in **IPARM** and **FPARM** as input to the package are consistent across processors. Furthermore, values output from the package in these arrays are guaranteed to be consistent across processors. The few exceptions to these rules are documented in Chapter 4.

3.2 Processor Numbering

For message passing machines, processors are assumed to be numbered starting with processor zero and increasing to the number of processors minus one. For PVM and MPI versions, it is assumed that PCG is being run on a user-specified processor group, and that each processor is running exactly one instance of PCG associated with the particular solve. Other machines, e.g. Intel Paragon, assume that exactly one instance of PCG is being run on each allocated processor for the particular solve.

For PVM versions, a processor group is used by PCG for the solve, whose name is formed by concatenating the string "PCG" with the final 3 digits of the **IPARM** variable **ICOMM**, yielding by default "PCG000," for example. All package communication is made within this group with the

associated processor numbering in that group. This admits the ability to run PCG on a subset of the allocated processors. For PVM versions, it is assumed that before entering the package, all desired processors have already joined this group. Note that for the T3D/PVM version, the group numbers are used, not the processing element numbers; if the PE numbers must be used, the processors must be added to the group in the proper order to assure that PE number equals the group processor number. PCG uses its own PVM message passing buffers for communication to avoid the possibility of corrupting the user's buffers.

For MPI versions, the IPARM variable ICOMM denotes the (intraprocessor) communicator in which communication takes place. By default, the value is MPI_COMM_WORLD. To isolate PCG package communication from other communication, a separate communicator can be formed and passed to PCG via ICOMM.

For calls to the package at the reverse communication level or iterative method level, no explicit ordering relationship is assumed between the processor numbers and the subvectors resident on the processors. Any ordering relationship is implicitly imposed by the user's matrix vector product and preconditioning operations. For a top-level call to the package, the exact relationship between processor numbers and matrix and vector elements in the linear system is described in Chapter 5.

3.3 Communication on Message Passing machines

For message passing versions, it is important that the package and the user code not interfere with each other's message types used for communication. Two mechanisms are allowed to coordinate user and package message types: isolated and non-isolated communication. To use isolated communication, the user sets the IPARM variables MSGMIN and MSGMAX to a range of message types disjoint from those used by the user's code. The second option of non-isolated communication allows the user to access the same message type range used by the package via the IPARM variable MSGTYP described in Chapter 4, which denotes the next available message type which may be used for interprocessor communication. This variable can be used to coordinate message type usage between the package and the user's code, if desired. For MPI versions a third alternative is available, namely, usage of different communication contexts to isolate message type usage via the communicator specified in ICOMM.

Table 3.1: Default Values of MSGMIN and MSGMAX

Machine	MSGMIN	MSGMAX
Intel iPSC/860	0	926258176
Intel Paragon	0	926258176
Connection Machine 5, node-level	0	cmmd_max_user_tag
PVM	0	2147483647
MPI	*	2147483647

For message passing machines, the native synchronous or asynchronous communication system calls are used to accomplish interprocessor communication on the given machine. The IPARM variables MSGMIN and MSGMAX denote the range of message type values used by the package. The default values used for message passing machines are listed in Table 3.1. See Chapter 9 for minimal

acceptable requirements for the size of this range. By default, the variable `MSGTYP` is initialized to `MSGMIN` by `_DFALT`.

For the Intel iPSC/860 and Paragon versions, a corresponding range of forced type messages is also reserved for use by the package, obtained by adding a value of 2^{30} to the range of normal message types. The free message counter `MSGTYP` refers both to the untyped message of that number and the corresponding typed message obtained in this way.

The user may use any message type outside this range for communication outside the package, with no danger of ambiguity of sent or received messages. Alternatively, the user may use the variable `MSGTYP`, which denotes the next free message type available to the package for use. This variable may be used, for example, for communication before or after the package is called, or for communication to perform the matrix-vector product or preconditioning when the package is called at the iterative method level (by placing `IPARM` in a common block accessible by the user routines) or reverse communication level. In such cases, the user must guarantee that on entry to the package, via either a call to the package or a return to the package from a user-written routine, the values of `MSGTYP` are consistent across processors, and are within the range of values denoted by `MSGMIN` and `MSGMAX` (see Chapter 9).

The message type range should be set as large as possible, to avoid costly global synchronizations which are performed by the package when the `MSGTYP` counter is reset to `MSGMIN`. If the range is not sufficiently large for a required sequence of communications, an error return from the package occurs.

PCG uses message types rather than excessive synchronizations to insure that communications take place correctly. It is the user's responsibility to insure that upon entry to PCG, no unreceived messages exist anywhere in the system which might have message types used by PCG, which might result in inaccurate results from PCG or deadlock of the system.

In cases in which the user code solves several unrelated linear systems at the same time with PCG, care should be maintained either that `MSGMIN` and `MSGMAX` are set to disjoint ranges for the unrelated solves, or that message type usage is properly coordinated manually between the different solves. For MPI versions, these problems may be circumvented straightforwardly by use of different communication contexts.

For the Intel Paragon version, the values returned by the system call `MYPTYPE` must be the same across copies of PCG running across processors.

The user is responsible for not upsetting the consistency of `MSGTYP` across processors, since such errors are difficult to detect and typically result in deadlocked processes.

3.4 Input and Output

The standard Fortran `write` statement is used for package output, using the I/O unit specified by the `IPARM` variable `NOUT`. For message passing machines, output is written from processor node zero only. Some specific message passing machines have I/O modes which may be set by the package and restored upon exit.

Depending on the `IPARM` variable `LEVOUT`, package error messages are printed to output. However, errors may be difficult to detect on some message passing machines, for example when an arithmetic error or other problem occurs on a node, causing the system to hang due to other processors waiting for communication from that node. Nonetheless, every effort is made for the package to cause an error return on all nodes whenever there is an error on any node.

3.5 Memory Allocation

The PCG package has two options for allocation of internal workspace memory. The first option is to use the arrays **IWK** and **FWK** to allocate memory. In this case, memory locations throughout these arrays are used for allocating temporary memory for the package. An alternate means of allocation, available for all versions except the generic Fortran 77, Fortran 77 / PVM and Fortran 77 / MPI versions, is to use automatic memory allocation, by means of the **malloc** function accessed through a C language interface. This is the default for these machines. The **IPARM** variable **MALLOC** is used to control which means of allocation is used. Other **IPARM** variables are used to indicate the amount of memory allocated, and so forth.

When the **malloc** function is used for memory allocation, **IWK** and **FWK** should be arrays of length one. Furthermore, **IWK** must be of type integer, and **FWK** must be of the same precision and arithmetic as is specified by the call to the package, to insure proper alignment on multi-word boundaries. Also, these length-one arrays are actually used for temporary storage by the package, so they should not be set to arbitrary dummy arrays.

When not using the **malloc** function, it is typical to call the package with an overestimate of the required workspace as specified in **NWI** and **NWF**, and then use the returned values **NWIUSD** and **NWFUSD** to set more economical values of **NWI** and **NWF** for future calls under the same conditions. See Chapter 9 for specific memory requirements for different methods.

3.6 Subroutine Naming

To assure Fortran 77 compatibility, all package subroutine names are limited to 6 characters in length. The letters **S**, **D**, **C** and **Z** are used as the first letter of precision- and arithmetic-specific routines of the package. The initial letter **X** is used for internal precision-independent utility routines such as output routines. The first letter **I** is used for certain integer-valued functions in the package. No other first letters are used for package routines. The same remarks apply to package routines written in other languages for some machines, such as **C** or assembler routines.

The PCG code is re-entrant and uses no common blocks or **SAVE** statements which can have an adverse effect on solving a variety of different systems by different calls to the package in different contexts in the same code.

3.7 Precisions and Arithmetics

The subroutine argument list for a given routine is identical across versions of the routine for different precisions and arithmetics. For a call to the package for a given precision and arithmetic, all floating point parameters passed in to the package must have that precision and arithmetic. All calls to the package to solve a given linear system of given precision and arithmetic must have consistent precision and arithmetic type, though the same user program may solve unrelated systems of different precision/arithmetic type by mixed calls to the package. The double precision complex versions of the package use "DOUBLE COMPLEX" to declare variables of this type.

Chapter 4

Parameter Arrays IPARM and FPARM

4.1 The Use of Parameter Arrays

The arrays **IPARM** and **FPARM** are used to communicate integer and floating point parameters to and from the package for control of the iteration process. A large number of parameters are made available, to allow the experienced user careful control of the iterative process. The beginning user should use the `_DFALT` routine to set default settings of these arrays and consult the final section of this chapter to determine the most important parameters to modify in a given application.

The two arrays are each divided into three sections: a reverse communication level section, an iterative method level section, and a top level section. A call to the package at any level reads and writes the parameters for that level as well as those for lower levels. That is, when the package is called from the top level, all sections are accessed; when called at the iterative method level, the iterative method and reverse communication sections only are accessed; and, when called at the reverse communication level, only the reverse communication section is accessed.

As mentioned in Section 2.1, the routine `_DFALT` can be used to set the arrays to default values. The **IPARM** array is currently of size 50, and the **FPARM** array is of size 30. These values are represented by the mnemonics **NIPARM** and **NFPARM** in the include file `pcg_fort.h`. The system header routine `pcg_fort.h` is supplied with the PCG package for Fortran compilers which implement an `INCLUDE` facility. This header file gives parameters which facilitate access into the **IPARM** and **FPARM** arrays. For example,

```
IPARM(IQSIDE) = QNONE
FPARM(ZETA) = .003
```

specifies that the **IPARM** variable **IQSIDE** is set to **QNONE** (= 0) as defined in the header file, and the **FPARM** variable **ZETA** is set to 0.003. Check with your local installation to find out how to access this header file.

It is recommended that the user code make use of the symbolic **IPARM** and **FPARM** definitions from this include file, to insure greater compatibility with future releases of the package.

The values of **IPARM** and **FPARM** variables must be consistent across processors for message passing versions, and the values for a call must be consistent with those values for any associated

previous initialization call to the package. The only exceptions to this rule are the timer and floating point operation entries on **FPARM** and the variables **NRU**, **NWI**, **NWF**, **NWIUSD**, **NWFUSD** and **IPTR** in **IPARM**, since these values have an interpretation which refers to the given processor.

Below is a complete list of the parameters, including for each one the name, location in the **IPARM** or **FPARM** array, default value and description. Available mnemonics for possible settings of the given parameter are listed in parentheses when appropriate. At the end of this chapter is a list of the more important parameters which are commonly modified for typical applications.

4.2 Integer Parameters, Reverse Communication Level

IPARM(1) NOUT (6) (Input) The Fortran input/output unit number to be used for output from the package.

IPARM(2) LEVOUT (0) (Input) The level of verbosity of output from the package. Higher numbers indicate progressively larger amounts of information sent to output. Possible settings are:

=0 (**LEVO**) : no output

=1 (**LEVERR**) : fatal error messages only

=2 (**LEVWRN**) : fatal and warning messages

=3 (**LEVIT**) : per-iteration information to monitor the convergence of the iterative method.

=4 (**LEVPRM**) : informative parameter information at the beginning and end of the run

=5 (**LEVALG**) : quantities related to the calculations of the algorithms.

IPARM(3) NRU (1) (Input, Output) For reverse communication or iterative method level calls, this parameter indicates the number of rows of the vector **U**. For message passing versions, it is the number of elements on the given processor; for all other versions, it is the total number of unknowns in the system. It is assumed that a vector is stored in memory as a contiguous sequence of elements, without redundancy or gaps. For calls at these levels, **NRU** can be different for each processor, for message passing versions. For top level calls, this value need not be set by the user, as it is filled out by the package from the given matrix information.

IPARM(4) ITSMAX (500) (Input) The maximum permissible number of iterations to be allowed for the iterative method.

IPARM(5) ITS (0) (Output) The actual number of iterations required by the iterative method.

IPARM(6) MALLOC (*) (Input) When set to 1 (**YES**), memory allocation is performed internally by calls to the C language **malloc** facility. This is available and is the default for all versions except for Fortran-only versions. When **MALLOC=1**, **IWK** and **FWK** need only be of length 1, though they must be arrays of the correct precision and arithmetic types. When set to 0 (**NO**), all memory is allocated within the user-supplied workspace arrays. This setting is required and is the default for Fortran-only versions.

IPARM(7) NWI (1) (Input) The number of integer words being supplied by the user in the array IWK. For message passing versions, it is the number of elements supplied on the given processor.

