The
Connection Machine
System

CMSSL for CM Fortran: CM-5 Edition, Volume I

Version 3.1

June 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, June 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

Contents

Volume II

Foreword

Software libraries are important tools in the use of computers. Libraries for scientific and engineering applications embody expert knowledge of data structures, algorithms, operating systems, compilers, computer architecture, applied mathematics, and numerical analysis. Libraries enhance productivity not only by providing preprogrammed functions, but, what is more important, by providing functions that have well-understood and documented storage requirements, execution time, and numerical behavior. Libraries make the architecture of computers more transparent to a user than programming languages by defining functions at a sufficiently high level for optimization beyond the capabilities of compilers. The portability of user programs is enhanced with respect to both performance and numerical behavior. Libraries substantially lower the cost of computation, improve productivity, and enhance the quality of the end result.

Developments in computer architecture represent particularly strong forces behind the evolution of libraries for high-performance computers. The new generation of high-performance architectures, scalable to several trillion operations per second, consists of thousands of processing units with local memories, and a network interconnecting the processor and memory units. The performance optimization of functions to be executed on such architectures requires careful attention to data allocation, data motion in distributed data structures, memory hierarchies, load balancing, and scheduling of pipelines. Fundamental changes of classical algorithms may be required. The efficient use of such scalable computer architectures is beyond the capability of state-of-the-art compiler technology.

Libraries provide significantly more powerful constructs than those available in most programming languages. The invocation of a library routine implies that a function be applied to the objects defined in a programming language, and that information about the objects be extracted, transformed, or used to generate new objects. The array syntax of recently introduced programming languages has a profound impact on the interface to a library and on its functionality and design. The array syntax of programming languages is in part motivated by the emergence of parallel computer architectures, particularly data parallel architectures. Concurrency occurs both in applying high-level functions to disjoint data sets, sometimes defined through a recursive procedure, and in each application of the function.

The Connection Machine Scientific Software Library (CMSSL) is created for languages with an array syntax and for data parallel architectures. The CMSSL is designed to handle concurrent application of a function to disjoint segments of arrays, and concurrent execution of each application. Concurrent application of the same function to segments of arrays implies computation on multiple instances, a very important feature for library routines on scalable architectures. The multiple-instance feature provides concurrency

control for the library independent of the control structures in the language to which it is interfaced. The multiple-instance paradigm enhances portability and is a new feature for scalable architectures.

In the CMSSL, efficient use of the Connection Machine system architecture is accomplished through a careful choice of data layout, efficient implementation of interprocessor data motion, and careful management of the local memory hierarchy and data paths in each processor. The library accepts any data layout that can be specified for any machine configuration. Internally, library functions may reallocate arrays for optimum performance, or to establish a common processor configuration for all operands in a function evaluation. Performance tuning through control of the data allocation is largely new to data parallel architectures, though some of the issues are analogous to those occurring in banked memory systems and systems with a cache.

The CMSSL achieves architectural independence with respect to data motion through a set of communication functions providing a shared memory view of the global address space. Efficient management of the resources for each processor is achieved through level 2 and level 3 Basic Linear Algebra Subroutines (BLAS). Blocking schemes are used for some BLAS functions, and for functions such as the Fast Fourier Transform, for which high-radix algorithms are advantageous with respect to performance.

In summary, the CMSSL is a library for languages with an array syntax and addresses many new issues related to concurrency control, data allocation and data motion in distributed data structures, language independence, and scalability. It is our hope that the CMSSL will serve the users of distributed-memory architectures well, and that it will evolve to include a broad set of basic functions frequently used in scientific and engineering applications, as well as higher-level functions for ordinary and partial differential equations, optimization, and signal processing.

S. Lennart Johnsson

Director of Computational Sciences, Thinking Machines Corporation
Gordon MacKay Professor of the Practice of Computer Science, Harvard University
Cambridge, Massachusetts
December 1992

About This Manual

Objectives

This manual describes the CM Fortran programming interface to the Connection Machine Scientific Software Library (CMSSL).

This manual describes CMSSL software for the Connection Machine supercomputer, model CM-5. (Note that throughout this book, statements made about the CM-200 also apply to the CM-2, unless otherwise noted.)

Intended Audience

Anyone writing CM Fortran programs that use the CMSSL software should read this document.

Organization

This manual is divided into two volumes with fourteen chapters:

Volume I

Chapter 1 Introduction to the CMSSL for CM Fortran
Describes the contents of the CMSSL. Discusses the data types supported and explains how to perform CMSSL operations on multiple independent data sets concurrently.

Chapter 2 Using the CMSSL CM Fortran Interface
Explains how to include CMSSL routine definitions in CM Fortran code, and how to compile, link, and execute CM Fortran programs that call CMSSL routines.

Chapter 3 Dense Matrix Operations
Describes the inner product, 2-norm, outer product, matrix vector multiplication, vector matrix multiplication, matrix multiplication, and infinity norm routines. Also describes the routine that performs matrix multiplication routine with external storage.

Chapter 4 Sparse Matrix Operations
Describes the routines that perform arbitrary elementwise sparse matrix operations, arbitrary block sparse matrix operations, and grid sparse matrix operations.

Chapter 5 Linear Solvers for Dense Systems
Describes the in-core linear solvers: Gaussian elimination (*LU* decomposition) routines, routines that solve linear systems using Householder transformations (the *QR* routines), matrix inversion, the Gauss-Jordan system solver. Also describes the external (out-of-core) Gaussian elimination and *QR* factorization routines.

Chapter 6 Linear Solvers for Banded Systems
Describes the banded system factorization and solver routines, which solve tridiagonal, block tridiagonal, pentadiagonal, and block pentadiagonal systems.

Chapter 7 Iterative Solvers
Describes routines that solve linear systems using Krylov space iterative methods.

Chapter 8 Eigensystem Analysis
Describes routines that perform eigensystem analysis of dense real symmetric tridiagonal systems, dense Hermitian systems, dense real symmetric systems, dense real systems, and sparse systems. Included are routines that use the Jacobi method, a *k*-step Lanczos method, and a *k*-step Arnoldi method. Also included are routines that reduce Hermitian matrices to real symmetric tridiagonal form (and perform the corresponding basis transformation).

Volume II

Chapter 9 Fast Fourier Transforms
Describes the simple and detailed complex-to-complex FFT routines; the real-to-complex and complex-to-real FFT routines; and array conversion utilities for the real-to-complex and complex-to-real FFTs.

Chapter 10 Ordinary Differential Equations
Describes routines that integrate ordinary differential equations (ODEs) explicitly using a fifth-order Runge-Kutta-Fehlberg formula.

Chapter 11 Linear Programming
Describes a routine that solves multi-dimensional minimization problems using the simplex linear programming method.

Chapter 12 Random Number Generators
Describes the Fast and VP random number generators.

Chapter 13 Statistical Analysis
Describes the histogram and range histogram routines.

Chapter 14 Communication Primitives
Describes the polyshift operation; the all-to-all rotation, broadcast, and reduction routines; a matrix transpose routine; the sparse gather and scatter, sparse vector gather and scatter, and block gather and scatter utilities; partitioning of an unstructured mesh and reordering of pointers; the partitioned gather and scatter utilities; the communication compiler; the vector move (extract and deposit) routines; routines that compute block cyclic permutations and permute an array along an axis; and send-to-NEWS and NEWS-to-send reordering.

Revision Information

This is the first edition of this manual.

Acknowledgments

The Arnoldi routines, which are new in this release, are the result of collaborative work with the Center for Research on Parallel Computation at Rice University.

Notation Conventions

The table below displays the notation conventions used in this manual.

Convention	Meaning
`bold typewriter`	UNIX and CM System Software commands, command options, and file names.
boldface sans serif	CM Fortran language elements, such as function and subroutine names and constants, when they appear embedded in text or in syntax lines.
italics	Parameter names, when they appear embedded in text or syntax lines.
bold italics	CM arrays, when they appear embedded in text or syntax lines.
`typewriter`	Code examples and code fragments.
`% bold typewriter` `typewriter`	In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in `regular typewriter` font.

Standard Abbreviations for Matrix Operations and Matrix Types

The following standard abbreviations are used in the CMSSL CM Fortran interfaces to identify matrix types. Further abbreviations will be introduced as more matrix types are supported.

CMSSL Matrix Type Abbreviations

dense general	**gen**
dense symmetric	**sym**
arbitrary elementwise sparse	**sparse**
arbitrary block sparse	**block_sparse**
grid sparse	**grid_sparse**
tridiagonal	**gen_tridiag**
pentadiagonal	**gen_pentadiag**
block tridiagonal	**block_tridiag**
block pentadiagonal	**block_pentadiag**

The following standard abbreviations are used in the CMSSL CM Fortran interfaces to identify matrix operations:

CMSSL Matrix Operation Abbreviations

factorization	**factor**
inversion	**invert**
multiplication	**mult**
solver	**solve**
polyshift	**pshift**

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1264
Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
Telephone:	(617) 234-4000

Chapter 1

Introduction to the CMSSL for CM Fortran

This chapter contains general information about the CM Fortran interface to the Connection Machine Scientific Software Library (CMSSL). The following topics are included:

- about the CMSSL
- contents of the CMSSL for CM Fortran
- data types supported
- notes on terminology
- support for multiple instances
- numerical stability for the linear algebra routines
- numerical complexity
- CM Fortran performance enhancements with CMSSL

1.1 About the CMSSL

The CMSSL is a rapidly growing set of numerical routines that support computational applications while exploiting the massive parallelism of the Connection Machine system. The CMSSL provides data parallel implementations of familiar numerical routines, offering new solutions for performance optimization, algorithm choice, and application design. The library can be linked with code written in CM Fortran.

The CMSSL includes dense and sparse matrix operations; routines for solving dense, banded, and sparse linear systems; eigensystem analysis routines; fast Fourier transforms; routines for solving ordinary differential equations; a routine that solves minimization problems using the simplex linear programming method; random number generators; and histogramming routines. The library also provides a set of communication functions that offer a strong base for the development of computational tools. These functions support computations on problems represented by both structured and unstructured grids. Many CMSSL routines have been implemented to allow parallel computation on either multiple independent objects or a single large object. Over time, the CMSSL will continue to grow into a complete set of standard scientific subroutines.

1.2 Contents of the CMSSL for CM Fortran

The CM Fortran interface to the CMSSL consists of a set of library routines and a safety mechanism.

1.2.1 Library Routines

Listed below are the operations included in the CMSSL for CM Fortran on the CM-5.

- **Dense Matrix Operations**
  - *Inner Product*

    The *multiple-instance* inner product routines compute one or more instances of an inner product of two vectors. Each *single-instance* inner product routine computes the global inner product over all axes of two source CM arrays. The inner product either overwrites the destination, is added to the destination, or is added to a second variable. For complex data, routines that take the conjugate of the first operand are provided.

  - *2-Norm*

    The *multiple-instance* 2-norm routine computes one or more instances of the 2-norm of a vector. The *single-instance* 2-norm routine computes the global 2-norm of a CM array.

- *Outer Product*

  The outer product routines compute one or more instances of an outer product of two vectors. The result either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array. For complex data, routines that take the conjugate of the second operand vector are provided.

- *Matrix Vector Multiplication*

  The matrix vector multiplication routines compute one or more matrix vector products. The result either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array. For complex data, routines that take the conjugate of the matrix are provided.

- *Vector Matrix Multiplication*

  The vector matrix multiplication routines compute one or more vector matrix products. The result either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array. For complex data, routines that take the conjugate of the matrix are provided.

- *Infinity Norm*

  Computes the infinity norm(s) of one or more matrices.

- *Matrix Multiplication*

  The matrix multiplication routines compute one or more matrix products. The result either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array. Routines that take the transpose of either or both operand matrices (or the conjugate of either matrix, for complex data) are provided.

- *Matrix Multiplication with External Storage*

  This routine performs the operation $Y = Y + AX$ where Y, A, and X are matrices and A is too large to fit into core memory.

- **Sparse Matrix Operations**
  - *Arbitrary Elementwise Sparse Matrix Operations*

    These routines compute the product of an arbitrary sparse matrix with a vector or dense matrix. The user application must store the non-zero elements of the sparse matrix in a packed vector. An associated setup routine provides options that may improve performance.

  - *Arbitrary Block Sparse Matrix Operations*

    These routines compute the product of a block sparse matrix with a vector or a dense matrix. Operand elements are gathered from the source vector or matrix, and product elements are scattered to the product vector or matrix, using pointers provided by the application. An associated setup routine provides options that may improve performance.

  - *Grid Sparse Matrix Operations*

    These routines perform matrix vector, vector matrix, and matrix matrix multiplication in which the operand arrays are distributed across the points of a regular structured grid. These routines support multiple instances and block matrices.

- **General Linear System Solvers (In-Core)**
  - *Gaussian Elimination Routines*
    - *LU factorization routine*

      This routine uses Gaussian elimination (with or without partial pivoting) to factor one or more instances of an $m \times n$ matrix A into a lower triangular matrix L and an upper triangular matrix U, $A=LU$.

    - *LU solver routines*

      These routines use the triangular factors L and U produced by the LU factorization routines to produce solutions to the systems $LUX=B$ or $(LU)^TX=B$. B may represent one or more right-hand sides for each instance of the systems of equations.

- *Triangular system solvers ("LU factor application routines")*

  These routines use the factors produced by the *LU* factorization routines to solve triangular systems of equations. Included are routines for solving one or more instances of triangular systems of equations of the form $LX=B$, $L^TX=B$, $UX=B$, and $U^TX=B$. B may represent one or more right-hand sides for each instance of the systems of equations.

- *LU utility routines*

  The CMSSL also provides a set of utility routines associated with the *LU* factorization routine. For example, there are routines that explicitly compute L and U from the representation used internally in the factorization routine; save and restore internal *LU* information to or from a file; and estimate the infinity norm of each matrix A^{-1}.

- *Routines for Solving Linear Systems Using Householder Transformations*

  - *QR factorization routine*

    This routine uses Householder transformations (with or without column pivoting) to factor one or more instances of an $m \times n$ matrix A, $m \geq n$, into a trapezoidal matrix Q and an upper triangular matrix R, $A=QR$. (When you specify pivoting, each matrix A is factored into three matrices: $A = QRP^{-1}$, where P is the permutation matrix that corresponds to the pivoting process.)

  - *QR solver routines*

    These routines use the Q and R factors produced by the *QR* factorization routines to solve one or more instances of the systems of equations $QRX=B$ or $(QR)^TX=B$. (With pivoting, these equations become $QRP^{-1}X = B$ and $(QRP^{-1})^TX=B$.) B may represent one or more right-hand sides for each instance of the systems of equations.

- *Triangular system solvers ("QR factor application routines")*

  These routines use the factors produced by the *QR* factorization routine to solve triangular systems of equations (trapezoidal systems for *Q*). Included are routines for solving one or more instances of triangular systems of equations of the form $RX=B$ and $R^TX=B$, and trapezoidal systems of the form $QX=B$ or $Q^TX=B$. *B* may represent one or more right-hand sides for each instance of the systems of equations.

- *QR utility routines*

  The CMSSL also provides a set of utility routines associated with the *QR* factorization routine. For example, there are routines that explicitly compute *R* from the representation used internally in the factorization routine; extract and deposit the diagonal of *R*; save and restore internal *QR* information to or from a file; apply the pivot permutation matrix to a supplied matrix or vector; and estimate the infinity norm.

- *Matrix Inversion*

  This routine inverts a square matrix *A* using the Gauss-Jordan routine.

- *Gauss-Jordan System Solver*

  This routine solves (with partial or total pivoting) a system of equations of the form $AX=B$ using a version of Gauss-Jordan elimination. *B* represents one or more right-hand sides.

- **General Linear System Solvers (External)**

  - *Gaussian Elimination with External Storage*

    - *External LU factorization routine*

      This routine uses block Gaussian elimination with partial pivoting to reduce an $n \times n$ matrix *A* to triangular form, where *A* is too large to fit into core memory.

- *External LU solver routine*

  Given the factors computed by the external *LU* factorization routine, this routine solves $AX = B$ for an arbitrary number of right-hand sides.

- *QR Factorization and Least Squares Solution with External Storage*

  - *External QR factorization routine*

    This routine uses block Householder reflections to perform the factorization $A = QR$, where the matrix A is $m \times n$ (with $m \geq n$) and is too large to fit into core memory.

  - *External QR solver routine*

    Given the factors computed by the external *QR* factorization routine, this routine solves $AX = B$ for an arbitrary number of right-hand sides.

- **Banded Linear System Solvers**

  - *Banded System Factorization and Solver Routines ("Unified")*

    These routines factor and solve tridiagonal, block tridiagonal, pentadiagonal, and block pentadiagonal systems. One routine performs the factorization. A second routine uses the resulting factors to solve one or more instances of systems of equations of the form $LUX = B$, where L and U are lower and upper (respectively) bidiagonal or block bidiagonal, or lower and upper (respectively) tridiagonal or block tridiagonal matrices, or permutations thereof. B represents one or more right-hand sides for each system of equations. You can choose from several algorithms: pipelined Gaussian elimination, pipelined Gaussian elimination with pairwise pivoting, substructuring with cyclic reduction, substructuring with balanced cyclic reduction, substructuring with pipelined Gaussian elimination, or substructuring with transpose.

  - *Banded System Factorization and Solver Routines*

    These routines perform the same operations as the banded solvers described above, and are included in the library primarily for compatibility with the CM-200. For each type of system, the library

provides separate factorization and solver routines as well as one routine that both factors and solves.

- **Iterative Solvers**
  - *Krylov-Based Iterative Solvers*

    Given a matrix A, a right-hand-side vector b, and a preconditioner $M = M_1 * M_2$, such that $A^{\sim} = M_1^{-1} A M_2^{-1}$, these routines solve the system $Ax = b$ using Krylov space iterative methods. Any matrix operations and preconditioning steps are provided by the user using a reverse communication interface.

- **Eigensystem Analysis of Real Symmetric Tridiagonal Systems**
  - *Reduction to Tridiagonal Form and Corresponding Basis Transformation*

    These routines reduce one or more real symmetric or complex Hermitian matrices to real symmetric tridiagonal form using Householder transformations. After this reduction occurs, for each instance, you can transform the coordinates of an arbitrary set of vectors from the basis of the original Hermitian matrix to that of the tridiagonal matrix, or vice versa.

  - *Eigenvalues of Real Symmetric Tridiagonal Matrices*

    This routine computes the eigenvalues of one or more real symmetric tridiagonal matrices using a parallel bisection algorithm.

  - *Eigenvectors of Real Symmetric Tridiagonal Matrices*

    This routine computes the eigenvectors corresponding to a given set of eigenvalues for one or more real symmetric tridiagonal matrices, using an inverse iteration algorithm.

- **Eigensystem Analysis of Dense Hermitian Systems**
  - *Eigensystem Analysis of Dense Hermitian Matrices*

    This routine computes the eigenvalues and eigenvectors of one or more real symmetric or complex Hermitian matrices.

- **Eigensystem Analysis of Dense Real Symmetric Systems**

  - *Generalized Eigensystem Analysis of Real Symmetric Matrices*

    Given a CM array containing one or more real symmetric matrices A, and a CM array containing corresponding positive definite matrices B, this routine solves $AQ = BQ\lambda$, computing the eigenvalues λ and, if desired, the eigenvectors for each instance.

  - *Eigensystem Analysis of Real Symmetric Matrices Using Jacobi Rotations*

    This routine computes the eigenvalues and eigenvectors of one or more real symmetric matrices using Jacobi rotations.

  - *Selected Eigenvalue and Eigenvector Analysis Using a* k*-Step Lanczos Method*

    This routine finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. B can be positive semi-definite and is the identity for the standard eigenproblem. The operator L must be real and symmetric with respect to B; that is, $BL = L^{-1}B$. The algorithm used is a k-step Lanczos algorithm with implicit restart.

- **Eigensystem Analysis of Dense Real Systems**

  - *Selected Eigenvalue and Eigenvector Analysis Using a* k*-Step Arnoldi Method*

    This routine finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. B is symmetric and can be positive semi-definite; it is the identity for the standard eigenproblem. The algorithm used is a k-step Arnoldi algorithm with implicit restart.

- **Eigensystem Analysis of Sparse Systems**

  The Lanczos and Arnoldi routines described above also apply to sparse systems.

- **Fast Fourier Transforms (FFTs)**
  - *Simple Complex-to-Complex FFT*

    Performs a complex-to-complex Fast Fourier Transform in the same direction along all axes of a data set.

  - *Detailed Complex-to-Complex FFT*

    Allows separate specification of the transform direction, scaling factor, and addressing mode along each data axis in a complex-to-complex FFT. Can improve performance over the Simple FFT in some cases. Supports multiple instances.

  - *Real-to-Complex and Complex-to-Real FFTs*

    The real-to-complex FFT computes the Fourier transform of real data; the complex-to-real FFT transforms conjugate symmetric sequences.

  - *Array Conversion Utilities*

    These utilities convert real arrays into complex arrays suitable for input to the real-to-complex FFT, and convert complex arrays (supplied in the format produced by the complex-to-real FFT) to real arrays.

- **Ordinary Differential Equations**
  - *Explicit Integration of Ordinary Differential Equations Using a Runge-Kutta Method*

    The initial value problem for a system of N coupled first-order ordinary differential equations (ODEs), $dy_i(x)/dx = f_i(x, y_1, \ldots, y_N)$ $i=1$, ..., N consists of finding the values $y_i(x_1)$ at some value x_1 of the independent variable x, given the values $y_i(x_0)$ of the dependent variables at x_0. This routine solves the initial value problem by integrating explicitly the set of equations above using a fifth-order Runge-Kutta-Fehlberg formula. Control of the step size during integration is automatic. The evaluation of the right-hand side and possibly the scaling array for accuracy control are provided by the user through a reverse communication interface.

- **Linear Programming**

  - *Dense Simplex*

    This routine solves multidimensional minimization problems using the simplex linear programming method. The goal is to find the minimum of a linear function of multiple independent variables. In the standard formulation, the problem is to minimize the inner product $c^T x$ subject to the conditions $Mx = b$, $0 \leq x \leq u$, where M is an m x n matrix, c is a coefficient vector, and $c^T x$ is referred to as the *cost*. The upper bound vector u may be infinity in one or more components.

- **Random Number Generators (RNGs)**

  - *Fast RNG*

    This lagged-Fibonacci RNG is faster than the standard RNG included in CM Fortran. It generates either real or integer pseudo-random numbers, allows user-controlled reinitialization and checkpointing, and allows users to save and restore the RNG state table.

  - *VP RNG*

    This lagged-Fibonacci RNG produces identical streams on CM partitions of different sizes. It generates either real or integer pseudo-random numbers, allows user-controlled reinitialization and checkpointing, and allows users to save and restore the RNG state table.

- **Statistical Analysis**

  - *Full Histogram*

    The full histogram records the distribution of all values within one or more source fields. Successive calls can provide an accumulation of totals.

  - *Range Histogram*

    The range histogram records the distribution of values within specified ranges of values within one or more source fields. Successive calls can provide an accumulation of totals.

- **Communication Primitives**

  - *Polyshift*

    This routine performs multidirectional and/or multidimensional array shifts in an array geometry.

  - *All-to-All Rotation (previously called "All-to-All Broadcast")*

    Given a real or complex array and a designated axis, this routine performs an in-place, stepwise rotation of every array value on the axis to every location along the axis.

  - *All-to-All Broadcast*

    Given source and destination CM arrays of the same type, with rank(destination array) = rank(source array) + 1, the all-to-all broadcast routine copies each instance of a source vector to the destination array and replicates it along a selected axis (the "broadcast axis") of the destination array.

  - *All-to-All Reduction*

    Given source and destination CM arrays with rank(source array) = rank(destination array) + 1, the all-to-all reduction routine combines sets of vectors within the source array and places each result in a corresponding vector of the destination array.

  - *Matrix Transpose*

    Given a CM array of any type and two designated axes, this routine exchanges the two axes and returns the result in a second CM array.

  - *Sparse Gather Utility*

    These routines gather elements of a vector into an array using pointers supplied by the application. Pre-processing is performed by an associated setup routine.

  - *Sparse Scatter Utility*

    These routines scatter elements of an array to a vector using pointers supplied by the application. Pre-processing is performed by an associated setup routine.

- *Sparse Vector Gather Utility*

  These routines perform the same operation as the sparse gather routines, except that the sparse vector gather operates on vectors rather than individual data elements.

- *Sparse Vector Scatter Utility*

  These routines perform the same operation as the sparse scatter routines, except that the sparse vector-scatter operates on vectors rather than individual data elements.

- *Block Gather and Scatter Utilities*

  These routines move a block of data from a source CM array into a destination CM array. The arrays must have the same rank (≥ 2), type (integer, real, or complex), precision, and layout, with at least one serial axis and at least one parallel axis. The gather or scatter operation occurs along a single, specified serial axis. In the simplest case, a block of data elements is moved from a two-dimensional source array (with one serial dimension and one parallel dimension) to a similar destination array. You can add instances by extending the parallel axis or by adding more axes (which may be serial or parallel).

- *Partitioning of an Unstructured Mesh and Reordering of Pointers*

  These routines allow you to reorder an array of pointers derived from a mesh so that the communication required by subsequent partitioned gather and scatter operations is reduced. Four routines are provided:

  - Given an *element nodes* array that describes an unstructured mesh, one routine produces the corresponding *dual connectivity* array.
  - Given a dual connectivity array, a second routine returns a permutation that reorders the mesh elements to form discrete *partitions*.
  - Given a pointers array and a permutation, a third routine reorders the pointers array along its last axis using the permutation.

- Given a pointers array, a fourth routine changes the pointer values for improved locality and returns the renumbering mapping.

If you derive a pointers array from a mesh, reorder it using the permutation returned by the partitioning routine, and then supply these reordered pointers to the setup routine for the partitioned gather or scatter operation, the setup routine takes advantage of data locality; the communication required by the gather or scatter is reduced.

- *Partitioned Gather Utility*

  These routines perform the same operations as the sparse gather and sparse vector gather routines. If you supply a pointers array that is reordered along its last axis to achieve data locality, the partitioned gather takes advantage of this locality, reducing communication time.

- *Partitioned Scatter Utility*

  These routines perform the same operations as the sparse scatter and sparse vector scatter routines. If you supply a pointers array that is reordered along its last axis to achieve data locality, the partitioned gather takes advantage of this locality, reducing communication time.

- *Communication Compiler*

  A set of routines that compute and use message delivery optimizations for basic data motion and combining operations (get, send, send with overwrite, and send with combining). The communication compiler allows you to compute an optimization (or *trace*) just once, and then use the trace many times in subsequent data motion and combining operations. This feature can yield significant time savings in applications that perform the same communication operation repeatedly. The communication compiler offers a variety of methods for computing a trace.

- *Vector Move (Extract and Deposit)*

  This routine moves a vector from a source array to a destination array of the same rank, data type, and processing element layout. An associated utility routine returns processing element layout and subgrid shape information for any CM array.

- *Computation of Block Cyclic Permutations*

  This routine computes the permutations required to transform one or more matrices from normal (elementwise consecutive) order to block cyclic order, and vice versa.

- *Permutation along an Axis*

  This routine permutes the rows or columns of one or more matrices, using a permutation that is supplied in an array.

- *Send-to-NEWS and NEWS-to-Send Reordering*

  On the CM-200, these routines allow you to change the ordering of specified axes of a CM array from send to NEWS ordering or vice-versa. On the CM-5, these routines have no effect because send and NEWS ordering are the same. They are provided only for compatibility with the CM-200. (Refer to the CM Fortran documentation set for information about send and NEWS ordering.)

Table 1 lists the CMSSL routines for CM Fortran on the CM-5, along with the chapters that describe them.

Table 1. CMSSL Routines for CM Fortran.

Operation	Chapter	Routines
Inner product	3	**gen_inner_product** **gen_inner_product_noadd** **gen_inner_product_addto** **gen_inner_product_c1** **gen_inner_product_c1_noadd** **gen_inner_product_c1_addto** **gbl_gen_inner_product** **gbl_gen_inner_product_noadd** **gbl_gen_inner_product_addto** **gbl_gen_inner_product_c1** **gbl_gen_inner_product_c1_noadd** **gbl_gen_inner_product_c1_addto**
2-norm	3	**gen_2_norm** **gbl_gen_2_norm**
Outer product	3	**gen_outer_product** **gen_outer_product_noadd** **gen_outer_product_addto** **gen_outer_product_c2** **gen_outer_product_c2_noadd** **gen_outer_product_c2_addto**
Matrix vector multiplication	3	**gen_matrix_vector_mult** **gen_matrix_vector_mult_noadd** **gen_matrix_vector_mult_addto** **gen_matrix_vector_mult_c1** **gen_matrix_vector_mult_c1_noadd** **gen_matrix_vector_mult_c1_addto**

Table 1 *(Continued)*

Operation	Chapter	Routines
Vector matrix multiplication	3	**gen_vector_matrix_mult** **gen_vector_matrix_mult_noadd** **gen_vector_matrix_mult_addto** **gen_vector_matrix_mult_c2** **gen_vector_matrix_mult_c2_noadd** **gen_vector_matrix_mult_c2_addto**
Infinity norm	3	**gen_infinity_norm**
Matrix multiplication	3	**gen_matrix_mult** **gen_matrix_mult_noadd** **gen_matrix_mult_addto** **gen_matrix_mult_t1** **gen_matrix_mult_t1_noadd** **gen_matrix_mult_t1_addto** **gen_matrix_mult_h1** **gen_matrix_mult_h1_noadd** **gen_matrix_mult_h1_addto** **gen_matrix_mult_t2** **gen_matrix_mult_t2_noadd** **gen_matrix_mult_t2_addto** **gen_matrix_mult_h2** **gen_matrix_mult_h2_noadd** **gen_matrix_mult_h2_addto** **gen_matrix_mult_t1_t2** **gen_matrix_mult_t1_t2_noadd** **gen_matrix_mult_t1_t2_addto**
Matrix multiplication with external storage	3	**gen_matrix_mult_ext**

Table 1 *(Continued)*

Operation	Chapter	Routines
Arbitrary elementwise sparse matrix operations	4	**sparse_matvec_setup** **sparse_matvec_mult** **sparse_mat_gen_mat_mult** **deallocate_sparse_matvec_setup** **sparse_vecmat_setup** **sparse_vecmat_mult** **gen_mat_sparse_mat_mult** **deallocate_sparse_vecmat_setup**
Arbitrary block sparse matrix operations	4	**block_sparse_setup** **block_sparse_matrix_vector_mult** **vector_block_sparse_matrix_mult** **block_sparse_mat_gen_mat_mult** **gen_mat_block_sparse_mat_mult** **deallocate_block_sparse_setup**
Grid sparse matrix operations	4	**grid_sparse_setup** **grid_sparse_matrix_vector_mult** **vector_grid_sparse_matrix_mult** **grid_sparse_mat_gen_mat_mult** **gen_mat_grid_sparse_mat_mult** **deallocate_grid_sparse_setup**
Gaussian elimination	5	**gen_lu_factor** **save_gen_lu** **restore_gen_lu** **gen_lu_solve** **gen_lu_solve_tra** **gen_lu_apply_l_inv** **gen_lu_apply_u_inv** **gen_lu_apply_l_inv_tra** **gen_lu_apply_u_inv_tra** **gen_lu_get_l** **gen_lu_get_u** **gen_lu_infinity_norm_inv** **deallocate_gen_lu**

Table 1 *(Continued)*

Operation	Chapter	Routines
Linear system solvers using Householder transformations	5	**gen_qr_factor** **save_gen_qr** **restore_gen_qr** **gen_qr_solve** **gen_qr_solve_tra** **gen_qr_apply_q** **gen_qr_apply_q_tra** **gen_qr_apply_r_inv** **gen_qr_apply_r_inv_tra** **gen_qr_get_r** **gen_qr_apply_p** **gen_qr_apply_p_inv** **gen_qr_zero_rows** **gen_qr_extract_diag** **gen_qr_deposit_diag** **gen_qr_infinity_norm_inv** **gen_qr_r_infinity_norm_inv** **deallocate_gen_qr**
Matrix inversion	5	**gen_gj_invert**
Gauss-Jordan system solver	5	**gen_gj_solve**
Gaussian elimination with external storage	5	**gen_lu_factor_ext** **gen_lu_solve_ext**
QR factorization and least squares solution with external storage	5	**gen_qr_factor_ext** **gen_qr_solve_ext**

Table 1 *(Continued)*

Operation	Chapter	Routines
Banded system factorization and solver routines (unified)	6	**gen_banded_factor** **gen_banded_solve** **deallocate_banded**
Banded system factorization and solver routines	6	**gen_tridiag_factor** **gen_tridiag_solve** **gen_tridiag_solve_factored** **gen_pentadiag_factor** **gen_pentadiag_solve** **gen_pentadiag_solve_factored** **block_tridiag_factor** **block_tridiag_solve** **block_tridiag_solve_factored** **block_pentadiag_factor** **block_pentadiag_solve** **block_pentadiag_solve_factored** **deallocate_banded_solve**
Krylov-based iterative solvers	7	**gen_iter_solve_setup** **gen_iter_solve** **deallocate_iter_solve**
Reduction to tridiagonal form and corresponding basis transformation	8	**sym_tred** **sym_to_tridiag** **tridiag_to_sym** **deallocate_sym_tred**
Eigenvalues of real symmetric tridiagonal matrices	8	**sym_tridiag_eigenvalues**
Eigenvectors of real symmetric tridiagonal matrices	8	**sym_tridiag_eigenvectors**

Table 1 *(Continued)*

Operation	Chapter	Routines
Eigensystem analysis of dense Hermitian matrices	8	**sym_tred_eigensystem**
Generalized eigensystem analysis of real symmetric matrices	8	**sym_tred_gen_eigensystem**
Eigensystem analysis using Jacobi rotations	8	**sym_jacobi_eigensystem**
Eigensystem analysis using a *k*-step Lanczos method	8	**sym_lanczos_setup** **sym_lanczos** **deallocate_sym_lanczos_setup**
Eigensystem analysis using a *k*-step Arnoldi method	8	**gen_arnoldi_setup** **gen_arnoldi** **deallocate_gen_arnoldi_setup**
Complex-to-complex FFT	9	**fft_setup** **fft** **fft_detailed** **deallocate_fft_setup**
Real-to-complex and complex-to-real FFT	9	**fft_setup** **fft_detailed** **deallocate_fft_setup**
Array conversion utilities for the FFT	9	**real_from_complex** **complex_from_real**
Explicit integration of ODEs (Runge-Kutta)	10	**ode_rkf_setup** **ode_rkf** **deallocate_ode_rkf_setup**
Dense simplex	11	**gen_simplex**

Table 1 *(Continued)*

Operation	Chapter	Routines
Fast RNG	12	**initialize_fast_rng** **fast_rng** **save_fast_rng_temps** **restore_fast_rng_temps** **fast_rng_state_field** **fast_rng_residue** **reinitialize_fast_rng** **deallocate_fast_rng**
VP RNG	12	**initialize_vp_rng** **vp_rng** **save_vp_rng_temps** **restore_vp_rng_temps** **vp_rng_state_field** **vp_rng_residue** **reinitialize_vp_rng** **deallocate_vp_rng**
Full histogram	13	**histogram**
Range histogram	13	**histogram_range**
Polyshift	14	**pshift_setup** **pshift_setup_looped** **pshift** **deallocate_pshift_setup**
All-to-all rotation	14	**all_to_all_setup** **all_to_all** **deallocate_all_to_all_setup**
All-to-all broadcast	14	**all_to_all_broadcast**
All-to-all reduction	14	**all_to_all_reduce**
Matrix transpose	14	**gen_matrix_transpose**

Table 1 *(Continued)*

Operation	Chapter	Routines
Sparse gather	14	**sparse_util_gather_setup** **sparse_util_gather** **deallocate_gather_setup**
Sparse scatter	14	**sparse_util_scatter_setup** **sparse_util_scatter** **deallocate_scatter_setup**
Sparse vector gather	14	**sparse_util_vec_gather_setup** **sparse_util_vec_gather** **deallocate_vec_gather_setup**
Sparse vector scatter	14	**sparse_util_vec_scatter_setup** **sparse_util_vec_scatter** **deallocate_vec_scatter_setup**
Block gather and scatter utilities	14	**block_gather** **block_scatter**
Mesh partitioning, pointer reordering	14	**generate_dual** **partition_mesh** **reorder_pointers** **renumber_pointers**
Partitioned gather	14	**part_gather_setup** **part_gather** **part_vector_gather** **deallocate_part_gather_setup**
Partitioned scatter	14	**part_scatter_setup** **part_scatter** **part_vector_scatter** **deallocate_part_scatter_setup**

Table 1 *(Continued)*

Operation	Chapter	Routines
Communication compiler	14	**comm_setup** **comm_get** **comm_send** **comm_send_add** **comm_send_and** **comm_send_max** **comm_send_min** **comm_send_or** **comm_send_xor** **comm_set_option** **deallocate_comm_setup**
Vector move (extract and deposit)	14	**vector_move** **vector_move_utils**
Computation of block cyclic permutations	14	**compute_fe_block_cyclic_perms**
Permutation along an axis	14	**permute_cm_matrix_axis_from_fe**
Send-to-NEWS and NEWS-to-send reordering	14	**send_to_news** **news_to_send**

1.2.2 Safety Mechanism

The CMSSL safety mechanism offers two basic features: it synchronizes the CM-5 processing elements and partition manager so that you can pinpoint the area of code that generated an error, and it performs error checking and reports errors at several levels of detail. You can use the CMSSL safety mechanism either by setting an environment variable or by using library calls within a program. The safety mechanism is described in Chapter 2.

1.3 Notes on Terminology

1.3.1 Data Types

Throughout this manual, the terms "real" and "complex" refer to both single-precision and double-precision data, unless otherwise noted. For example, an array described as a "complex CM array" can be either single-precision complex or double-precision complex.

1.3.2 Array Axis Descriptions

In array descriptions throughout this manual, row and column axes are distinguished as follows:

- "The axis that counts the rows," "the row axis," and *row_axis* refer to *axis 1* in Figure 1.
- "The axis that counts the columns," "the column axis," and *col_axis* refer to *axis 2* in Figure 1.

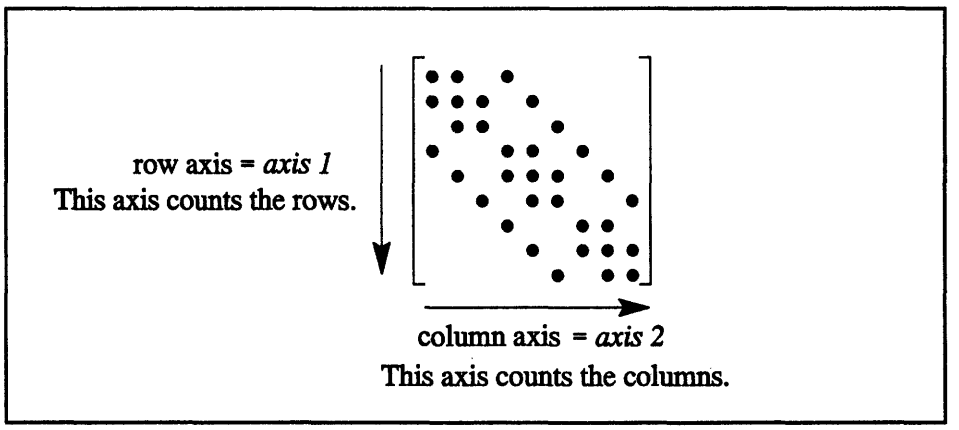

Figure 1. Row and column axes.

1.3.3 Processing Elements and Subgrids

Some sections of this manual contain implementation and performance information, and use the term *processing element*. The CM system component that serves

as the processing element depends on the CM Fortran execution model. Under the vector-units model, a processing element is a vector unit. Under the (SPARC) nodes model, a processing element is a SPARC processing node. The CM Fortran utility **CMF_NUMBER_OF_PROCESSORS** returns the number of processing elements available in the current execution model.

The mapping of array elements to processing elements is performed by the run-time system, and depends on the number of processing elements available to execute the program. You can control this mapping using the detailed axis descriptors of the **CMF$LAYOUT** directive, or using the CM Fortran utility **CMF_ALLOCATE_DETAILED_ARRAY**.

The elements of an array residing within one processing element are said to be *local* to that processing element. The *subgrid* associated with a processing element consists of the array elements that are local to the processing element, as well as any "garbage elements" (padding) required by the size constraints involved in mapping array elements to processing elements. The subgrids of an array are all the same size and are located at the same memory address within each processing element. The *subgrid extent* of an axis is the number of array elements in the subgrid along that axis.

An axis is *local* if the array elements along the axis reside within one processing element. An axis is *non-local* or *global* if the array elements along the axis span multiple processing elements.

In most cases, you do not need to understand the implementation of a CMSSL routine at the level of processing elements in order to use the routine. Implementation and performance information is provided for users who want to tune and optimize their code.

1.4 Data Types Supported

Table 2 shows the data types supported for each CMSSL operation. Within each subroutine call, all CM array arguments must match in data type and precision, unless the argument descriptions indicate otherwise.

Table 2. Data Types Supported by the CMSSL for CM Fortran.

Operation	Data Type				
	Integer	Real-4	Real-8	Cmplx-8	Cmplx-16
Inner product		X	X	X	X
2-norm		X	X	X	X
Outer product		X	X	X	X
Matrix vector multiplication		X	X	X	X
Vector matrix multiplication		X	X	X	X
Infinity norm		X	X	X	X
Matrix multiplication		X	X	X	X
Matrix multiplication with external storage		X	X	X	X
Arbitrary elementwise sparse matrix operations		X	X	X	X
Arbitrary block sparse matrix operations		X	X	X	X
Grid sparse matrix operations		X	X	X	X
Gaussian elimination		X	X	X	X
Linear solvers using Householder transformations		X	X	X	X
Matrix inversion		X	X	X	X
Gauss-Jordan system solver		X	X	X	X
Gaussian elimination with external storage		X	X	X	X
QR factorization with external storage		X	X	X	X
Banded system solvers		X	X	X	X
Iterative solvers		X	X		
Reduction to tridiagonal form		X	X	X	X
Corresponding basis transformation		X	X	X	X
Eigenvalues of real symmetric tridiagonal matrices		X	X		
Eigenvectors of real symmetric tridiagonal matrices		X	X		
Eigensystem analysis of dense Hermitian matrices		X	X	X	X
Generalized eigenanalysis of real symmetric matrices		X	X		
Eigensystem analysis using Jacobi rotations		X	X		
Selected eigenvalues/eigenvectors (Lanczos)		X	X		
Selected eigenvalues/eigenvectors (Arnoldi)		X	X		

Table 2 *(Continued)*

Operation	Data Type				
	Integer	Real-4	Real-8	Cmplx-8	Cmplx-16
Simple complex-to-complex FFT				X	X
Detailed complex-to-complex FFT				X	X
Real-to-complex FFT				X	X
Complex-to-real FFT				X	X
Array conversion utilities		X	X	X	X
ODEs (Runge-Kutta method)		X	X		
Dense simplex		X	X		
Fast RNG	X	X	X		
VP RNG	X	X	X		
Histogram	X	X	X		
Histogram range	X	X	X		
Polyshift	X	X	X	X	X
All-to-all rotation	X	X	X	X	X
All-to-all broadcast	X	X	X	X	X
All-to-all reduction	X	X	X	X	X
Matrix transpose	X	X	X	X	X
Sparse gather utility	X	X	X	X	X
Sparse scatter utility	X	X	X	X	X
Sparse vector gather utility	X	X	X	X	X
Sparse vector scatter utility	X	X	X	X	X
Block gather and scatter utilities	X	X	X	X	X
Mesh partitioning, pointer reordering	X				
Partitioned gather utility	X	X	X	X	X
Partitioned scatter utility	X	X	X	X	X
Communication compiler	X	X	X	X	X
Vector move	X	X	X	X	X
Computation of block cyclic permutations	X	X	X	X	X
Permutation along an axis	X	X	X	X	X
Send-to-NEWS, NEWS-to-send reordering	X	X	X	X	X

1.5 Support for Multiple Instances

Many CMSSL routines support *multiple instances*: that is, they allow you to perform multiple independent operations on different data sets concurrently. Table 3 shows which operations currently support multiple instances in CM Fortran.

Table 3. CMSSL Support for Multiple Instances in CM Fortran.

Operation	Instances	
	Single	Multiple
Inner product	X	X
2-norm	X	X
Outer product	X	X
Matrix vector multiplication	X	X
Vector matrix multiplication	X	X
Infinity norm	X	X
Matrix multiplication	X	X
Matrix multiplication with external storage	X	
Arbitrary elementwise sparse matrix operations	X	
Arbitrary block sparse matrix operations	X	
Grid sparse matrix operations	X	X
Gaussian elimination	X	X
Linear solvers using Householder transformations	X	X
Matrix inversion	X	
Gauss-Jordan system solver	X	
Gaussian elimination with external storage	X	
QR factorization with external storage	X	
Banded system solvers	X	X
Iterative solvers	X	

Table 3 *(Continued)*

Operation	Instances	
	Single	Multiple
Reduction to tridiagonal form	X	X
Corresponding basis transformation	X	X
Eigenvalues of real symmetric tridiagonal matrices	X	X
Eigenvectors of real symmetric tridiagonal matrices	X	X
Eigenanalysis of dense Hermitian matrices	X	X
Generalized eigenanalysis of real symmetric matrices	X	X
Eigenanalysis using Jacobi rotations	X	X
Selected eigenvalues/eigenvectors (Lanczos)	X	
Selected eigenvalues/eigenvectors (Arnoldi)	X	
Simple complex-to-complex FFT	X	
Detailed complex-to-complex FFT	X	X
Real-to-complex FFT	X	X
Complex-to-real FFT	X	X
Array conversion utilities	X	X
ODEs (Runge-Kutta method)	X	
Dense simplex	X	
Fast RNG		X
VP RNG		X
Histogram	X	
Histogram range	X	
Polyshift	X	X
All-to-all rotation	X	X
All-to-all broadcast	X	X
All-to-all reduction	X	X
Matrix transpose	X	X

Table 3 *(Continued)*

Operation	Instances	
	Single	Multiple
Sparse gather utility	X	
Sparse scatter utility	X	
Sparse vector gather utility	X	
Sparse vector scatter utility	X	
Block gather and scatter utilities	X	X
Mesh partitioning, pointer reordering	X	
Partitioned gather utility	X	
Partitioned scatter utility	X	
Communication compiler	X	X
Vector move	X	
Computation of block cyclic permutations	X	X
Permutation along an axis	X	X
Send-to-NEWS, NEWS-to-send reordering	X	X

1.5.1 Defining Multiple Independent Data Sets

To perform a CMSSL operation on multiple independent data sets concurrently, you must embed the multiple independent instances of each operand or result argument in a CM array. The axes of the array fall into two mutually exclusive groups:

- The *data axes* define the geometry of the individual instances of the operand or result.
- The *instance axes* label the multiple instances.

For example, Figure 2 illustrates a matrix vector multiplication operation in which four independent products are computed simultaneously. The four destination vectors are embedded in a two-dimensional CM array with one data axis (the vertical direction in the figure) and one instance axis; the four source vectors are similarly embedded in another CM array. The source matrices are embedded in

a three-dimensional CM array. The instances within each array are labeled 1 through 4.

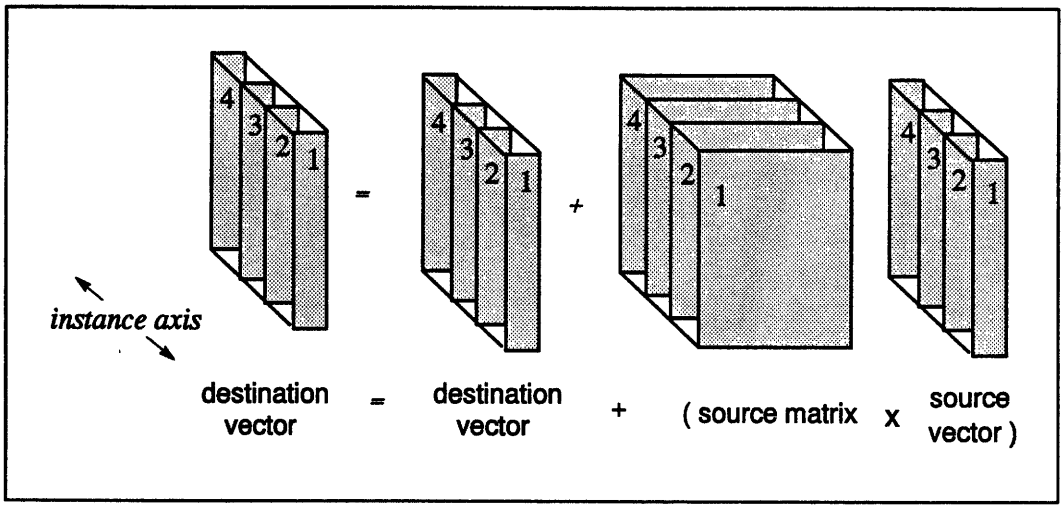

Figure 2. A multiple-instance matrix vector multiplication problem.

The structure defined by the data axes is the object of interest — the logical unit on which the routine operates. This structure is sometimes referred to as a *cell*. The instance axes define the geometry of the larger structure, or *frame*, in which the cells are embedded. The three-dimensional array shown in Figure 2 is a frame containing four two-dimensional cells.

The product of the extents of the instance axes is the total number of instances. The product of the extents of the data axes is the size of the cell.

1.5.2 Notation Used for CM Arrays and Embedded Matrices

Throughout this book, CM array names are printed in ***bold italics***. If a CM array contains multiple instances of a matrix, the same name is often used for the CM array and each matrix instance it contains. The name is printed in ***bold italics*** to denote the CM array, and in *italics* to denote the embedded matrix. For example, the text might refer to "a CM array ***A*** containing one or more matrices *A*."

1.5.3 Rules for Data Axes and Instance Axes

When you organize your data to form cells and frames for a multiple-instance operation, follow these rules:

- All operand and result arrays must have the same number of instance axes.
- Counting up through the axes of the arrays, starting with axis 1 and excluding the data axes, corresponding instance axes must occur in the same order in each operand or result array.
- The corresponding instance axes of each operand or result array must have identical extents. In some cases (indicated in the man pages for specific routines), corresponding instance axes must also have identical layout directives.
- The extents of the data axes must be defined so that the operation makes sense. For example, in matrix multiplication, the data axis extents of the operand and result matrices must obey the standard rules for axis extents in matrix multiplication. Specific requirements for data axis extents are provided in the descriptions of individual routines in later chapters.
- Except where explicitly noted, the CMSSL supports all combinations of layout directives for data axes and instance axes. The layout that results in best performance depends on the operation. However, in most cases performance is best when the cells are local to a processing element. To achieve this state, use the detailed axis descriptors of the CM Fortran **CMF$LAYOUT** directive. Instance axes are typically defined as parallel axes (**:news** or **:send**). Some of the descriptions of individual routines in this book contain specific information about optimizing array layouts.

Most CMSSL routines impose few or no restrictions on where the instance axes can occur in an array. This flexibility helps you avoid the transposes you might have to perform if, for example, instance axes were required to be the last axes of an array. (Transposes involve communication, and therefore exact a performance price.)

1.5.4 Specifying Single-Instance vs. Multiple-Instance Operations

CMSSL routines that support multiple instances have the same calling sequence for single-instance and multiple-instance operations. The methods you must use to specify single-instance and multiple-instance operations depend on the type of

routine you are calling. Specific information is provided in the man pages included in Chapters 3 through 14. Several examples are discussed below.

Example 1. Matrix Vector Multiplication

When you call the matrix vector multiplication routine, **gen_matrix_vector_mult**, the dimensionality of the arguments you supply determines whether the routine performs a single-instance or multiple-instance operation, as follows:

- To perform a single-instance operation, specify each vector argument as a one-dimensional CM array and each matrix argument as a two-dimensional CM array. (Alternatively, you declare these arguments to have more dimensions, but all instance axes must have extent 1.)
- To perform a multiple-instance operation, embed the multiple instances of each vector argument in a CM array of rank greater than 1, and embed the multiple instances of each matrix argument in a CM array of rank greater than 2.

This routine requires you to specify which axes you are using as data axes for each matrix or vector argument. Chapter 3 provides details.

Example 2: Solving Linear Systems Using Householder Transformations (In-Core *QR* Factorization and Solver Routines)

Figure 3 through Figure 5 show how a multiple-instance problem is set up for the in-core routines that solve linear systems using Householder transformations (the "*QR*" routines). The three-dimensional array *A* in Figure 3 contains four matrices to be factored (four instances of the matrix *A*). Each matrix *A* has dimensions *m* x *n*, and is (optionally) contained in a larger matrix embedded in the array *A*. The data axes, which count the rows and columns of the matrices, can be any two axes of the array *A*; you need not use the first and second axes for this purpose.

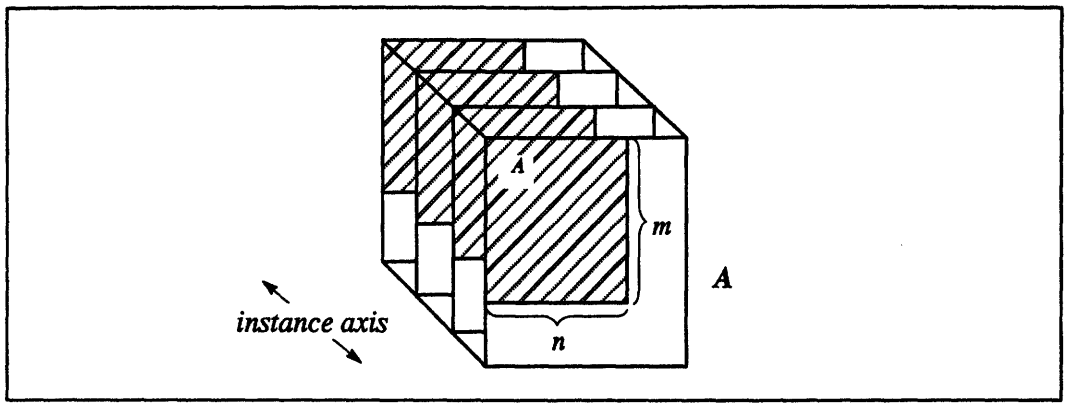

Figure 3. The matrices to be factored have size *m* X *n* and may be contained in larger matrices embedded in *A*.

Figure 4 shows four instances of the linear system $AX = B$. The right-hand-side matrices B are embedded in the array B; you must use the same axes to count the rows and columns of the instances in B as in A. The parameter r represents the number of columns, or right-hand-side vectors, in each matrix B; thus, each B has size m X r.

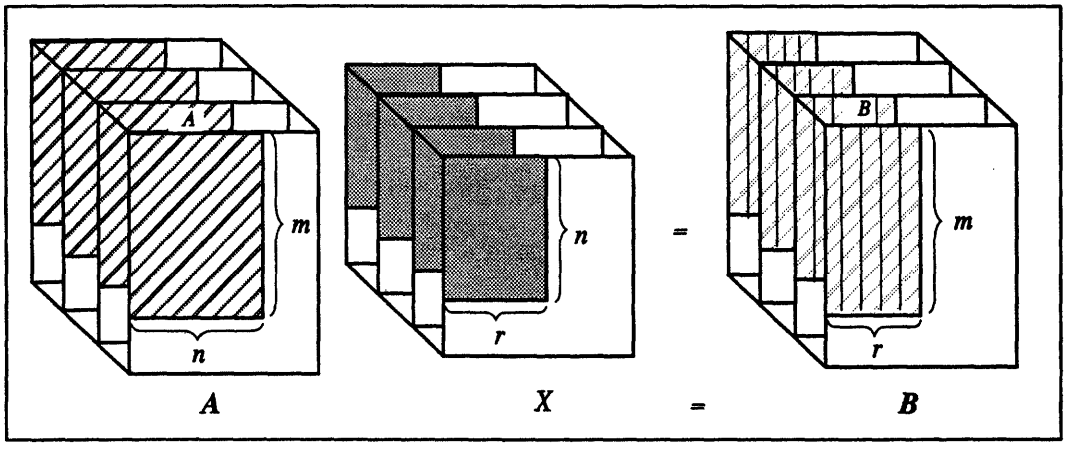

Figure 4. The linear systems $AX = B$

Upon completion of the solver routine, the first n rows of each matrix B are overwritten with the least squares solution to $AX = B$, as shown in Figure 5. For more information about the *QR* routines, refer to Chapter 5.

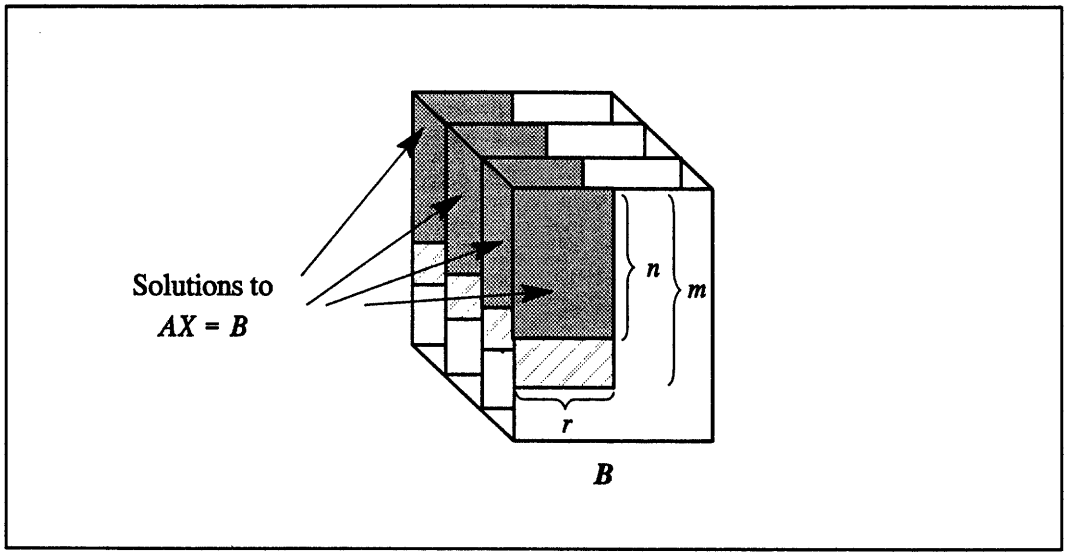

Figure 5. The matrices embedded in the array *B* are partially overwritten with the solutions when the solver routine completes.

Example 3: Fast Fourier Transforms

When you call the Detailed complex-to-complex FFT (CCFFT) routine, you can supply a multidimensional CM array and specify whether you want to perform a forward transform, an inverse transform, or no transform along each axis. You can also specify axes along which no transform is performed but address bits are reversed. The axes that are transformed or bit-reversed are the data axes, and define the cell; the axes along which you perform no transformation are the instance axes.

The Simple CCFFT performs a transform along each axis of the supplied array, and therefore does not support multiple instances.

In addition to the CCFFT, the CMSSL provides a real-to-complex FFT (RCFFT) for computing the Fourier transform of real data, and a complex-to-real FFT (CRFFT) for the transformation of *conjugate symmetric* complex sequences. The Fourier Transform of a real or conjugate symmetric sequence can be computed using half the storage and half the arithmetic of a CCFFT. The RCFFT and CRFFT support multiple instances in a manner similar to that of the CCFFT.

For detailed information about multidimensional and multiple-instance FFTs, refer to Chapter 9.

Example 4: Polyshift

The polyshift routines perform multidirectional and/or multidimensional sequences of **CSHIFT** and/or **EOSHIFT** operations on a CM array. The axes along which shifts are performed are the data axes; all other axes are the instance axes. Chapter 14 provides details.

Example 5: All-to-All Rotation

The all-to-all rotation routines perform a stepwise rotation along a selected axis of an arbitrary array. Every array element visits every location along the axis. Each step corresponds to a data permutation along the axis, and is typically followed by computations.

In the all-to-all rotation, the axis along which the rotation occurs is the data axis, and all other axes are instance axes. Each one-dimensional cell undergoes an all-to-all rotation. Within a multidimensional array, the multiple instances of the all-to-all rotation have different permutation patterns. For example, if the elements of a two-dimensional array are rotated along the rows, each row may have a different permutation path.

For more information about the all-to-all rotation, refer to Chapter 14.

Example 6: Random Number Generators

The random number generators support multiple instances in the sense that they produce multiple streams of random numbers (one stream per processing element or one stream per array element). Chapter 12 provides details.

1.6 Numerical Stability for the Linear Algebra Routines

Some of the descriptions of linear algebra routines in later chapters include information about *numerical stability.* In this book, numerical stability is defined in the standard way: an algorithm is stable if the computed result is the exact solution of a slightly different problem. For example, if A is the input matrix, the computed result is the true result corresponding to the matrix $A + E$, where E is

small in norm compared with A.* Most of the algorithms used by the CMSSL are numerically stable in this sense. However, a few are only *conditionally stable*, which means that the numerical stability may depend on the condition number of the problem. For information about the stability of specific routines, refer to the descriptions of the routines in later chapters.

1.7 Numerical Complexity

The following table lists the numbers of floating-point operations performed by some of the CMSSL routines.

Table 4. Number of Floating-Point Operations (flops) Performed by CMSSL Routines

Routine	# flops (real operands)	# flops (complex operands)
Vector length = q, number of instances = I:		
gen_inner_product	$2qI$	$8qI$
gen_inner_product_noadd	$(2q-1)I$	$(8q-2)I$
gen_inner_product_addto	$2qI$	$8qI$
gen_inner_product_c1	$2qI$	$8qI$
gen_inner_product_c1_noadd	$(2q-1)I$	$(8q-2)I$
gen_inner_product_c1_addto	$2qI$	$8qI$
Product of axis extents = Q:		
gbl_gen_inner_product	$2Q$	$8Q$
gbl_gen_inner_product_noadd	$2Q-1$	$8Q-2$
gbl_gen_inner_product_addto	$2Q$	$8Q$
gbl_gen_inner_product_c1	$2Q$	$8Q$
gbl_gen_inner_product_c1_noadd	$2Q-1$	$8Q-2$
gbl_gen_inner_product_c1_addto	$2Q$	$8Q$

* For a more formal definition, see Golub, G. H. and C. F. Van Loan, *Matrix Computations*, 2d ed. (Baltimore: Johns Hopkins University Press, 1989).

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Vector length = q, number of instances = I:		
gen_2_norm	$[(2q-1)+8]I^*$	$(4q-1)+8]I^*$
Product of axis extents = Q:		
gbl_gen_2_norm	$(2Q-1)+8^*$	$(4Q-1)+8^*$
Matrix size = p x q, vector lengths = p and q, number of instances = I:		
gen_outer_product	$2pqI$	$8pqI$
gen_outer_product_noadd	pqI	$6pqI$
gen_outer_product_addto	$2pqI$	$8pqI$
gen_outer_product_c2	$2pqI$	$8pqI$
gen_outer_product_c2_noadd	pqI	$6pqI$
gen_outer_product_c2_addto	$2pqI$	$8pqI$
Matrix size = p x q, vector lengths = p and q, number of instances = I:		
gen_matrix_vector_mult	$2pqI$	$8pqI$
gen_matrix_vector_mult_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_matrix_vector_mult_addto	$2pqI$	$8pqI$
gen_matrix_vector_mult_c1	$2pqI$	$8pqI$
gen_matrix_vector_mult_c1_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_matrix_vector_mult_c1_addto	$2pqI$	$8pqI$
Matrix size = p x q, vector lengths = p and q, number of instances = I:		
gen_vector_matrix_mult	$2pqI$	$8pqI$
gen_vector_matrix_mult_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_vector_matrix_mult_addto	$2pqI$	$8pqI$
gen_vector_matrix_mult_c2	$2pqI$	$8pqI$
gen_vector_matrix_mult_c2_noadd	$(2pq-p)I$	$(8pq-2p)I$
gen_vector_matrix_mult_c2_addto	$2pqI$	$8pqI$

* The additional 8 flops are for the square root operation.

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix size = m x n, number of instances = I:		
gen_infinity_norm		
Matrix sizes = p x q and q x r, number of instances = I:		
gen_matrix_mult	$2pqrI$	$8pqrI$
gen_matrix_mult_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_addto	$2pqrI$	$8pqrI$
gen_matrix_mult_ext	$2pqr$	$8pqr$
Matrix sizes = q x p and q x r, number of instances = I:		
gen_matrix_mult_t1	$2pqrI$	$8pqrI$
gen_matrix_mult_t1_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_t1_addto	$2pqrI$	$8pqrI$
gen_matrix_mult_h1	$2pqrI$	$8pqrI$
gen_matrix_mult_h1_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_h1_addto	$2pqrI$	$8pqrI$
Matrix sizes = p x q and r x q, number of instances = I:		
gen_matrix_mult_t2	$2pqrI$	$8pqrI$
gen_matrix_mult_t2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_t2_addto	$2pqrI$	$8pqrI$
gen_matrix_mult_h2	$2pqrI$	$8pqrI$
gen_matrix_mult_h2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_h2_addto	$2pqrI$	$8pqrI$
Matrix sizes = q x p and r x q, number of instances = I:		
gen_matrix_mult_t1_t2	$2pqrI$	$8pqrI$
gen_matrix_mult_t1_t2_noadd	$(2pqr-pr)I$	$(8pqr-2pr)I$
gen_matrix_mult_t1_t2_addto	$2pqrI$	$8pqrI$
Matrix sizes = p x q and q x r:		
gen_matrix_mult_ext	$2pqr$	$8pqr$

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Sparse matrix in packed vector form has length n:		
sparse_matvec_mult	$2n$	$8n$
sparse_vecmat_mult	$2n$	$8n$
Sparse matrix in packed vector form has length n; matrix has r columns:		
sparse_mat_gen_mat_mult	$2nr$	$8nr$
gen_mat_sparse_mat_mult	$2nr$	$8nr$
Block sparse matrix has p blocks of size m x n:		
block_sparse_matrix_vector_mult	$2mnp$	$8mnp$
vector_block_sparse_matrix_mult	$2mnp$	$8mnp$
block_sparse_mat_gen_mat_mult	$2mnp$	$8mnp$
gen_mat_block_sparse_mat_mult	$2mnp$	$8mnp$
Block size = p x q; product of extents of grid axes = N; number of instances = I:		
grid_sparse_matrix_vector_mult	$2pqNI$	$8pqNI$
vector_grid_sparse_matrix_mult	$2pqNI$	$8pqNI$
Block sizes = p x q, q x r, and p x r; product of extents of grid axes = N; number of instances = I:		
grid_sparse_mat_gen_mat_mult	$2pqrNI$	$8pqrNI$
gen_mat_grid_sparse_mat_mult	$2pqrNI$	$8pqrNI$

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
Matrix size = m x n, r = number of right-hand sides, I = number of instances:		
gen_lu_factor	$[m-(n/3)]n^2I$	$4[m-(n/3)]n^2I$
gen_lu_solve	$2r(2m-n)nI$	$8r(2m-n)nI$
gen_lu_solve_tra	$2r(2m-n)nI$	$8r(2m-n)nI$
gen_lu_apply_l_inv	$r(2m-n)nI$	$4r(2m-n)nI$
gen_lu_apply_u_inv	$r(2m-n)nI$	$4r(2m-n)nI$
gen_lu_apply_l_inv_tra	$r(2m-n)nI$	$4r(2m-n)nI$
gen_lu_apply_u_inv_tra	$r(2m-n)nI$	$4r(2m-n)nI$
Matrix size = m x n, r = number of right-hand sides, I = number of instances:		
gen_qr_factor	$2[m-(n/3)]n^2I$	$8[m-(n/3)]n^2I$
gen_qr_solve	$r(4m-n)nI$	$4r(4m-n)nI$
gen_qr_solve_tra	$r(4m-n)nI$	$4r(4m-n)nI$
gen_qr_apply_q	$2rmnI$	$8rmnI$
gen_qr_apply_q_tra	$2rmnI$	$8rmnI$
gen_qr_apply_r_inv	$r(2m-n)nI$	$4r(2m-n)nI$
gen_qr_apply_r_inv_tra	$r(2m-n)nI$	$4r(2m-n)nI$
Matrix size = n x n, r = number of right-hand sides:		
gen_gj_invert	$2n^3$	$8n^3$
gen_gj_solve	$2/3(n^3+2n^2r)$	$8/3(n^3+8n^2r)$
Matrix size = n x n, r = number of right-hand sides:		
gen_lu_factor_ext	$(2/3)n^3$	$(8/3)n^3$
gen_lu_solve_ext	$2n^2r$	$8n^2r$
Matrix size = m x n, r = number of right-hand sides:		
gen_qr_factor_ext	$2[m-(n/3)]n^2$	$8[m-(n/3)]n^2$
gen_qr_solve_ext	$r(4m-n)n$	$4r(4m-n)n$

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
gen_banded_factor	See **gen_tridiag_factor**	
gen_banded_solve	and related routines, below	

n = number of equations or block equations (=length of diagonal)
I = number of instances; r = number of right-hand sides per system
b = block size = length of axis 1 of a, b, and c in **block_tridiag_factor**
p = number of processing elements spanned by axis *axis*
q = product of numbers of processing elements spanned by the instance axes (pq = the total number of processing elements)

The flop count for each **_solve** routine is the sum of the flop counts for the corresponding **_factor** and **_solve_factored** routines.

gen_tridiag_factor*		
CMSSL_pipeline_ge	$8nI$	$26nI$
CMSSL_pge_piv[_val]	$12nI$	$41nI$
CMSSL_substr_cr	$I(14n+14p\log p)$	$I(54n+54p\log p)$
CMSSL_substr_pge	$I(14n+8p)$	$I(54n+26p)$
CMSSL_substr_transp	$I(14n+7p)$	$I(54n+24p$
CMSSL_substr_bcr	$14nI+14\max(pq\log p, (p-1)I)$ (real)	
	$54nI+54\max(pq\log p, (p-1)I)$ (complex)	
gen_tridiag_solve_factored*		
CMSSL_pipeline_ge	$6nIr$	$22nIr$
CMSSL_pge_piv[_val]	$7nIr$	$30nIr$
CMSSL_substr_cr	$Ir(9n+4\log p)$	$Ir(38n+16p\log p)$
CMSSL_substr_pge	$Ir(9n+5p)$	$Ir(38n+22p)$
CMSSL_substr_transp	$Ir(9n+5p)$	$Ir(38n+22p)$
CMSSL_substr_bcr	$9nIr+9\max(pq\log p, (p-1)Ir)$ (real)	
	$38nIr+38\max(pq\log p, (p-1)Ir$ (complex)	

* Whenever the equation axis (axis *axis*) is local to a processing element, the flop count is equal to that of **CMSSL_pipeline_ge**. Furthermore, some flop counts involving p and q are valid only for problems that "fit the machine" (problems that do not require garbage masks).

Table 4 *(Continued)*

Routine	# flops (real operands)	# flops (complex operands)
block_tridiag_factor		
CMSSL_pipeline_ge	$6nIb^2$	$24nIb^2$
CMSSL_substr_cr	$Ib^3(14n+14p\log p)$	$Ib^3(56n+56p\log p)$
CMSSL_substr_pge	$Ib^3(14n+6p)$	$Ib^3(56n+24p)$
CMSSL_substr_bcr	$b^3(14nI+14\max(pq\log p, (p-1)I))$ (real)	
	$b^3(56nI+56\max(pq\log p, (p-1)I))$ (complex)	
block_tridiag_solve_factored		
CMSSL_pipeline_ge	$6nIrb^2$	$24nIrb^2$
CMSSL_substr_cr	$Irb^2(10n+10p\log p)$	$Irb^2(40n+40p\log p)$
CMSSL_substr_pge	$Irb^2(10n+6p)$	$Irb^2(40n+24p)$
CMSSL_substr_bcr	$b^2(10nIr+10\max(pq\log p, (p-1)Ir))$ (real)	
	$b^2(40nIr+40\max(pq\log p, (p-1)Ir))$ (complex)	

Matrix size = n; number of vectors to be transformed = r:

sym_tred	$(4/3)n^3$
sym_to_tridiag	$2n^2r$
tridiag_to_sym	$2n^2r$

N = length of active axis; I = product of other axis lengths:

fft	$5N\log N$
fft_detailed	$5IN\log N$

m = number of rows, n = number of columns:

gen_simplex	approx. $2mn$ flops/iteration

* Whenever the equation axis (axis *axis*) is local to a processing element, the flop count is equal to that of **CMSSL_pipeline_ge**. Furthermore, some flop counts involving p and q are valid only for problems that "fit the machine" (problems that do not require garbage masks).