IPARM(8) NWF (1) (Input) The number of floating point values being supplied by the user in the array FWK. For message passing versions, it is the number of elements supplied on the given processor.

IPARM(9) NWIUSD (0) (Output) When MALLOC= 0 and NWIUSD is not greater than NWI, this is the largest number of integer workspace values of IWK that were in use at any given time. Thus a subsequent call of the package under identical circumstances with this value for NWI would result in adequate workspace for the call. When MALLOC= 1 and NWIUSD is not greater than NWI, this value represents the largest amount of internally allocated integer memory at any given time. If NWIUSD is greater than NWI, then this value denotes the total number of integer values the package was attempting to use when available memory was exhausted. It should be noted that for CM Fortran versions, this and the other values related to memory management refer only to front-end memory.

IPARM(10) NWFUSD (0) (Output) When MALLOC= 0 and NWFUSD is not greater than NWF, this is the largest number of floating point workspace values of FWK that were in use at any given time. Thus a subsequent call of the package under identical circumstances with this value for NWF would result in adequate workspace for the call. When MALLOC= 1 and NWFUSD is not greater than NWF, this value represents the largest amount of internally allocated floating point memory at any given time. If NWFUSD is greater than NWF, then this value denotes the total number of floating point values the package was attempting to use when available memory was exhausted. It should be noted that for CM Fortran versions, this and the other values related to memory management refer only to front-end memory.

IPARM(11) IPTR (*) (Input, Output) A pointer used by the package to reference the memory allocated by an initialization call to the package. It is required by subsequent calls which make reference to this initialization, up to the termination call. By use of different copies of this parameter, it is possible for a single code to solve totally different problems without the need to terminate one problem before starting another. Its default value is the same as the value of MALLOC, which is used internally as a null pointer.

IPARM(12) NTEST (-1) (Input) The value of the stopping test. For a complete discussion of stopping tests, see Section 8.1.

IPARM(13) IQSIDE (1) (Input) Indicates the basic placement of the preconditioner. Allowable values are:

- =0 (QNONE) : preconditioning not used ($Q_L = Q_R = I$)
- =1 (QLEFT) : left preconditioning ($Q_L = Q, Q_R = I$)
- =2 (QRIGHT) : right preconditioning ($Q_L = I, Q_R = Q$)
- =3 (QSPLIT) : two-sided or split preconditioning

IPARM(14) IUINIT (-1) (Input) Specifies the value of the initial guess U on input to the package. Admissible values are:

- =-2 (USZERO) : the vector U is uninitialized and will be initially set to zero by the package
- =-1 (DFALT) : interpret IUINIT based on the iterative method selected. For BCG-type methods, including the QMR-type methods, perform the action of USRAND; otherwise interpret as USZERO. Note in either case it is not necessary for the user to set U before calling the package.
- = 0 (UZERO) : the vector has already been set to zero
- = 1 (UNZERO) : the vector has been initialized to a nonzero value
- = 2 (USRAND) : set $u^{(0)}$ to a vector of random entries scaled so that $\|Au^{(0)}\| = (.1)\|b\|$. This technique has the advantage of smoothing the convergence of many biconjugate gradient type methods.
- = 3 (UPRAND) : perturb the user-supplied $u^{(0)}$ by a vector v of random entries scaled so that $\|Av\| = (.1)\|b - Au^{(0)}\|$. This technique has the advantage of smoothing the convergence of many biconjugate gradient type methods.

IPARM(15) NEEDRC (0) (Input) When the package is called at the reverse communication level, this flag denotes the vectors and other values the user wishes the package to supply whenever the package requests a stopping test to be performed by the user (when NTEST=TSTUSR and IREQ=JTEST). Each bit of NEEDRC is set to YES= 1 or NO= 0 to denote whether the reverse communication routine is required to supply the requested quantity. In particular,

$$\text{NEEDRC} = 1 \cdot \text{NU} + 2 \cdot \text{NR} + 4 \cdot \text{NQLR} + 8 \cdot \text{NQRQLR},$$

where:

- NU is YES= 1 or NO= 0 based on whether $u^{(n)}$ is to be supplied,
- NR is YES= 1 or NO= 0 based on whether $r^{(n)}$ is to be supplied (through the argument IVA),
- NQLR is YES= 1 or NO= 0 based on whether $Q_L r^{(n)}$ is to be supplied (through the argument IVQL), and
- NQRQLR is YES= 1 or NO= 0 based on whether $Q_{RQ_L} r^{(n)}$ is to be supplied (through the argument IVQR).

See Section 2.3 for a more complete discussion.

IPARM(16) NS1 (10) (Input) For truncated iterative methods such as Orthomin, this quantity indicates the truncation factor for the method, which basically indicates how many previous direction vectors are to be saved for use by the method.

IPARM(17) NS2 (10) (Input) For the restarted methods such as GMRES, this indicates the restart frequency for the method.

IPARM(18) ICKSTG (-1) (Input) When set to 1 (YES), this flag causes the package to terminate if stagnation is detected, i.e., if the value of the stopping criterion does not change significantly over a large number of iterations. This test can be used to save wasted computation time when convergence cannot be attained. If set to 0 (NO), this test is not performed. When set to -1 (DFALT), the default action is taken based on which iterative method is

used: stagnation checking is performed for all iterative methods except for QMR-type methods, which often stagnate for a large part of the run before successful convergence.

IPARM(19) IUEXAC (0) (Input) A value of 1 (YES) indicates that the vector UEXACT does indeed contain the true solution u to the linear system and thus may be used for certain computations such as error checking; otherwise 0 (NO) is used to denote that UEXACT should not be used by the package.

IPARM(20) IDOT (1) (Input) For versions of the package which use complex arithmetic, a value of 1 (YES) indicates that true inner products are to be used by the iterative method, i.e. $x^*y = \bar{x}^T y$, where \bar{x} is the complex conjugate of x . If set to 0 (NO), then the pseudo-inner product $x^T y$ is used. Inner product calculations for the stopping test are unaffected by this parameter.

IPARM(21) ISTATS (0) (Input) If set to 1 (YES), the package computes certain quantities at the end of the iteration process, such as the true relative residual $\|r^{(n)}\|/\|r^{(0)}\|$ and, if possible, the true relative error $\|e^{(n)}\|/\|e^{(0)}\|$. These quantities require extra work to compute but may be used to determine how many digits of accuracy in the solution were produced by the iterative method. If this information is not desired, then this parameter is set to 0 (NO). Setting this value to 0 typically results in slightly lower run times.

IPARM(22) ITIMER (0) (Input) For Connection Machine versions, the number of the timer to be used by the package for timing purposes. If set to a value outside the admissible range of CM timers, then no timing is performed. The user must not tamper with this timer while a call to the package is being performed, e.g. within a user-supplied matrix-vector product routine. For versions other than the Connection Machine version, this variable has no effect.

IPARM(23) ICOMM (*) (Input) For PVM and MPI versions, a specifier to indicate the processor group within which PCG is being run and communication must occur. The use of this variable is described in Chapter 3. For PVM versions, the default value is 0; for MPI versions, the default value is MPI_COMM_WORLD.

IPARM(24) MSGMIN (*) (Input) For message passing versions, the minimal allowable message type to be used by the package. See Chapter 3 for a discussion of message passing and a list of default settings for different versions.

IPARM(25) MSGMAX (*) (Input) For message passing versions, the maximal allowable message type to be used by the package for interprocessor communication for the associated call. See Chapter 3 for a discussion of message passing and a list of default settings for different versions.

IPARM(26) MSGTYP (*) (Input) For message passing versions, the next available message type which may be used by the package for interprocessor communication for the associated call. See Chapter 3 for a discussion of message passing and a list of default settings for different versions.

IPARM(27) ICLEV (0) (Input) An internal variable for the package which must be set to 0 by the user.

IPARM(28) - IPARM(30) Reserved for future expansion.

4.3 Integer Parameters, Iterative Method Level

IPARM(31) - IPARM(40) Reserved for future expansion.

4.4 Integer Parameters, Top Level

IPARM(41) ISCALE (1) (Input) If this option is selected, then the package applies scaling to the linear system before applying the preconditioning and the iterative method, and removes the scaling on exit from the package. The matrix and vectors are unscaled to their original state (up to roundoff error) upon any return from a top-level call. See Section 6.1 for a full description of choices for this parameter.

IPARM(42) ICTRAN (1) (Input) For complex arithmetic versions, this parameter controls the calculation of the transpose-matrix vector product which is used by iterative methods which require them. A value of 1 (YES) indicates that the conjugate transpose of A is used: $v \leftarrow A^*u$. A value of 0 (NO) indicates that simply the transpose is used: $v \leftarrow A^T u$. This switch also affects the preconditioner: it controls whether Q_L^*/Q_R^* or Q_L^T/Q_R^T are used, in an analogous way. This parameter does not affect calls to the package at the iterative method or reverse communication level; the user can create the same effect by appropriately defining the `mysuba` and `mysubq` routines. Normally, the user will select `ICTRAN=1`.

IPARM(43) - IPARM(50) Reserved for future expansion.

4.5 Floating Point Parameters, Reverse Communication Level

FPARM(1) CTIMER (0.) (Input, Output) This parameter is incremented by the elapsed amount of CPU time spent in the reverse communication layer of the package during the given call. It should be noted that for message passing machines, each processor supplies its own separate value for all timings.

FPARM(2) RTIMER (0.) (Input, Output) This parameter is incremented by the elapsed amount of real time spent in the reverse communication layer of the package during the given call. It should be noted that for message passing machines, each processor supplies its own separate value for all timings.

FPARM(3) FLOPSR (0.) (Input, Output) This parameter is incremented by the number of floating point operations performed in the reverse communication layer of the package during the given call. A floating point operation is understood to be a single or double precision operation on real quantities. For message passing versions, this is the number of operations executed on the given processor; for other versions, it is the total number of operations performed.

FPARM(4) ZETA (.0001) (Input) The value of ζ to be used for the stopping test, as described in Section 8.1.

FPARM(5) STPTST (1.) (Output) The actual final value of the stopping criterion.

FPARM(6) ALPHA (1.) (Input) For the BAS iterative method (see Chapter 7), the extrapolation factor α to be used. Specifically, the method implemented is defined by: $u^{(n+1)} = u^{(n)} + \alpha Q_R Q_L r^{(n)}$.

FPARM(7) RELRSD (1.) (Output) If ISTATS=1, this is the final true relative residual $\|r^{(n)}\|/\|r^{(0)}\|$.

FPARM(8) RELERR (1.) (Output) If ISTATS=1 and IUEXAC=1, this is the final true relative error $\|e^{(n)}\|/\|e^{(0)}\|$.

FPARM(9) - FPARM(10) Reserved for future expansion.

4.6 Floating Point Parameters, Iterative Method Level

FPARM(11) CTIMEI (0.) (Input, Output) This parameter is incremented by the elapsed amount of CPU time spent in the iterative method and reverse communication layers of the package during the given call. Time within the matrix-vector product and preconditioning operations is included. It should be noted that for message passing machines, each processor supplies its own separate value for all timings.

FPARM(12) RTIMEI (0.) (Input, Output) This parameter is incremented by the elapsed amount of real time spent in the iterative method and reverse communication layers of the package during the given call. Time within the matrix-vector product and preconditioning operations is included. It should be noted that for message passing machines, each processor supplies its own separate value for all timings.

FPARM(13) FLOPSI (0.) (Input, Output) This parameter is incremented by the number of floating point operations performed in the iterative method and reverse communication layers of the package during the given call. Floating point operations within the matrix-vector product and preconditioning operations are included if the package is called at the top level; otherwise, they must be calculated by the user if they are desired. A floating point operation is understood to be a single or double precision operation on real quantities. For message passing versions, this is the number of operations executed on the given processor; for other versions, it is the total number of operations performed.

FPARM(14) - FPARM(20) Reserved for future expansion.

4.7 Floating Point Parameters, Top Level

FPARM(21) CTIMET (0.) (Input, Output) This parameter is incremented by the elapsed amount of CPU time spent in all layers of the package during the given call. It should be noted that for message passing machines, each processor supplies its own separate value for all timings.

FPARM(22) RTIMET (0.) (Input, Output) This parameter is incremented by the elapsed amount of real time spent in all layers of the package during the given call. It should be noted that for message passing machines, each processor supplies its own separate value for all timings.

FPARM(23) FLOPST (0.) (Input, Output) This parameter is incremented by the number of floating point operations performed in all layers of the package during the given call. A floating point operation is understood to be a single or double precision operation on real quantities. For message passing versions, this is the number of operations executed on the given processor; for other versions, it is the total number of operations performed.

FPARM(24) - FPARM(40) Reserved for future expansion.

4.8 Which Parameters to Set

For simple top level calls to the package with IJOB=0, to call the package successfully it is usually adequate to set only the following parameters:

- ZETA, the convergence tolerance
- ITSMAX, the maximal allowable number of iterations
- NWI/NWF, available workspace memory (only important if internal memory allocation cannot be used)

For calls at the lower levels, the following is also important:

- NRU, the number of unknowns of U (for message passing machines, how many on the given processor)

Chapter 5

Matrix Storage Modes

This chapter describes matrix formats accessible through a top-level call to the package. If the user desires to access the package with a format not listed here, Sections 2.2 and 2.3 may be consulted for means of accessing the iterative algorithms with user-defined matrix formats. The major format supplied with the current release of PCG is the Regular Grid Stencil format, which is applicable to problems derived from structured grids. A future release of the package will contain formats for more general problems. Each format is specified to the package by a two-letter designator, denoted generically “ff”.

5.1 The Regular Grid Stencil Format

The Regular Grid Stencil Format (denoted by the two-letter mnemonic “GR”) is designed to accommodate structured matrices arising from physical problems for which a regular topologically rectangular or toroidal grid is used. The Regular Grid Stencil format closely resembles the standard diagonal storage format used on scalar and vector machines (see [Saad 1990], [Oppe/Joubert/Kincaid 1988]) and contains it as a special case.

In brief, the Regular Grid Stencil format can be described as follows:

- A physical grid of dimension $NDIM$ is allowed. In the current implementation, $NDIM$ can range from 1 to 32, while for most cases of interest, $NDIM$ is 2 or 3.
- A topologically Cartesian grid is assumed. The variable NB specifies the number of degrees of freedom or unknowns at each grid point.
- A “stencil” is assumed at each grid point and is taken to be the same for each point of the grid. This stencil, which relates the given point to a total of $NSTEN$ grid points (typically the point itself and $NSTEN-1$ neighbors), is specified in terms of the relative grid offsets along each grid axis for each point of the stencil. The matrix then has up to $NB \times NSTEN$ nonzero elements per row.
- The grid is assumed to be toroidal or to “wrap around” on each axis, so that points at each edge may be related to the points at the opposite edge. A non-toroidal grid may be easily accommodated by setting to zero the appropriate relation coefficients at the boundaries.

- For message passing implementations, the user specifies the number of grid points of each processor's subgrid along each axis, which is the same for each processor, as well as the number of processors along each axis. For this case, a package routine for the mapping between processors and subgrids is supplied to which the user must conform. For data parallel, serial and shared memory versions, the user supplies only the number of elements of the global grid along each axis.
- A row-major ordering of the grid dimensions is assumed. It is assumed that the matrix is ordered such that the x -dimension of the global grid varies fastest, then the y -dimension, and so forth. Similarly, for message passing versions the processor subgrids are assumed to be ordered within the global grid with a natural ordering, with x varying fastest, then y , and so forth.

5.2 Array Specification

We now give a more thorough description of this format. A matrix in this format is specified by integer arrays IA and JA and a floating point array A. The dimensioning of vectors U, UEXACT and B compatible with this format are also described below. The array IA is an integer array of size 5. For the CM Fortran versions, it is a front end array. The entries are defined as follows:

- IA(1)=NDIM , the number of physical dimensions of the grid.
- IA(2)=NSTEN , the number of grid points in each stencil relation.
- IA(3)=NB , the number of degrees of freedom at each grid point.
- IA(4)=MAXORD, the ordering of the axes for the matrix stored in A.
- IA(5)=VAXORD, the ordering of the axes for vectors.

The values of IA(4) and IA(5) can be set to -1=DFALT to give a simple default ordering of the axes for the matrix and vectors. The interpretation for these default settings is machine-specific and is defined later in this chapter. The defaults are set to give performance as high as possible in typical situations for the given machine.

The choices for IA(4) and IA(5) are listed below. Using different settings for these in some cases gives higher performance for the matrix vector product operation. These added options may be safely ignored in favor of the default orderings, if desired. It should be noted that only the default ordering is available for CM Fortran versions.

Here, NBR and NBC denote the row and column dimensions corresponding to the number of degrees of freedom NB, and NG denotes the sequence of dimensions NXS, NYS, ... for the physical grid.

- IA(4)=ORRC SG= 0: A(NBR,NBC,NSTEN,NG)
- IA(4)=ORRC GS= 1: A(NBR,NBC,NG,NSTEN)
- IA(4)=ORR SCG= 2: A(NBR,NSTEN,NBC,NG)
- IA(4)=ORR SGC= 3: A(NBR,NSTEN,NG,NBC)
- IA(4)=ORR GCS= 4: A(NBR,NG,NBC,NSTEN)
- IA(4)=ORR GSC= 5: A(NBR,NG,NSTEN,NBC)

- IA(4)=ORCRSG= 6: A(NBC,NBR,NSTEN,NG)
- IA(4)=ORCRGS= 7: A(NBC,NBR,NG,NSTEN)
- IA(4)=ORCSRG= 8: A(NBC,NSTEN,NBR,NG)
- IA(4)=ORCSGR= 9: A(NBC,NSTEN,NG,NBR)
- IA(4)=ORCGRS=10: A(NBC,NG,NBR,NSTEN)
- IA(4)=ORCGSR=11: A(NBC,NG,NSTEN,NBR)
- IA(4)=ORSRCG=12: A(NSTEN,NBR,NBC,NG)
- IA(4)=ORSRGC=13: A(NSTEN,NBR,NG,NBC)
- IA(4)=ORSCRG=14: A(NSTEN,NBC,NBR,NG)
- IA(4)=ORSCGR=15: A(NSTEN,NBC,NG,NBR)
- IA(4)=ORSGRC=16: A(NSTEN,NG,NBR,NBC)
- IA(4)=ORSGCR=17: A(NSTEN,NG,NBC,NBR)
- IA(4)=ORGRCS=18: A(NG,NBR,NBC,NSTEN)
- IA(4)=ORGRSC=19: A(NG,NBR,NSTEN,NBC)
- IA(4)=ORGCRS=20: A(NG,NBC,NBR,NSTEN)
- IA(4)=ORGCSR=21: A(NG,NBC,NSTEN,NBR)
- IA(4)=ORGSRG=22: A(NG,NSTEN,NBR,NBC)
- IA(4)=ORGSCR=23: A(NG,NSTEN,NBC,NBR)
- IA(5)=ORRG=0: U(NB,NG)
- IA(5)=ORGR=1: U(NG,NB)

The array JA is an integer array dimensioned JA(NDIM,NSTEN+3). For the CM Fortran versions, it is a front end array. The entries are defined as follows:

- JA(IDIM,ISTEN), $1 \leq \text{IDIM} \leq \text{NDIM}$, $1 \leq \text{ISTEN} \leq \text{NSTEN}$: these values define the stencil. The value specified by IDIM and ISTEN gives the relative offset of stencil point ISTEN in the dimension specified by IDIM.
- JA(IDIM,NSTEN+1), $1 \leq \text{IDIM} \leq \text{NDIM}$: these values specify the grid size. For message passing versions, the value specified by IDIM gives the size of the subgrid on the given processor in that dimension. For all other versions, this value gives the size of the entire global grid in the given dimension. In either case, the size of the grid is measured in terms of the number of NB-size blocks.
- JA(IDIM,NSTEN+2), $1 \leq \text{IDIM} \leq \text{NDIM}$: these values are used only for message passing versions. In these cases, the value specified by IDIM gives the number of processors allocated along that dimension.
- JA(IDIM,NSTEN+3), $1 \leq \text{IDIM} \leq \text{NDIM}$: these values are used only for message passing versions. In these cases, the value specified by IDIM denotes the manner in which the processors are numbered on that axis. A value of 1 (=GRGRAY, using the mnemonic defined in the pcg_fort.h header file described in Chapter 4) denotes that the subdomains are mapped to the processors by a gray code mapping, and 0 (=GRBIN) denotes the identity map.

The dimensioning and layout of A, U, UEXACT and B for the various machines are now described. Basically, the vectors U, UEXACT and B each contain NB unknowns for each of the grid points, and A contains an NB×NB block for each of the NSTEN stencil relations at each grid point. We define

NXS, NYS, ... to indicate the subgrid sizes (in NB-size blocks) for a message passing version, and the global grid size for all other versions. The dimensioning statements shown below may be used; vectors UEXACT and B are dimensioned analogously to U.

- CMF versions: The axis ordering must be of the following form, corresponding to the choices IA(4)=0=ORRG and IA(5)=0=ORRCSG:

```

      DIMENSION A(NBR      ,NBC      ,NSTEN  , NXS, NYS, ...)
CMF$  LAYOUT   A(:SERIAL, :SERIAL, :SERIAL,      ,      , ...)
      DIMENSION U(NBR      ,      ,      , NXS, NYS, ...)
CMF$  LAYOUT   U(:SERIAL,      ,      ,      ,      , ...)

```

Alternate axis weightings are permitted, but for best performance, the leading dimensions pertaining to NB and NSTEN should be serial, and the other axes for the array and vectors should have matching weights, so that, for example, A(:, :, :, IX, IY, ...) and U(:, IX, IY, ...) are properly aligned. The arrays must all be back-end arrays.

- All other versions: The following is the default ordering, which corresponds to IA(4)=1=ORGR and IA(5)=18=ORGRCS:

```

      DIMENSION A(NXS, NYS, ..., NBR, NBC, NSTEN)
      DIMENSION U(NXS, NYS, ..., NBR)

```

In all cases, if NB=1, then any of the axes involving NB may be omitted.

The definition of this matrix format is illustrated below by a code section which performs the matrix vector product for the Cray YMP case, with NDIM=2. The modulus function IMODF is defined below. The actual code in the package is much more heavily optimized; this example is included only for illustrative purposes.

```

DO IX = 1, NXS
  DO IY = 1, NYS
    DO IBR = 1, NB
      B(IX,IY,IBR) = 0.
      DO IBC = 1, NB
        DO ISTEEN = 1, NSTEN
          B(IX,IY,IBR) = B(IX,IY,IBR) + A(IX,IY,IBR,IBC,ISTEEN)*
&          U(IMODF(IX+JA(1,ISTEEN),NXS),
&          IMODF(IY+JA(2,ISTEEN),NYS),IBC)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO

```

5.3 Processor Embedding for Message Passing Versions

A final remark is required to describe the processor mapping for message passing versions assumed by this matrix format: The subgrids are numbered lexicographically, in such a way that the x -dimension varies fastest, then the y -dimension, and so forth. Subgrid numbers start at zero and range up to a value of $\left[\prod_{IDIM=1}^{NDIM} JA(IDIM, NSTEN + 2) \right] - 1$. To map a subgrid to a processor, first an optional gray code is imposed along any or all of the grid axes. The resulting number, ranging from zero to $\left[\prod_{IDIM=1}^{NDIM} JA(IDIM, NSTEN + 2) \right] - 1$, denotes the processor number associated with that subgrid. Note that if no gray code is used, then the subgrid number as numbered by the lexicographical ordering equals the processor number.