1.8 CM Fortran Performance Enhancements with CMSSL

The following CM Fortran intrinsic and utilities yield better performance when the program is linked with CMSSL:

MATMUL
CMF_RANDOM
CMF_ORDER
CMF_SORT
CMF_RANK

For example, when CMSSL is linked in, a call to **CMF_random** generates a call to **fast_rng**; a call to **CMF_randomize**(*seed*) generates a call to **initialize_fast_rng** with the same seed value and default parameters for *table_lag*, *short_lag*, and *width*. (The CMSSL random number generators are described in Chapter 12.)

NOTE

Since **CMF_RANDOMIZE** generates a call to **initialize_fast_rng** when CMSSL is linked in, if you call **initialize_fast_rng** explicitly after calling **CMF_RANDOMIZE** (for example, to change the CMSSL Fast RNG parameters), you will receive **initialize_fast_rng** return code -1, which indicates that a prior initialization was overwritten. This code is informational only and does not indicate an error.

Chapter 2

Using the CM Fortran CMSSL Interface

This chapter contains information about running CM Fortran programs that call CMSSL routines. The following topics are included:

- creating a CM Fortran CMSSL program
- using the CMSSL safety mechanism
- on-line sample code and man pages
- further reading

2.1 Creating a CM Fortran CMSSL Program

To use the CMSSL from within a CM Fortran program, follow these steps:

1. Read the *CMSSL Release Notes* for information specific to the CMSSL release you are using and for updates to the manual. Important information such as switches for linking with the CMSSL may change from release to release.

2. Include the header file `/usr/include/cm/cmssl-cmf.h` if you are calling a CMSSL function or a CMSSL subroutine that uses predefined symbolic constants.

3. Place calls to CMSSL routines into CM Fortran code.

4. Use the CM Fortran `cmf` command to compile your code.

5. Use the `-lcmsslcm5` or `-lcmsslcm5vu` switch to link with the CMSSL for the CM-5.

The rest of this chapter discusses these steps in detail.

2.1.1 Including the CMSSL Header File

A CM Fortran program that calls the CMSSL can access the appropriate header file if you place the following line at the top of any program unit that makes a CMSSL call:

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

This file declares the return values of CMSSL functions and defines symbolic constants used as parameter values for some CMSSL routines.

The `INCLUDE` line is *required* only in CM Fortran code that contains CMSSL function calls or uses predefined CMSSL symbolic constants. However, we recommend that you include the header file wherever you use the CMSSL. It is easier to do this at the outset than to remember to add the `INCLUDE` line should you add a CMSSL function call to your code in the future. Also, in the future, the library is likely to make greater use of symbolic constants, which require the definitions provided in the header file.

If the CM Fortran compiler cannot find the CMSSL include files, check your partition manager for the existence of a path to the appropriate directory. If the files appear to be missing, consult your system administrator or your Thinking Machines Corporation customer service representative.

2.1.2 Calling CMSSL Routines

To invoke a CMSSL routine from within a CM Fortran program, first make sure you are using compatible versions of CM Fortran and the CMSSL. The *CMSSL Release Notes* shipped with the version you are using include a section describing which version of the CM Fortan compiler is required. Treat the CMSSL routine as you would any other subroutine or function.

2.1.3 Compiling and Linking

After writing a CM Fortran program that calls CMSSL routines, compile it and link it with the library. Compiling a CM Fortran CMSSL program is the same as compiling other CM Fortran programs: use the `cmf` command. To compile the program *program* on a CM-5 and link it with the CMSSL for the CM-5, issue one of the following command lines at the UNIX prompt:

For the CM Fortran vector-units model:

```
%cmf -cm5 -vu -o program program.fcm -lcmsslcm5vu
```

For the CM Fortran (SPARC) nodes model:

```
%cmf -cm5 -sparc -o program program.fcm -lcmsslcm5
```

Using the Correct Version of CMOST

The CMSSL is a layered product. Any CMSSL version requires a specific CMOST version and a specific CM Fortran version. If these dependencies are not observed, proper operation of the CMSSL routines is unlikely. Consult the most current version of the *CMSSL Release Notes* to find out which versions of CM Fortran and CMOST are required by the current CMSSL.

2.1.4 Executing CMSSL Programs

Execute a CM Fortran CMSSL program just as you would any compiled CM Fortran program.

2.2 A Note about Aligning Arrays

Many CMSSL routines fail when supplied with a CM array that has been aligned (using the CM Fortran **CMF$ ALIGN** directive) to an array of higher rank. CM Fortran reuses the geometry of the **ALIGN** target, rather than the **ALIGN** source, causing the CMSSL array rank checks to fail. It is recommended that you avoid using arrays that are aligned to arrays of higher rank.

2.3 Using the CMSSL Safety Mechanism

You can use the CMSSL safety mechanism in two ways:

- by setting the environment variable **CMSSL_SAFETY**
- by using the calls **CMSSL_get_safety** and **CMSSL_set_safety** in a program

2.3.1 Safety Mechanism Features

The CMSSL safety mechanism offers two basic features: it synchronizes the CM-5 parallel processing elements and partition manager so that you can pinpoint the area of code that generated an error, and it performs error checking and reports errors at several levels of detail.

Synchronization

The CM-5 parallel processing elements and partition manager operate asynchronously with respect to one another. Without the CMSSL safety mechanism, an error that occurs in the parallel processing elements is not reported to the partition manager until the next time the partition manager requests information from or checks the status of the elements. Such a request or status check is known as an *implicit synchronization* because it has the side effect of synchronizing the processing elements and partition manager, allowing the processing elements to report any accumulated errors. When an implicit synchronization occurs, there is no way to tell exactly when the reported error occurred, or which module of code produced it.

The CMSSL safety mechanism addresses this problem by forcing *explicit synchronization* between the parallel processing elements and the partition manager before, after, and within each CMSSL call in your code. The safety mechanism traps and reports errors, indicating when the errors occurred in relation to the synchronization points.

Error Checking and Reporting

The safety mechanism can perform error checking and generate run-time error information at several levels of detail. You can turn safety checking on at any level during all or part of a program. One level checks for errors in the usage and

arguments of the CMSSL calls in your program; a more detailed level also checks for errors generated by internal CMSSL routines. Examples of errors found and reported by the safety mechanism include the following:

- A supplied or returned data element that should be numerical is not; for example, it is identified as a "Not a Number" (NaN) or as infinity. (NaNs are defined in the IEEE Standard for Binary Floating-Point Arithmetic.)
- The code generates a division by 0 (for example, because of bad data, a user error, or an internal software problem).
- The code references a memory location that it has not initialized. The safety mechanism identifies this kind of error by writing NaNs to all allocated processing element memory. If the code references a memory location without first explicitly assigning it a numerical value, the NaN at that location causes further errors that make the original erroneous reference easy to find. (This is the same strategy used by CM Fortran safety checking when you include the **-safety=10** option on the **cmf** command line.)

As more debugging checks and safety levels are added in future releases, CMSSL safety checking will become more exhaustive.

2.3.2 Levels of Error Checking

The CMSSL safety mechanism currently provides the following levels:

0 (**off**)	Turns off the safety mechanism. Explicit synchronization and error checking are not performed. This level is appropriate for production runs of code that has already been thoroughly tested.
1 (**on**)	Checks for and reports errors caused by incorrect usage or arguments in high-level-language CMSSL calls. Performs explicit synchronization before and after each call and locates each error with respect to the synchronization points. This safety level is appropriate during program development or during runs for which a small performance penalty can be tolerated.
9 (**full**)	Checks for and reports all level 1 errors, and in addition any errors generated by the lower levels of code that are called by the high- level-language CMSSL calls. Performs

explicit synchronization in these lower levels of code and locates each error with respect to the synchronization points. This level performs all implemented error checking and exacts a very high performance price. It is appropriate for detailed debugging when a problem occurs. If you cannot analyze and correct the problem, provide your local site coordinator, applications engineer, or Thinking Machines Corporation customer service representative with the output generated by level 9 safety checking.

At levels 1 and 9, some safety mechanism error messages are displayed at the terminal when you run the program; other information appears in the backtrace when you use a debugger such as **cmdbx**.

If you report a software problem to your local site coordinator, applications engineer, or Thinking Machines Corporation customer service representative, you may be asked to run your program with the CMSSL safety mechanism enabled at a level other than 0, 1, or 9. These additional levels are used for pinpointing problems in the internal software or for obtaining internal status information.

2.3.3 Setting the CMSSL Safety Environment Variable

To set the CMSSL safety level using the **CMSSL_SAFETY** environment variable, issue the command

```
setenv CMSSL_SAFETY { 0 | 1 | 9 | off | on | full }
```

choosing one of the listed options. As indicated above, 0 is equivalent to **off**, 1 to **on**, and 9 to **full**.

The advantage of using the **CMSSL_SAFETY** environment variable is that you can set or change the safety level without recompiling your code.

2.3.4 Using CMSSL Safety from within a Program

To set the CMSSL safety level, issue the following call and specify the desired level in the integer argument *n*:

cmssl_set_safety (*n*)

To obtain the current CMSSL safety level, issue the following call:

cmssl_get_safety ( )

The advantage of using these calls from within a program is that you can set or obtain the safety level at any point within your code. However, you must recompile the code each time you change these calls.

NOTE

The inner product, 2-norm, outer product, matrix vector multiplication, vector matrix multiplication, and matrix multiplication routines described in Chapter 3 perform error checking only when the CMSSL safety mechanism is on. Therefore, we strongly recommend that you turn CMSSL safety on when testing new programs that call these routines.

2.4 On-Line Sample Code and Man Pages

Included with the CMSSL are sample on-line programs that demonstrate how to call each CMSSL routine. You are encouraged to experiment with these sample programs. Also included with the CMSSL are on-line man pages for all routines.

The on-line sample programs are located in subdirectories of the CMSSL examples directory. The default location for the examples directory is

`/usr/examples/cmssl`.

Examples for the operation *operation* are included in the subdirectory

operation`/cmf`

or

operation`/`*sub-operation*`/cmf`

of the examples directory. For example, the sample code for the routine that performs eigenvector analysis using the Jacobi method is located in the subdirectory

```
eigen/jacobi/cmf
```

of the examples directory. If you do not find the on-line examples in `/usr/examples/cmssl`, check with your system administrator (or the person who installs the CMSSL at your site) to find out where they were installed.

To read the on-line man page for a routine, enter the command

man *routine_name*

at the UNIX prompt.

2.5 Further Reading

For more detailed information about CM Fortran, consult the latest versions of the books listed below.

- *Getting Started in CM Fortran*

  Offers a brief introduction to using the CM Fortran language.

- *CM Fortran Programming Guide*

  Offers a more detailed, task-oriented introduction to all the major features of the CM Fortran language.

- *CM Fortran User's Guide*

  Includes complete descriptions of how to compile, link, and execute CM Fortran code, as well as how to use the CM Fortran Utility Library.

- *CM Fortran Utility Library Reference Manual*

  Provides reference and usage information about the procedures in the CM Fortran Utility Library.

- *CM Fortran Reference Manual*

  The definitive description of the CM Fortran language.

Chapter 3

Dense Matrix Operations

This chapter describes the CM Fortran interface to the CMSSL dense matrix operations. One section is devoted to each of the following:

- inner product
- 2-norm
- outer product
- matrix vector multiplication
- vector matrix multiplication
- infinity norm
- matrix multiplication
- matrix multiplication with external storage
- references

NOTE

The inner product, 2-norm, outer product, matrix vector multiplication, vector matrix multiplication, and matrix multiplication routines perform error checking only when the CMSSL safety mechanism is on. Therefore, we strongly recommend that you turn CMSSL safety on when testing new programs that call these routines.

3.1 Inner Product

The *multiple-instance* inner product routines compute one or more instances of an inner product of two vectors. The inner product either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array (with the results placed in the destination CM array).

Given CM arrays x, y, z, and u containing multiple instances of the vectors x, y, z, and u, respectively, the multiple-instance inner product routines perform the operations listed below for each instance.

Routine	Operation	Data Types
gen_inner_product	$z = z + x^{T}y$	real or complex
gen_inner_product_noadd	$z = x^{T}y$	real or complex
gen_inner_product_addto	$z = u + x^{T}y$	real or complex
gen_inner_product_c1	$z = z + x^{H}y$	complex only
gen_inner_product_c1_noadd	$z = x^{H}y$	complex only
gen_inner_product_c1_addto	$z = u + x^{H}y$	complex only

Each *single-instance* (**gbl_**) inner product routine computes the global inner product over all axes of two source CM arrays. The inner product either overwrites the destination front-end scalar variable, is added to the destination front-end scalar variable, or is added to a second front-end scalar variable (with the results placed in the destination front-end scalar variable).

Given CM arrays x and y, and scalars α and β, the single-instance inner product routines perform the operations listed below. In these formulas, the inner product occurs over *all* axes of the arrays x and y.

Routine	Operation	Data Types
gbl_gen_inner_product	$\alpha = \alpha + x^{T}y$	real or complex
gbl_gen_inner_product_noadd	$\alpha = x^{T}y$	real or complex
gbl_gen_inner_product_addto	$\alpha = \beta + x^{T}y$	real or complex

Version 3.1, June 1993

gbl_gen_inner_product_c1	$\alpha = \alpha + x^H y$	complex only
gbl_gen_inner_product_c1_noadd	$\alpha = x^H y$	complex only
gbl_gen_inner_product_c1_addto	$\alpha = \beta + x^H y$	complex only

Details are provided in the man page that follows.

Inner Product

The multiple-instance inner product routines compute one or more instances of an inner product of two vectors. The single-instance (**gbl_**) inner product routines compute the global inner product over all axes of two source CM arrays.

SYNTAX

gen_inner_product	(*z, x, y, x_vector_axis, y_vector_axis, ier*)
gen_inner_product_noadd	(*z, x, y, x_vector_axis, y_vector_axis, ier*)
gen_inner_product_addto	(*z, x, y, u, x_vector_axis, y_vector_axis, ier*)
gen_inner_product_c1	(*z, x, y, x_vector_axis, y_vector_axis, ier*)
gen_inner_product_c1_noadd	(*z, x, y, x_vector_axis, y_vector_axis, ier*)
gen_inner_product_c1_addto	(*z, x, y, u, x_vector_axis, y_vector_axis, ier*)
gbl_gen_inner_product	(α, *x, y, ier*)
gbl_gen_inner_product_noadd	(α, *x, y, ier*)
gbl_gen_inner_product_addto	(α, *x, y,* β, *ier*)
gbl_gen_inner_product_c1	(α, *x, y, ier*)
gbl_gen_inner_product_c1_noadd	(α, *x, y, ier*)
gbl_gen_inner_product_c1_addto	(α, *x, y,* β, *ier*)

ARGUMENTS

z — CM array of the same data type and precision as *x* and *y*, and rank one less than that of *x* and *y*. The axes of *z* must match the instance axes of *x* and *y* in order of declaration and extents. Thus, each pair of vectors in *x* and *y*, respectively, corresponds to a single value *z* in *z*.

x — When you call one of the multiple-instance (**gen_**) routines, *x* must be a real or complex CM array of rank ≥ 2, with at least one

non-serial instance axis. Contains one or more instances of *x*, the first vector in the pair of vectors whose inner product is to be computed. (For a single-instance problem, declare any instance axes to have extent 1.) Axis *x_vector_axis* of *x* and axis *y_vector_axis* of *y* must have the same extent. The remaining axes of *x* and *y* (the instance axes) must match in order of declaration and extents.

When you call one of the single-instance (**gbl_gen_**) routines, *x* must be a real or complex CM array of rank ≥ 1.

y

When you call one of the multiple-instance (**gen_**) routines, *y* must be a CM array of the same rank and data type as *x*, with at least one non-serial instance axis. Contains one or more instances of *y*, the second vector in the pair of vectors whose inner product is to be computed. (For a single-instance problem, declare any instance axes to have extent 1.) Axis *x_vector_axis* of *x* and axis *y_vector_axis* of *y* must have the same extent. The remaining axes of *x* and *y* (the instance axes) must match in order of declaration and extents.

When you call one of the single-instance (**gbl_gen_**) routines, *y* must be a CM array of the same data type, precision, rank, axis extents, and layout as *x*.

u

CM array of the same data type as *x* and *y*, rank one less than that of *x* and *y*, and the same shape and layout as *z*. The axes of *u* must match the instance axes of *x* and *y* in order of declaration and extents. Thus, each pair of vectors *x* and *y* in *x* and *y*, respectively, corresponds to a single value *u* in *u*.

α

Front-end scalar variable of the same data type as *x* and *y*.

β

Front-end scalar variable of the same data type as *x* and *y*.

x_vector_axis

Scalar integer variable. Identifies the axis of *x* along which the vectors lie.

y_vector_axis

Scalar integer variable. Identifies the axis of *y* along which the vectors lie.

ier

Scalar integer variable. Return code. Upon return from one of the multiple-instance (**gen_**) routines, contains one of the following values:

0 Successful return.

-1 *z*, *x*, and *y* (and *u*, in **_addto** calls) are not all valid CM arrays.

-2 *x* and *y* do not have the same rank.

-3 Axis *x_vector_axis* of *x* and axis *y_ vector_axis* of *y* do not have the same extent.

-4 The instance axes of *x* and *y* do not match in order of declaration and extents.

-8 *z* and *u* do not have the same shape and layout.

-10 *z*, *x*, and *y* (and *u*, in **_addto** calls) are not all of the same data type and precision.

-11 The data type is not real or complex (single or double precision).

-12 You called **gen_inner_product_c1**, **gen_inner_product_c1_noadd**, or **gen_inner_product_c1_addto**, but supplied data of a type other than complex.

-13 *x_vector_axis* or *y_vector_axis* is a bad axis number (it must be at least 1 and at most equal to the rank of the corresponding array).

-22 *z* does not have rank one less than that of *x*.

-24 The axes of *z* do not match the instance axes of *x* in order of declaration and extents.

Upon return from one of the single-instance (**gbl_gen_**) routines, contains one of the following values:

-1 *x* and *y* are not both valid CM arrays.

-2 *x* and *y* do not have the same rank.

-5 *x* and *y* do not have the same shape and layout.

-30 *x* and *y* do not have the same data type and precision.

-31 The data type is not real or complex (single or double precision).

-32 You called **gbl_gen_inner_product_c1**, **gbl_gen_inner_product_c1_noadd**, or **gbl_gen_inner_product_c1_addto**, but supplied data of a type other than complex.

DESCRIPTION

Multiple-Instance Routines. The multiple-instance inner product routines perform the operations listed below. The inner product either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array (with the results placed in the destination CM array).

Routine	Operation	Data Types
gen_inner_product	$z = z + x^{T}y$	real or complex
gen_inner_product_noadd	$z = x^{T}y$	real or complex
gen_inner_product_addto	$z = u + x^{T}y$	real or complex
gen_inner_product_c1	$z = z + x^{H}y$	complex only
gen_inner_product_c1_noadd	$z = x^{H}y$	complex only
gen_inner_product_c1_addto	$z = u + x^{H}y$	complex only

These routines require the source CM arrays to be at least two-dimensional, with at least one non-serial instance axis. (The reason for this restriction is that the destination array must have rank one less than that of the source CM arrays, but must also be a CM array — and therefore not completely serial.) Thus, to compute the inner product of a *single* pair of vectors, you must either declare any instance axes to have extent 1, or use the single-instance inner product routines.

Upon successful completion of **gen_inner_product** or **gen_inner_product_c1**, the inner product of each vector pair x and y in x and y, respectively, is added to the corresponding value in z.

Upon successful completion of **gen_inner_product_noadd** or **gen_inner_product_c1_noadd**, the inner product of each vector pair x and y in x and y, respectively, overwrites the corresponding value in z.

Upon successful completion of **gen_inner_product_addto** or **gen_inner_product_c1_addto**, the inner product of each vector pair x and y in x and y, respectively (added to the corresponding value in u) overwrites the corresponding value in z.

Single-Instance Routines. The single-instance inner product routines perform the operations listed below. In these formulas, the inner product occurs over *all* axes of the arrays x and y. The inner product either overwrites the destination front-end scalar variable, is added to the destination front-end scalar variable, or is added to a second front-end scalar variable (with the results placed in the destination front-end scalar variable).

Routine	Operation	Data Types
gbl_gen_inner_product	$\alpha = \alpha + x^T y$	real or complex
gbl_gen_inner_product_noadd	$\alpha = x^T y$	real or complex
gbl_gen_inner_product_addto	$\alpha = \beta + x^T y$	real or complex
gbl_gen_inner_product_c1	$\alpha = \alpha + x^H y$	complex only
gbl_gen_inner_product_c1_noadd	$\alpha = x^H y$	complex only
gbl_gen_inner_product_c1_addto	$\alpha = \beta + x^H y$	complex only

Upon successful completion of **gbl_gen_inner_product** or **gbl_gen_inner_product_c1**, the global inner product of x and y is added to α.

Upon successful completion of **gbl_gen_inner_product_noadd** or **gbl_gen_inner_product_c1_noadd**, the global inner product of x and y overwrites α.

Upon successful completion of **gbl_gen_inner_product_addto** or **gbl_gen_inner_product_c1_addto**, the global inner product of x and y (added to β) overwrites α.

NOTES

Overlapping Variables. The arrays x and y may be the same variable; the arrays z and u may be the same variable.

Numerical Complexity: Multiple-Instance Routines. If the vectors contained in x and y have length q, then for I instances, the number of floating-point operations for real operands is

- $2qI$ for **gen_inner_product, gen_inner_product_c1, gen_inner_product_addto,** and **gen_inner_product_c1_addto**
- $(2q-1)I$ for **gen_inner_product_noadd** and **gen_inner_product_c1_noadd**

while the number of floating-point operations for complex operands is

- $8qI$ for **gen_inner_product, gen_inner_product_c1, gen_inner_product_addto,** and **gen_inner_product_c1_addto**
- $(8q-2)I$ for **gen_inner_product_noadd** and **gen_inner_product_c1_noadd**

Numerical Complexity: Single-Instance Routines. If the product of the axis extents in each array (x and y) is Q, then the number of floating-point operations for real operands is

- $2Q$ for **gbl_gen_inner_product, gbl_gen_inner_product_c1, gbl_gen_inner_product_addto,** and **gbl_gen_inner_product_c1_addto**
- $2Q-1$ for **gbl_gen_inner_product_noadd** and **gbl_gen_inner_product_c1_noadd**

while the number of floating-point operations for complex operands is

- $8Q$ for **gbl_gen_inner_product, gbl_gen_inner_product_c1, gbl_gen_inner_product_addto,** and **gbl_gen_inner_product_c1_addto**
- $8Q-2$ for **gbl_gen_inner_product_noadd** and **gbl_gen_inner_product_c1_noadd**

EXAMPLES

Sample CM Fortran code that uses the inner product routines can be found on-line in the subdirectory `inner-product/cmf/` of a CMSSL examples directory whose location is site-specific.

3.2 2-Norm

The *multiple-instance* 2-norm routine, **gen_2_norm**, computes one or more instances of the 2-norm of a vector. Given a CM array x containing multiple instances of a vector x, **gen_2_norm** performs the following operation for each instance:

Data Type	Operation
real	$z = (x^T x)^{1/2} = \|x\|_2$
complex	$z = (x^H x)^{1/2} = \|x\|_2$

The *single-instance* 2-norm routine, **gbl_gen_2_norm**, computes the global 2-norm of a CM array as defined below. In these formulas, the norm is computed over all axes of the array x.

Data Type	Operation
real	$\alpha = (x^T x)^{1/2} = \|x\|_2$
complex	$\alpha = (x^H x)^{1/2} = \|x\|_2$

The norm is always a real value. Details are provided in the man page that follows.

2-Norm

The multiple-instance 2-norm routine, **gen_2_norm**, computes one or more instances of the 2-norm of a vector. The single-instance 2-norm routine, **gbl_gen_2_norm**, computes the global 2-norm of a CM array.

SYNTAX

gen_2_norm (*z, x, x_vector_axis, ier*)

gbl_gen_2_norm (*α, x, ier*)

ARGUMENTS

z	Real CM array of the same precision as *x* and rank one less than that of *x*. The axes of *z* must match the instance axes of *x* in order of declaration and extents. Thus, each vector *x* in *x* corresponds to a single value *z* in *z*.
α	Real front-end scalar variable.
x	When you call **gen_2_norm**, *x* must be a real or complex CM array of rank ≥ 2, with at least one non-serial instance axis. It contains one or more instances of the vector *x* whose 2-norm you want to compute. (For a single-instance problem, declare any instance axes to have extent 1.) When you call **gbl_gen_2_norm**, *x* must be a real or complex CM array of rank ≥ 1.
x_vector_axis	Scalar integer variable. Identifies the axis of *x* along which the vectors lie.
ier	Scalar integer variable. Return code. Upon return from **gen_2_norm**, contains one of the following values: 0 Successful return. -1 *z* and *x* are not valid CM arrays. -2 *x* does not have a rank of at least 2.

-13 *x_vector_axis* is a bad axis number (it must be at least 1 and at most equal to the rank of *x*).

-22 *z* does not have rank one less than that of *x*.

-24 The axes of *z* do not match the instance axes of *x* in order of declaration and extents.

-40 *z* and *x* do not have the same precision.

-41 *z* has a data type other than real.

-42 *x* has a data type other than real or complex.

Upon return from **gbl_gen_2_norm**, contains one of the following values:

0 Successful return.

-1 *x* is not a valid CM array.

-31 The data type is not real or complex (single or double precision).

DESCRIPTION

Multiple-Instance Routine. For each instance, **gen_2_norm** performs the following operation:

Data Type	Operation
real	$z = (x^T x)^{1/2} = \|\|x\|\|_2$
complex	$z = (x^H x)^{1/2} = \|\|x\|\|_2$

The **gen_2_norm** routine requires the source CM array to be at least two-dimensional, with at least one non-serial instance axis. (The reason for this restriction is that the destination array must have rank one less than that of the source CM array, but must also be a CM array — and therefore not completely serial.) Thus, to compute the 2-norm of a *single* vector, you must either declare any instance axes to have extent 1, or use the single-instance 2-norm routine.

Upon successful completion of **gen_2_norm**, the 2-norm of each vector in *x* overwrites the corresponding value in *z*.

Single-Instance Routine. The **gbl_gen_2_norm** routine performs the operations listed below. In these formulas, the norm is computed over all axes of the array x.

Data Type	Operation
real	$\alpha = (x^T x)^{1/2} = \|x\|_2$
complex	$\alpha = (x^H x)^{1/2} = \|x\|_2$

Upon successful completion of **gbl_gen_2_norm**, the global 2-norm of x overwrites α.

NOTES

Numerical Complexity: Multiple-Instance Routine. If the vectors contained in x have length q, then for I instances, the number of floating-point operations used by **gen_2_norm** is $[(2q-1)+8]I$ for real operands or $[(4q-1)+8]I$ for complex operands. (8 is the flop count for the square root operation.)

Numerical Complexity: Single-Instance Routine. If the product of the axis extents of x is Q, then the number of floating-point operations used by **gbl_gen_2_norm** is $(2Q-1)+8$ for real operands or $(4Q-1)+8$ for complex operands. (8 is the flop count for the square root operation.)

EXAMPLES

Sample CM Fortran code that uses the 2-norm routines can be found on-line in the subdirectory

```
inner-product/cmf/
```

of a CMSSL examples directory whose location is site-specific.

3.3 Outer Product

The outer product routines compute one or more instances of an outer product of two vectors. The result either overwrites the destination CM array, is added to the destination CM array, or is added to a second CM array.

Given CM arrays x, y, A, and B containing multiple instances of the vectors x and y and the matrices A and B, respectively, the outer product routines perform the operations listed below for each instance. In these descriptions, y^T and y^H denote y transpose and y Hermitian, respectively.

Routine	Operation	Data Types
gen_outer_product	$A = A + xy^T$	real or complex
gen_outer_product_noadd	$A = xy^T$	real or complex
gen_outer_product_addto	$A = B + xy^T$	real or complex
gen_outer_product_c2	$A = A + xy^H$	complex only
gen_outer_product_c2_noadd	$A = xy^H$	complex only
gen_outer_product_c2_addto	$A = B + xy^H$	complex only

The man page following this section provides details.

Outer Product

The routines described below compute one or multiple instances of an outer product of two vectors.

SYNTAX

gen_outer_product	(*A, x, y, row_axis, col_axis, x_vector_axis, y_vector_axis, ier*)
gen_outer_product_noadd	(*A, x, y, row_axis, col_axis, x_vector_axis, y_vector_axis, ier*)
gen_outer_product_addto	(*A, x, y, B, row_axis, col_axis, x_vector_axis, y_vector_axis, ier*)
gen_outer_product_c2	(*A, x, y, row_axis, col_axis, x_vector_axis, y_vector_axis, ier*)
gen_outer_product_c2_noadd	(*A, x, y, row_axis, col_axis, x_vector_axis, y_vector_axis, ier*)
gen_outer_product_c2_addto	(*A, x, y, B, row_axis, col_axis, x_vector_axis, y_vector_axis, ier*)

ARGUMENTS

A	CM array of type real or complex and rank greater than or equal to 2. Contains one or more instances of the destination matrix, *A*, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). Upon completion, each matrix instance is overwritten by the result of the outer product call.
x	CM array of the same type and precision as *A* and rank one less than that of *A*. Contains one or more instances of the first source vector, *x*, embedded along axis *x_vector_axis*. Axis *x_vector_axis* of *x* must have the same extent as axis *row_axis* of *A*. The remaining axes of *x* must match the instance axes of *A* in length and order of declaration. Thus, each vector in *x* corresponds to a matrix in *A*.

y — CM array of the same type and precision as *A* and rank one less than that of *A*. Contains one or more instances of the second source vector, *y*, embedded along axis *y_vector_axis*. Axis *y_vector_axis* of *y* must have the same extent as axis *col_axis* of *A*. The remaining axes of *y* must match the instance axes of *A* in length and order of declaration. Thus, each vector in *y* corresponds to a matrix in *A*.

B — CM array of the same type, precision, rank, shape, and layout as *A*. Contains one or more embedded matrices *B* defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). The remaining axes must match the instance axes of *A* in length and order of declaration. Thus, each matrix in *B* corresponds to a matrix in *A*. This argument is used only in the **gen_outer_product_addto** and **gen_outer_product_c2_addto** calls. These calls add each outer product to the corresponding matrix within *B* and place the result in the corresponding matrix within *A*. The contents of *B* are not changed by the operation (unless *B* and *A* are the same variable).

row_axis — Scalar integer between 1 and the rank of *A*. The axis of *A* and *B* that counts the rows of the embedded matrix or matrices.

col_axis — Scalar integer between 1 and the rank of *A*. The axis of *A* and *B* that counts the columns of the embedded matrix or matrices.

x_vector_axis — Scalar integer between 1 and the rank of *x*. The axis of *x* along which the elements of each embedded vector lie.

y_vector_axis — Scalar integer between 1 and the rank of *y*. The axis of *y* along which the elements of each embedded vector lie.

ier — Scalar integer variable. On return, contains one of the following error codes (if the CMSSL safety mechanism is turned on):

- 0 Successful return.
- -1 The rank of *A* < 2, or the rank of *x* or *y* is not equal to (rank of *A*) - 1.
- -2 The extent of *x* along axis *x_vector_axis* is not equal to the number of rows in the matrices in *A*; or the extent of *y* along axis *y_vector_axis* is not equal to the number of columns in the matrices in *A*.

Version 3.1, June 1993

-4 *A*, *x*, *y*, and *B* are not all the same data type (real or complex), or you supplied non-complex data when calling one of the conjugate (**_c2**) routines.

-8 The geometry of *B* differs from the geometry of *A*, or the instance axes of *x* and *y* do not match those of *A* in length and order of declaration.

-16 One or more of *row_axis*, *col_axis*, *x_vector_axis*, and *y_vector_axis* are less than 1 or greater than the rank of the associated CM array.

-32 *A*, *x*, *y*, or *B* is not a CM array.

DESCRIPTION

For each instance, the outer product routines perform the operations listed below. In these descriptions, y^T and y^H denote *y* transpose and *y* Hermitian, respectively.

Routine	Operation	Data Types
gen_outer_product	$A = A + xy^T$	real or complex
gen_outer_product_noadd	$A = xy^T$	real or complex
gen_outer_product_addto	$A = B + xy^T$	real or complex
gen_outer_product_c2	$A = A + xy^H$	complex only
gen_outer_product_c2_noadd	$A = xy^H$	complex only
gen_outer_product_c2_addto	$A = B + xy^H$	complex only

In elementwise notation, for each instance **gen_outer_product** computes

$$A(i,j) = A(i,j) + x(i) * y(j)$$

and **gen_outer_product_c2** computes

$$A(i,j) = A(i,j) + x(i) * \overline{y(j)}$$

where $\overline{y(j)}$ denotes the conjugate of $y(j)$.

NOTES

Distinct Variables. *A* must be a variable distinct from the source arrays, *x* and *y*. The source arrays can be the same variable, and *B* and *A* can be the same variable.

Numerical Stability. The algorithm for the outer product is numerically stable.

Numerical Complexity. If the matrices embedded in *A* and *B* have axis extents ($p \times q$), axis *x_vector_axis* of *x* has extent *p*, and axis *y_vector_axis* of *y* has extent *q*, then for *I* instances, the number of floating-point operations for real operands is

- $2pqI$ for **gen_outer_product** and **gen_outer_product_addto**
- pqI for **gen_outer_product_noadd**

while the number of floating-point operations for complex operands is

- $8pqI$ for **gen_outer_product** and **gen_outer_product_addto**
- $6pqI$ for **gen_outer_product_noadd**

Each conjugate routine performs the same number of floating-point operations as the corresponding non-conjugate routine.

EXAMPLES

Sample CM Fortran program that uses the outer product routines can be found on-line in the subdirectory

```
outer-product/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.1, June 1993

3.4 Matrix Vector Multiplication

The matrix vector multiplication routines compute one or more matrix vector products. Given CM arrays *y*, *x*, *v*, and *A* containing multiple instances of the vectors *y*, *x*, and *v* and the matrix *A*, respectively, the matrix vector multiplication routines perform the operations listed below for each instance. In these descriptions, $\overline{A}$ denotes the conjugate of *A*.

Routine	Operation	Data Types
gen_matrix_vector_mult	$y = y + Ax$	real or complex
gen_matrix_vector_mult_noadd	$y = Ax$	real or complex
gen_matrix_vector_mult_addto	$y = v + Ax$	real or complex
gen_matrix_vector_mult_c1	$y = y + \overline{A}x$	complex only
gen_matrix_vector_mult_c1_noadd	$y = \overline{A}x$	complex only
gen_matrix_vector_mult_c1_addto	$y = v + \overline{A}x$	complex only

The man page following this section provides details.

Matrix Vector Multiplication

The matrix vector multiplication routines compute one or more instances of a matrix vector product.

SYNTAX

gen_matrix_vector_mult	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_matrix_vector_mult_noadd	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_matrix_vector_mult_addto	(*y, A, x, v, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_matrix_vector_mult_c1	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_matrix_vector_mult_c1_noadd	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_matrix_vector_mult_c1_addto	(*y, A, x, v, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)

ARGUMENTS

y CM array of rank greater than or equal to 1 and type real or complex. Contains one or more instances of the destination vector *y*, embedded along axis *y_vector_axis*. Axis *y_vector_axis* of *y* must have the same extent as axis *row_axis* of *A*. Upon completion, each vector instance is overwritten by the result of the matrix vector multplication call.

A CM array of the same type and precision as *y* and rank one greater than that of *y*. Contains one or more instances of the matrix *A*, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). The remaining axes must match the instance axes of *y* in length and order of declaration. Thus, each matrix in *A* corresponds to a vector in *y*. The contents of *A* are not changed during execution.

x — CM array of the same rank, type, and precision as *y*. Contains one or more instances of *x*, the vector that is to be multiplied by the matrix *A*, embedded along axis *x_vector_axis*. Axis *x_vector_axis* of *x* must have the same extent as axis *col_axis* of *A*. The remaining axes of *x* must match the instance axes of *y* in length and order of declaration. Thus, each vector in *x* corresponds to a vector in *y*. The contents of *x* are not changed during execution.

v — CM array of the same rank, type, precision, shape, and layout as *y*. This argument is used only in the **gen_matrix_vector_mult_addto** and **gen_matrix_vector_mult_c1_addto** calls. It contains one or more instances of the vector *v* that is to be added to the matrix vector product, embedded along axis *y_vector_axis*. The contents of *v* are not changed during execution, unless *v* is the same variable as *y*.

y_vector_axis — Scalar integer between 1 and the rank of *y*. The axis of *y* and *v* along which the elements of the embedded vectors lie.

row_axis — Scalar integer between 1 and the rank of *A*. The axis of *A* that counts the rows of the embedded matrix or matrices.

col_axis — Scalar integer between 1 and the rank of *A*. The axis of *A* that counts the columns of the embedded matrix or matrices.

x_vector_axis — Scalar integer between 1 and the rank of *x*. The axis of *x* along which the elements of the embedded vectors lie.

ier — Scalar integer variable. Upon return, contains one of the following error codes (if the CMSSL safety mechanism is turned on):

0	Normal return.
-1	Rank(*x*) ≠ rank(*y*) ≠ rank(*A*) -1.
-2	Axis *row_axis* of *A* and axis *y_vector_axis* of *y* do not have the same extent, or axis *col_axis* of *A* and axis *x_vector_axis* of *x* do not have the same extent.
-4	Matrix or vectors are not of the same data type (real or complex); or you supplied non-complex data when calling one of the conjugate (**_c1**) routines.
-8	Instance axes of the input CM arrays do not match in length and order of declaration; or *y* and *v* do not have the same

rank, shape, and layout.

-32 *A*, *x*, *y*, or *v* is not a CM array.

DESCRIPTION

For each instance, the matrix vector multiplication routines perform the operations listed below. In these descriptions, $\overline{A}$ denotes the conjugate of A.

Routine	Operation	Data Types
gen_matrix_vector_mult	$y = y + Ax$	real or complex
gen_matrix_vector_mult_noadd	$y = Ax$	real or complex
gen_matrix_vector_mult_addto	$y = v + Ax$	real or complex
gen_matrix_vector_mult_c1	$y = y + \overline{A}x$	complex only
gen_matrix_vector_mult_c1_noadd	$y = \overline{A}x$	complex only
gen_matrix_vector_mult_c1_addto	$y = v + \overline{A}x$	complex only

NOTES

Distinct Variables. The arrays *y*, *A*, and *x* must be distinct variables. However, *v* and *y* can be the same variable.

Numerical Stability. The algorithm is numerically stable.

Numerical Complexity. If the matrices embedded in *A* have axis extents (p x q), axis *x_vector_axis* of *x* has extent q, and axis *y_vector_axis* of *y* has extent p, then for I instances, the number of floating-point operations performed is shown below.

	Real Operands	Complex Operands
gen_matrix_vector_mult	$2pqI$	$8pqI$
gen_matrix_vector_mult_addto	$2pqI$	$8pqI$
gen_matrix_vector_mult_noadd	$(2pq - p)\ I$	$(8pq - 2p)\ I$

Each conjugate routine performs the same number of floating-point operations as the corresponding non-conjugate routine.

EXAMPLES

Sample CM Fortran code that uses the matrix vector multiplication routines can be found on-line in the subdirectory

```
matrix-vector/cmf/
```

of a CMSSL examples directory whose location is site-specific.

3.5 Vector Matrix Multiplication

The vector matrix multiplication routines compute one or more vector matrix products. Given CM arrays *y*, *x*, *v*, and *A* containing multiple instances of the vectors *y*, *x*, and *v* and the matrix *A*, respectively, the vector matrix multiplication routines perform the operations listed below for each instance. In these descriptions, $\overline{A}$ denotes the conjugate of *A*.

Routine	Operation	Data Types
gen_vector_matrix_mult	$y^T = y^T + x^T A$	real or complex
gen_vector_matrix_mult_noadd	$y^T = x^T A$	real or complex
gen_vector_matrix_mult_addto	$y^T = v^T + x^T A$	real or complex
gen_vector_matrix_mult_c2	$y^T = y^T + x^T \overline{A}$	complex only
gen_vector_matrix_mult_c2_noadd	$y^T = x^T \overline{A}$	complex only
gen_vector_matrix_mult_c2_addto	$y^T = v^T + x^T \overline{A}$	complex only

The man page following this section provides details.

Vector Matrix Multiplication

The vector matrix multiplication routines compute one or more instances of a vector matrix product.

SYNTAX

gen_vector_matrix_mult	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_vector_matrix_mult_noadd	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_vector_matrix_mult_addto	(*y, A, x, v, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_vector_matrix_mult_c2	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_vector_matrix_mult_c2_noadd	(*y, A, x, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)
gen_vector_matrix_mult_c2_addto	(*y, A, x, v, y_vector_axis, row_axis, col_axis, x_vector_axis, ier*)

ARGUMENTS

y — CM array of rank greater than or equal to 1 and type real or complex. Contains one or more instances of the destination vector *y*, embedded along axis *y_vector_axis*. Axis *y_vector_axis* of *y* must have the same extent as axis *col_axis* of *A*. Upon successful completion, each vector instance is overwritten by the result of the vector matrix multplication call.

A — CM array of the same type and precision as *y* and rank one greater than that of *y*. Contains one or more instances of the matrix *A*, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). The remaining axes must match the instance axes of *y* in length and order of declaration. Thus, each matrix in *A* corresponds to a vector in *y*. The contents of *A* are not changed during execution.

x — CM array of the same rank, type, and precision as *y*. Contains one or more instances of the vector *x* that is to be multiplied by the matrix *A*, embedded along axis *x_vector_axis*. Axis *x_vector_axis* of *x* must have the same extent as axis *row_axis* of *A*. The remaining axes of *x* must match the instance axes of *y* in length and order of declaration. Thus, each vector in *x* corresponds to a vector in *y*. The contents of *x* are not changed during execution.

v — CM array of the same rank, type, precision, shape, and layout as *y*. This argument is used only in the **gen_vector_matrix_mult_addto** and **gen_vector_matrix_mult_c2_addto** calls. It contains one or more instances of the vector *v* that is to be added to the vector matrix product, embedded along axis *y_vector_axis*. The contents of *v* are not changed during execution, unless *v* is the same variable as *y*.

y_vector_axis — Scalar integer between 1 and the rank of *y*. The axis of *y* and *v* along which the elements of the embedded vectors lie.

row_axis — Scalar integer between 1 and the rank of *A*. The axis of *A* that counts the rows of the embedded matrix or matrices.

col_axis — Scalar integer between 1 and the rank of *A*. The axis of *A* that counts the columns of the embedded matrix or matrices.

x_vector_axis — Scalar integer between 1 and the rank of *x*. The axis of *x* along which the elements of each vector lie.

ier — Scalar integer variable. Upon return, contains one of the following error codes (if the CMSSL safety mechanism is turned on):

0	Normal return.
-1	Rank(*x*) ≠ rank(*y*) ≠ rank(*A*) -1.
-2	Axis *col_axis* of *A* and axis *y_vector_axis* of *y* do not have the same extent, or axis *row_axis* of *A* and axis *x_vector_axis* of *x* do not have the same extent.
-4	Matrix or vectors are not of the same data type (real or complex); or you supplied non-complex data when calling one of the conjugate (**_c2**) routines.
-8	Instance axes of the input CM arrays do not match in length and order of declaration; or *y* and *v* do not have the same

rank, shape, and layout.
-32 *A*, *x*, *y*, or *v* is not a CM array.

DESCRIPTION

For each instance, the vector matrix multiplication routines perform the operations listed below. In these descriptions, $\overline{A}$ denotes the conjugate of A.

Routine	Operation	Data Types
gen_vector_matrix_mult	$y^T = y^T + x^T A$	real or complex
gen_vector_matrix_mult_noadd	$y^T = x^T A$	real or complex
gen_vector_matrix_mult_addto	$y^T = v^T + x^T A$	real or complex
gen_vector_matrix_mult_c2	$y^T = y^T + x^T \overline{A}$	complex only
gen_vector_matrix_mult_c2_noadd	$y^T = x^T \overline{A}$	complex only
gen_vector_matrix_mult_c2_addto	$y^T = v^T + x^T \overline{A}$	complex only

NOTES

Distinct Variables. The arrays *y*, *A*, and *x* must be distinct variables. However, *v* and *y* can be the same variable.

Numerical Stability. The algorithm is numerically stable.

Numerical Complexity. If the matrices embedded in *A* have axis extents ($p \times q$), axis *x_vector_axis* of *x* has extent q, and axis *y_vector_axis* of *y* has extent p, then for I instances, the number of floating-point operations performed is shown below.

	Real Operands	Complex Operands
gen_vector_matrix_mult	$2pqI$	$8pqI$
gen_vector_matrix_mult_addto	$2pqI$	$8pqI$
gen_vector_matrix_mult_noadd	$(2pq - q)\ I$	$(8pq - 2q)\ I$

Each conjugate routine performs the same number of floating-point operations as the corresponding non-conjugate routine.

EXAMPLES

Sample CM Fortran code that uses the vector matrix multiplication routines can be found on-line in the subdirectory

```
vector-matrix/cmf/
```

of a CMSSL examples directory whose location is site-specific.

3.6 Infinity Norm

Given a CM array A containing one or more matrices A, the **gen_infinity_norm** routine computes the infinity norm of each matrix A. Details are provided in the man page that follows.

The infinity norm of a matrix A^{-1} can be estimated based on the QR or LU factors of A using the method developed by Hager (see reference 10 listed in Section 3.9). The **gen_lu_infinity_norm_inv** and **gen_qr_infinity_norm_inv** routines, described in Chapter 5, perform these estimations.

Infinity Norm

Given one or more matrices embedded in a CM array, the **gen_infinity_norm** routine computes the infinity norms of the matrices.

SYNTAX

gen_infinity_norm (*a, A, n1, n2, row_axis, col_axis, ier*)

ARGUMENTS

a

Real CM array with the same rank and precision as *A*. Axes *row_axis* and *col_axis* must have extent 1.

Upon successful completion of **gen_infinity_norm**, the infinity norm of each matrix *A* within *A* is placed in the corresponding position of *a*. For example, if *A* has dimensions 16 x 16 x 4 x 128, with *row_axis* = 2 and *col_axis* = 3, then upon completion, *a*(*r*, 1, 1, *s*) contains the infinity norm of the matrix consisting of *A*(*r*, :, :, *s*).

A

Real or complex CM array of rank ≥ 2.

When you call **gen_infinity_norm**, *A* must contain one or more embedded matrices *A* whose infinity norms you want to compute. Each matrix *A* is assumed to be dense with dimensions *n1* x *n2*. The axis identified by *row_axis* must count the rows of the embedded matrices; the axis identified by *col_axis* must count the columns of the matrices. Axes *row_axis* and *col_axis* may have extents greater than *n1* and *n2*, respectively; that is, each instance of *A* may be contained in the upper left-hand *n1* x *n2* elements of a larger matrix within *A*.

n1

Scalar integer variable. The number of rows in each matrix embedded in *A*.

n2

Scalar integer variable. The number of columns in each matrix embedded in *A*.

Version 3.1, June 1993

row_axis Scalar integer variable. The axis that counts the rows of the matrices *A* embedded in *A*.

col_axis Scalar integer variable. The axis that counts the columns of the matrices *A* embedded in *A*.

ier Scalar integer variable. Return code; set to 0 upon successful return, or to one of the following error codes:

-1	*n1* is invalid.
-2	*n2* is invalid.
-8	The rank of *a* is not equal to the rank of *A*.
-32	*A* is not real or complex, *a* is not real, or *A* and *a* do not have the same precision.
-64	*row_axis* or *col_axis* is invalid.

DESCRIPTION

Given one or more matrices *A* embedded in a CM array *A*, the **gen_infinity_norm** routine computes the infinity norm of each *A*.

The infinity norm of a matrix *A*, denoted here by $\| A \|_\infty$, is defined by

$$\| A \|_\infty = \max_{\| x \|_\infty = 1} \| Ax \|_\infty$$

where the infinity norm of a vector, $\| x \|_\infty$, is defined as the maximum of the absolute values of the vector components:

$$\| x \|_\infty = \max_i | x_i |$$

The infinity-norm condition number of a matrix *A* is equal to the product of $\| A \|_\infty$ and $\| A^{-1} \|_\infty$.

EXAMPLES

Sample CM Fortran code that uses the **gen_infinity_norm** routine can be found on-line in the subdirectory

```
infinity-norm/cmf/
```

of a CMSSL examples directory whose location is site-specific.

3.7 Matrix Multiplication

The matrix multiplication routines compute one or more matrix products. Given CM arrays A, B, C, and D containing multiple instances of the matrices A, B, C, and D, respectively, the matrix multiplication routines perform the operations listed below for each instance. In these descriptions, A^T and A^H denote A transpose and A Hermitian, respectively.

Routine	Operation	Data Types
gen_matrix_mult	$C = C + AB$	real or complex
gen_matrix_mult_noadd	$C = AB$	real or complex
gen_matrix_mult_addto	$C = D + AB$	real or complex
gen_matrix_mult_t1	$C = C + A^TB$	real or complex
gen_matrix_mult_t1_noadd	$C = A^TB$	real or complex
gen_matrix_mult_t1_addto	$C = D + A^TB$	real or complex
gen_matrix_mult_h1	$C = C + A^HB$	complex only
gen_matrix_mult_h1_noadd	$C = A^HB$	complex only
gen_matrix_mult_h1_addto	$C = D + A^HB$	complex only
gen_matrix_mult_t2	$C = C + AB^T$	real or complex
gen_matrix_mult_t2_noadd	$C = AB^T$	real or complex
gen_matrix_mult_t2_addto	$C = D + AB^T$	real or complex
gen_matrix_mult_h2	$C = C + AB^H$	complex only
gen_matrix_mult_h2_noadd	$C = AB^H$	complex only
gen_matrix_mult_h2_addto	$C = D + AB^H$	complex only
gen_matrix_mult_t1_t2	$C = C + A^TB^T$	real or complex
gen_matrix_mult_t1_t2_noadd	$C = A^TB^T$	real or complex
gen_matrix_mult_t1_t2_addto	$C = D + A^TB^T$	real or complex

The algorithm used depends on the axis extents of the arrays supplied. The man page following this section provides details about this routine.

Matrix Multiplication

The matrix multiplication routines compute one or more matrix products.

SYNTAX

gen_matrix_mult	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_noadd	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_addto	(*C, A, B, D, row_axis, col_axis, ier*)
gen_matrix_mult_t1	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_t1_noadd	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_t1_addto	(*C, A, B, D, row_axis, col_axis, ier*)
gen_matrix_mult_h1	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_h1_noadd	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_h1_addto	(*C, A, B, D, row_axis, col_axis, ier*)
gen_matrix_mult_t2	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_t2_noadd	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_t2_addto	(*C, A, B, D, row_axis, col_axis, ier*)
gen_matrix_mult_h2	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_h2_noadd	(*C, A, B, D, row_axis, col_axis, ier*)
gen_matrix_mult_h2_addto	(*C, A, B, D, row_axis, col_axis, ier*)
gen_matrix_mult_t1_t2	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_t1_t2_noadd	(*C, A, B, row_axis, col_axis, ier*)
gen_matrix_mult_t1_t2_addto	(*C, A, B, D, row_axis, col_axis, ier*)

ARGUMENTS

C CM array of type real or complex and rank greater than or equal to 2. Contains one or more instances of the destination matrix *C*, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). Axis *row_axis* of ***C*** must have the same extent as axis *row_axis* of ***A***. Axis *col_axis* of ***C*** must have the same extent as axis *col_axis* of ***B***.

Upon successful completion, each matrix instance within ***C*** is overwritten by the result of the matrix multiplication call.

A CM array of the same rank, type, and precision as ***C***. Contains one or more instances of the left-hand factor matrix *A*, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). Axis *col_axis* of ***A*** must have the same extent as axis *row_axis* of ***B***. The contents of ***A*** are not changed during execution.

B CM array of the same rank, type, and precision as ***C***. Contains one or more instances of the right-hand factor matrix *B*, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). The contents of ***B*** are not changed during execution.

D CM array of the same rank, type, precision, shape, and layout as ***C***. This argument is used only in the calls whose names end in "**_addto**." It contains one or more instances of the matrix *D* that is to be added to the matrix product, defined by axes *row_axis* (which counts the rows) and *col_axis* (which counts the columns). The contents of ***D*** are not changed during execution, unless ***D*** and ***C*** are the same variable.

row_axis Scalar integer between 1 and the rank of ***C***. The axis of ***C***, ***A***, ***B***, and ***D*** that counts the rows of the embedded matrix or matrices.

col_axis Scalar integer between 2 and the rank of ***C***. The axis of ***C***, ***A***, ***B***, and ***D*** that counts the columns of the embedded matrix or matrices.

ier Scalar integer variable. Upon return, contains one of the following error codes (if the CMSSL safety mechanism is turned on):

0	Normal return.
-1	Ranks of provided arrays are different or are not

at least 2.

-2 Non-conforming *A* and *B* axis extents.

-4 Non-conforming *A* and *C* axis extents.

-8 Non-conforming *B* and *C* axis extents.

-16 The instance axes of *A*, *B*, and *C* do not match in length and order of declaration; or *C* and *D* do not have the same rank, shape, and layout.

-32 *row_axis* and/or *col_axis* is less than 1 or greater than the rank of the arrays.

-64 *C*, *A*, *B*, or *D* is not a CM array.

-128 *C*, *A*, and *B* (and *D*, in **_addto** calls) do not all have the same data type (real or complex), or you supplied non-complex data when calling one of the conjugate (**_h1** or **_h2**) routines.

DESCRIPTION

For each instance, the matrix multiplication routines perform the operations listed below. In these descriptions, A^T and A^H denote A transpose and A Hermitian, respectively.

Routine	Operation	Data Types
gen_matrix_mult	$C = C + AB$	real or complex
gen_matrix_mult_noadd	$C = C$	real or complex
gen_matrix_mult_addto	$C = D + AB$	real or complex
gen_matrix_mult_t1	$C = C + A^TB$	real or complex
gen_matrix_mult_t1_noadd	$C = A^TB$	real or complex
gen_matrix_mult_t1_addto	$C = D + A^TB$	real or complex
gen_matrix_mult_h1	$C = C + A^HB$	complex only
gen_matrix_mult_h1_noadd	$C = A^HB$	complex only
gen_matrix_mult_h1_addto	$C = D + A^HB$	complex only
gen_matrix_mult_t2	$C = C + AB^T$	real or complex
gen_matrix_mult_t2_noadd	$C = AB^T$	real or complex

gen_matrix_mult_t2_addto	$C = D + AB^T$	real or complex
gen_matrix_mult_h2	$C = C + AB^H$	complex only
gen_matrix_mult_h2_noadd	$C = AB^H$	complex only
gen_matrix_mult_h2_addto	$C = D + AB^H$	complex only
gen_matrix_mult_t1_t2	$C = C + A^TB^T$	real or complex
gen_matrix_mult_t1_t2_noadd	$C = A^TB^T$	real or complex
gen_matrix_mult_t1_t2_addto	$C = D + A^TB^T$	real or complex

The algorithm used depends on the axis extents of the arrays supplied.

For calls that do not transpose the matrices, the arrays conform correctly with the following axis extents for *row_axis* and *col_axis*:

Array	axis_1 extent	axis_2 extent
A	*p*	*q*
B	*q*	*r*
C	*p*	*r*
D	*p*	*r*

For calls that transpose the matrix *A*, the arrays conform correctly with the following axis extents for *row_axis* and *col_axis*:

Array	axis_1 extent	axis_2 extent
A	*q*	*p*
B	*q*	*r*
C	*p*	*r*
D	*p*	*r*

For calls that transpose the matrix *B*, the arrays conform correctly with the following axis extents for *row_axis* and *col_axis*:

Array	axis_1 extent	axis_2 extent
A	*p*	*q*
B	*r*	*q*
C	*p*	*r*
D	*p*	*r*

For calls that transpose both *A* and *B*, the arrays conform correctly with the following axis extents for *row_axis* and *col_axis*:

Array	axis_1 extent	axis_2 extent
A	*q*	*p*
B	*r*	*q*
C	*p*	*r*
D	*p*	*r*

NOTES

Distinct Variables. All input arrays must be distinct, except that *C* and *D* can be the same variable.

Numerical Stability. The algorithm is numerically stable.

Numerical Complexity. If the matrices embedded in *A* have the axis extents listed in the Description section, then for *I* instances, the number of floating-point operations performed is shown below.

	Real Operands	Complex Operands
gen_matrix_mult	$2\,pqrI$	$8\,pqrI$
gen_matrix_mult_addto	$2\,pqrI$	$8\,pqrI$
gen_matrix_mult_noadd	$(2\,pqr - pr)\,I$	$(8\,pqr - 2pr)\,I$

Each conjugate routine performs the same number of floating-point operations as the corresponding non-conjugate routine.

EXAMPLES

Sample CM Fortran code that uses the matrix multiplication routines can be found on-line in the subdirectory

```
matrix-multiply/cmf/
```

of a CMSSL examples directory whose location is site-specific.

3.8 Matrix Multiplication with External Storage

The **gen_matrix_mult_ext** routine performs the operation

$$Y = Y + AX$$

where Y is a matrix of size $n1 \times m$, X is a matrix of size $n2 \times m$, and A is a matrix of size $n1 \times n2$ that is too large to fit into core memory. The man page that follows provides details.

Matrix Multiplication with External Storage

The **gen_matrix_mult_ext** routine performs the operation $Y = Y + AX$ where Y is a matrix of size $n1 \times m$, X is a matrix of size $n2 \times m$, and A is a matrix of size $n1 \times n2$ that is too large to fit into core memory.

SYNTAX

gen_matrix_mult_ext (*Y, X, m, n1, n2, blk, type, unit, ier*)

ARGUMENTS

Y — CM array of rank 2, the same data type as *A* (real or complex), and size *n1* x *m*. Upon return, contains *Y* + *AX*.

X — CM array of rank 2, the same data type as *A*, and size *n2* x *m*.

m — Scalar integer variable. The number of columns in *X* and *Y*.

n1 — Scalar integer variable. The number of rows in *A* and *Y*.

n2 — Scalar integer variable. The number of rows in *X* and columns in *A*.

blk — Scalar integer variable. Block size. The matrix *A* is partitioned into blocks of *blk* columns, or panels. See the Notes section, below, for guidelines for choosing *blk*.

type — Scalar integer variable. The data type. Specify one of the following values:

CMSSL_single_real	real*4
CMSSL_double_real	real*8
CMSSL_single_complex	complex*8
CMSSL_double_complex	complex*16

unit — Scalar integer. Valid unit number associated with the file that contains the matrix *A* stored in serial order (see the Notes below.) Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). The **gen_matrix_mult_ext** routine reads the matrix from *unit* and does not modify the file.

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to -1 if the routine encounters an I/O error on *unit*.

DESCRIPTION

The **gen_matrix_mult_ext** routine performs the operation $Y = Y + AX$ where Y is a matrix of size $n1 \times m$, X is a matrix of size $n2 \times m$, and A is a matrix of size $n1 \times n2$ that is too large to fit into core memory.

NOTES

Include the CMSSL Header File. Because the routine described above uses symbolic constants, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL symbolic constants.

File Unit. The I/O unit *unit* must be assigned to a file before you call **gen_matrix_mult_ext**. In CM Fortran, file assignment is done with the **CMF_FILE_OPEN** utility (or an equivalent utility for a device or socket). For information regarding parallel I/O in general, see the *CM-5 I/O System Programming Guide*. For information about the CM Fortran interface to parallel I/O, see the *CM Fortran Utility Library Reference Manual*. As described in this manual, there are essentially two modes of external storage: Fixed Machine Size (FMS) and Serial Order (SO). Serial order is the familiar Fortran row-major order and is the one used by the external matrix multiplication routine. Therefore, A must be stored in serial order in file unit *unit*. In this order, the data is portable across the CM-5 external storage systems (DataVault, Scalable Disk Array, HIPPI).

Choosing the Block Size. The **gen_matrix_mult_ext** routine partitions the matrix A into block columns, or panels, A_i, of size $n \times blk$:

$$A = [\, A_1, A_2, ..., A_m \,].$$

The last panel, A_m, contains fewer than blk columns if blk is not a divisor of n. The block size should be large enough to optimize machine utilization. Besides the allocation of X and Y, the in-core memory requirement for **gen_matrix_mult_ext** is approximately $(2v + 16)n*blk$ bytes, where v is the number of bytes in the data type of A.

EXAMPLES

Sample CM Fortran code that uses the external matrix multiplication routine can be found on-line in the subdirectory

```
external/matrix-multiply/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.1, June 1993

3.9 References

For more information about the basic linear routines for dense matrices, see the following references:

1. Dongarra, J. J., J. Du Croz, S. Hammarling, and R. J. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms.* Argonne National Laboratories, Mathematics and Computer Science Division, Technical Memorandum 41, November 1986.

2. Dongarra, J. J., J. Du Croz, I. Duff, and S. Hammerling. *A Set of Level 3 Basic Linear Algebra Subprograms.* Argonne National Laboratories, Mathematics and Computer Science Division, Reprint No. 1, August 1988.

3. Dongarra, J. J., J. Du Croz, I. Duff, and S. Hammerling. *A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs.* Argonne National Laboratories, Mathematics and Computer Science Division, Reprint No. 2, August 1988.

4. Golub, G. H., and Van Loan, C. F. *Matrix Computations.* 2d ed. Baltimore: Johns Hopkins University Press, 1989; or any basic linear algebra text.

5. Johnsson, S. L. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing* **4** (1987): 133-72.

6. Johnsson, S. L., T. Harris, and K. K. Mathur. Matrix Multiplication on the Connection Machine. In *Proceedings of Supercomputing '89,* ACM Press, New York, 1989. Pp. 326-32.

7. Cannon, L. E. *A Cellular Computer to Implement the Kalman Filter Algorithm.* Ph.D. diss., Montana State University, 1969.

8. Mathur, K. K. and S. L. Johnsson. Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer. Thinking Machines Corporation Technical Report TR-216, 1992.

9. Johnsson, S. L. and L. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for Distributed Memory Architectures and Languages with Array Syntax. *Int. J. Supercomputer App.* **6**, no. 4 (1992): 322-50.

For more information specifically about the infinity norm, see the following references:

10. Hager, W. W. Condition Estimates. *SIAM J. Sci. Stat. Comput.* **5** (1984): 311–16.

11. Higham, N. J. Experience with a Matrix Norm Estimator. *SIAM J. Sci. Stat. Comput.* **11**, no. 4 (1990): 804–9.

12. Higham, N. J. FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation. (Algorith 674) *ACM Trans. Math. Soft.* **14** (1988): 381–96.

Chapter 4

Sparse Matrix Operations

This chapter describes the CM Fortran interface to the basic linear algebra operations for sparse matrices. One section is devoted to each of the following topics:

- introduction
- arbitrary elementwise sparse matrix operations
- arbitrary block sparse matrix operations
- grid sparse matrix operations
- references

4.1 Introduction

The CMSSL provides routines for basic linear algebra operations on sparse matrices representing structured and unstructured grids. Both elementwise and block sparse matrices are supported. The following operations are provided for arbitrary elementwise sparse matrices, arbitrary block sparse matrices, and grid sparse matrices:

- sparse matrix x vector
- vector x sparse matrix
- sparse matrix x dense matrix
- dense matrix x sparse matrix

NOTE

The sparse matrix routines described in this chapter are intended for general sparse matrices and (in the case of the grid sparse matrix operations) for certain banded sparse matrices. For banded sparse matrices that cannot be handled by the grid sparse matrix operations, you can improve performance significantly by writing your own multiplication routine that exploits the band structure. More routines for banded sparse matrices are planned for future CMSSL releases.

4.1.1 Arbitrary Sparse Matrix Operations

The primary intent of the arbitrary sparse matrix operations is to provide the basic building blocks for more complex sparse applications — for example, a sparse iterative solver, or computation of the eigenvalues of the sparse matrices by the Lanczos or Arnoldi method.

For applications that do not perform explicit sparse linear algebra operations, but want to make use of some communication primitives used by the sparse basic linear algebra functions, the CMSSL provides two utility functions: the gather utility and the scatter utility. These utilities, which are described in Chapter 14, are intended for use in applications such as the solution of partial differential equations on unstructured discretizations, and optimization problems represented by sparse matrices occurring in network flow problems. A communication compiler and a partitioning routine are also provided (see Chapter 14).

Storage Representations

Two separate storage representations of the sparse matrix are supported (see references 2 and 3 listed in Section 4.5). These data mappings are referred to as the elementwise sparse matrix mapping and the block sparse matrix mapping. In the elementwise data mapping, the zero data values of the matrix are ignored and the

non-zero data values are stored row-wise. In the block sparse mapping, the sparse matrix is stored as a collection of dense block matrices. In its full matrix representation, this block matrix storage scheme is extremely flexible. The dense blocks need not be composed of contiguous rows and columns, and may overlap in any way. One possible application for the block sparse representation is the finite element method. Structured finite element grids lead to a grid block sparse data layout; unstructured grids result in an arbitrary block sparse layout. The two storage schemes are described in more detail in Sections 4.2 and 4.3.

Gathering and Scattering

The CMSSL sparse matrix operations can be described briefly in three steps (see reference 1). First, the source vector (or matrix) elements are "gathered" into local vectors. The relevant local operation (matrix vector or matrix matrix) is then performed. Finally, the results of the local operations are "scattered" back to the destination vector (or matrix). If there is collision at the destination, the colliding data values are added. (Note that in this context, "local" means local to a block, and does not refer to the lower-level implementation or to processing elements.) Examples illustrating the gathering and scattering processes are provided in Sections 4.2 and 4.3.

Optimization Switches

The arbitrary sparse matrix functions described in this chapter provide two optimization switches. These optimizations are based on the premise that the applications will use these sparse functions repeatedly. A marginal setup cost can therefore be incurred before the first call to the sparse functions. The setup cost is then amortized over several calls to the sparse matrix functions.

The first optimization switch allows the application to preprocess the "gather" phase of the operation (see references 4 and 5). This strategy usually results in a significant improvement in the performance of the function. The pre-processing phase requires additional processing element storage. The amount of storage required is a strong function of the sparsity of the matrix and is determined at run time by the setup functions. It is highly recommended that this additional storage be freed as soon as the application is finished with the sparse functions. The deallocation routines are also described in this chapter.

The second optimization feature provided by the sparse matrix operations is the ability to permute the array elements randomly (see references 4, 6, 7, and 8).

This process is referred to as a random permutation throughout this chapter. Random permutations of the array elements are particularly useful in reducing the routing conflicts that occur, and can reduce the time for data motion significantly. Setup routines are provided to compute the random permutations. The setup routines return the relevant array masks and the location of the vector (or matrix) elements after the random permutations. Most applications use the sparse matrix vector products to produce vectors (or matrices) that are then used in other operations such as inner products and local arithmetic. With the proper use of the masks, those other operations are invariant to the location of the vector (or matrix) elements. Thus, the products need not be permuted back in the inner loop of an application. Applications intending to use the sparse matrix functions are strongly encouraged to use both the optimization switches.

Optimizing Array Layout

As with most other CMSSL operations, the performance of the sparse matrix operations is a very strong function of the compiler layout directives used by the application. In particular, the block sparse functions perform significantly better when each dense block composing the sparse matrix is local to (contained within) a processing element. You can achieve this result by using the detailed axis descriptors of the CM Fortran **CMF$LAYOUT** directive.

4.1.2 Grid Sparse Matrix Operations

The grid sparse matrix routines operate on data from grid-based applications. Coefficient matrix elements residing at each grid point *P* are multiplied by vector or matrix elements residing at point *P* and its nearest-neighbor points. The result is placed in product vector or matrix elements residing at point *P*. These routines support multiple instances and block matrices.

Like the arbitrary sparse matrix routines, the grid sparse routines are designed with the assumption that the application will use these functions repeatedly. A marginal setup cost can therefore be incurred before the first call to the functions. The setup cost is then amortized over several calls.

4.2 Arbitrary Elementwise Sparse Matrix Operations

This section introduces the arbitrary elementwise sparse matrix operations. For detailed information about the routines and their arguments, refer to the man page at the end of this section.

4.2.1 The Arbitrary Elementwise Sparse Matrix Routines

Given a sparse matrix and a vector or dense matrix, the arbitrary elementwise sparse matrix routines compute the product of the sparse matrix with the vector or dense matrix. The following routines are provided:

sparse_matvec_mult	Multiplies a sparse matrix by a vector.
sparse_vecmat_mult	Multiplies a vector by a sparse matrix.
sparse_mat_gen_mat_mult	Multiplies a sparse matrix by a dense matrix.
gen_mat_sparse_mat_mult	Multiplies a dense matrix by a sparse matrix.

The two routines in which the sparse matrix is the left-hand operand (**sparse_matvec_mult** and **sparse_mat_gen_mat_mult**) use the following setup and deallocation routines:

sparse_matvec_setup
deallocate_sparse_matvec_setup

The two routines in which the sparse matrix is the right-hand operand (**sparse_vecmat_mult** and **gen_mat_sparse_mat_mult**) use the following setup and deallocation routines:

sparse_vecmat_setup
deallocate_sparse_vecmat_setup

For information about setup and deallocation, refer to the Description section of the man page following this section.

4.2.2 Storage of Sparse Matrices

Before calling the arbitrary elementwise sparse matrix routines, you must create a vector (one-dimensional CM array) *A* to represent the sparse matrix. You must

also supply associated vectors, ***rows***, ***cols***, ***row_segments***, and ***A_mask***, and integer values, *nrow* and *ncol*. Refer to the man page following this section for descriptions of these arguments. The following example is based on the argument definitions in the man page.

Example

The sparse matrix

$$\begin{bmatrix} 1 & 0 & 4 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 5 & 0 & 1 & 0 \end{bmatrix}$$

is represented by the vector

A = [1 4 2 3 5 1]

along with associated vectors

rows = [1 1 2 3 4 4]

cols = [1 3 2 4 1 3]

row_segments = [T F T T T F]

In this case, since you need not mask any elements of ***A***, you can supply the scalar logical value **.true.** for the ***A_mask*** argument.

If you defined ***A*** to have extent 10, that is,

A = [1 4 2 3 5 1 0 0 0 0]

then the corresponding vectors would be

rows = [1 1 2 3 4 4 0 0 0 0]

cols = [1 3 2 4 1 3 0 0 0 0]

row_segments = [T F T T T F F F F F]

A_mask = [T T T T T T F F F F]

Although other representations are possible, the current implementation requires that the elements of each row be contiguous with one another.

For discussions of this common method of storing sparse matrices, see references 2 and 3 in the list in Section 4.5.

4.2.3 Saving the Trace

One preprocessing step in arbitrary elementwise sparse matrix operations is the calculation of an optimization, or *trace*, for the communication pattern required by the multiplication. The trace depends on the sparsity of the matrix (that is, the positions of the non-zero elements); matrices with the same sparsity result in the same trace.

If you set the *itrace* argument to 0 when you call **sparse_matvec_setup** or **sparse_vecmat_setup**, the trace will be computed separately for each multiplication operation. However, if you are performing more than one multiplication operation with matrices that have identical sparsities, you can improve performance significantly by having the setup routine calculate the trace once and save it; you pass this trace to subsequent multiplication routine calls. To activate this option, set *itrace* = 1 when you call **sparse_matvec_setup** or **sparse_vecmat_setup**. If the sparsity of the matrix changes, you must call the setup routine again to calculate a new trace.

The trade-off for the improved performance when you set *itrace* = 1 is that saving a trace requires a substantial amount of CM memory. To free this extra memory, you must call **deallocate_sparse_matvec_setup** or **deallocate_sparse_vecmat_setup** after all of the sparse matrix vector products associated with one setup call have finished.

4.2.4 Random Permutation of Source and Destination Array Element Locations

Along with the sparse matrix A, you must supply the arbitrary elementwise sparse matrix routines with a source array, x, containing one or more vectors or dense matrices to be multiplied by the sparse matrix; and a destination array, y, containing corresponding vectors or dense matrices into which the results of the multiplication are to be placed. The source and destination arrays may be the same variable. They have rank 1 if you are calling **sparse_matvec_mult** or

sparse_vecmat_mult, or rank 2 if you are calling **sparse_mat_gen_mat_mult** or **gen_mat_sparse_mat_mult**. When you call one of the setup routines, the contents of *x* and *y* are ignored; only the geometry is examined.

The **sparse_matvec_setup** or **sparse_matvec_setup** argument *irandom*, if set to 1, activates an option that uses an internal random permutation generator to return permutations of the source and destination array element locations. These permutations affect all subsequent multiplication calls associated with the setup call, as follows:

- Before calling the multiplication routines, you must permute the elements of *x* using the source array permutation returned by the setup routine.
- The multiplication routine permutes the elements of *y* using the destination array permutation returned by the setup routine. Thus, the product array is returned in permuted form.