The following routines IGRID and IPROC, map between the processor number on the given machine and the subgrid number. These routines are accessible by the user. They require input arrays of size $2*NDIM$, namely the array starting at $JA(1, NSTEN+2)$, which specify the number of processors along each axis as well as the encoding scheme for each axis, whether gray coding or non-encoded.

```
FUNCTION IGRID (IPROC,NDIM,NPAX)
*   The user should supply JA(1,NSTEN+2) in the array NPAX.
  INCLUDE 'pcg_fort.h'
  INTEGER NPAX(*)
  IGRID = 0
  ITMP = IPROC
  IPRDAX = 1
  DO 1 IAXIS = 1, NDIM
    IPRCAX = IMODF(ITMP,NPAX(IAXIS))
    IF (NPAX(NDIM+IAXIS) .EQ. GRGRAY) THEN
      IGRID = IGRID + IPRDAX*IBIN(IPRCAX)
    ELSE
      IGRID = IGRID + IPRDAX* (IPRCAX)
    ENDIF
    ITMP = (ITMP-IPRCAX)/NPAX(IAXIS)
    IPRDAX = IPRDAX*NPAX(IAXIS)
1  CONTINUE
  END

*
FUNCTION IPROC (IGRID,NDIM,NPAX)
*   The user should supply JA(1,NSTEN+2) in the array NPAX.
  INCLUDE 'pcg_fort.h'
  INTEGER NPAX(*)
  IPROC = 0
  ITMP = IGRID
  IPRDAX = 1
  DO 1 IAXIS = 1, NDIM
    IPRCAX = IMODF(ITMP,NPAX(IAXIS))
    IF (NPAX(NDIM+IAXIS) .EQ. GRGRAY) THEN
```

```

        IPROC = IPROC + IPRDAX*IGRAY(IPRCAX)
    ELSE
        IPROC = IPROC + IPRDAX*      (IPRCAX)
    ENDIF
    ITMP = (ITMP-IPRCAX)/NPAX(IAxis)
    IPRDAX = IPRDAX*NPAX(IAxis)
1  CONTINUE
    END
*
    FUNCTION IMODF ( I, N )
*  Modulus function based on "floor" instead of "truncation".
    IF (I .GE. 0) THEN
        IMODF = MOD(I,N)
    ELSE
        IMODF = I + ((-I-1)/N+1)*N
    ENDIF
    END

```

The functions IGRAY and IBIN are defined as gray code and reverse gray code mappings as shown below. The routine INEQV returns the bitwise logical exclusive-or of its arguments.

```

    FUNCTION IBIN (IGRAY)
    IBIN = IGRAY
    NRSGR = IGRAY/2
10  IBIN = INEQV (IBIN,NRSGR)
    NRSGR = NRSGR/2
    IF (NRSGR .NE. 0) go to 10
    END
*
    FUNCTION IGRAY (IBIN)
    NRSB = IBIN/2
    IGRAY = INEQV (IBIN,NRSB)
    END
*
    FUNCTION INEQV (M,N)
    MTEMP = M
    NTEMP = N
    INEQV = 0
    IPOW = 1
10  CONTINUE
        IF (MTEMP.EQ.0 .AND. NTEMP.EQ.0) GO TO 20
        MTEMP2 = MOD(MTEMP,2)
        NTEMP2 = MOD(NTEMP,2)
        IF (MTEMP2.EQ.0 .AND. NTEMP2.EQ.1) THEN
            INEQV = INEQV + IPOW

```

```
ELSE IF (MTEMP2.EQ.1 .AND. NTEMP2.EQ.0) THEN
  INEQV = INEQV + IPOW
ENDIF
MTEMP = MTEMP/2
NTEMP = NTEMP/2
IPOW = IPOW *2
GO TO 10
20 CONTINUE
END
```

Chapter 6

Matrix Preprocessing and Preconditioning

6.1 Matrix Preprocessing/Scaling

This section describes available preprocessing options in the PCG package. These user-selectable options are optionally applied to the matrix before the preconditioner is calculated and the iterative method is applied. These selections typically improve the iteration process.

It should be noted that whenever the PCG package is exited, the matrix preprocessing options are undone, restoring the matrix to its original form. In particular, after any exit from the top level of the package (and not merely a termination call), the matrix is restored to its original state. However, small changes in the matrix may occur, due to roundoff error. In the same way, the right hand side vector b and other vectors are converted to their appropriate values relevant to the original linear system upon exit from the package. The current release of PCG implements matrix scaling as a preprocessing option.

Matrix scaling is typically used to improve the condition number of the given matrix. Scaling is requested via the IPARM variable ISCALE (see Chapter 4). If this option is selected, then the package multiplies the original matrix $A = \{a_{ij}\}$ by diagonal matrices $D_L = \text{diag}\{d_{L,i}\}$ and $D_R = \text{diag}\{d_{R,i}\}$ to form the new system $(D_L A D_R)(D_R^{-1} u) = (D_L b)$ before the application of the preconditioning and the iterative method. It is typically desirable to apply scaling to the given matrix, to alleviate possible scaling variations in the particular equations or unknowns. Scaling typically improves the condition number of the matrix, which usually improves the convergence of the iterative method.

The following is a list of admissible scaling options:

ISCALE=0 (=SCNONE): no scaling is requested: $D_L = D_R = I$.

ISCALE=1 (=SCSD) (the default): split diagonal scaling is requested, with $d_{L,i} = d_{R,i} = 1/\sqrt{|a_{ii}|}$. This scaling is typically useful if the matrix is symmetric or nearly so (or in the complex case, nearly Hermitian). If the matrix is also positive definite, then the main diagonal entries are necessarily nonzero, so this scaling can be computed. On the other hand, if some main diagonal entries are nearly zero, then the scaling may result in poor convergence.

ISCALE=2 (=SCLD): left diagonal scaling is requested, with $d_{L,i} = 1/a_{ii}$ and $D_R = I$. This scaling may fail or hinder convergence if some main diagonal entries of A are nearly zero, and furthermore this scaling typically destroys symmetry or near-symmetry (or in the complex case, near-Hermitianness) if the matrix possesses one of these properties. On the other hand, this choice may be better than the ISCALE=SCSD choice for indefinite, nonsymmetric or complex matrices, since this scaling necessarily forces nonzero main diagonal entries of A to be 1, which may force the eigenvalues of the matrix into the right half of the complex plane, which improves convergence for many iterative methods.

ISCALE=3 (=SCLR): left rowsum scaling is requested, with $d_{L,i} = 1/\sum_j |a_{ij}|$, $D_R = I$. This scaling may be useful if some main diagonal entries of A are zero or nearly zero, making main diagonal scalings problematic. Note that this scaling typically destroys the symmetry or near-symmetry (or in the complex case, near-Hermitianness) of the matrix.

6.2 Matrix Preconditioning

This section describes preconditioners available in the PCG package. A list of available preconditioners is given, as well as guidelines on which ones to use for a given problem.

The preconditioning operators Q_L and Q_R defined in (1.2) are typically chosen so that the iterative method being used converges more quickly for the preconditioned operator $Q_L A Q_R$ than it does for the original matrix A . This is typically done by making $Q_L A Q_R$ as close as possible to the identity while at the same time requiring that it is computationally efficient to apply Q_L or Q_R to a vector.

A given preconditioner may be applied to the linear system (1) in several ways. If a matrix Q is available such that $QA \doteq I$ in some sense, then we may either let $Q_L = Q$ and $Q_R = I$ (left preconditioning), or alternatively $Q_L = I$ and $Q_R = Q$ (right preconditioning). Furthermore, in some cases a given preconditioner Q may naturally decompose into the product of matrices $Q = (LU)^{-1}$, where for example $L \doteq U^*$ when A is nearly symmetric (or, in the complex case, nearly Hermitian). In this latter case, it may be beneficial to select $Q_L = L^{-1}$, $Q_R = U^{-1}$ (split preconditioning).

For some iterative methods (e.g. Orthomin and GMRES), split preconditioning in such circumstances is of benefit, because in such cases if A is nearly symmetric, then the preconditioned matrix A is also, and the near-symmetry commonly improves the convergence. On the other hand, some iterative methods such as preconditioned CG are designed specifically to work with a Hermitian left preconditioner. The IPARM variable IQSIDE is used to control the placement of the preconditioner. In Chapter 7 recommendations are given for the placement of the preconditioner for the various iterative methods.

The name of a preconditioner is denoted by a two-letter mnemonic, denoted generically here as "pp". This name is coupled with a two-letter matrix format name for which the given preconditioner is implemented, to yield an external preconditioning routine name as described in Section 2.1.

We now describe the preconditioners currently supplied by the package:

RI: *Richardson's method preconditioner*. This is simply the null preconditioner ($Q_L = Q_R = I$) (available for all matrix formats).

JA: *Jacobi preconditioning.* To define this preconditioner, we let D be the diagonal matrix whose entries are the main diagonal entries of A . Then for left preconditioning we set $Q_L = D^{-1}$, and for right preconditioning $Q_R = D^{-1}$. In the case of split preconditioning, we let $Q_L = Q_R = (|D|)^{1/2}$, where $|D|$ is the matrix obtained by taking the elementwise moduli or absolute values of the elements of D . Note that it is an error for any diagonal entry of D to be zero or if a main diagonal matrix entry is missing. This preconditioning is available for all matrix formats and for all versions except for CM Fortran versions.

For many matrix problems, it is unclear beforehand which preconditioner will work best for the given problem, so some amount of trial and error may be required on the user's part in order to find a good preconditioner. However, the following general guidelines may help in this determination.

- *Jacobi.* For many problems, Jacobi preconditioning will be most cost-effective in terms of raw megaflop rates. This preconditioner can be computed for most matrix problems, it is highly vectorizable and parallelizable, and it usually improves the convergence of the iterative method by correcting possible scaling differences between the particular equations. However, in some cases, such as when a main diagonal element of the matrix is nearly zero, this preconditioner may degrade performance. To apply this preconditioner, it is usually most efficient to request diagonal matrix scaling as a preprocessing option (the default—see Chapter 4) and use the Richardson's method preconditioner, and `IQSIDE=QNONE`, rather than specifying Jacobi preconditioning directly.
- *Incomplete Factorizations.* When they can be computed, incomplete factorization preconditioners, such as incomplete Cholesky and modified incomplete Cholesky, often give dramatic reductions in the iteration counts and timings for solving linear systems. On the other hand, for a given matrix, these preconditioners may not be readily computable, and if they are computable, they may give poor convergence, particularly if a negative pivot is encountered during the approximate factorization process. Furthermore, these preconditioners are difficult to parallelize effectively. Modified incomplete Cholesky often gives better performance than incomplete Cholesky, but it is also more prone to forming negative pivots. If either factorization exists, however, it may give much better convergence than simple Jacobi preconditioning. However, this may still be offset by the additional computational effort for the incomplete factorization schemes. Incomplete factorization preconditioners will be made available in a future release of PCG.

Chapter 7

Choices of Iterative Method

This chapter describes available iterative methods (also known as iterative accelerators) in the PCG package. This includes a list of available iterative methods as well as guidelines for which method to use for a given type of problem.

7.1 General Considerations

An iterative method takes as input the linear system $Au = b$, initial guess $u^{(0)}$ and preconditioning operators Q_L and Q_R , and from these generates a sequence of iterates $\{u^{(i)}\}_{i>0}$. Other information may also be supplied to the iterative method, such as stopping test, convergence tolerance, frequency of restarting, and so forth. The iterative method may also provide auxiliary information such as residuals, convergence history, or eigenvalue estimates for the preconditioned operator.

Importantly, iterative acceleration techniques do not require explicit information concerning the structure of the matrices A , Q_L and Q_R but rather need only have information on how to apply these operators to a vector (and, in some cases, how to apply the conjugate transpose operator to a vector). This independence of iterative method from matrix format makes it convenient to supply the iterative method to the user to use in conjunction with a user-defined matrix format. The PCG package allows two levels of access for this: either a direct communication interface at the iterative method level, in which the matrix vector product operation and preconditioning operations are encapsulated into user-written subroutines with a specific argument list (see Section 2.2), or alternatively a reverse communication interface, in which the package iterative routine returns periodically to the user to request that these operations be done by the user (see Section 2.3).

The method $u^{(n+1)} = u^{(n)} + \alpha Q_R Q_L r^{(n)}$ is known as the extrapolated (unaccelerated) basic iterative method and is considered the simplest iterative method. Other iterative methods have more complex derivations and typically give considerably faster convergence than the unaccelerated basic iterative method.

Some iterative methods, such as biconjugate gradient, require also that the operator A^* be supplied, that is, that the user provide the operation of A^* times a vector. Furthermore, for these iterative methods, the Q_L^* and Q_R^* operators must be supplied, depending on the specification given by the IPARM variable IQSIDE; see Sections 2.2 and 2.3 for more details.

Most iterative methods compute as a by-product certain inner products which can be used to derive an inexpensive stopping test. The IPARM variable NTEST can be used to specify that this default stopping test for the given iterative method should be used. The definitions of the "default" stopping test for each iterative method are specified below; see also Section 8.1.

Most iterative methods also generate eigenvalue estimates for the operator $Q_L A Q_R$. These estimates may be used in connection with the stopping test in order to approximate a stopping criterion based on the scaled true error, such as $\|e^{(n)}\|/\|u\| \leq \zeta$. A future release of the PCG package will include eigenvalue estimators for the iterative methods.

7.2 Available Iterative Methods

The PCG package contains the following iterative methods. Their names are referred to generically as meth. These may be specified either in the top-level call to the package, or by an iterative method level call or reverse communication level call as described in Sections 2.2 and 2.3. In each case, unless otherwise mentioned, the default stopping test is $\|Q_L r^{(n)}\|/\|Q_L r^{(0)}\| \leq \zeta$.

- _BAS :** *The extrapolated basic iterative method.* For this method, $u^{(n+1)} = u^{(n)} + \alpha Q_R Q_L r^{(n)}$, where α is the FPARM variable ALPHA. This method is guaranteed to converge in exact arithmetic if the eigenvalues of $Q_L A Q_R$ are strictly contained in the circle centered at $1/\alpha$ of radius $1/|\alpha|$.
- _CG :** *Conjugate gradient acceleration.* This is the standard preconditioned conjugate gradient method, with preconditioning matrix $Q_R Q_L$. The method is guaranteed to converge in exact arithmetic whenever A and $Q_R Q_L$ are Hermitian positive definite and ICTRAN=IDOT=1, in which case the quantity $(e^{(n)}, A e^{(n)})$ is minimized over all choices in the relevant shifted Krylov space. The default stopping test is $\sqrt{|(r^{(n)}, Q_R Q_L r^{(n)})|}/\sqrt{|(r^{(0)}, Q_R Q_L r^{(0)})|} \leq \zeta$; if A and $Q_R Q_L$ are not both Hermitian positive definite, then this stopping test may give faulty results.
- _BCG** *The biconjugate gradient method.* This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE). When the preconditioned operator $Q_L A Q_R$ is Hermitian, this method gives the same iterates as the unpreconditioned conjugate gradient method applied to $Q_L A Q_R$, at about double the work. The method works effectively for many non-Hermitian problems but may break down for such problems.
- _LRES** *The Lanczos/Orthores algorithm.* This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE). This is the three-term 2-sided nonsymmetric Lanczos process. The method gives the same iterates as biconjugate gradient in exact arithmetic and has roughly the same numerical properties.
- _CGS** *The biconjugate gradient squared method, CGS.* This method is related to the biconjugate gradient algorithm, does not require the transpose, and may require as little as half the total work as biconjugate gradient; however, it is usually more prone to numerical breakdown than biconjugate gradient.
- _CGST** *The biconjugate gradient squared method with stabilization, CGSTAB.* The method is similar to CGS but often gives improved convergence behavior compared to CGS.

- _CGS2** *The Bi-CGSTAB2 method.* The method is a modification to the CGSTAB method which generally improves the performance on matrices with complex eigenvalues.
- _SQMR** *Simplified QMR method.* This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE). This is the simple three-term QMR algorithm, without look-ahead. This method is similar to biconjugate gradient but typically gives smoother convergence behavior and fewer numerical breakdowns.
- _TFQF** *Transpose-free QMR method.* This QMR-type algorithm due to Freund is related to the standard QMR algorithm but does not require transpose matrix operations.
- _DMIN** *The truncated Orthomin method.* For this algorithm, the quantity $\|Q_L r^{(n)}\|$ is minimized over a Krylov space, when IDOT=1. The variable NS1 defines the truncation factor, the number of previous vectors used, so that for IDOT=1, NS1=0 gives a simple steepest descent type algorithm, and NS1=1 gives the preconditioned conjugate residual method when $Q_L A Q_R$ is Hermitian. In general, the algorithm is guaranteed to converge in exact arithmetic if the Hermitian part $((Q_L A Q_R) + (Q_L A Q_R)^*)/2$ of the preconditioned operator is positive definite. Truncated Orthomin is often useful when the preconditioned operator is nearly Hermitian, for which case NS1 should be set to 1 or greater. In general, increasing NS1 tends to improve the convergence but also requires more work per iteration.
- _ORES** *The truncated Orthores or Arnoldi method.* When IDOT=1, NS1=1 and $Q_L A Q_R$ is Hermitian, this method gives the same iterates as the conjugate gradient method applied to $Q_L A Q_R$. For nonsymmetric problems, increasing the truncation factor NS1 generally improves the convergence behavior but also requires more work per iteration.
- _IOM** *The incomplete orthogonalization method, IOM.* This algorithm gives the same iterates as Orthores in exact arithmetic, but often has better numerical properties. The parameter NS1 is the truncation factor; increasing this factor generally improves the convergence behavior but also requires more work per iteration.
- _GMRS** *The restarted GMRES algorithm.* This algorithm minimizes the quantity $\|Q_L r^{(n)}\|$ over the relevant Krylov space. The variable IDOT does not affect this algorithm. The variable NS2 specifies the restart frequency for this algorithm. The algorithm is guaranteed to converge in exact arithmetic if the Hermitian part $((Q_L A Q_R) + (Q_L A Q_R)^*)/2$ of the preconditioned operator is positive definite. Increasing NS2 typically increases the convergence rate but also increases the average work per iteration. It should be noted that when IQSIDE=QRIGHT, this method implements the flexible GMRES algorithm, FGMRES, due to Saad, which allows Q_R to vary between iterations. For such cases, the user should pick a stopping test that does not depend on Q_R .
- _GMRH** *The restarted GMRES algorithm, Householder reflection version.* This algorithm gives the same iterates as standard GMRES, but typically has improved numerical behavior, particularly when high-accuracy solution is required. The algorithm is a Householder reflection version due to Walker. It should be pointed out, though, that this algorithm is significantly more expensive than standard GMRES: no algorithmic shortcuts are used which might jeopardize the high-accuracy solution of the problem.

- _CGNR** *The conjugate gradient method applied to the normal equations, residual-minimizing version.* This algorithm applies standard preconditioned conjugate gradients to the normal equations system $[(Q_L A Q_R)^*(Q_L A Q_R)]Q_R^{-1}u = (Q_L A Q_R)^*Q_L b$ (or alternatively the system $[(Q_L A Q_R)^T(Q_L A Q_R)]Q_R^{-1}u = (Q_L A Q_R)^T Q_L b$ when ICTRAN=0). This algorithm minimizes $\|Q_L r^{(n)}\|$ over the relevant Krylov space, when ICTRAN=IDOT=1. This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE). This method is guaranteed to converge in exact arithmetic when the preconditioned operator is nonsingular; however, convergence may be slow or impossible in finite precision arithmetic due to roundoff error.
- _LSQR** *The LSQR normal equations algorithm.* This algorithm gives the same iterates as CGNR in exact arithmetic, but typically has improved numerical properties. This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE).
- _CGNE** *The conjugate gradient method applied to the normal equations, error-minimizing version (Craig's method).* This algorithm applies standard preconditioned conjugate gradients to the normal equations system $[(Q_L A Q_R)(Q_L A Q_R)^*][(Q_L A Q_R)^{-*}Q_R^{-1}u] = Q_L b$ (or $[(Q_L A Q_R)(Q_L A Q_R)^T][(Q_L A Q_R)^{-T}Q_R^{-1}u] = Q_L b$ when ICTRAN=0). This algorithm minimizes $\|Q_R^{-1}e^{(n)}\|$ over the relevant Krylov space, when ICTRAN=IDOT=1. This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE). This method is guaranteed to converge in exact arithmetic when the preconditioned operator is nonsingular; however, convergence may be slow or impossible in finite precision arithmetic due to roundoff error. When the preconditioned matrix has a large condition number, this algorithm does a better job than CGNR at minimizing an error rather than a residual.
- _LSQE** *The LSQE normal equations algorithm.* This algorithm gives the same iterates as CGNE in exact arithmetic, but typically has improved numerical properties. This method requires use of the A^* operator (and Q_L^* and Q_R^* also, depending on IQSIDE).

7.3 Guidelines for Choosing an Iterative Method

When A is Hermitian positive definite, and a preconditioner Q is also available which is HPD, then the conjugate gradient method with $Q_L = Q$, $Q_R = I$ may be used. Other methods which minimize other norms of the error may be considered in some cases. An example is the conjugate residual method. Furthermore, techniques such as polynomial preconditioning may decrease the total amount of work required, particularly on some parallel computers for which inner products are expensive.

When A is only Hermitian but Q is HPD (e.g. $Q = I$), then the conjugate residual method may be used. This can be implemented with Orthomin using NS1=1. If the preconditioned matrix $Q_L A Q_R$ is nearly Hermitian, then truncated Orthomin acceleration with NS1 \geq 1 is typically useful. This method reduces to the conjugate residual method for $Q_L A Q_R$ when this matrix is Hermitian, and even for a general nonsingular matrix the method guarantees that $\|Q_L r^{(n)}\|$ is nonincreasing. Increasing NS1 generally improves the convergence but also requires more work per iteration.

The restarted GMRES algorithm converges whenever the Hermitian part of $Q_L A Q_R$, namely $[(Q_L A Q_R) + (Q_L A Q_R)^*]/2$, is positive definite, which requires that the eigenvalues of $Q_L A Q_R$ all have positive real part. This method always guarantees that $\|Q_L r^{(n)}\|$ is nonincreasing. Choosing

the restart frequency $NS2$ to be larger generally gives better convergence but requires more work per iteration. Restarted GMRES may also converge for non-Hermitian indefinite problems, if $NS2$ is set sufficiently large.

For many non-Hermitian problems, particularly non-Hermitian indefinite problems, variants of the biconjugate gradient method give best convergence. Methods such as biconjugate gradient, CGSTAB2 and QMR often perform well. However, some experimentation with these methods is often necessary in order to find the best method for the particular class of problems.

The conjugate gradient method applied to the normal equations is guaranteed to converge for every nonsingular matrix $Q_L A Q_R$ in exact arithmetic; however, the convergence is often very slow, or may even be impossible in finite precision arithmetic, due to the fact that the normal equations system has squared condition number compared to the original system.

Chapter 8

Stopping Tests and Error Conditions

8.1 Stopping Tests

The IPARM variable NTEST specifies a particular stopping criterion to test convergence of the iterative method. In addition to this criterion, all iterative methods in the package terminate when the number of iterations performed attains the number specified by the IPARM variable ITSMAX. When selected, these variables specify that the iterative method is to terminate when $STPTST \leq ZETA (\equiv \zeta)$ is satisfied, where STPTST and ZETA are the relevant FPARM variables. The definition of STPTST is controlled by NTEST and denotes the stopping test to be used.

If NTEST is set to 0=TST0, then no stopping test is performed, except for the test based on ITSMAX. This is useful if the reverse communication layer of the package is being accessed and a user-defined stopping test is performed (see Section 2.3). Take note, however, that many iterative methods break down or diverge if iterated past convergence, so it is desirable that some test be made for convergence in order to avoid such situations.

Admissible values for NTEST are:

- =-3 (TSTUSR) : only permissible when calling the package at the reverse communication level. In this case, the reverse communication routine allows the user to perform a stopping test by sending back a message of the form IREQ=JTEST and supplying the vectors requested by the user via the IPARM variable NEEDRC.
- =-2 (TSTEXA) : use the exact stopping test, with $STPTST = \|u^{(n)} - u\|/\|u\|$. Requires IUEXAC=1. This option is mainly used for testing purposes, since the exact solution is required.
- =-1 (TSTDFA) : use the "default" stopping test for the iterative method. This stopping test depends on the iterative method and is typically computationally inexpensive to compute. See Chapter 7 for a definition of the default stopping test for each iterative method.
- =0 (TST0) : no stopping test performed.
- =1 (TSTSE) : the Scaled Error stopping test:
 $STPTST = \|u^{(n)} - u\|/\|u\|$. Requires IUEXAC=1.
- =2 (TSTSR) : the Scaled Residual stopping test:
 $STPTST = \|r^{(n)}\|/\|b\|$.