Note that the *source* array permutation must be applied by your application, while the *destination* array permutation is applied by the multiplication routine.

This feature involves a marginal preprocessing cost, but is extremely useful for minimizing the routing conflicts that occur during the data motion phase of the multiplication. In some cases, the permutations can reduce the communication time and thus improve performance significantly. If you set *irandom* to 0, an identity permutation is returned for both arrays.

The setup routine returns the source and destination array permutations in the integer arrays ***where_is_x*** and ***where_is_y***, respectively. If the source and destination arrays have rank 2, each permutation moves elements within columns only; each location remains in its original column. If the source and destination arrays are the same variable, the same permutation is returned in ***where_is_x*** and ***where_is_y***. If you set *irandom* to 0, you may conserve memory by declaring ***where_is_x*** and ***where_is_y*** as scalar integers with the value 0.

Along with the source and destination arrays, you must supply the routines with two integer arguments, *x_length* and *y_length*, containing the true extents of the first axes of *x* and *y*, respectively. That is, *x_length* contains the number of active elements (for rank 1) or rows (for rank 2) in *x*, and *y_length* contains the number of active elements (for rank 1) or rows (for rank 2) in *y*. In the permuted source array that you provide to the multiplication routine, it is possible that the active elements will no longer be confined to the first *x_length* locations (or rows, for rank 2).

If your source or destination array contains elements that must be masked, you must supply the setup routine with a corresponding logical array, ***x_mask*** or ***y_mask***, that has the same axis extents and layout directives as ***x*** or ***y***, respectively. The setup routine ignores the initial contents of the masks. On return from the setup, the values of the masks reflect the permutations returned in ***where_is_x*** and ***where_is_y***. If the source or destination array requires no masking, you may provide the scalar logical value **.true.** for ***x_mask*** or ***y_mask***, respectively. (An example is provided below.)

NOTE

Product elements resulting from the multiplication are sent to the permuted ***y*** locations; thus, the ***y*** returned by each multiplication routine is the permuted destination array. Optionally, you may use the information in ***y_mask*** and ***where_is_y*** to permute the elements of ***y*** back to their original positions after the multiplication occurs. However, most applications do not require you to do this.

For detailed definitions of the returned values of ***x_mask***, ***y_mask***, ***where_is_x***, and ***where_is_y***, refer to the man page at the end of this section. The example below is based on the argument definitions in the man page.

For a discussion of random permutation of source vector element locations, see references 4, 6, 7, and 8 listed in Section 4.5.

Example

Suppose you want to multiply the sparse matrix

$$\begin{bmatrix} 1 & 0 & 4 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 5 & 0 & 1 & 0 \end{bmatrix}$$

by a vector of length 4. In this example, ***x*** is one-dimensional with declared extent 6, and *x_length* is 4:

$$x = [x_1 \; x_2 \; x_3 \; x_4 \; - \; -]$$

(The symbol - indicates a masked data element.) If you set *irandom* = 0, you can supply the scalar value 0 for ***where_is_x*** and ***where_is_y***; you need not permute your source array before supplying it to the multiplication routine; and the multiplication routine does not permute the destination elements.

However, suppose you set *irandom* to 1, and **sparse_matvec_setup** assigns ***where_is_x*** the values

where_is_x = [6 1 3 2 4 5]

and ***x_mask*** the values

x_mask = [T T T F F T].

In this case, you must permute the source array element locations as follows:

$$x = [x_2 \; x_4 \; x_3 \; - \; - \; x_1]$$

That is, you must use this template when permuting the elements of each ***x*** you supply in subsequent **sparse_matvec_mult** calls associated with this setup.

In this same example, suppose *y* has declared extent 4 and true extent *y_length* = 4. In this case, since there is no need for a destination mask, you can supply the single scalar value ***y_mask*** = **.true.**. If **sparse_matvec_setup** assigned ***where_is_y*** the values

where_is_y = [2 1 4 3],

then the destination array is

$$y = [2x_2 \quad x_1+4x_3 \quad 5x_1+x_3 \quad 3x_4].$$

Arbitrary Elementwise Sparse Matrix Operations

Given a sparse matrix and a vector or dense matrix, the routines described below compute the product of the sparse matrix with the vector or dense matrix.

SYNTAX

sparse_matvec_setup (*A_mask, row_segments, rows, cols, y, x, y_mask, x_mask, where_is_x, where_is_y, y_length, x_length, irandom, itrace, trace, ier*)

sparse_matvec_mult (*y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask, itrace, trace, ier*)

sparse_mat_gen_mat_mult (*y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask, itrace, trace, ier*)

deallocate_sparse_matvec_setup (*trace, itrace*)

sparse_vecmat_setup (*A_mask, row_segments, rows, cols, y, x, y_mask, x_mask, where_is_x, where_is_y, y_length, x_length, irandom, itrace, trace, ier*)

sparse_vecmat_mult (*y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask, itrace, trace, ier*)

gen_mat_sparse_mat_mult (*y, A, x, rows, cols, row_segments, y_mask, A_mask, x_mask, itrace, trace, ier*)

deallocate_sparse_vecmat_setup (*trace, itrace*)

ARGUMENTS

y CM array of the same rank as *x* and the same data type (real or complex) as *A*. May be the same variable as *x*. The contents of this array are ignored by the setup routines. Upon return from one of the multiplication routines, contains the product of the sparse matrix and *x*.

A Real or complex CM array of rank 1 containing — in packed storage — the non-zero elements of the sparse matrix. The elements of each row must be contiguous with one another. The extent of *A* may be larger than the number of non-zero elements in the sparse matrix.

A_mask If *A* contains elements that need masking, ***A_mask*** must be a logical CM array of rank 1 with the same extent and layout directives as *A*; it is used as a mask for *A*. Set an element of ***A_mask*** to **.true.** if the corresponding element of *A* is to be treated as a non-zero element of the sparse matrix. Supply values for ***A_mask*** before calling the setup routine; then use the same ***A_mask*** when calling the associated multiplication routine.

If *A* does not contain elements that need masking, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***A_mask***.

row_segments Logical CM array of rank 1 with the same extent and layout directives as *A*. Contains information about the sparsity of the matrix. Set an element of ***row_segments*** to **.true.** if and only if the corresponding element of *A* is the first non-zero element in a row of the sparse matrix. Supply values for ***row_segments*** before calling the setup routine. The setup routine does not alter the values of ***row_segments***. You must supply the same ***row_segments*** values when calling the associated multiplication routine; do not modify ***row_segments*** between the setup call and the associated multiplication call.

rows Integer, *one-based* CM array of rank 1 with the same extent and layout directives as *A*. When you call the setup routine, each element of ***rows*** must contain the row number, in the sparse matrix, of the corresponding element of *A*. Do not modify ***rows*** after the setup routine returns; you must supply the multiplication routine with the values contained in ***rows*** upon return from the associated setup routine.

cols Integer, *one-based* CM array of rank 1 with the same extent and layout directives as *A*. When you call the setup routine, each element of ***cols*** must contain the column number, in the sparse matrix, of the corresponding element of *A*. Do not modify ***cols*** after the setup routine returns; you must supply the multiplication routine with the values contained in ***cols*** upon return from the associated setup routine.

x CM array of rank 1 or 2 and of the same data type and precision as *A*. May be the same variable as *y*. The contents of this array are ignored by the setup routines. Before calling the multiplication routine, you must apply to *x* the permutation, if any, indicated by the values that the setup routine assigned to ***where_is_x***.

y_mask If *y* contains elements that need masking, ***y_mask*** must be a logical CM array with the same rank, axis extents, and layout directives as *y*; it is used as a mask for the destination array. The setup routine ignores the initial contents. On return from the setup routine, ***y_mask*** has the following values:

- If *irandom* = 0 and *y* has rank 1, ***y_mask*** (1:*y_length*) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *irandom* = 0 and *y* has rank 2, then within each column of ***y_mask***, ***y_mask***(1:*y_length, l*) = **.true.** and all other elements of the column are **.false.**.
- If *irandom* = 1 and *y* has rank 1, then ***y_mask***(***where_is_y*** (1:*y_length*)) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *irandom* = 1 and *y* has rank 2, then within each column of ***y_mask***, ***y_mask***(***where_is_y*** (*k,l*), *l*)) = **.true.** for *k* = 1:*y_length*; all other elements of the column are **.false.**.

Do not modify ***y_mask*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***y_mask*** by the setup routine.

If *y* does not contain elements that need masking, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***y_mask***.

x_mask If *x* contains elements that need masking, ***x_mask*** must be a logical CM array with the same rank, axis extents, and layout directives as *x*; it is used as a mask for the source array. The setup routine ignores the initial contents. On return from the setup routine, ***x_mask*** has the following values:

- If *irandom* = 0 and *x* has rank 1, ***x_mask***(1:*x_length*) = **.true.** and all other elements of ***x_mask*** are **.false.**.

- If *irandom* = 0 and *x* has rank 2, then within each column of ***x_mask***, ***x_mask***(1:*x_length*, *j*) = **.true.**; all other elements of the column are **.false.**.
- If *irandom* = 1 and *x* has rank 1, then ***x_mask***(***where_is_x***(1:*x_length*)) = **.true.**; all other elements of ***x_mask*** are **.false.**.
- If *irandom* = 1 and *x* has rank 2, then within each column of ***x_mask***, ***x_mask***(***where_is_x***(*k*, *l*), *l*)) = **.true.** for *k* = 1:*x_length*; all other elements of the column are **.false.**.

Do not modify ***x_mask*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***x_mask*** by the setup routine.

If *x* does not contain elements that need masking, you can conserve processing element memory by supplying the scalar logical value **.true.** for ***x_mask***.

where_is_x If you set *irandom* = 0, you can conserve processing element memory by supplying the scalar integer value 0 for ***where_is_x***.

If you set *irandom* = 1, ***where_is_x*** must be an integer CM array with the same rank, axis extents, and layout directives as *x*. The initial contents are ignored. On return from the setup routine, ***where_is_x*** has the following values:

- If *irandom* = 0, ***where_is_x***(*k*) (for rank 1) or ***where_is_x***(*k*, *l*) (for rank 2) is simply *k*.
- If *irandom* = 1 and *x* has rank 1, ***where_is_x***(*k*) is the location to which the *k*th source array location must be mapped.
- If *irandom* = 1 and *x* has rank 2, ***where_is_x***(*k*,*l*) is the row number to which location (*k*, *l*) of the source array must be mapped.

where_is_y If you set *irandom* = 0, you can conserve processing element memory by supplying the scalar integer value 0 for ***where_is_y***.

If you set *irandom* = 1, ***where_is_y*** must be an integer CM array with the same rank, axis extents, and layout directives as *y*. The

initial contents are ignored. On return from the setup routine, ***where_is_y*** has the following values:

- If *irandom* = 0, ***where_is_y***(*k*) (for rank 1) or ***where_is_y***(*k*,*l*) (for rank 2) is simply *k*.
- If *irandom* = 1 and *y* has rank 1, ***where_is_y***(*k*) is the location to which the *k*th destination array location will be mapped by the multiplication routine.
- If *irandom* = 1 and *y* has rank 2, ***where_is_y***(*k*,*l*) is the row number to which location (*k*, *l*) of the destination array will be mapped by the multiplication routine.

y_length Scalar integer variable. The true extent of the first axis of *y*.

x_length Scalar integer variable. The true extent of the first axis of *x*.

irandom Scalar integer variable. Must be 0 or 1. Setting *irandom* to 1 causes the setup routine to return random permutations of the source and destination array element locations. If *irandom* is 0, identity permutations are returned.

itrace Scalar integer variable. Must be 0 or 1. When you call the setup routine, set *itrace* to 1 if you want the setup routine to calculate and save an optimization, or *trace*, for the communication pattern corresponding to the sparsity of the matrix. Set *itrace* to 0 if you want each multiplication routine to calculate the trace individually. The setup routine modifies the contents of *itrace*. Do not modify *itrace* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *itrace* upon return from the setup routine.

trace Scalar integer variable. Internal variable. If you supplied *itrace* = 1 when calling the setup routine, then on return from the setup routine, *trace* contains the address in CM memory where the trace is stored. Do not modify *trace* after the setup routine returns; you must supply the associated multiplication and deallocation routines with the value contained in *trace* upon return from the setup routine.

ier Scalar integer variable. Upon return from the setup routines, *ier* contains one of the following codes:

0	Normal return.
-1	*irandom* is not equal to 0 or 1.
-2	*itrace* is not equal to 0 or 1.
-4	*x_length* is greater than the extent of the first axis of ***x***, or *y_length* is greater than the extent of the first axis of ***y***.
-8	***x***, ***x_mask***, and ***where_is_x*** do not have the same shape, or ***y***, ***y_mask***, and ***where_is_y*** do not have the same shape.
-16	***A_mask***, ***row_segments***, ***rows***, and ***cols*** do not have the same shape.
-64	*trace* is too large to fit in available memory.

Upon return from the multiplication routines, *ier* contains one of the following codes:

0	Normal return.
-1	***A***, ***x***, or ***y*** does not contain real or complex data.

DESCRIPTION

The arbitrary elementwise sparse matrix routines perform the operations listed below. (In the formulas below, x and y denote vectors while X and Y denote matrices. However, lowercase letters are used for both cases everywhere else in this text.)

sparse_matvec_mult	$y = Ax$	multiplies a sparse matrix by a vector
sparse_vecmat_mult	$y^T = x^T A$	multiplies a vector by a sparse matrix
sparse_mat_gen_mat_mult	$Y = AX$	multiplies a sparse matrix by a dense matrix
gen_mat_sparse_mat_mult	$Y^T = X^T A$	multiplies a dense matrix by a sparse matrix

The sparse matrix and the vector must be of the same data type (real or complex) and the same precision; the sparse matrix is stored in packed form (vector argument ***A***), as described in the argument list.

To multiply a sparse matrix by a vector or dense matrix, follow these steps:

1. Call **sparse_matvec_setup.**
2. Call **sparse_matvec_mult** or **sparse_mat_gen_mat_mult.**

   To compute more than one product using sparse matrices that all have identical sparsities, follow one call to **sparse_matvec_setup** with multiple calls to

sparse_matvec_mult or **sparse_mat_gen_mat_mult**. If the sparsity changes, start with Step 1 again.

3. After all **sparse_matvec_mult** or **sparse_mat_gen_mat_mult** calls associated with the same **sparse_matvec_setup** call, call **deallocate_sparse_matvec_setup** to deallocate the CM storage space required by the setup routine.

To multiply a vector or dense matrix by a sparse matrix, follow these steps:

1. Call **sparse_vecmat_setup.**

2. Call **sparse_vecmat_mult** or **gen_mat_sparse_mat_mult.**

   To compute more than one product using sparse matrices that all have identical sparsities, follow one call to **sparse_vecmat_setup** with multiple calls to **sparse_vecmat_mult** or **gen_mat_sparse_mat_mult**. If the sparsity changes, start with Step 1 again.

3. After all **sparse_vecmat_mult** or **gen_mat_sparse_mat_mult** calls associated with the same **sparse_vecmat_setup** call, call **deallocate_sparse_vecmat_setup** to deallocate the CM storage space required by the setup routine.

More than one setup may be active at a time. That is, you may call the setup routine more than once without calling the deallocation routine.

Setup Phase. The setup routine analyzes the sparsity of the matrix, allocates CM storage space for the matrix vector multiplication, and places appropriate values in variables required by the multiplication routines.

The setup routine provides two options that may improve performance significantly:

- If you set *itrace* = 1, the setup routine calculates and saves the trace corresponding to the sparsity of the matrix for use in subsequent calls to the multiplication routines. The setup routine also allocates the additional storage space required for the trace.

- If you set *irandom* = 1, the setup routine returns random permutations of the source and destination array element locations in ***where_is_x*** and ***where_is_y***, respectively. (If the source and destination arrays are the same variable, the same permutation is applied to both arrays.) You must apply the permutation indicated in ***where_is_x*** to the source arrays you supply in subsequent multiplication calls. The permutation indicated in ***where_is_y*** is applied to the destination array by the multiplication routine. An example is provided in Section 4.2.4.

Multiplication Phase. Given a source CM array, *x*, and a sparse matrix represented as a packed vector *A*, each multiplication routine computes the product of the sparse matrix with *x* and returns the product in the CM array *y*.

Deallocation Phase. The **deallocate_sparse_matvec_setup** and **deallocate_sparse_vecmat_setup** routines deallocate the memory that was allocated for a trace in a previous call to **sparse_matvec_setup** or **sparse_vecmat_setup**, respectively. Each setup call in which *itrace* = 1 should be followed (after one or more associated calls to the multiplication routines) by a deallocation call. In fact, it is good practice to issue a call to the deallocation routine for *every* setup call. (If *itrace* was set to 0 in the setup call, the deallocation call has no effect.)

NOTES

Argument Values. Do not alter the contents of *trace*, ***rows***, ***cols***, ***row_segments***, ***x_mask***, ***y_mask***, ***where_is_x***, ***where_is_y***, or *itrace* between a call to the setup routine and a subsequent, associated call to a multiplication routine, for the following reasons:

- You must supply the multiplication routine with the values that the setup routine assigns to *trace*, ***rows***, ***cols***, ***row_segments***, and *itrace*.
- You must supply the deallocation routine with the values that the setup routine assigns to *trace* and *itrace*.
- If you set *irandom* to 1 when calling the setup routine, you must use the values that the setup routine assigns to ***x_mask*** and ***where_is_x*** to permute the elements of each *x* you supply in subsequent multiplication calls. (Refer to the on-line sample code for an example.) The values that the setup routine assigns to ***y_mask*** and ***where_is_y*** determine the permutation that the multiplication routine will apply to the destination elements.

If the setup routine permutes the source array element locations (*irandom* = 1), it also alters the contents of ***rows*** (and of ***cols***, if *itrace* = 0) appropriately to reflect the permutation so that the multiplication will occur correctly. (The multiplication routines use the contents of ***rows*** to perform the communication for the multiplication. If you supplied *itrace* = 0 to the setup routine, the multiplication routines also use the information stored in ***cols***.)

The product array *y* is the only argument updated by a call to one of the multiplication routines.

Overlapping Variables. For square matrices, you can use the same variable for *x* as for *y*, and you can use the same variable for *where_is_x* as for *where_is_y*.

Numerical Stability. The arbitrary elementwise sparse matrix operations are stable.

Numerical Complexity. If the vector *A* has length *n*, the sparse matrix vector and vector sparse matrix multiplication operations require approximately $2n$ floating-point operations if *A* is real, or approximately $8n$ floating-point operations if *A* is complex.

If the vector *A* has length *n* and *x* has *r* columns, the sparse matrix dense matrix and dense matrix sparse matrix operations require approximately $2nr$ floating-point operations if *A* is real, or $8nr$ floating-point operations if *A* is complex.

EXAMPLES

Sample CM Fortran code that uses the arbitrary elementwise sparse matrix routines can be found on-line in the subdirectory

```
sparse-matrix-vector/cmf
```

of a CMSSL examples directory whose location is site-specific.

4.3 Arbitrary Block Sparse Matrix Operations

This section introduces the arbitrary block sparse matrix operations. For detailed information about the routines and their arguments, refer to the man page at the end of this section.

4.3.1 The Arbitrary Block Sparse Matrix Routines

Given a block sparse matrix, a vector or dense matrix, and gathering and scattering pointer arrays, the arbitrary block sparse matrix routines compute the product of the block sparse matrix with the vector or dense matrix. The following routines are provided:

block_sparse_setup	Allocates processing element memory for the operation.
block_sparse_matrix_vector_mult	Multiplies a block sparse matrix by a vector.
vector_block_sparse_matrix_mult	Multiplies a vector by a block sparse matrix.
block_sparse_mat_gen_mat_mult	Multiplies a block sparse matrix by a dense matrix.
gen_mat_block_sparse_mat_mult	Multiplies a dense matrix by a block sparse matrix.
deallocate_block_sparse_setup	Deallocates memory allocated by **block_sparse_setup.**

For information about setup and deallocation, refer to the Description section of the man page following this section.

4.3.2 Block Representation, Gathering, and Scattering

Each block of data in a block sparse matrix is identified by a set of *m* row numbers and *n* column numbers. A block may overlap itself or other blocks. Blocks need not be contiguous, and the rows and columns within a block need not be contiguous.

When you call the block sparse matrix routines, you must embed the blocks of the block sparse matrix in a three-dimensional CM array, *A*, with declared extents *A*(*dim_1*, *dim_2*, *dim_3*) and true extents *A*(*m*, *n*, *p*). The first two axes represent the rows and columns of the blocks (which are assumed to be dense); the third axis counts the blocks. Thus, each of *p* blocks is represented by an *m* x *n* dense matrix within *A*. Rows and columns must be preserved in this representation; that is, elements of a block that occur in the same row (or column) in the block sparse matrix must occur in the same row (or column) when embedded in *A*.

The source array, *x*, and destination array, *y*, may be of rank 1 or 2 (with axes of any lengths), may be the same variable, and are assumed to be dense. The elements to be multiplied with each block of *A* are gathered from the source array, and the results of each block multiplication are scattered to form the destination array. You must supply the **block_sparse_setup** routine with two arrays, ***x_pointers*** and ***y_pointers***, containing pointers for gathering elements from the source array and scattering elements to the destination array, respectively.

The ***x_pointers*** and ***y_pointers*** arrays indicate the locations of the blocks within the block sparse matrix. The location of element *A*(*i*, *j*, *k*) within the block sparse matrix is given by (***y_pointers***(*i*, *k*), ***x_pointers***(*j*, *k*)). (See Example 1 in Section 4.3.5.)

The elements of ***x_pointers*** identify the *x* elements that are to be multiplied by the blocks of *A*; the elements of ***y_pointers*** identify the *y* locations to which the resulting product elements are to be scattered.

For **block_sparse_matrix_vector_mult**, the gather operation can be expressed in array notation as

$$\text{forall}(i = 1{:}n,\ j = 1{:}p)u(i, j) = x(x\_pointers(i, j))$$

and the scatter operation can be expressed as

$$\text{forall}(i = 1{:}m,\ j = 1{:}p)\quad y(y\_pointers(i, j)) = y(y\_pointers(i, j)) + v(i, j)$$

where *u*(*i*,*j*) (for $i = 1{:}n, j = 1{:}p$) contains the source array elements to be multiplied with the *j*th block of *A*, and *v*(*i*,*j*) (for $i = 1{:}m, j = 1{:}p$) contains the resulting product elements to be scattered to the destination array. For **vector_block_sparse_matrix_mult**, the same definitions apply, but *m* and *n* are switched. For **block_sparse_mat_gen_mat_mult** and **gen_mat_block_sparse_mat_mult**, these definitions are extended by one dimension.

Detailed definitions of ***x_pointers*** and ***y_pointers*** are provided in the man page. Section 4.3.5 presents examples of how gathering and scattering work in block

sparse matrix vector multiplication and dense matrix block sparse matrix multiplication.

4.3.3 Saving the Trace

The block sparse matrix routines calculate an optimization, or *trace*, for the communication pattern required by the multiplication. The trace depends on the contents of the pointer array ***x_pointers***. The trace can be computed by each multiplication routine. However, if you are performing more than one block sparse matrix operation, and if the operations all use the same pointer array ***x_pointers***, you can reduce communication time and thus improve performance significantly by having **block_sparse_setup** calculate the trace once and save it; you pass this trace to subsequent multiplication routine calls. To activate this option, set *itrace* = 1 when you call **block_sparse_setup**. If the contents of ***x_pointers*** change, you must call **block_sparse_setup** again. (The contents of ***y_pointers*** must also remain constant for all multiplication calls following a single **block_sparse_setup** call.)

The trade-off for the improved performance when you set *itrace* = 1 is that saving a trace requires a substantial amount of processing element memory. To free this extra memory, you must call **deallocate_block_sparse_setup** after all the block sparse matrix operations associated with one setup call have finished.

4.3.4 Random Permutation of Source and Destination Array Element Locations

The **block_sparse_setup** argument *irandom*, if set to 1, activates an option that uses an internal random permutation generator to return permutations of the source and destination array element locations. These permutations affect all subsequent block sparse multiplication calls associated with the setup call, as follows:

- Before calling the multiplication routines, you must permute your source array elements, using the source array permutation returned by the setup routine.
- The multiplication routine permutes the elements of the destination array using the destination array permutation returned by the setup routine. Thus, the destination array is returned in permuted form.

Note that the *source* array permutation must be applied by your application, while the *destination* array permutation is applied by the multiplication routine.

This feature involves a marginal preprocessing cost, but is extremely useful for minimizing the routing conflicts that occur during the data motion phase of the multiplication. In some cases, the permutations can reduce the communication time and thus improve performance significantly. If you set *irandom* to 0, an identity permutation is returned for both arrays.

The setup routine returns the source and destination array permutations in the integer arrays ***where_is_x*** and ***where_is_y***, respectively. If the source and destination arrays have rank 2, each permutation moves elements within columns only; each location remains in its original column.

Along with the source and destination arrays, you must supply the block sparse matrix routines with two integer arguments, *x_length* and *y_length*, containing the true extents of the first axes of ***x*** and ***y***, respectively. That is, *x_length* contains the number of active elements (for rank 1) or rows (for rank 2) in ***x***, and *y_length* contains the number of active elements (for rank 1) or rows (for rank 2) in ***y***. In the permuted source array that you provide to the multiplication routine, it is possible that the active elements will no longer be confined to the first *x_length* locations (or rows, for rank 2).

When you call the setup routine, you must also supply two logical arrays, ***x_mask*** and ***y_mask***, that have the same axis extents and layout directives as ***x*** and ***y***, respectively. The setup routine ignores the initial contents of the masks. On return from the setup, the values of the masks reflect the permutations returned in ***where_is_x*** and ***where_is_y***. (Examples are provided in Section 4.3.5.)

NOTE

Product elements from the block multiplications are scattered to the permuted *y* locations; thus, the *y* returned by each multiplication routine is the permuted destination array. Optionally, you may use the information in ***y_mask*** and ***where_is_y*** to permute the elements of *y* back to their original locations after the multiplication occurs. However, most applications do not require you to do this; for example, inner product computations on destination vectors are invariant under random permutation.

For detailed definitions of the returned values of ***x_mask***, ***y_mask***, ***where_is_x***, and ***where_is_y***, refer to the man page at the end of this section. The examples below are based on the argument definitions in the man page.

For a discussion of random permutation of source vector element locations, see references 4, 6, 7, and 8 listed in Section 4.5.

4.3.5 Examples

The following two examples show how gathering, scattering, and random permutation of the source and destination arrays work in

- block sparse matrix vector multiplication
- dense matrix block sparse matrix multiplication

These examples are based on the argument descriptions in the man page following this section. They use letters instead of numbers for array element values in some cases for clarity.

Example 1: Block Sparse Matrix Vector Multiplication

In this example, $m = 5$, $n = 4$, and $p = 3$. The coefficient block sparse matrix contains three blocks, each of size (5 x 4). It is represented by the CM array *A*,

which has declared extents (10, 10, 3) and true extents (5, 4, 3). The first (5 x 4) elements of each dense matrix within *A* have the following values:

$$A(:,:,1) = \begin{bmatrix} a & f & k & p \\ b & g & l & q \\ c & h & m & r \\ d & i & n & s \\ e & j & o & t \end{bmatrix} \quad A(:,:,2) = \begin{bmatrix} u & z & e & j \\ v & a & f & k \\ w & b & g & l \\ x & c & h & m \\ y & d & i & n \end{bmatrix} \quad A(:,:,3) = \begin{bmatrix} o & t & y & j \\ p & u & z & k \\ q & v & a & l \\ r & w & b & m \\ s & x & c & n \end{bmatrix}$$

The *x* argument is a vector with declared extent 8 and true extent *x_length* = 6. The elements to be gathered from *x* are assumed to be, originally, in the first six locations of the vector:

$$x = [\, x_1 \;\; x_2 \;\; x_3 \;\; x_4 \;\; x_5 \;\; x_6 \;\; - \;\; - \,]$$

(The symbol - indicates a masked data element.) In this example, when you call **block_ sparse_setup**, you set *irandom* = 1. The setup routine permutes the source array element locations, assigns ***x_mask*** the values

x_mask = [T F T F T T T T]

and assigns ***where_is_x*** the values

where_is_x = [7 3 5 6 1 8 4 2]

These values indicate that the source array element locations must be permuted as follows:

$$x = [\, x_5 \;\; - \;\; x_2 \;\; - \;\; x_3 \;\; x_4 \;\; x_1 \;\; x_6 \,]$$

That is, you must use this template when permuting the elements of each *x* you supply in subsequent multiplication calls associated with this setup call.

The ***x_pointers*** array must have declared extents (10 x 3) and true extents (*n* x *p*) or (4 x 3). In this example, you supply the following values in the first (4 x 3) locations of ***x_pointers*** when you call **block_sparse_setup**:

$$x\_pointers = \begin{bmatrix} 1 & 4 & 1 \\ 5 & 2 & 1 \\ 2 & 6 & 3 \\ 4 & 3 & 2 \end{bmatrix}$$

The ***x_pointers*** array determines the contents of the vectors of length $n = 4$ that will be multiplied on the left by the blocks of A. Element (i, j) of ***x_pointers*** contains the original location of the x element that is to be multiplied by the ith column of the jth block of A.

Note that the values of the ***x_pointers*** elements are all less than or equal to *x_length* = 6, since the elements to be gathered from x originally reside in the first six locations of x.

Given that you supplied the above ***x_pointers*** values to **block_sparse_setup**, the **block_sparse_matrix_vector_mult** routine will multiply the blocks $A(:, :, 1)$, $A(:, :, 2)$, and $A(:, :, 3)$ shown above by the following vectors, respectively:

$$u(:, 1) = \begin{bmatrix} x_1 \\ x_5 \\ x_2 \\ x_4 \end{bmatrix} \qquad u(:, 2) = \begin{bmatrix} x_4 \\ x_2 \\ x_6 \\ x_3 \end{bmatrix} \qquad u(:, 3) = \begin{bmatrix} x_1 \\ x_1 \\ x_3 \\ x_2 \end{bmatrix}$$

The results are the product vectors $v(:, 1)$, $v(:, 2)$, and $v(:, 3)$. For example,

$$v(:, 1) = A(:, :, 1)\,u(:, 1) = \begin{bmatrix} a & f & k & p \\ b & g & l & q \\ c & h & m & r \\ d & i & n & s \\ e & j & o & t \end{bmatrix} \begin{bmatrix} x_1 \\ x_5 \\ x_2 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1a + x_5f + x_2k + x_4p \\ x_1b + x_5g + x_2l + x_4q \\ x_1c + x_5h + x_2m + x_4r \\ x_1d + x_5i + x_2n + x_4s \\ x_1e + x_5j + x_2o + x_4t \end{bmatrix}$$

In this example, y has declared extent 8 and true extent *y_length* = 6, but is a different variable than x (and therefore undergoes a different permutation than x). The contents of v are scattered to form y using the pointers you supplied in the ***y_pointers*** argument when you called **block_sparse_setup**.

The ***y_pointers*** array must have declared extents (10 X 3) and true extents (m X p) or (5 X 3). Suppose you supplied the following values in the first (5 X 3) locations of ***y_pointers***:

$$y\_pointers = \begin{bmatrix} 3 & 5 & 1 \\ 4 & 2 & 3 \\ 1 & 4 & 2 \\ 5 & 3 & 6 \\ 1 & 2 & 1 \end{bmatrix}$$

Element (*i, j*) of ***y_pointers*** contains the original location of the *y* element to which element *v*(*i,j*) is to be scattered.

Note that the values of the ***y_pointers*** elements are all less than or equal to *y_length* = 6, since the locations of *y* to receive scattered product elements are originally the first six locations.

If you had set *irandom* to 0 when calling **block_sparse_setup**, the ***y_pointers*** values shown above would have caused **block_sparse_matrix_vector_mult** to assign *y* the values

$$y = \begin{bmatrix} v(3,1) + v(5,1) + v(1,3) + v(5,3) \\ v(2,2) + v(5,2) + v(3,3) \\ v(1,1) + v(4,2) + v(2,3) \\ v(2,1) + v(3,2) \\ v(4,1) + v(1,2) \\ v(4,3) \\ - \\ - \end{bmatrix}$$

Note that colliding values are added.

However, since you set *irandom* to 1 when calling **block_sparse_setup**, the contents of *v* are sent, not to the *y* locations specified in ***y_pointers***, but to the *new* locations to which those locations are mapped during the random permutation. If the setup routine assigned ***y_mask*** the values

y_mask = [F T F T T T T T]

and assigned ***where_is_y*** the values

where_is_y = [8 6 7 4 5 2 1 3]

then **block_sparse_matrix_vector_mult** assigns *y* the values

$$y = \begin{bmatrix} - \\ v(4,3) \\ - \\ v(2,1) + v(3,2) \\ v(4,1) + v(1,2) \\ v(2,2) + v(5,2) + v(3,3) \\ v(1,1) + v(4,2) + v(2,3) \\ v(3,1) + v(5,1) + v(1,3) + v(5,3) \end{bmatrix}$$

Recall that the location of element $A(i, j, k)$ within the block sparse matrix is given by (***y_pointers***(i, k), ***x_pointers***(j, k)). Applying this formula to the elements $A(:, :, 1)$, we see that this block is positioned as shown below in the original block sparse matrix:

	1	2	3	4	5
1	*c,e*	*m,o*		*r,t*	*h,j*
2					
3	*a*	*k*		*p*	*f*
4	*b*	*l*		*q*	*g*
5	*d*	*n*		*s*	*i*

This example illustrates the fact that blocks can be self-overlapping and can have non-contiguous rows and columns.

Example 2: Dense Matrix Block Sparse Matrix Multiplication

In this example, $m = 4$, $n = 3$, and $p = 2$. The coefficient block sparse matrix contains two blocks, each of size (4 x 3), and is embedded in the CM array *A*. *A* has declared extents (10, 10, 2) and true extents (4, 3, 2). The first (4 x 3) elements of each block have the following values:

$$A(:, :, 1) = \begin{bmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{bmatrix} \qquad A(:, :, 2) = \begin{bmatrix} m & q & u \\ n & r & v \\ o & s & w \\ p & t & x \end{bmatrix}$$

The *x* argument is a matrix with declared extents (8 x 8) and true extents (6 x 8); thus, *x_length* = 6. The elements to be gathered from *x* are assumed to be, originally, in the first (6 x 8) locations of the matrix:

$$x = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In this example, when you call **block_sparse_setup**, you set *irandom* = 1. The setup routine permutes the source array element locations within each column, assigns ***x_mask*** the values

$$x\_mask = \begin{bmatrix} T & T & T & F & T & T & F & T \\ T & T & T & T & T & T & T & T \\ F & T & F & T & T & T & T & F \\ T & T & T & T & F & F & T & F \\ T & T & T & T & F & T & T & T \\ F & F & T & T & T & T & T & T \\ T & F & F & F & T & F & F & T \\ T & T & T & T & T & T & T & T \end{bmatrix}$$

and assigns ***where_is_x*** the values

$$\textit{where\_is\_x} = \begin{bmatrix} 2 & 8 & 1 & 2 & 7 & 5 & 3 & 1 \\ 5 & 4 & 4 & 6 & 6 & 8 & 2 & 7 \\ 1 & 5 & 6 & 8 & 2 & 3 & 8 & 8 \\ 8 & 2 & 2 & 4 & 1 & 2 & 5 & 2 \\ 4 & 1 & 8 & 5 & 3 & 6 & 4 & 5 \\ 7 & 3 & 5 & 3 & 8 & 1 & 6 & 6 \\ 3 & 6 & 7 & 1 & 5 & 7 & 1 & 4 \\ 6 & 7 & 3 & 7 & 4 & 4 & 7 & 3 \end{bmatrix}$$

These values indicate that the source array element locations must be permuted as follows:

$$x = \begin{bmatrix} x_{31} & x_{52} & x_{13} & 0 & x_{45} & x_{66} & 0 & x_{18} \\ x_{11} & x_{42} & x_{43} & x_{14} & x_{35} & x_{46} & x_{27} & x_{48} \\ 0 & x_{62} & 0 & x_{64} & x_{55} & x_{36} & x_{17} & 0 \\ x_{51} & x_{22} & x_{23} & x_{44} & 0 & 0 & x_{57} & 0 \\ x_{21} & x_{32} & x_{63} & x_{54} & 0 & x_{16} & x_{47} & x_{58} \\ 0 & 0 & x_{33} & x_{24} & x_{25} & x_{56} & x_{67} & x_{68} \\ x_{61} & 0 & 0 & 0 & x_{15} & 0 & 0 & x_{28} \\ x_{41} & x_{12} & x_{53} & x_{34} & x_{65} & x_{26} & x_{37} & x_{38} \end{bmatrix}$$

That is, you must use this template when permuting the elements of each ***x*** you supply in subsequent multiplication calls associated with this setup call.

The ***x_pointers*** array must have declared extents (10 X 2) and true extents (*m* X *p*) or (4 X 2). In this example, you supply the following values in the first (4 X 2) locations of ***x_pointers*** when you call **block_sparse_setup**:

$$\textit{x\_pointers} = \begin{bmatrix} 2 & 3 \\ 1 & 5 \\ 6 & 4 \\ 3 & 1 \end{bmatrix}$$

The ***x_pointers*** array determines the contents of the matrices of extent (n x m) = (3 x 4) that will be multiplied on the right by the blocks of A. If element (i, j) of ***x_pointers*** contains the value k, then the **gen_mat_block_sparse_mat_mult** routine will multiply the ith row of the jth block of A by $x((where\_is\_x(k, l),l)$, when computing the lth row of the product.

Note that the values of the ***x_pointers*** elements are all less than or equal to *x_length* = 6, since the elements to be gathered from x originally reside in the first six rows of x.

Given that you supplied the above ***x_pointers*** values to **block_sparse_setup**, the **gen_mat_block_sparse_mat_mult** routine will multiply the blocks $A(:, :, 1)$ and $A(:, :,2)$ shown above by the following matrices, respectively:

$$u(:, :, 1) = \begin{bmatrix} x_{21} & x_{11} & x_{61} & x_{31} \\ x_{22} & x_{12} & x_{62} & x_{32} \\ x_{23} & x_{13} & x_{63} & x_{33} \end{bmatrix} \qquad u(:, :, 2) = \begin{bmatrix} x_{31} & x_{51} & x_{41} & x_{11} \\ x_{32} & x_{52} & x_{42} & x_{12} \\ x_{33} & x_{53} & x_{43} & x_{13} \end{bmatrix}$$

The results are the product matrices $v(:, :, 1)$ and $v(:, :, 2)$. For example,

$$v(:, :, 1) = u(:, :, 1)\, A(:, :, 1) = \begin{bmatrix} x_{21} & x_{11} & x_{61} & x_{31} \\ x_{22} & x_{12} & x_{62} & x_{32} \\ x_{23} & x_{13} & x_{63} & x_{33} \end{bmatrix} \begin{bmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{bmatrix}$$

In this example, y has declared extents (8 x 8) and true extents (6 x 8); thus, *y_length* = 6. Also, y is defined to be the same variable as x, and therefore undergoes the same random permutation as x. The contents of v are scattered to form y using the pointers you supplied in the ***y_pointers*** argument when you called **block_sparse_setup**.

The ***y_pointers*** array must have declared extents (10 x 2) and true extents (n x p) or (3 x 2). Suppose you supplied the following values in the first (3 x 2) locations of ***y_pointers***:

$$y\_pointers = \begin{bmatrix} 3 & 2 \\ 1 & 3 \\ 6 & 2 \end{bmatrix}$$

If element (i, j) of ***y_pointers*** contains the value k, then element $v(i, l, j)$ is scattered to location $y(\textbf{\textit{where\_is\_y}}((k, l),l)$.

Note that the values of the ***y_pointers*** elements are all less than or equal to *y_length* = 6, since the locations of *y* to receive scattered product elements are originally the first six rows of *y*.

If you had set *irandom* to 0 when calling **block_sparse_setup**, the ***y_pointers*** values shown above would have caused **gen_mat_block_sparse_mat_mult** to scatter the contents of *v* to the first (*m* x *n*) or (4 x 3) locations of *y* as follows:

$$y = \begin{bmatrix} v_{211} & v_{221} & v_{231} & 0 & 0 & 0 & 0 & 0 \\ v_{112} + v_{312} & v_{122} + v_{322} & v_{132} + v_{332} & 0 & 0 & 0 & 0 & 0 \\ v_{111} + v_{212} & v_{121} + v_{222} & v_{131} + v_{232} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_{311} & v_{321} & v_{331} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note that colliding values are added; and that since, in this example, each matrix $v(:, :, j)$ has dimensions (3 x 3), only the first three columns of *y* receive scattered product elements.

However, since you set *irandom* to 1 when calling **block_sparse_setup**, the contents of *v* are sent, not to the *y* locations specified in ***y_pointers***, but to the *new* locations to which those locations were mapped during the random permutation. Since the setup routine applied the same permutation to the source array and destination array element locations, it assigned ***y_mask*** the values

$$y\_mask = \begin{bmatrix} T & T & T & F & T & T & F & T \\ T & T & T & T & T & T & T & T \\ F & T & F & T & T & T & T & F \\ T & T & T & T & F & F & T & F \\ T & T & T & T & F & T & T & T \\ F & F & T & T & T & T & T & T \\ T & F & F & F & T & F & F & T \\ T & T & T & T & T & T & T & T \end{bmatrix}$$

and assigned ***where_is_y*** the values

$$where\_is\_y = \begin{bmatrix} 2 & 8 & 1 & 2 & 7 & 5 & 3 & 1 \\ 5 & 4 & 4 & 6 & 6 & 8 & 2 & 7 \\ 1 & 5 & 6 & 8 & 2 & 3 & 8 & 8 \\ 8 & 2 & 2 & 4 & 1 & 2 & 5 & 2 \\ 4 & 1 & 8 & 5 & 3 & 6 & 4 & 5 \\ 7 & 3 & 5 & 3 & 8 & 1 & 6 & 6 \\ 3 & 6 & 7 & 1 & 5 & 7 & 1 & 4 \\ 6 & 7 & 3 & 7 & 4 & 4 & 7 & 3 \end{bmatrix}$$

Therefore, **gen_mat_block_sparse_mat_mult** assigns *y* the values

$$y = \begin{bmatrix} v_{111} + v_{212} & 0 & v_{231} & 0 & 0 & 0 & 0 & 0 \\ v_{211} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{321} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{122} + v_{322} & v_{132} + v_{332} & 0 & 0 & 0 & 0 & 0 \\ v_{112} + v_{312} & v_{121} + v_{222} & v_{331} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & v_{131} + v_{232} & 0 & 0 & 0 & 0 & 0 \\ v_{311} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & v_{221} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Arbitrary Block Sparse Matrix Operations

Given a block sparse matrix, a vector or dense matrix, and gathering and scattering pointer arrays, the routines described below compute the product of the block sparse matrix with the vector or dense matrix.

SYNTAX

block_sparse_setup (*x_mask, y_mask, where_is_x, where_is_y, x_pointers, y_pointers, m, n, p, x_length, y_length, irandom, itrace, trace, trace_mask, setup, ier*)

block_sparse_matrix_vector_mult (*y, A, x, x_pointers, y_pointers, y_mask, m, n, p, x_length, y_length, setup, trace, trace_mask, ier*)

vector_block_sparse_matrix_mult (*y, A, x, x_pointers, y_pointers, y_mask, m, n, p, x_length, y_length, setup, trace, trace_mask, ier*)

block_sparse_mat_gen_mat_mult (*y, A, x, x_pointers, y_pointers, y_mask, m, n, p, x_length, y_length, setup, trace, trace_mask, ier*)

gen_mat_block_sparse_mat_mult (*y, A, x, x_pointers, y_pointers, y_mask, m, n, p, x_length, y_length, setup, trace, trace_mask, ier*)

deallocate_block_sparse_setup (*trace, trace_mask*)

ARGUMENTS

The following definitions assume that the coefficient block sparse matrix is embedded in a three-dimensional CM array, *A*; that *A* is declared with extents (*dim_1*, *dim_2*, *dim_3*); that the portion of *A* containing valid data has extents (*m*, *n*, *p*); and that the pointer arrays ***x_pointers*** and ***y_pointers*** are one-based.

y — CM array of type real or complex. Destination array. May be the same variable as *x*. The rank of *y* is 1 for **block_sparse_matrix_vector_mult** and **block_sparse_vector_matrix_mult**, and 2 for **block_sparse_mat_gen_mat_mult** and **gen_mat_block_sparse_mat_mult**. These multiplication routines assign values to *y* by using the pointers supplied in ***y_pointers*** to scatter elements from the block products. If you set *irandom* = 1 when calling

block_sparse_setup, the product elements are scattered to the permuted locations of *y*. (The permutation is given by the values of ***where_is_y*** assigned by the setup routine.)

Any initial values of *y* are overwritten by the multiplication routines. If the pointer array ***y_pointers*** causes more than one block product element to update the same *y* element during the scatter operation, the colliding values are added.

A CM array of rank 3 and the same data type and precision as *y*. Represents the block sparse matrix. The first two axes have true extents *m* and *n*, respectively, and count the rows and columns of the dense blocks. The third axis has true extent *p* and counts the blocks. Thus, *A* contains *p* dense blocks, each of size *m* x *n*. The location of element *A*(*i*, *j*, *k*) within the block sparse matrix is given by (***y_pointers***(*i*, *k*), ***x_pointers***(*j*, *k*)).

x CM array of the same data type and precision as *y*. Source array. Assumed to be dense. May be the same variable as *y*. The rank of *x* is 1 for **block_sparse_matrix_vector_mult** and **vector_block_sparse_matrix_mult**, and 2 for **block_sparse_mat_gen_mat_mult** and **gen_mat_block_sparse_mat_mult**. If you set *irandom* = 1 when calling **block_sparse_setup**, then before calling any of the multiplication routines, you must permute the elements of *x* using the permutation returned in ***where_is_x***.

x_mask Logical CM array. Mask for the source array *x*. Must have the same axis extents and layout directives as *x* (rank 1 or 2). The initial values are ignored. On return from **block_sparse_setup**, ***x_mask*** has the following values:

- If *irandom* = 0 and *x* has rank 1, ***x_mask***(1:*x_length*) = **.true.**; all other elements of ***x_mask*** are **.false.**.
- If *irandom* = 0 and *x* has rank 2, then within each column of ***x_mask***, ***x_mask***(1:*x_length*, *l*) = **.true.**; all other elements of the column are **.false.**.
- If *irandom* = 1 and *x* has rank 1, then ***x_mask***(***where_is_x***(1:*x_length*)) = **.true.**; all other elements of ***x_mask*** are **.false.**.

- If *irandom* = 1 and *x* has rank 2, then within each column of ***x_mask***, ***x_mask***(***where_is_x***(*k*,*l*), *l*) = **.true.** for *k* = 1:*x_length*; all other elements of the column are **.false.**.

y_mask Logical CM array. Mask for the destination array *y*. Must have the same axis extents and layout directives as *y* (rank 1 or 2). The initial values that you supply to **block_sparse_ setup** are ignored. On return from **block_sparse_setup**, ***y_mask*** has the following values:

- If *irandom* = 0 and *y* has rank 1, ***y_mask*** (1:*y_length*) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *irandom* = 0 and *y* has rank 2, then within each column of ***y_mask***, ***y_mask***(1:*y_length*, *l*) = **.true.** and all other elements of the column are **.false.**.
- If *irandom* = 1 and *y* has rank 1, then ***y_mask***(***where_is_y***(1:*y_length*)) = **.true.**; all other elements of ***y_mask*** are **.false.**.
- If *irandom* = 1 and *y* has rank 2, then within each column of ***y_mask***, ***y_mask***(***where_is_y*** (*k*,*l*), *l*)) = **.true.** for *k* = 1:*y_length*; all other elements of the column are **.false.**.

Do not modify ***y_mask*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***y_mask*** by **block_sparse_setup**.

where_is_x Integer CM array. Must have the same axis extents and layout directives as *x* (rank 1 or 2). The initial values are ignored. On return from **block_sparse_setup**, ***where_is_x*** has the following values:

- If *irandom* = 0, ***where_is_x***(*k*) (for rank 1) or ***where_ is_x***(*k*,*l*) (for rank 2) is simply *k*.
- If *irandom* = 1 and *x* has rank 1, ***where_is_x***(*k*) is the location to which the *k*th source array location must be mapped.
- If *irandom* = 1 and *x* has rank 2, ***where_is_x***(*k*,*l*) is the row number to which location (*k*, *l*) of the source array must be mapped.

where_is_y Integer CM array. Must have the same axis extents and layout directives as *y* (rank 1 or 2). The initial values are ignored. On return from **block_sparse_setup**, ***where_is_y*** has the following values:

- If *irandom* = 0, ***where_is_y***(*k*) (for rank 1) or ***where_is_y***(*k,l*) (for rank 2) is simply *k*.
- If *irandom* = 1 and *y* has rank 1, ***where_is_x***(*k*) is the location to which the *k*th destination array location will be mapped by the multiplication routine.
- If *irandom* = 1 and *y* has rank 2, ***where_is_x***(*k,l*) is the row number to which location (*k, l*) of the destination array will be mapped by the multiplication routine.

x_pointers Integer CM array of rank 2. *Must be one-based.* The elements of ***x_pointers*** identify the original locations of the ***x*** elements that are to be gathered into vectors (if ***x*** has rank 1) or matrices (if ***x*** has rank 2) to be multiplied by the blocks of ***A***.

Before calling **block_sparse_setup**, use the following guidelines to create ***x_pointers***:

- If you are planning to use **block_sparse_matrix_vector_mult** or **block_sparse_mat_gen_mat_mult** (in which *A* is the left-hand operand), declare ***x_pointers*** with extents (*dim_2, dim_3*) and place valid data in the subarray whose axes have extents *n* and *p*.
- If you are planning to use **vector_block_sparse_matrix_mult** or **gen_mat_block_sparse_mat_mult** (in which *A* is the right-hand operand), declare ***x_pointers*** with extents (*dim_1, dim_3*) and place valid data in the subarray whose axes have extents *m* and *p*.
- Assign the elements of ***x_pointers*** values less than or equal to *x_length*.

If ***x*** has rank 1 and ***x_pointers***(*i,j*) = *k*, then

- **block_sparse_matrix_vector_mult** multiplies the *i*th column of the *j*th block of *A* by ***x***(***where_is_x***(*k*)).

- **vector_block_sparse_matrix_mult** multiplies the *i*th row of the *j*th block of *A* by *x*(***where_is_x***(*k*)).

If *x* has rank 2 and ***x_pointers***(*i,j*) = *k*, then

- **block_sparse_mat_gen_mat_mult** multiplies the *i*th column of the *j*th block of *A* by *x*(***where_is_x***(*k,l*),*l*) when computing the *l*th column of the block product.

- **gen_mat_block_sparse_mat_mult** multiplies the *i*th row of the *j*th block of *A* by *x*(***where_is_x*** (*k,l*),*l*) when computing the *l*th row of the block product.

When defining ***x_pointers***, refer to the rules for using the same variable for two arguments, presented in the description below.

The ***x_pointers*** values you supply to the **block_sparse_setup** routine should refer to the original (unpermuted) locations of *x*. If you set *irandom* = 1, **block_sparse_setup** modifies the values of ***x_pointers*** so that the gathering operation occurs correctly. Do not modify the contents of ***x_pointers*** between the setup call and the associated multiplication call(s). When you call one of the multiplication routines, you must supply the values assigned to ***x_pointers*** by **block_sparse_setup**.

y_pointers Integer CM array of rank 2. *Must be one-based.* The elements of ***y_pointers*** identify the original (unpermuted) locations of the *y* elements that are to receive scattered product elements.

Before calling **block_sparse_setup**, use the following guidelines to create ***y_pointers***:

- If you are planning to use **block_sparse_matrix_vector_mult** or **block_sparse_mat_gen_mat_mult** (in which *A* is the left-hand operand), declare ***y_pointers*** with extents (*dim_1, dim_3*) and place valid data in the subarray whose axes have extents *m* and *p*.

- If you are planning to use **gen_mat_block_sparse_mat_mult** or **vector_block_sparse_matrix_mult** (in which *A* is the right-hand operand), declare ***y_pointers*** with extents (*dim_2, dim_3*) and place valid data in the subarray whose axes have extents *n* and *p*.

- Assign the elements of ***y_pointers*** values less than or equal to *y_length*.

The values of ***y_pointers*** determine the scattering pattern as follows:

- If *y* has rank 1 and ***y_pointers***$(i,j) = k$, then the *i*th element from the *j*th block product is scattered to *y*(***where_is_y***(k)).
- If *y* has rank 2 and ***y_pointers***$(i,j) = k$, then element (i, l) from the *j*th block product is scattered to *y*(***where_is_y***$(k,l),l)$.

When defining ***y_pointers***, refer to the rules for using the same variable for two arguments, presented in the description below.

The ***y_pointers*** values you supply to the **block_sparse_setup** routine should refer to the original locations of *y*. If you set *irandom* = 1, **block_sparse_setup** modifies the values of ***y_pointers*** so that the scattering operation occurs correctly. Do not modify the contents of ***y_pointers*** between the setup call and the associated multiplication call(s). When you call the multiplication routines, you must supply the values that **block_sparse_setup** assigned to ***y_pointers***.

m — Scalar integer variable. The true extent of the first axis of *A*. Also the true extent of the first axis of ***y_pointers*** (for operations in which *A* is the left-hand operand) or of ***x_pointers*** (for operations in which *A* is the right-hand operand).

n — Scalar integer variable. The true extent of the second axis of *A*. Also the true extent of the first axis of ***x_pointers*** (for operations in which *A* is the left-hand operand) or of ***y_pointers*** (for operations in which *A* is the right-hand operand).

p — Scalar integer variable. The true extent of the third axis of *A*, and the true extent of the second axis of ***x_pointers*** and ***y_pointers***.

x_length — Scalar integer variable. The true extent of the first axis of the source array ***x***. Prior to permuting the data elements of ***x***, you must arrange your data so that the first *x_length* locations (for rank 1) or rows (for rank 2) of ***x*** contain the elements to be gathered.

y_length Scalar integer variable. The true extent of the first axis of the destination array *y*. The multiplication routines assume that the original, unpermuted locations of *y* that are to receive scattered product elements are first *y_length* locations (for rank 1) or first *y_length* rows (for rank 2).

irandom Scalar integer variable. Must contain 0 or 1. Setting *irandom* to 1 causes the setup routine to return random permutations of the source and destination array element locations. If *irandom* is 0, identity permutations are returned.

itrace Scalar integer variable. Must contain 0 or 1. Set *itrace* to 1 to calculate and save an optimization, or *trace*, for the communication pattern corresponding to the contents of ***x_pointers*** and ***y_pointers***. Set *itrace* to 0 to have the multiplication routine calculate the trace.

trace Scalar integer variable. Internal variable. The initial value you supply when you call **block_sparse_setup** is ignored. Upon return from **block_sparse_setup**, *trace* contains a value that you must supply when you make associated calls to the multiplication routines and to **deallocate_block_sparse_setup**.

trace_mask Scalar integer variable. Internal variable. The initial value you supply when you call **block_sparse_setup** is ignored. Upon return from **block_sparse_setup**, *trace_mask* contains a value that you must supply when you make associated calls to the multiplication routines and to **deallocate_block_sparse_setup**.

setup Scalar integer variable. Internal variable. The initial value you supply when you call **block_sparse_setup** is ignored. Upon return from **block_sparse_setup**, *setup* contains a value that you must supply when you make associated calls to the multiplication routines.

ier Scalar integer variable. Upon return from **block_sparse_setup**, contains one of the following codes:

0	Successful return.
-1	The supplied arguments had mismatched shapes or did not follow the rules for using the same variable for two arguments, presented in the description below.

Upon return from one of the multiplication routines, contains one of the following codes:

0	Successful return.
-1	The supplied arguments had mismatched shapes or did not follow the rules for using the same variable for two arguments, presented in the description below.

DESCRIPTION

Setup and Deallocation. Follow these steps to perform one multiplication operation (or multiple operations, sequentially):

1. Call **block_sparse_setup.**

2. Call one or more of the multiplication routines listed below. (In the formulas below, x and y denote vectors while X and Y denote matrices. However, lowercase letters are used for both cases everywhere else in this text.)

   block_sparse_matrix_vector_mult $y = Ax$
   block sparse matrix x vector

   vector_block_sparse_matrix_mult $y^T = x^T A$
   vector x block sparse matrix

   block_sparse_mat_gen_mat_mult $Y = AX$
   block sparse matrix x dense matrix

   gen_mat_block_sparse_mat_mult $Y^T = X^T A$
   dense matrix x block sparse matrix

   To compute more than one product using the same gathering and scattering pointer arrays, follow one call to **block_sparse_setup** with multiple calls to the multiplication routines. If the pointer arrays change, start with Step 1 again.

3. After all multiplication calls associated with the same **block_sparse_setup** call, call **deallocate_block_sparse_setup** to deallocate the storage space required by the setup routine.

More than one setup may be active at a time. That is, you may call the setup routine more than once without calling the deallocation routine.

Setup Phase. Given the sparsity of the block sparse matrix *A*, the gathering and scattering pointers (***x_pointers*** and ***y_pointers***) to be used in subsequent multiplications, and the shapes of the source array *x* and destination array *y*, **block_sparse_setup** initializes masks and location vectors for *x* and *y* and assigns appropriate values to internal variables required by the multiplication routines. The setup routine modifies the contents of ***x_pointers*** and ***y_pointers***.

The setup routine provides two options that may improve performance significantly:

- If you set *itrace* = 1, **block_sparse_setup** saves the trace associated with ***x_pointers*** and ***y_pointers*** for use in subsequent calls to the multiplication routines. The setup routine also allocates the additional storage space required for the trace.
- If you set *irandom* = 1, **block_sparse_setup** returns random permutations of the source and destination array element locations in ***where_is_x*** and ***where_is_y***, respectively, and returns the new masks for the source and destination arrays in ***x_mask*** and ***y_mask***, respectively. (If the source and destination arrays are the same variable, the same permutation is applied to both arrays.) You must apply the permutation indicated in ***where_is_x*** to the source arrays you supply in subsequent multiplication calls. The permutation indicated in ***where_is_y*** is applied to the destination array by the multiplication routine.

Multiplication Phase. Given a sparse matrix, *A*, represented in block form, and source and destination arrays *x* and *y*, respectively, the multiplication routines compute the product of *A* with *x* and return the product in *y*.

Deallocation Phase. The **deallocate_block_sparse_setup** routine deallocates the storage space that was allocated for a trace in a previous call to **block_sparse_setup**. Each **block_sparse_setup** call in which *itrace* = 1 should be followed (after one or more associated multiplication routine calls) by a **deallocate_block_sparse_setup** call. In fact, it is good practice to issue a call to the **deallocate_block_sparse_setup** routine for *every* call to **block_sparse_setup**. (If *itrace* was set to 0 in the **block_sparse_setup** call, **deallocate_block_sparse_setup** has no effect.)

Using the Same Variable for Two Arguments. Several possibilities exist for using the same variable for two arguments. However, the current release supports the following two cases only:

- Each argument uses a different variable.

- ***x_mask*** and ***y_mask*** are the same variable. In this case, ***where_is_x*** and ***where_is_y*** must be the same variable, and ***x_pointers*** and ***y_pointers*** must be the same variable. If you do not follow this rule, an error code (*ier* = -1) is returned.

NOTES

Argument Values. The internal variables *trace*, *trace_mask*, and *setup* are required for communicating information between the setup phase and the multiplication phase. The application must not modify the contents of these variables. Similarly, after a call to the setup routine, the application should not modify the contents of pointers ***x_pointers*** and ***y_pointers***.

The destination array *y* is the only argument updated by the multiplication routines.

Use of Setup Routine. The setup routine must be called whenever the sparsity of the sparse system, represented by pointer arrays ***x_pointers*** and ***y_pointers***, changes. For performance reasons, the cost of the setup phase should be amortized over several multiplications.

Use of Deallocation Routine. If *itrace* was set to 1 in the **block_sparse_setup** call, be sure to call **deallocate_block_sparse_setup** to deallocate storage space after all of the block sparse matrix operations associated with the setup call have finished.

Numerical Stability. The block sparse matrix operations are stable.

Numerical Complexity. Each block sparse matrix operation requires approximately $2mnp$ floating-point operations for real operands, or $8mnp$ floating-point operations for complex operands.

Performance Hints. Performance is best when the blocks are local to a processing element. You may meet this condition by using the **:serial** layout directive on axes 1 and 2, by using a very high weight on these axes, or by using the detailed layout axis descriptors, **:procs** and **:blocks**. Typical examples are as follows:

```
CMF$LAYOUT  A(:SERIAL, :SERIAL, :NEWS)
CMF$LAYOUT SRC_POINTERS(:SERIAL, :NEWS)
CMF$LAYOUT DEST_POINTERS(:SERIAL, :NEWS)
    REAL A(24, 24, 16000)
    INTEGER SRC_POINTERS(24, 16000)
    INTEGER DEST_POINTERS(24, 16000)
```

```
CMF$LAYOUT  A(:SERIAL, 100000:NEWS, :NEWS)
CMF$LAYOUT SRC_POINTERS(100000:NEWS, :NEWS)
CMF$LAYOUT DEST_POINTERS(100000:NEWS, :NEWS)
    REAL A(81, 81, 8000)
    INTEGER SRC_POINTERS(81, 8000)
    INTEGER DEST_POINTERS(81, 8000)
```

EXAMPLES

Sample CM Fortran code that uses the block sparse matrix operations can be found on-line in the subdirectory

block-sparse/cmf/

of a CMSSL examples directory whose location is site-specific.

4.4 Grid Sparse Matrix Operations

This section introduces the grid sparse matrix operations. For detailed information about the routines and their arguments, refer to the man page at the end of this section.

4.4.1 The Grid Sparse Matrix Routines

Given coefficient arrays, an operand array, and a product array on a 1-, 2-, or 3-dimensional grid, the grid sparse matrix routines compute the product of the grid sparse matrix represented by the coefficient arrays with the vector or dense matrix represented by the operand array. The following routines are provided:

grid_sparse_setup	Sets up the multiplication operation and allocates the necessary partition manager workspace.
grid_sparse_matrix_vector_mult	Multiplies a grid sparse matrix by a vector.
vector_grid_sparse_matrix_mult	Multiplies a vector by a grid sparse matrix.
grid_sparse_mat_gen_mat_mult	Multiplies a grid sparse matrix by a dense matrix.
gen_mat_grid_sparse_mat_mult	Multiplies a dense matrix by a grid sparse matrix.
deallocate_grid_sparse_setup	Deallocates the partition manager workspace allocated by **grid_sparse_setup**.

For information about setup and deallocation, refer to the Description section of the man page at the end of this section.

4.4.2 Grid Sparse Matrix Representation

The grid sparse matrix routines operate on data that is arranged on a grid. Coefficient matrix elements residing at each grid point *P* are multiplied by vector or

matrix elements residing at point *P* and its nearest-neighbor points. The result is placed in product vector or matrix elements residing at point *P*. This section describes these grid operations in detail. Section 4.4.3 describes the matrix representation of these operations. The matrix representation is provided for informational purposes only; applications must use the grid representation described in this section.

Grid Representation

The grid sparse matrix routines assume that you are working with the arrays listed below, and that these arrays are arranged in a 1-, 2-, or 3-dimensional grid.

- Three, five, or seven *coefficent arrays*:
  - Three arrays, ***a***, ***b***, and ***c***, if the grid is 1-dimensional.
  - Five arrays, ***a***, ***b***, ***c***, ***d***, and ***e***, if the grid is 2-dimensional.
  - Seven arrays, ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, and ***g***, if the grid is 3-dimensional.
- An *operand array*, ***x***.
- A *product array*, ***y***.

Each grid point is associated with either an *element* or a dense *block* of each array.

For example, Figure 6 shows a 1-dimensional grid with 7 points. Each grid point is associated with one element of each array. In this example, ***a***, ***b***, ***c***, ***x***, and ***y*** all have rank 1.

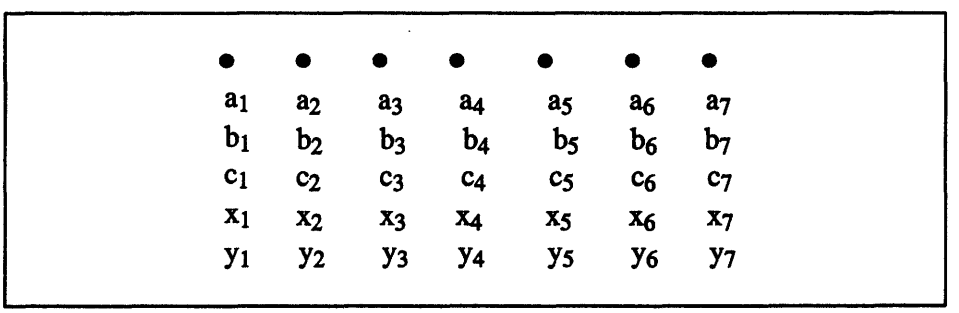

Figure 6. 1-dimensional grid; each grid point is associated with one element of each array.

Figure 7 shows a 2-dimensional grid with dimensions (3 x 3). Again, each grid point is associated with one element of each array. In this example, ***a, b, c, d, e,*** ***x,*** and ***y*** all have rank 2. The axes of each array correspond to the axes of the grid.

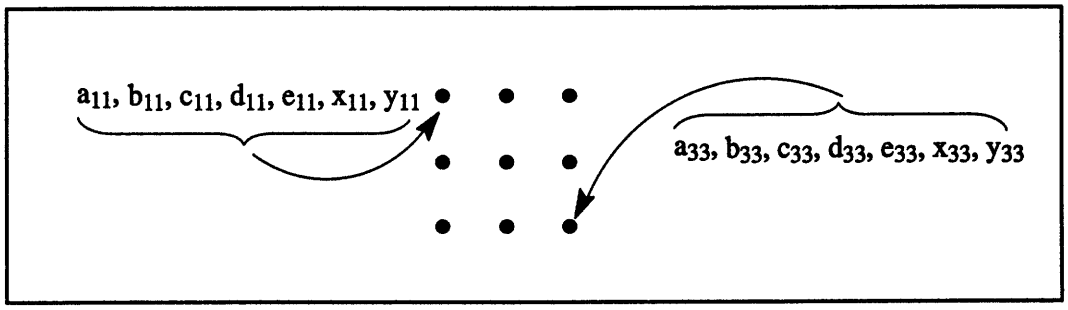

Figure 7. 2-dimensional grid; each grid point is associated with one element of each array.

In contrast, Figure 8 shows a 1-dimensional, 4-point grid, each point of which is associated with a dense *block* of each array. Specifically, each grid point is associated with a (2 x 2) block of each of the coefficient arrays ***a, b,*** and ***c***; a length-2 block of ***x***; and a length-2 block of ***y***. In this example, the arrays ***a, b,*** and ***c*** have rank 3; the first two axes define the block and the third axis corresponds to the grid. The arrays ***x*** and ***y*** have rank 2; the first axis defines the block (which is a vector) and the second corresponds to the grid. In CM Fortran, you would declare these arrays as ***a***(2, 2, 4), ***b***(2, 2, 4), ***c***(2, 2, 4), ***x***(2, 4), and ***y***(2, 4).

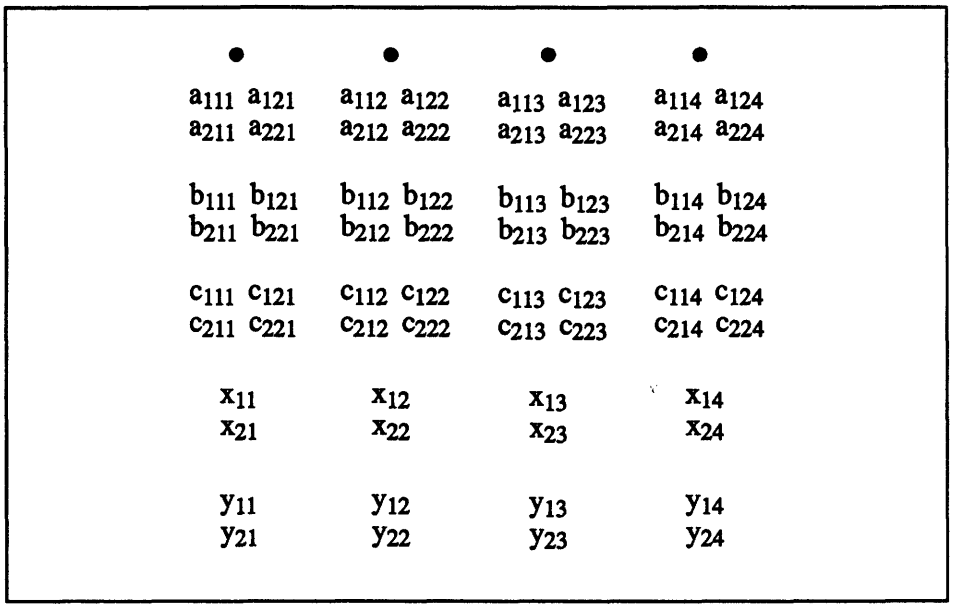

Figure 8. 1-dimensional grid; 2-dimensional coefficient blocks, 1-dimensional operand and product blocks.

Finally, Figure 9 shows the same 1-dimensional grid of length 4, but this time each grid point is associated with a (2 x 2) block of each of the coefficient arrays *a*, *b*, and *c*; a (2 x 2) block of *x*; and a (2 x 2) block of *y*. In this example, the arrays *a*, *b*, *c*, *x*, and *y* all have rank 3. In CM Fortran, you would declare the arrays as follows:

```
real, array (2,2,4): a, b, c, x, y
```

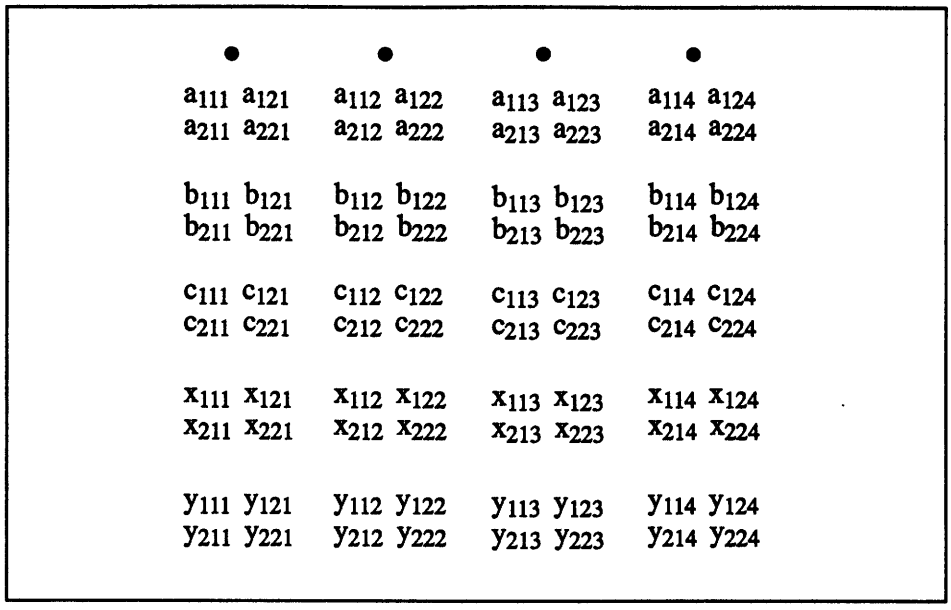

Figure 9. 1-dimensional grid; 2-dimensional coefficient, operand and product blocks.

Grid Axes, Block Axes, and Instance Axes

As the examples in Figure 6 through Figure 9 illustrate, each of the arrays ***a***, ***b***, ***c***[, ***d***, ***e***[, ***f***, ***g***]], ***x***, and ***y*** has one, two, or three axes corresponding to the grid. These axes are called the *grid axes*.

In addition, each of the arrays may have one or two axes that define the dense block associated with each grid point. These axes are called the *block axes*.

Finally, each array may have multiple instances, defined by any number of *instance axes*. (For a discussion of instance axes, refer to Chapter 1.)

Grid Multiplication

The grid sparse matrix routines multiply the elements of the arrays ***a***, ***b***, ***c***[, ***d***, ***e***[, ***f***, ***g***]] by the elements of ***x*** and place the results in ***y***. To compute this product, the routines multiply the elements or blocks of ***a***, ***b***, ***c***[, ***d***, ***e***[, ***f***, ***g***]] at a given grid point by the elements of ***x*** that reside at the same grid point and its nearest-neighbor grid points.

Figure 10 illustrates this process for a 1-dimensional grid. Each point (except the boundary points) has two nearest neighbors. At each point *P*,

- The block of ***a*** is multiplied by the block of ***x*** at point *P*–1.
- The block of ***b*** is multiplied by the block of ***x*** at point *P*.
- The block of ***c*** is multiplied by the block of ***x*** at point *P*+1.

The sum of the results is placed in the block of ***y*** at point *P*. The boundary conditions are under user control. For example, in Figure 10, elements a_{111}, a_{121}, a_{211}, a_{221}, c_{114}, c_{124}, c_{214}, and c_{224} would normally be 0; but you may wish to supply other values, depending on your application.

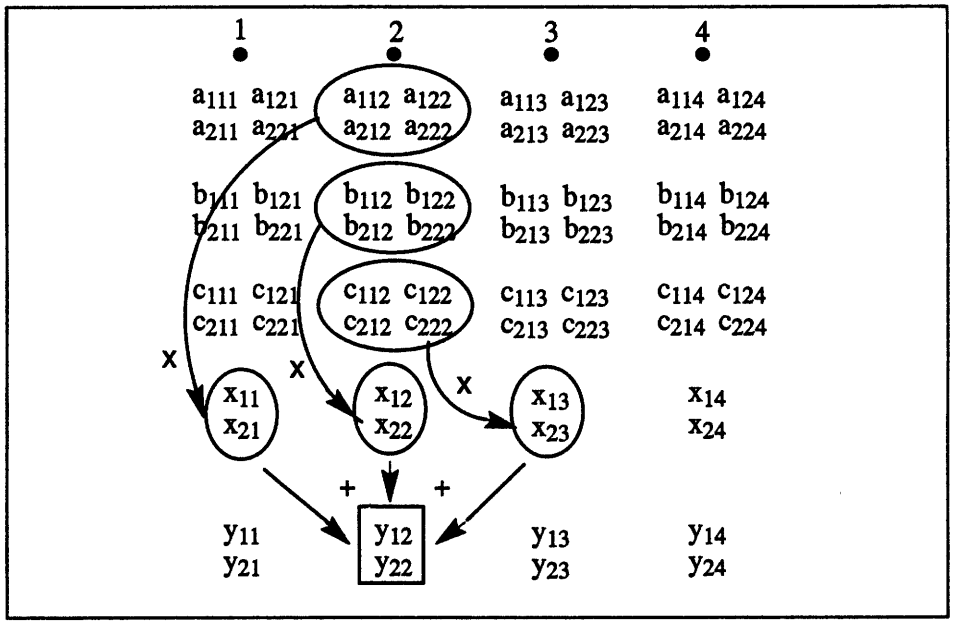

Figure 10. Grid multiplication at a non-boundary point.

Functionally, the routines perform a circular shift (**CSHIFT** in CM Fortran) of the array ***x***. Thus, the high boundary point takes the place of the missing nearest neighbor to the low boundary point, as shown in Figure 11.

Version 3.1, June 1993

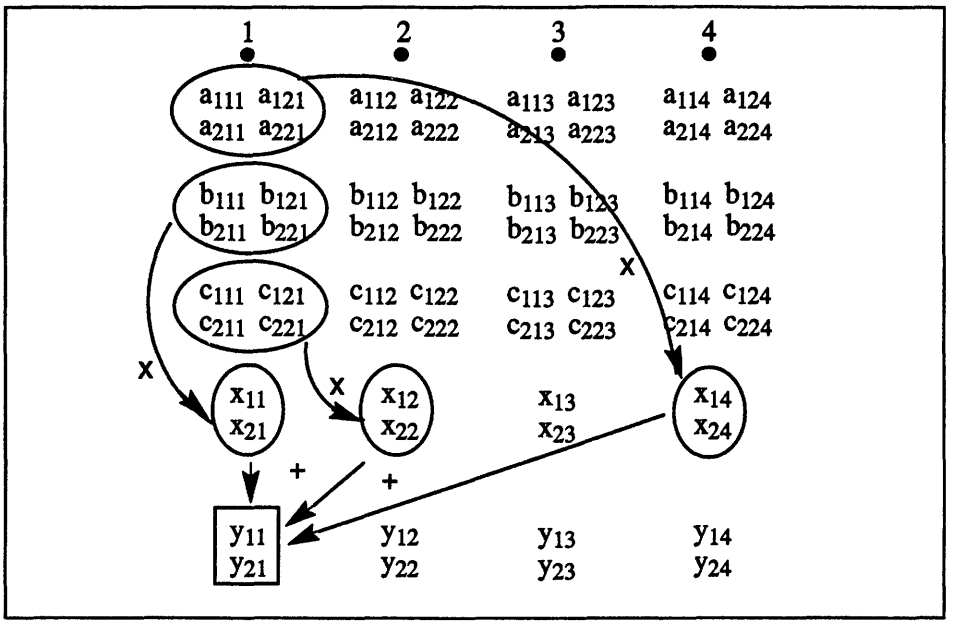

Figure 11. Grid multiplication at a boundary point.

In a 2-dimensional grid, each non-boundary point has four nearest neighbors. An element of ***c*** is multiplied by the element of ***x*** at the same point; elements of ***a, b, d,*** and ***e*** are multiplied by the elements of ***x*** at the nearest neighbors.

Similarly, in a 3-dimensional grid, each non-boundary point has six nearest neighbors. An element of ***d*** is multiplied by the element of ***x*** at the same point; elements of ***a, b, c, e,*** and ***f*** are multiplied by the elements of ***x*** at the nearest neighbors.

The man page at the end of this section includes formulas for grid sparse matrix multiplication; rules for grid, block, and instance axes; and performance hints.

4.4.3 Matrix Representation of the Grid Sparse Matrix Operations

This section describes the grid sparse matrix operations in terms of matrices. This matrix representation is provided for informational purposes only; applications must use the grid representation described in Section 4.4.2.

A grid sparse matrix is a sparse matrix that represents data originating on a grid. The non-zero elements of the matrix contain the data from the grid; the *positions*

of the non-zero elements reflect the numbering scheme used to order the points of the grid.

When the grid multiplication described above is represented in matrix form, the elements of the arrays ***a***, ***b***, ***c***[, ***d***, ***e***[, ***f***, ***g***]] form a grid sparse matrix A, and the elements of x form a vector or matrix. The routines compute the product Ax or x^TA and place the results in the vector or matrix y.

The elements of ***a***, ***b***, ***c***[, ***d***, ***e***[, ***f***, ***g***]] associated with one grid point appear on the same row of A; the positions of these non-zero elements in the row ensure that each element is multiplied by the correct element of x.

The exact locations of non-zero elements in the grid sparse matrix depend on the numbering scheme used to label the points of the original grid. Typical labeling schemes may result in the patterns shown in Figure 12 (tridiagonal, 5-diagonal, and 7-diagonal matrices for 1-, 2-, and 3-dimensional grids, respectively). The dots indicate the positions of non-zero elements or dense blocks.

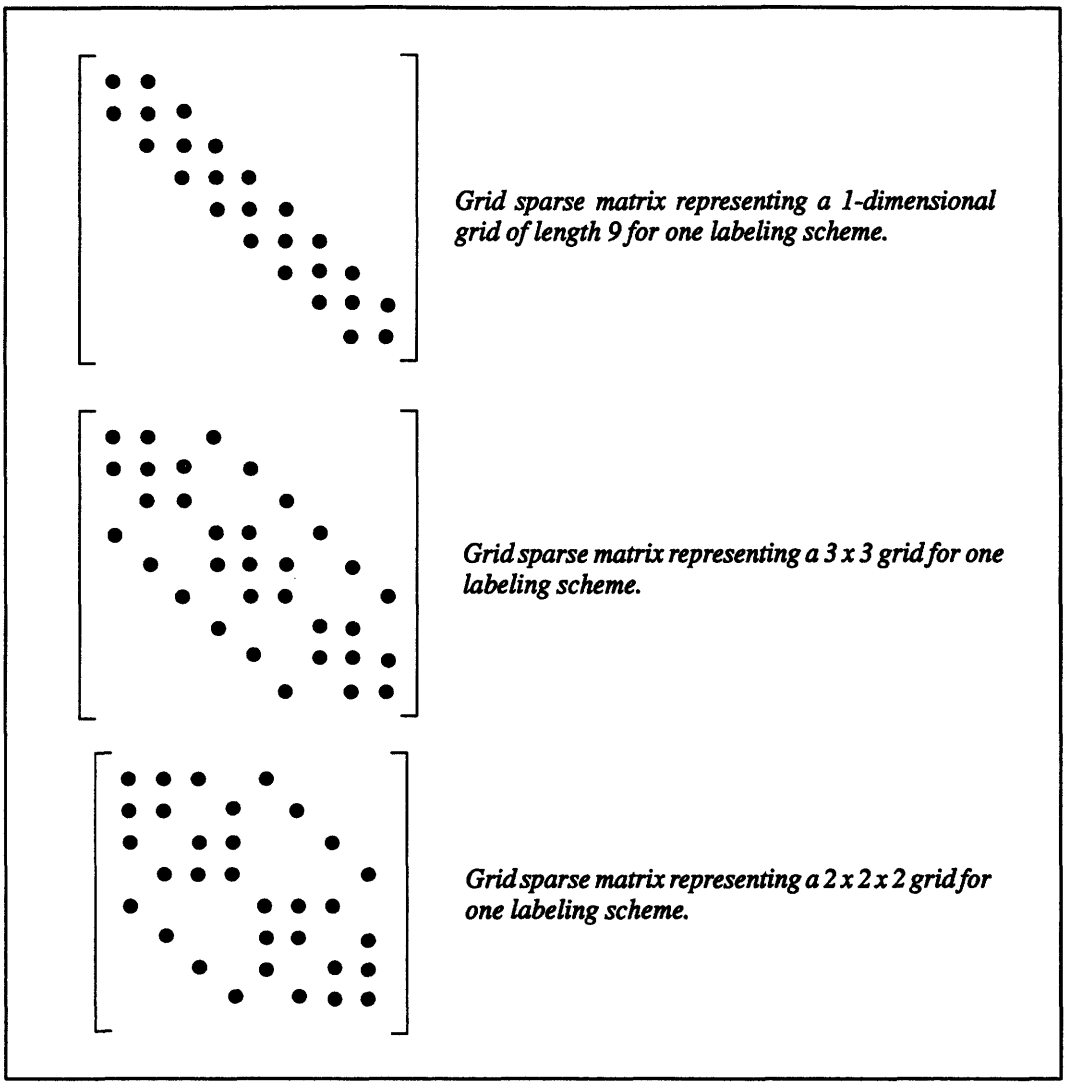

Figure 12. Matrix forms for 1-, 2-, and 3-dimensional grids using a common numbering scheme. Dots indicate the positions of non-zero elements or dense blocks.

For grid sparse matrix representations with the labeling schemes shown in Figure 12, each coefficient array *a, b, c*[, *d, e*[, *f, g*]] consists of the elements of a diagonal, as shown in Figure 13.

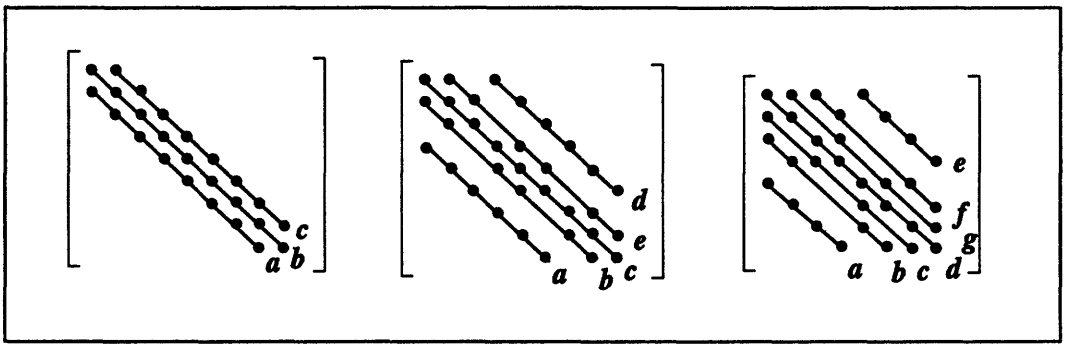

Figure 13. Coefficient array representation of common forms of grid sparse matrices.

Note that some of the elements of *a, b, c*[, *d, e*[, *f, g*]] associated with boundary points on the grid do not appear in these matrices. The arrays you supply must include these boundary elements; be sure to set the boundary values appropriately for your application.

Figure 14 shows the matrix representations of the grid sparse matrix operations.

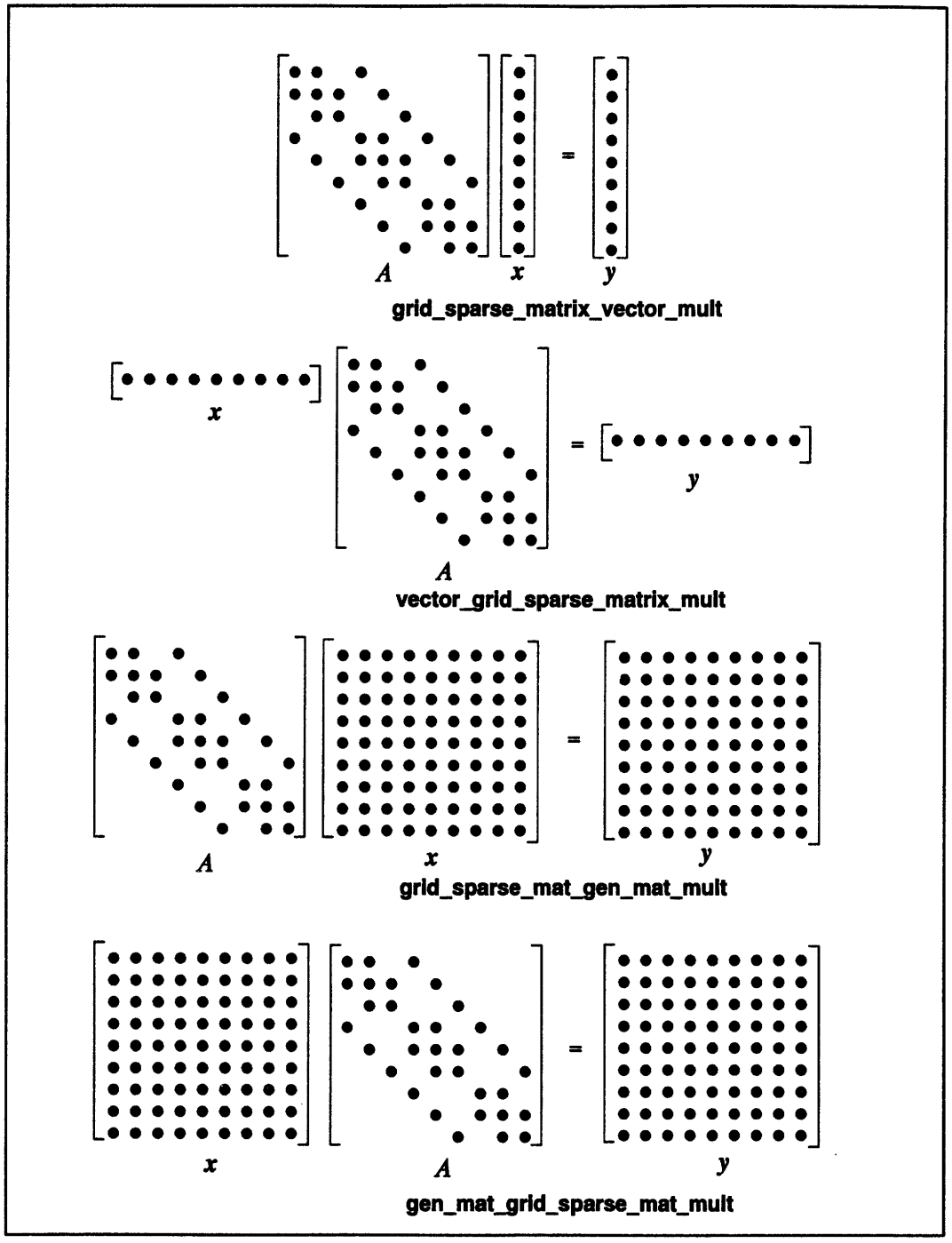

Figure 14. The grid sparse matrix operations in matrix form.

The **grid_sparse_matrix_vector_mult** and **vector_grid_sparse_matrix_mult** routines support the following possibilities:

- Each dot of the grid sparse matrix is a single element; each dot of the vector is a single element.
- Each dot of the grid sparse matrix is a p x q block; each dot of the vector is a vector of length q (for **grid_sparse_matrix_vector_mult**) or length p (for **vector_grid_sparse_matrix_mult**).

The **grid_sparse_mat_gen_mat_mult** and **gen_mat_grid_sparse_mat_mult** routines support the following possibility:

- Each dot of the grid sparse matrix is a p x q block. For **grid_sparse_mat_gen_mat_ mult**, each dot of the dense matrix x is a q x r block and each dot of the dense matrix y is an p x r block. For **gen_mat_grid_sparse_mat_mult**, each dot of the dense matrix x is an r x p block and each dot of the dense matrix y is an r x q block.

Each operation can occur in multiple instances.

Grid Sparse Matrix Operations

Given coefficient arrays, an operand array, and a product array on a 1-, 2-, or 3-dimensional grid, the routines described below compute the product of the grid sparse matrix represented by the coefficient arrays with the vector or dense matrix represented by the operand array.

SYNTAX

grid_sparse_setup (*x, x_axes, setup, ier*)

grid_sparse_matrix_vector_mult (*ier, setup, y_axes, coeff_axes, x_axes, y, x, a, b, c*[, *d, e*[, *f, g*]])

vector_grid_sparse_matrix_mult (*ier, setup, y_axes, coeff_axes, x_axes, y, x, a, b, c*[, *d, e*[, *f, g*]])

grid_sparse_mat_gen_mat_mult (*ier, setup, coeff_axes, y, x, a, b, c*[, *d, e*[, *f, g*]])

gen_mat_grid_sparse_mat_mult (*ier, setup, coeff_axes, y, x, a, b, c*[, *d, e*[, *f, g*]])

deallocate_grid_sparse_setup (*setup*)

ARGUMENTS

x — CM array of type real or complex. Represents the operand vector or dense matrix in the operation $y = Ax$ or $y = x^T A$, where A is a grid sparse matrix represented by the coefficient arrays *a, b, c*[, *d, e*[, *f, g*]].

x_axes — Front-end integer vector. The length of *x_axes* must be equal to the rank of *x*. Each element of *x_axes* is one of the following symbolic constants, describing the corresponding axis of *x*:

CMSSL_block_axis
CMSSL_grid_axis
CMSSL_instance_axis

y_axes Front-end integer vector. The length of *y_axes* must be equal to the rank of *y*. Each element of *y_axes* is one of the following symbolic constants, describing the corresponding axis of *y*:

CMSSL_block_axis
CMSSL_grid_axis
CMSSL_instance_axis

coeff_axes Front-end integer vector. The length of *coeff_axes* must be equal to the rank of the arrays ***a, b, c***[, ***d, e***[, ***f, g***]]. Each element of *coeff_axes* is one of the following symbolic constants, describing the corresponding axis of ***a, b, c***[, ***d, e***[, ***f, g***]]:

CMSSL_block_axis
CMSSL_grid_axis
CMSSL_instance_axis

y CM array of the same data type and precision as *x*. Represents the product vector or dense matrix in the operation $y = Ax$ or $y = x^T A$, where *A* is a grid sparse matrix represented by the coefficient arrays ***a, b, c***[, ***d, e***[, ***f, g***]].

a, b, c[, ***d, e***[, ***f, g***]] Coefficient CM arrays of the same data type and precision as *x*. Must have the same rank, axis extents, and layout directives. See description below.

setup Scalar integer. Internal variable. When you call a multiplication routine or the deallocation routine, you must supply the *setup* value assigned by the associated setup call.

ier Scalar integer. Return code; set to 0 upon successful return, or to -1 if any of the restrictions on axis labels are violated.

DESCRIPTION

Follow these steps to perform one multiplication operation, or multiple operations, sequentially:

1. Call **grid_sparse_setup.**
2. Call one or more of the following multiplication routines:
   - **grid_sparse_matrix_vector_mult** (grid sparse matrix X vector)
   - **vector_grid_sparse_matrix_mult** (vector X grid sparse matrix)

- **grid_sparse_mat_gen_mat_mult** (grid sparse matrix x dense matrix)
- **gen_mat_grid_sparse_mat_mult** (dense matrix x grid sparse matrix)

To compute more than one product in which the shape, axis types, and axis declaration order of the operand vectors (or dense matrices) remain the same, follow one call to **grid_sparse_setup** with multiple calls to the multiplication routines. If the shape, axis types, or axis declaration order of the operand vector (or dense matrix) changes, you must start with Step 1 again.

3. After all multiplication calls associated with the same **grid_sparse_setup** call, call **deallocate_block_sparse_setup** to deallocate the partition manager work space allocated by the setup routine.

More than one setup may be active at a time. That is, you may call the setup routine more than once without calling the deallocation routine.

The grid sparse matrix routines support multiple instances. By specifying instance axes, you may perform multiple concurrent operations with each multiplication call.

Axis Types. Each of the arrays *a, b, c*[, *d, e*[, *f, g*]], *x*, and *y* has one, two, or three axes corresponding to the grid. These axes are called the *grid axes.*

In addition, each of the arrays may have one or two axes that define the block associated with each grid point. These axes are called the *block axes.*

Finally, each array may have multiple instances, defined by any number of *instance axes.* (For a discussion of instance axes, refer to Chapter 1.)

Grid Multiplication. The grid sparse matrix routines multiply the elements of the arrays *a, b, c*[, *d, e*[, *f, g*]] by the elements of *x* and place the results in *y*. To compute this product, the routines multiply the elements or blocks of *a, b, c*[, *d, e*[, *f, g*]] at a given grid point by the elements of *x* that reside at the same grid point and its nearest-neighbor grid points. The boundary conditions are under user control. Functionally, the routines perform a circular shift (**CSHIFT** in CM Fortran) of the array *x*. Thus, the high boundary point takes the place of the missing nearest neighbor to the low boundary point.

The formulas for grid sparse matrix multiplication are shown below. In these formulas, *a1, a2,* and *a3* are the grid axes of *x*.

For **grid_sparse_matrix_vector_mult** and **grid_sparse_mat_gen_mat_mult:**

⊖ is one of the following operations, performed with respect to the block axes:

- multiplication of single elements
- matrix vector multiplication
- matrix matrix multiplication

On a 1-dimensional grid (one grid axis labeled a_1):

$$y = a \ominus \mathbf{CSHIFT}(x, \text{shift} = -1, \text{dim} = a1) + b \ominus x + c \ominus \mathbf{CSHIFT}(x, \text{shift} = +1, \text{dim} = a1)$$

On a 2-dimensional grid (two grid axes labeled a_1 and a_2):

$$y = a \ominus \mathbf{CSHIFT}(x, \text{shift} = -1, \text{dim} = a1) + b \ominus \mathbf{CSHIFT}(x, \text{shift} = -1, \text{dim} = a2) + c \ominus x + d \ominus \mathbf{CSHIFT}(x, \text{shift} = +1, \text{dim} = a1) + e \ominus \mathbf{CSHIFT}(x, \text{shift} = +1, \text{dim} = a2)$$

On a 3-dimensional grid (three grid axes labeled a_1, a_2, and a_3):

$$y = a \ominus \mathbf{CSHIFT}(x, \text{shift} = -1, \text{dim} = a1) + b \ominus \mathbf{CSHIFT}(x, \text{shift} = -1, \text{dim} = a2) + c \ominus \mathbf{CSHIFT}(x, \text{shift} = -1, \text{dim} = a3) + d \ominus x + e \ominus \mathbf{CSHIFT}(x, \text{shift} = +1, \text{dim} = a1) + f \ominus \mathbf{CSHIFT}(x, \text{shift} = +1, \text{dim} = a2) + g \ominus \mathbf{CSHIFT}(x, \text{shift} = +1, \text{dim} = a3)$$

For **vector_grid_sparse_matrix_mult** and **gen_mat_grid_sparse_mat_mult**:

$\ominus$ is one of the following operations, performed with respect to the block axes:

- multiplication of single elements
- vector matrix multiplication
- matrix matrix multiplication

On a 1-dimensional grid (one grid axis labeled a_1):

$$y = \mathbf{CSHIFT}(a \ominus x, \text{shift} = +1, \text{dim} = a1) + b \ominus x + \mathbf{CSHIFT}(c \ominus x, \text{shift} = -1, \text{dim} = a1)$$

On a 2-dimensional grid (two grid axes labeled a_1 and a_2):

```
y = CSHIFT (a ⊖ x, shift  = +1, dim = a1) +
    CSHIFT (b ⊖ x, shift  = +1, dim = a2) +
    c ⊖ x +
    CSHIFT (d ⊖ x, shift  = -1, dim = a1) +
    CSHIFT (e ⊖ x, shift  = -1, dim = a2)
```

On a 3-dimensional grid (three grid axes labeled a_1, a_2, and a_3):

```
y = CSHIFT (a ⊖ x, shift  = +1, dim = a1) +
    CSHIFT (b ⊖ x, shift  = +1, dim = a2) +
    CSHIFT (c ⊖ x, shift  = +1, dim = a3) +
    d ⊖ x +
    CSHIFT (e ⊖ x, shift  = -1, dim = a1) +
    CSHIFT (f ⊖ x, shift  = -1, dim = a2) +
    CSHIFT (g ⊖ x, shift  = -1, dim = a3)
```

Rules for Grid Axes, Block Axes, and Instance Axes. The grid sparse matrix routines impose the following requirements with regard to the structure of the arrays ***a, b, c*[, *d, e*[, *f, g*]], *x*,** and ***y***:

- If the grid is 1-dimensional with N points, then each array has one grid axis of extent N.
- If the grid is 2-dimensional with N_1 x N_2 points, then each array has two grid axes of extents N_1 and N_2, respectively.
- If the grid is 3-dimensional with N_1 x N_2 x N_3 points, then each array has three grid axes of extents N_1, N_2, and N_3, respectively.
- The valid combinations of block axes are as follows:
  - For **grid_sparse_matrix_vector_mult** or **vector_grid_sparse_matrix_mult**:

	Number of block axes
***a, b, c*[, *d, e*[, *f, g*]]**	0 or 2
x	0 or 1
y	0 or 1

The first combination (number of block axes = 0) is for multiplication of single elements. The second combination (number of block axes = 2 for coefficients matrix and 1 for vectors) is for matrix vector or vector matrix multiplication.

- For **grid_sparse_mat_gen_mat_mult** or **gen_mat_grid_sparse_mat_mult:**

	Number of block axes
a, b, c[, *d, e*[, *f, g*]]	2
x	2
y	2

This combination is for matrix matrix multiplication.

- The grid axes, block axes, and instance axes can occur in any order (which you specify when calling the routines), with the condition that the grid, block, and instance axes must occur in the same order in the arrays *a, b, c*[, *d, e*[, *f, g*]]. That is, the arrays *a, b, c*[, *d, e*[, *f, g*]] must all have the same shape (axis declaration order and extents). They must also have the same layout directives.

NOTES

Include the CMSSL Header File. The grid sparse matrix routines use symbolic constants. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL functions and symbolic constants.

Argument Values. The internal variable *setup* is required for communicating information between the setup phase and the multiplication phase. The application must not modify the contents of this variable.

Use of Setup Routine. The setup routine must be called whenever the shape, axis types, or axis declaration order of *x* changes. For performance reasons, the cost of the setup phase should be amortized over several multiplications.

Use of Deallocation Routine. Be sure to call **deallocate_grid_sparse_setup** to deallocate work space after all of the grid sparse matrix operations associated with the setup call have finished.

Numerical Stability. The grid sparse matrix operations are stable.

Numerical Complexity. If I is the number of instances (the product of the extents of the instance axes), N is the product of the extents of the grid axes, and each block of the coefficient arrays has axis extents p x q, then the **grid_sparse_matrix_vector_mult** and **vector_grid_sparse_matrix_mult** operations require $2pqNI$ floating-point operations for real operands, or $8pqNI$ floating-point operations for complex operands.

If I is the number of instances, N is the product of the extents of the grid axes, each block of the coefficient arrays has axis extents p x q, each block of x is q x r, and each block of y is p x r, then the **grid_sparse_mat_gen_mat_mult** and **gen_mat_grid_sparse_mat_mult** operations require $2pqrNI$ floating-point operations for real operands, or $8pqrNI$ floating-point operations for complex operands.

Performance Hints. Performance is strongly dependent on layout, and is best when the axes representing the vectors and matrices are local to a processing element. You may meet this condition by using the **:serial** layout directive or using the detailed axis descriptors of the CM Fortran **CMF$LAYOUT** directive. Otherwise, the routines reshape the arrays, incurring a performance cost.

EXAMPLES

Sample CM Fortran code that uses the grid sparse matrix operations can be found on-line in the subdirectory

```
grid-sparse/cmf/
```

of a CMSSL examples directory whose location is site-specific.

4.5 References

The following references provide further information about sparse matrix operations:

1. Blelloch, G. *Vector Models for Data Parallel Computing*. Cambridge, MA: MIT Press, 1990.

2. Duff, I. S., et al. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1986.

3. George, A., and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

4. Mathur, K. K. *On the Use of Randomized Address Maps in Unstructured Three-Dimensional Finite Element Simulation.* Thinking Machines Corporation Technical Report CS90-4, 1990.

5. McKenna, M., and Mesirov, J. Personal communication, 1991.

6. Ranade, A. How to Emulate Shared Memory. *Proc. 28th Annual Symp. Theory of Comp.* ACM (1981): 263-77.

7. Ranade, A., et al. The Fluent Abstract Machine. *Advanced Research in VLSI*. MIT Press, 1988. Pp. 71-94.

8. Valiant, L. G. A Scheme for Fast Parallel Communication. *SIAM Journal of Computing* **11** (1982): 350-61.

9. Young, D. and D. Kincaid. The ITPACK Package for Large Sparse Linear Systems. In *Elliptic Problem Solvers,* ed. M. Schkultz. New York: Academic Press, 1981. Pp. 163-85.

Chapter 5

Linear Solvers for Dense Systems

This chapter describes the CM Fortran interface to the CMSSL general linear system solver routines. One section is devoted to each of the following topics:

- introduction
- Gaussian elimination (*LU* decomposition)
- solving linear systems using Householder transformations (*QR* decomposition)
- matrix inversion and the Gauss-Jordan solver
- Gaussian elimination with external storage
- *QR* factorization and least squares solution with external storage
- references

5.1 Introduction

Listed below are the CMSSL routines for solving dense linear systems. All routines accept either real or complex data.

- In-core solvers:
  - Routines that use Gaussian elimination (with or without pivoting) to decompose one or more matrices A into their LU factors; use those factors to solve the linear systems $AX = B$ or $A^TX = B$ (where B consists of one or more right-hand-side vectors); and perform related operations:

gen_lu_factor	**gen_lu_apply_l_inv_tra**
save_gen_lu	**gen_lu_apply_u_inv_tra**
restore_gen_lu	**gen_lu_get_l**
gen_lu_solve	**gen_lu_get_u**
gen_lu_solve_tra	**gen_lu_infinity_norm_inv**
gen_lu_apply_l_inv	**deallocate_gen_lu**
gen_lu_apply_u_inv	

  - Routines that use Householder transformations (with or without column pivoting) to decompose one or more matrices A into their QR factors; use those factors to solve the linear systems $AX = B$ or $A^TX = B$ (where B consists of one or more right-hand-side vectors); and perform related operations:

gen_qr_factor	**gen_qr_get_r**
save_gen_qr	**gen_qr_apply_p**
restore_gen_qr	**gen_qr_apply_p_inv**
gen_qr_solve	**gen_qr_zero_rows**
gen_qr_solve_tra	**gen_qr_extract_diag**
gen_qr_apply_q	**gen_qr_deposit_diag**
gen_qr_apply_q_tra	**gen_qr_infinity_norm_inv**
gen_qr_apply_r_inv	**gen_qr_r_infinity_norm_inv**
gen_qr_apply_r_inv_tra	**deallocate_gen_qr**

  - A routine, **gen_gj_invert**, that inverts a square matrix in place, using the Gauss-Jordan routine.
  - A routine, **gen_gj_solve**, that solves (with partial or total pivoting) a system of equations of the form $AX = B$ using a version of Gauss-Jordan elimination. B contains one or more right-hand-side vectors.

- External solvers:
  - Routines that use block Gaussian elimination with partial pivoting to decompose a matrix A (which is too large to fit into core memory) into its LU factors, and use those factors to solve the linear system $AX = B$:

    gen_lu_factor_ext
    gen_lu_solve_ext

  - Routines that use block Householder reflections to perform the factorization $A = QR$, where A is a matrix that is too large to fit into core memory, and use the QR factors to solve the linear system $AX = B$:

    gen_qr_factor_ext
    gen_qr_solve_ext

5.1.1 Embedding Coefficient Matrices within Larger Matrices

All of the in-core CMSSL general linear system solver routines allow you to embed the systems to be solved within larger matrices that have more rows and columns than the number of equations to be solved or the number of unknowns, respectively. You must specify, in the calling sequences, the axis lengths of the systems to be solved. For example, if the systems to be solved have dimensions m x n, with rows and columns counted by axes *row_axis* and *col_axis* respectively, you may declare axes *row_axis* and *col_axis* to have extents greater than m and n, respectively; but the routine will work only with the upper left-hand m x n elements of the matrix defined by *row_axis* and *col_axis*. The man pages for the individual solvers provide detailed information about axis extents.

5.1.2 Choosing an Algorithm

Use these guidelines when deciding whether to use the QR or LU routines, and whether to use pivoting:

- In most cases, the QR routines without pivoting or the LU routines with partial pivoting suffice. These two options are both stable for almost all practical purposes.

- For rank-deficient matrices, use the *QR* routines with pivoting. (Section 5.3.6 discusses working with ill-conditioned matrices.) It is unnecessary and wasteful of time to use the *QR* routines with pivoting for well-conditioned matrices.

- For matrices that are diagonally dominant, or where *LU* decomposition without pivoting is known to work, use the *LU* routines without pivoting. This option will not yield correct results if the matrix is ill-conditioned or if zeros appear on the diagonal during the elimination process.

5.2 Gaussian Elimination

Given a CM array A containing one or more instances of a dense matrix A, and a CM array B containing corresponding right-hand sides B, the CMSSL Gaussian elimination routines perform the following operations:

- Use Gaussian elimination, with or without partial pivoting, to factor each matrix A into two matrices, L and U. When pivoting is used, the effects of the pivoting are included in L.
- Use these LU factors to solve the system $AX = B$ or $A^TX = B$, where B consists of one or more right-hand-side vectors.
- Apply U^{-1}, $(U^{-1})^T$, L^{-1}, or $(L^{-1})^T$ to any supplied matrix. (These routines use the L and U factors to solve triangular systems of the form $LX=B$, $L^TX=B$, $UX=B$, and $U^TX=B$.)
- Produce L or U separately.
- Estimate the infinity norm of each matrix A^{-1}.
- Save and restore internal information about the LU factors.

The Gaussian elimination routines (commonly referred to as the "LU routines") are listed below. Throughout this section, the notation M^{-T} is used for $(M^{-1})^T = (M^T)^{-1}$. For detailed descriptions of these routines (including calling sequences, argument definitions, definitions of the L and U factors, and information about usage), refer to the man page at the end of this section.

gen_lu_factor	Factors each instance of a matrix A into L and U.
save_gen_lu	Saves internal information about the LU factors in a file.
restore_gen_lu	Restores internal information about the LU factors from a file.
gen_lu_solve	Uses the LU factors returned by **gen_lu_factor** to solve the system(s) $AX = B$.
gen_lu_solve_tra	Uses the LU factors returned by **gen_lu_factor** to solve the system(s) $A^TX = B$.
gen_lu_apply_l_inv	Given the LU factors returned by **gen_lu_factor**, applies L^{-1} to any supplied matrix or vector.

gen_lu_apply_u_inv Given the *LU* factors returned by **gen_lu_factor**, applies U^{-1} to any supplied matrix or vector.

gen_lu_apply_l_inv_tra Given the *LU* factors returned by **gen_lu_factor**, applies L^{-T} to any supplied matrix or vector.

gen_lu_apply_u_inv_tra Given the *LU* factors returned by **gen_lu_factor**, applies U^{-T} to any supplied matrix or vector.

gen_lu_get_l Given the *LU* factors returned by **gen_lu_factor**, produces the factor *L* separately.

gen_lu_get_u Given the *LU* factors returned by **gen_lu_factor**, produces the factor *U* separately.

gen_lu_infinity_norm_inv
Given the *LU* factors returned by **gen_lu_factor**, estimates the infinity norm of each matrix A^{-1}. Uses the method developed by Hager; see reference 5 listed in Section 5.7.)

deallocate_gen_lu Deallocates the processing element memory required by the above routines.

5.2.1 Blocking and Load Balancing

The *LU* routines use blocking and load balancing. These strategies are described in the section on computation of block cyclic permutations in Chapter 14. For details about how the *LU* routines implement blocking, see reference 4 listed in Section 5.7.

5.2.2 Numerical Stability

The stability of Gaussian elimination is a function of the size of the linear system and the growth factor (see references 1 and 3). For extreme cases, the growth factor may be very large. For most systems, it is highly unlikely that the growth factor for Gaussian elimination with partial pivoting will be very large, and for all practical purposes Gaussian elimination with partial pivoting is stable.

5.2.3 Saving and Restoring the *LU* State

The *LU* factorization routine generates internal state variables required for computing the solution. These variables are not made available as arrays to user applications because their sizes and contents are CM configuration-dependent. However, it is sometimes desirable to save the internal state to a file for future use. The **save_gen_lu** and **restore_gen_lu** routines allow you to save and restore the internal *LU* state.

The *LU* routines allow you to have more than one factorization "active" at a time; for example, the sequence of calls

```
setup_X = gen_lu_factor(X, ...)
setup_Y = gen_lu_factor(Y, ...)
call gen_lu_solve(B_X, X, setup_X, ...)
call gen_lu_solve(B_Y, Y, setup_Y, ...)
```

is valid. You may, however, want to use **save_gen_lu** and **restore_gen_lu** to carry the internal state over between program runs.

It is not intended that the save and restore routines be used to conserve memory. The state variables are very small compared to the size of the typical matrix *A*.

Gaussian Elimination

Given a CM array A containing one or more instances of a dense matrix A, and a CM array B containing corresponding right-hand sides B, the routines described below use Gaussian elimination (with or without partial pivoting) to factor each A into two matrices, L and U, described below; use the LU factors to solve the linear systems $AX = B$ or $A^TX = B$; apply matrices derived from L and U to each B; provide access to the L and U factors; and estimate the infinity norm of each A^{-1}. When pivoting is used, the effects of the pivoting are included in L.

SYNTAX

setup = **gen_lu_factor** (*A, m, n, row_axis, col_axis, nblock, pivoting_strategy, ier*)

save_gen_lu (*setup, unit, iostat, ier*)

setup = **restore_gen_lu** (*A, m, n, row_axis, col_axis, nblock, pivoting_strategy, unit, iostat, ier*)

gen_lu_solve (*B, A, setup, nrhs, ier*)

gen_lu_solve_tra (*B, A, setup, nrhs, ier*)

gen_lu_apply_l_inv (*B, A, setup, nrhs, ier*)

gen_lu_apply_u_inv (*B, A, setup, nrhs, ier*)

gen_lu_apply_l_inv_tra (*B, A, setup, nrhs, ier*)

gen_lu_apply_u_inv_tra (*B, A, setup, nrhs, ier*)

gen_lu_get_l (*B, A, setup, ier*)

gen_lu_get_u (*B, A, setup, ier*)

gen_lu_infinity_norm_inv (*a, A, setup, ier*)

deallocate_gen_lu (*setup*)

ARGUMENTS

In the descriptions that follow, **save_gen_lu** and **restore_gen_lu** are called the *LU save and restore* routines; **gen_lu_solve** and **gen_lu_solve_tra** are called the *LU solver* rou-

tines; **gen_lu_apply_l_inv**, **gen_lu_apply_u_inv**, **gen_lu_apply_l_inv_tra**, and **gen_lu_apply_u_inv_tra** are called the *LU factor application* routines; **gen_lu_get_l** and **gen_lu_get_u** are called the *LU get-factor* routines; and **gen_lu_infinity_norm_inv** is called the *LU infinity norm* routine.

Also, in this description, A and B refer to the active matrices with which the routines work. These matrices may be contained (as the upper left-hand submatrices) in larger matrices within A and B, respectively. Details are provided below.

Finally, the notation M^{-T} is used for $(M^{-1})^T = (M^T)^{-1}$.

setup Scalar integer variable. Setup ID returned by **gen_lu_factor** and **restore_gen_lu**. When you call any of the other *LU* routines, you must supply the value returned by the corresponding **gen_lu_factor** or **restore_gen_lu** call.

B CM array of the same type (real or complex) as A. The instance axes of B must match those of A in order of declaration and extents. When you call **gen_lu_get_l** or **gen_lu_get_u**, A and B must have the same rank, axis extents, and layout directives.

Solver and Factor Application Routines. When you call one of the *LU* solver or factor application routines, B must contain one or more instances of B, where each B consists of one or more right-hand-side vectors. Upon return from **gen_lu_solve** or **gen_lu_solve_tra**, each B within B is overwritten by the solution(s) to $AX = B$ or $A^TX = B$, respectively. Upon return from a factor application routine, each B is overwritten by the product $L^{-1}B$, $U^{-1}B$, $L^{-T}B$, or $U^{-T}B$.

For the solver and factor application routines, the following restrictions hold:

- If each instance B within B consists of only one right-hand-side vector (*nrhs* = 1), you may represent B in either of the following ways:
  - It may have rank 2 with number of columns = 1. In this case, each B has dimensions m x 1 (and may consist of the upper left-hand m x 1 elements of a larger matrix). The rows of each B must be counted by axis *row_axis* (from the **gen_lu_factor** call); the single column must be counted by axis *col_axis*.

- It may have rank 1. In this case, each *B* has dimension *m* (and may consist of the first *m* elements of a larger vector). The elements of each *B* must be counted by axis *row_axis* (if *row_axis* < *col_axis*) or by axis (*row_axis* - 1) (if *col_axis* < *row_axis*). For an example, see the Notes section.

- If each *B* within *B* consists of multiple right-hand-side vectors (*nrhs* > 1), then each *B* has dimensions *m* x *nrhs*, and may consist of the upper left-hand *m* x *nrhs* elements of a larger matrix. The rows and columns of *B* must be counted by axes *row_axis* and *col_axis*, respectively.

Get-Factor Routines. When you call **gen_lu_get_l** or **gen_lu_get_u**, *B* must have the same rank, axis extents, and layout directives as *A*. Upon completion of **gen_lu_get_l**, each *m* x *n* instance *B* within *B* defined by axes *row_axis* and *col_axis* is overwritten with the factor *L* of the corresponding *A* within *A*. Upon completion of **gen_lu_get_u**, each instance *B* within *B* is overwritten with the factor *U* of the corresponding *A* within *A*.

The *L* factor produced by **gen_lu_get_l** contains the effects of pivoting. Furthermore, the *L* and *U* factors produced by **gen_lu_get_l** and **gen_lu_get_u** are in block cyclic form. To obtain the factors in elementwise consecutive order, you may use the **compute_fe_block_cyclic_perms** and **permute_cm_matrix_axis_from_fe** routines.

Do not use the arrays obtained from the get-factors routines as input to the solver or factor application routines.

a Real CM array with the same rank and precision as *A*. The axes identified by *row_axis* and *col_axis* in the **gen_lu_factor** call must have extent 1. Thus, each matrix *A* embedded in *A* corresponds to a real number in ***a***.

Upon successful completion of **gen_lu_infinity_norm_inv**, the estimated infinity norm of the *inverse* of each matrix *A* within *A* is placed in the corresponding position of ***a***.

A Real or complex CM array. When you call **gen_lu_factor**, *A* should contain one or more instances of a coefficient matrix *A* to be factored. Each *A* is assumed to be dense with dimensions *m* x *n*. *m* must be greater than or equal to *n*; but if you specify *pivoting_*

strategy = **CMSSL_no_pivoting**, the current implementation requires $m = n$. Upon completion of **gen_lu_factor**, each A in A is overwritten with its LU factors.

The axes identified by *row_axis* and *col_axis* may have extents greater than m and n, respectively; that is, each instance of A may be contained in the upper left-hand m x n elements of a larger matrix wihin A.

When you call any of the other LU routines, A must have the same data type, rank, and shape (axis extents and layout directives, including orderings and weights) as the original A that was factored. You must also be using the same partition size as when you originally factored A. Supply in A the LU factors returned in A by **gen_lu_factor.**

m — Scalar integer variable. The number of rows in each coefficient matrix A within A. Also, the number of rows in each right-hand side B (or, if each B is a single vector, the number of elements in B). m must be greater than or equal to n; but if you specify *pivoting_strategy* = **CMSSL_no_pivoting**, the current implementation requires $m = n$.

If you intend to call **gen_lu_infinity_norm_inv**, m must equal n, since each matrix A within A must be invertible, and therefore square.

n — Scalar integer variable. The number of columns in each coefficient matrix A within A. m must be greater than or equal to n; but if you specify *pivoting_strategy* = **CMSSL_no_pivoting**, the current implementation requires $m = n$.

If you intend to call **gen_lu_infinity_norm_inv**, m must equal n, since each matrix A within A must be invertible, and therefore square.

row_axis — Scalar integer variable. Identifies the axis of A that counts the rows of each coefficient matrix A.

col_axis — Scalar integer variable. Identifies the axis of A that counts the columns of each coefficient matrix A.

nblock — Scalar integer variable. Blocking factor. The blocking factor you specify when you call **gen_lu_factor** is also used in any subsequent LU solver, factor application, get-factor, or infinity norm calls in

which you supply the setup ID returned by **gen_lu_factor**. Use these guidelines when choosing an *nblock* value:

- For typical applications, *nblock* = 8 is a good choice. An *nblock* value of 16 or even 32 may yield faster factorization in some cases.
- *nblock* should always be $\leq n$; *nblock* values > *n* use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*. If that is not possible, choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions.

pivoting_strategy Scalar integer variable specifying the pivoting strategy to be used. The value must be one of the following symbolic constants:

CMSSL_partial_pivoting
Selects partial pivoting. The pivot is chosen from the pivot column; rows are, in effect, permuted. Note that this implementation does not use block or parallel pivoting; it finds one pivot row at a time.

CMSSL_no_pivoting
Selects no pivoting. The pivot is taken from the block cyclic diagonal.

nrhs Scalar integer variable. The number of columns in each instance *B* within *B*. If each *B* is a single vector, supply 1 for the value of *nrhs*.

unit Scalar integer. Valid unit number associated with the file to or from which the *LU* state is to be written or read. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a device or socket with a unit number). The **save_gen_lu** and **restore_gen_lu** calls write and read data using **CMF_CM_ARRAY_TO_FILE_SO** and

CMF_CM_ARRAY_FROM_FILE_SO, respectively. You must rewind the file before calling **restore_gen_lu**.

iostat Scalar integer variable. Upon return, contains the status of the I/O operation. If *ier* = 0, *iostat* contains the number of bytes written or read. For the meanings of other *iostat* codes, refer to the descriptions of **CMF_CM_ARRAY_TO_FILE_SO** (for **save_gen_lu**) and **CMF_CM_ARRAY_FROM_FILE_SO** (for **restore_gen_lu**) in the CM Fortran documentation set.

ier Scalar integer variable. Return code; set to 0 upon successful return.

Values between -1 and -9, inclusive, indicate problems with one or more of the CM arrays containing matrices in any of the *LU* calls:

- -1 Invalid array home. The array must be a CM array.
- -2 Invalid rank; must be ≥ 2.
- -3 Invalid column extent; must be $\geq m$.
- -4 Invalid row extent; must be $\geq n$.
- -9 Invalid data type; must be real or complex (single- or double-precision).

Values that are multiples of -10 indicate problems with non-array arguments:

- -10 System failed to allocate the setup object, *setup*.
- -20 *m*, *n*, or *nrhs* is invalid; all must be > 0 and *m* must be greater than or equal to *n*.
- -30 *row_axis* or *col_axis* is invalid. $1 \leq$ *row_axis*, *col_axis* $\leq$ rank (*A*) must be true, and *row_axis* and *col_axis* must not be equal.
- -40 *nblock* is invalid; it must be greater than or equal to 1.
- -50 *pivoting_strategy* is invalid; must be **CMSSL_partial_pivoting** or **CMSSL_no_pivoting**.
- -60 *nrhs* is invalid.
- -80 You specified *m* not equal to *n* with **CMSSL_no_pivoting** in a factorization call, or you specified *m* not equal to *n* in the factorization call associated with this call to the infinity norm routine. These combinations are invalid.
- -100 *setup* is invalid. (You did not supply the value returned by **gen_lu_factor**.)

Values between -102 and -108, inclusive, indicate problems with the consistency of *A* or *B* in a solver, factor application, or get-factor routine:

- -102 The rank of *A* or *B* is invalid (must be ≥ 2 for *A* or ≥ 1 for *B*), or is inconsistent with the rank of *A* in the factorization call.
- -105 The extents of the instance axes of *A* or *B* are inconsistent with those of *A* in the factorization call.
- -106 *B* must have the same layout directives as *A* when you call **gen_lu_get_u** or **gen_lu_get_l**.
- -108 The data type of *A* or *B* is inconsistent with that of *A* in the factorization call.

The **save_gen_lu** and **restore_gen_lu** routines return the following value if they encounter an I/O error:

- -200 I/O error. See the value of *iostat* for more information.

DESCRIPTION

Given a CM array *A* containing one or more instances of a coefficient matrix *A*, and a CM array *B* containing corresponding right-hand sides *B*, the *LU* routines perform the operations listed below. All of the LU routines support multiple instances.

gen_lu_factor Uses Gaussian elimination (with or without partial pivoting) to factor each matrix instance *A* into two matrices, *L* and *U*, described below. If pivoting is specified, the effects of the pivoting are included in *L*.

save_gen_lu Saves internal information about the *LU* factors in a file.

restore_gen_lu Restores internal information about the *LU* factors from a file.

gen_lu_solve Uses the *LU* factors returned by **gen_lu_factor** to solve the system(s) $AX = B$.

gen_lu_solve_tra Uses the *LU* factors returned by **gen_lu_factor** to solve the system(s) $A^TX = B$.

gen_lu_apply_l_inv	Given the *LU* factors returned by **gen_lu_factor**, applies L^{-1} to *B*.
gen_lu_apply_u_inv	Given the *LU* factors returned by **gen_lu_factor**, applies U^{-1} to *B*.
gen_lu_apply_l_inv_tra	Given the *LU* factors returned by **gen_lu_factor**, applies L^{-T} to *B*.
gen_lu_apply_u_inv_tra	Given the *LU* factors returned by **gen_lu_factor**, applies U^{-T} to *B*.
gen_lu_get_l	Given the *LU* factors returned by **gen_lu_factor**, produces the factor *L* separately.
gen_lu_get_u	Given the *LU* factors returned by **gen_lu_factor**, produces the factor *U* separately.
gen_lu_infinity_norm_inv	Estimates the infinity norm of each matrix A^{-1}, given the *LU* factors of each *A* as computed by the **gen_lu_factor** routine.
deallocate_gen_lu	Deallocates the processing element memory required by the above routines.

Setup and Deallocation. The **gen_lu_factor** and **restore_gen_lu** routines allocate processing element storage space and return a setup ID. You must supply this setup ID in subsequent *LU* solver, factor application, get-factor, and infinity norm calls, or the **save_gen_lu** routine, as long as you are working with the same set of factors; you must also supply it to **deallocate_gen_lu**. You can follow one call to **gen_lu_factor** or **restore_gen_lu** with multiple calls to the other *LU* routines, thus avoiding the overhead of factoring the same matrix or matrices repeatedly.

The **deallocate_gen_lu** routine deallocates the memory needed for a particular factorization, and invalidates the associated setup ID. Attempts to use a deallocated setup ID result in errors.

You can work with more than one set of *LU* factors at a time by calling **gen_lu_factor** or **restore_gen_lu** more than once without calling **deallocate_gen_lu**. Be sure to supply the correct setup ID in each subsequent *LU* call. When you have finished working with a set of factors, be sure to use **deallocate_gen_lu** to deallocate the associated memory. Repeated calls to **gen_lu_factor** or **restore_gen_lu** without deallocation can cause you to run out of memory.

Factorization Routine. The **gen_lu_factor** routine uses Gaussian elimination (with or without pivoting) to factor each instance of A into two matrices, L and U. One common representation for this factorization is

$$PA = LU$$

where P is a permutation matrix resulting from the pivoting process, and L and U are lower triangular and upper triangular, respectively. However, because of details of the implementation of the *LU* routines, in this description we represent the factorization as

$$A = LU$$

where the effects of pivoting are included in L. Thus, L is the inverse of the operator defined by the sequence of row operations performed in the Gaussian elimination process (which occurs in block cyclic order). The row operations include the row interchanges, if pivoting is specified. Therefore, L is not necessarily lower triangular, and U is not upper triangular. See **The *LU* Factors Defined**, below, for details.

Upon completion of **gen_lu_factor**, each instance of A within A is overwritten with data giving the *LU* factors of A. When you call the *LU* solver, factor application, and get-factors routines, you must supply the same A that was returned by **gen_lu_factor**. To obtain the L and U factors separately, use the get-factor routines.

Save and Restore Routines. You may save internal information about the *LU* factors in a file for use in later calls to the other *LU* routines. To save the *LU* information, call **save_gen_lu** after the factorization is complete but before deallocating the storage space. To restore the *LU* information, rewind the file and call **restore_gen_lu**; this call is typically followed by calls to the other *LU* routines.

Solver Routines. To solve $AX = B$, **gen_lu_solve** performs forward elimination:

$A = LU$; let $UX = C$

$C = L^{-1}B$

followed by back substitution:

$X = U^{-1}C = U^{-1}(L^{-1}B)$

Similarly, to solve $A^{T}X = B$, **gen_lu_solve_tra** performs forward elimination:

$A^{T} = U^{T}L^{T}$; let $L^{T}X = C$

$C = U^{-T}B$

followed by back substitution:

$$X = L^{-T}C = L^{-T}(U^{-T}B)$$

Upon completion of the solver routines, each B within $\boldsymbol{B}$ is overwritten with the solution.

Factor Application Routines. The **gen_lu_apply_l_inv, gen_lu_apply_u_inv, gen_lu_apply_l_inv_tra,** and **gen_lu_apply_u_inv_tra** routines allow you to apply matrices derived from the LU factors to arbitrary matrices or vectors B contained in $\boldsymbol{B}$. Upon completion of the routine, each B in $\boldsymbol{B}$ is overwritten with the specified product ($L^{-1}B$, $U^{-1}B$, $L^{-T}B$, or $U^{-T}B$). Thus, these routines use the L and U factors to solve triangular systems of the form $LX=B$, $L^TX=B$, $UX=B$, and $U^TX=B$.

In most cases, you should use a solver routine, rather than using the factor application routines separately, to solve $AX = B$ or $A^TX = B$. Using the factor application calls may require an extra permutation in the case of no pivoting. For details about exactly how the LU factors and their inverses are defined, see **The *LU* Factors Defined**, below.

Get-Factor Routines. The **gen_lu_get_l** and **gen_lu_get_u** routines provide access to the L and U factors separately. Upon completion of **gen_lu_get_l**, each B within $\boldsymbol{B}$ contains the factor L for the corresponding coefficient matrix A within $\boldsymbol{A}$. Upon completion of **gen_lu_get_u**, each B within $\boldsymbol{B}$ contains the factor U for the corresponding coefficient matrix A within $\boldsymbol{A}$. The rows and columns of the factors are counted by axes *row_axis* and *col_axis*, respectively.

The L factor produced by **gen_lu_get_l** contains the effects of pivoting. Furthermore, the L and U factors produced by **gen_lu_get_l** and **gen_lu_get_u** are in block cyclic form. To "undo" the block cyclic ordering, you may use the **compute_fe_block_cyclic_perms** and **permute_cm_matrix_axis_from_fe** routines. (For an example, see the on-line sample code in the subdirectory `block-cyclic/cmf/` of the CMSSL examples directory.) For details about exactly how the LU factors and their inverses are defined, see **The *LU* Factors Defined**, below.

Infinity Norm Routine. Given the LU factors returned by **gen_lu_factor**, the **gen_lu_infinity_norm_inv** routine estimates the infinity norm of each matrix A^{-1}. Upon successful completion, the infinity norm of each A^{-1} is placed in the position of $\boldsymbol{a}$ corresponding to A.

The infinity norm of a matrix M, denoted here by $\| M \|_\infty$, is defined by

$$\| M \|_\infty = \max_{\| x \|_\infty = 1} \| Mx \|_\infty$$

where the infinity norm of a vector, $\| x \|_\infty$, is defined as the maximum of the absolute values of the vector components:

$$\| x \|_\infty = \max_i | x_i |$$

The infinity-norm condition number of a matrix M is equal to the product of $\| M \|_\infty$ and $\| M^{-1} \|_\infty$.

The *LU* Factors Defined (Square Case). The following definitions apply to the case in which $m = n$. Effectively, the **gen_lu_factor** routine factors a block cyclic permutation, A_c, of each matrix A that you supply in A. In a factorization with pivoting, the matrix A_c is factored into

$$A_c = P^{-1} L_c U_c$$

where L_c is lower triangular, U_c is upper triangular, and P is the permutation matrix resulting from the pivoting process. In a factorization without pivoting, the factorization is

$$A_c = L_c U_c$$

where L_c is lower triangular and U_c is upper triangular.

The *LU* factors of A are defined in terms of A_c and its factors as follows:

- *Case 1: Factorization with pivoting*

  By definition,

  $$A_c = P_1^{-1} A P_2$$

  where P_1 is the permutation giving the correspondence between standard and block cyclic row order, and P_2 is the permutation giving the correspondence between standard and block cyclic column order. (These permutations depend on the array size and layout, the partition size, and the blocking factor you supply.) We therefore have

$$A = LU = P_1 A_c P_2^{-1}$$
$$= P_1 (P^{-1} L_c U_c) P_2^{-1}$$
$$= P_1 P^{-1} L_c (P_1^{-1} P_1) U_c P_2^{-1}$$

from which we choose to define

$$L = P_1 P^{-1} L_c P_1^{-1}$$
$$U = P_1 U_c P_2^{-1}$$

- *Case 2: Factorization without pivoting*

  The no-pivoting case requires that no small pivots be encountered during the elimination process. Therefore, the factorization routine pre-permutes the matrix A to assure that the diagonal elements of A also appear on the diagonal of A_c. Internally, A is pre-permuted to obtain

  $$A^* = A P_1 P_2^{-1}$$

  By definition, we have

  $$A_c = P_1^{-1} A^* P_2$$

  from which it follows that

  $$A_c = P_1^{-1} A P_1$$

  and therefore

  $$A = LU = P_1 A_c P_1^{-1}$$
  $$= P_1 L_c U_c P_1^{-1}$$
  $$= P_1 L_c (P_1^{-1} P_1) U_c P_1^{-1}$$

  from which we choose to define

  $$L = P_1 L_c P_1^{-1}$$
  $$U = P_1 U_c P_1^{-1}$$

The **gen_lu_get_l** and **gen_lu_get_u** routines return the L and U factors defined above. The inverses L^{-1}, U^{-1}, L^{-T}, U^{-T} applied by the factor application routines are derived from the L and U factors defined above and are true inverses; that is, $L^{-1}L = LL^{-1} = I = U^{-1}U = UU^{-1}$. (The inverses are also true in the block cyclic space; that is, $L_c^{-1}L_c = L_c L_c^{-1} = I = U_c^{-1}U_c = U_c U_c^{-1}$.)

The definitions above generalize to the non-square case ($m > n$) using the same principles.

NOTES

NaNs and Infinities. As mentioned above, the matrices *A* and *B* may be contained (as the upper left-hand submatrices) in larger matrices within the arrays *A* and *B*, respectively. In this case, if there are NaNs or infinities in the larger matrix outside of *A* or *B*, it is possible that other locations outside of *A* or *B* could become NaNs or infinities as well.

Distinct Variables. The input CM arrays *A* and *B* must be distinct variables.

Include the CMSSL Header File. The **gen_lu_factor** routine is a function and uses symbolic constants. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL functions and symbolic constants.

Argument Values. The internal variable *setup* is required for communicating information between the factorization routine and the other *LU* routines. The application must not modify the contents of this variable.

Saving and Restoring the *LU* State. If you want to save the internal state in one program run and restore it in a different run, you must save the array of factored matrices in a file in addition to saving the internal state using **save_gen_lu**. Be sure to save the array in a different file than that used for saving the state. When you read the array back into memory prior to restoring the internal state, you must use the same partition size as when you originally performed the factorization; and the restored array must have exactly the same shape (axis extents and layout directives, including orderings and weights) as when you saved it.

Nondegeneracy Required. In the current release, each matrix *A* within *A* must have a column space of rank *n* when you call one of the solver routines.

Rank of *B*. The following example illustrates the options for defining the rank of *B*. Suppose *A*, *n*, *m*, *row_axis*, and *col_axis* are defined as follows:

A (5, 10, 5)
m = *n* = 5
row_axis = 1
col_axis = 3

and each ***B*** in ***B*** is a single vector. You may define ***B*** in either of the two following (equivalent) ways:

B (5, 10, 1)
B (5, 10)

On the other hand, if you define

A (5, 10, 5)
$m = n = 5$
row_axis = 3
col_axis = 1

then the possibilities for ***B*** are as follows:

B (1, 10, 5)
B (10, 5)

Performance. Performance improves for larger subgrid sizes (and therefore depends upon the layout of ***A***). For information on subgrids, refer to the CM Fortran documentation set.

To optimize performance, follow these guidelines:

- Ensure that the subgrid length in each dimension is a multiple of *nblock*. If that is not possible, choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions.
- Lay out ***A*** so that the subgrid sizes along axes *row_axis* and *col_axis* differ from one another by no more than a factor of 4 or 5.
- Use axis extents exactly equal to m x n for the matrices ***A*** and m x *nrhs* for the matrices ***B***. Use the same processing element layout for the arrays ***A*** and ***B***.

Numerical Complexity. If the matrices ***A*** have dimensions (m x n), the matrices ***B*** have *nrhs* right-hand sides, and I is the number of instances (the product of all axis extents except axes *row_axis* and *col_axis*), then:

- The *LU* factorization routine requires approximately $[m-(n/3)]n^2I$ floating-point operations for real operands and $4[m-(n/3)]n^2I$ floating-point operations for complex operands.
- The *LU* solver routine requires approximately $2*nrhs*(2m-n)nI$ floating-point operations for real operands and $8*nrhs*(2m-n)nI$ floating-point operations for complex operands.

Performance Cost of Pivoting. The cost of pivoting is very much dependent on size and layout. The extra cost of pivoting is greatest for relatively small matrices. For very large matrices (using nearly all processing element memory), the performance of the factorization with pivoting is comparable to the performance without pivoting, whereas the solver remains about 50% slower for the pivoting version.

EXAMPLES

Sample CM Fortran code that uses the *LU* routines can be found on-line in the subdirectories

```
lu/cmf/
```

and

```
infinity-norm/cmf/
```

of a CMSSL examples directory whose location is site-specific.

5.3 Routines for Solving Linear Systems Using Householder Transformations ("*QR*" Routines)

This section describes the CMSSL routines for solving linear systems using Householder transformations (commonly referred to as the "*QR*" routines). The following topics are included:

- the *QR* routines and their functions
- *QR* factorization
- Householder algorithm
- blocking, load balancing, and the *QR* factors defined
- numerical stability
- the pivoting option: working with ill-conditioned systems
- saving and restoring the *QR* state

For detailed descriptions of the *QR* routines (including calling sequences, argument definitions, and information about usage), refer to the man page at the end of this section.

Throughout this section, the following conventions are used:

- $\overline{M}$ denotes the conjugate of M.
- M^{-T} denotes $(M^{-1})^T = (M^T)^{-1}$.
- "*A*" refers to a matrix being factored and "*B*" refers to the right-hand side(s). One or more instances of *A* and *B* are embedded (possibly within larger matrices) in the CM arrays *A* and *B*, respectively; the operations described are performed on all instances concurrently. The man page provides details about *A* and *B*.

5.3.1 The *QR* Routines and Their Functions

Given a CM array *A* containing one or more instances of a coefficient matrix *A*, and a CM array *B* containing corresponding right-hand sides *B*, the CMSSL provides the routines and operations listed below.

Factorization routine:

gen_qr_factor — Uses Householder transformations to factor each matrix instance A into two matrices, Q and R, (or, if pivoting is specified, three matrices, Q, R, and P^{-1}), described in Section 5.3.2.

Save and restore routines:

save_gen_qr — Saves internal information about the QR factors in a file.

restore_gen_qr — Loads internal information about the QR factors from a file.

Solver routines:

gen_qr_solve — Uses the QR factors returned by **gen_qr_factor** to solve the system(s) $AX = B$.

gen_qr_solve_tra — Uses the QR factors returned by **gen_qr_factor** to solve the system(s) $A^{T}X = B$.

Factor application routines: These routines use the factors produced by the QR factorization routine to solve triangular systems of the form $RX=B$ and $R^{T}X=B$ and trapezoidal systems of the form $QX=B$ or $Q^{T}X=B$.

gen_qr_apply_q — Given the QR factors returned by **gen_qr_factor**, applies Q (or $\overline{Q}$, in the case of complex data) to B for each instance.

gen_qr_apply_q_tra — Given the QR factors returned by **gen_qr_factor**, applies Q^{T} (or Q^{H}, in the case of complex data) to B for each instance. Note that since Q is orthogonal (or unitary, in the complex case), $Q^{H}=Q^{-1}$.

gen_qr_apply_r_inv — Given the QR factors returned by **gen_qr_factor**, applies R^{-1} to B for each instance.

gen_qr_apply_r_inv_tra — Given the *QR* factors returned by **gen_qr_factor**, applies R^{-T} to *B* for each instance.

Get-*R* routine:

gen_qr_get_r — Given the *QR* factors returned by **gen_qr_factor**, produces the factor *R* for each instance.

Pivot application routines:

gen_qr_apply_p — Given the *QR* factors returned by **gen_qr_factor**, applies *P* to *B* for each instance, where *P* is the permutation matrix that corresponds to the pivoting process. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

gen_qr_apply_p_inv — Given the *QR* factors returned by **gen_qr_factor**, applies $P^{-1} = P^{T}$ to *B* for each instance. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

Zeroing routine:

gen_qr_zero_rows — Zeroes the final rows of each two-dimensional matrix contained in *B*. The rows are counted in block cyclic order.

Diagonal manipulation routines:

gen_qr_extract_diag — Given the *QR* factors returned by **gen_qr_factor**, returns the block cyclic diagonal entries of *R* for each instance.

gen_qr_deposit_diag — Given the *QR* factors returned by **gen_qr_factor**, overwrites the block cyclic diagonal entries of each instance of *R* with values you supply.

Infinity norm routines:

gen_qr_infinity_norm_inv	Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of each matrix A^{-1}. Uses the method developed by Hager (see reference 5 listed in Section 5.7).
gen_qr_r_infinity_norm_inv	Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of each $(R^*)^{-1}$, where R^* is the block cyclic upper-left corner formed by discarding any trailing columns of R that contain zeros on the block cyclic diagonal.

Deallocation routine:

deallocate_gen_qr	Deallocates the processing element memory required by the above routines.

Memory Allocation and Deallocation

You must call the factorization or restore routine before calling a solver, get-*R*, factor application, pivot application, zeroing, diagonal manipulation, or infinity norm routine. You can follow one call to the factorization routine with multiple calls to these other routines, thus avoiding the overhead of factoring the same matrices repeatedly. The deallocation routine deallocates the processing element memory allocated by the factorization routine and required by the other *QR* routines. For more information about these points, refer to the man page at the end of this section.

5.3.2 *QR* Factorization

When you call the *QR* factorization routine and specify no pivoting, each matrix A is factored into two matrices:

$$A = QR$$

When you call the factorization routine and specify pivoting, each matrix A is factored into three matrices:

$$A = QRP^{-1}$$

where P is the permutation matrix that corresponds to the pivoting process. The factors are defined in more detail in Section 5.3.4.

NOTE

Sections 5.3.3 and 5.3.4 contain detailed information about how the *QR* routines are implemented. This information may help you choose optimal values for the *nblock* (blocking factor) and *back_solve_strategy* arguments. However, if you do not need the detailed descriptions in these sections, the argument descriptions in the man page will probably provide you with enough information to choose reasonable values for these arguments.

5.3.3 Householder Algorithm

This section provides more details about the Householder algorithm implemented in the *QR* factorization and solver routines. For simplicity, the algorithm is described for the no-pivoting case and without accounting for blocking and load balancing. For details about the operations performed if you specify pivoting, and the operations performed by the transpose solver routine, see the man page at the end of this section. For information about blocking and load balancing, see Section 5.3.4 and Chapter 14.

NOTE

This section assumes that A and B are real. For complex matrices, replace "orthogonal" with "unitary" and replace "Q^T" with "Q^H."

The Householder reduction algorithm computes a series of n Householder matrices that, when successively multiplied by the coefficient matrix A, yield an upper triangular matrix R. As an example, suppose the coefficient matrix A has size (4 x 3). First, a Householder transformation H_1 is calculated, such that H_1 applied to the first column of A yields a vector that is 0 except for its first component. H_1 is applied to all the columns of A. Next, a Householder transformation H_2 is calculated such that H_2 preserves the new first column of A and H_2 applied to the new second column of A yields a vector that is 0 except for its first and second components. H_2 is applied to all the columns of A. Finally, a Householder transformation H_3 is calculated such that H_3 preserves the first 2 columns of A and H_3 applied to the last column of A yields a vector that is 0 except for its first, second, and third components. (See Figure 15.)

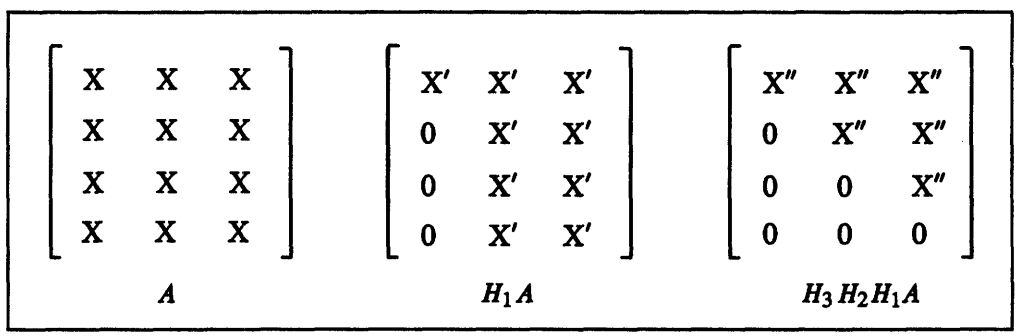

Figure 15. Successive Householder transformations (discounting blocking and load balancing).

Each Householder transformation is defined by a corresponding vector. The transformation associated with the vector v is given by the following formula:

$$Hw = w - \left(\frac{2 < w, v >}{< v, v >} \right) v$$

Here the notation $< x,y >$ means the scalar product of the vectors x and y. If x and y have n components, then

$$< x,y > = \sum_{i=1}^{n} x_i y_i$$

(If x and y are complex, then

$$< x,y > = \sum_{i=1}^{n} x_i \overline{y_i} \quad)$$

After n Householder transformations, the upper triangular matrix R has been computed as $R = (H_n \; H_{n-1} \ldots H_1)A$. Since each H is orthogonal, the product $H_n \ldots H_1$ is also orthogonal, and is defined as Q^T. The nontrivial portion of Q is a series of Householder vectors.

Upon return from the factorization routine, the upper triangular portion of A is overwritten by R, and the lower triangular portion of A is overwritten by the Householder vectors that form the non-trivial portion of Q, as shown in Figure 16. Specifically, components $j+1{:}m$ of the j^{th} Householder vector are stored in $A(j+1{:}m, j)$ for all j less than m. (Bear in mind that this description does not take blocking and load balancing into effect.)