=3 (TSTSLR) : the Scaled Q_L -preconditioned Residual stopping test:

$$\text{STPTST} = \|Q_L r^{(n)}\| / \|Q_L b\|.$$

=4 (TSTSR) : the Scaled $Q_R Q_L$ -preconditioned Residual stopping test:

$$\text{STPTST} = \|Q_R Q_L r^{(n)}\| / \|Q_R Q_L b\|.$$

=5 (TSTRE) : the Relative Error stopping test:

$$\text{STPTST} = \|u^{(n)} - u\| / \|u^{(0)} - u\|. \text{ Requires IUBHAV=1.}$$

=6 (TSTRR) : the Relative Residual stopping test:

$$\text{STPTST} = \|r^{(n)}\| / \|r^{(0)}\|.$$

=7 (TSTRLR) : the Relative Q_L -preconditioned Residual stopping test:

$$\text{STPTST} = \|Q_L r^{(n)}\| / \|Q_L r^{(0)}\|.$$

=8 (TSTRRR) : the Relative $Q_R Q_L$ -preconditioned Residual stopping test:

$$\text{STPTST} = \|Q_R Q_L r^{(n)}\| / \|Q_R Q_L r^{(0)}\|.$$

For most cases, the default stopping test (NTEST=TSTDFA) provides a reasonable measure of convergence. For many applications it is desirable that the scaled error $\|u^{(n)} - u\| / \|u\|$ be small. If a good preconditioner is available so that $Q_R Q_L A$ is well-conditioned, then the scaled $Q_R Q_L$ -preconditioned residual stopping test (TSTRRR) will approximate the scaled error test, since $\|u^{(n)} - u\| / \|u\| \leq \text{cond}(Q_R Q_L A) \cdot \|Q_R Q_L r^{(n)}\| / \|Q_R Q_L b\|$, where $\text{cond}(Q_R Q_L A) = \| [Q_R Q_L A] \| \cdot \| [Q_R Q_L A]^{-1} \|$ denotes the condition number. The parameter ZETA may be chosen to account for the effect of this condition number.

8.2 Error Conditions

The parameter IER is used by PCG to communicate to the user warning and error messages incurred during system solution. An IER value less than zero denotes a fatal error, causing an immediate return from the package (after various cleanups), and a value greater than zero denotes a warning, which does not cause termination. A value of zero denotes no error or warning.

If several warnings occur during the call, or a warning is followed by an error, then IER will reflect the most recent warning or error encountered. In such cases, the IPARM variable LEVOUT may be used to output messages for all warnings and errors encountered during the call. Using this parameter may also give more descriptive warning and error information than indicated in the lists given below.

The following are possible values of the error code IER that denote fatal errors:

IER=-1 *Unknown error.* A fatal system error external to the package has occurred.

IER=-2 *Insufficient integer workspace.* The package was unable to allocate adequate integer workspace (either in IWK or system memory, depending on the setting of IPARM variable MALLOC). The IPARM variable NWIUSD indicates the total number of values being requested when the package exited.

- IER=-3 *Insufficient floating point workspace.* The package was unable to allocate adequate floating point workspace (either in FWK or system memory, depending on the setting of IPARM variable MALLOC). The IPARM variable NWFUSD indicates the total number of values being requested when the package exited.
- IER=-4 *Invalid subroutine argument used.* An argument such as IJOB has been set to an illegal value.
- IER=-5 *Inadmissible i/fparm parameter.* One of the values in IPARM or FPARM is not within the required range or is in conflict with some other setting.
- IER=-6 *Breakdown in iterate calculation.* The formation of the iterate $u^{(n)}$ was impossible due to some condition such as division by zero in the algorithm.
- IER=-7 *Breakdown in basis vector calculation.* The formation of a vector used to form $u^{(n)}$ was impossible due to some condition such as division by zero in the algorithm.
- IER=-8 *Iterative method has stagnated.* The stopping criterion specified has not changed appreciably over a number of iterations.
- IER=-9 *Missing or zero main diagonal matrix entry.* An operation was requested which requires that all the diagonal entries of the matrix A be nonzero, which condition was violated.
- IER=-10 *Error in performing stopping test.* The stopping test could not be performed, due to some condition such as division by zero or an attempt to take the square root of a negative number in a context in which this is not permitted.
- IER=-11 *Error in matrix specification.* The specification of the matrix supplied by the user does not conform to the guidelines given in Chapter 5.
- IER=-12 *Error in interprocessor communication.* An error was encountered when using interprocessor communication routines.
- IER=-13 *Error in input/output.* An error was encountered when attempting to perform output from the package.
- IER=-14 *Error in memory allocation.* An error was encountered in the package routines which perform memory allocation.
- IER=-15 *Error in array specification.* An invalid CM Fortran back-end array descriptor was encountered.
- IER=-16 *Error in computing preconditioner.* An error such as division by zero has been encountered in computing the preconditioner.
- IER=-17 *Error in matrix vector product.* An error has occurred in the setup or application of a matrix vector product.

The following are possible values of the error code IER that denote nonfatal warnings:

- IER=1 *Unknown warning.* A nonfatal system error external to the package has occurred.

- IER=2 *Failure to converge in ITMAX iterations.* The convergence criterion was not satisfied within the specified maximum number of iterations.
- IER=3 *ZETA may be too small for convergence.* The requested stopping tolerance is near or below the machine unit roundoff error.
- IER=4 *Error in computing convergence statistics.* An error was encountered, such as division by zero, in computing convergence statistics according to the IPARM variable ISTATS; some of the resulting values may be incorrect.

Chapter 9

Package Requirements

The purpose of this chapter is to describe various package requirements, such as memory requirements for different choices of options. The package returns an error message if these requirements cannot be met. This chapter helps the user estimate beforehand what resources will be required by the package.

9.1 Memory Requirements

As mentioned earlier, requirements for integer workspace are measured in integer words, and requirements for floating point workspace are measured in floating point values of the working precision and arithmetic. If the IPARM variable `MALLOC` is set to 1, then memory allocation is done by use of system calls, up to available system memory, and the user need not estimate the required amount of memory. When `MALLOC=0`, the arrays `IWK` and `FWK` are used for memory allocation. In these cases, `NWIUSD` and `NWFUSD` estimate the amount of memory which would have been allocated.

The memory usage estimates given here are approximate but are always overestimates of the memory required, never underestimates. These values may be used with a call to the package, and after the call the values of `NWIUSD` and `NWFUSD` may be used to revise `NWI` and `NWF`. Values given below may be rather pessimistic; in many cases, internal memory usage optimizations occur which save memory use. The values of `NWIUSD` and `NWFUSD` returned by the package are deterministic, in the sense that they cannot change due to changes in the system's message passing scheduling or input/output, when the package is called with identical parameters.

The total workspace required can be subdivided into components: the components required for the iterative method, the matrix-vector product, the preconditioner, the preprocessing operations, and the general package bookkeeping. We omit here any discussion of the back-end memory requirements for CM Fortran versions, since such memory is always allocated by system calls without use of `IWK` or `FWK`.

The following variables are assumed for the memory usage counts:

`NRU` : the IPARM variable indicating the size of a vector (or on-processor portion, for SPMD versions).

`NS1, NS2` : the IPARM variables defined in Chapter 4.

`NB, NDIM, NSTEN` : the variables from the GR matrix format specification described in Chapter 5.

NPRECT : for the GR matrix format in SPMD versions, this is the number of grid points of a certain rectangular region which contains the on-processor subgrid as well as some points of neighbor processors used for boundary updates. Specifically, it is the smallest rectangle which contains the on-processor subgrid as well as all off-processor points it is connected to by the stencil relations expressed in JA. For example, for a 2-D subgrid of size $NXS \times NYS$, this rectangle would contain the on-processor subgrid as well as a row of points on each of the neighbors, resulting in $NPRECT = (NXS+2) * (NYS+2)$.

NPBDRY : $NPRECT - NRU / NB$

NCP : for the GR matrix format in SPMD versions, the total number of processor subgrids which are contained in or overlap the circumscribed rectangle described above. Note that this value is never more than the total number of processor subgrids. For example, for a 2-D 5-point stencil, this value would typically be 9, unless the processor partitioning along either axis is less than 3.

NWIV, NWFV : the number of integer and floating point values required to store a single vector.

NWITOT, NWFOTOT : the total number of integer and floating point values required by PCG:

$$\begin{aligned}NWITOT &= NWIGENL + NWISCL + NWIPREC + NWIMV + NWIIT + NWISTAT + NWITST \\NWFOTOT &= NWFGENL + NWFSCSCL + NWFPPREC + NWFMMV + NWFIFIT + NWFSTAT + NWFOTST\end{aligned}$$

where

NWIGENL, NWFGENL : the basic integer and floating point storage which are always needed.

NWISCL, NWFSCSCL : the integer and floating point storage for matrix scaling.

NWIPREC, NWFPPREC : the integer and floating point storage for preconditioners which are zero in current version.

NWIMV, NWFMMV : the integer and floating point storage for matrix-vector product.

NWIIT, NWFIFIT : the integer and floating point storage for iterations.

NWISTAT, NWFSTAT : the integer and floating point storage for statistic calculation. $NWFSTAT = 0$.

NWITST, NWFOTST : the integer and floating point storage for stopping test calculation. $NWFOTST = 0$.

A list of parameter values defining memory requirements for the different algorithms, for the uniprocessor, SPMD and CM Fortran cases are given in Tables 9.1 and 9.2. For specific examples of memory calculations, see the *PCG Examples Manual*. Note that $NR' = 1$ if $NR = 1$, or if IPARM variable NTEST is TSTSR or TSTRR; otherwise, $NR' = 0$. (For the definition of NR, see the definition of NEEDRC in Chapter 4.)

Table 9.1: Basic Memory Requirements

Parameter	CMF	SPMD	Uniprocessor
NWIV	20	0	0
NWFV	0	NRU+2	NRU+2
NWIGENL	32 + 2*NWIV	32 + 2*NWIV	32 + 2*NWIV
NWFGENL	31 + 2*NWFV	31 + 2*NWFV	31 + 2*NWFV
NWISTAT (ISTATS=0)	0	0	0
NWISTAT (ISTATS=1)	4*NWIV	20+ 4*NWFV	20+4*NWFV
NWITST (NTEST=TSTO, TSTDFA)	0	0	0
NWITST (NTEST≠TSTO, TSTDFA)	2*NWIV	2*NWFV	2*NWFV
NWIMV (GR format)	1*NWIV	25+2*NSTEN+6*NCP	9+2*NSTEN
NWFMV (GR format)	NWFV	3*NPBDY*NB+6*NCP	0
NWISCL (GR format)	6+2*NWIV	13+2*NCP+1*NWIV	9+NWIV
NWFSCL (GR format)	2*NWFV	NPBDY*NB+2*NCP+1*NWFV	NWFV

Table 9.2: Additional Memory for Different Methods

Method	NWIT	NWFIT
BAS	22+5*NWIV	7+5*NWFV
CG	21+5*NWIV	12+5*NWFV
BCG	26+9*NWIV	12+9*NWFV
LRES	32+12*NWIV	13+12*NWFV
CGS	29+11*NWIV	12+11*NWFV
CGST	30+13*NWIV	15+13*NWFV
CGS2	40+25*NWIV	15+25*NWFV
SQMR	29+21*NWIV	22+21*NWFV
TFQF	35+18*NWIV	16+18*NWFV
OMIN(IQSIDE=0, 1, NR'=0)	29+(2*NS1+3)*NWIV	NS1+13+(2*NS1+3)*NWFV
OMIN (general case)	29+(3*NS1+5)*NWIV	NS1+13+(3*NS1+5)*NWFV
ORES(IQSIDE=0, 1, NR'=0)	32+(2*NS1+4)*NWIV	NS1+12+(2*NS1+4)*NWFV
ORES (general case)	32+(4*NS1+6)*NWIV	NS1+12+(4*NS1+6)*NWFV
IOM (IQSIDE=0, 1, NR'=0)	32+(2*NS1+7)*NWIV	5*NS1+31+(2*NS1+7)*NWFV
IOM (general case)	32+(3*NS1+9)*NWIV	5*NS1+31+(3*NS1+9)*NWFV
GMRS(IQSIDE=0, 1, NR'=0)	35+(NS2+4)*NWIV	NS2**2+9*NS2+31+(NS2+4)*NWFV
GMRS (general case)	35+(3*NS2+8)*NWIV	NS2**2+9*NS2+31+(3*NS2+8)*NWFV
GMRH	38+(2*NS2+7)*NWIV	NS2**2+7*NS2+26+(2*NS2+7)*NWFV
CGNR	27+12*NWIV	10+12*NWFV
LSQR	31+13*NWIV	14+13*NWFV
CGNE	27+12*NWIV	10+12*NWFV
LSQE	31+16*NWIV	20+16*NWFV

9.2 Message Type Requirements

PCG requires that the range of message types used in message passing versions, as specified by IPARM variables MSGMIN and MSGMAX, be at least a certain size, depending on the requested solution algorithms and other options.

Whenever a sequence of communications is initiated, the required number of message types added to the current next-available message type as indicated in the IPARM variable MSGTYP must allow for a global synchronization to be performed without the maximum message type MSGMAX being exceeded. When message types are exhausted, a global synchronization is performed and MSGTYP is reset to MSGMIN. Whenever the package is entered, via either a call to the package or a return from a user-defined subroutine, the variable MSGTYP must not exceed MSGMAX.