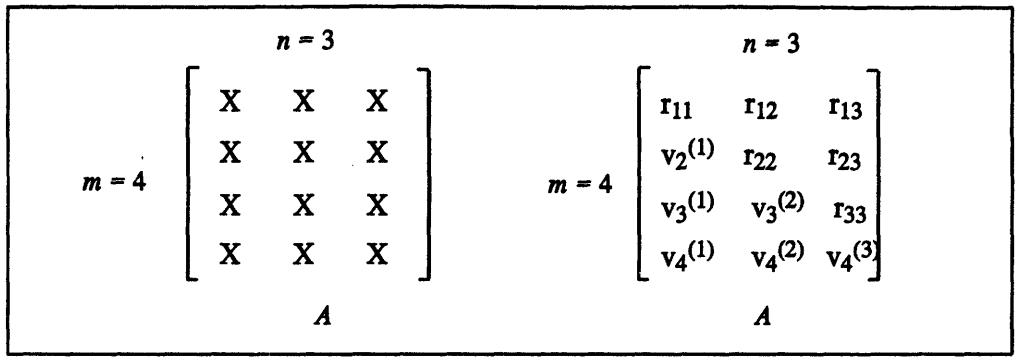

Figure 16. The upper triangular portion of *A* is overwritten by *R* and the lower triangular portion is overwritten by Householder vectors.

The stored Householder vectors are normalized so that $v(1:j-1) = 0$ and $v(j) = 1$ for the j^{th} Householder vector. This normalization makes it possible to store only the essential part of each Householder vector in the strictly lower triangular portion of *A*, leaving room in the upper triangular portion (including the diagonal) for *R*.

The *QR* solver routine applies Q^T (the *n* Householder transformations corresponding to the *n* vectors stored in the strictly lower triangular part of *A*) to the right-hand sides that comprise the matrix *B*. In this way, the linear system $AX = B$ is transformed to $RX = Q^TB$. This upper triangular system is solved by *r* concurrent back substitutions, where *r* is the number of right-hand-side vectors. For each right-hand-side vector *b* in *B*, the system $Ax = b$ becomes

$$x = R^{-1}Q^Tb$$

Again discounting blocking and load balancing, the solver overwrites the first *n* rows of the right-hand-side matrix *B* with the least squares solution to $AX = B$. The remaining $m - n$ rows of *B* are undefined.

5.3.4 Blocking, Load Balancing, and the *QR* Factors Defined

The *QR* routines use blocking and load balancing to optimize performance. Blocking and load balancing are described in detail in the section on computation of block cyclic permutations in Chapter 14, and in reference 11 listed in Section 5.7. This section discusses the arguments that affect blocking and load balancing in the *QR* routines.

NOTE

This section assumes that A and B are real. For complex matrices, replace "orthogonal" with "unitary" and replace "Q^T" with "Q^H."

Choosing a Blocking Factor

The *QR* routines require you to supply a blocking factor in the *nblock* argument. (The blocking factor is defined in the section on computation of block cyclic permutations in Chapter 14.) If you specify pivoting, the current implementation requires that you supply a blocking factor of 1. In other cases, use these guidelines when choosing an *nblock* value:

- For typical applications, *nblock* = 8 is a good choice. An *nblock* value of 16 may yield faster factorization in some cases.
- *nblock* should always be $\leq n$; *nblock* values > n use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*. If that is not possible, choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions.

The *QR* Factors Defined

Effectively, the **gen_qr_factor** routine factors a block cyclic permutation, A_c, of each matrix A that you supply in A. In a factorization with pivoting, the matrix A_c is factored into

$$A_c = Q_c R_c P_c^{-1}$$

where R_c is upper triangular, Q_c is orthogonal (or unitary, in the complex case), and P_c is the permutation matrix resulting from the pivoting process. In a factorization without pivoting, the factorization is

$$A_c = Q_c R_c$$

where R_c is upper triangular and Q_c is orthogonal (or unitary).

The definitions of Q and R in terms of A_c and its factors depend on the value you supply in the **gen_qr_factor** *back_solve_strategy* argument. The two possible values are **CMSSL_qr_post_permute** and **CMSSL_qr_pre_permute**. The factor definitions are provided below for the square case ($m = n$). Note that the **CMSSL_qr_pre_permute** strategy does not work with pivoting, and requires that $m = n$. Details about the two back solve strategies are provided in the subsections that follow.

- *Case 1: Post-permutation; no pivoting; m = n*

  By definition,

  $$A_c = P_1^{-1} A P_2$$

  where P_1 is the permutation giving the correspondence between standard and block cyclic row order, and P_2 is the permutation giving the correspondence between standard and block cyclic column order. (These permutations depend on the array size and layout, the partition size, and the blocking factor you supply.) We therefore have

  $$\begin{aligned} A = \quad QR &= P_1 A_c P_2^{-1} \\ &= P_1 (Q_c R_c) P_2^{-1} \\ &= P_1 Q_c (P_1^{-1} P_1) R_c P_2^{-1} \end{aligned}$$

  from which we choose to define

  $$\begin{aligned} Q &= P_1 Q_c P_1^{-1} \\ R &= P_1 R_c P_2^{-1} \end{aligned}$$

- *Case 2: Post-permutation with pivoting; m = n*

  This case is just like the Case 1 except that we include P, the permutation matrix that corresponds to the pivoting process. We have

  $$\begin{aligned} A = \quad QRP^{-1} &= P_1 A_c P_2^{-1} \\ &= P_1 (Q_c R_c P_c^{-1}) P_2^{-1} \\ &= P_1 Q_c (P_1^{-1} P_1) R_c (P_2^{-1} P_2) P_c^{-1} P_2^{-1} \\ &= (P_1 Q_c P_1^{-1}) (P_1 R_c P_2^{-1}) (P_2 P_c^{-1} P_2^{-1}) \end{aligned}$$

  from which we choose to define

$$Q = P_1 Q_c P_1^{-1}$$
$$R = P_1 R_c P_2^{-1}$$
$$P^{-1} = P_2 P_c^{-1} P_2^{-1}$$

- *Case 3: Pre-permutation; no pivoting; m = n*

  In this case, the factorization routine pre-permutes the matrix A to obtain

  $$A^* = A P_1 P_2^{-1}$$

  By definition, we have

  $$A_c = P_1^{-1} A^* P_2$$

  from which it follows that

  $$A_c = P_1^{-1} A P_1$$

  and therefore

  $$\begin{aligned} A = QR \; &= P_1 A_c P_1^{-1} \\ &= P_1 Q_c R_c P_1^{-1} \\ &= P_1 Q_c \; (P_1^{-1} P_1) \; R_c P_1^{-1} \end{aligned}$$

  from which we choose to define

  $$Q = P_1 Q_c P_1^{-1}$$
  $$R = P_1 R_c P_1^{-1}$$

In the square case, the **gen_lu_get_r** routine returns the R factors defined above. The matrices R^{-1} and R^{-T}, Q, and Q^T applied by the factor application routines are derived from the Q and R factors defined above, and the inverses are true inverses; that is, $R^{-1}R = RR^{-1} = I = Q^T Q = QQ^T$. (The inverses are also true in the block cyclic space; that is, $R_c^{-1} R_c = R_c R_c^{-1} = I = Q_c^T Q_c = Q_c Q_c^T$.)

Finally, the definitions above generalize to the non-square case ($m > n$) using the same principles:

- *Case 4: Post-permutation; m > n*

  The matrices R, R^{-1}, and R^{-T} are defined for this case in Figure 17. The matrices in this figure have the following dimensions, assuming A has dimensions $m \times n$:

R_c	$n \times n$
P_1	$m \times m$
P_2	$n \times n$
I	$(m - n) \times (m - n)$ identity matrix

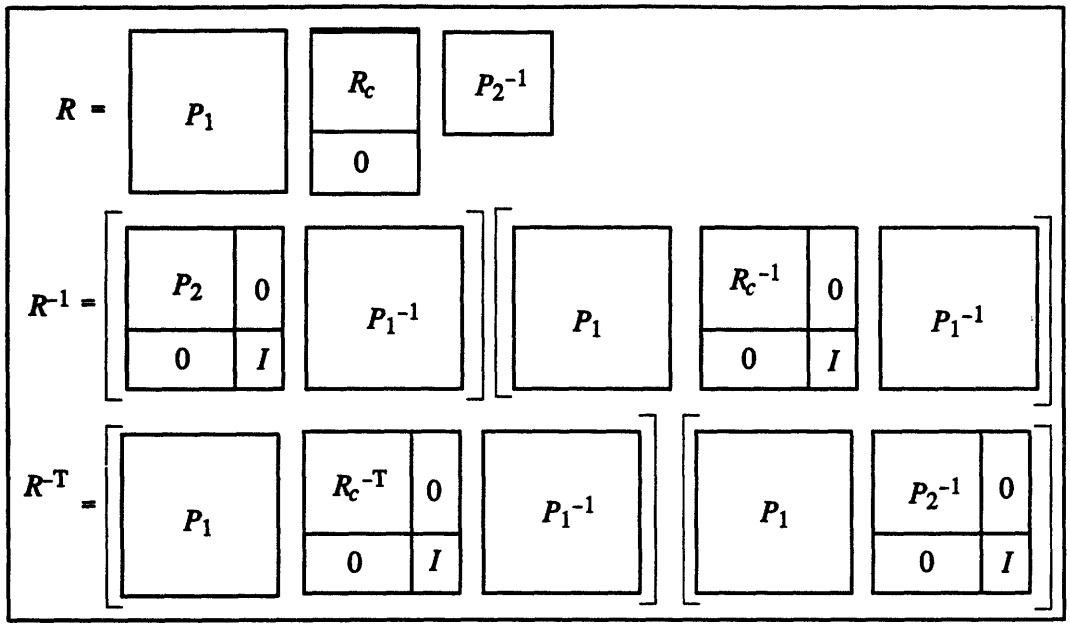

Figure 17. Definitions of R, R^{-1}, and R^{-T} in the post-permutation case with $m > n$.

R^{-1} and R^{-T} have the following effects:

- When R^{-1} operates on a matrix, the first n block cyclic rows are operated on, and end up in the first n rows. The last $m - n$ block cyclic rows are permuted into the last $m - n$ rows.
- When R^{-T} operates on a matrix, the first n rows are operated on and end up in the first n block cyclic rows. The last $m - n$ rows end up in the last $m - n$ block cyclic rows.

Summary of Factor Definitions (Square Case)

The factorization routine operates on A with a sequence of block Householder transformations that result in the matrix $R = (P_1\ R_c\ P_2^{-1})$ in the post-permutation case, or $R = (P_1\ R_c\ P_1^{-1})$ in the pre-permutation case. The sequence of transformations, which is orthogonal by construction, is defined as Q^T. Thus, the factorization yields

$Q^TA = (P_1\ R_c\ P_2^{-1})$, or $A = Q\ (P_1\ R_c\ P_2^{-1})$ (post-permutation, no pivoting)
$Q^TA = (P_1\ R_c\ P_2^{-1})P^{-1}$, or $A = Q\ (P_1\ R_c\ P_2^{-1})P^{-1}$ (post-permutation, pivoting)
$Q^TA = (P_1\ R_c\ P_1^{-1})$, or $A = Q\ (P_1\ R_c\ P_1^{-1})$ (pre-permutation, no pivoting)

where $Q = P_1 Q_c P_1^{-1}$ and $P^{-1} = P_2 P_c^{-1} P_2^{-1}$.

When the factorization routine returns, the block cyclic upper triangle of A is overwritten with $(P_1\ R_c\ P_2^{-1})$ (post-permutation) or $(P_1\ R_c\ P_1^{-1})$ (pre-permutation). The remaining elements of A are used internally to reconstruct Q.

Figure 18 is a simple example showing the shape of the non-zero entries of $(P_1\ R_c\ P_2^{-1})$ when A is a (32 x 32) matrix laid out as an (8 x 8) subgrid on each processing element of a (4 x 4)-processing-element layout. The block size in this example is 2.

Choosing a Back Solve Strategy

This section provides background information about the two back solve strategies and guidelines for choosing a strategy.

Because R_c is permuted by P_1 and P_2, the back substitution process may require further permutations in order to arrive at the solution to the original linear system. The *back_solve_strategy* argument allows you to determine when these further permutations occur:

- The **CMSSL_qr_pre_permute** strategy causes the factor routine to permute the columns of A prior to the factorization.
- The **CMSSL_qr_post_permute** strategy causes the solver routine to permute the rows of the solution after the back substitution is complete.

The **CMSSL_qr_post_permute** always works; however, your choice of back solve strategy may affect performance. Follow these guidelines:

- If the matrices A are not square, you *must* choose **CMSSL_qr_post_permute**. Specifying **CMSSL_qr_pre_permute** with non-square matrices yields an error.
- If you are specifying pivoting, you *must* choose **CMSSL_qr_post_ permute**. Specifying **CMSSL_qr_pre_permute** with pivoting yields an error.
- If the matrices A are square, each A has square subgrids, and you are not pivoting, the permutations are not required and your choice of back solve strategy has no effect on performance.

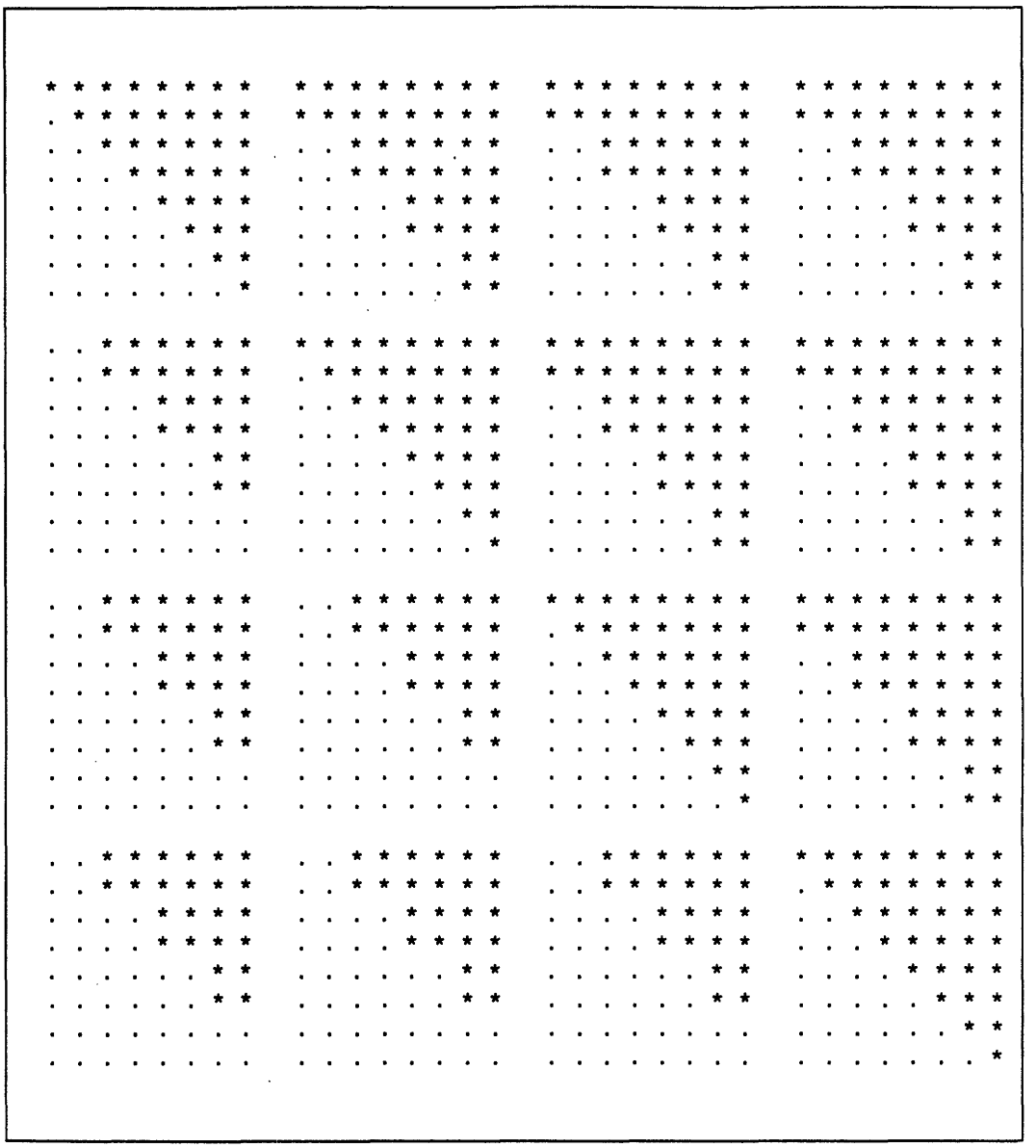

Figure 18. Non-zero entries in ($P_1 R_c P_2^{-1}$) for a (32 x 32) matrix laid out as an (8 x 8) subgrid on each processing element of a (4 x 4)-processing-element layout. The block size is 2.

- If the matrices *A* are square but do not have square subgrids, and you are not pivoting, then use these guidelines:
  - If the layouts of *A* and *B* coincide (most typically in this context, this means that the matrix axis extents are exactly *n* x *n* for each *A* and *n* x *nrhs* for each *B*, and that the layout of processing elements

is the same for A and B), then to optimize performance, choose the back solve strategy that moves less data. The two strategies move the following amounts of data for each instance:

- **CMSSL_qr_pre_permute** moves n^2 elements (the number of elements in each matrix A).
- **CMSSL_qr_post_permute** moves $n\Sigma r_i$ elements, where the sum is over all calls made to the solve routine after one call to the factor routine, and r_i is the number of right-hand-side vectors in the ith call to the solve routine.

Therefore, if $n^2 < n\Sigma r_i$ or $n < \Sigma r_i$, choose **CMSSL_qr_pre_permute**; if $n > \Sigma r_i$, choose **CMSSL_qr_post_permute**. If $n = \Sigma r_i$, the two strategies are likely to yield approximately the same performance.

- If the layouts of A and B do not coincide, choose **CMSSL_qr_post_permute**, which does not move any elements in this case (as compared with **CMSSL_qr_pre_permute**, which moves n^2 elements).

Back Solve Strategy Details

The following descriptions of the two back solve strategies are for readers who need more details about the permutations. For simplicity, this discussion covers the no-pivoting case. For details about the operations performed if you specify pivoting, and the operations performed by the transpose solver routine, see the man page at the end of this section.

In the following descriptions, bear in mind that

$Q = P_1 Q_c P_1^{-1}$ (both back solve strategies)
$Q^T = P_1 Q_c^T P_1^{-1}$ (both back solve strategies)
$R = P_1 R_c P_2^{-1}$ (post-permutation)
$R = P_1 R_c P_1^{-1}$ (pre-permutation)

In the **CMSSL_qr_post_permute** strategy, the solver routine backsolves the system

$$(P_1 R_c P_1^{-1})\, y = (P_1 Q_c^T P_1^{-1})\, b$$

to yield

$$y = (P_1 R_c P_1^{-1})^{-1} (P_1 Q_c^T P_1^{-1})\, b$$

for each right-hand-side vector b in the matrix B. The original system $Ax = b$ then becomes

$$(P_1 Q_c\ P_1^{-1})\ (P_1\ R_c\ P_2^{-1})\ x = b$$

$$(P_1\ R_c\ P_2^{-1})\ x = (P_1 Q_c^{T} P_1^{-1})\ b$$

$$P_1\ R_c\ (P_1^{-1}\ P_1)\ P_2^{-1}\ x = (P_1 Q_c^{T} P_1^{-1})\ b$$

$$(P_1\ P_2^{-1})\ x = (P_1\ R_c\ P_1^{-1})^{-1}\ (P_1 Q_c^{T} P_1^{-1})\ b$$

$$x = (P_2\ P_1^{-1})\ y$$

If you choose **CMSSL_qr_pre_permute** when you call **gen_qr_factor**, the factor routine multiplies each A on the right with $P_1 P_2^{-1}$ by doing a send to rearrange the columns before performing the factorization. (If A has square subgrids, then $P_1 = P_2$, so this permutation is the identity and no send is performed.) The factorization yields

$$Q^{T} A\ (P_1 P_2^{-1}) = (P_1 Q_c^{T} P_1^{-1})\ A\ (P_1\ P_2^{-1}) = (P_1\ R_c\ P_2^{-1})$$

Thus, for each right-hand-side vector b in the matrix B, the original system

$$Ax = b$$

or

$$(P_1 Q_c^{T} P_1^{-1})\ A\ \ (P_1 P_2^{-1} P_2 P_1^{-1})\ x = (P_1 Q_c^{T} P_1^{-1})\ b$$

is equivalent to

$$(P_1\ R_c\ P_2^{-1})\ (P_2\ P_1^{-1})\ x = (P_1 Q_c^{T} P_1^{-1})\ b$$

$$(P_1\ R_c\ P_1^{-1})\ x = (P_1 Q_c^{T} P_1^{-1})\ b$$

$$x = (P_1\ R_c\ P_1^{-1})^{-1}\ (P_1 Q_c^{T} P_1^{-1})\ b$$

In this case, the solver routine produces the desired result without a post-permutation. Finally, note that if A is $(m \times n)$ with $m > n$, then P_1 is $(m \times m)$, and multiplying A by P_1 on the right does not make sense. This is why the **CMSSL_qr_pre_permute** strategy requires A to be square.

Solver Routine Results

Assuming that A is (m x n), when the solver routine returns, the first n rows of the right-hand-side matrix B are overwritten with the least squares solution to $AX = B$. The remaining $m - n$ rows of B are undefined on return.

5.3.5 Numerical Stability

The orthogonalization methods used in the *QR* factorization have guaranteed stability; there is no "growth factor" as with Gaussian elimination. Even for extremely poorly conditioned matrices, the *QR* factorization routine with no pivoting produces small residuals.

However, if the matrix to be factored is truly singular, the pivoting option is recommended (see Section 5.3.6).

The *QR* solver performs both a forward solve and a backsolve. The forward solve is the application of a sequence of (block) Householder transformations, and is stable (see reference 1 listed in Section 5.7). The backsolve is triangular; for infomation on its stability, see reference 8.

5.3.6 The Pivoting Option: Working with Ill-Conditioned Systems

To use the *QR* pivoting option, supply the value **CMSSL_column_pivoting** (or **CMSSL_column_pivoting_scale**; see Section 5.3.7) for the *pivoting_strategy* argument when you call **gen_qr_factor**. (See the man page at the end of this section for details about the calling sequence.)

Why Use Pivoting?

Pivoting is useful in the following ways:

- It allows you to determine the column rank of the matrix A accurately. In contrast, when you perform the factorization without pivoting, it is relatively easy to misjudge the column rank of A.
- It gives you more options for working with ill-conditioned matrices.

In the current release, using the *QR* factor and solve routines with pivoting is the recommended method for working with ill-conditioned matrices.

Determining the Column Rank of the Matrix *A*

This section describes the advantage of using pivoting in determining the column rank of the matrix *A*. Throughout this discussion, a *tiny* number is a number that is tiny *relative to the norm of the matrix A*.

A column of *A* that is dependent or close to dependent on the previous columns (indicating that *A* is ill-conditioned) will appear, during one of the elimination steps in the factorization process, as a column of zeros or tiny numbers. If **gen_qr_factor** encounters such a column and you have specified no pivoting, the routine either fails or places a zero or tiny number on the diagonal of the corresponding column of *R*. In fact, a zero or tiny number on the diagonal of *R always* means that the corresponding column of *A* was dependent (or almost dependent, respectively) on the previous columns. Thus, if *R* contains columns with zeros or tiny numbers on the diagonal, you can assume that *A* is singular or ill-conditioned.

Suppose one wants to determine the column rank of *R* (which equals the column rank of *A*, since *Q* is orthogonal). When counting the linearly independent columns of *R*, one strategy might be to discount any column with a zero or tiny number on the diagonal. But this strategy can be misleading. For example, consider the matrix

$$R = \begin{matrix} 1 & 1 & 1 & 1 & 2^{-1/2} \\ & e & e & e & 2^{-1/2} \\ & & e & e & e \\ & & & e & e \\ & & & & e \end{matrix}$$

where *e* is tiny. The values of *e* on the diagonal indicate that *A* (and *R*) are ill-conditioned. However, if you ignore the columns with *e* on the diagonal, you conclude that the matrix has column rank 1, whereas in fact, it has column rank 2 (the first and last columns are linearly independent).

In contrast, when you specify pivoting, each time **gen_qr_factor** processes a column, it examines the remaining columns and moves the one with the greatest vector 2-norm forward (to a lower column position) in the matrix. Therefore,

columns with zeros or tiny numbers end up in the last column positions of R. In the above example, if you had specified pivoting, R would be

$$R = \begin{matrix} 1 & 2^{-1/2} & 1 & 1 & 1 \\ & 2^{-1/2} & e & e & e \\ & & e & e & e \\ & & & e & e \\ & & & & e \end{matrix}$$

This time, you would discount the last three columns and correctly conclude that the column rank is 2.

It is important to note that if R has no zeros or tiny numbers on the diagonal, you *cannot* safely conclude that A is well-conditioned. For example, consider the matrix

$$R = \begin{matrix} 1 & 1 & 0 & 0 \\ & u & 1 & 0 \\ & & u & 1 \\ & & & u \end{matrix}$$

where e is tiny and $u = e^{1/2}$ (which is "large"). This matrix has no zeros or tiny numbers on the diagonal. However, its condition number is on the order of $1/u^3 = 1/(e^{3/2})$, which is large; thus, the matrix is ill-conditioned.

Strategies for Working with Ill-Conditioned Matrices

In most ill-conditioned problems, the dependent columns occur at the end of the matrix, so that pivoting gains you no special advantage. However, in most cases you do not know ahead of time whether this condition is true for a given matrix. Furthermore, in extremely ill-conditioned cases, **gen_qr_factor** without pivoting may fail altogether because of underflow when processing a dependent column. Therefore, pivoting is a safer strategy when working with matrices that may be ill-conditioned. However, since pivoting also exacts a performance cost, you may want to call the factorization routine without pivoting first, as in the following strategy:

1. Factor without pivoting:
    a. Call **gen_qr_factor** without pivoting.
    b. Call **gen_qr_infinity_norm_inv** to estimate the infinity norm of A^{-1}; call **gen_infinity_norm** to obtain the infinity norm of A; and thus find the condition number of A.

c. If A is well-conditioned, proceed with the solve routine. If A is ill-conditioned, you may still wish to call the solve routine, bearing in mind that your relative error will be large. Alternatively, try Step 2.

2. Factor with pivoting, if necessary.

   a. Call **gen_qr_factor** with pivoting.

   b. Call **gen_qr_extract_diag** to extract the block cyclic diagonal entries of R. If there are zeros or tiny numbers at the end of the block cyclic diagonal, A is ill-conditioned. (Remember that if there are no zeros or tiny entries at the end of the block cyclic diagonal, you cannot be sure the matrix is well-conditioned.)

   c. Change any tiny entries at the end of the block cyclic diagonal to zeros.

   d. Call **gen_qr_deposit_diag** to deposit the modified block cyclic diagonal entries back into R.

   e. Call **gen_qr_r_infinity_norm_inv**. This routine estimates the infinity norm of $(R^*)^{-1}$, where R^* is the upper-left corner of R formed by discarding any trailing columns of R that contain zeros on the block cyclic diagonal. Find the condition number of R^*. If R^* is ill-conditioned, you may wish to use **gen_qr_extract_diag** and **gen_qr_deposit_diag** to change the last block cyclic diagonal entries of R^* to zeros and then repeat this step. When you have finally discarded enough columns to obtain an R^* that is well-conditioned, you will know that you can solve the corresponding portion of your original problem (by discarding some portions of the right-hand side) with confidence.

5.3.7 Scaling

The *pivoting_strategy* argument of **gen_qr_factor** allows you to select scaling as well as pivoting. The values

CMSSL_column_pivoting_scale
CMSSL_no_pivoting_scale

have the same effects as

CMSSL_column_pivoting
CMSSL_no_pivoting

respectively, except that the first two select scaling while the second two do not.

If you select scaling, the **gen_qr_factor** routine uses a scaling factor to eliminate the possibility that $\|col\|^2$ yields underflow or overflow, where *col* is a column of A used in the elimination process. In particular, **gen_qr_factor** replaces $(\Sigma a_i^2)^{1/2}$ with $S(\Sigma(a_i/S)^2)^{1/2}$, where S is the scaling factor and a_i are the elements of *col*. The scaling factor S is defined by $(\|col\|_\infty)^{1/2} = (\max(col))^{1/2}$.

Scaling is not usually necessary; it is required only when $\|A\|^2$ is close to underflow or overflow, for any matrix A within A. (Note that underflow of $\|col\|^2$ does not cause a problem if *col* is a column with zeros or tiny numbers at the end of the block cyclic diagonal.) Because scaling involves a significant performance cost, especially in the case of pivoting, you should use it only when necessary.

5.3.8 Saving and Restoring the *QR* State

The *QR* factorization routine generates internal state variables required for computing the solution. These variables are not made available as arrays to user applications because their sizes and contents are CM configuration-dependent. However, it is sometimes desirable to save the internal state to a file for future use. The **save_gen_qr** and **restore_gen_qr** routines allow you to save and restore the internal *QR* state.

The *QR* routines allow you to have more than one factorization "active" at a time; for example, the sequence of calls

```
setup_X = gen_qr_factor(X, ...)
setup_Y = gen_qr_factor(Y, ...)
call gen_qr_solve(B_X, X, setup_X, ...)
call gen_qr_solve(B_Y, Y, setup_Y, ...)
```

is valid. You may, however, want to use **save_gen_qr** and **restore_gen_qr** to carry the internal state over between program runs.

It is not intended that the save and restore routines be used to conserve memory. The state variables are very small compared to the size of the typical matrix A.

Solving Linear Systems Using Householder Transformations

Given a CM array A containing one or more embedded coefficient matrices A, and a CM array B containing corresponding embedded right-hand sides B, the routines listed below use Householder transformations (with or without pivoting) to factor each A into two matrices, Q and R, described below; use the QR factors to solve the linear systems $AX = B$ or $A^TX = B$; and perform related operations.

SYNTAX

Factorization routine:

setup = **gen_qr_factor**	(*A, m, n, row_axis, col_axis, nblock, pivoting_strategy, back_solve_strategy, ier*)

Save and restore routines:

save_gen_qr	(*setup, unit, iostat, ier*)
setup = **restore_gen_qr**	(*A, m, n, row_axis, col_axis, nblock, pivoting_strategy, back_solve_strategy, unit, iostat, ier*)

Solver routines:

gen_qr_solve	(*B, A, setup, nrhs, ier*)
gen_qr_solve_tra	(*B, A, setup, nrhs, ier*)

Factor application routines:

gen_qr_apply_q	(*B, A, setup, nrhs, ier*)
gen_qr_apply_q_tra	(*B, A, setup, nrhs, ier*)
gen_qr_apply_r_inv	(*B, A, setup, nrhs, ier*)
gen_qr_apply_r_inv_tra	(*B, A, setup, nrhs, ier*)

Version 3.1, June 1993

Get-*R* routine:

gen_qr_get_r (*B, A, setup, ier*)

Pivot application routines:

gen_qr_apply_p (*B, A, setup, nrhs, ier*)

gen_qr_apply_p_inv (*B, A, setup, nrhs, ier*)

Zeroing routine:

gen_qr_zero_rows (*B, A, setup, limit, nrhs, ier*)

Diagonal manipulation routines:

gen_qr_extract_diag (*d, A, setup, ier*)

gen_qr_deposit_diag (*A, d, setup, ier*)

Infinity norm routines:

gen_qr_infinity_norm_inv (*a, A, setup, ier*)

gen_qr_r_infinity_norm_inv (*a, A, setup, ier*)

Deallocation routine:

deallocate_gen_qr (*setup*)

ARGUMENTS

In the descriptions below, *A* and *B* refer to the active matrices with which the routines work. These matrices may be contained (as the upper left-hand submatrices) in larger matrices within the arrays *A* and *B*, respectively. Details are provided below.

Also, throughout these descriptions, $\overline{Q}$ denotes the conjugate of Q, and the notation M^{-T} is used for $(M^{-1})^T = (M^T)^{-1}$.

setup Scalar integer variable. Setup ID returned by **gen_qr_factor** and **restore_gen_qr**. When you call any of the other *QR* routines, you

must supply the value returned by the corresponding **gen_qr_factor** or **restore_gen_qr** call.

B CM array of the same data type as *A*. The instance axes of *B* must match those of *A* in order of declaration and extents. When you call **gen_qr_get_r**, *A* and *B* must have the same rank, axis extents, and layout directives. *B* must be distinct from *A*.

Solver, Factor Application, Pivot Application, and Zeroing Routines. When you call one of the *QR* solver, factor application, pivot application, or zeroing routines, *B* must contain one or more instances of *B*, where each *B* consists of one or more right-hand-side vectors. The following restrictions hold:

- If each instance *B* within *B* consists of only one right-hand-side vector (*nrhs* = 1), you may represent *B* in either of the following ways:
  - It may have rank 2 with number of columns = 1. In this case, each *B* has dimensions *m* x 1 (and may consist of the upper left-hand *m* x 1 elements of a larger matrix). The rows of each *B* must be counted by axis *row_axis* (from the **gen_qr_factor** call); the single column must be counted by axis *col_axis*.
  - It may have rank 1. In this case, each *B* has dimension *m* (and may consist of the first *m* elements of a larger vector). The elements of each *B* must be counted by axis *row_axis* (if *row_axis* < *col_axis*) or by axis (*row_axis* - 1) (if *col_axis* < *row_axis*). For an example, see the Notes section.
- If each *B* within *B* consists of multiple right-hand-side vectors (*nrhs* > 1), then each *B* has dimensions *m* x *nrhs*, and may consist of the upper left-hand *m* x *nrhs* elements of a larger matrix. The rows and columns of *B* must be counted by axes *row_axis* and *col_axis*, respectively.

Upon successful completion of **gen_qr_solve**, the first *n* rows of each matrix *B* are overwritten with the least squares solution to *AX* = *B*. The remaining *m* - *n* rows of *B* are undefined.

With $m > n$, the system $A^TX = B$ is underdetermined. Upon successful completion of **gen_qr_solve_tra**, each $B[1{:}m]$ is overwritten with the minimal 2-norm solution to this underdetermined system.

Upon completion of a factor application routine, each *B* within ***B*** is overwritten by the product QB, $\overline{Q}B$, Q^TB, $\overline{Q}^TB$, $R^{-1}B$, or $R^{-T}B$.

Upon completion of a pivot application routine, each *B* within ***B*** is overwritten by the product PB or $P^{-1}B$, where P is the permutation matrix that corresponds to the pivoting process.

The **gen_qr_zero_rows** routine zeroes the last m - *limit* block cyclic rows of each two-dimensional matrix defined by axes *row_axis* and *col_axis* of ***B***.

Get-*R* Routine. When you call **gen_qr_get_r**, ***B*** must have the same rank, axis extents, layout directives, and data type as ***A***. Upon completion, each m x n matrix *B* within ***B*** contains the factor *R* of the corresponding matrix *A* within ***A***. (*R* is a block cyclic upper triangle, as described below.) The rows and columns of each *B* are represented by *row_axis* and *col_axis*, respectively. These axes may have extents greater than m and n, respectively; that is, each *B* may be contained (as the upper left-hand m x *nrhs* elements) in a larger matrix within ***B***.

d CM array of the same rank and type as ***A***. Contains one or more instances of a vector of length greater than or equal to n; these vectors must lie along axis *row_axis*. Axis *col_axis* must have extent 1. All remaining (instance) axes of ***d*** must match, in order of declaration and extents, the instance axes of ***A***. Thus, each matrix *A* embedded in ***A*** corresponds to a vector embedded in ***d***.

Upon return from **gen_qr_extract_diag**, the first n elements of each vector within ***d*** are the block cyclic diagonal entries of the *R* factor of the corresponding *A* within ***A***.

When you call **gen_qr_deposit_diag**, you must supply in the first n elements of each vector within ***ds*** the values you wish to deposit into the block cyclic diagonal of the *R* factor of the corresponding *A* within ***A***.

a Real CM array with the same rank and precision as ***A***. Axes *row_axis* and *col_axis* must have extent 1. Thus, each matrix *A* embedded in ***A*** corresponds to a real number in ***a***.

Upon successful completion of **gen_qr_infinity_norm_inv**, the estimated infinity norm of the *inverse* of each matrix *A* within *A* is placed in the corresponding position of ***a***.

Upon successful completion of **gen_qr_r_infinity_norm_inv**, the estimated infinity norm of the *inverse* of each *R** within *A* is placed in the corresponding position of ***a***. The supplied *A* contains the *R* factors of the matrices *A*, returned by **gen_qr_ factor**. *R** is the block cyclic upper-left corner of *R* formed by discarding any trailing columns of *R* that contain zeros on the block cyclic diagonal.

A

Real or complex CM array of rank greater than or equal to 2. Must be distinct from ***B***.

Factor Routine. When you call **gen_qr_factor**, *A* should contain one or more instances of a coefficient matrix *A* to be factored. Each *A* is assumed to be dense with dimensions *m* x *n*, with rows counted by axis *row_axis* and columns counted by axis *col_axis*. These axes may have extents greater than *m* and *n*, respectively; that is, each *A* may be contained (as the upper left-hand *m* x *n* elements) in a larger matrix within *A*. Upon successful completion of **gen_qr_factor**, the block cyclic upper triangle of *A* is overwritten by *R*. The remaining elements of *A* are used internally to reconstruct *Q*.

All Other Routines. When you call any of the other *QR* routines, *A* must have the same data type, rank, and shape (axis extents and layout directives, including orderings and weights) as the original *A* that was factored. You must also be using the same partition size as when you originally factored *A*. Supply in *A* the *QR* factors returned in *A* by **gen_qr_factor**.

m

Scalar integer variable. The number of rows in each matrix *A* embedded in *A*. Must be greater than or equal to *n*.

If you intend to call **gen_qr_infinity_norm_inv**, *m* must equal *n*, since each matrix *A* within *A* must be invertible, and therefore square.

n

Scalar integer variable. The number of columns in each matrix *A* embedded in *A*. Must be less than or equal to *m*.

If you intend to call **gen_qr_infinity_norm_inv**, *m* must equal *n*, since each matrix *A* within *A* must be invertible, and therefore square.

nrhs Scalar integer variable. The number of columns of each right-hand side *B* within ***B***. Must be greater than or equal to 1.

row_axis Scalar integer variable. The axis that counts the rows of the matrices *A* embedded in ***A***. The extent of this axis must be at least *m*; *row_axis* must be in the range 1 through the rank of ***A***, inclusive; and *row_axis* and *col_axis* must not be equal.

col_axis Scalar integer variable. The axis that counts the columns of the matrices *A* embedded in ***A***. The extent of this axis must be at least *n*; *col_axis* must be in the range 1 through the rank of ***A***, inclusive; and *row_axis* and *col_axis* must not be equal.

nblock Scalar integer variable. Blocking factor. If you specify pivoting (see *pivoting_strategy*, below), you must supply 1 for *nblock*. Otherwise, use these guidelines when choosing an *nblock* value:

- For typical applications, *nblock* = 8 is a good choice. An *nblock* value of 16 may yield faster factorization in some cases.
- *nblock* should always be less than or equal to *n*; *nblock* values > *n* use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*. If that is not possible, choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions.

pivoting_strategy Scalar integer variable specifying the pivoting strategy to be used. Specify one of the following values:

CMSSL_no_pivoting	No pivoting, no scaling
CMSSL_no_pivoting_scale	No pivoting, scaling
CMSSL_column_pivoting	Column pivoting, no scaling
CMSSL_column_pivoting_scale	Column pivoting, scaling

For a discussion of pivoting, see Section 5.3.6. For information about scaling, see the Notes section below.

back_solve_strategy

Scalar integer variable. Specifies the back substitution strategy. A value of **CMSSL_qr_post_permute** is always acceptable; the value **CMSSL_qr_pre_permute** is used to enhance performance in special cases, as described in Section 5.3.4.

A value of **CMSSL_qr_post_permute** indicates that the rows of the solution are to be permuted by the **gen_qr_solve** routine after the backsolve is completed. A value of **CMSSL_qr_pre_permute** specifies that the columns of the matrices *A* are to be permuted by **gen_qr_factor** prior to the factorization.

CMSSL_qr_pre_permute requires that $m = n$, and that *pivoting_strategy* = **CMSSL_no_pivoting**.

limit

Scalar integer variable. Must be in the range from 1 through m. Determines how many rows within each two-dimensional matrix defined by *row_axis* and *col_axis* of ***B*** will be changed to 0 by **gen_qr_zero_rows**. This routine zeroes the last m - *limit* block cyclic rows of each two-dimensional matrix defined by axes *row_axis* and *col_axis* of ***B***.

unit

Scalar integer. Valid unit number associated with the file to or from which the *QR* state is to be written or read. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a device or socket with a unit number). The **save_gen_qr** and **restore_gen_qr** calls write and read data using **CMF_CM_ARRAY_TO_FILE_SO** and **CMF_CM_ARRAY_FROM_FILE_SO**, respectively. You must rewind the file before calling **restore_gen_qr**.

iostat

Scalar integer variable. Upon return, contains the status of the I/O operation. If *ier* = 0, *iostat* contains the number of bytes written or read. For the meanings of other *iostat* codes, refer to the descriptions of **CMF_CM_ARRAY_TO_FILE_SO** (for **save_gen_qr**) and **CMF_CM_ARRAY_FROM_FILE_SO** (for **restore_gen_qr**) in the CM Fortran documentation set.

ier

Scalar integer variable. Return code; set to 0 upon successful return.

Values between -1 and -9, inclusive, indicate problems with one or more of the CM arrays containing matrices in any of the *QR* calls:

-1 Invalid array home. The array must be a CM array.
-2 Invalid rank; must be ≥ 2.
-3 Invalid column extent; must be ≥ *m*.
-4 Invalid row extent; must be ≥ *n*.
-9 Invalid data type; must be real or complex (single- or double-precision).

Values that are multiples of -10 indicate problems with non-array arguments:

-10 System failed to allocate the setup object, *setup*.
-20 *m*, *n*, or *nrhs* is invalid; all must be > 0 and *m* must be greater than or equal to *n*.
-30 *row_axis* or *col_axis* is invalid. 1 ≤ *row_axis*, *col_axis* ≤ rank (*A*) must be true, and *row_axis* and *col_axis* must not be equal.
-40 *nblock* is invalid. It must be greater than or equal to 1, or equal to 1 if you specify **CMSSL_column_pivoting**.
-50 *pivoting_strategy* is invalid; must be **CMSSL_column_pivoting**, **CMSSL_no_pivoting**, **CMSSL_column_pivoting_scale**, or **CMSSL_no_pivoting_scale**.
-60 *nrhs* is invalid.
-70 *back_solve_strategy* is invalid; must be **CMSSL_pre_permute** or **CMSSL_post_permute**.
-80 You specified an invalid combination of *pivoting_strategy*, *back_solve_strategy*, and/or *m* not equal to *n*; or you specified *m* not equal to *n* in the factorization call associated with this call to **gen_qr_infinity_norm_inv**.
-100 *setup* is invalid. (You did not supply the value returned by **gen_qr_factor**.)

Values between -102 and -108, inclusive, indicate problems with the consistency of *A* or *B* in one of the *QR* routines following a factorization call:

-102 The rank of *A* or *B* is invalid (must be ≥ 2 for *A* or ≥ 1 for *B*), or is inconsistent with the rank of *A* in the factorization call.
-105 The extents of the instance axes of *A* or *B* are inconsistent with those of *A* in the factorization call.
-106 *B* must have the same layout directives as *A* when

you call **gen_qr_get_r**.

-108 The data type of A or B is inconsistent with that of A in the factorization call.

The **save_gen_qr** and **restore_gen_qr** routines return the following value if they encounter an I/O error:

-200 I/O error. See the value of *iostat* for more information.

DESCRIPTION

Given a CM array A containing one or more instances of a coefficient matrix A, and a CM array B containing corresponding instances of a right-hand-side B, the following routines and operations are provided:

Factorization routine:

gen_qr_factor Uses Householder transformations to factor each matrix instance A into two matrices, Q and R, (or, if pivoting is specified, three matrices, Q, R, and P^{-1}), described below.

Save and restore routines:

save_gen_qr Saves internal information about the QR factors in a file.

restore_gen_qr Loads internal information about the QR factors from a file.

Solver routines:

gen_qr_solve Uses the QR factors returned by **gen_qr_factor** to solve the system(s) $AX = B$.

gen_qr_solve_tra Uses the QR factors returned by **gen_qr_factor** to solve the system(s) $A^{T}X = B$.

Factor application routines:

gen_qr_apply_q Given the QR factors returned by **gen_qr_factor**, applies Q (or $\overline{Q}$, in the case of complex data) to B for each instance.

gen_qr_apply_q_tra Given the *QR* factors returned by **gen_qr_factor**, applies Q^T (or Q^H, in the case of complex data) to *B* for each instance. Note that since *Q* is orthogonal (or unitary, in the complex case), $Q^H = Q^{-1}$.

gen_qr_apply_r_inv Given the *QR* factors returned by **gen_qr_factor**, applies R^{-1} to *B* for each instance.

gen_qr_apply_r_inv_tra Given the *QR* factors returned by **gen_qr_factor**, applies R^{-T} to *B* for each instance.

Get-*R* routine:

gen_qr_get_r Given the *QR* factors returned by **gen_qr_factor**, produces the factor *R* for each instance.

Pivot application routines:

gen_qr_apply_p Given the *QR* factors returned by **gen_qr_factor**, applies *P* to *B* for each instance, where *P* is the permutation matrix that corresponds to the pivoting process. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

gen_qr_apply_p_inv Given the *QR* factors returned by **gen_qr_factor**, applies $P^{-1} = P^T$ to *B* for each instance. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

Zeroing routine:

gen_qr_zero_rows Zeroes the last *m* - *limit* block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of *B*.

Diagonal manipulation routines:

gen_qr_extract_diag Given the *QR* factors returned by **gen_qr_factor**, returns the block cyclic diagonal entries of *R* for each instance.

gen_qr_deposit_diag	Given the *QR* factors returned by **gen_qr_factor**, overwrites the block cyclic diagonal entries of each instance of *R* with values you supply.

Infinity norm routines:

gen_qr_infinity_norm_inv	Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of each matrix A^{-1}.
gen_qr_r_infinity_norm_inv	Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of each $(R^*)^{-1}$, where R^* is the block cyclic upper-left corner formed by discarding any trailing columns of *R* that contain zeros on the block cyclic diagonal.

Deallocation routine:

deallocate_gen_qr	Deallocates the processing element memory allocated by the factorization routine.

Memory Allocation and Deallocation. You must call either **gen_qr_factor** or **restore_gen_qr** before calling **save_gen_qr**, the get-*R* routine, or a solver routine, factor application, pivot application, zeroing, diagonal manipulation, or infinity norm routine. You can follow one call to **gen_qr_factor** or **restore_gen_qr** with multiple calls to these other routines, thus avoiding the overhead of factoring the same matrices repeatedly.

The **deallocate_gen_qr** routine deallocates the processing element memory allocated by the factorization routine and required by the other *QR* routines. Be sure to call **deallocate_gen_qr** when you have finished working with a set of *QR* factors.

You can work with more than one set of *QR* factors at a time by calling **gen_qr_factor** or **restore_gen_qr** more than once without calling **deallocate_gen_qr**. However, repeated calls to **gen_qr_factor** or **restore_gen_qr** without deallocation can cause you to run out of memory.

Factorization Routine. The **gen_qr_factor** routine uses Householder transformations to factor each matrix *A* embedded in *A*. If you specify **CMSSL_no_pivoting** in the *pivoting_strategy* argument, each *A* is factored into two matrices:

$$A = QR$$

If you specify **CMSSL_column_pivoting**, each A is factored into three matrices:

$$A = QRP^{-1}$$

The factors are defined in the section called **The *QR* Factors Defined**, below. Upon completion of **gen_qr_factor**, the block cyclic upper triangle of A is overwritten by R. The remaining elements of A are used internally to reconstruct Q.

When you call the get-R routine or a solver, factor application, pivot application, zeroing, diagonal manipulation, or infinity norm routine, you must supply the same A that was returned by **gen_qr_factor**.

Save and Restore Routines. You may save internal information about the *QR* factors in a file for use in later calls to the other *QR* routines. To save the *QR* information, call **save_gen_qr** after the factorization is complete but before deallocating the storage space. To restore the *QR* information, rewind the file and call **restore_gen_qr**; this call is typically followed by calls to the other *QR* routines.

Solver Routine. Given the values returned in A by **gen_qr_factor**, the **gen_qr_ solve** routine solves one or more instances of the system

$$AX = B$$

where A and B are corresponding instances within A and B, respectively. If the size of each A is (m x n), and the size of each B is (m x $nrhs$), then upon successful return from **gen_qr_solve**, the first n rows of each B are overwritten with the least squares solution to $AX = B$. The remaining $m - n$ rows of B are undefined.

Steps Performed by Solver Routine. If you specified no pivoting, since $A = QR$, $AX = B$ is equivalent to $X = R^{-1}Q^{T}B$. Therefore, to solve $AX = B$, the **gen_qr_solve** routine performs the following steps:

1. Apply Q^{T} to B.
2. Apply R^{-1} to $Q^{T}B$.

To perform these steps yourself, you would

1. Call **gen_qr_apply_q_tra** to apply each Q^{T} to the corresponding right-hand side, B.
2. Call **gen_qr_apply_r_inv** to apply R^{-1} to the result from Step 1.

If you specified pivoting, since $A = QRP^{-1}$, $AX = B$ is equivalent to $X = PR^{-1}Q^{T}B$. Therefore, to solve $AX = B$, the **gen_qr_solve** routine performs the following steps:

1. Apply Q^{T} to B.
2. Apply R^{-1} to $Q^{T}B$.
3. Apply P to $R^{-1}Q^{T}B$.

To perform these steps yourself, you would

1. Call **gen_qr_apply_q_tra** to apply each Q^{T} to the corresponding right-hand side, B.
2. Call **gen_qr_apply_r_inv** to apply R^{-1} to the result from Step 1.
3. Call **gen_qr_apply_p** to apply P to the result from Step 2.

Transpose Solver Routine. The **gen_qr_solve_tra** routine solves one or more instances of the system

$$A^{T}X = B$$

where A and B are corresponding instances within A and B, respectively. Specifically, the first n elements of a column of B give the right-hand sides for a system

$$A^{T}X[1{:}m] = B[1{:}n]$$

With $m > n$, this is an underdetermined system. Upon completion of **gen_qr_solve_tra**, each $B[1{:}m]$ is overwritten with the minimal 2-norm solution (not to be confused with the least squares solution) to this underdetermined system.

Steps Performed by Transpose Solver Routine. If you specified no pivoting, since $A = QR$, $A^{T}X = B$ is equivalent to $X = QR^{-T}B$. Therefore, to solve $A^{T}X = B$, the **gen_qr_solve** routine performs the following steps:

1. Apply R^{-T} to B.
2. Apply Q to $R^{-T}B$.

To perform these steps yourself, you would

1. Call **gen_qr_apply_r_inv** to apply each R^{-T} to the corresponding right-hand side, B.

2. Call **gen_qr_zero_rows** to zero the last $m - n$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of the result from Step 1. (Note: Step 2 is not required if the last $m - n$ block cyclic rows of each B were set to zero prior to Step 1.)

3. Call **gen_qr_apply_q** to apply Q to the result from Step 2.

Step 2 is required so that inactive data in the last $m - n$ rows of the right-hand-sides B does not affect the solution. This zeroing is required only when you are solving $A^TX = B$, not when you are solving $AX = B$.

If you specified pivoting, since $A = QRP^{-1}$, $A^TX = B$ is equivalent to $X = QR^{-T}P^{-1}B$. Therefore, to solve $A^TX = B$, the **gen_qr_solve_tra** routine performs the following steps:

1. Apply P^{-1} to B.

2. Apply R^{-T} to $P^{-1}B$.

3. Apply Q to $R^{-T}P^{-1}B$.

To perform these steps yourself, you would

1. Call **gen_qr_apply_p_inv** to apply each P^{-1} to the corresponding right-hand side, B.

2. Call **gen_qr_apply_r_inv_tra** to apply R^{-T} to the result from Step 1.

3. Call **gen_qr_zero_rows** to zero the last $m - n$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of the result from Step 2. (Note: Step 3 is not required if the last $m - n$ block cyclic rows of each B were set to zero prior to Step 1.)

4. Call **gen_qr_apply_q** to apply Q to the result from Step 3.

Factor Application Routines. The **gen_qr_apply_q, gen_qr_apply_q_tra, gen_qr_apply_r_inv, and gen_qr_apply_r_inv_tra** routines allow you to apply matrices derived from the QR factors to arbitrary matrices or vectors B contained in $\boldsymbol{B}$. Upon completion of the routine, each B in $\boldsymbol{B}$ is overwritten with the specified product (QB, $\overline{Q}B$, Q^TB, $\overline{Q}^TB$, $R^{-1}B$, or $R^{-T}B$). Thus, these routines use the factors produced by the QR factorization routine to solve triangular systems of the form $RX=B$ and $R^TX=B$ and trapezoidal systems of the form $QX=B$ or $Q^TX=B$.

To apply R to an arbitrary matrix or vector B, use the **gen_qr_get_r** routine to obtain R, and then perform the multiplication explicitly. To apply R^T to an arbitrary matrix or

vector B, either transpose R to obtain R^T, or use the fact that $R^TB = (B^TR)^T$; thus, apply B^T to R and transpose the result. If B is a vector, transposing B and B^TR (which is also a vector) is much less costly than transposing R would be.

Get-*R* Routine. The **gen_qr_get_r** routine provides access to the R factors of the coefficient matrices A. Upon completion, each B within $\boldsymbol{B}$ contains the factor R of the corresponding coefficient matrix A within $\boldsymbol{A}$. (Note that R is a block cyclic upper triangle.) The rows and columns of each B are represented by the same axes that defined the rows and columns of the matrices A within $\boldsymbol{A}$ in the **gen_qr_factor** call.

Pivot Application Routines. The **gen_qr_apply_p** and **gen_qr_apply_p_inv** routines allow you to apply the permutation matrix P that corresponds to the pivoting process, and its transpose $P^T = P^{-1}$, to arbitrary matrices or vectors B contained in $\boldsymbol{B}$. Upon completion of **gen_qr_apply_p**, each B within $\boldsymbol{B}$ is overwritten by the product PB. Upon completion of **gen_qr_apply_p_inv**, each B within $\boldsymbol{B}$ is overwritten by the product $P^{-1}B$. These routines are useful if you want to perform separately the permutations that the solver routines perform when pivoting is specified, as described above. Use these routines only if you specified pivoting in the associated call to **gen_qr_factor**.

Zeroing Routines. The **gen_qr_zero_rows** routine zeroes the last m - *limit* block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of $\boldsymbol{B}$. This routine is useful if you want to perform separately the zeroing that the transpose solver routine performs, as described above.

Extract and Deposit Diagonal Routines. The **gen_qr_extract_diag** routine returns in $\boldsymbol{d}$ the block cyclic diagonal entries of the factor R of each matrix A within $\boldsymbol{A}$. The **gen_qr_deposit_diag** routine overwrites the block cyclic diagonal entries of each R with values you supply in $\boldsymbol{d}$. These routines are useful in working with matrices that may be ill-conditioned.

Infinity Norm Routines. Given the QR factors returned by **gen_qr_factor**, the **gen_qr_infinity_norm_inv** routine estimates the infinity norm of each matrix A^{-1}. Upon successful completion of **gen_qr_infinity_norm_inv**, the infinity norm of each A^{-1} is placed in the position of $\boldsymbol{a}$ corresponding to A.

The **gen_qr_r_infinity_norm_inv** routine estimates the infinity norm of each $(R^*)^{-1}$, where R^* is the block cyclic upper-left corner of R formed by discarding any trailing columns of R that contain zeros on the block cyclic diagonal. This routine is useful in working with matrices that may be ill-conditioned. Upon successful completion of **gen_qr_r_infinity_norm_inv**, the infinity norm of each $(R^*)^{-1}$ is placed in the position of $\boldsymbol{a}$ corresponding to the matrix A of which R is a factor.

The infinity norm of a matrix M, denoted here by $\| M \|_\infty$, is defined by

$$\| M \|_\infty = \max_{\| x \|_\infty = 1} \| Mx \|_\infty$$

where the infinity norm of a vector, $\| x \|_\infty$, is defined as the maximum of the absolute values of the vector components:

$$\| x \|_\infty = \max_i | x_i |$$

The infinity-norm condition number of a matrix M is equal to the product of $\| M \|_\infty$ and $\| M^{-1} \|_\infty$.

The *QR* Factors Defined (Square Case). The following definitions apply to the square case ($m = n$). For information about the non-square case, see Section 5.3.4. Effectively, the **gen_qr_factor** routine factors a block cyclic permutation, A_c, of each matrix A that you supply in A. In a factorization with pivoting, the matrix A_c is factored into

$$A_c = Q_c R_c P_c^{-1}$$

where R_c is upper triangular, Q_c is orthogonal (or unitary, in the complex case), and P_c is the permutation matrix resulting from the pivoting process. In a factorization without pivoting, the factorization is

$$A_c = Q_c R_c$$

where R_c is upper triangular and Q_c is orthogonal (or unitary).

The definitions of Q and R in terms of A_c and its factors depend on the value you supply in the **gen_qr_factor** *back_solve_strategy* argument. The two possible values are **CMSSL_qr_post_permute** and **CMSSL_qr_pre_permute**. The factor definitions are provided below for the square case ($m = n$). Note that the **CMSSL_qr_pre_permute** strategy does not work with pivoting, and requires that $m = n$.

- *Case 1: Post-permutation; no pivoting; $m = n$*

  By definition,

  $$A_c = P_1^{-1} A P_2$$

  where P_1 is the permutation giving the correspondence between standard and block cyclic row order, and P_2 is the permutation giving the correspondence

between standard and block cyclic column order. (These permutations depend on the array size and layout, the partition size, and the blocking factor you supply.) We therefore have

$$\begin{aligned} A = \quad QR &= P_1 A_c P_2^{-1} \\ &= P_1 (Q_c R_c) P_2^{-1} \\ &= P_1 Q_c (P_1^{-1} P_1) R_c P_2^{-1} \end{aligned}$$

from which we choose to define

$$Q = P_1 Q_c P_1^{-1}$$
$$R = P_1 R_c P_2^{-1}$$

- *Case 2: Post-permutation with pivoting; m = n*

  This case is just like the Case 1 except that we include P, the permutation matrix that corresponds to the pivoting process. We have

$$\begin{aligned} A = \quad QRP^{-1} &= P_1 A_c P_2^{-1} \\ &= P_1 (Q_c R_c P_c^{-1}) P_2^{-1} \\ &= P_1 Q_c (P_1^{-1} P_1) R_c (P_2^{-1} P_2) P_c^{-1} P_2^{-1} \\ &= (P_1 Q_c P_1^{-1}) (P_1 R_c P_2^{-1}) (P_2 P_c^{-1} P_2^{-1}) \end{aligned}$$

  from which we choose to define

$$Q = P_1 Q_c P_1^{-1}$$
$$R = P_1 R_c P_2^{-1}$$
$$P^{-1} = P_2 P_c^{-1} P_2^{-1}$$

- *Case 3: Pre-permutation; no pivoting; m = n*

  In this case, the factorization routine pre-permutes the matrix A to obtain

$$A^* = A P_1 P_2^{-1}$$

  By definition, we have

$$A_c = P_1^{-1} A^* P_2$$

  from which it follows that

$$A_c = P_1^{-1} A P_1$$

  and therefore

$$\begin{aligned} A = QR &= P_1 A_c P_1^{-1} \\ &= P_1 Q_c R_c P_1^{-1} \\ &= P_1 Q_c (P_1^{-1} P_1) R_c P_1^{-1} \end{aligned}$$

  from which we choose to define

$$Q = P_1 Q_c P_1^{-1}$$
$$R = P_1 R_c P_1^{-1}$$

In the square case, the **gen_lu_get_r** routine returns the R factors defined above. The matrices R^{-1} and R^{-T}, Q, and Q^T applied by the factor application routines are derived from the Q and R factors defined above, and the inverses are true inverses; that is, $R^{-1}R = RR^{-1} = I = Q^TQ = QQ^T$. (The inverses are also true in the block cyclic space; that is, $R_c^{-1}R_c = R_cR_c^{-1} = I = Q_c^TQ_c = Q_cQ_c^T$.)

The definitions above generalize to the non-square case ($m > n$) using the same principles.

NOTES

NaNs and Infinities. As mentioned above, the matrices A and B may be contained (as the upper left-hand submatrices) in larger matrices within the arrays A and B, respectively. In this case, if there are NaNs or infinities in the larger matrix outside of A or B, it is possible that other locations outside of A or B could become NaNs or infinities as well.

Distinct Variables. The input CM arrays A and B must be distinct variables.

Include the CMSSL Header File. The **gen_qr_factor** routine uses symbolic constants. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls this routine. This file declares the types of the CMSSL functions and symbolic constants.

Saving and Restoring the *QR* State. If you want to save the internal state in one program run and restore it in a different run, you must save the array of factored matrices in a file in addition to saving the internal state using **save_gen_qr**. Be sure to save the array in a different file than that used for saving the state. When you read the array back into memory prior to restoring the internal state, you must use the same partition size as when you originally performed the factorization; and the restored array must have exactly the same shape (axis extents and layout directives, including orderings and weights) as when you saved it.

Nondegeneracy Required. Each matrix A within A must have a column space of rank n when you call one of the *QR* solver routines without pivoting.

Rank of *B*. The following example illustrates the options for defining the rank of ***B***. Suppose ***A***, *n*, *m*, *row_axis*, and *col_axis* are defined as follows:

A (5, 10, 5)
m = *n* = 5
row_axis = 1
col_axis = 3

and each ***B*** in ***B*** is a single vector. You may define ***B*** in either of the two following (equivalent) ways:

B (5, 10, 1)
B (5, 10)

On the other hand, if you define

A (5, 10, 5)
m = *n* = 5
row_axis = 3
col_axis = 1

then the possibilities for ***B*** are as follows:

B (1, 10, 5)
B (10, 5)

Performance. Performance improves for larger subgrid sizes (and therefore depends upon the layout of ***A***). For information on subgrids, refer to the CM Fortran documentation set.

To optimize performance, follow these guidelines:

- Ensure that the subgrid length in each dimension is a multiple of *nblock*. If that is not possible, choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions.
- Lay out ***A*** so that the subgrid sizes along axes *row_axis* and *col_axis* differ from one another by no more than a factor of 4 or 5.
- Use axis extents exactly equal to *m* x *n* for the matrices ***A*** and *m* x *nrhs* for the matrices ***B***. Use the same processing element layout for the arrays ***A*** and ***B***.

Scaling. The *pivoting_strategy* values **CMSSL_column_pivoting_scale** and **CMSSL_no_pivoting_scale** have the same effects as **CMSSL_column_pivoting** and **CMSSL_no_**

pivoting, respectively, except that the first two select scaling while the second two do not.

If you select scaling, **gen_lu_factor** uses a scaling factor to eliminate the possibility that $||col||^2$ yields underflow or overflow, where *col* is a column of *A* used in the elimination process. In particular, **gen_lu_factor** replaces $(\Sigma a_i^2)^{1/2}$ with $S(\Sigma(a_i/S)^2)^{1/2}$, where *S* is the scaling factor and a_i are the elements of *col*. The scaling factor *S* is defined by $(||col||_\infty)^{1/2} = (\max(col))^{1/2}$.

Scaling is not usually necessary; it is required only when $||A||^2$ is close to underflow or overflow, for any matrix *A* within *A*. (Note that underflow of $||col||^2$ does not cause a problem if *col* is a column with zeros or tiny numbers at the end of the block cyclic diagonal.) Because scaling involves a significant performance cost, especially in the case of pivoting, you should use it only when necessary.

Numerical Complexity. If the matrices *A* have dimensions (m x n), the matrices *B* have *nrhs* right-hand sides, and *I* is the number of instances (the product of all axis extents except axes *row_axis* and *col_axis*), then:

- The *QR* factorization routine requires approximately $2n^2(m - n/3)I$ floating-point operations for real operands and $8n^2(m - n/3)I$ floating-point operations for complex operands.
- The *QR* solver routines require approximately $nrhs * n(4m - n)I$ floating-point operations for real operands and $4nrhs * n(4m - n)I$ floating-point operations for complex operands.

Performance Cost of Pivoting. The factorization routine is approximately twice as slow with pivoting than without. (Almost half the performance cost results from the fact that pivoting requires a block size of 1.) These performance figures are closely tied to the current implementation, and may change in future releases.

EXAMPLES

Sample CM Fortran code that uses the routines described above can be found on-line in the subdirectories

```
householder/cmf/
```

and

```
infinity-norm/cmf/
```

of a CMSSL examples directory whose location is site-specific.

5.4 Matrix Inversion and the Gauss-Jordan System Solver

The matrix inversion routine, **gen_gj_invert**, and the Gauss-Jordan solver routine, **gen_gj_solve**, both use the same variant of the Gauss-Jordan algorithm.

The Gauss-Jordan algorithm requires pivoting if the system is not symmetric positive definite. The **gen_gj_invert** and **gen_gj_solve** routines support two pivoting strategies (partial and total pivoting), described in the context of the inversion routine, below.

5.4.1 Matrix Inversion

Conceptually, the inversion procedure progressively transforms the original matrix A into the identity matrix, I, while progressively transforming the identity matrix into the solution — the inverse of the original matrix, A^{-1}. Figure 19 shows a simplified view of this process. It ignores the details that are introduced by permuting rows and columns and by inverting A in place.

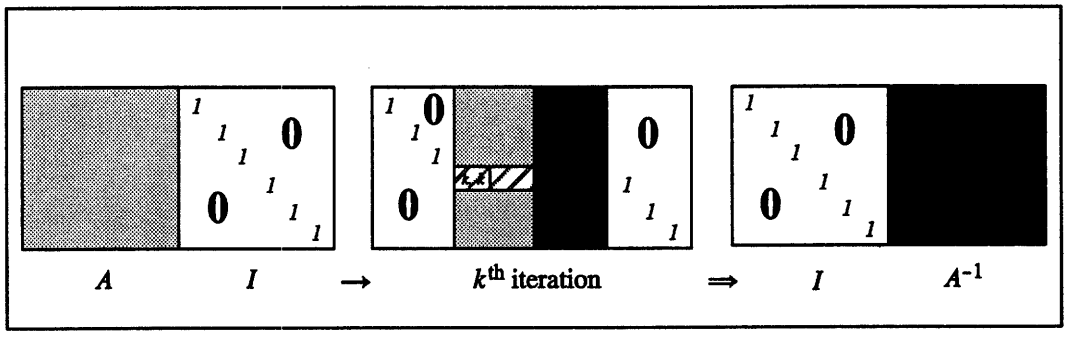

Figure 19. Matrix A becomes I while I becomes A^{-1}.

The pivoting strategy you specify when you call the inversion routine determines the size of the search space for the pivot, as follows:

- If you choose partial pivoting, the pivot element is chosen from the pivot row, and columns are (in effect) permuted. (This is a variant of the conventional partial pivoting method, in which the pivot element is chosen from the pivot column.)

- If you choose total pivoting, the pivot element is chosen from a submatrix and both rows and columns are permuted.

These strategies are illustrated in the next two figures. As in Figure 19, row and column permutations and in-place inversion are ignored.

With partial pivoting, the pivot search is conducted along the pivot row. Figure 20 shows the k^{th} iteration. The pivot search is conducted along the k^{th} row; previous iterations have begun to replace the principal diagonal of A with 1s and successive columns with 0s. The partial pivot search determines the maximum value for row k.

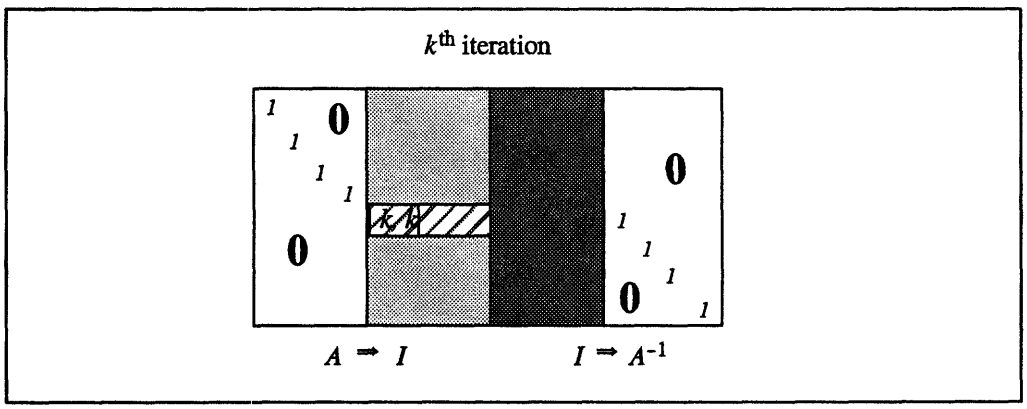

Figure 20. Partial pivoting searches pivot row of *A*.

With total pivoting, the pivot search is conducted within the submatrix below and to the right of the pivot element, inclusive. Figure 21 illustrates this case.

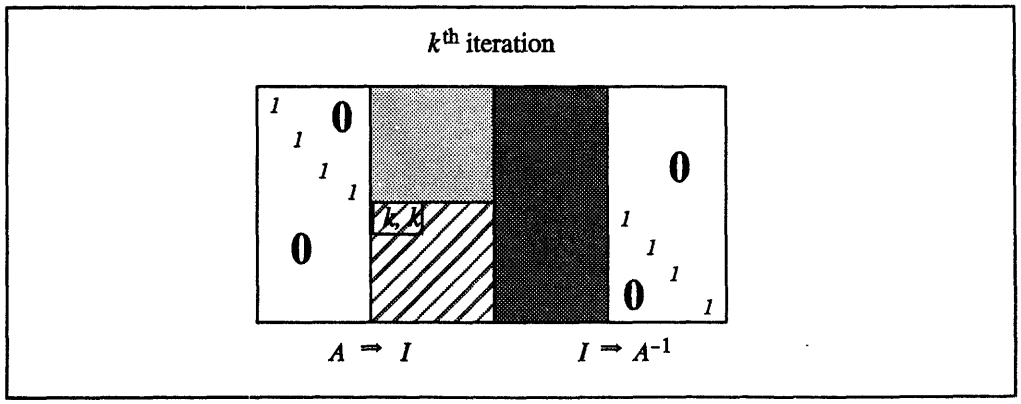

Figure 21. Total pivoting searches lower right-hand submatrix of *A*.

The total pivoting strategy is numerically more stable but slower than the partial pivoting strategy. For an explanation of the difference in stability, see the works by Golub and Van Loan and by Wilkinson listed in Section 5.7.

At each pivoting iteration, this variant of the algorithm subtracts multiples of the pivot row from the rows *above* as well as from the rows below the pivot row. As a result, the upper triangular matrix is brought to zero along with the lower triangular matrix. This method is different from the Gaussian elimination method, which subtracts multiples of the pivot row from only the rows below it, and thus does not zero the upper triangular matrix. Note that the original matrix is never actually replaced by the identity matrix; the space that would otherwise be "wasted" by 1s and 0s is filled with the accumulated inverse solution. The inversion result is thus efficiently returned in place.

5.4.2 The Gauss-Jordan Solver

The **gen_gj_solve** routine uses the same algorithm and pivoting strategies as the **gen_gj_invert** routine.

5.4.3 Stability and Performance

The variant of the Gauss-Jordan algorithm implemented in the CMSSL (with row pivoting instead of the usual column pivoting) has been shown to be conditionally stable in the following sense: its residual is about as small as the residual from standard Gaussian elimination with column pivoting, as long as the matrix is well conditioned and pivot growth is moderate. For ill-conditioned matrices, this variant fails about as often as Gaussian elimination. For further details, see Dekker and Hoffman, listed in Section 5.7.

If the system of equations is known to be poorly conditioned or the condition of the system is unknown, the *LU* routines are recommended with respect to stability. The *LU* factor and solve routines may yield better performance than **gen_gj_solve**; and using the *LU* factor and solve routines to solve $AX = I$ yields significantly better performance than using **gen_gj_invert** to invert a matrix.

The **CMSSL_total_pivoting** method of pivoting is more numerically stable, but slower than the **CMSSL_partial_pivoting** method.

Matrix Inversion

This function inverts a matrix in place, using a variant of the Gauss-Jordan algorithm. The data type of the source array must be either real or complex.

SYNTAX

pivot_min = **gen_gj_invert** (*A*, *size*, *pivoting_strategy*, *ier*)

ARGUMENTS

A — 2-dimensional CM array of type real or complex. Contains, and may be larger than, the square matrix to be inverted. Must be of size (*size*, *size*) or larger.