In general, the message type range should be set reasonably large, so that global synchronizations are not required often. The variable definitions given in the previous section are also assumed for this section. We also define the following:

NPROCS : the total number of processors.

LOG2NPC : the ceiling of the logarithm base 2 of **NPROCS**.

The message type requirements are given in Table 9.3. To get the value for a particular machine, the maximum value over all the selected operations listed in the Table 9.3 being used by the package should be taken.

Table 9.3: Message Type Requirements

	iPSC/860	Paragon	cmmd	pvm	mpi
Global sync.	0	0	0	1	0
Inner product	LOG2NPC	0	0	LOG2NPC	0
GR format ops	NCP	NCP	NCP	NCP	NCP

9.3 Counting Floating Point Operations

The floating point operations are counted for each of the three layers of the package. A call to the reverse communication layer accumulates in the FPARM variable FLOPSR the accumulated floating point operations in the reverse communication layer. For a call to the iterative method layer, FLOPSI contains the accumulation from FLOPSR as well as floating point operations performed in the iterative method layer. Finally, from a call to the top level, FLOPST contains floating point operations performed at all levels.

Matrix vector product and preconditioning operations are counted in FLOPSI, but when the package is accessed at the iterative method level, these are not counted, since these operations are performed by the user. The initializations for package matrix vector products and preconditioning operations are always counted in FLOPST.

The FPARM variables for flop counting and timings are not absolute. Instead, for each package call the incremental expenditure for that call is added to the respective values. Since the `_DFALT`

routine initializes these values to zero, the returned values for a single call to the package will be exactly the total expenditures, though for multiple repeated calls these values are cumulative for all relevant calls.

A floating point operation is defined to be an operation on real numbers of the working precision for that call to the package. Operations are counted according to the guidelines in [McMahon 1988]. In particular, operations on real values of working precision are counted according to the Table 9.4.

Table 9.4: Operation Count Scale

Operation	Count
plus, minus, times	1
comparison	1
real-integer conversion	1
reciprocal	3
divide	4
square root	4
sine, cosine, tangent, arctangent	8
exponential, logarithm	8

The operation counts returned by the package are calculated by the package, not extracted from the CPU. An effort is made not to overcount operations. For example, if it is necessary to operate on an array padded with zeros, the operations on zeros are not counted, and redundant operations on different processors are not counted multiple times.

9.4 Package Timings

Package timings are handled in a manner similar to floating point operation counting. The FPARM variables for timings are cumulative: each call to the package increments the relevant values for the level at which the call occurs, and timers for lower levels are incremented by the relevant portion of the time spent in the given level. All times are in seconds.

Pure Fortran 77 versions of PCG do not include calls to system timer routines. For these cases, the installer of the PCG package should edit the file `xtimer.f` to include system calls to obtain system and wallclock time.

Chapter 10

Sample Programs

Below are several sample programs which illustrate the use of the package. Other example programs are provided in the *PCG Examples Manual*.

10.1 The Model Problem

The following simple programs solve Laplace's equation on a 2-D rectangular region using the conjugate gradient method and null preconditioning for the familiar 5-point stencil. Versions are given for the Cray YMP, the Intel iPSC/860, and the Connection Machine in CM Fortran.

10.2 Cray YMP Example

```
c-----  
c---Cray YMP version---  
  include '/usr/local/include/pcg_fort.h'  
  parameter (ndim=2, nsten=5, nb=1, nxs=12, nys=12)  
  parameter (icentr=1, inorth=2, isouth=3, ieast=4, iverst=5)  
  integer   ia(5), ja(ndim,nsten+3)  
  dimension a(nxs,nys,nsten), u(nxs,nys), b(nxs,nys)  
  dimension iwk(1), iparm(niparm)  
  dimension fwk(1), fparm(nfparm)  
  external srigr, scg  
  
c  
c---set ia---  
  ia(1) = ndim  
  ia(2) = nsten  
  ia(3) = nb  
  ia(4) = DFALT  
  ia(5) = DFALT  
c---initialize ja to zero---  
  do 1 iaxis = 1, ndim
```

```

do 1 isten = 1, nsten+3
  1 ja(iaxis,isten) = 0
c---define stencil---
  ja(2,inorth) = 1
  ja(2,isouth) = -1
  ja(1,ieast) = 1
  ja(1,iwest) = -1
c---specify grid size---
  ja(1,nsten+1) = nxs
  ja(2,nsten+1) = nys
c---set a interior---
  do 2 iy = 1, nys
  do 2 ix = 1, nxs
    a(ix,iy,icentr) = 4e0
    a(ix,iy,inorth) = -1e0
    a(ix,iy,isouth) = -1e0
    a(ix,iy,ieast) = -1e0
  2 a(ix,iy,iwest) = -1e0
c---eliminate a where boundary conditions are to be applied---
  do 11 iy = 1, nys
  11 a( 1, iy,iwest) = 0e0
  do 12 iy = 1, nys
  12 a(nxs, iy,ieast) = 0e0
  do 13 ix = 1, nxs
  13 a( ix, 1,isouth) = 0e0
  do 14 ix = 1, nxs
  14 a( ix,nys,inorth) = 0e0
c---initialize i/fparm---
  call sdfalt (iparm,fparm)
  iparm(LEVOU) = LEVIT
  iparm(ITSMAX) = 1000
  fparm(ZETA) = .001e0
c---set vector b---
  h = 1./(nxs+1)
  do 20 ix = 1, nxs
  do 20 iy = 1, nys
  20 b(ix,iy) = h**2
c---solve---
  call spcg ( JIRT, srigr, scg, ia, ja, a, u, u, b,
    & iw, fwk, iparm, fparm, ier )
c
  end
c-----

```

10.3 CM Fortran Example

```
c-----  
c---CM Fortran version---  
  include '/usr/local/include/pcg_fort.h'  
  parameter (ndim=2, nsten=5, nb=1, nxs=12, nys=12)  
  parameter (icentr=1, inorth=2, isouth=3, ieast=4, iverest=5)  
  integer ia(5), ja(ndim, nsten+3)  
cmf$ layout ia(:serial), ja(:serial, :serial)  
  dimension a(nsten, nxs, nys), u(nxs, nys), b(nxs, nys)  
cmf$ layout a(:serial, , ) u( , ), b( , )  
  dimension iwk(1), iparm(niparm)  
cmf$ layout iwk(:serial), iparm(:serial)  
  dimension fwk(1), fparm(nfparm)  
cmf$ layout fwk(:serial), fparm(:serial)  
  external srigr, scg  
  
c  
c---set ia---  
  ia(1) = ndim  
  ia(2) = nsten  
  ia(3) = nb  
  ia(4) = DFALT  
  ia(5) = DFALT  
  
c---initialize ja to zero---  
  do 1 iaxis = 1, ndim  
  do 1 isten = 1, nsten+3  
  1 ja(iaxis, isten) = 0  
c---define stencil---  
  ja(2, inorth) = 1  
  ja(2, isouth) = -1  
  ja(1, ieast) = 1  
  ja(1, iverest) = -1  
c---specify grid size---  
  ja(1, nsten+1) = nxs  
  ja(2, nsten+1) = nys  
c---set a interior---  
  a( , :, :) = -1e0  
  a(icentr, :, :) = 4e0  
c---eliminate a where boundary conditions are to be applied---  
  a(inorth, : , nys) = 0e0  
  a(isouth, : , 1) = 0e0  
  a(ieast, nxs, : ) = 0e0  
  a(iverest, 1 , : ) = 0e0  
c---initialize i/fparm---  
  call sdfalt (iparm, fparm)
```

```

        iparm(LEVOUT) = LEVIT
        iparm(ITSMAX) = 1000
        fparm(ZETA)   = .001e0
c---set vector b---
        h = 1./(nxs+1)
        b = h**2
c---solve---
        call spcg ( JIRT, srigr, scg, ia, ja, a, u, u, b,
&                iwk, fwk, iparm, fparm, ier )
c
        end
c-----

```

10.4 Intel IPSC/860 Example

```

c-----
c---Intel iPSC/860 version---
        include '/usr/local/include/pcg_fort.h'
        include '/usr/ipsc/include/fcube.h'
        parameter (ndim=2, nsten=5, nb=1, nxs=12, nys=12, nxp=2, nyp=2)
        parameter (icentr=1, inorth=2, isouth=3, ieast=4, iwest=5)
        integer   ia(5), ja(ndim,nsten+3)
        dimension a(nxs,nys,nsten), u(nxs,nys), b(nxs,nys)
        dimension iwk(1), iparm(niparm)
        dimension fwk(1), fparm(nfparm)
        external srigr, scg
c
c---set ia---
        ia(1) = ndim
        ia(2) = nsten
        ia(3) = nb
        ia(4) = DFALT
        ia(5) = DFALT
c---initialize ja to zero---
        do 1 iaxis = 1, ndim
          do 1 isten = 1, nsten+3
            1  ja(iaxis,isten) = 0
c---define stencil---
        ja(2,inorth) = 1
        ja(2,isouth) = -1
        ja(1,ieast)  = 1
        ja(1,iwest)  = -1
c---specify subgrid size---
        ja(1,nsten+1) = nxs
        ja(2,nsten+1) = nys

```

```

c---specify number of processors in each direction---
    ja(1,nsten+2) = nxp
    ja(2,nsten+2) = nyp
c---use gray code mapping for processor numbering---
    ja(1,nsten+3) = GRGRAY
    ja(2,nsten+3) = GRGRAY
c---set a interior---
    do 2 iy = 1, nys
    do 2 ix = 1, nxs
    a(ix,iy,icentr) = 4e0
    a(ix,iy,inorth) = -1e0
    a(ix,iy,isouth) = -1e0
    a(ix,iy,ieast ) = -1e0
    2  a(ix,iy,iwest ) = -1e0
c---eliminate a where boundary conditions are to be applied---
    ipx = mod(igrd(mynode(),ndim,ja(1,nsten+2)),nxp)
    ipy =   igrd(mynode(),ndim,ja(1,nsten+2))/nxp
    if (ipx .eq. 0 ) then
        do 11 iy = 1, nys
11     a( 1 , iy,iwest ) = 0e0
        endif
        if (ipx .eq. nxp-1) then
            do 12 iy = 1, nys
12     a(nxs, iy,ieast ) = 0e0
            endif
        if (ipy .eq. 0 ) then
            do 13 ix = 1, nxs
13     a( ix, 1 ,isouth) = 0e0
            endif
        if (ipy .eq. nyp-1) then
            do 14 ix = 1, nxs
14     a( ix,nys,inorth) = 0e0
            endif
c---initialize i/fparm---
    call sdfalt (iparm,fparm)
    iparm(LEVOUT) = LEVIT
    iparm(ITSMAX) = 1000
    fparm(ZETA)  = .001e0
c---set vector b---
    h = 1./(nxs-1)
    do 20 ix = 1, nxs
    do 20 iy = 1, nys
20    b(ix,iy) = h**2
c---solve---
    call spcg ( JIRT, srigr, scg, ia, ja, a, u, u, b,

```

```
&          iwk, fwk, iparm, fparm, ier )
```

```
c
```

```
end
```

```
c-----
```

Chapter 11

Troubleshooting

This chapter contains tips for dealing with problems encountered when using PCG. Below is a list of problems and suggested solutions.

- *The program hangs.*
 - Set LEVOUT to LEVIT or higher to monitor per-iteration results.
 - Check the computational requirements for the given algorithm and problem size to get an estimation of the expected run time on the given machine, and compare to the actual run time.
 - Check for consistency of MSGTYP and other parameters across processors, and make sure that the package message types are not being used for communications by the user program.
 - Check for invalid floating point values being passed in via user arrays which might cause the code to stall on some machines.
- *The stopping test indicates convergence, but the solution or residual I compute suggests that the approximate solution is not very accurate.*
 - Use the ISTATS parameter to recheck the residual norms. There may be significant discrepancy for some algorithms, especially the CGS algorithm and related algorithms. If necessary, choose an alternative algorithm such as GMRES.
 - If the matrix is ill-conditioned, then an algorithm designed to minimize the residual may not minimize the error very well. Choose an algorithm which minimizes a different norm of the residual.
- *The error code indicates that I have missed a parameter, but I cannot tell which one.*
 - Set LEVOUT to at least LEVERR to receive more descriptive output.
- *I get an error message saying that the main diagonal of the matrix is missing.*
 - Check the settings of IA, JA and A for a possible error in the way the matrix is stored.

●● *The iterative method I selected does not converge.*

- Some linear systems, especially nonsymmetric indefinite ones, are particularly difficult for any iterative method to solve. See Chapter 7 for a discussion of which iterative method to use.

Acknowledgments

This project has been supported by ARPA grant #DABT63-92-C-0024 and DOE. We also thank D. M. Young, D. Kincaid, D. Cline, G. Weigand, and R. Lucas for their suggestions and support.

References

General Iterative Methods

1. Dongarra, Jack J., Iain S. Duff, Danny C. Sorensen and Henk A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia: Society for Industrial and Applied Mathematics, 1991.
2. Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, second edition. Baltimore: Johns Hopkins University Press, 1989.
3. Hageman, Louis A. and David M. Young, *Applied Iterative Methods*. New York: Academic Press, 1981.
4. David M. Young, *Iterative Solution of Large Linear Systems*. New York: Academic Press, 1971.

Iterative Acceleration Techniques

1. Ashby, Stephen F., Thomas A. Manteuffel and Paul E. Saylor, "A Taxonomy for Conjugate Gradient Methods," *SIAM J. Numer. Anal.*, vol. 27, no. 6, pp. 1542-1568, December 1990.
2. Stanley C. Eisenstat, Howard C. Elman and Martin H. Schultz, "Variational Iterative Methods for Nonsymmetric Systems of Linear Equations," *SIAM J. Numer. Anal.*, vol. 2, no. 2, April 1983, pp. 345-357.
3. Hestenes, M. R. and E. L. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *J. Res. Nat. Bur. Standards*, vol. 49, 1952, pp. 409-436.
4. Wayne Joubert, "Lanczos Methods for the Solution of Nonsymmetric Systems of Linear Equations," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 3, July 1992, pp. 926-943.
5. W. D. Joubert and G. F. Carey, "Parallelizable restarted iterative methods for nonsymmetric linear systems. Part I: Theory, Part II: Parallel implementation", *International Journal of Computer Mathematics*, 44 (1992), pp. 243-267.
6. Joubert, Wayne D. and Thomas A. Manteuffel, "Iterative Methods for Nonsymmetric Linear Systems," in *Iterative Methods for Large Linear Systems*, David R. Kincaid and Linda J. Hayes eds., Boston: Academic Press, 1990, pp. 149-171.

7. C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *J. Res. Nat. Bur. Standards*, vol. 45, 1950, pp. 255-282.
8. C. C. Paige and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations," *SIAM J. Numer. Anal.*, vol. 12, 1975, pp. 617-629.
9. Christopher C. Paige and Michael A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares," *ACM Transactions on Mathematical Software*, vol. 8, no. 1, March 1982, pp. 43-71.
10. Peter Sonneveld, "CGS, a Fast Lanczos-type Solver for Nonsymmetric Linear Systems," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, Jan. 1989, pp. 36-52.
11. Youcef Saad and Martin H. Schultz "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems", *SIAM J. Sci. Stat. Comp.*, 7 (1986), pp. 856-869.
12. P. K. W. Vinsome, "ORTHOMIN, an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations," in *4th Symposium of Numerical Simulation of Reservoir Performance of the Society of Petroleum Engineers of the AIME*, Los Angeles, Calif., 1976, Paper SPE 5739.

Preconditioners

1. Ashcraft, C. Cleveland and Roger G. Grimes, "On Vectorizing Incomplete Factorization Preconditioners," *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 1, January 1988, pp. 122-151.
2. Duff, I. S. and G. A. Meurant, "The Effect of Ordering on Preconditioned Conjugate Gradient," *BIT*, 29:635-657, 1989.
3. Gustafsson, I., "A Class of First Order Factorization Methods," *BIT*, vol. 18, 1978, pp. 142-156.
4. Gustafsson, I., "Modified Incomplete Cholesky (MIC) Method," in *Preconditioning Methods: Theory and Applications*, D. J. Evans, ed. New York: Gordon and Breach, 1983, pp. 265-293.
5. Wayne Joubert and Thomas Oppe, "Improved SSOR and Incomplete Choleski Solution of Linear Equations on Shared Memory and Distributed Memory Parallel Computers," Report CNA-253, Center for Numerical Analysis, the University of Texas at Austin, November 1991, also Report FSU-SCRI-91V-171, Supercomputer Computations Research Institute, Florida State University, Tallahassee, FL, to appear in *Numerical Linear Algebra with Applications*.
6. Manteuffel, T. A., "Shifted Incomplete Cholesky Factorization," in *Sparse Matrix Proceedings 1978*, I. S. Duff and G. W. Stewart eds. Philadelphia: SIAM Publications, 1979.
7. Meijerink, J. A. and H. A. van der Vorst, "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M -Matrix," *Mathematics of Computation*, vol. 31, no. 137, Jan. 1977, pp. 148-162.

Software

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK User's Guide*, Philadelphia: SIAM, 1992.
2. Steven F. Ashby and Mark K. Seager, "A Proposed Standard for Iterative Linear Solvers, Version 1.0," Lawrence Livermore National Laboratory, Report UCRL-102860, January 1990.
3. P. V. Bangalore, A. Skjellum, C. Baldwin, and S. G. Smith, "Dense and Iterative Concurrent Linear Algebra in the Multicomputer Toolbox," in *Proceedings of the Scalable Parallel Libraries Conference*, October 6-8, 1993, Mississippi State University. Los Alamitos, CA: IEEE Computer Society Press, 1994, pp. 132-141.
4. Richard Barrett, Mike Berry, Tony Chan, Jim Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Chuck Romine and Henk van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia: SIAM, 1993.
5. Rudnei Dias da Cunha and Tim Hopkins, "PIM 1.1: The Parallel Iterative Methods Package for Systems of Linear Equations User's Guide," Computing Laboratory, University of Kent at Canterbury, United Kingdom, 1994.
6. J. J. Dongarra, J. R. Bunch, C. B. Moler, G. W. Stewart, *LINPACK Users' Guide*. Philadelphia: SIAM, 1979.
7. Ian S. Duff, Michele Marrone and Giuseppe Radicati, "A proposal for user level sparse BLAS," Technical Report TR/PA/92/85, CERFACS, Toulouse, France, 1992.
8. William Gropp and Barry Smith, "Scalable, Extensible, and Portable Numerical Libraries," in *Proceedings of the Scalable Parallel Libraries Conference*, October 6-8, 1993, Mississippi State University. Los Alamitos, CA: IEEE Computer Society Press, 1994, pp. 87-93.
9. Michael Heroux, "A proposal for a sparse BLAS toolkit," Technical Report TR/PA/92/90, CERFACS, Toulouse, France, 1992.
10. Heroux, Michael A., Phuong Vu and Chao Yang, "A Parallel Preconditioned Conjugate Gradient Package for Solving Sparse Linear Systems on a Cray Y-MP," 1991, to appear in IMACS journal *Applied Numerical Mathematics*.
11. Wayne Joubert, Peter Highnam and Graham Carey, "PCG/CM: A Package for the Iterative Solution of Large Sparse Linear Systems on the Connection Machine," in *Proceedings of Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Jack J. Dongarra, Ken Kennedy, Paul Messina, Danny C. Sorensen and Robert Voight, eds. Philadelphia: SIAM, 1992.
12. W. Joubert and G. F. Carey, "PCG: A Software Package for the Iterative Solution of Linear Systems on Scalar, Vector and Parallel Computers," in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, eds. Richard F. Sincovec, David E. Keyes, Michael R. Leuze, Linda R. Petzold, Daniel A. Reed. Philadelphia: SIAM, 1993, pp. 515-518.

13. D. R. Kincaid, T. C. Oppe, and W. D. Joubert, "An Overview of NSPCG: A Nonsymmetric Preconditioned Conjugate Gradient Package," *Computer Physics Communications*, vol. 53, 1989.
14. David R. Kincaid, Thomas C. Oppe and Wayne D. Joubert, "An Introduction to the NSPCG Software Package," *International Journal for Numerical Methods in Engineering*, vol. 27, 1989.
15. D. R. Kincaid, J. R. Respass, D. M. Young, and R. G. Grimes, "ITPACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods," *ACM Transactions on Mathematical Software*, Vol. 8, No. 3, September 1982, pp. 302-322.
16. Frank H. McMahon, "The Livermore FORTRAN Kernels Test of the Numerical Performance Range," in *Performance Evaluation of Supercomputers*, ed. Joanne L. Martin, Amsterdam: North Holland, 1988, pp. 143-186.
17. Thomas C. Oppe, Wayne D. Joubert and David R. Kincaid, "NSPCG User's Guide Version 1.0," Report CNA-216, Center for Numerical Analysis, University of Texas at Austin, April 1988, 85pp.
18. Youcef Saad, "SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations," unpublished report, RIACS, Mail Stop 230-5, NASA Ames Research Center, Moffett Field, CA 94035, May 21, 1990.
19. Scientific Computing Associates, New Haven, CT, *PCGPACK User's Guide*, 1984.
20. Mark K. Seager, "A SLAP for the Masses," in G. F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, pp. 135-155, John Wiley and Sons Ltd, New York, 1989.

Index

- BAS, 46
- BCG, 1, 20, 46
- Biconjugate gradient, 1, 20, 35–37, 39
- Breakdown, 36, 42

- CGS, 1, 36, 46, 55
- CGSTAB, 37
- Cholesky, 34
- CM, 2, 6, 8, 10–12, 19, 21, 26, 27, 34, 42, 44, 45, 49
- Communication, 1, 13–15, 17, 20, 21, 35, 40, 42, 47
- Condition number, 32, 38, 39, 41
- Conjugate gradient, 1, 3, 36–39, 49
- Conjugate residual, 37, 38
- Convergence, 10, 18, 20, 24, 32–41, 43, 55
- Cray, 2, 28, 49

- Default, 6, 13, 14, 16–21, 26, 28, 32, 34, 36, 40, 41
- Diagonal scaling, 32, 33

- Error code, 7, 41, 42, 55

- Floating point operations, 22–24, 47
- Floating point parameters, 6, 16, 17

- GMRES, 1, 3, 20, 33, 37, 38, 55
- Gray code, 27, 29, 30
- Grid, 25–27, 29, 45

- Hermitian, 3, 32, 33, 36–39
- HPD, 38

- Incomplete Cholesky, 34
- Incomplete factorization, 34
- Initialization, 4, 5, 9, 11, 12, 18, 19
- Integer parameters, 6
- Intel, 2, 13–15, 49

- Jacobi preconditioning, 34

- Krylov space, 36–38

- Lanczos, 36
- Laplace's equation, 49

- Matrix formats, 1, 25, 33, 34
- Matrix scaling, 2, 32, 34
- Matrix storage, 1, 5, 25
- Matrix-vector product, 1, 7, 8, 11, 15, 21, 23, 44
- Memory allocation, 6, 16, 18, 24, 42, 44
- Memory requirements, 16, 44–46
- Message, 14, 15, 18, 21, 40, 44, 47, 55
- Message passing, 2, 13–15, 17–19, 21–24, 26–29, 44, 47
- Message type, 14, 15, 21, 47
- MPI, 2, 13–16, 21, 47
- Multiple linear systems, 12

- Normal equations, 38, 39

- Orthomin, 20, 33, 37, 38
- Orthores, 36, 37

- Performance, 1, 2, 26, 28, 34, 37
- Polynomial preconditioning, 38
- Precision, 2, 4, 6, 7, 16, 18, 22–24, 38, 39, 44, 48
- Preconditioners, 1, 11, 33, 34
- Preconditioning, 1, 3, 5, 7, 8, 12, 14, 15, 19, 22, 23, 32–36, 47, 49
- Processor numbering, 14
- Processor subgrids, 26, 45
- PVM, 2, 13, 14, 16, 21, 47

- QMR, 1, 20, 21, 37, 39

Reverse communication, 1, 4, 7, 9, 11, 14, 15,
17, 18, 20, 22, 23, 35, 36, 40, 47

Reverse gray code, 30

Richardson's method, 33, 34

Roundoff error, 22, 32, 38, 43

Rowsum scaling, 33

Scaling, 22, 32-34

Stencil, 25-27, 45, 49

Subgrid, 26-29, 45

Symmetric, 3, 32, 33

Timings, 22-24, 34, 47, 48

Top level, 4, 17, 18, 23, 24, 32, 47

Troubleshooting, 55

User-defined subroutines, 7

Vector operations, 8

Workspace, 6, 16, 18, 19, 24, 41, 42, 44