Upon successful completion, the data in the upper left-hand (*size*, *size*) area of *A* is overwritten with the inverted matrix.

size — Scalar integer greater than 0. The number of rows (or columns) in the matrix to be inverted.

pivoting_strategy — Scalar integer representing the pivoting strategy used. Value must be one of the following symbolic constants (or integer equivalent):

CMSSL_partial_pivoting (0)
Modified partial pivoting. Column pivoting, where the pivot is chosen from the pivot row; columns are, in effect, permuted.

CMSSL_total_pivoting (1)
Conventional total pivoting, where the pivot is chosen from the submatrix below and to the right of the pivot element; both columns and rows are permuted.

ier — Error code. Scalar integer variable set to 0 if the routine succeeds, and to 1 otherwise. A value of 1 indicates either that one or more arguments were incorrect (for example, *size* = 0), or that a floating-point exception occurred (indicating that the matrix is singular or ill-conditioned).

RETURNED VALUE

pivot_min Real double-precision scalar variable. The magnitude of the smallest pivot used. A very small pivot is evidence that the matrix is close to singular (non-invertible). If an error or a floating-point exception occurs, the routine returns a double-precision zero and sets *ier* to 1.

DESCRIPTION

The **gen_gj_invert** routine inverts a (*size* x *size*) real or complex matrix in place, using a rehabilitated Gauss-Jordan algorithm. If the matrix *A* is smaller than its containing array, *A*, then the remainder of the values in *A* are left untouched, as shown in Figure 22.

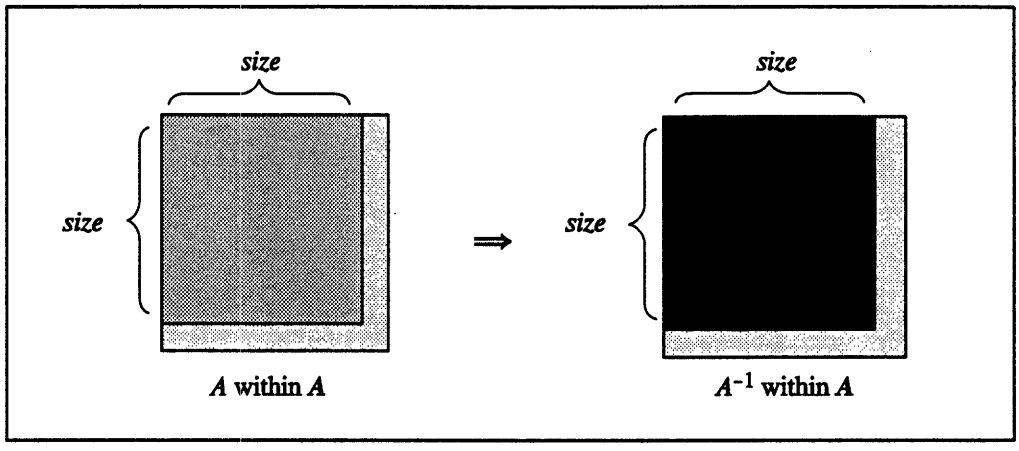

Figure 22. Matrix *A* is inverted; the rest of *A* is unchanged.

NOTES

Include the CMSSL Header File. The matrix inversion routine is a function; it returns the double-precision value *pivot_min*. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of the main program file. This file declares the type of the CMSSL functions and symbolic constants.

Numerical Complexity. Given a matrix with dimensions (n x n), the number of floating-point operations is $2n^3$ for real operands and $8n^3$ for complex operands.

EXAMPLES

Sample CM Fortran code that uses the matrix inversion routine can be found on-line in the subdirectory

```
invert-and-solve/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Gauss-Jordan System Solver

Given two matrices, A and B, where B contains one or more right-hand-side vectors, this routine solves a system of linear equations $AX = B$ and overwrites B with the solution $X = A^{-1} B$. A numerically well-behaved variant of the Gauss-Jordan algorithm is used.

The two source arrays must be separate and distinct and they must have the same data type: either real or complex.

SYNTAX

pivot_min = **gen_gj_solve** (*A, B, size, nrhs, pivoting_strategy, ier*)

ARGUMENTS

A	2-dimensional CM array of type real or complex. Contains, and may be larger than, the (*size* x *size*) square matrix of coefficients, *A*. Must be of shape (*size*, *size*) or larger.
B	1- or 2-dimensional CM array of type real or complex. Contains, and may be larger than, the (*size* x *nrhs*) matrix *B* that contains the right-hand-side vectors (b_1 ... b_{nrhs}). If there are multiple right-hand sides, *rhs* must be of shape (*size*, *nrhs*) or larger. However, if *nrhs* = 1, then *rhs* can be either a vector of length greater than or equal to *size*, or a matrix of shape (*size*, 1) or larger. Upon successful return, *rhs* is overwritten by the solutions.
size	Scalar integer variable. The number of rows (and columns) in the matrix *A*.
nrhs	Scalar integer variable. The number of right-hand-side vectors in the matrix *B*. Must be less than or equal to the smaller dimension of *A*.
pivoting_strategy	Scalar integer variable representing the pivoting strategy used. Value must be one of the following symbolic constants (or integer equivalent): **CMSSL_partial_pivoting** (0) Modified partial pivoting. Column pivoting, where the pivot is chosen from the pivot row; columns are, in effect, permuted.

CMSSL_total_pivoting (1)
Conventional total pivoting, where the pivot is chosen from the submatrix below and to the right of the pivot element; both columns and rows are permuted.

ier Error code. Scalar integer variable set to 0 if the routine succeeds, and to 1 otherwise.

RETURNED VALUE

pivot_min Real double-precision scalar variable. The magnitude of the smallest pivot used. A very small pivot is evidence that the matrix is close to singular. If an error or a floating-point exception occurs, the routine returns a double-precision zero and sets *ier* to 1.

DESCRIPTION

Given a matrix *A* of shape (*size*, *size*) contained within *A*, and a second matrix *B* that contains *nrhs* right-hand-side vectors, $b_1 \ldots b_n$ and is contained in *B*, this function solves for *X* in *AX* = *B* and overwrites *B* with *X*, as shown in Figure 23. Matrix *A* is left untouched.

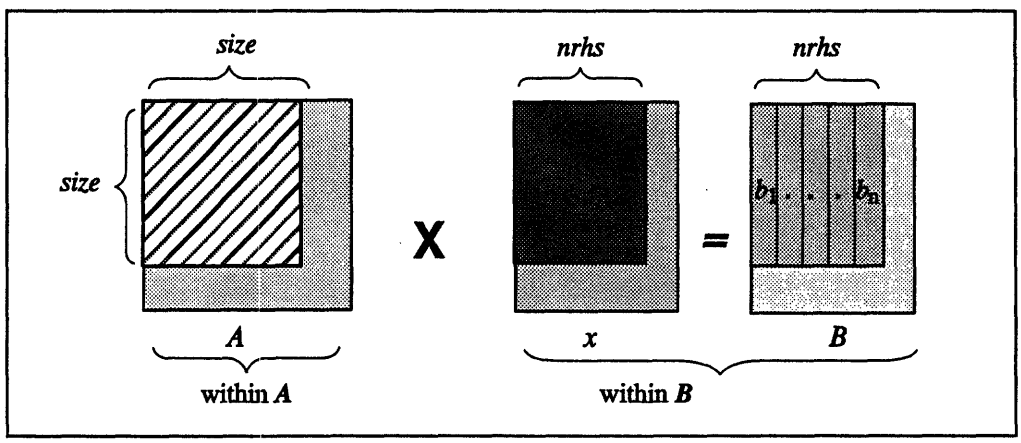

Figure 23. Linear system solved for multiple right-hand-side vectors $b_1 \ldots b_n$.

This operation is equivalent to performing *nrhs* column solves on *B*. That is, within *B*, each column *b* is replaced by (A^{-1} *b*). Note that while formally $X = A^{-1} B$, as implemented, this routine does not perform an explicit multiplication by A^{-1}.

NOTES

Include the CMSSL Header File. The Gauss-Jordan system solver routine is a function; it returns the double-precision value *pivot_min*. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of the main program file. This file declares the type of the CMSSL functions and symbolic constants.

Distinct Variables. The input CM arrays must be distinct variables.

Numerical Complexity. Given an A with dimensions (n x n), and B with dimensions (n x r), the number of floating-point operations is (2/3) $n^3 + 2n^2 r$ for real operands and (8/3) $n^3 + 8n^2 r$ for complex operands.

As an artifact of the implementation, this linear system solver routine copies A and B into a temporary array with dimensions (*size* x [*size* + *nrhs*]).

EXAMPLES

Sample CM Fortran code that uses the Gauss-Jordan system solver can be found on-line in the subdirectory

```
invert-and-solve/cmf/
```

of a CMSSL examples directory whose location is site-specific.

5.5 Gaussian Elimination with External Storage

The routines described in this section solve a linear system of equations ***AX=B*** where ***A*** is a real or complex matrix of size $n \times n$ that is too large to fit into core memory. The method used for reducing ***A*** to triangular form is block Gaussian elimination with partial pivoting. The *L* and *U* factors are stored externally and can later be used to solve ***AX=B*** for an arbitrary number of right-hand sides.

Details are provided in the man page that follows.

Gaussian Elimination with External Storage

The routines described below solve the linear system of equations $AX=B$ where A is a real or complex matrix of size $n \times n$ that is too large to fit into core memory. The method used for reducing A to triangular form is block Gaussian elimination with partial pivoting. The L and U factors are stored externally and can later be used to solve $AX=B$ for an arbitrary number of right-hand sides.

SYNTAX

gen_lu_factor_ext (*n, blk, type, unit1, unit2, unit3, ier*)

gen_lu_solve_ext (*B, nrhs, n, blk, type, unit2, unit3, ier*)

ARGUMENTS

n — Scalar integer variable. The size of the matrix A that is stored on an external device. Also, the number of rows in the matrix B.

blk — Scalar integer variable. Block size. The matrix A is partitioned into blocks of *blk* columns, or panels. See the Notes section, below, for guidelines for choosing *blk*.

You must use the same block size for both the factor and the solve routine.

type — Scalar integer variable. The data type. Specify one of the following values:

CMSSL_single_real	real*4
CMSSL_double_real	real*8
CMSSL_single_complex	complex*8
CMSSL_double_complex	complex*16

B — CM array of rank 2, the same data type as A, and size $n \times nrhs$. On input, must contain the *nrhs* right-hand sides. On return, contains the *nrhs* solutions to $AX = B$.

nrhs — Scalar integer variable. The number of columns in B.

unit1 — Scalar integer. Valid unit number associated with the file that contains the matrix A stored in serial order (see the Notes below.)

Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). Data stored in *unit1* is not modified unless *unit1* = *unit2*.

unit2 Scalar integer. Valid unit number associated with the file that will contain the *LU* factors on return from **gen_lu_factor_ext**. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). If *unit2* = *unit1*, the original matrix *A* is overwritten by its *LU* factors.

unit3 Scalar integer. Valid unit number associated with the file that will contain internal information about the *LU* factors on return from **gen_lu_factor_ext**. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number).

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

-1 I/O error on *unit1*.

-2 I/O error on *unit2*.

-3 I/O error on *unit3*.

-4 Invalid type.

DESCRIPTION

The routines described in this man page solve the linear system of equations $AX=B$, where A is a real or complex matrix of size $n \times n$ that is too large to fit into core memory. The **gen_lu_factor_ext** routine reads blocks of *blk* columns of A from *unit1*, uses block Gaussian elimination with partial pivoting to reduce A to triangular form, writes the *LU* factors to *unit2*, and writes information about them to *unit3*. The **gen_lu_solve_ext** routine reads the factors from *unit2* and *unit3*, solves $AX=B$ for an arbitrary number of right-hand sides, and returns the *nrhs* solutions in the B argument.

The **gen_matrix_mult_ext** routine, described in Chapter 3, can be used to check the accuracy of the result. The best possible accuracy for the solution to $Ax = b$ is obtained-when $\|Ax-b\|_\infty / \|A\|_\infty \|x\|_\infty = \varepsilon$, where ε is the machine accuracy. The quantity $AX - B$ can be computed with **gen_matrix_mult_ext** for all right-hand sides at once.

NOTES

Include the CMSSL Header File. Because the routines described above use symbolic constants, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL symbolic constants.

File Units. The I/O units *unit1*, *unit2*, and *unit3* must be assigned to files before you call the routines that access them. In CM Fortran, file assignment is done with the **CMF_FILE_OPEN** utility (or an equivalent utility for a device or socket). For information regarding parallel I/O in general, see the *CM-5 I/O System Programming Guide*. For information about the CM Fortran interface to parallel I/O, see the *CM Fortran Utility Library Reference Manual*. As described in this manual, there are essentially two modes of external storage: Fixed Machine Size (FMS) and Serial Order (SO). Serial order is the familiar Fortran row-major order and is the one used by the external *LU* routines. Therefore, *A* must be stored in serial order in file unit *unit1*. In this order, the data is portable across the CM-5 external storage systems (DataVault, Scalable Disk Array, HIPPI).

The file associated with *unit2* will store as much data as the original matrix (that is, n^2 data elements), whereas the file associated with *unit3* will contain much less data, the exact amount depending on the machine configuration.

If you set *unit2* = *unit1*, the original matrix *A* is overwritten by its *LU* factors.

Partition Size. The partition size used to solve the system of equations must be identical to the one used previously to factor the matrix.

Choosing the Block Size. The block Gaussian elimination algorithm partitions the matrix into block columns, or panels, A_i, of size n x *blk*:

$$A = [\, A_1, A_2, \ldots, A_m \,].$$

The last panel, A_m, contains fewer than *blk* columns if *blk* is not a divisor of *n*. It is important to choose the block size *blk* as large as possible in order to minimize the I/O cost and optimize machine utilization. The in-core memory requirement for **gen_lu_factor_ext** is approximately $(5v + 16)n*blk$ bytes, where v is the number of bytes in the data type of *A*.

Choosing the block size *blk* to be a multiple of 16 may also improve performance. This is because the blocking factor, *nblock*, in the in-core *LU* routine **gen_lu_factor** (upon

which **gen_lu_factor_ext** is built) is set internally to 16. Given this fact and the memory requirement mentioned above, it is possible to choose a reasonable *blk* value for a given n and a given amount of core memory.

Complexity Analysis. The **gen_lu_factor_ext** routine requires $(2/3)n^3$ operations for real operands and $(8/3)n^3$ operations for complex operands. The amount of data transferred during the block *LU* triangularization is

$$\frac{n(n + blk)\ (2n + blk)}{3\ blk}.$$

For blk/n small, this quantity becomes $O(2n^3 / 3blk)$. Given these numbers, the average floating-point operation (flop) rate for the in-core *LU* routine, and the data transfer rate between the CM and the external storage system, it is possible to make a very rough estimate of the time required for the out-of-core factorization. On the CM-5, for example, a conservative choice for the flop rate for the in-core *LU* routine is 10 Mflops per vector unit, while the data transfer rate on the Scalable Disk Array is roughly 1 Mbyte/sec per disk. With p vector units and q disks, the estimated time for a problem of size n is

$$T_{arith} = \frac{u}{3}\frac{n^3}{10p}\,10^{-6} \text{ seconds and } T_{transfer} = \frac{2}{3}\frac{n^3}{blk}\frac{v}{q}\,10^{-6} \text{ seconds}$$

where $u = 2$ for real operands and $u = 8$ for complex operands, and v is the number of bytes in the matrix data type. Hence, the total time is

$$T = \frac{n^3}{3}\left[\frac{u}{10p} + \frac{2v}{q*blk}\right]10^{-6} \text{ seconds.}$$

Choosing, for example, $u = 2$, $v = 8$ (that is, a data type of real*8), $n = 10{,}000$, $blk = 1200$, $p = 128$ vector units, and $q = 8$, we have $T_{arith} = 520$ seconds and $T_{transfer} = 555$ seconds, and hence a total time of $T \approx 18$ minutes. Only the order of magnitude of such an estimate should be considered significant.

EXAMPLES

Sample CM Fortran code that uses the *LU* routines can be found on-line in the subdirectory

```
external/lu/cmf/
```

of a CMSSL examples directory whose location is site-specific.

5.6 *QR* Factorization and Least Squares Solution with External Storage

The routines described in this section perform a *QR* factorization of a real or complex matrix A of size $m \times n$ (with $m \geq n$) that is too large to fit into core memory. The method uses Householder reflections. The Q and R factors are stored externally and can later be used to solve $AX=B$ for an arbitrary number of right-hand sides.

Details are provided in the man page that follows.

QR Factorization and Least Squares Solution with External Storage

The routines described below perform a *QR* factorization of a real or complex matrix *A* of size $m \times n$ (with $m \geq n$) that is too large to fit into core memory. The method uses Householder reflections. The *Q* and *R* factors are stored externally and can later be used to solve *AX=B* for an arbitrary number of right-hand sides.

SYNTAX

gen_qr_factor_ext (*m, n, blk, type, pivoting_strategy, unit1, unit2, unit3, ier*)

gen_qr_solve_ext (*B, nrhs, m, n, blk, type, pivoting_strategy, unit2, unit3, ier*)

ARGUMENTS

m — Scalar integer variable. The number of rows in the matrix *A* that is stored on an external device. Also, the number of rows in the matrix *B*.

n — Scalar integer variable. The number of columns in the matrix *A* that is stored on an external device.

blk — Scalar integer variable. Block size. The matrix *A* is partitioned into blocks of *blk* columns, or panels. See the Notes section, below, for guidelines for choosing *blk*.

You must use the same block size for both the factor and the solve routine.

type — Scalar integer variable. The data type. Specify one of the following values:

CMSSL_single_real	real*4
CMSSL_double_real	real*8
CMSSL_single_complex	complex*8
CMSSL_double_complex	complex*16

pivoting_strategy — Scalar integer variable specifying the pivoting strategy to be used. The only values currently available are as follows:

CMSSL_no_pivoting	No pivoting, no scaling
CMSSL_no_pivoting_scale	No pivoting, scaling

For a description of scaling, see Section 5.3.7.

B CM array of rank 2, the same data type as *A*, and size *m* x *nrhs*. On input, must contain the *nrhs* right-hand sides. On return, the first *n* rows of ***B*** contain the *nrhs* solutions to ***AX*** = ***B***.

nrhs Scalar integer variable. The number of columns in ***B***.

unit1 Scalar integer. Valid unit number associated with the file that contains the matrix *A* stored in serial order (see the Notes below.) Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). Data stored in *unit1* is not modified unless *unit1* = *unit2*.

unit2 Scalar integer. Valid unit number associated with the file that will contain the *QR* factors on return from **gen_qr_factor_ext**. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). If *unit2* = *unit1*, the original matrix *A* is overwritten by its *QR* factors.

unit3 Scalar integer. Valid unit number associated with the file that will contain internal information about the *QR* factors on return from **gen_qr_factor_ext**. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number).

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

-1	I/O error on *unit1*.
-2	I/O error on *unit2*.
-3	I/O error on *unit3*.
-4	Invalid type.

DESCRIPTION

The routines described in this man page perform a *QR* factorization of a real or complex matrix *A* of size *m* x *n* (with $m \geq n$) that is too large to fit into core memory. The **gen_qr_factor_ext** routine reads blocks of *blk* columns of *A* from *unit1*, uses block Householder reflections to factor *A*, writes the *QR* factors to *unit2*, and writes information about them to *unit3*. The **gen_qr_solve_ext** routine reads the factors from *unit2* and *unit3*, solves *AX=B* for an arbitrary number of right-hand sides, and returns the *nrhs* solutions in the first *n* rows of *B*.

The **gen_matrix_mult_ext** routine, described in Chapter 3, can be used to check the accuracy of the result. The best possible accuracy for the solution to $Ax = b$ is obtained when $\|Ax-b\|_\infty / \|A\|_\infty \|x\|_\infty = \varepsilon$, where ε is the machine accuracy. The quantity $AX - B$ can be computed with **gen_matrix_mult_ext** for all right-hand sides at once.

NOTES

Include the CMSSL Header File. Because the routines described above use symbolic constants, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL symbolic constants.

File Units. The I/O units *unit1*, *unit2*, and *unit3* must be assigned to files before you call the routines that access them. In CM Fortran, file assignment is done with the **CMF_FILE_OPEN** utility (or an equivalent utility for a device or socket). For information regarding parallel I/O in general, see the *CM-5 I/O System Programming Guide*. For information about the CM Fortran interface to parallel I/O, see the *CM Fortran Utility Library Reference Manual*. As described in this manual, there are essentially two modes of external storage: Fixed Machine Size (FMS) and Serial Order (SO). Serial order is the familiar Fortran row-major order and is the one used by the external *QR* routines. Therefore, *A* must be stored in serial order in file unit *unit1*. In this order, the data is portable across the CM-5 external storage systems (DataVault, Scalable Disk Array, HIPPI).

The file associated with *unit2* will store as much data as the original matrix (that is, *nm* data elements), whereas the file associated with *unit3* will contain much less data, the exact amount depending on the machine configuration.

If you set *unit2* = *unit1*, the original matrix *A* is overwritten by its *QR* factors.

Partition Size. The partition size used to solve the system of equations must be identical to the one used previously to factor the matrix.

Choosing the Block Size. The block Householder factorization algorithm partitions the matrix into block columns, or panels, A_i, of size m x *blk*:

$$A = [\, A_1, A_2, ..., A_l \,].$$

The last panel, A_l, contains fewer than *blk* columns if *blk* is not a divisor of n. It is important to choose the block size *blk* as large as possible in order to minimize the I/O cost and optimize machine utilization. The in-core memory requirement for **gen_qr_factor_ext** is approximately $(5v + 16)m*blk$ bytes, where v is the number of bytes in the data type of A.

Choosing the block size *blk* to be a multiple of 16 may also improve performance. This is because the blocking factor, *nblock*, in the in-core *QR* routine **gen_qr_factor** (upon which **gen_qr_factor_ext** is built) is set internally to 16. Given this fact and the memory requirement mentioned above, it is possible to choose a reasonable *blk* value for a given n and a given amount of core memory.

Least Squares Solution. The least squares solution obtained from **gen_qr_solve_ext** is unique only if the problem has full rank (rank $A = n$). Unlike the in-core *QR* routines, the current out-of-core version does not provide a way for you to determine the rank of A.

Complexity Analysis. The **gen_qr_factor_ext** routine requires $2n^2[m - (n/3)]$ operations for real operands and $8n^2[m - (n/3)]$ operations for complex operands. The amount of data transferred during the block *QR* factorization is

$$\frac{n(n + blk)}{blk}\left[m + \frac{blk-n}{3}\right].$$

For *blk*/n small, this quantity becomes

$$O\left[\frac{n^2}{blk}\left\{m - \frac{n}{3}\right\}\right].$$

Given these numbers, the average floating-point operation (flop) rate for the in-core *QR* routine, and the data transfer rate between the CM and the external storage system, it is possible to make a very rough estimate of the time required for the out-of-core factorization. On the CM-5, for example, a conservative choice for the flop rate for the in-core *QR* routine is 10 Mflops per vector unit, while the data transfer rate on the

Scalable Disk Array is roughly 1 Mbyte/sec per disk. With p vector units and q disks, the estimated time for a problem of size m x n is

$$T_{arith} = \frac{un^2}{10p}\left\{m - \frac{n}{3}\right\} 10^{-6} \text{ seconds and } T_{transfer} = \frac{n^2}{blk}\left\{m - \frac{n}{3}\right\}\frac{v}{q} 10^{-6} \text{ seconds}$$

where $u = 2$ for real operands and $u = 8$ for complex operands, and v is the number of bytes in the matrix data type. Hence, the total time is

$$T = n^2\left[m - \frac{n}{3}\right]\left[\frac{u}{10p} + \frac{v}{q*blk}\right] 10^{-6} \text{ seconds.}$$

Choosing, for example, $u = 2$, $v = 8$ (that is, a data type of real*8), $m = 10{,}000$, $n = 5{,}000$, $blk = 1200$, $p = 128$ vector units, and $q = 8$, we have $T_{arith} = 325$ seconds and $T_{transfer} = 174$ seconds, and hence a total time of $T \approx 10$ minutes. Only the order of magnitude of such an estimate should be considered significant.

EXAMPLES

Sample CM Fortran code that uses the *LU* routines can be found on-line in the subdirectory `external/qr/cmf/` of a CMSSL examples directory whose location is site-specific.

5.7 References

For information about general linear solvers, see the following references:

1. Golub, G.H., and C. F. Van Loan. *Matrix Computations.* 2d ed. Baltimore: Johns Hopkins University Press, 1989.

2. Johnsson, S. L. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing* **4** (1987): 133–72.

3. Wilkinson, J. H. Error Analysis of Direct Methods of Matrix Inversion. *J. Assoc. Comput. Mach.* **8** (1961): 281–330.

For information about how the slicewise *LU* and *QR* routines implement blocking, and other aspects of the *LU* and *QR* operations, refer to

4. Dongarra, J. J., I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* Philadelphia: SIAM, 1991.

5. Hager, W. W. Condition Estimates. *SIAM J. Sci. Stat. Comput.* **5** (1984): 311–16.

6. Higham, N. J. Experience with a Matrix Norm Estimator. *SIAM J. Sci. Stat. Comput.* **11**, no. 4 (1990): 804–9.

7. Higham, N. J. FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation. (Algorithm 674) *ACM Trans. Math. Soft.* **14** (1988): 381–96.

8. Higham, N. J. The Accuracy of Solutions to Triangular Systems. *SIAM J. Numer. Anal.* **26**, no.5 (1989):1252–65.

9. Johnsson, S. L. A Computational Array for the *QR*-method. *Proceedings of the Conference on Advanced Research in VLSI.* Ed. P. Penfield, Jr. Artech House, 1982. Pp. 123–29.

10. Lichtenstein, W. and S. L. Johnsson. *Block Cyclic Linear Algebra.* Thinking Machines Corporation Technical Report TR-215, 1992.

11. Schreiber, R. S., and C. F. Van Loan. A Storage Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. Stat. Comput.* **10**, no. 1 (1989): 53–57.

12. Vavasis, S. *Implementation of QR factorization on the Connection Machine CM-2.* Personal communication.

13. Wilkinson, J. H. *The Algebraic Eigenvalue Problem.* Oxford: Clarendon Press, 1965.

14. Van de Geijn, R. A. *Massively Parallel LINPACK Benchmark on the Intel Touchstone Delta and iPSC/860 Systems.* Technical Report, University of Texas at Austin, 1991.

For an analysis of the numerical behavior of the algorithm used in the matrix inversion and Gauss-Jordan reduction routines, see the following:

15. Dekker, T. J., and W. Hoffman. Rehabilitation of the Gauss-Jordan Algorithm. *Numerische Mathematik* **54** (1989): 591-99.

Chapter 6

Linear Solvers for Banded Systems

This chapter describes the CM Fortran interface to the CMSSL banded linear system solver routines. The banded system routines factor and solve tridiagonal, pentadiagonal, block tridiagonal, and block pentadiagonal systems of equations. They solve multiple systems of equations, each with one or more right-hand sides, for both real and complex data types. A choice of algorithms is offered.

The multiple-instance capability of the banded system routines in CMSSL is particularly useful in connection with Fourier Analysis Cyclic Reduction, or Alternating Direction Methods. You can specify the axis along which the systems are to be solved. No data reordering or transposition is necessary for the solution of systems along any axis.

On the CM-5, the CMSSL includes two sets of banded system routines that offer nearly the same functionality:

- A "unified" set of routines. This set includes one factorization routine and one solver routine that work on all four banded system types (tridiagonal, pentadiagonal, block tridiagonal, and block pentadiagonal). Section 6.1 describes these routines.
- A set that includes three routines (a factorization routine, a solver routine, and a routine that both factors and solves) for each of the four banded system types. These routines are included in the library for compatibility with the CM-200, and are described in Section 6.2.

The two sets of banded system routines use the same array arguments. Only the ordering of some of the arrays in the calling sequence differs. For example, the "unified" routines list the arrays representing diagonals in order from lowermost to uppermost (***a***, ***b***, ***c***, ***d***, ***e***) while the other routines list them from uppermost to lowermost (***e***, ***d***, ***c***, ***b***, ***a***). In addition, the "unified" routines allow you to supply a pivot value — a feature not included in the other routines.

Detailed descriptions of the banded system routines, including calling sequences, argument definitions, and usage information, are provided in the man pages in this chapter. Section 6.3 lists references.

6.1 Banded System Factorization and Solver Routines (Unified)

This section describes the "unified" banded system factorization and solver routines. The following topics are covered:

- the routines and their functions
- algorithms used
- how to set up your data

6.1.1 The Routines and Their Functions

The unified banded system routines are listed below.

gen_banded_factor Given tridiagonal or block tridiagonal matrices A (represented by three arrays), or pentadiagonal, or block pentadiagonal matrices A (represented by five arrays), this routine performs the factorization $A = LU$ for each matrix, where L and U are lower and upper (respectively) bidiagonal or block bidiagonal, or lower and upper (respectively) tridiagonal or block tridiagonal matrices, or permutations thereof.

gen_banded_solve Given the factors computed by **gen_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

deallocate_banded This routine deallocates the memory required by the factorization and solver routines.

6.1.2 Algorithms Used

When calling the banded system routines, you must specify the algorithm to be used. The following algorithms are available:

CMSSL_pipeline_ge Pipelined Gaussian elimination.

CMSSL_pge_piv Pipelined Gaussian elimination with pairwise pivoting. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine uses **CMSSL_pipeline_ge** instead.

CMSSL_pge_piv_val Pipelined Gaussian elimination with pairwise pivoting; replace zero pivots with a supplied value. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine uses **CMSSL_pipeline_ge** instead.

CMSSL_substr_cr Substructuring with cyclic reduction.

CMSSL_substr_bcr Substructuring with balanced cyclic reduction.

CMSSL_substr_pge Substructuring with pipelined Gaussian elimination.

CMSSL_substr_transp Substructuring with transpose. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns an error code.

The last four algorithms listed above involve a "divide and conquer" scheme based on substructuring, and differ in the technique used to solve the reduced system of equations. Performance is strongly influenced by the data layout.

NOTE

If the axis along which the diagonal elements or blocks lie (axis *vector_axis* in the argument list) is serial, the routine *always* uses Gaussian elimination (with pivoting, if you selected pivoting and have a tridiagonal system).

The algorithm descriptions that follow apply to tridiagonal systems. Block tridiagonal algorithms are the obvious extensions of the elementwise ones;

pentadiagonal solvers are based on the block tridiagonal solvers and use the same algorithms.

Pipelined Gaussian Elimination

If you select pipelined Gaussian elimination and you supply multiple systems to be solved, all of which are distributed over the same set of processing elements, then pipelining is used to achieve load balance. Figure 24 illustrates pipelining. In the figure, six systems of equations are distributed over four processing elements. The systems are represented by dashed lines. A solid line represents a set of equations on which a processing element is actively working.

Figure 24 shows that there is a pipeline setup (and shutdown) phase proportional to the number of equations per system per processing element, and the number of processing elements over which the systems are distributed. When there are many more systems per processing element than processing elements assigned to each system, then all processing elements are active for most of the time, and good load balance is achieved. In the current vector unit implementation, the situation is somewhat more complex in that vectorization in each processing element is performed over sets of eight systems. The actual implementation corresponds to the case where each (dashed) line in Figure 24 represents eight systems of equations. Hence, the pipeline setup and shutdown times are more significant than the figure indicates. For few systems per processing element and for many processing elements, the vectorization adversely affects performance, while a signficant gain in performance is achieved when there are many more systems per processing element than there are processing elements.

The current implementation of Gaussian elimination computes a reciprocal of the diagonal elements in order to minimize the number of divisions required when there are multiple right-hand sides per system of tridiagonal equations. The number of additions and multiplications for *RHS* right-hand sides per system of N equations is $(3+5RHS)(N-1)$. In addition, there are N divisions. For an instance factor of I, both numbers are multiplied by I.

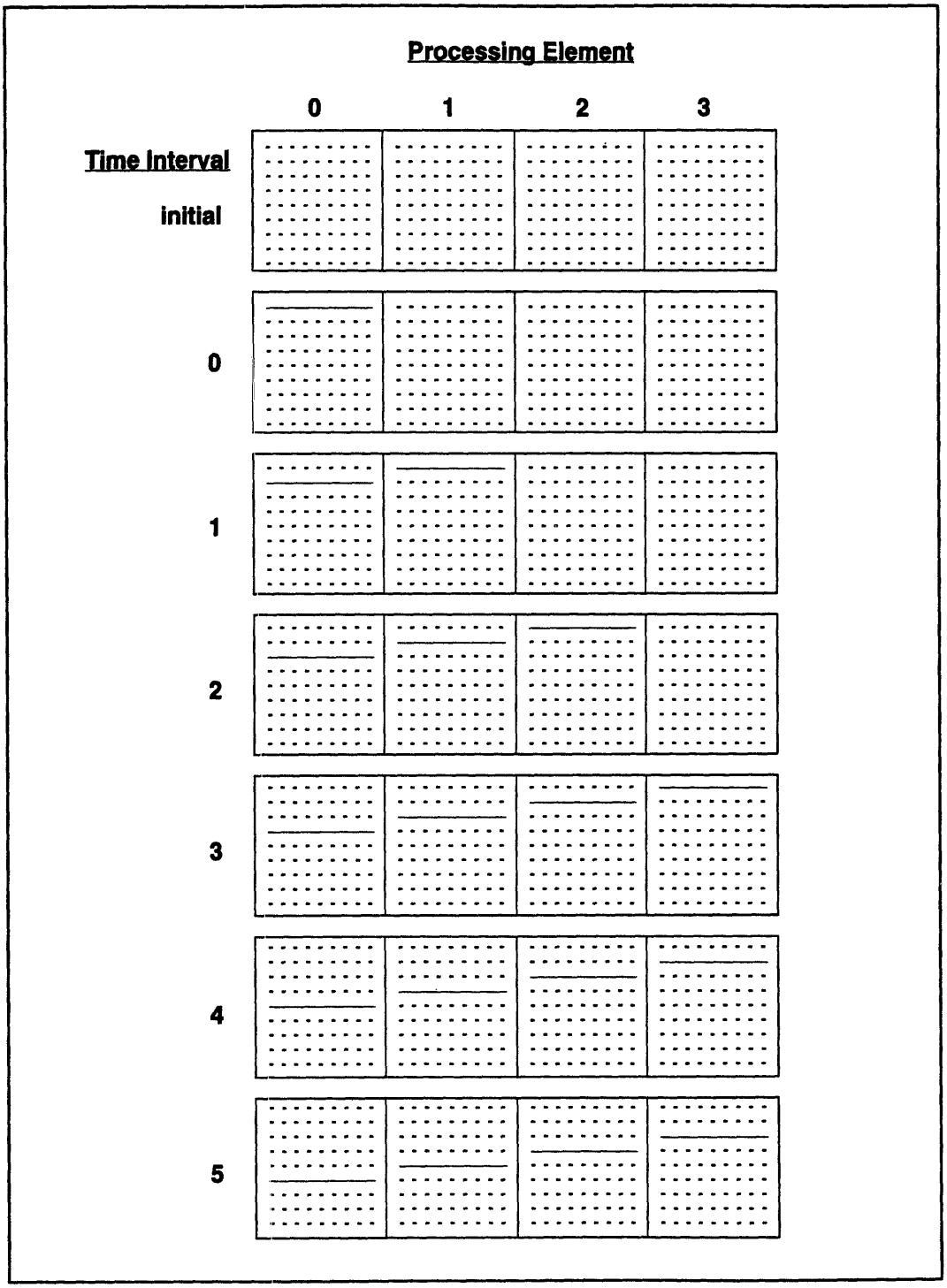

Figure 24. Pipelined Gaussian elimination.

Pipelined Gaussian Elimination with Pairwise Pivoting

Pairwise pivoting refers to the exchange of a pair of adjacent rows whenever that exchange results in a larger divisor for use in eliminating the subdiagonal element.

Substructuring

Substructuring reduces the number of equations (or block equations) on a processing element to a single equation (or single block equation). This is accomplished by means of a staggered forward and backward Gaussian elimination, as described in references 6 and 4 in Section 6.3. After the elimination, a reduced tridiagonal system must be solved using cyclic reduction, balanced cyclic reduction, or pipelined Gaussian elimination.

Cyclic reduction is discussed below. Balanced cyclic reduction can be used in the multiple-instance case to improve the load balance over that of standard cyclic reduction. During each stage of the cyclic reduction, the number of instances returned in parallel doubles. In substructuring with transpose, the reduced tridiagonal system of equations resulting from the substructuring is transposed so that the Gaussian elimination is done locally; the results are transposed back into the original geometry.

Cyclic Reduction

A tridiagonal system of irreducible linear equations $Ax = y$, where A is of dimension $N = 2^{n-1}$, can be presented in matrix vector form as

$$\begin{bmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & a_3 & b_3 & c_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \\ & & & & & a_N & b_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ \\ x_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \\ \\ y_N \end{bmatrix}$$

Odd-even cyclic reduction consists of a reduction phase succeeded by a back-substitution phase. Using subscripts for equation numbers and superscripts to

denote reduction and back-substitution steps, cyclic reduction is defined by the following set of equations:

Reduction

$$a_i^j = e_i \, a_{i-2^{j-1}}^{j-1}$$

$$c_i^j = f_i \, c_{i+2^{j-1}}^{j-1}$$

$$b_i^j = b_i^{j-1} + e_i \, c_{i-2^{j-1}}^{j-1} + f_i \, a_{i+2^{j-1}}^{j-1}$$

$$y_i^j = y_i^{j-1} + e_i \, y_{i-2^{j-1}}^{j-1} + f_i \, y_{i+2^{j-1}}^{j-1}$$

$$e_i = -\frac{a_i^{j-1}}{b_{i-2^{j-1}}^{j-1}}$$

$$f_i = -\frac{c_i^{j-1}}{b_{i+2^{j-1}}^{j-1}}$$

where $i = 2^j, 2 \times 2^j, 3 \times 2^j, \ldots, 2^n - 2^j$, for reduction steps $j = 1, 2, \ldots, n-1$.

The initial conditions are

$$a_i^0 = a_i \quad b_i^0 = b_i, \quad c_i^0 = c_i, \text{ and } y_i^0 = y_i.$$

After $n - 1$ reduction steps, only one equation of the following form remains:

$$a_{2^{n-1}}^{n-1} x_0 + b_{2^{n-1}}^{n-1} x_{2^{n-1}} + c_{2^{n-1}}^{n-1} x_{2^n} = y_{2^{n-1}}^{n-1}$$

A correct solution for

$$x_{2^{n-1}}$$

is obtained, with $x_0 = x_{N+1} = 0$. Remaining variables are obtained through back-substitution using the following equations:

Back-Substitution

$$x_{2^{n-1}} = \frac{y_{2^{n-1}}^{n-1}}{b_{2^{n-1}}^{n-1}}$$

$$x_i = \frac{y_i^{j-1} - a_i^{j-1} x_{i-2^{j-1}} - b_i^{j-1} x_{i+2^{j-1}}}{b_i^{j-1}}$$

where $i = \{ 2^{j-1}, 3 \times 2^{j-1}, 5 \times 2^{j-1}, \ldots, 2^n - 2^{j-1} \}$, and $j = \{n - 1, n - 2, \ldots, 1\}$.

In the above algorithm, 12 arithmetic operations are needed per equation in the reduction computation, and 5 per unknown in the back-substitution. A careful count gives a total of $17N - 18n + 2$ arithmetic operations, disregarding index computations.

Hints for Choosing an Algorithm

Performance is best for a given array when the axis along which the blocks or elements lie is serial. When this axis is not serial, use the following guidelines when choosing an algorithm:

- If there is only one instance, substructuring with cyclic reduction yields the best performance.
- As the number of instances per processing element increases, balanced cyclic reduction begins to yield the best performance.

These statements are highly dependent on the exact array size and layout used in a given problem. Thus, these guidelines are rough, and you are encouraged to experiment with different algorithms to find the one best suited to your problem. For example, if you have a multidimensional array in which tridiagonal systems

lying along different axes will be solved with separate calls to the banded solvers, you could use substructuring with cyclic reduction along one axis, pipelined Gaussian elimination along another axis, and substructuring with balanced cyclic reduction along a third axis.

NOTE

If you are working with a single- or multiple-instance element-wise tridiagonal or pentadiagonal system with one right-hand side, and axis *axis* is local to a processing element, you will probably achieve better performance by writing the operation in CM Fortran than by using the CMSSL banded system solver routines. This is especially true in the case of pentadiagonal systems.

Accuracy

Numerical experiments have suggested that pipelined Gaussian elimination produces the most accurate solution.

Numerical Stability

Odd-even cyclic reduction is stable for diagonally dominant or positive definite systems. For poorly conditioned systems, the algorithm may be unstable. The algorithm can be stabilized (see reference 1 in Section 6.3), but the current implementation does not include a stablization scheme.

Gaussian elimination is numerically more stable than odd-even cyclic reduction. The current implementation does not support any data-dependent pivoting.

For an analysis of the stability of Gaussian elimination and odd-even cyclic reduction, see (for instance) references 1, 3, and 8.

6.1.3 How to Set Up Your Data

Tridiagonal and Pentadiagonal Systems

When you factor and solve an elementwise tridiagonal or pentadiagonal system, you must represent each coefficient matrix in the form of three vectors (for tridiagonal systems) or five vectors (for pentadiagonal systems). You must also supply an integer, *vector_axis*, that identifies the axis of each of these three or five vectors along which the matrix elements lie (that is, the non-instance axis).

In addition, when you call the solve routine, you must supply the argument ***B***, the CM array that contains the right-hand-side vectors *B* and is overwritten with the solution *X*.

The detailed requirements for these arrays (and the other required arguments) are provided in the man page at the end of this section. Illustrations and examples are provided below.

For tridiagonal systems, you must supply three CM arrays, ***c***, ***b***, and ***a***, containing the upper, main, and lower diagonal elements, respectively.

To solve a single system, specify the ***c***, ***b***, and ***a*** array arguments as vectors. Figure 25 shows the simplest case: solving a single system with a single right-hand side. Within matrix *A*, the vectors ***c***, ***b***, and ***a*** are shown holding the principal and off-diagonal values. The array ***B*** is shown as two vectors: the right-hand-side vector *B* and the solution vector *X*. Notice that although they represent shorter diagonals, the vectors ***c*** and ***a*** are of the same length as ***b***. The first element of ***a*** and the last element of ***c*** are set to zero during execution.

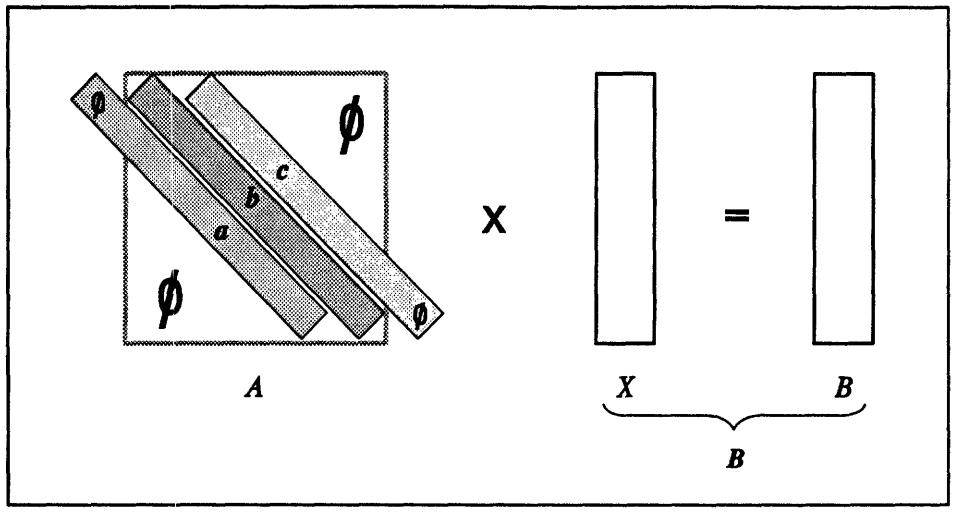

Figure 25. A single tridiagonal system with a single right-hand side.

In matrix notation, the single-system, single-solution case can be represented as shown in Figure 26.

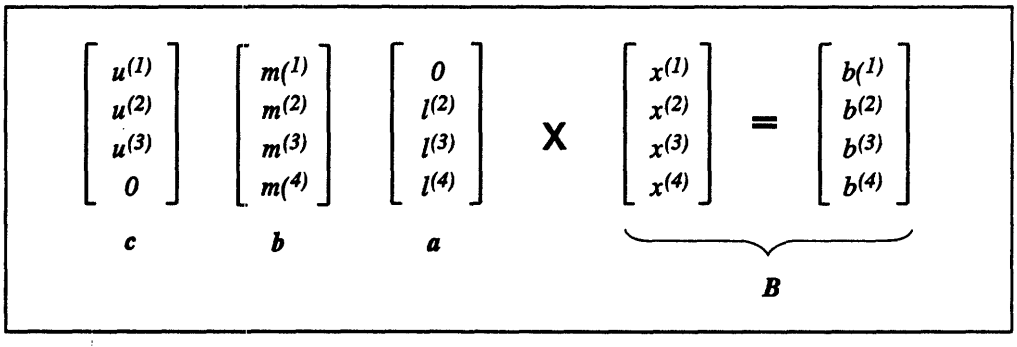

Figure 26. Matrix notation of single tridiagonal system with single right-hand side.

To solve for multiple right-hand sides, specify the ***B*** argument with a serial dimension equal to the number of right-hand sides. Figure 27 shows a single tridiagonal system with *nrhs* right-hand sides. Note that the multiple right-hand-side vectors, $b^{(1)} \ldots b^{(nrhs)}$, and their associated solution vectors, $x^{(1)} \ldots x^{(nrhs)}$, are laid out along a serial dimension of length *nrhs*.

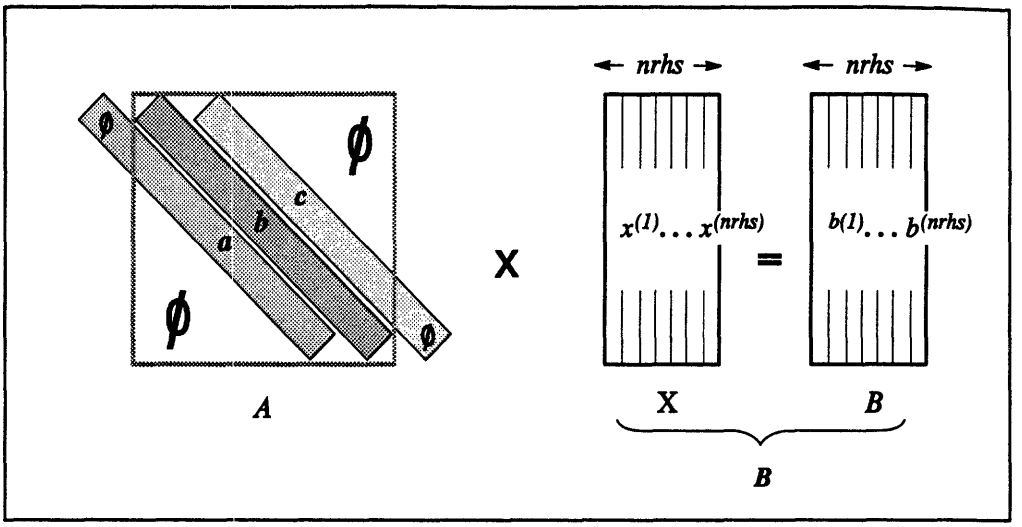

Figure 27. Single tridiagonal system with multiple right-hand sides and solutions.

To solve multiple systems in parallel, specify ***c***, ***b***, and ***a*** with at least 2 dimensions (one data axis and one instance axis) each. The data axis (specified as *vector_axis* in the argument list) represents the coefficients of each system. The instance axis specifies how many systems are represented.

Figure 28 shows multiple concurrent systems, each with a single right-hand side. The *n* instances of the matrix *A* are represented by the *n* sets of tridiagonal values in ***c***, ***b***, and ***a***. Similarly, *rhs* consists of the set $[b_1 \ldots b_n]$ of *n* right-hand-side vectors, and *solution* consists of the set $[x_1 \ldots x_n]$ of *n* solutions. In this case, there is only one right-hand-side vector for each system; each is overwritten by the one solution vector for that system.

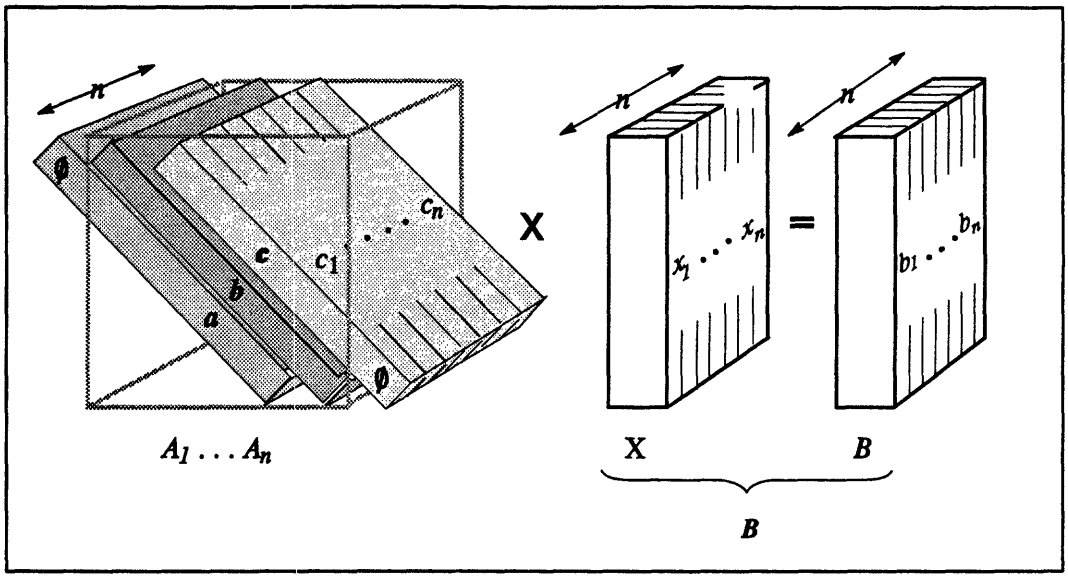

Figure 28. Multiple tridiagonal systems with single right-hand side for each system.

In matrix notation, the multiple-system, single-solution case can be represented as shown in Figure 29.

$$\begin{bmatrix} u_1^{(1)} & u_2^{(1)} & u_3^{(1)} \\ u_1^{(2)} & u_2^{(2)} & u_3^{(2)} \\ u_1^{(3)} & u_2^{(3)} & u_3^{(3)} \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} m_1^{(1)} & m_2^{(1)} & m_3^{(1)} \\ m_1^{(2)} & m_2^{(2)} & m_3^{(2)} \\ m_1^{(3)} & m_2^{(3)} & m_3^{(3)} \\ m_1^{(4)} & m_2^{(4)} & m_3^{(4)} \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ l_1^{(2)} & l_2^{(2)} & l_3^{(2)} \\ l_1^{(3)} & l_2^{(3)} & l_3^{(3)} \\ l_1^{(4)} & l_2^{(4)} & l_3^{(4)} \end{bmatrix}$$

c $\qquad$ b $\qquad$ a

$$\mathbf{X} \quad \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ x_1^{(3)} & x_2^{(3)} & x_3^{(3)} \\ x_1^{(4)} & x_2^{(4)} & x_3^{(4)} \end{bmatrix} \quad = \quad \begin{bmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \\ b_1^{(2)} & b_2^{(2)} & b_3^{(2)} \\ b_1^{(3)} & b_2^{(3)} & b_3^{(3)} \\ b_1^{(4)} & b_2^{(4)} & b_3^{(4)} \end{bmatrix}$$

solution $\qquad$ *rhs*

Figure 29. Matrix notation of multiple tridiagonal systems with one right-hand side each.

Figure 30 shows *n* systems, each with *nrhs* right-hand sides. Remember that the axis of length *nrhs* must be serial.

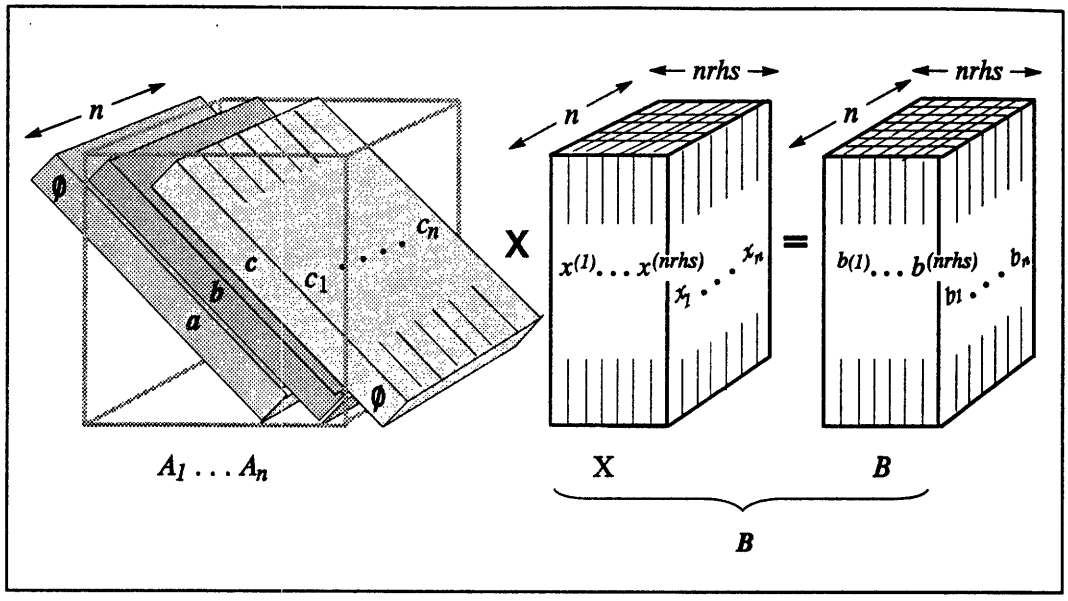

Figure 30. Multiple tridiagonal systems with multiple right-hand sides for each system.

Pentadiagonal systems are represented in the same way as tridiagonal systems, except that you must supply five CM arrays, ***e***, ***d***, ***c***, ***b***, and ***a***, containing the elements of the five diagonals of the coefficient matrices, as shown in Figure 31.

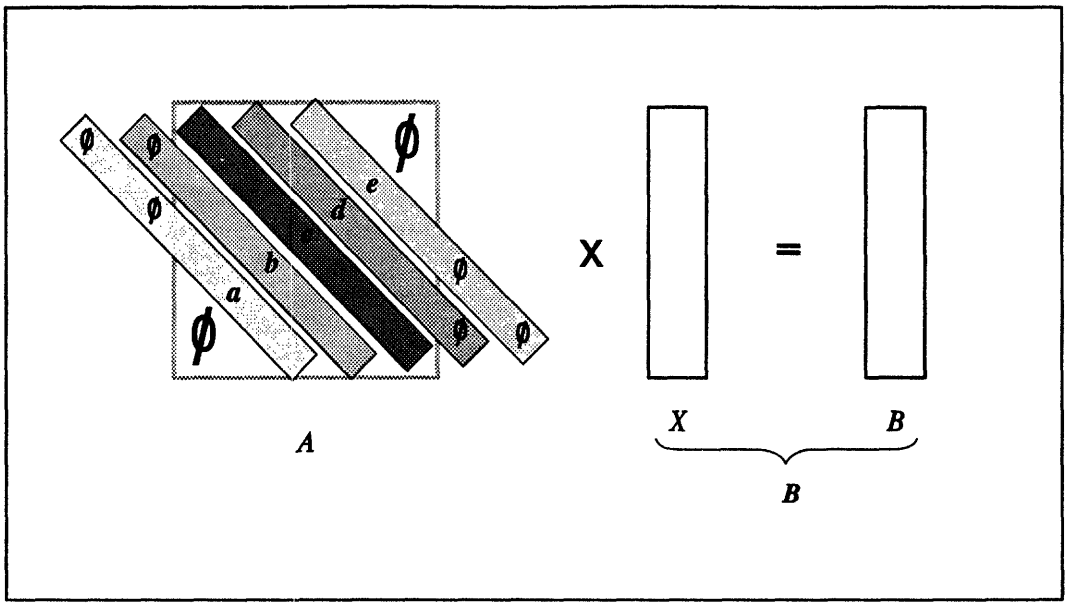

Figure 31. A single pentadiagonal system with a single right-hand side.

Block Tridiagonal and Block Pentadiagonal Systems

When you factor and solve a block tridiagonal or block pentadiagonal system, the routines assume that each matrix A is represented by three arrays (for block tridiagonal systems) or five arrays (for block pentadiagonal systems). More specifically,

- For block tridiagonal systems, you must supply three CM arrays, ***c***, ***b***, and ***a***, containing the square blocks of the coefficient matrices.
- For block pentadiagonal systems, you must supply five CM arrays, ***e***, ***d***, ***c***, ***b***, and ***a***, containing the square blocks of the coefficient matrices.

The detailed requirements for these arrays (and the other required arguments) are provided in the man page at the end of this section.

Figure 32 shows a block tridiagonal system with one instance. In the equation $AX = B$, each $n \times n$ block of A is multiplied by a vector of length n within X to produce a vector of length n within B.

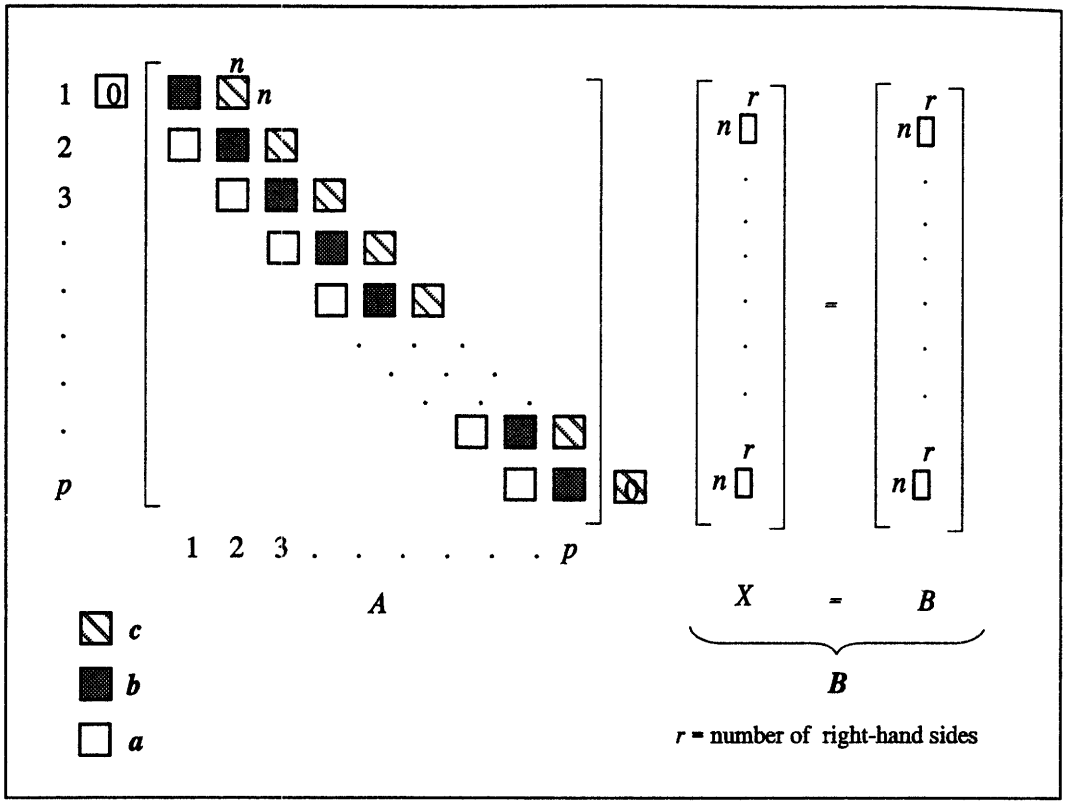

Figure 32. Single-instance block tridiagonal system.

Block pentadiagonal systems are represented in exactly the same manner, except that instead of three arrays of blocks (***c***, ***b***, and ***a***), there are five (***a*** through ***e***).

6.1.4 Need for Interface Blocks

If you supply an array section, rather than an entire array, for ***B***, ***a***, ***b***, ***c***, ***d***, or ***e***, you must use an interface block to ensure that the subsection axis corresponding to any array axis that is required to be serial, is also defined as serial. An example is provided below. For information about interface blocks and about passing array sections, refer to the CM Fortran documentation set.

In this example, the user application declares the input diagonals as follows:

```
      real , array (nblk,nblk,neqn) :: a,b,c,d,e
cmf$  layout a(:serial,:serial,:news)
```

```
cmf$  layout b(:serial,:serial,:news)
cmf$  layout c(:serial,:serial,:news)
cmf$  layout d(:serial,:serial,:news)
cmf$  layout e(:serial,:serial,:news)
      real , array (nblk,neqn) :: rhs
cmf$  layout rhs(:serial, :news)
```

In this case, the interface blocks are as follows:

```
      interface
      subroutine gen_banded_factor( sys_type,
     $     a, b, c, d, e,
     $     axis, work, type, pivot_value, nblock, ierror )
      implicit none

      real a(:,:,:) ,b(:,:,:) ,c(:,:,:) ,d(:,:,:), e(:,:,:)
cmf$  layout a(:serial,:serial,:news)
cmf$  layout b(:serial,:serial,:news)
cmf$  layout c(:serial,:serial,:news)
cmf$  layout d(:serial,:serial,:news)
cmf$  layout e(:serial,:serial,:news)

      integer sys_type, axis, work, type
      real  pivot_value
      integer  nblock, ierror
      end interface


      interface
      subroutine gen_banded_solve(sys_type,
     $     rhs, a, b, c, d, e,
     $     axis, work, ierror )
      implicit none

      real a(:,:,:) ,b(:,:,:), c(:,:,:) ,d(:,:,:), e(:,:,:)
      real rhs(:,:)
cmf$  layout a(:serial,:serial,:news)
cmf$  layout b(:serial,:serial,:news)
cmf$  layout c(:serial,:serial,:news)
cmf$  layout d(:serial,:serial,:news)
cmf$  layout e(:serial,:serial,:news)
cmf$  layout rhs(:serial,         :news)

      integer sys_type, axis, work, ierror
      end interface
```

The calls using array subsections are as follows, where $1 \leq s, t \leq neqn$:

```
      call gen_banded_factor(3, a(:,:,s:t), b(:,:,s:t),
     $     c(:,:,s:t), d(:,:,s:t), e(:,:,s:t), 3, work,
     $     1, 0, 0, ier)
      call gen_banded_solve(3, rhs(:,s:t), a(:,:,s:t),
     $     b(:,:,s:t), c(:,:,s:t), d(:,:,s:t), e(:,:,s:t),
     $     3, work, ier)
```

Banded System Factorization and Solver Routines (Unified)

Given one or more instances of a tridiagonal, block tridiagonal, pentadiagonal, or block pentadiagonal matrix A, the routines described below factor A, solve the system(s) $AX = B$ (where B is an array containing one or more right-hand sides), and overwrite B with the solution. Pairwise pivoting is available for tridiagonal systems. A and B must have the same data type (real or complex) and precision (single or double). In the syntax below, the solution X and the right-hand-side B are both represented by the array ***B***.

SYNTAX

sys_type = 0 (tridiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, vector_axis, work, type, pivot_value, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, vector_axis, work, ier*)
deallocate_banded (*work*)

sys_type = 1 (block tridiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, vector_axis, work, type, pivot_value, nblock, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, vector_axis, work, ier*)
deallocate_banded (*work*)

sys_type = 2 (pentadiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, **d**, **e**, vector_axis, work, type, pivot_value, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, **d**, **e**, vector_axis, work, ier*)
deallocate_banded (*work*)

sys_type = 3 (block pentadiagonal system):

gen_banded_factor (*sys_type, **a**, **b**, **c**, **d**, **e**, vector_axis, work, type, pivot_value, nblock, ier*)
gen_banded_solve (*sys_type, **B**, **a**, **b**, **c**, **d**, **e**, vector_axis, work, ier*)
deallocate_banded (*work*)

ARGUMENTS

sys_type Scalar integer variable indicating the type of system being solved. Must have one of the following values:

0 Tridiagonal

1 Block tridiagonal

2 Pentadiagonal

3 Block pentadiagonal

B CM array that contains one or more right-hand sides. Must have the same data type and precision as ***a***, ***b***, and ***c*** (for a tridiagonal system) or ***a***, ***b***, ***c***, ***d***, and ***e*** (for a pentadiagonal system). The solve routine overwrites this array with the solution.

For *sys_type* = 0 or 2 (tridiagonal or pentadiagonal systems), you may set up ***B*** in either of the following ways:

- ***B*** may have rank one greater than that of ***c***, ***b***, and ***a*** (for a tridiagonal system) or ***e***, ***d***, ***c***, ***b***, and ***a*** (for a pentadiagonal system). The first axis counts the right-hand sides and must be defined as **:serial**. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

- If there is only one right-hand side per instance, you may omit the first axis. That is, ***B*** may have the same rank as ***e***, ***d***, ***c***, ***b***, and ***a***. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

For *sys_type* = 1 or 3 (block tridiagonal or block pentadiagonal systems), you may set up ***B*** in either of the following ways:

- ***B*** may have rank equal to that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors to be multiplied by the blocks of A in the equation $Ax = B$. This axis must be defined as **:serial**, and has extent n if the blocks are $n \times n$. The second axis counts the right-hand sides, and must also be defined as **:serial**. Axis *vector_axis* counts the subvectors within each right-hand side. The remain-

ing axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

- If there is only one right-hand side per instance, you may omit the second axis. That is, ***B*** may have rank one less than that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors, must be defined as **:serial**, and has extent *n* if the blocks are *n* x *n*. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

a, b, c In tridiagonal or block tridiagonal systems: Real or complex CM arrays containing the elements or blocks that form the lower (***a***), main (***b***), and upper (***c***) diagonals of all instances of *A*. These three arrays must be distinct and must have the same shape, layout, data type, and precision. The first element or block along axis *vector_axis* axis of ***a***, and the last element or block along axis *vector_axis* of ***c***, are set to zero during execution.

For *sys_type* = 0 (tridiagonal systems), each array must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For *sys_type* = 1 (block tridiagonal systems), each array must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of *A*. These axes must be defined as **:serial**, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all three arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

a, b, c, d, e In pentadiagonal or block pentadiagonal systems: Real or complex CM arrays containing the elements or blocks that form the five diagonals of all instances of *A*. The array ***a*** represents the lowermost diagonal; the array ***e*** represents the uppermost diagonal. These five arrays must be distinct, but must all have the same shape, layout, data type, and precision. The first element of ***b***, the last element of ***d***, the first two elements of ***a***, and the last two elements of ***e*** are set to zero during execution.

For *sys_type* = 2 (pentadiagonal systems), ***a, b, c, d,*** and ***e*** must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For *sys_type* = 3 (block pentadiagonal systems), ***a, b, c, d,*** and ***e*** must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of *A*. These axes must be defined as **:serial**, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all five arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

vector_axis — Scalar integer variable. The axis of ***a, b, c, d,*** and ***e*** along which the diagonal elements or blocks of *A* lie. The value of *vector_axis* must be at least 1, but less than or equal to the rank of ***a, b, c, d,*** and ***e***). Performance is best if the axis identified by *vector_axis* is defined as **:serial**, and second best if it is defined as NEWS-ordered.

work — Integer front-end array of rank 1 and extent ≥ 20. Internal variable. Upon completion of a factor routine, *work* contains information required by the associated solve routine.

type — Scalar integer that has one of the symbolic constant values (or equivalent numeric values) listed below. Selects the algorithm.

CMSSL_pipeline_ge (3)
Pipelined Gaussian elimination.

CMSSL_pge_piv (9)
Pipelined Gaussian elimination with pairwise pivoting. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine uses **CMSSL_pipeline_ge** instead.

CMSSL_pge_piv_val (10)
Pipelined Gaussian elimination with pairwise pivoting; replace zero pivots with a supplied value. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine uses **CMSSL_pipeline_ge** instead.

CMSSL_substr_cr (1)
Substructuring with cyclic reduction.

CMSSL_substr_bcr (4)
Substructuring with balanced cyclic reduction.

CMSSL_substr_pge (2)
Substructuring with pipelined Gaussian elimination.

CMSSL_substr_transp (5)
Substructuring with transpose. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns *ier* = -5.

If the axis along which the diagonal elements or blocks lie (axis *vector_axis*) is serial, the routine *always* uses Gaussian elimination (with pivoting, if you selected pivoting and have a tridiagonal system).

pivot_value Scalar variable of the same data type as the banded system. When *type* = 10, this value replaces any zero pivots the routine encounters. This value is ignored if *type* is not equal to 10.

nblock Scalar integer variable. Specifies the blocking factor used internally in the calculation of inverses. Must be < *n*. If you set *nblock* to 0, the routines choose a predefined value that depends on the size of the blocks you supply. For systems where $n \geq 32$, there may be some benefit in experimenting with *nblock* to obtain optimum performance. If you set *nblock* to an invalid value (for example, a negative number), the routines use a blocking factor of 1.

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

-1 Input arrays have inconsistent ranks.

-2 Axes that should be serial are not.

-3 Input arrays have inconsistent data types.

-4 Returned when *sys_type* = 1 or 3. The first two axes of ***a***, ***b***, ***c***, ***d***, or ***e*** do not have equal extents;

or, the ***B*** axis that counts the elements within the subvectors to be multiplied by the blocks of *A* does not have the same extent as the first axis of ***a***, ***b***, ***c***, ***d***, or ***e***.

-5 Invalid *type*.

-6 Invalid *vector_axis*.

-7 The input arrays are not conformable.

-8 There is an error in the *work* parameter.

1000 A zero pivot was encountered when **CMSSL_pge_piv** was specified.

DESCRIPTION

The banded system factorization and solver routines perform the following operations:

gen_banded_factor Given tridiagonal or block tridiagonal matrices *A* (represented by three arrays), or pentadiagonal, or block pentadiagonal matrices *A* (represented by five arrays), this routine performs the factorization $A = LU$ for each matrix, where *L* and *U* are lower and upper (respectively) bidiagonal or block bidiagonal, or lower and upper (respectively) tridiagonal or block tridiagonal matrices, or permutations thereof.

gen_banded_solve Given the factors computed by **gen_tridiag_factor**, and corresponding arrays *B* each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each *B* with the solution.

deallocate_banded This routine deallocates the memory required by the factorization and solver routines.

Separation of the Factorization and Solution Phases. Separation of the factorization and solution phases allows you to factor one or more instances of a matrix once, and then call the solve routine multiple times, supplying the same factors but different right-hand-side arrays each time — thus avoiding the overhead of repeated factorization of the same matrices.

Upon return from the factor routine, the three or five arrays that contained the matrices *A* on input contain information required by the solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Memory Allocation and Deallocation. Each time you call the factor routine, a buffer, represented by the *work* argument, is allocated in memory. When you call the solve routine, you must supply the value returned in *work* by the associated previous factor call. The *work* buffer remains allocated until you explicitly deallocate it with **deallocate_banded**. Thus, you can call the factor routine, perform other operations, and later call the solve routine one or more times. You can also factor other sets of matrices, thus creating different *work* buffers, and keep multiple *work* buffers allocated at the same time.

Be sure to call **deallocate_banded** to deallocate buffer space whenever you have finished working with one set of factors. If you call the factor routine repeatedly without deallocating buffer space, you will eventually run out of memory.

NOTES

Private Argument Values. The internal variable *work* is required for communicating information between the factorization and solver phases. The application must not modify the contents of this variable.

Preservation of Argument Values. Upon return from the factor routine, the arrays ***a, b, c, d,*** and ***e*** contain information required by the solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Distinct Variables. No overlapping of variables is allowed in these routines.

Caution. The buffer space associated with *work* depends on the size of the matrix or matrices being factored. Therefore, if you call the factor routine more than once, be sure to call **deallocate_banded** to deallocate the associated buffer space between factorization calls, or use a different *work* array. Otherwise, a second call with the same *work* array will allocate different buffer space but represent it with the same *work* value as in the first call, and the buffer space associated with the first call will become inaccessible.

Performance Hints. Performance is best if the axes listed below are defined as **:serial,** and second-best if they are defined as NEWS-ordered.

- The axis of ***a***, ***b***, ***c***, ***d***, and ***e*** along which the blocks or elements lie (axis *vector_axis*).
- The axis of ***B*** along which the subvectors or elements lie.

If you are working with a single- or multiple-instance elementwise tridiagonal or pentadiagonal system with one right-hand side, and axis *axis* is local to a processing element, you will probably achieve better performance by writing the operation in CM Fortran than by using the CMSSL banded system solver routines. This is especially true in the case of pentadiagonal systems.

ADI Applications. The multiple-instance implementation of the banded system solvers is excellent for applications of the alternating-direction implicit method, where a solution along each axis is required.

Need for Interface Blocks. If you supply an array section, rather than an entire array, for ***B***, ***a***, ***b***, ***c***, ***d***, or ***e***, you must use an interface block to ensure that the subsection axis corresponding to any array axis that is required to be serial, is also defined as serial. For information about interface blocks and about passing array sections, refer to the CM Fortran documentation set.

EXAMPLES

Sample CM Fortran code that uses the banded system factorization and solver routines can be found on-line in the subdirectory

```
tridiag/cmf/
```

of a CMSSL examples directory whose location is site-specific.

6.2 Banded System Factorization and Solver Routines

The banded system factorization and solver routines described in this section are provided primarily for compatibility with the CM-200. They include a factorization routine, a solver routine, and a combined factorization and solver routine for each of the banded system types (tridiagonal, pentadiagonal, block tridiagonal, and block pentadiagonal).

These routines support multiple instances and accept either real or complex data. They provide the same algorithms as the unified banded system routines (see Section 6.1.2), with the following exceptions:

- The **gen_tridiag_solve** routine does not allow you to specify an algorithm. It uses substructuring with cyclic reduction (**CMSSL_substr_cr**), unless the axis along which the diagonal elements lie (axis *vector_axis* in the argument list) is serial (the recommended layout), in which case standard Gaussian elimination is used.

- The algorithm **CMSSL_pge_piv_val** is not available with these routines; that is, you cannot supply a pivot value.

Like the unified banded system routines, the routines described in this section require an interface block if you supply an array section, rather than an entire array, for the ***B***, ***a***, ***b***, ***c***, ***d***, or ***e*** arguments. Refer to Section 6.1.4 for an example.

The man page that follows provides calling sequences, argument definitions, and usage information. Data is set up in the same way as for the unified banded system routines.

Banded System Factorization and Solver Routines

Given one or more instances of a tridiagonal, block tridiagonal, pentadiagonal, or block pentadiagonal matrix A, the routines described below factor A, solve the system(s) $AX = B$ (where B is an array containing one or more right-hand sides), and overwrite B with the solution. Pairwise pivoting is available for tridiagonal systems. A and B must have the same data type (real or complex) and precision (single or double). In the syntax below, the solution X and the right-hand-side B are both represented by the array ***B***.

SYNTAX

gen_tridiag_factor (*c, b, a, vector_axis, tolerance, work, type, ier*)

gen_tridiag_solve_factored (*B, c, b, a, vector_axis, tolerance, work, type, ier*)

gen_tridiag_solve (*B, c, b, a, vector_axis, tolerance, ier*)

gen_pentadiag_factor (*e, d, c, b, a, vector_axis, tolerance, work, type, ier*)

gen_pentadiag_solve_factored (*B, e, d, c, b, a, vector_axis, tolerance, work, type, ier*)

gen_pentadiag_solve (*B, e, d, c, b, a, vector_axis, tolerance, type, ier*)

block_tridiag_factor (*c, b, a, vector_axis, tolerance, work, type, nblock, ier*)

block_tridiag_solve_factored (*B, c, b, a, vector_axis, tolerance, work, type, ier*)

block_tridiag_solve (*B, c, b, a, vector_axis, tolerance, type, nblock, ier*)

block_pentadiag_factor (*e, d, c, b, a, vector_axis, tolerance, work, type, nblock, ier*)

block_pentadiag_solve_factored (*B, e, d, c, b, a, vector_axis, tolerance, work, type, ier*)

block_pentadiag_solve (*B, e, d, c, b, a, vector_axis, tolerance, type, nblock, ier*)

deallocate_banded_solve (*work*)

ARGUMENTS

In the descriptions below, the following terms are used to refer to the banded system routines: The *factor routines* are those ending with the suffix **_factor**. The *solve rou-*

tines are those ending with the suffix **_solve_factored**. The *factor-and-solve routines* are those ending with the suffix **_solve**. The routines prefixed with **gen_** are referred to as the *elementwise* routines; the routines prefixed with **block_** are the *block* routines.

B CM array that contains one or more right-hand sides. Must have the same data type and precision as ***c***, ***b***, and ***a*** (for a tridiagonal system) or ***e***, ***d***, ***c***, ***b***, and ***a*** (for a pentadiagonal system). The solve and factor-and-solve routines overwrite this array with the solution.

For the elementwise banded system routines, you may set up ***B*** in either of the following ways:

- ***B*** may have rank one greater than that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the right-hand sides and must be defined as **:serial**. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

- If there is only one right-hand side per instance, you may omit the first axis. That is, ***B*** may have the same rank as ***e***, ***d***, ***c***, ***b***, and ***a***. Axis *vector_axis* counts the elements within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

For the block banded system routines, you may set up ***B*** in either of the following ways:

- ***B*** may have rank equal to that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors to be multiplied by the blocks of A in the equation $Ax = B$. This axis must be defined as **:serial**, and has extent n if the blocks are $n \times n$. The second axis counts the right-hand sides, and must also be defined as **:serial**. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

- If there is only one right-hand side per instance, you may omit the second axis. That is, ***B*** may have rank one less than that of ***e***, ***d***, ***c***, ***b***, and ***a***. The first axis counts the elements within the subvectors, must be defined as **:serial**,

Version 3.1, June 1993

and has extent *n* if the blocks are *n* x *n*. Axis *vector_axis* counts the subvectors within each right-hand side. The remaining axes are instance axes that match those of ***e***, ***d***, ***c***, ***b***, and ***a*** in extent, layout, and order of declaration.

c, b, a In tridiagonal or block tridiagonal systems: Real or complex CM arrays containing the elements or blocks that form the upper (***c***), main (***b***), and lower (***a***) diagonals of all instances of *A*. These three arrays must be distinct and must have the same shape, layout, data type, and precision. The last element or block along axis *vector_axis* of ***c***, and the first element or block along axis *vector_axis* of ***a***, are set to zero during execution.

For the elementwise tridiagonal routine, each array must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For the block tridiagonal routine, each array must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of *A*. These axes must be defined as **:serial**, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in the same order in all three arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

e, d, c, b, a In pentadiagonal or block pentadiagonal systems: Real or complex CM arrays containing the elements or blocks that form the five diagonals of all instances of *A*. The array ***a*** represents the lowermost diagonal; the array ***e*** represents the uppermost diagonal. These five arrays must be distinct, but must all have the same shape, layout, data type, and precision. The first element of ***b***, the last element of ***d***, the first two elements of ***a***, and the last two elements of ***e*** are set to zero during execution.

For the elementwise pentadiagonal routine, ***e***, ***d***, ***c***, ***b***, and ***a*** must have rank greater than or equal to 1. The *vector_axis* argument identifies the axis along which the diagonal elements lie.

For the block pentadiagonal routine, ***e***, ***d***, ***c***, ***b***, and ***a*** must have rank greater than or equal to 3. The first two axes count the rows and columns of the blocks of *A*. These axes must be defined as **:serial**, and have the same extent since the blocks must be square. The remaining axes include the instance axes (if any) and the axis along which the blocks lie. These remaining axes must occur in

the same order in all five arrays. The *vector_axis* argument identifies the axis along which the blocks lie.

vector_axis Scalar integer variable. The axis of ***e***, ***d***, ***c***, ***b***, and ***a*** along which the diagonal elements or blocks of *A* lie. The value of *vector_axis* must be at least 1, but less than or equal to the rank of ***e***, ***d***, ***c***, ***b***, and ***a***. Performance is best if the axis identified by *vector_axis* is defined as **:serial**, and second best if it is defined as NEWS-ordered.

tolerance Scalar real variable. Ignored on the CM-5.

work Integer front-end array of rank 1 and extent ≥ 20. Internal variable. Upon completion of a factor routine, *work* contains information required by the associated solve routine.

type Integer that has one of the following symbolic constant values (or the equivalent numeric value):

CMSSL_pipeline_ge (3)
Pipelined Gaussian elimination.

CMSSL_pge_piv (9)
Pipelined Gaussian elimination with pairwise pivoting. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine uses **CMSSL_pipeline_ge** instead.

CMSSL_substr_cr (1)
Substructuring with cyclic reduction.

CMSSL_substr_bcr (4)
Substructuring with balanced cyclic reduction.

CMSSL_substr_pge (2)
Substructuring with pipelined Gaussian elimination.

CMSSL_substr_transp (5)
Substructuring with transpose. This algorithm is available for tridiagonal systems only. If you specify it with a pentadiagonal or block system, the routine returns *ier* = -5.

If the axis along which the diagonal elements or blocks lie (axis *vector_axis*) is serial, the routines *always* use pipelined Gaussian elimination (with pivoting, if you selected pivoting and have a tridiagonal system).

nblock Scalar integer variable. Specifies the blocking factor used internally in the calculation of inverses. Must be < *n*. If you set *nblock* to 0, the routines choose a predefined value that depends on the size of the blocks you supply. For systems where $n \geq 32$, there may be some benefit in experimenting with *nblock* to obtain optimum performance. If you set *nblock* to an invalid value (for example, a negative number), the routines use a blocking factor of 1.

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

- -1 Input arrays have inconsistent ranks.
- -2 Axes that should be serial are not.
- -3 Input arrays have inconsistent data types.
- -4 In one of the block routines, the first two axes of ***e, d, c, b,*** or ***a*** do not have equal extents; or, the ***B*** axis that counts the elements within the sub-vectors to be multiplied by the blocks of *A* does not have the same extent as the first axis of ***e, d, c, b,*** or ***a.***
- -5 Invalid *type*.
- -6 Invalid *vector_axis*.
- -7 The input arrays are not conformable.
- -8 There is an error in the *work* parameter.

DESCRIPTION

The banded system factorization and solver routines perform the operations listed below. The factorization routine performs the factorization $A = LU$ for each matrix A, where L and U are lower and upper (respectively) bidiagonal or block bidiagonal, or

lower and upper (respectively) tridiagonal or block tridiagonal matrices, or permutations thereof.

gen_tridiag_factor Given tridiagonal matrices A (represented by three arrays), this routine factors the matrices.

gen_tridiag_solve_factored Given the factors computed by **gen_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

gen_tridiag_solve Given tridiagonal matrices A (represented by three arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX $(=LUX) = B$, and overwrites each B with the solution.

gen_pentadiag_factor Given pentadiagonal matrices A (represented by five arrays), this routine factors the matrices.

gen_pentadiag_solve_factored Given the factors computed by **gen_pentadiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

gen_pentadiag_solve Given pentadiagonal matrices A (represented by five arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX $(=LUX) = B$, and overwrites each B with the solution.

block_tridiag_factor Given block tridiagonal matrices A (represented by three arrays), this routine factors the matrices.

block_tridiag_solve_factored Given the factors computed by **block_tridiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to $LUX = B$, and overwrites each B with the solution.

block_tridiag_solve Given block tridiagonal matrices A (represented by three arrays), and corresponding arrays B each

containing one or more right-hand-side vectors, this routine computes the solutions to AX (=LUX) = B, and overwrites each B with the solution.

block_pentadiag_factor Given block pentadiagonal matrices A (represented by five arrays), this routine factors the matrices.

block_pentadiag_solve_factored Given the factors computed by **block_pentadiag_factor**, and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to LUX = B, and overwrites each B with the solution.

block_pentadiag_solve Given block pentadiagonal matrices A (represented by five arrays), and corresponding arrays B each containing one or more right-hand-side vectors, this routine computes the solutions to AX (=LUX) = B, and overwrites each B with the solution.

deallocate_banded_solve This routine deallocates the memory required by the above factorization and solver routines.

Separation of the Factorization and Solution Phases. Calling a factor routine followed by the associated solve routine is equivalent to calling the associated factor-and-solve routine. However, separation of the factorization and solution phases allows you to factor one or more instances of a matrix once, and then call the appropriate solve routine multiple times, supplying the same factors but different right-hand-side arrays each time — thus avoiding the overhead of repeated factorization of the same matrices.

Upon return from a factor routine, the three or five arrays that contained the matrices A on input contain information required by the corresponding solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Memory Allocation and Deallocation. Each time you call one of the factor routines, a buffer, represented by the *work* argument, is allocated in memory. When you call one of the solve routines, you must supply the value returned in *work* by the associated previous factor call. The *work* buffer remains allocated until you explicitly deallocate it with **deallocate_banded_solve**. Thus, you can call a factor routine, perform other operations, and later call the corresponding solve routine one or more times. You can also factor other sets of matrices, thus creating different *work* buffers, and keep multiple *work* buffers allocated at the same time.

Be sure to call **deallocate_banded_solve** to deallocate buffer space whenever you have finished working with one set of factors. If you call a factor routine repeatedly without deallocating buffer space, you will eventually run out of memory.

NOTES

Private Argument Values. The internal variable *work* is required for communicating information between the factorization and solver phases. The application must not modify the contents of this variable.

Preservation of Argument Values. Upon return from a factor routine, the arrays ***e, d, c, b,*** and ***a*** contain information required by the corresponding solve routine. The contents of these arrays must therefore be preserved between the factor and solve calls.

Distinct Variables. No overlapping of variables is allowed in these routines.

Deallocation. Be sure to call **deallocate_banded_solve** to deallocate buffer space whenever you have finished working with one set of factors. If you call a factor routine repeatedly without deallocating buffer space, you will eventually run out of memory.

Caution. The buffer space associated with *work* depends on the size of the matrix or matrices being factored. Therefore, if you call a factor routine more than once, be sure to call **deallocate_banded_solve** to deallocate the associated buffer space between factorization calls, or use a different *work* array. Otherwise, a second call with the same *work* array will allocate different buffer space but represent it with the same *work* value as in the first call, and the buffer space associated with the first call will become inaccessible.

Performance Hints. Performance is best if the axes listed below are defined as **:serial,** and second-best if they are defined as NEWS-ordered.

- The axis of ***a, b, c, d,*** and ***e*** along which the blocks or elements lie (axis *vector_axis*).
- The axis of ***B*** along which the subvectors or elements lie.

If you are working with a single- or multiple-instance elementwise tridiagonal or pentadiagonal system with one right-hand side, and axis *axis* is local to a processing element, you will probably achieve better performance by writing the operation in CM Fortran than by using the CMSSL banded system solver routines. This is especially true in the case of pentadiagonal systems.

ADI Applications. The multiple-instance implementation of the banded system solvers is excellent for applications of the alternating-direction implicit method, where a solution along each axis is required.

Need for Interface Blocks. If you supply an array section, rather than an entire array, for ***B***, ***a***, ***b***, ***c***, ***d***, or ***e***, you must use an interface block to ensure that the subsection axis corresponding to any array axis that is required to be serial, is also defined as serial. For information about interface blocks and about passing array sections, refer to the CM Fortran documentation set.

EXAMPLES

Sample CM Fortran code that uses the banded system factorization and solver routines can be found on-line in the subdirectory

```
tridiag/cmf/
```

of a CMSSL examples directory whose location is site-specific.

6.3 References

For more information about banded system solvers, see the following references:

1. Buneman, O. *A Compact Non-Iterative Poisson Solver.* Stanford University Institute for Plasma Research, Report No. 294, 1969.

2. Buzbee, B. L., G. H. Golub, and C. W. Nielson. On Direct Methods for Solving Poisson's Equations. *SIAM J. Num. Analysis* **7** (1970): 627–56.

3. Golub, G. H., and C. F. Van Loan. *Matrix Computations.* 2d ed. Baltimore: Johns Hopkins University Press, 1989. Pp. 194–220.

4. Ho, C. and S. L. Johnsson. Optimizing Tridiagonal Solvers for Alternating Direction Methods on Boolean Cube Multiprocessors. *SIAM J. Sci. Stat. Comput.* **11**, no. 3 (1990): 563–92.

5. Hockney, R. W., and C. R. Jesshope. *Parallel Computers 2.* Bristol and Philadelphia: Adam Hilger, 1988. Pp. 475–89.

6. Johnsson, S. L. Solving Tridiagonal Systems on Ensemble Architectures. *SIAM J. Sci. Stat. Comp.* **8**, no. 3 (1987): 354–92.

7. Malcom M.A. and J. Palmer. A Fast Method for Solving a Class of Tridiagonal Systems of Linear Equations. *Comm. ACM* **17** (1974): 14–17.

8. Wilkinson, J. H. *The Algebraic Eigenvalue Problem.* Oxford: Clarendon Press, 1965.

Chapter 7

Iterative Solvers

This chapter describes the Krylov-based iterative solvers included in the CMSSL. Section 7.2 provides references.

7.1 Krylov-Based Iterative Solvers

One important approach to solving large sparse linear systems is the use of iterative solvers based on Krylov subspace techniques. A well-known example of these techniques is the conjugate gradient (CG) method, which is used for symmetric positive definite systems. CG is a Lanczos-based method that reduces the problem matrix to a symmetric tridiagonal matrix. There are also Lanczos-based algorithms for non-symmetric systems; these entail the numerical problems associated with non-symmetric tridiagonal systems. Another class of non-symmetric algorithms is based on the Arnoldi procedure with its greater computational and storage requirements. Except for the Arnoldi-based restarted GMRES algorithm, all the algorithms currently included in the CMSSL are Lanczos-based algorithms.

For psuedo-code and references for many of the non-symmetric Lanczos-based algorithms, see reference 1 listed in Section 7.2. References 2 and 3 also supply useful background.

7.1.1 CMSSL Iterative Solver Routines

Given a matrix A, a right-hand-side vector b, and a preconditioner $M = M_1 * M_2$ such that $A^{\sim} = M_1^{-1} A M_2^{-1}$, the **gen_iter_solve** routine (together with its

associated setup and deallocation routines) solves the system $Ax = b$ using Krylov space iterative methods. Any matrix operations (that is, matrix vector or vector matrix products) and preconditioning steps (for example, solve $My = z$) are provided by the user using a reverse communication interface. The type of matrix and its internal representation are completely arbitrary, and depend on the user application. Similarly, the vectors can be represented by any rank array.

For details about the syntax, arguments, and usage of the iterative solvers, refer to the man page at the end of this section.

7.1.2 Algorithms

The iterative solvers offer the algorithms listed below. For detailed information about the algorithms, see the indicated references. (Full references are provided in Section 7.2.)

CMSSL_cg	Conjugate gradient. A Lanczos-based algorithm for symmetric positive definite systems. (Note that this method will not work for non-symmetric systems.) See reference 4.
CMSSL_cgs	Conjugate gradient squared of Sonneveld. See reference 5.
CMSSL_bcg	Bi-conjugate gradient of Fletcher. See reference 6.
CMSSL_bicgstab	Bi-conjugate gradient with stabilization of Van der Vorst. See reference 7.
CMSSL_bicgstab2	Bi-conjugate gradient with stabilization of Gutknecht. See reference 8.
CMSSL_gmres	Restarted Generalized Minimal Residual algorithm. See reference 9.
CMSSL_qmrcgs	Transpose-free Quasi-Minimal Residual (QMR) algorithm of Freund. See reference 10.
CMSSL_qmrlal	QMR with with a three-term look-ahead Lanczos algorithm of Freund, Gutknecht

	and Nachtigal. See references 11, 12, and 13.
CMSSL_qmr2	QMR based on two-term recursions for generating the Lanczos basis vectors without look-ahead. See reference 14.
CMSSL_qmrs	QMR squared of Freund and Szeto. See reference 15.
CMSSL_qmrbicgstab	QMR based on BICSTAB of Chan, Szeto and Tong. See reference 16.
CMSSL_qmrbicgstab2	A modified version of QMRBICGSTAB. See reference 16.
CMSSL_qcgs	Quasi-minimized CGS. See reference 17.

7.1.3 Acknowledgments

We wish to thank Roland Freund and Noel Nachtigal for providing us with the original Fortran 77 version of their code. We have converted their code to CM Fortran and included it in the algorithms available in **gen_iter_solve**.

7.1.4 Example

The example below shows how to use the iterative solvers, and is based on the information in the man page. The application in this example provides three routines:

trid_matvec (*z, y, a, b, c, n*)	Multiplies *Ay* and places the results in *z*. *A* is represented by the diagonals ***a***, ***b***, and ***c***.
trid_vecmat (*z, y, a, b, c, n*)	Multiplies *yA* and places the results in *z*. *A* is represented by the diagonals ***a***, ***b***, and ***c***.
diag_solve (*z, y, M, n*)	Solves the system $Mz = y$.

```
      call gen_iter_solve_setup(setup_iter,x,info,ier)

      ido = CMSSL_IDO_START
      do while ( (ido .ge. CMSSL_IDO_SOLVE_MIN) .and. (ido .le.
     &            CMSSL_IDO_SOLVE_MAX ) )

         call gen_iter_solve(ido,x,b,z,y,info,finfo,setup_iter
     &                      ,ier)


         reverse_comm: select case ( ido )

         case ( CMSSL_IDO_AY )
c
c        user supplied  z = A y
c
            call trid_matvec(z,y,a,b,c,n)

         case ( CMSSL_IDO_ATY )
c
c        user supplied  z = (A)T y
c
            call trid_vecmat(z,y,a,b,c,n)

         case ( CMSSL_IDO_SOLVE_M )
c
c        user supplied  z = (M)-1  y
c
            call diag_solve(z,y,M_1,n)

         case ( CMSSL_IDO_SOLVE_MT )
c
c        user supplied  z = ((M)T)-1  y
c
            call diag_solve(z,y,M_1,n)

         case ( CMSSL_IDO_SOLVE_M1 )
c
c        user supplied  z = (M1)-1  y
c
            call diag_solve(z,y,M1_1,n)

         case ( CMSSL_IDO_SOLVE_M2T )
c
c        user supplied  z = ((M2)T)-1  y
c
            call diag_solve(z,y,M2_1,n)

         case ( CMSSL_IDO_SOLVE_M2 )
c
```

```
c        user supplied  z = (M2)-1  y
c
            call diag_solve(z,y,M2_1,n)

         end select reverse_comm

      enddo


      call deallocate_iter_solve(setup_iter)
```

Iterative Solvers

Given a matrix A, a right-hand-side vector b, and a preconditioner $M = M_1 * M_2$ such that $A^{\sim} = M_1^{-1} A M_2^{-1}$, the routines described below solve the system $Ax = b$ using Krylov space iterative methods. Any matrix operations (that is, matrix vector or vector matrix products) and preconditioning steps (for example, solve $My = z$) are provided by the user using a reverse communication interface. The type of matrix and its internal representation are completely arbitrary, and depend on the user application. Similarly, the vectors can be represented by any rank array.

SYNTAX

gen_iter_solve_setup (*setup, x_template, info, ier*)

gen_iter_solve (*ido, x, b, z, y, info, finfo, setup, ier*)

deallocate_iter_solve (*setup*)

ARGUMENTS

setup — Scalar integer variable. Internal variable. The initial value you supply to **gen_iter_solve_setup** is ignored. When you call **gen_iter_solve** or **deallocate_iter_solve**, supply the value assigned to *setup* by the associated setup call. Do not change the value of *setup* after the setup routine returns.

x_template — Real (single- or double-precision) CM array with the same shape and layout as ***x***.

info — Integer front-end array of rank 1 and length **CMSSL_iter_info_size**. When you call **gen_iter_solve_setup**, set *info* as indicated below. *Do not* change the values of *info* after the setup routine returns. Upon return from **gen_iter_solve**, *info*(**CMSSL_iter_iter**) contains the current (last) iteration step number; *info*(**CMSSL_iter_kspace_used**) contains the current size of the Lanczos subspace used in restarted GMRES or the current number of Lanczos vectors used by the look-ahead Lanczos algorithm (QMRLAL).

The values of the symbolic constants are defined in the CMSSL header file.

Version 3.1, June 1993

info(**CMSSL_Iter_algorithm**) Specifies the algorithm to be used. Supply one of the values listed below. For references, see Sections 7.1.2 and 7.2.

CMSSL_cg	Conjugate gradient. A Lanczos-based algorithm for symmetric positive definite systems. (Note that this method will not work for non-symmetric systems.)
CMSSL_cgs	Conjugate gradient squared of Sonneveld.
CMSSL_bcg	Bi-conjugate gradient of Fletcher.
CMSSL_bicgstab	Bi-conjugate gradient with stabilization of Van der Vorst.
CMSSL_bicgstab2	Bi-conjugate gradient with stabilization of Gutknecht.
CMSSL_gmres	Restarted Generalized Minimal Residual algorithm.
CMSSL_qmrcgs	Transpose-free Quasi-Minimal Residual (QMR) algorithm of Freund.
CMSSL_qmrlal	QMR with with a three-term look-ahead Lanczos algorithm of Freund, Gutknecht and Nachtigal.
CMSSL_qmr2	QMR based on two-term recursions for generating the Lanczos basis vectors without look-ahead.
CMSSL_qmrs	QMR squared of Freund and Szeto.
CMSSL_qmrbicgstab	QMR based on BICSTAB of Chan, Szeto and Tong.

CMSSL_qmrbicgstab2 A modified version of QMRBICGSTAB.

CMSSL_qcgs Quasi-minimized CGS.

info(**CMSSL_iter_init**) Determines the contents of the initial guess, x_0. Set to 0 to specify $x_0 = 0$; set to 1 to use the initial input value of x for x_0.

info(**CMSSL_iter_random_start**) Determines the initial residual value, r_0. Set to 0 to specify $r_0 = b - Ax_0$; set to 1 to use a random value for r_0.

info(**CMSSL_iter_maxiter**) Maximum number of iterations.

info(**CMSSL_iter_precond**) Set to 1 for preconditioning, or 0 for no preconditioning. (See Description section for a discussion of preconditioning.)

info(**CMSSL_iter_kspace**) If

info(**CMSSL_iter_algorithm**) = **CMSSL_gmres**

this parameter specifies the maximum size of the Lanczos subspace used by GMRES (the maximum number of Lanczos vectors stored between restarts).

If

info(**CMSSL_iter_algorithm**) = **CMSSL_qmrlal**

this parameter specifies the maximum number of Lanc-

zos vectors to look ahead on breakdown.

info(**CMSSL_iter_omega**) Specifies the convergence criterion, ω. Set to 1 to use a convergence criterion of

$$\omega = \| r \|_2 / \| b \|_2$$

or set to 2 to use a convergence criterion of ω =

$$\| r \|_\infty / \| A \|_\infty \| x \|_1 + \| b \|_\infty$$

where

$$r = b - Ax$$
$$\| r \|_n = (|r(1)|^n + |r(2)|^n + \ldots)^{1/n}$$
$$\| r \|_\infty = \max |r(i)|$$

For **CMSSL_gmres**, the convergence criterion ω is set to the magnitude of the last Givens rotation used in reducing the upper Hessenberg matrix to upper triangular form.

info(**CMSSL_iter_residual**) The value you supply is used only if you specified one of the following algorithms:

CMSSL_qmrcgs
CMSSL_qmr2
CMSSL_qmrs
CMSSL_qmrlal
CMSSL_qmrbicgstab
CMSSL_qmrbicgstab2
CMSSL_qcgs

If set to 0, this parameter causes **gen_iter_solve** to test an estimated residual for convergence against the convergence criterion, ω

(see above) at the end of each iteration loop.

If you set this parameter to any value greater than 0, **gen_iter_solve** explicitly computes $r = b - Ax$ every *info*(**CMSSL_iter_residual**) iterations. Thus, the extra matrix vector product needed to compute r can be amortized over several iterations. Convergence is only checked every *info*(**CMSSL_iter_residual**) iterations.

If *info*(**CMSSL_algorithm**) = **CMSSL_qmrlal**, $r = b - Ax$ is explicitly computed every iteration when *info*(**CMSSL_residual**) is not equal to 0.

info(**CMSSL_iter_output_x**)

If you set this parameter to $m > 0$, x will contain intermediate values of the solution before convergence or breakdown every m iteration steps. If $m \geq$ *info*(**CMSSL_iter_maxiter**), x is guaranteed to contain the most recent value on return only when *ido* = **CMSSL_ido_end**, **CMSSL_ido_error**, or **CMSSL_ido_ breakdown**.

If you set *info*(**CMSSL_iter_output_x**) ≤ 0, it is reset to *info*(**CMSSL_iter_maxiter**). In this case, x is guaranteed to contain the most recent value on return only when *ido* = **CMSSL_ido_end**.

info(**CMSSL_iter_kspace_used**) The input value is ignored. If *info*(**CMSSL_iter_algorithm**) = **CMSSL_gmres**, this parameter returns the size of the Lanczos subspace used by restarted GMRES during the current iteration loop.

If

info(**CMSSL_iter_algorithm**) = **CMSSL_qmrlal**

this parameter returns the number of Lanczos vectors used during the look-ahead procedure to avoid breakdown during the current iteration loop.

info(**CMSSL_iter_iter**) The input value is ignored. This parameter returns the current iteration loop count.

ido Scalar integer variable used for reverse communication. The first time you call **gen_iter_solve** in a reverse communication loop, set *ido* = **CMSSL_ido_start**. On return, *ido* is set to one of the values listed below. All symbolic constants are defined in the CMSSL header file. M is the preconditioner, $M = M_1M_2$, such that $A^{\sim} = M_1^{-1}AM_2^{-1}$.

CMSSL_ido_Ay The user application must perform the matrix vector multiplication Ay and place the results in z.

CMSSL_ido_ATy The user application must perform the matrix transpose vector multiplication, A^Ty, or equivalently the vector matrix product, y^TA, and place the results in z.

CMSSL_ido_solve_M The user application must solve the system $Mz = y$ (compute $M^{-1}y$ and

place the results in z). If M is the identity I, replace z by the value in y.

CMSSL_ido_solve_MT The user application must solve the system $M^T z = y$ (compute $M^{-T}y$ and place the results in z). If M is the identity I, replace z by the value in y.

CMSSL_ido_solve_M1 The user application must solve the system $M_1 z = y$ (compute $M_1^{-1}y$ and place the results in z). If M_1 is the identity I, replace z by the value in y.

CMSSL_ido_solve_M2T The user application must solve the system $M_2^T z = y$ (compute $M_2^{-T}y$ and place the results in z). If M_2 is the identity I, replace z by the value in y.

CMSSL_ido_solve_M2 The user application must solve the system $M_2 z = y$ (compute $M_2^{-1}y$ and place the results in z). If M_2 is the identity I, replace z by the value in y.

CMSSL_ido_end Convergence has occurred; that is, the convergence criterion ω is $\leq$ the initial input value of *finfo*(**CMSSL_iter_tol**).

CMSSL_ido_error The maximum number of iterations specified in *info*(**CMSSL_iter_maxiter**) has been reached without convergence.

CMSSL_ido_breakdown A breakdown in the algorithm (for example, division by 0) has occurred.

During the reverse communication loop, when **CMSSL_ido_solve_min** $\leq$ *ido* $\leq$ **CMSSL_ido_solve_max**, the algorithm is running normally and has not yet converged. If the value of *ido* falls outside this range, the algorithm has terminated without convergence (that is, either the maximum number of iterations has

Version 3.1, June 1993

been reached or the algorithm has suffered a breakdown from which it cannot continue).

x Real (single- or double-precision) CM array of any rank and shape. On input, supply the initial guess to be used when *info*(**CMSSL_iter_init**) = 1. If you set *info*(**CMSSL_iter_output_x**) = $m > 0$, the intermediate value of the solution is returned in ***x*** every m iterations. When *ido* = **CMSSL_ido_end**, ***x*** contains the converged solution. When *ido* = **CMSSL_ido_error** or **CMSSL_ido_breakdown**, the value of ***x*** in the current iteration loop is returned.

b Real (single- or double-precision) CM array with the same shape, layout, and precision as ***x***. Contains the right-hand side of the system to be solved.

z Real (single- or double-precision) CM array with the same shape, layout, and precision as ***x***. Input argument; used only after the user application performs an operation that **gen_iter_solve** requested through reverse communication. Must contain the results of the user-supplied operation.

y Real (single- or double-precision) CM array with the same shape, layout, and precision as ***x***. Used only when **gen_iter_solve** returns with an *ido* value requesting the user application to operate on an array. Contains the array to be operated on by the user application.

finfo Real front-end array with rank 1, the same precision as ***x***, and length **CMSSL_iter_finfo_size** (a symbolic constant defined in the CMSSL header file). On input, set the values of *finfo* to supply the following information:

finfo(**CMSSL_iter_tol**) On input, this parameter specifies the convergence tolerance, *tol*. If $\omega \leq tol$, **gen_iter_solve** returns with *ido* = **CMSSL_ido_end**, indicating that convergence has occurred.

On output, this parameter contains the value of ω computed for the current iteration loop.

finfo(**CMSSL_iter_anorm**) Estimate of the infinity norm of A, $||A||_\infty$, to be used in computing the convergence criterion when *info*(**CMSSL_iter_omega**) = 2.

You can supply different values for these parameters each time you call **gen_iter_solve**.

ier Scalar integer variable. Error code. Upon return from **gen_iter_solve_setup**, contains one of the following values:

0	No error condition.
-**CMSSL_iter_algorithm**	*info*(**CMSSL_iter_algorithm**) is invalid.
-**CMSSL_iter_kspace**	*info*(**CMSSL_iter_kspace**) is invalid for *info*(**CMSSL_iter_algorithm**) = **CMSSL_gmres** or **CMSSL_qmrlal**.
-**CMSSL_iter_maxiter**	*info*(**CMSSL_iter_maxiter**) is < 1.

The *ier* argument to the **gen_iter_solve** routine is reserved for future use.

DESCRIPTION

Setup and Deallocation. To use the iterative solvers, follow these steps:

1. Call **gen_iter_solve_setup.**

   This routine generates a setup ID and returns it in the *setup* argument. You must supply this *setup* value in all subsequent **gen_iter_solve** and **deallocate_iter_solve** calls associated with this setup call.

2. Call **gen_iter_solve**. Supply the *setup* value assigned by the setup routine, and the same *info* values you supplied to **gen_iter_solve_setup**. In particular, different algorithms require different amounts of internal storage; so if you change the algorithm, call **deallocate_iter_solve** and then call **gen_iter_solve_setup** again. (This involves very little overhead compared to the rest of the algorithm).

Each call to **gen_iter_solve** runs the specified algorithm until intermediate operations must be supplied by the user application; **gen_iter_solve** then requests these operations through the reverse communication interface.

3. When **gen_iter_solve** returns with an *ido* value requesting action by the user application, you must supply the requested operation, place the results in the array *z*, and call **gen_iter_solve** again. Continue until the returned *ido* value indicates convergence, maximum number of iterations exceeded, or breakdown.

4. You can solve other systems by repeating Steps 2 and 3, as long as the rank and shape of x remain the same and the input values you supplied in the setup routine's *info* argument still apply. If the rank and shape of x change or you need to change any of the *info* values, start with Step 1 again.

5. After all **gen_iter_solve** calls associated with the same call to **gen_iter_solve_setup**, call **deallocate_iter_solve** to deallocate the memory required by the setup routine.

More than one setup may be active at a time, as long as they use different setup parameters. That is, you may call the setup routine more than once without calling the deallocation routine.

Iteration. The **gen_iter_solve** routine attempts to solve $Ax = b$ using one of several Krylov space iterative solution algorithms. Depending on the algorithm and the value of the parameter *info*(**CMSSL_iter_residual**), either the actual residual $r = b - Ax$ or an estimate of the residual is used to compute a convergence parameter ω which is returned in *finfo*(**CMSSL_iter_tol**) at each step of the iteration. The iterations continue until $\omega <$ the initial input value of *finfo*(**CMSSL_iter_tol**), an algorithmic breakdown has occurred, or the number of iterations has exceeded the value supplied in *info*(**CMSSL_iter_maxiter**).

Reverse Communication Interface. The iterative solvers require the user application to provide

- routines that multiply a given vector y by A or A^T (alternatively, a vector matrix multiplication, $A^Ty = (y^TA)^T$)

- routines that solve the systems $Mz = y$, $M^Tz = y$, $M_1z = y$, $M_2^Tz = y$, and $M_2z = y$, where $M = M_1M_2$ and the preconditioned matrix $A^{\sim} = M_1^{-1}AM_2^{-1}$

When **gen_iter_solve** requires one of these user-supplied operations, it returns, setting *ido* to indicate which operation is required, and providing the vector y upon which the

user application is to operate. The user-supplied routine must place the results of the requested operation in *z* and call **gen_iter_solve** again.

Preconditioning. If you set *info*(**CMSSL_iter_precond**) = 1, **gen_iter_solve** asks you to solve systems involving the preconditioner, $M = M_1 M_2$, where the preconditioned system is given by $A^{\sim} = M_1^{-1} A M_2^{-1}$, $A^{\sim} M_2 x = M_1^{-1} b$.

NOTES

Include the CMSSL Header File. The iterative solvers use symbolic constants defined in the CMSSL header file. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines.

Acknowledgments. We wish to thank Roland Freund and Noel Nachtigal for providing us with the original Fortran 77 version of their code. We have converted their code to CM Fortran and included it in the algorithms available in **gen_iter_solve**.

EXAMPLES

Sample CM Fortran code that uses the iterative solvers can be found on-line in the subdirectory

```
iter-solvers/cmf
```

of a CMSSL examples directory whose location is site-specific.

7.2 References

For more information about iterative solvers, see the following references:

1. Tong, C. H. *A Comparative Study of Preconditioned Lanczos Methods for Non-symmetric Linear Sytems.* Sandia Technical Report SAND91-8240, 1992.

2. Nachtigal, N.M., S.C.Reddy, and L.N. Trefethen. *How Fast Are Non-Symmetric Matrix Iterations?* Dept. of Mathematics, MIT, Numerical Analysis Report 90-2, 1990.

3. Freund, R.W., G. H. Golub, and N. M. Nachtigal. *Iterative Solution of Linear Systems.* Preprint.

4. Golub, G.H. and C.F. Van Loan. *Matrix Computations.* 2d. ed. Baltimore: Johns Hopkins University Press, 1989.

5. Sonneveld, P. CGS, a Fast Lanczos-Type Solver for Non-Symmetric Linear Systems. *SIAM J. Sci. Stat. Comput.* **10** (1989): 36–52.

6. Fletcher, R. Conjugate Gradient Methods for Indefinite Systems. In *Proc. Dundee Conference* on Numerical Analysis, *1975, Lecture Notes in Mathematics 506*, ed. G.A. Watson. Berlin: Springer-Verlag, 1976. Pp. 73–89.

7. Van der Vorst, H.A. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Non-Symmetric Linear Systems. *SIAM J. Sci. Stat. Comput.* **13** (1992): 631–644.

8. Gutknecht, M.H. *Variants of BICGSTAB for Matrices with Complex Spectrum.* IPS Research Report No. 91-14, Interdisciplinary Project Center for Supercomputing. ETH-Zetrum, CH-8092, Zurich, 1991.

9. Saad, Y. and M.H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Non-Symmetric Linear Systems. *SIAM J. Sci. Stat. Comput.* **7** (1986): 856–869.

10. Freund, R.W. *A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems.* RIACS Technical Report 91.18, NASA Ames Research Center, Moffett Field, California.

11. Freund, R. W., M.H. Gutknecht, and N.M. Nachtigal. An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices. To appear in *SIAM J. Sci. Stat. Comput.* **14** (1993).

12. Freund, R.W. and N.M. Nachtigal. *An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices, Part II.* RIACS Technical Report 90.46, NASA Ames Research Center, Moffett Field, CA, 1990.

13. Freund, R.W. and N.M. Nachtigal. *QMR:* A Quasi-Minimal Residual Method for Non-Hermitian Linear Systems. *Numer. Math.* **60** (1991): 315-39.

14. Freund, R. W. and N.M. Nachtigal. *An Implementation of the QMR Method Based on Coupled Two-Term Recurrences.* RIACS Technical Report 92.15, NASA Ames Research Center, Moffett Field, CA, 1992.

15. Freund, R.W. and T. Szeto. A Quasi-Minimal Residual Squared Algorithm for Non-Hermitian Linear Systems. In *Proceedings of the 1992 Copper Mountain Conference on Iterative Methods*, 1992.

16. Chan, T. F., E. Gallopoulos, V .Simoncini, T. Szeto, and C. H. Tong. *QMRCGSTAB: A Quasi-Minimal Residual Variant of the BI-CGSTAB Algorithm for Non-Symmetric Systems.* CSRD Technical Report 1231, University of Illinois at Urbana-Champaign, Illinois, 1992.

17. Tong, C. H. A Comparative Study of Preconditioned Lanczos Methods for Non-Symmetric Linear Sytems. Sandia Technical Report SAND91-8240, 1992.

Chapter 8

Eigensystem Analysis

This chapter describes the CMSSL eigensystem analysis routines. Section 8.1 provides guidelines for choosing the appropriate routine. Sections 8.2 through 8.9 describe the routines in detail. Section 8.10 lists references.

The CMSSL includes the following eigenanalysis routines:

Reduction to symmetric tridiagonal form and eigensystem analysis of real symmetric tridiagonal matrices:

sym_tred	Reduces one or more Hermitian matrices to real symmetric tridiagonal form. (Section 8.2)
sym_to_tridiag	For each matrix instance, transforms the coordinates of arbitrary vectors from the basis of the original Hermitian matrix to that of the tridiagonal matrix. (Section 8.2)
tridiag_to_sym	For each matrix instance, transforms the coordinates of arbitrary vectors from the basis of the tridiagonal matrix to that of the original Hermitian matrix. (Section 8.2)
deallocate_sym_tred	Deallocates the processing element storage space required by the above three routines. (Section 8.2)
sym_tridiag_eigenvalues	Computes all the eigenvalues of one or more real symmetric tridiagonal matrices of the same order. (Section 8.3)

sym_tridiag_eigenvectors Determines the eigenvectors corresponding to a set of eigenvalues of one or more real symmetric tridiagonal matrices of the same order. (Section 8.4)

Eigensystem analysis of dense Hermitian matrices:

sym_tred_eigensystem Computes the eigenvalues and, if desired, the eigenvectors of one or more Hermitian matrices. Combines the functionality of the **sym_tred**, **sym_tridiag_eigenvalues**, **sym_tridiag_eigenvectors**, **tridiag_to_sym**, and **deallocate_sym_tred** routines. (Section 8.5)

Eigensystem analysis of dense real symmetric matrices:

sym_tred_gen_eigensystem Given a CM array containing one or more real symmetric matrices A, and a CM array containing corresponding positive definite matrices B, this routine solves $AQ = BQ\lambda$, computing the eigenvalues λ and, if desired, the eigenvectors for each instance. (Section 8.6)

sym_jacobi_eigensystem Uses Jacobi rotations to compute the eigenvalues and, if desired, the eigenvectors of one or more dense real symmetric matrices. (Section 8.7)

sym_lanczos Finds selected eigenpairs of a linear operator, L, that is real and symmetric with respect to a positive semi-definite real matrix B ($BL = L^TB$). Uses the implicit restarted k-step Lanczos update algorithm. Has an associated setup routine (**sym_lanczos_setup**) and deallocation routine (**deallocate_sym_lanczos_setup**). (Section 8.8)

Eigensystem analysis of dense real matrices:

gen_arnoldi Finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. B is symmetric and can be positive semi-definite; it is the identity for the

standard eigenproblem. The algorithm used is a *k*-step Arnoldi algorithm with implicit restart. This routine has an associated setup routine (**gen_arnoldi_setup**) and deallocation routine (**deallocate_gen_arnoldi_setup**). (Section 8.9)

The **sym_lanczos** and **gen_arnoldi** routines also apply perform eigensystem analysis of sparse systems.

8.1 Introduction

The selection of a CMSSL eigenanalysis routine depends on

- whether the problem is Hermitian
- how many eigenvalues and/or eigenvectors are desired
- whether the system is dense or sparse

If the system is *not Hermitian*, the only function provided is **gen_arnoldi**, which allows you to compute selected eigenvalue-eigenvector pairs. In the current implementation, the projected matrix (that is, the projection of the original problem onto the basis defined by the Arnoldi vectors) is stored and processed on the partition manager (this applies to **sym_lanczos** as well). Because of the performance difference between the CM and the partition manager, the routine is aimed at computing an invariant subspace of dimension much smaller than the original problem. Under this condition, the matrix vector operation, which is performed on the CM, dominates the computation. Communication with the **gen_arnoldi** routine occurs through *reverse communication*; the user must provide a matrix vector product on request through this interface. One may wish simply to write a subroutine to provide this matrix vector product. However, the reverse communication mechanism may eliminate the need to encapsulate this matrix vector product within a separate subroutine. In either case, it is important to exploit whatever structure the problem may have when computing this matrix vector product, since it will be the most time-consuming part of the computation for large systems. The choice of the subspace parameters k and nv may influence the convergence significantly. The parameter nv should be at least equal to $2k$, but selecting a larger value may often accelerate convergence if there is enough memory to accomodate the nv Arnoldi vectors. When the desired eigenvalues are clustered, it is sometimes faster to compute more eigenpairs than originally

sought, but with a tolerance parameter *tol1* larger than the one desired (*tol2*, say). Assume that p eigenpairs are sought, choose $k > p$ and *tol1* >> *tol2*. By the time the k eigenpairs are converged with the prescribed tolerance *tol1*, the first p eigenpairs may have converged with an accuracy of *tol2*. This may happen in fewer iterations than would be required for convergence had k been set to p and the tolerance parameter set to *tol2*. All of these choices are problem-dependent; it will take some experience to find the best configuration for a given problem class. In general, it does not take much more time to compute the eigenvalues and eigenvectors (*iparam*(2) > 0) than to compute the eigenvalues only (*iparam*(2) ≤ 0) (this also applies to **sym_lanczos**). This attractive feature is a property of the k-step Arnoldi algorithm with implicit restart (see reference 12 in Section 8.10).

If the system is *real symmetric and sparse* and only selected eigenpairs are desired, it is advisable to use **sym_lanczos**, the Hermitian version of **gen_arnoldi**. All the above comments apply here as well. The case where interior eigenvalues are desired deserves special consideration. In this case, the Lanczos algorithm will converge very slowly, if at all. The **sym_lanczos** routine (and the **gen_arnoldi** routine) should then be used in shift-and-invert mode through the reverse communication mechanism, as described in Sections 8.8 and 8.9. Instead of performing a matrix vector operation at each step, one must solve the linear system of equations $(A-\sigma I)x = b$. The operator $(A-\sigma I)^{-1}$ has the same eigenvectors as A, but the eigenvalues in the vicinity of σ are well separated, while those at both extremes of the spectrum are clustered around zero. As a result, convergence for eigenvalues located around σ is dramatically improved. Of course, the problem is now to solve an indefinite system of equations. For sparse systems, it is tempting to use an iterative method. However, this only makes sense if the system of equations can be preconditioned efficiently. Otherwise, the number of matrix vector operations used repeatedly in the iterative solutions of the linear systems will be prohibitive. In fact, there will be about as many matrix vector operations as would be needed for the standard Lanczos algorithm to converge. In general, you may enhance convergence by applying the algorithm to a function of the matrix where the desired part of the transformed spectrum is separated from the unwanted part.

If the system is *dense Hermitian* and all eigenpairs are required, then one should use **sym_tred_eigensystem**. For most cases, the Jacobi method implemented in **sym_jacobi_eigensystem** does not seem to provide a competitive alternative at this point for comparable accuracy. The **sym_tred_eigensystem** routine encapsulates four routines: **sym_tred**, which reduces the matrix to tridiagonal form; **sym_tridiag_eigenvalues**, which computes all the eigenvalues of the tridiagonal matrix; **sym_tridiag_eigenvectors**, which computes all or selected eigenvectors using inverse iteration; and **tridiag_to_sym**, which transforms the tridiagonal

eigenvectors to the eigenvectors of the original matrix. There is a great deal of flexibility to be gained by calling these routines separately. Indeed, it is often true that out of a very large vector space, only a limited number of eigendirections are useful. By calling the four eigenroutines individually, one can select the eigenvectors of interest after inspecting the eigenvalue spectrum. By doing so, one can also handle much larger problems than when all the eigenvectors are computed.

It is not currently possible to compute selected eigenvalues only. However, the performance of **sym_tridiag_eigenvalues** (which implements parallel bisection and takes advantage of IEEE arithmetic) is sufficient that the time to compute all the eigenvalues is usually very small compared to the reduction and eigenvector extraction. Note also that the routines that solve the tridiagonal eigenproblem do not take advantage of deflation. Therefore, for large problems, you may wish to check for potential deflations beforehand (see Sections 8.3.3 and 8.4.4).

The **sym_tridiag_eigenvalues** and **sym_tridiag_eigenvectors** routines include two parameters to set:

- The absolute error tolerance for the computed eigenvalues, which can be set as small as desired (machine precision times the 1-norm of the matrix is the default). Setting the tolerance to a higher value is not recommended, as it may cause inverse iteration to fail later.

- The grouping criterion for eigenvalues. The eigenvectors associated with grouped eigenvalues are orthogonalized. This parameter is not usually an input parameter in standard scientific libraries. We provide it because its usual value, $10^{-3}\| T \|_\infty$, which is the default value in **sym_tridiag_eigenvectors**, is much too large in general, and entails unnecessary reorthogonalization between eigenvectors, an unbalanced and expensive computation on distributed memory architecture. Although some new algorithm may solve this problem elegantly in the near future, we strongly recommend setting the *group* argument to a much smaller value than the default ($10^{-5}\| T \|_\infty$, or even $10^{-6}\| T \|_\infty$). Of course, one should assess the orthogonality of the eigenvectors obtained. One should also realize that orthogonality to machine precision is unnecessary for most practical applications. Orthogonality to the square root of machine precision usually suffices.

All the above considerations for **sym_tred_eigensystem** also apply to **sym_tred_gen_eigensystem**, which computes all the eigenpairs of dense real symmetric general eigensystems. If you want to compute only selected eigenvectors of such systems, you can call the components of **sym_tred_gen_eigensystem** separately,

as described in Section 8.6. Note that you can also use **sym_lanczos** and **gen_arnoldi** to solve generalized eigenvalue problems by setting the input argument *type* to 'G.'

Finally, nothing prevents the use of **sym_lanczos** or **gen_arnoldi** in shift-and-invert mode to extract a few eigenpairs in the middle of the spectrum of a dense matrix. In that case, a dense solver routine can be used at each iteration. It is unlikely that this approach will compete with the Householder reduction to tridiagonal form when the matrix fits into memory. However, even though the CMSSL does not currently provide eigensolvers with external storage, it is possible, using this approach, to compute selected eigenpairs of a matrix that is too large to fit into core memory. The first step of an *out-of-core dense* algorithm is to factor the matrix $A-\sigma I$ (where σ is the value in the neighborood of which a few eigenvalues are sought) using the CMSSL external *LU* factorization routine. Then call the external *LU* solver routine with **sym_lanczos** (for symmetric problems) or **gen_arnoldi** (for non-symmetric problems) in shift-and-invert mode.

8.2 Reduction to Tridiagonal Form and Corresponding Basis Transformation

The CMSSL provides a routine that reduces Hermitian matrices to real symmetric tridiagonal form using Householder transformations. After the reduction occurs, two other routines can be used to transform the coordinates of sets of vectors from the bases of the original Hermitian matrices to those of the tridiagonal matrices, or vice versa. The routines are as follows:

sym_tred	Reduces one or more Hermitian matrices to real symmetric tridiagonal form.
sym_to_tridiag	For each matrix instance, transforms the coordinates of arbitrary vectors from the basis of the original Hermitian matrix to that of the tridiagonal matrix.
tridiag_to_sym	For each matrix instance, transforms the coordinates of arbitrary vectors from the basis of the tridiagonal matrix to that of the original Hermitian matrix.
deallocate_sym_tred	Deallocates the processing element storage space required by the above routines.

Detailed descriptions of these routines are provided in the man page at the end of this section.

8.2.1 Blocking and Load Balancing

The reduction to tridiagonal form and basis transformation routines use blocking and load balancing to enhance performance. These strategies are described in the section on computation of block cyclic permutations in Chapter 14.

8.2.2 Numerical Stability

The routines described in this section are stable.

Reduction to Tridiagonal Form and Corresponding Basis Transformation

Given one or more Hermitian matrices, the **sym_tred** routine uses Householder transformations to reduce each matrix to real symmetric tridiagonal form. Given the transformations performed by **sym_tred**, the **sym_to_tridiag** routine transforms the coordinates of arbitrary vectors from the basis of the original Hermitian matrix to that of the tridiagonal matrix; the **tridiag_to_sym** routine transforms the coordinates of arbitrary vectors from the basis of the tridiagonal matrix to that of the original Hermitian matrix. The **deallocate_sym_tred** routine deallocates the storage space required by **sym_tred**, **sym_to_tridiag**, and **tridiag_to_sym**.

SYNTAX

setup = **sym_tred** (*d, e, A, n, row_axis, col_axis, nblock, ier*)

sym_to_tridiag (*B, A, setup, nrhs, ier*)

tridiag_to_sym (*B, A, setup, nrhs, ier*)

deallocate_sym_tred (*setup*)

ARGUMENTS

In this description, *A* and *B* refer to the active matrices within the CM arrays *A* and *B* with which the routines work. *A* and *B* may be contained (as the upper left-hand submatrices) in larger matrices within *A* and *B*, respectively. Details are provided below.

setup — Scalar integer variable. Setup ID. When you call **sym_to_tridiag**, **tridiag_to_sym**, or **deallocate_sym_tred**, you must supply the value returned by **sym_tred**.

d — Real CM array of the same rank as *A*. Axis *row_axis* must have extent 1; axis *col_axis* must have extent $\geq n$. The remaining axes are instance axes matching those of *A* in order of declaration and extents. Thus, each vector within *d* corresponds to a matrix *A* within *A*. Upon completion of **sym_tred**, elements 1 through *n* of each vector in *d* contain the main diagonal elements of the real symmetric tridiagonal matrix to which the corresponding matrix *A* was reduced.

e Real CM array of the same rank as *A*. Axis *row_axis* must have extent 1; axis *col_axis* must have extent $\geq n$. The remaining axes are instance axes matching those of *A* in order of declaration and extents. Thus, each vector within *e* corresponds to a matrix *A* within *A*. Upon completion of **sym_tred**, elements 2 through *n* of each vector in *e* contain the off-diagonal elements of the real symmetric tridiagonal matrix to which the corresponding matrix *A* was reduced. (The first element in each vector in *e* is undefined.)

B CM array of the same data type as *A*. When you call **sym_to_tridiag** or **tridiag_to_sym**, *B* must contain one or more instances of a rank-2 array, *B*; each *B*, in turn, must consist of the vector(s) whose coordinates are to be transformed by **sym_to_tridiag** or **tridiag_to_sym**. Upon completion of **sym_to_tridiag** or **tridiag_to_sym**, each *B* within *B* is overwritten with same vectors expressed in the coordinates of the new basis.

The instance axes of *B* must match those of *A* in order of declaration and extents. Each *B* within *B* has dimensions *n* x *nrhs*, and may consist of the upper left-hand *n* x *nrhs* elements of a larger matrix. The rows and columns of *B* must be counted by axes *row_axis* and *col_axis*, respectively.

A Real or complex CM array containing one or more Hermitian matrices, *A*. Each *A* within *A* is assumed to be dense and square with dimensions *n* x *n*. The axes identified by *row_axis* and *col_axis* may have extents greater than *n*; that is, each instance of *A* may be contained in the upper left-hand *n* x *n* elements of a larger matrix within *A*.

Upon completion of **sym_tred**, each *A* within *A* is overwritten with information about the Householder transformations used to reduce *A* to a tridiagonal matrix. When you call **sym_to_tridiag** or **tridiag_to_sym**, you must supply the values contained in *A* upon completion of **sym_tred**.

n Scalar integer variable. The number of rows and columns in each Hermitian matrix *A* within *A*.

row_axis Scalar integer variable. The axis of *A* that counts the rows of each Hermitian matrix *A*.

col_axis Scalar integer variable. The axis of *A* that counts the columns of each Hermitian matrix *A*.

nblock Scalar integer variable. Blocking factor. Use these guidelines when choosing an *nblock* value:

- For typical applications, *nblock* = 2 is a good choice. For very large matrices, nblock = 4 or even 8 may yield faster reduction.
- *nblock* should always be $\leq n$; *nblock* values > *n* use excess time and especially memory.
- The amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- For optimal performance, ensure that the subgrid length is a multiple of *nblock* in both dimensions. If that is not possible, choose an *nblock* value that is smaller than the subgrid lengths in both dimensions.

nrhs Scalar integer variable. The number of vectors in each *B* within *B*.

ier Scalar integer variable. Return code; set to 0 upon successful return. The following codes indicate errors:

-1	Length of axis *row_axis* of *A* is < *n*; must be $\geq n$.
-2	Length of axis *col_axis* of *A* is < *n*; must be $\geq n$.
-8	Rank of *A* is < 2; must be ≥ 2.
-32	Data type of *A*, *B*, *d*, or *e* is not real or complex.
-64	*row_axis* or *col_axis* is invalid. $1 \leq$ *row_axis*, *col_axis* $\leq$ rank (*A*) must be true, and *row_axis* and *col_axis* must not be equal.
-128	*nblock* is invalid; must be ≥ 1.

DESCRIPTION

Given a real or complex CM array *A* containing one or more Hermitian matrices *A*, the routines described in this man page perform the following operations:

sym_tred $T = Q^H A Q$ (*T* is stored in *d* and *e*)

sym_to_tridiag	$B = QB$
tridiag_to_sym	$B = Q^{H}B$

The **sym_tred** routine uses Householder transformations to reduce each A to real symmetric tridiagonal form. Given the transformations performed by **sym_tred**, and a CM array B containing one or more instances of a rank-2 array B of vectors, the **sym_to_tridiag** routine transforms the coordinates of each set of vectors from the basis of the original Hermitian matrix to that of the tridiagonal matrix; the **tridiag_to_sym** routine transforms the coordinates of each set of vectors from the basis of the tridiagonal matrix to that of the original Hermitian matrix.

The **deallocate_sym_tred** routine deallocates the storage space required by **sym_tred**, **sym_to_tridiag**, and **tridiag_to_sym**.

Setup and Deallocation. The **sym_tred** routine allocates processing element storage space and returns a setup ID. You must supply this setup ID in subsequent **sym_to_tridiag** and **tridiag_to_sym** calls as long as you are working with the same reduction; you must also supply it to **deallocate_sym_tred**. You can follow one call to **sym_tred** with multiple calls to the **sym_to_tridiag** and **tridiag_to_sym** routines.

The **deallocate_sym_tred** routine deallocates the memory needed for a particular reduction, and invalidates the associated setup ID. Attempts to use a deallocated setup ID result in errors.

You can work with more than one set of reductions at a time by calling **sym_tred** more than once without calling **deallocate_sym_tred**. Be sure to supply the correct setup ID in each subsequent **sym_to_tridiag** or **tridiag_to_sym** call. When you have finished working with a reduction, be sure to use **deallocate_sym_tred** to deallocate the associated memory. Repeated calls to **sym_tred** without deallocation can cause you to run out of memory.

Reduction to Tridiagonal Form. The **sym_tred** routine uses Householder transformations to reduce each A within A to real symmetric tridiagonal form. Upon completion of **sym_tred**, each A within A is overwritten with information about the Householder transformations used to reduce A to a real symmetric tridiagonal matrix. Each resulting tridiagonal matrix is represented by the corresponding instances of the vectors d and e.

Basis Transformation. The **sym_to_tridiag** routine transforms the coordinates of the vectors in each B within B from the basis of the corresponding original Hermitian matrix to that of the tridiagonal matrix. The **tridiag_to_sym** routine transforms the coordinates of the vectors in each B from the basis of the corresponding tridiagonal matrix to that of the original Hermitian matrix. Upon completion of **sym_to_tridiag** or

tridiag_to_sym, each B within B is overwritten with same vectors expressed in the coordinates of the new basis.

NOTES

Distinct Variables. The input CM arrays A and B must be distinct variables. The arrays d and e must also be distinct.

Include the CMSSL Header File. The **sym_tred** routine is a function. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL functions and symbolic constants.

Preservation of Argument Values. The internal variable *setup* is required for communicating information between the reduction to tridiagonal form routine and the basis transformation routines. The application must not modify the contents of this variable.

Numerical Stability. These routines are stable.

Numerical Complexity. Reduction to tridiagonal form uses $(4/3)n^3$ floating-point operations. However, because **sym_tred** does not exploit symmetry, the CM implementation actually uses $2n^3$ floating-point operations. The **sym_to_tridiag** and **tridiag_to_sym** routines use $2n^2 * nrhs$ floating-point operations.

EXAMPLES

Sample CM Fortran code that uses the routines described above can be found on-line in the subdirectory

```
tred/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.3 Eigenvalues of Real Symmetric Tridiagonal Matrices

The **sym_tridiag_eigenvalues** routine computes all the eigenvalues of one or more real symmetric tridiagonal matrices of the same order. A detailed description of this routine is provided in the man page at the end of this section.

8.3.1 Parallel Bisection Algorithm

You can compute the spectra of one or more tridiagonal matrices with **sym_ tridiag_eigenvalues**. Subsequently, you can compute selected, or possibly all, eigenvectors using **sym_tridiag_eigenvectors**.

Parallel bisection is the algorithm currently implemented for the eigenvalue computation. The serial bisection algorithm (see reference 3 in Section 8.10) extracts one eigenvalue at a time by recursively dividing in two equal parts an initial interval known to contain the desired eigenvalue. In a data parallel environment, a matrix of order N is partitioned over N/n processing elements, and each processing element can compute up to n eigenvalues, provided it has access to all the matrix elements. Processing element i computes eigenvalues $ni + 1, \ldots, n(i+1)$, thereby slicing its own portion of the spectrum. The union of all Gershgorin disks provides an initial search interval which is known to contain all eigenvalues.

At each bisection step, one needs to determine the number of eigenvalues smaller than the midpoint x of the current interval. This number is obtained by evaluating the non-linear Sturm sequence. Independent sequences corresponding to independent eigenvalue computations can be evaluated concurrently on different processing elements provided each processing element has a copy of the relevant matrix. A preprocessing step therefore distributes a copy of the matrix to all processing elements slicing its spectrum. This is accomplished in $N/n-1$ nearest-neighbor communication steps on the ring of N/n processing elements that share the matrix elements. In the case of multiple instances, matrices laid out on disjoint sets of processing elements are diagonalized concurrently, while matrices laid out on identical sets of processing elements are diagonalized in sequence.

A somewhat more detailed description of the parallel bisection implementation is given in the Fall 1991 issue (Volume 1, Number 3) of the CMSSL newsletter.

8.3.2 Accuracy

The input parameter *tolerance* controls the absolute error in the eigenvalues computation. Although the bisection algorithm is accurate enough to extract eigenvalues to relative accuracy, only absolute accuracy is supported in this release. The parameter *tolerance* should be set to the error tolerated in the computation of the absolutely smallest eigenvalue. If *tolerance* is a non-positive number, it is internally set to *tolerance* = ε ||T||, where ε is the machine precision and ||T|| is the 1-norm of the matrix. In the case of multiple instances, the internal tolerance is the smallest tolerance over all matrices. This criterion will in general provide high relative accuracy for the algebraically largest eigenvalues but not for the tiny ones. In case a tiny eigenvalue is of the same order of magnitude as the default tolerance value, consider restarting the eigenvalue computation with a smaller (but positive) *tolerance*. This situation may occur because the matrices are assumed to be unreduced (see Section 8.3.3 below). If a matrix is not unreduced, tiny eigenvalues that correspond to a small submatrix are computed with a default *tolerance* that corresponds to the full matrix. Because the norm of the full matrix could be much larger than the norm of the submatrix, the eigenvalues of the submatrix are not computed as accurately as they would have been had the original matrix been deflated beforehand.

8.3.3 Restriction

Prior deflation of the matrix plays an important role in the standard bisection algorithm. The current version of **sym_tridiag_eigenvalues** does not perform deflation. Input matrices are assumed unreduced. In case the square of a subdiagonal element is zero, it is replaced with the smallest number representable on the machine to avoid the evaluation of 0/0 in the non-linear Sturm recurrence. This situation could occur because there is no overflow check in the Sturm recurrence computation (IEEE arithmetic guarantees that the sign of an overflowed quantity is preserved). This alteration of the matrix entries, when it occurs, contributes an uncertainty of $(UN)^{1/2}$, where UN is the underflow threshold, a very tiny quantity.

Nothing prevents you from deflating the matrix beforehand and calling in sequence **sym_tridiag_eigenvalues** with array sections that contain unreduced submatrices. Resulting submatrices could be diagonalized in parallel using multiple instances with an array of higher dimensionality, but it is quite unlikely that the submatrices will be of the same order. For the same reason, this preprocessing step is only likely to be useful in the single-instance case.

Eigenvalues of Real Symmetric Tridiagonal Matrices

The **sym_tridiag_eigenvalues** routine computes all the eigenvalues of one or more real symmetric tridiagonal matrices of the same order. The diagonal and subdiagonal matrix elements are stored in two CM vectors or arrays.

SYNTAX

sym_tridiag_eigenvalues (***d, e, axis, tolerance, ier***)

ARGUMENTS

d	Real CM array containing the diagonal elements of one or more symmetric tridiagonal matrices. On successful completion of **sym_tridiag_eigenvalues**, the diagonal elements of each matrix are overwritten with the sorted eigenvalues of the matrix; the algebraically smallest eigenvalue is placed in the first element.
e	Real CM array of the same shape and layout as ***d***, containing the off-diagonal elements of the symmetric tridiagonal matrices. The first element in each instance can have any value. On return, each element of ***e*** is squared and the first element is set to zero.
axis	Scalar integer variable. The axis of ***d*** and ***e*** along which the elements of each matrix lie (the non-instance axis).
tolerance	Scalar real variable. Absolute error tolerance for the computed eigenvalues. When *tolerance* is non-positive, it is reset internally as described in Section 8.3.2.
ier	Scalar integer variable. Set to 0 on successful completion.

DESCRIPTION

The **sym_tridiag_eigenvalues** routine computes $Tx = \lambda x$, where T is stored in ***d*** and ***e*** and the eigenvalues λ are returned in ***d***.

EXAMPLES

Sample CM Fortran code that uses the **sym_tridiag_eigenvalues** routine can be found on-line in the directory

```
eigen/realsymtrid/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.4 Eigenvectors of Real Symmetric Tridiagonal Matrices

The **sym_tridiag_eigenvectors** routine computes the eigenvectors corresponding to a given set of eigenvalues of one or more real symmetric tridiagonal matrices of the same order.

8.4.1 Inverse Iteration Algorithm

Given a matrix T and λ an approximate eigenvalue of T, inverse iteration is the inverse power method applied to $(T - \lambda I)$. The essential computation of inverse iteration is the solution of linear systems of equations of the form

$$(T - \lambda I)x = b \quad (1)$$

The matrix $(T - \lambda I)$ is close to singular when λ is an approximate eigenvalue. Unlike the CM-200 version, the CM-5 implementation of inverse iteration uses numerical pivoting in the solution of the very ill-conditioned system of equations (1).

The starting vectors for inverse iteration are independent normalized random vectors, and at least two inverse iterations are performed. Eigenvectors corresponding to clustered eigenvalues are orthogonalized using the modified Gram-Schmidt algorithm. The segmented SCAN operation allows for orthogonalization within clusters, but this is clearly an unbalanced computation.

8.4.2 Accuracy

The eigenvalues supplied in the *f* input array must be accurate enough for the associated eigenvectors to be determined accurately by inverse iteration. This will generally be the case when the eigenvalues are computed using **sym_tridiag_eigenvalues** with the tolerance set internally, assuming no tiny eigenvalue is of the same order of magnitude as this default tolerance (see Section 8.3.2).

Eigenvectors corresponding to close eigenvalues are ill-conditioned. Extracting independent and orthogonal eigenvectors corresponding to pathologically close eigenvalues is a hard problem. In particular, eigenvectors associated with grouped eigenvalues must be orthogonalized. This is achieved using the modified Gram-Schmidt algorithm. Eigenvalues λ_i and λ_j are grouped if $|\lambda_i - \lambda_j| \leq$

group $\|T\|_\infty$, where $\|T\|_\infty$ is the infinity norm of the matrix and *group* is the grouping criterion. The standard value for the grouping criterion is 10^{-4} (see reference 4 in Section 8.10). It is, however, a rather subjective matter. Numerical experiments show that rather large fluctuations in the grouping criterion do not drastically influence orthogonality between eigenvectors for practical purposes (see reference 6). They do have a drastic influence on performance, however. Even though the default value is *group* = 10^{-4}, it is strongly recommended that you experiment with much lower grouping criteria. In most practical cases, a value of *group* = 10^{-5}, for example, has proved satisfactory.

8.4.3 Applicability

Unlike **sym_tridiag_eigenvalues**, which computes all eigenvalues of one or more matrices, **sym_tridiag_eigenvectors** can compute selected eigenvectors of one or more matrices. As many eigenvectors are computed as there are eigenvalues in the ***f*** array. Therefore, the ***f*** and ***Q*** arrays must have the same shape, except for the extra dimension of ***Q*** that will hold the eigenvectors. The extra axis of ***Q*** (identified by the *eigenvector_axis* argument) must have a length equal to the order of the matrices represented by ***d*** and ***e***. Selected eigenvalues for which the eigenvectors are sought can be supplied in an array subsection. However, ***f*** must have the same rank as ***d*** (or ***e***). (For detailed descriptions of all arguments, see the man page at the end of this section.)

To illustrate the above, let $A(100)$ and $B(100)$ be one-dimensional arrays containing the diagonal and subdiagonal elements of a tridiagonal matrix of order 100. Let $D(100)$ be the array containing all its eigenvalues as returned by **sym_ tridiag_eigenvalues**. Assume only the eigenvectors corresponding to the 10 largest eigenvalues are sought. One can allocate an array $Z(100, 10)$ to store the 10 eigenvectors. In this case, a proper call to **sym_tridiag_eigenvectors** would be

```
sym_tridiag_eigenvectors(A, B, 1, D(91:100), Z, 1, group, ier)
```

8.4.4 Restriction

Prior deflation of the matrix plays an important role in the standard inverse iteration algorithm (see reference 4 in Section 8.10). The original matrix is the direct sum of submatrices when negligible subdiagonal elements occur (see Section 8.3.3). The input eigenvalues have an index pointing to the submatrix to which they belong, and the subproblems are processed in sequence. Eigenvectors

belonging to different submatrices are exactly orthogonal since they span orthogonal vector spaces. The current version of **sym_tridiag_eigenvectors** does not perform deflation. Input matrices are assumed unreduced. As a result, eigenvectors associated with close eigenvalues that belong to different submatrices will be orthogonalized numerically, a less accurate solution to the problem of finding orthogonal vectors which belong to naturally orthogonal spaces.

As with **sym_tridiag_eigenvalues**, nothing prevents you from deflating the matrix beforehand and solving the subproblems in sequence using array subsections. In such a case, the submatrices will most likely be determined before calling **sym_tridiag_eigenvalues**, and the eigenvalues belonging to different submatrices will have been extracted independently (in particular, eigenvalues will be sorted within submatrices and not across the original matrix). Calling **sym_tridiag_eigenvectors** in sequence to solve the independent subproblems with appropriately shaped subsections of the eigenvector array will then yield exactly orthogonal eigenvectors associated with orthogonal subspaces. As with **sym_tridiag_eigenvalues**, this will not in general lead to subproblems of the same size that could be solved concurrently in a multiple-instances fashion.

8.4.5 Performance

Since the tridiagonal system solver routines **gen_tridiag_factor** and **gen_tridiag_solve** are called during the execution of **sym_tridiag_eigenvectors**, prescriptions given for those functions in order to obtain good performance apply here as well. In particular, lay out the eigenvectors on a serial dimension for best performance.

Eigenvectors of Real Symmetric Tridiagonal Matrices

The **sym_tridiag_eigenvectors** routine determines the eigenvectors corresponding to a set of eigenvalues of one or more real symmetric tridiagonal matrices of the same order. The diagonal and subdiagonal matrix elements and the eigenvalues are stored in CM vectors or arrays. The eigenvectors are stored in a multidimensional CM array.

SYNTAX

sym_tridiag_eigenvectors (*d, e, axis, f, Q, eigenvector_axis, group, ier*)

ARGUMENTS

d	Real CM array containing the diagonal elements of one or more symmetric tridiagonal matrices. The axis along which the elements of each matrix lie (the non-instance axis) is identified by the *axis* argument.
e	Real CM array of the same shape and layout as *d*. Contains the off-diagonal elements of the symmetric tridiagonal matrices. The axis along which the elements of each matrix lie (the non-instance axis) is identified by the *axis* argument. The first element in each instance is arbitrary and is set to zero on return.
axis	Scalar integer variable. The non-instance axis of *d* and *e* (the axis along which the matrix elements lie).
f	CM array containing the eigenvalues for which the eigenvectors are sought. Must have the same rank as *d*. The instance axes must match those of *d* in order of declaration and extents. Within each instance, the eigenvalues belonging to the same spectrum must be sorted in non-decreasing order (with the algebraically smallest eigenvalue stored in the first array element). The extent of the axis identified by *axis* can be smaller in *f* than it is in *d*, as described in Section 8.4.3.

Q CM array that contains the eigenvectors on return. Must have rank one greater than that of *f*. You must specify the index of the extra dimension in the *eigenvector_axis* argument. The array section obtained by collapsing this extra dimension must be of the same shape as *f* (see Section 8.4.3). Thus, for each eigenvalue in *f*, there is an associated vector lying along axis *eigenvector_axis* of *Q*. Upon completion, this vector contains the eigenvector corresponding to the eigenvalue.

eigenvector_axis Scalar integer variable. The axis of *Q* along which the returned eigenvectors lie. The extent of axis *eigenvector_axis* is the order of the tridiagonal matrices.

group Scalar real variable. Eigenvalues that differ by less than $group\|T\|_\infty$ (where $\|T\|_\infty$ is the infinity norm of the matrix) are grouped together and their corresponding eigenvectors are orthogonalized. When *group* is non-positive, it is reset internally as described in Section 8.4.2.

ier Scalar integer variable. Set to 0 on successful completion. On error, contains one of the following codes:

1	The rank of *f* is not the same as the rank of *d*.
2	The rank of *Q* is not the equal to (rank of *d*) + 1.
3	The shape of the array section corresponding to a fixed index of *Q* along dimension *eigenvector_axis* does not have the same shape as *f*.
1000+*n*	*n* eigenvectors are not determined after 5 inverse iterations. The non-converged eigenvectors are set to 0 on return.

DESCRIPTION

Given one or more symmetric tridiagonal matrices represented by the CM arrays *d* and *e*, and a CM array *f* containing a set of eigenvalues for each matrix, the **sym_tridiag_eigenvectors** routine computes the eigenvector corresponding to each eigenvalue — that is, it computes

$$TQ = Q * \mathrm{DIAG}(f)$$

where T is stored in d and e. Upon return, the eigenvectors are contained in the CM array Q.

EXAMPLES

Sample CM Fortran code that uses the **sym_tridiag_eigenvectors** routine can be found on-line in the subdirectory

```
eigen/realsymtrid/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.5 Eigensystem Analysis of Dense Hermitian Matrices

The **sym_tred_eigensystem** routine combines the functionality of the following routines:

- **sym_tred**
- **sym_tridiag_eigenvalues**
- **sym_tridiag_eigenvectors**
- **tridiag_to_sym**
- **deallocate_sym_tred**

Given a CM array containing one or more Hermitian matrices, **sym_tred_ eigensystem** computes the eigenvalues and, if desired, the eigenvectors of each matrix.

The **sym_tred_eigensystem** routine offers a convenient packaging of the five routines listed above. On the other hand, the **sym_tridiag_eigenvectors** routine allows you the flexibility of computing only selected eigenvectors of each matrix, whereas **sym_tred_eigensystem** computes either all or none of the eigenvectors.

For a detailed description of **sym_tred_eigensystem**, see the man page at the end of this section.

8.5.1 Accuracy

The *tolerance* parameter controls the accuracy of the eigenvalues after reduction to tridiagonal form (see Section 8.3.2). Note that requesting extra accuracy for the eigenvalues of the intermediate tridiagonal matrix improves the quality of the eigenvalues of the original matrix only to the extent that roundoff errors incurred in the reduction to tridiagonal form do not dominate. The *group* parameter sets the grouping criterion for the eigenvalues after reduction to tridiagonal form (see Section 8.4.2).

Eigensystem Analysis of Dense Hermitian Matrices

Given a CM array containing one or more complex Hermitian matrices, **sym_tred_ eigensystem** computes the eigenvalues and, if desired, the eigenvectors of each matrix.

SYNTAX

sym_tred_eigensystem (*d, Q, A, n, row_axis, col_axis, nblock, evects_flag, tolerance, group, ier*)

ARGUMENTS

d

Real CM array with the same rank as *A*. Axis *row_axis* must have extent 1; axis *col_axis* must have extent $\geq n$. The remaining axes are instance axes matching those of *A* in order of declaration and extents. Thus, each vector within *d* corresponds to a matrix *A* within *A*. Upon completion of **sym_tred_eigensystem**, elements 1 through *n* of each vector in *d* contain the eigenvalues of the corresponding matrix *A*, sorted in non-decreasing order (with the algebraically smallest eigenvalue stored in the first element).

Q

CM array with the same rank and data type as *A*. The axes identified by *row_axis* and *col_axis* must have extents $\geq n$; the remaining axes are instance axes that must match those of *A* in order of declaration and extents. Thus, for each matrix *A* within *A* there is a corresponding two-dimensional array of dimensions at least *n* x *n* within *Q*. If *evects_flag* is set to 1, then upon return, the eigenvectors of each *A* within *A* are placed in the upper-left-hand *n* x *n* elements of the corresponding two-dimensional array within *Q*. The eigenvectors lie along the axis identified by *col_axis*. The eigenvectors are sorted so that they are returned in the same order as the eigenvalues to which they correspond.

A	Real or complex CM array containing one or more Hermitian matrices, *A*. Each *A* within ***A*** is assumed to be dense and square with dimensions $n \times n$. The axes identified by *row_axis* and *col_axis* must have extent *n*. Upon return, each *A* within ***A*** is overwritten with information about the Householder transformations used to reduce *A* to a real symmetric tridiagonal matrix.
n	Scalar integer variable. The number of rows and columns in each Hermitian matrix *A* within ***A***.
row_axis	Scalar integer variable. The axis of ***A*** that counts the rows of each Hermitian matrix *A*.
col_axis	Scalar integer variable. The axis of ***A*** that counts the columns of each Hermitian matrix *A*.
nblock	Scalar integer variable. Blocking factor. For typical applications, *nblock* = 2 is a good choice. For very large matrices, *nblock* = 4 or even 8 may yield faster reduction. The amount of auxiliary storage used is proportional to *nblock*, so if memory is tight a smaller *nblock* may be a better choice.
evects_flag	Scalar integer variable. If you set *evects_flag* to 0, only the eigenvalues are computed. If you set *evects_flag* to 1, both eigenvalues and eigenvectors are computed.
tolerance	Scalar real variable. Controls the absolute accuracy of the eigenvalues after reduction to tridiagonal form. When *tolerance* is non-positive, it is reset internally as described in Section 8.3.2.
group	Scalar real variable. Grouping criterion for eigenvalues after reduction to tridiagonal form. Corresponding eigenvectors are orthogonalized. When *group* is non-positive, it is reset internally as described in Section 8.4.2.
ier	Scalar integer variable. Return code; set to 0 upon successful return. The following codes indicate errors: -1 Length of axis *row_axis* of ***A*** is < *n*; must be $\geq n$.

-2	Length of axis *col_axis* of ***A*** is < *n*; must be ≥ *n*.
-8	Rank of ***A*** is < 2; must be ≥ 2.
-32	Data type of ***A*** is not real or complex.
-64	*row_axis* or *col_axis* is invalid. 1 ≤ *row_axis*, *col_axis* ≤ rank(***A***) must be true, and *row_axis* and *col_axis* must not be equal.
-128	*nblock* is invalid; must be ≥ 1.
1	The rank of ***d*** is not the same as the rank of ***A***.
2	The rank of ***Q*** is not the equal to the rank of ***A***.
3	The axes of ***Q*** other than axes *row_axis* and *col_axis* do not match the instance axes of ***A*** in order of declaration and extents.
1000+*n*	*n* eigenvectors are not determined after 5 inverse iterations. The non-converged eigenvectors are set to 0 on return.

DESCRIPTION

Given a CM array, ***A***, containing one or more Hermitian matrices, **sym_tred_ eigensystem** computes the eigenvalues of each matrix. If *evects_flag* is set to 1, then **sym_tred_eigensystem** also computes the eigenvector associated with each eigenvalue — that is, it computes

$AQ = Q * \mathrm{DIAG}(d)$.

EXAMPLES

Sample CM Fortran code that uses the **sym_tridiag_eigensystem** routine can be found on-line in the subdirectory

`eigen/tred/cmf/`

of a CMSSL examples directory whose location is site-specific.

8.6 Generalized Eigensystem Analysis of Real Symmetric Matrices

Given a CM array A containing one or more real symmetric matrices A, and a CM array B containing corresponding positive definite matrices B, the **sym_tred_gen_eigensystem** routine solves

$$AQ = BQ\lambda,$$

computing the eigenvalues λ and, if desired, the eigenvectors for each instance. In the case where B is the identity matrix, **sym_tred_gen_eigensystem** performs the same operation as **sym_tred_eigensystem.**

Like **sym_tred_eigensystem, sym_tred_gen_eigensystem** offers a convenient packaging of a series of component operations. Calling **sym_tred_gen_eigensystem** to solve $AQ = BQ\lambda$ is equivalent to performing the following operations:

1. Use **sym_tred_eigensystem** to solve for the eigenvalues and eigenvectors of B. Let λ_B denote the diagonal matrix of eigenvalues of B, and Q_B denote the matrix of eigenvectors of B.

2. Use **gen_matrix_mult** to compute the matrix $B^{-1/2} = Q_B\ (\lambda_B)^{-1/2}\ Q_B{}^T$.

3. Use **gen_matrix_mult** to compute the symmetric matrix $A^* = B^{-1/2}\ A\ B^{-1/2}$.

4. Use **sym_tred_eigensystem** to compute the eigenvalues and eigenvectors of A^*, or call the components of **sym_tred_eigensystem** separately (see Section 8.5) if you want to compute only selected eigenvectors. The eigenvalues of A^* are the same as the eigenvalues of A. Let Q_{A^*} denote the matrix of eigenvectors of A^*.

5. Use **gen_matrix_mult** to compute the eigenvectors of A, $Q_A = B^{-1/2}Q_{A^*}$.

Note that step 2 requires B to be positive definite.

Calling the component routines separately as described above is useful if you want to compute only selected eigenvectors of each matrix. In its current implementation, **sym_tred_gen_eigensystem** computes either none or all of the eigenvectors.

For a detailed description of **sym_tred_gen_eigensystem,** see the man page at the end of this section.

8.6.1 Accuracy

Like **sym_tred_eigensystem**, which is described in Section 8.5, **sym_tred_gen_eigensystem** first reduces each symmetric matrix A to tridiagonal form. The *tolerance* argument controls the accuracy of the eigenvalues after reduction to tridiagonal form, as described in the section on **sym_tridiag_eigenvalues** (Section 8.3). Note that requesting extra accuracy for the eigenvalues of the intermediate tridiagonal matrix improves the quality of the eigenvalues of the original matrix only to the extent that roundoff errors incurred in the reduction to tridiagonal form do not dominate.

The *group* parameter sets the grouping criterion for the eigenvalues after reduction to tridiagonal form, as described in the section on **sym_tridiag_eigenvectors** (Section 8.4).

Generalized Eigensystem Analysis of Real Symmetric Matrices

Given a CM array *A* containing one or more real symmetric matrices *A*, and a CM array *B* containing corresponding positive definite matrices *B*, **sym_tred_gen_eigensystem** solves $AQ=BQ\lambda$, computing the eigenvalues λ and, if desired, the eigenvectors for each instance.

SYNTAX

sym_tred_gen_eigensystem (*d, Q, B, A, n, row_axis, col_axis, nblock, evects_flag, tolerance, group, ier*)

ARGUMENTS

d Real CM array with the same rank and precision as *A*. Axis *row_axis* must have extent 1; axis *col_axis* must have extent $\geq n$. The remaining axes are instance axes matching those of *A* in order of declaration and extents. Thus, each vector within *d* corresponds to a matrix *A* within *A*. Upon completion of **sym_tred_gen_eigensystem**, elements 1 through *n* of each vector in *d* contain the eigenvalues of the corresponding matrix in *A*, sorted in non-decreasing order (with the algebraically smallest eigenvalue stored in the first element).

Q Real CM array with the same rank and precision as *A*. The axes identified by *row_axis* and *col_axis* must have extents $\geq n$; the remaining axes are instance axes that must match those of *A* in order of declaration and extents. Thus, for each matrix *A* within *A* there is a corresponding two-dimensional array of dimensions at least $n \times n$ within *Q*. If *evects_flag* is set to 1, then upon return, the eigenvectors of each matrix within *A* are placed in the upper-left-hand $n \times n$ elements of the corresponding two-dimensional array within *Q*. The eigenvectors lie along axis *col_axis*, and are sorted so that they are returned in the same order as the eigenvalues to which they correspond.

B	Real CM array of the same rank, shape, and precision as ***A***. For each symmetric matrix *A* within ***A***, ***B*** contains a corresponding positive definite matrix *B*, with rows and columns defined by axes *row_axis* and *col_axis*, respectively. Upon return, each matrix ***B*** within ***B*** is overwritten.
A	Real CM array of rank ≥ 2 containing one or more dense, square, symmetric matrices *A*, with rows and columns counted by axes *row_axis* and *col_axis*, respectively. Axes *row_axis* and *col_axis* must have extent *n*. Upon return, each matrix *A* within ***A*** is overwritten with information about the Householder transformations used to reduce the matrix to symmetric tridiagonal form.
n	Scalar integer variable. The number of rows and columns in each symmetric matrix *A* within *A*.
row_axis	Scalar integer variable. The axis of *A* that counts the rows of each symmetric matrix *A*.
col_axis	Scalar integer variable. The axis of *A* that counts the columns of each symmetric matrix *A*.
nblock	Scalar integer variable. Blocking factor. For typical applications, *nblock* = 2 is a good choice. For very large matrices, *nblock* = 4 or even 8 may yield faster reduction. The amount of auxiliary storage used is proportional to *nblock*, so if memory is tight a smaller *nblock* may be a better choice.
evects_flag	Scalar integer variable. If you set *evects_flag* to 0, only the eigenvalues are computed. If you set *evects_flag* to 1, both eigenvalues and eigenvectors are computed.
tolerance	Scalar real variable. Controls the absolute accuracy of the eigenvalues after reduction to tridiagonal form. When *tolerance* is non-positive, it is reset internally as described in the section on **sym_tridiag_eigenvalues**.
group	Scalar real variable. Grouping criterion for eigenvalues after reduction to tridiagonal form. Corresponding eigenvectors are orthogonalized. When *group* is non-positive, it is reset internally as described in the section on **sym_tridiag_eigenvectors**.

ier Scalar integer variable. Return code; set to 0 upon successful return. The error codes are the same as for **sym_tred_eigensystem**, with one addition:

-10	One or more matrices within ***B*** are not positive definite.

DESCRIPTION

Given a CM array ***A*** containing one or more real symmetric matrices ***A***, and a CM array ***B*** containing corresponding positive definite matrices ***B***, **sym_tred_gen_eigensystem** solves $AQ=BQ\lambda$, computing the eigenvalues λ. If *evects_flag* is set to 1, the routine also computes the eigenvectors for each instance — that is, it computes

$AQ = BQ * \mathrm{DIAG}(d)$.

EXAMPLES

Sample CM Fortran code that uses the **sym_tred_gen_eigensystem** routine can be found on-line in the subdirectory

```
eigen/general/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.7 Eigensystem Analysis of Real Symmetric Matrices Using Jacobi Rotations

The **sym_jacobi_eigensystem** routine computes the eigenvalues and eigenvectors of one or more dense real symmetric matrices using the Jacobi method.

In the Jacobi method, iterative sweeps are made through each supplied matrix. In each sweep, successive rotations are applied to the matrix to zero out each off-diagonal element. A sweep consists of the application of $n(n-1)/2$ rotations, where n is the order of the matrix. As each new element is zeroed out, the elements previously zeroed generally become non-zero again. However, with each sweep, the square root of the sum of the squares of the off-diagonal elements, $\sigma = (\Sigma[\text{off-diagonal}]^2)^{1/2}$, decreases. With successive sweeps, the off-diagonal elements approach 0, the matrix approaches a diagonal matrix, and the diagonal elements approach the eigenvalues. Eigenvectors are obtained by applying the Jacobi rotations to the basis of unit vectors.

For a detailed description of **sym_jacobi_eigensystem**, refer to the man page at the end of this section.

8.7.1 Accuracy

The accuracy of the Jacobi method can be described as follows. Provided the convergence criterion is met on return (see the description of the *tolerance* argument in the man page), the absolute error in the computed eigenvalues is

$$\|A\|_F * \max(p(n) * machine\_epsilon, tolerance)$$

where $\|A\|_F$ is the Frobenius norm of A, defined as $\|A\|_F = [\Sigma a_{ij}^2]^{1/2}$, and in practice $p(n)=O(n)$.

The errors in the computed eigenvectors (measured as the angles between the computed eigenvectors and the true eigenvectors) are bounded as follows:

$$\text{angle\_error}(i) < \frac{\|A\|_F * \max(p(n) * machine\_epsilon, tolerance)}{\text{gap}(i)}$$

where gap(i) is the absolute difference between eigenvalue(i) and the next nearest eigenvalue.

Eigensystem Analysis of Real Symmetric Matrices Using Jacobi Rotations

Given a real CM array containing one or more dense symmetric matrices, the **sym_jacobi_eigensystem** routine computes the eigenvalues and eigenvectors of each matrix.

SYNTAX

sym_jacobi_eigensystem (*A, axis_1, axis_2, nsweeps, tolerance, d, Q, evects_flag, ier*)

ARGUMENTS

A — Real CM array of rank greater than or equal to 2, containing one or more dense symmetric matrices *A* whose eigenvalues you want to compute. The declared extents of axes *axis_1* and *axis_2* define the dimensions of the matrices *A*, and must be equal. The values of *A* may be modified by **sym_jacobi_eigensystem.**

axis_1 — Scalar integer variable. Identifies one of the two axes of *A* that count the rows and columns of the embedded matrices *A*.

axis_2 — Scalar integer variable. If *axis_1* identifies the axis of *A* that counts the rows of the embedded matrices *A*, then *axis_2* must identify the axis that counts the matrices' columns; or vice versa.

nsweeps — Scalar integer variable. On input, specifies the maximum number of sweeps to be performed for any supplied matrix. Typical input values lie in the range from 10 to 20. On return, contains the maximum number of sweeps actually performed across all the matrix instances.

tolerance — Scalar real variable. Convergence criterion. *Must have the same precision as A*, and must be > 0. When, for any matrix instance $A = (a_{ij})$, the value $\sigma/\|A\|_F$ decreases below the input value of *tolerance*, the routine stops processing that instance. In this context, σ is the square root of the sum of the squares of the off-diagonal elements, $\sigma = (\Sigma[\text{off-diagonal}]^2)^{1/2}$. $\|A\|_F$ is the

Frobenius norm of A, defined as $\|A\|_F = [\Sigma a_{ij}^2]^{1/2}$. Upon return, *tolerance* contains the largest current value of $\sigma/\|A\|_F$ occurring across all matrix instances A.

d Real CM array with the same rank as ***A***. Must have the same axis extents as ***A***, except that either *axis_1* or *axis_2* must have extent 1. (You may have different layout directives for ***d*** and ***A***.) Thus, each matrix A embedded in ***A*** corresponds to a vector embedded in ***d***; upon return, the eigenvalues of a matrix A in ***A*** are placed in the corresponding vector in ***d***, with the smallest eigenvalue in the first element of the vector.

Q If you set *evects_flag* to 0, you can supply the scalar value 0 for ***Q***. If you set *evects_flag* to 1, ***Q*** must be a real CM array with the same rank and axis extents as ***A***. (It may have different layout directives than ***A*** and ***d***.) Upon return, the eigenvectors for each matrix A within ***A*** are placed in the columns of the corresponding matrix within ***Q***. The eigenvectors are sorted to correspond to the order of the eigenvalues returned in ***d***. Thus, the eigenvector corresponding to the ith eigenvalue of a matrix in ***d*** is returned in column i of the corresponding matrix in ***Q***.

evects_flag Scalar integer variable. Indicates whether eigenvectors are to be computed. If you set *evects_flag* to 0, **sym_jacobi_eigensystem** computes only the eigenvalues; if you set *evects_flag* to 1, both eigenvalues and eigenvectors are computed.

ier Scalar integer variable. Error code. Set to 0 upon successful return, or to one of the following codes:

- -1 ***A***, ***d***, or ***Q*** is not of type real.
- -2 ***A***, ***d***, and ***Q*** do not all have the same rank, or have rank < 2.
- -4 *axis_1* or *axis_2* is < 1 or > rank(***A***), or *axis_1* = *axis_2*.
- -8 The extents of axes *axis_1* and *axis_2* are not equal (that is, the supplied matrix or matrices are not square).
- -16 ***d*** does not conform to the requirements listed above.

-32 *Q* does not conform to the requirements listed above.

DESCRIPTION

Given a real CM array *A* containing one or more dense symmetric matrices *A*, the **sym_jacobi_eigensystem** routine computes the eigenvalues and eigenvectors of each matrix and returns them in the CM arrays ***d*** and ***Q***, respectively — that is, it computes

$AQ = Q * \mathrm{DIAG}(d)$.

In the Jacobi method, iterative sweeps are made through each supplied matrix. In each sweep, successive rotations are applied to the matrix to zero out each off-diagonal element. A sweep consists of the application of $n(n-1)/2$ rotations, where n is the order of the matrix. As each new element is zeroed out, the elements previously zeroed generally become non-zero again. However, with each sweep, the square root of the sum of the squares of the off-diagonal elements, $\sigma = (\Sigma[\text{off-diagonal}]^2)^{1/2}$, decreases. With successive sweeps, the off-diagonal elements approach 0, the matrix approaches a diagonal matrix, and the diagonal elements approach the eigenvalues. Eigenvectors are obtained by applying the Jacobi rotations to the basis of unit vectors.

The **sym_jacobi_eigensystem** routine stops processing each matrix instance $A = (a_{ij})$ as soon as one of the following conditions is met *for that instance*:

- The routine has made *nsweeps* sweeps.
- The value $\sigma/\|A\|_F$ has fallen below the input value of *tolerance*. ($\|A\|_F$ is the Frobenius norm of A, defined as $\|A\|_F = [\Sigma a_{ij}^2]^{1/2}$.)

Upon return, **sym_jacobi_eigensystem** provides the eigenvalues and eigenvectors in the CM arrays ***d*** and ***Q***, as follows:

- The sorted eigenvalues of a matrix A in A are placed in the corresponding vector in ***d***, with the smallest eigenvalue in the first element of the vector.
- The eigenvectors for each matrix A within A are placed in the columns of the corresponding matrix within ***Q***. The eigenvectors are sorted to correspond to the order of the eigenvalues returned in ***d***. Thus, the eigenvector corresponding to the ith eigenvalue of a matrix in ***d*** is returned in column i of the corresponding matrix in ***Q***.

NOTES

Argument Values Modified. Since the values of *A*, *nsweeps*, and *tolerance* may be modified upon return, be sure to reset these arguments to the desired values if you call **sym_jacobi_eigensystem** in a loop.

EXAMPLES

Sample CM Fortran code that uses the **sym_jacobi_eigensystem** routine can be found on-line in the subdirectory

```
eigen/jacobi/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.8 Selected Eigenvalue and Eigenvector Analysis Using a *k*-Step Lanczos Method

The **sym_lanczos** routine finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem

$$Lx = \lambda Bx.$$

B can be positive semi-definite and is the identity for the standard eigenproblem. The operator L must be real and symmetric with respect to B:

$$BL = L^{T}B$$

The algorithm used is a k-step Lanczos algorithm with implicit restart (see reference 12 in Section 8.10). The **sym_lanczos** routine uses a reverse communication interface. You must call **sym_lanczos** iteratively; **sym_lanczos** returns control to the calling program whenever it requires the action of the operator L or B on a vector. You must supply the routines that perform these actions.

For a detailed description of **sym_lanczos** and its associated setup and deallocation routines, **sym_lanczos_setup** and **deallocate_sym_lanczos_setup**, refer to the man page following this section.

8.8.1 The *k*-Step Lanczos Algorithm

The k-step Lanczos algorithm with implicit restart is described in full detail in reference 12. The k-step Lanczos algorithm first performs k steps of the Lanczos factorization of L,

$$LV^{(k)} = V^{(k)}T^{(k)} + r^{(k)}e_k^{T} \qquad (1)$$

where $V = [v_1, v_2, ..., v_k]$ has columns orthonormal with respect to B, T is a tridiagonal matrix of order k, and $r^{(k)}e_k^{T}$ is called the *residual vector*. The starting Lanczos vector v_1 is generated internally if you set the argument *info* to 0 on input; otherwise you must supply it. The goal is to update the original Lanczos factorization of size k (1) in order to drive the residual vector iteratively to zero. This is achieved by forcing the starting vector v_1 into a subspace spanned by the eigenvectors corresponding to the k desired eigenvalues. This purification of the starting vector is accomplished by filtering out the components corresponding to eigenvalues not in the desired portion of the spectrum. To this aim, the sequence (1) is advanced $nv - k$ steps further. The Rayleigh-Ritz procedure applied to the Lanczos subspace of dimension nv yields approximations to k desired eigenva-

lues, but also approximations to *nv* - *k* unwanted eigenvalues. The filtering process is done implicitly through *QR* factorizations of *T* using those "unwanted" *nv* - *k* Ritz values as shifts. The Lanczos vectors and the residual are updated accordingly to yield an updated Lanczos *k*-step factorization of the same form as (1). The updated Lanczos factorization is then advanced again *nv* - *k* steps, and implicit filtering performed. Call this sequence of operations a Lanczos update iteration. The *k*-step method iterates until *k* Ritz values approximate the *k* desired eigenvalues to prescribed accuracy. Error bound *bounds*(*i*) associated with Ritz value *ritz*(*i*) is given by the product of the norm of the current residual and the last component of the eigenvector corresponding to *ritz*(*i*). The convergence criterion for the Ritz value *ritz*(*i*) is $bounds(i) \leq tol \mid ritz(i) \mid$, $i = 1, ..., k$, where *tol* is an input tolerance argument that defaults to machine precision.

8.8.2 Input Arguments and Data Structures

The argument *k* is usually set to the desired number of eigenvalues. The total size of the Lanczos subspace, *nv*, must be at least *k* or 2*k*, depending on whether the eigenvectors are sought, but has no upper bound other than the size of the eigenproblem (or the memory available in the machine). It is generally recommended that $nv = 2k$ even if eigenvectors are not requested. Taking $nv > 2k$ may enhance convergence, but this is problem-dependent. The cost of an implicit restart iteration is roughly $2n * nv^2$ flops.

The *nv* columns of the matrix *V* (the Lanczos vectors) are stored as rows in the CM array *vec*(1:*nv*,...). The subdiagonal of the tridiagonal matrix *T* is stored in the array *work* starting at location *ipntr*(5)+1, while the diagonal is stored in *work* starting at location *ipntr*(5)+*nv*. The current residual vector is stored in the CM array ***resid***. Internally generated exact shifts (i.e., "unwanted" Ritz values) are used when *iparam*(1) = 1. This is the recommended option. However, it is also possible to supply *nv* - *k* external shift values by setting *iparam*(1) = 0. It may be advantageous to supply the roots of a specially constructed filter polynomial (e.g., Tchebyschev polynomials) when *a priori* knowledge about the spectrum is available. Polynomials of degree higher than *nv* – *k* may be applied in a cyclic fashion, supplying *nv* – *k* roots at a time.

The maximum number of Lanczos update iterations is specified in *iparam*(3). The Ritz values are found in the array *ritz* stored in *work* at location *ipntr*(6). Residual bounds are in the array *bounds* stored in *work* starting at location *ipntr*(7). After the final iteration, the first *k* values in *ritz* contain the desired

eigenvalues, and the k vectors stored in *vec*(k+1:2k,...) are the corresponding eigenvectors.

8.8.3 Multiple Eigenvalues

You can extract multiple eigenvalues with **sym_lanczos**, provided the argument *tol* is set to a very small value (close to machine precision). This is possible even though there is no blocking in the current version of **sym_lanczos** and *iparam*(4), the block size for the Lanczos recurrence, is set to 1. The online example illustrates the extraction of multiple eigenvalues of the discretized Laplace operator in three dimensions.

8.8.4 Convergence Properties and Spectral Transformations

The argument *which* allows you to specify the location of the desired eigenvalues to some extent. You can compute either largest (algebraically or abolutely) or smallest (algebraically or absolutely) eigenvalues, or half the eigenvalues from each end of the spectrum. In general, eigenvalues located at both ends of the spectrum emerge first in the Lanczos process. Their convergence rate is proportional to their relative separation, that is, their absolute separation divided by the spread of the spectrum (the total extent of the spectrum on the real axis). Absolutely large eigenvalues always converge rapidly, unless they are tightly clustered. On the other hand, absolutely small eigenvalues are usually much slower to converge, either because they are not at either end of the spectrum or because due to a large spread, their relative separation will be small. This will be true even when they are well separated in an absolute sense.

To accelerate convergence of absolutely small eigenvalues in the standard eigenproblem $Ax = \lambda x$, it is profitable to operate with the inverse operator $L = A^{-1}$ instead of A, since tiny eigenvalues of A are the absolutely largest eigenvalues of L. More generally (see reference 13), if eigenvalues of A around σ are desired, it is profitable to operate with $L = (A - \sigma I)^{-1}$ instead of A, since eigenvalues of A close to σ are mapped into absolutely largest eigenvalues of L. For the generalized eigenvalue problem $Ax = \lambda Bx$, the transformed operator is chosen to be $L = (A - \sigma B)^{-1}B$. Although L is not symmetric, it is symmetric with respect to B. This formulation has the advantage of leaving the eigenvectors unchanged (see references 14 and 15). Other transformations can be used — for example, the Cayley transform, $L = (A - \sigma B)^{-1}(A + \sigma B)$. Of course, eigenvalue approximations returned by **sym_lanczos** must be transformed appropriately to give

approximations to (generalized) eigenvalues of the original operator when any of these transformations are used.

Using transformed operators as described above entails solving linear systems of equations, as well as choosing the shift(s) σ. These operations (as all matrix-vector operations) are left to the user through a reverse communication interface described below. Typical spectral transformations are listed in Table 5. (The "type" values in the table are the values you must specify in the *type* argument on input.) Examples showing how to use the reverse communication interface in these cases are provided below.

Table 5. Examples of eigenproblems and spectral transformations. *M* must be positive semi-definite. Proper use of the reverse communication interface for these cases is described below.

Eigenproblem	Type	Mode	*L*	*B*
$Ax = \lambda x$	I	Regular	A	I
$Ax = \lambda x$	I	Shift-invert	$(A - \sigma I)^{-1}$	I
$Kx = \lambda Mx$	G	Shift-invert	$(K - \sigma M)^{-1}M$	M
$Kx = \lambda Mx$	G	Cayley transf.	$(K - \sigma M)^{-1}(K + \sigma M)$	M

8.8.5 Reverse Communication Interface

The aim of the reverse communication interface is to isolate the matrix-vector operations from the *k*-step Lanczos code. Such operations are performed by routines you supply, on data structures which are the most natural to the problem at hand. To this end, you must call **sym_lanczos** iteratively. It returns control to the calling routine whenever the action of operators *L* or *B* on vectors is required. The reverse communication flag, *ido*, which must be 0 on input to the first call to **sym_lanczos**, dictates which operator is to be applied. The source and destination vectors are the arrays ***src*** and ***dst***, respectively. An extra source array, ***src1***, is needed in some cases.

For standard eigenvalue problems, there is no distinction between *ido* = 1 and *ido* = -1. In both cases, the operation $y = Lx$ is required, where x and y are the source and destination vectors, respectively. For the generalized eigenvalue problem,

the operation $y = Lx$ is always done in two steps, since L is a product of operators. The only difference between *ido* = 1 and *ido* = -1 is that when *ido* = 1, the product Bx is already available in the array *src1* and need not be computed, whereas it must be computed explicitly when *ido* = -1. The value *ido* = -1 is returned by **sym_lanczos** at the first iteration to force the starting vector into the range of L (see reference 14). For generalized eigenproblems, **sym_lanczos** also returns the value *ido* = 2, calling for the operation $y = Bx$ to be executed.

We now give examples of reverse communication interfaces for the problems listed in Table 5. We assume the vectors are represented as one-dimensional arrays:

```
      real resid(n),w(3,n),vec(nv,n),temp_array(),
     &    src(n), src1(n), dst(n)
CMF$LAYOUT resid(),w(:serial,),vec(:serial,),temp_array()
CMF$LAYOUT src(), src1(), dst()
```

and the setup is called successfully:

```
      ....
      call sym_lanczos_setup(resid,vec,w,nv,setup,ier)
      ....
```

Case 1

Suppose we want to solve the standard eigenvalue problem $Ax = \lambda x$ in regular mode. Then $L = A$ and $B = I$. Assume that a call to **matvecA**(*A*,*x*,*y*) computes $y = Ax$. The reverse communication would occur as follows:

```
      .....
      ido=0
10    continue
      call sym_lanczos(ido, 'I', which, k, tol, resid,
     &                 nv, vec, iparam, src, src1, dst,
     &                 ipntr, w, work, lwork, info, setup)
      if (ido .eq. -1 .or. ido .eq. 1) then
          call matvecA (A, src, dst)
      else
          stop
      end if
      go to 10
```

Case 2

Assume now we want to solve $Ax = \lambda x$ in shift-invert mode. Then $L = (A-\sigma I)^{-1}$ and $B = I$. Assume that a call to **solve**(*A,sigma,x,y*) solves $(A-\sigma I)x = y$. Reverse communication would occur as follows:

```
      ...
      ido = 0
10    continue
      call sym_lanczos (ido, 'I', which, k, tol, resid,
     &                  nv, vec, iparam, src, src1, dst,
     &                  ipntr, w, work, lwork, info, setup)
      if (ido .eq. -1 .or. ido .eq. 1) then
          call solve (A, sigma, src, dst)
      else
          stop
      end if
      go to 10
```

Case 3

Suppose now we want to solve $Ax = \lambda Mx$ in shift-invert mode. Then $L = (A-\sigma M)^{-1}M$ and $B = M$. Assume that a call to **matvecM**(*M,x,y*) computes $y = Mx$ and a call to **solve**(*A, M,sigma,x,y*) solves $(A-\sigma M)x = y$. We would have in this case

```
      ...
      ido = 0
10    continue
      call sym_lanczos (ido, 'G', which, k, tol, resid,
     &                   nv, vec, iparam, src, src1, dst,
     &                   ipntr, w, work, lwork, info, setup)
      if (ido .eq. -1) then
          call matvecM (M, src, temp_array)
          call solve (A, M, sigma, dst, temp_array)
          else if (ido .eq. 1) then
          call solve (A, M, sigma, dst, src1)
      else if (ido .eq. 2) then
          call matvecM (M, src, dst)
      else
          stop
      end if
      go to 10
```

Case 4

Finally, suppose we want to solve $Ax = \lambda Mx$ in Cayley mode. Then $L = (A - \sigma M)^{-1}(A + \sigma M)$ and $B = M$. Assume that a call to **matvecM**(*M,x,y*) computes $y = Mx$, a call to **matvecA**(*A,x,y*) computes $y = Ax$, and a call to **solve**(*A, M,sigma,x,y*) solves $(A - \sigma M)x = y$. Reverse communication for this case would be as follows:

```
      ...
      ido = 0
10    continue
      call sym_lanczos (ido, 'G', which, k, tol, resid,
     &                  nv, vec, iparam, src, src1, dst,
     &                  ipntr, w, work, lwork, info, setup)
      if (ido .eq. -1) then
          call matvecM (M, src, dst)
          call matvecA (A, src, temp_array)
          temp_array=temp_array+sigma*dst)
          call solve (A, M, sigma, dst, temp_array)
      else if (ido .eq. 1) then
          call matvecA (A, src, dst)
          dst=dst+sigma*src1
          src1=dst
          call solve (A, M, sigma, dst, src1)
      else if (ido .eq. 2) then
          call matvecM (M, src, dst)
      else
          stop
      end if
      go to 10
```

8.8.6 Data Layout Requirement

The CM arrays ***resid***, ***src***, ***src1***, ***dst***, ***vec***, and ***w*** must adhere to several constraints with regard to shape and layout. Arrays ***resid***, ***src***, ***src1***, and ***dst*** each contain a vector, while ***vec*** and ***w*** are collections of vectors. You may represent each vector with an array of arbitrary dimension, in the manner that is the most natural with respect to the matrix-vector operations. *The product of the axis extents of the arrays representing the vectors must be equal to the size of the eigenproblem.* Arrays ***resid***, ***src***, ***src1***, and ***dst*** must have the same shape and layout. Furthermore, ***vec*** and ***w*** must each have an extra (instance) axis, which must be the first axis and must have extent at least ***nv*** in ***vec*** and at least 3 in ***w***. This axis must be made local to a processing element so that the vectors, which have identical shape and layout, are "stacked up" in memory. This is accomplished by declaring the instance axis **:serial** in the calling program using a **CMF$LAYOUT** directive.

For example, in the one-dimensional case where the size of the eigenproblem is *n*, array declarations would be as follows:

```
      real vec(nv,n),w(3,n),resid(n),src(n),src1(n),dst(n)
CMF$LAYOUT vec(:serial,),w(:serial,),resid()
CMF$LAYOUT src(),src1(),dst()
```

In the two-dimensional case where the size of the problem is *n1 * n2 = n*, the array declarations would be

```
      real vec(nv,n1,n2),w(3,n1,n2),resid(n1,n2)
      real src(n1,n2),src1(n1,n2),dst(n1,n2)
CMF$LAYOUT vec(:serial,,),w(:serial,,),resid(,)
CMF$LAYOUT src(,),src1(,),dst(,)
```

8.8.7 On-Line Example

The on-line example illustrates the use of **sym_lanczos** to extract a few eigenpairs of a discretized Laplace operator in three dimensions. Vectors are represented as three-dimensional arrays, the natural data structure for this problem. Because the three dimensions are equal, there is a three-fold degeneracy of the eigenvalues. For that reason the convergence is rather slow even though the largest eigenvalues are extracted. The tolerance is set close to machine precision to ensure extraction of multiple eigenvalues. For the location of the on-line example, see the man page.

8.8.8 Acknowledgments

The **sym_lanczos** routine is a CM Fortran adaptation for the CM of a Fortran77 code written by D. Sorensen and P. Vu at the Center for Research on Parallel Computation, Rice University (see reference 12). The portions of the code operating on front-end arrays make use of LAPACK (see reference 16) and BLAS routines which have been integrated so that **sym_lanczos** is self-contained.

Selected Eigenvalue and Eigenvector Analysis Using a *k*-Step Lanczos Method

The **sym_lanczos** routine finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. B can be positive semi-definite and is the identity for the standard eigenproblem. The operator L must be real and symmetric with respect to B; that is, $BL = L^T B$. The algorithm used is a k-step Lanczos algorithm with implicit restart. The routine uses a reverse communication interface. You must call **sym_lanczos** iteratively; **sym_lanczos** returns control to the calling program whenever it requires the action of the operator L or B on a vector. You must supply the routines that perform these actions.

SYNTAX

sym_lanczos_setup (*resid, vec, w, nv, setup, ier*)

sym_lanczos (*ido, type, which, k, tol, resid, nv, vec, iparam, src, src1, dst, ipntr, w, work, lwork, info, setup*)

deallocate_sym_lanczos_setup (*setup*)

ARGUMENTS

ido

Scalar integer variable. Reverse communication flag. *ido* must be zero on the first call to **sym_lanczos**. The **sym_lanczos** routine sets *ido* to indicate the type of operation to be performed by the calling program. The calling program has the responsibility of carrying out the requested operation and calling **sym_lanczos** again. The values of *ido* have the meanings listed below. All values except 0 are returned to the calling program.

0	The calling program supplies this value on the first call to **sym_lanczos**.
-1	The calling program must compute $y = Lx$, where *src* contains x *dst* contains y

This value is for the initialization phase, and is used to force the starting vector into the range of *L*.

1 The calling program must compute $y = Lx$, where

src contains x

dst contains y

src1 contains Bx

2 The calling program must compute $y = Bx$, where

src contains x

dst contains y

3 The calling program must compute and store the shifts in the first $nv - k$ locations of *work*. This value is returned only if you previously assigned *iparam*(1) the value 0.

99 The computation is complete.

After the initialization phase, when the routine is used in either the shift-invert mode or the Cayley transform mode (see the Description section below), the vector Bx is already available; you need not recompute it in forming Lx.

type Front-end string variable declared as character*1. The value you supply specifies the type of eigenvalue problem, as follows:

'I' Standard eigenvalue problem, $Ax = \lambda x$

'G' Generalized eigenvalue problem, $Ax = \lambda Bx$

which Front-end string variable declared as character*2. Supply one of the following values:

'LA' Compute the k largest (algebraic) eigenvalues.

'SA' Compute the k smallest (algebraic) eigenvalues.

'LM' Compute the k largest (in magnitude) eigenvalues.

'SM' Compute the k smallest (in magnitude) eigenvalues.

'BE' Compute k eigenvalues, half from each end of the spectrum. When k is odd, compute one more from the high end than from the low end.

k Scalar integer variable. The number of eigenvalues of L to be computed.

tol Scalar real variable. The stopping criterion. The relative accuracy of the ith Ritz value is considered acceptable if *bounds*(i) $\leq$ *tol**ABS(*ritz*(i)), where *bounds*(k) and *ritz*(k) are arrays located within *work*, with starting locations *work*(*ipntr*(7)) and *work*(*ipntr*(6)), respectively. The error bound *bounds*(i) associated with Ritz value *ritz*(i) is given by the product of the norm of the current residual and the last component of the eigenvector corresponding to *ritz*(i). If the *tol* value you supply is less than or equal to 0, *tol* defaults to the machine precision.

resid Real CM array of rank greater than or equal to 1. The product of the axis extents must be equal to the size of the eigenproblem. If you set *info* to 0, ***resid*** is set to a random initial residual vector internally. If *info* is not 0, you must supply the initial residual vector in ***resid***.

Upon final return, ***resid*** contains the final residual vector.

nv Scalar integer variable. The declared extent of the first axis of ***vec***. Must be less than or equal to the size of the eigenproblem. This value determines how many Lanczos vectors are generated at each iteration. After the startup phase, in which k Lanczos vectors are generated, the algorithm generates ($nv - k$) Lanczos vectors at each subsequent update iteration.

If *iparam*(2) is less than or equal to 0, then *nv* must be greater than or equal to k. If *iparam*(2) is greater than 0, then *nv* must be greater than or equal to $2k$.

It is generally recommended that $nv = 2k$ even if eigenvectors are not requested. Taking $nv > 2k$ may enhance convergence, but this is problem-dependent. The cost of an implicit restart iteration is roughly $2n * nv^2$ flops.

vec Real CM array of rank one greater than that of ***resid***. The first axis must have extent at least *nv* and must be serial. The remaining axes must match the axes of ***resid*** in order of declaration, extents, and layout. Upon successful final return,

- ***vec***(1:*k*, :, ..., :) are the Lanczos vectors.
- If requested by *iparam*(2), ***vec***(*k*+1:2*k*, :, ..., :) are the eigenvectors corresponding to (and in the same order as) the converged eigenvalues.

iparam One-dimensional front-end integer array of length 5.

iparam(1) Specifies the method for selecting the implicit shifts. Supply one of the values listed below. The shifts selected at each iteration are used to filter out the components of the unwanted eigenvector.

0 The shifts are to be provided by the user via reverse communication when *ido* = 3.

1 **sym_lanczos** applies exact shifts with respect to the reduced tridiagonal matrix. Using exact shifts is equivalent to restarting the iteration from the beginning after updating the starting vector with a linear combination of Ritz vectors associated with the desired eigenvalues.

iparam(2) Specifies whether eigenvectors are to be computed, as follows:

iparam(2) ≤ 0 Compute only the eigenvalues.

iparam(2) > 0 Compute both eigenvalues and eigenvectors.

iparam(3) On input, specifies the maximum number of Lanczos update iterations allowed. On return, is set to the actual number of Lanczos update iterations performed.

iparam(4) Block size to be used in the recurrence. Must be set to 1 in the current release.

iparam(5) On return, specifies the number of converged eigenvalues.

src Real CM array of the same rank, shape, and layout as ***resid***. Contains the current operand vector *x*.

src1 Real CM array of the same rank, shape, and layout as ***resid***. Contains the vector Bx (used in shift-invert mode).

dst Real CM array of the same rank, shape, and layout as ***resid***. Contains the current result vector y.

ipntr One-dimensional front-end integer array of length 7. On return, contains pointers to mark the locations in the *work* array for matrices and/or vectors used by the Lanczos iteration.

- *ipntr*(1) Reserved for internal use.
- *ipntr*(2) Reserved for internal use.
- *ipntr*(3) Reserved for internal use.
- *ipntr*(4) Points to the next available location in *work* that is untouched by the program.
- *ipntr*(5) Points to the starting location of the (nv+1) x 2 tridiagonal matrix in *work*.
- *ipntr*(6) Points to the starting location of the Ritz values array, *ritz*, in *work*. Upon successful final return, the first k values of *ritz* are the desired eigenvalues, returned in increasing order.
- *ipntr*(7) Points to the starting location of the error bounds array, *bounds*, in *work*.

w Real CM array with rank one greater than that of ***resid***. The first axis must have extent at least 3 and must be serial. The remaining axes must match the axes of ***resid*** in order of declaration, extents, and layout. This array is used internally.

work Real one-dimensional front-end array of length *lwork*. If *iparam*(2) is greater than 0, the eigenvectors of the final tridiagonal matrix (see *ipntr*(5)) are returned in the first k^2 locations of *work*, stored by columns.

lwork Scalar integer variable. Supply the declared dimension of *work*. If

LW1 = $nv(nv + 1)$

LW2 = $k(k + 4)$

LW3 = $4nv + 2$

then

- If *iparam*(2) is less than or equal to 0, *lwork* must be at least LW1 + LW3.
- If *iparam*(2) is greater than 0, *lwork* must be at least MAX(LW1, LW2) + LW3.

info Scalar integer variable. The input value affects the initial residual vector, as follows:

- If *info* = 0, ***resid*** is set to a random initial residual vector internally.
- If *info* is not 0, you must supply the initial residual vector in ***resid.***

On return, *info* contains one of the following error codes:

0	Normal exit.
-2	*k* must be positive.
-3	*nv* must be greater than *k* (or 2*k*, when eigenvectors are requested), and less than or equal to the size of the eigenproblem.
-4	The maximum number of Lanczos update iterations must be greater than zero.
-5	*which* must be one of the following: 'LM', 'SM', 'LA', 'SA' or 'BE'.
-6	*type* must be 'I' or 'G'.
-7	The length of *work* is not sufficien..
-8	Error return from the tridiagonal eigenvalue calculation.
-9	Starting vector is zero.
-9999	Maximum number of Lanczos update iterations have occurred.

setup One-dimensional integer array of length 3. Internal variable. When you call **sym_lanczos** or **deallocate_sym_lanczos_setup**, supply the values returned by **sym_lanczos_setup**.

ier Scalar integer variable. Set to 0 upon successful return. Upon return from **sym_lanczos_setup**, may contain the following error codes:

- -1 The first dimension of ***vec*** or ***w*** is not declared **:serial**.
- -2 The serial dimension of ***vec*** or ***w*** has extent less than *nv* or 3, respectively.
- -3 (rank ***vec***), (rank ***w***), and (rank ***resid*** + 1) are not equal.
- -4 The sections of ***vec*** and ***w*** containing the vectors and indexed by the first dimension do not have the same shape as ***resid***.

DESCRIPTION

Intended Use. The **sym_lanczos** routine solves the following eigenproblems:

- $Ax = \lambda x$, A symmetric, $L = A$, $B = I$.
- $Ax = \lambda Mx$, A symmetric, M symmetric positive definite, $L = M^{-1}A$, $B = M$.
- $Kx = \lambda Mx$, K symmetric, M symmetric semi-definite, $L = (K-\sigma M)^{-1}M$, $B = M$ (shift-invert mode).
- $Kx = \lambda KGx$, K symmetric positive semi-definite, KG symmetric indefinite, $L = (K-\sigma KG)^{-1}K$, $B = K$ (shift-invert mode).
- $Ax = \lambda Mx$, A symmetric, M symmetric positive definite, $L = (A - \sigma M)^{-1}(A + \sigma M)$, $B = M$ (Cayley transform mode).

Setup and Deallocation. To use **sym_lanczos**, follow these steps:

1. Call **sym_lanczos_setup**.

This routine generates three setup IDs and returns them in the array *setup* of length 3. You must supply this *setup* array in all subsequent **sym_lanczos** and **deallocate_sym_lanczos_setup** calls associated with this setup call.

2. Call **sym_lanczos** iteratively, as described under **Reverse Communication Interface**, below.

   You can use the same *setup* array to solve more than one eigenproblem sequentially, as long as the array geometries are the same. You can also have more than one setup active at a time.

3. Call **deallocate_sym_lanczos_setup.**

   This routine deallocates the memory associated with the three setup IDs.

Returned Eigenvalues and Eigenvectors. Upon successful final return,

- The k desired eigenvalues are located (in algebraically increasing order) in the first k locations of *ritz*. The argument *ipntr*(6) points to the starting location of the *ritz* array within *work*.
- If eigenvectors are requested (*iparam*(2) > 0), the corresponding eigenvectors are returned in *vec*(k+1:2k, :, ..., :).

Reverse Communication Interface. The aim of the reverse communication interface is to isolate the matrix-vector operations from the k-step Lanczos code. Such operations are performed by routines you supply, on data structures which are the most natural to the problem at hand. To this end, you must call **sym_lanczos** iteratively. It returns control to the calling routine whenever the action of operators L or B on vectors is required. The reverse communication flag, *ido*, which must be 0 on input to the first call to **sym_lanczos**, dictates which operator is to be applied. For standard eigenvalue problems, there is no distinction between *ido* = 1 and *ido* = -1. In both cases, the operation $y = Lx$ is required, where x and y are the source and destination vectors, ***src*** and ***dst***, respectively. For the generalized eigenvalue problem, the operation $y = Lx$ is always done in two steps, since L is a product of operators. The only difference between *ido* = 1 and *ido* = -1 is that when *ido* = 1, the product Bx is already available in ***src1*** and need not be computed, whereas it must be computed explicitly when *ido* = -1. The value *ido* = -1 is returned by **sym_lanczos** at the first iteration to force the starting vector into the range of L (see reference 14). For generalized eigenproblems, **sym_lanczos** also returns the value *ido* = 2, calling for the operation $y = Bx$ to be executed.

NOTES

Use of Array *w*. Do not use the CM array *w* as temporary workspace.

Data Layout. The CM arrays ***resid***, ***src***, ***src1***, ***dst***, ***vec***, and ***w*** must adhere to several constraints with regard to shape and layout. Arrays ***resid***, ***src***, ***src1***, and ***dst*** each contain a vector, while ***vec*** and ***w*** are collections of vectors. You may represent each vector with an array of arbitrary dimension, in the manner that is the most natural with respect to the matrix-vector operations. *The product of the axis extents of the arrays representing the vectors must be equal to the size of the eigenproblem.* Arrays ***resid***, ***src***, ***src1***, and ***dst*** must have the same shape and layout. Furthermore, ***vec*** and ***w*** must each have an extra (instance) axis, which must be the first axis and must have extent at least ***nv*** in ***vec*** and at least 3 in ***w***. This axis must be made local to a processing element so that the vectors, which have identical shape and layout, are "stacked up" in memory. This is accomplished by declaring the instance axis **:serial** in the calling program using a **CMF$LAYOUT** directive.

For example, in the one-dimensional case where the size of the eigenproblem is ***n***, array declarations would be as follows:

```
      real vec(nv,n),w(3,n),resid(n),src(n),src1(n),dst(n)
CMF$LAYOUT vec(:serial,),w(:serial,),resid()
CMF$LAYOUT src(),src1(),dst()
```

In the two-dimensional case where the size of the problem is ***n1 * n2 = n***, the array declarations would be

```
      real vec(nv,n1,n2),w(3,n1,n2),resid(n1,n2)
      real src(n1,n2),src1(n1,n2),dst(n1,n2)
CMF$LAYOUT vec(:serial,,),w(:serial,,),resid(,)
CMF$LAYOUT src(,),src1(,),dst(,)
```

On-Line Example. The on-line example illustrates the use of **sym_lanczos** to extract a few eigenpairs of a discretized Laplace operator in three dimensions. Vectors are represented as three-dimensional arrays, the natural data structure for this problem. Because the three dimensions are equal, there is a three-fold degeneracy of the eigenvalues. For that reason the convergence is rather slow even though the largest eigenvalues are extracted. The tolerance is set close to machine precision to ensure extraction of multiple eigenvalues. For the location of the on-line example, see below.

Acknowledgments. The **sym_lanczos** routine is a CM Fortran adaptation for the CM of a Fortran77 code written by D. Sorensen and P. Vu at the Center for Research on Parallel Computation, Rice University (see reference 12 in Section 8.10). The portions

of the code operating on front-end arrays make use of LAPACK (see reference 16) and BLAS routines which have been integrated so that **sym_lanczos** is self-contained.

EXAMPLES

Sample CM Fortran code that uses the **sym_lanczos** routine can be found on-line in the subdirectory

```
eigen/lanczos/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.9 Selected Eigenvalue and Eigenvector Analysis Using a *k*-Step Arnoldi Method

The **gen_arnoldi** routine finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem

$$Lx = \lambda Bx.$$

B is symmetric and can be positive semi-definite; it is the identity for the standard eigenproblem.

The algorithm used is a k-step Arnoldi algorithm with implicit restart (see reference 12 in Section 8.10). The **gen_arnoldi** routine uses a reverse communication interface. You must call **gen_arnoldi** iteratively; **gen_arnoldi** returns control to the calling program whenever it requires the action of the operator L or B on a vector. You must supply the routines that perform these actions.

For a detailed description of **gen_arnoldi** and its associated setup and deallocation routines, **gen_arnoldi_setup** and **deallocate_gen_arnoldi_setup**, refer to the man page following this section.

If L is symmetric with respect to B ($BL = L^TB$), you can save significant time by using **sym_lanczos** (described in the Version 3.0 CMSSL documentation) rather than **gen_arnoldi**.

8.9.1 The *k*-Step Arnoldi Algorithm

The k-step Arnoldi algorithm with implicit restart is described in full detail in reference 12. The k-step Arnoldi algorithm first performs k steps of the Arnoldi factorization of L,

$$LV^{(k)} = V^{(k)}H^{(k)} + r^{(k)}e_k^T \qquad (1)$$

where $V = [v_1, v_2, ..., v_k]$ has columns orthonormal with respect to B, H is a Hessenberg matrix of order k, and $r^{(k)}e_k^T$ is called the *residual vector*. The starting Arnoldi vector v_1 is generated internally if you set the argument *info* to 0 on input; otherwise you must supply it. The goal is to update the original Arnoldi factorization of size k (1) in order to drive the residual vector iteratively to zero. This is achieved by forcing the starting vector v_1 into a subspace spanned by the eigenvectors corresponding to the k desired eigenvalues. This purification of the starting vector is accomplished by filtering out the components corresponding to eigenvalues not in the desired portion of the spectrum. To this aim, the sequence

(1) is advanced *nv* - *k* steps further. The Rayleigh-Ritz procedure applied to the Arnoldi subspace of dimension *nv* yields approximations to *k* desired eigenvalues, but also approximations to *nv* - *k* unwanted eigenvalues. The filtering process is done implicitly through *QR* factorizations of *H* using those "unwanted" *nv* - *k* Ritz values as shifts. The Arnoldi vectors and the residual are updated accordingly to yield an updated Arnoldi *k*-step factorization of the same form as (1). The updated Arnoldi factorization is then advanced again *nv* - *k* steps, and implicit filtering performed. Call this sequence of operations an Arnoldi update iteration. The *k*-step method iterates until *k* Ritz values approximate the *k* desired eigenvalues to a prescribed accuracy, *tol*, which defaults to machine precision.

8.9.2 Input Arguments and Data Structures

The argument *k* is usually set to the desired number of eigenvalues. The total size of the Arnoldi subspace, *nv*, must be at least *k* or 2*k*, depending on whether the eigenvectors are sought, but has no upper bound other than the size of the eigenproblem (or the memory available in the machine). It is generally recommended that *nv* = 2*k* even if eigenvectors are not requested. Taking *nv* > 2*k* may enhance convergence, but this is problem-dependent. The cost of an implicit restart iteration is roughly $2n * nv^2$ flops.

The *nv* columns of the matrix *V* (the Arnoldi vectors) are stored as rows in the CM array ***vec***(1:*nv*,...). The current residual vector is stored in the CM array ***resid***. Internally generated exact shifts (i.e., "unwanted" Ritz values) are used when *iparam*(1) = 1. This is the recommended option. However, it is also possible to supply *nv* - *k* external shift values by setting *iparam*(1) = 0. It may be advantageous to supply the roots of a specially constructed filter polynomial when *a priori* knowledge about the spectrum is available. Polynomials of degree higher than *nv* – *k* may be applied in a cyclic fashion, supplying *nv* – *k* roots at a time.

The maximum number of Arnoldi update iterations is specified in *iparam*(3). The real and imaginary parts of the Ritz values are found in the arrays *ritzr* and *ritzi*, stored in *work* at locations *ipntr*(6) and *ipntr*(7), respectively. Residual bounds are in the array *bounds* stored in *work* starting at location *ipntr*(8). After the final iteration, the first *k* values in *ritzr* and *ritzi* contain the real and imaginary parts, respectively, of the desired eigenvalues, and the *k* vectors stored in ***vec***(*k*+1:2*k*,...) are the corresponding eigenvectors. In the case of complex conjugate pairs, the eigenvalue with positive imaginary part is always first. Hence, if the *j*th and (*j*+1)st eigenvalues are the conjugate pair $\alpha+i\beta$ and $\alpha-i\beta$, then *ritzr*(*j*)

= *ritzr*(*j*+1) = α, *ritzi*(*j*) = β, and *ritzi*(*j*+1) = $-\beta$. Corresponding eigenvectors are *u*+*iv* and *u*−*iv*, with *u* = *vec*(*k*+*j*, :, ..., :) and *v* = *vec*(*k*+*j*+1, :, ..., :).

8.9.3 Reverse Communication Interface

The aim of the reverse communication interface is to isolate the matrix-vector operations from the *k*-step Arnoldi code. Such operations are performed by routines you supply, on data structures which are the most natural to the problem at hand. To this end, you must call **gen_arnoldi** iteratively. It returns control to the calling routine whenever the action of operators *L* or *B* on vectors is required. The reverse communication flag, *ido*, which must be 0 on input to the first call to **gen_arnoldi**, dictates which operator is to be applied. The source and destination vectors are contained in the arrays *src* and *dst*, respectively. An extra source array, *src1*, is needed in some cases.

For standard eigenvalue problems, there is no distinction between *ido* = 1 and *ido* = −1. In both cases, the operation *y* = *Lx* is required, where *x* and *y* are the source and destination vectors, respectively. For the generalized eigenvalue problem, the operation *y* = *Lx* is always done in two steps, since *L* is a product of operators. The only difference between *ido* = 1 and *ido* = −1 is that when *ido* = 1, the product *Bx* is already available in the array *src1* and need not be computed, whereas it must be computed explicitly when *ido* = −1. The value *ido* = −1 is returned by **gen_arnoldi** at the first iteration to force the starting vector into the range of *L* (see reference 14). For generalized eigenproblems, **gen_arnoldi** also returns the value *ido* = 2, calling for the operation *y* = *Bx* to be executed.

We now give examples of reverse communication. We assume the vectors are represented as one-dimensional arrays:

```
      real resid(n),w(3,n),vec(nv,n),temp_array(n)
      real src(n),src1(n),dst(n)
CMF$LAYOUT resid(),w(:serial,),vec(:serial,),temp_array()
CMF$LAYOUT src(), src1(), dst()
```

and the setup is called successfully:

```
      ....
      call gen_arnoldi_setup(resid,vec,w,nv,setup,ier)
      ....
```

Example 1

Suppose we want to solve $Ax = \lambda x$ in regular mode ($L = A$ and $B = I$). Assume that a call to **matvecA**(A, x, y) computes $y = Ax$, and that exact shifts are used. Reverse communication would occur as follows:

```
      ...
      ido = 0
 10   continue

      call gen_arnoldi(ido, 'I', which, k, tol, resid, nv, vec,
     &                 iparam, src, src1, dst, ipntr, w, work,
     &                 lwork, info,setup)

      if (ido .eq. -1 .or. ido .eq. 1) then

          call matvecA (A, src, dst)

      else
          stop
      end if
      go to 10
```

Example 2

Suppose we want to solve $Ax = \lambda x$ in shift-invert mode. Then $L = (A-\sigma I)^{-1}$ and $B = I$; σ may be complex. Assume that a call to **solve**(A, σ, x, y) solves $(A-\sigma I)x = y$ (possibly in complex arithmetic), and that exact shifts are used. Reverse communication would occur as follows:

```
      ...
      ido = 0
 10   continue

      call gen_arnoldi(ido, 'I', which, k, tol, resid, nv, vec,
     &                 iparam, src, src1, dst,
     &                 ipntr, w, work, lwork, info, setup)

      if (ido .eq. -1 .or. ido .eq. 1) then

          call solve (A, sigma, complex_array, src)
          dst = real(complex_array)

      else
          stop
      end if
      go to 10
```

Example 3

Suppose now we want to solve $Ax = \lambda Mx$ in shift-invert mode. Then $L = (A-\sigma M)^{-1}M$ and $B = M$; σ may be complex. Assume that a call to **matvecM**(M,x,y) computes $y = Mx$, a call to **solve**(A, M, σ, x, y) solves $(A-\sigma M)x = y$ (possibly in complex arithmetic), and exact shifts are used. We would have in this case

```
      ...
      ido = 0
 10   continue

      call gen_arnoldi (ido, 'G', which, k, tol, resid,
     &                  nv, vec, iparam, src, src1, dst,
     &                  ipntr, w, work, lwork, info, setup)

      if (ido .eq. -1) then

          call matvecM (M, src, temp_array)
          call solve (A, M, sigma, complex_array, temp_array)
          dst = real(complex_array)


      else if (ido .eq. 1) then

          call solve (A, M, sigma, complex_array, src1)
          dst = real(complex_array)

      else if (ido .eq. 2) then

          call matvecM(M, src, dst)

      else
          stop
      end if
      go to 10
```

8.9.4 Data Layout Requirement

The CM arrays ***resid***, ***src***, ***src1***, ***dst***, ***vec***, and ***w*** must adhere to several constraints with regard to shape and layout. Arrays ***resid***, ***src***, ***src1***, and ***dst*** each contain a vector, while ***vec*** and ***w*** are collections of vectors. You may represent each vector with an array of arbitrary dimension, in the manner that is the most natural with respect to the matrix-vector operations. *The product of the axis extents of the arrays representing the vectors must be equal to the size of the eigenproblem.* Arrays ***resid***, ***src***, ***src1***, and ***dst*** must have the same shape and layout. Further-

more, *vec* and *w* must each have an extra (instance) axis, which must be the first axis and must have extent at least *nv* in *vec* and at least 3 in *w*. This axis must be made local to a processing element so that the vectors, which have identical shape and layout, are "stacked up" in memory. This is accomplished by declaring the instance axis **:serial** in the calling program using a **CMF$LAYOUT** directive.

For example, in the one-dimensional case where the size of the eigenproblem is *n*, array declarations would be as follows:

```
      real vec(nv,n),w(3,n),resid(n),src(n),src1(n),dst(n)
CMF$LAYOUT vec(:serial,),w(:serial,),resid()
CMF$LAYOUT src(),src1(),dst()
```

In the two-dimensional case where the size of the problem is *n1* * *n2* = *n*, the array declarations would be

```
      real vec(nv,n1,n2),w(3,n1,n2),resid(n1,n2)
      real src(n1,n2),src1(n1.n2),dst(n1,n2)
CMF$LAYOUT vec(:serial,,),w(:serial,,),resid(,)
CMF$LAYOUT src(,),src1(,),dst(,)
```

8.9.5 On-Line Example

The on-line **gen_arnoldi** example is taken from the aeronautical industry. The so-called Tolosa matrix comes from the Aerospatiale Aircraft Division in Toulouse, France. It is part of the Harwell-Boeing collection (see reference 18). The eigenvalues with largest imaginary part are of interest to engineers. The matrix is very sparse with a block structure and is of order N=90+5k where k is an integer greater than 1. In the example, we choose k=782; hence, N=4000. The default of normality, which grows exponentially with N for this matrix, accounts for the discrepancy between the estimated bounds and the actual residuals (see reference 19). For the location of the on-line example, see the man page.

8.9.6 Acknowledgments

The **gen_arnoldi** routine is a CM Fortran adaptation for the CM of a Fortran77 code written by D. Sorensen and P. Vu at the Center for Research on Parallel Computation, Rice University (see reference 12). The portions of the code operating on front-end arrays make use of LAPACK (see reference 16) and BLAS routines which have been integrated so that **gen_arnoldi** is self-contained.

We thank S.Godet-Thobie at CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique) for providing us with the Tolosa matrix and the routines to build it, which are included in the on-line example.

Selected Eigenvalue and Eigenvector Analysis Using a *k*-Step Arnoldi Method

The **gen_arnoldi** routine finds selected solutions $\{\lambda, x\}$ to the real standard or generalized eigenvalue problem $Lx = \lambda Bx$. B is symmetric and can be positive semi-definite; it is the identity for the standard eigenproblem. The operator L must be real but not necessarily symmetric. The algorithm used is a k-step Arnoldi algorithm with implicit restart. The routine uses a reverse communication interface. You must call **gen_arnoldi** iteratively; **gen_arnoldi** returns control to the calling program whenever it requires the action of the operator L or B on a vector. You must supply the routines that perform these actions.

SYNTAX

gen_arnoldi_setup (*resid, vec, w, nv, setup, ier*)

gen_arnoldi (*ido, type, which, k, tol, resid, nv, vec, iparam, src, src1, dst, ipntr, w, work, lwork, info, setup*)

deallocate_gen_arnoldi_setup (*setup*)

ARGUMENTS

ido

Scalar integer variable. Reverse communication flag. *ido* must be zero on the first call to **gen_arnoldi**. The **gen_arnoldi** routine sets *ido* to indicate the type of operation to be performed by the calling program. The calling program has the responsibility of carrying out the requested operation and calling **gen_arnoldi** again. The values of *ido* have the meanings listed below. All values except 0 are returned to the calling program.

0	The calling program supplies this value on the first call to **gen_arnoldi**.
-1	The calling program must compute $y = Lx$, where *src* contains x *dst* contains y

This value is for the initialization phase, and is used to force the starting vector into the range of *L*.

1 The calling program must compute $y = Lx$, where

src contains x

dst contains y

src1 contains Bx

2 The calling program must compute $y = Bx$, where

src contains x

dst contains y

3 The calling program must compute and store the real and imaginary parts of the shifts in the first $2(nv - k)$ locations of *work*. This value is returned only if you previously assigned *iparam*(1) the value 0.

99 The computation is complete.

After the initialization phase, when the routine is used in shift-invert mode (see the Description section below), the vector Bx is already available; you need not recompute it in forming Lx.

type Front-end string variable declared as character*1. The value you supply specifies the type of eigenvalue problem, as follows:

'I' Standard eigenvalue problem, $Ax = \lambda x$

'G' Generalized eigenvalue problem, $Ax = \lambda Bx$

which Front-end string variable declared as character*2. Supply one of the following values:

'LM' Compute the k eigenvalues of largest magnitude.

'SM' Compute the k eigenvalues of smallest magnitude.

'LR' Compute the k eigenvalues with largest real part.

'SR' Compute the k eigenvalues with smallest real part.

'LI' Compute the k eigenvalues with largest imaginary part.

'SI' Compute the k eigenvalues with smallest imaginary part.

k Scalar integer variable. The number of eigenvalues of L to be computed.

tol Scalar real variable. The stopping criterion. The relative accuracy of the jth Ritz value is considered acceptable if $bounds(j) \leq tol*|(ritz(j)|$, where $ritz(j)=ritzr(j)+i*ritzi(j)$, and where $ritzr(k)$, $ritzi(k)$, and $bounds(k)$ are arrays located within *work*, with starting locations $work(ipntr(6))$, $work(ipntr(7))$, and $work(ipntr(8))$, respectively. The error bound $bounds(j)$ associated with Ritz value $ritz(j)$ is given by the product of the norm of the current residual and the last component of the eigenvector corresponding to $ritz(j)$. If the *tol* value you supply is less than or equal to 0, *tol* defaults to the machine precision.

resid Real CM array of rank greater than or equal to 1. The product of the axis extents must be equal to the size of the eigenproblem. If you set *info* to 0, ***resid*** is set to a random initial residual vector internally. If *info* is not 0, you must supply the initial residual vector in ***resid.***

Upon final return, ***resid*** contains the final residual vector.

nv Scalar integer variable. The declared extent of the first axis of ***vec***. Must be less than or equal to the size of the eigenproblem. This value determines how many Arnoldi vectors are generated at each iteration. After the startup phase, in which k Arnoldi vectors are generated, the algorithm generates $(nv - k)$ Arnoldi vectors at each subsequent update iteration.

If $iparam(2)$ is less than or equal to 0, then nv must be greater than or equal to k. If $iparam(2)$ is greater than 0, then nv must be greater than or equal to $2k$.

It is generally recommended that $nv = 2k$ even if eigenvectors are not requested. Taking $nv > 2k$ may enhance convergence, but this is problem-dependent. The cost of an implicit restart iteration is roughly $2n * nv^2$ flops.

vec Real CM array of rank one greater than that of ***resid***. The first axis must have extent at least *nv* and must be serial. The remaining axes must match the axes of ***resid*** in order of declaration, extents, and layout. Upon successful final return,

- ***vec***(1:*k*, :, ..., :) are the Arnoldi vectors.
- If requested by *iparam*(2), ***vec***(*k*+1:2*k*, :, ..., :) are the eigenvectors corresponding to (and in the same order as) the converged eigenvalues. In the case of complex conjugate pairs, the eigenvalue with positive imaginary part is always first. Hence, if the *j*th and (*j*+1)st eigenvalues are the conjugate pair $\alpha+i\beta$ and $\alpha-i\beta$, then *ritzr*(*j*) = *ritzr*(*j*+1) = α, *ritzi*(*j*) = β, and *ritzi*(*j*+1) = $-\beta$. Corresponding eigenvectors are $u+iv$ and $u-iv$, with u = ***vec***(*k*+*j*, :, ..., :) and v = ***vec***(*k*+*j*+1, :, ..., :).

iparam One-dimensional front-end integer array of length 5.

iparam(1) Specifies the method for selecting the implicit shifts. Supply one of the values listed below. The shifts selected at each iteration are used to filter out the components of the unwanted eigenvector.

0 The shifts are to be provided by the user via reverse communication when *ido* = 3. The real and imaginary parts of the *nv* eigenvalues of the Hessenberg matrix *H* are returned in the parts of the *work* array corresponding to *ritzr* and *ritzi*, respectively.

1 **gen_arnoldi** applies exact shifts with respect to the current Hessenberg matrix *H*. Using exact shifts is equivalent to restarting the iteration from the beginning after updating the starting vector with a linear combination of Ritz vectors associated with the desired eigenvalues.

iparam(2) Specifies whether eigenvectors are to be computed, as follows:

iparam(2) ≤ 0 Compute only the eigenvalues.

iparam(2) > 0 Compute both eigenvalues and eigenvectors.

iparam(3) On input, specifies the maximum number of Arnoldi update iterations allowed. On return, is set to the actual number of Arnoldi update iterations performed.

iparam(4) Block size to be used in the recurrence. Must be set to 1 in the current release.

iparam(5) On return, specifies the number of converged eigenvalues, *nconv*.

src Real CM array of the same rank, shape, and layout as ***resid***. Contains the current operand vector *x*.

src1 Real CM array of the same rank, shape, and layout as ***resid***. Contains the vector *Bx* (used in shift-invert mode).

dst Real CM array of the same rank, shape, and layout as ***resid***. Contains the current result vector *y*.

ipntr One-dimensional front-end integer array of length 8. On return, contains pointers to mark the locations in *work* array for matrices and/or vectors used by the Arnoldi iteration.

ipntr(1) Reserved for internal use.

ipntr(2) Reserved for internal use.

ipntr(3) Reserved for internal use.

ipntr(4) Points to the next available location in *work* that is untouched by the program.

ipntr(5) Points to the starting location of the (*nv*+1) x *nv* upper Hessenberg matrix in *work*.

ipntr(6) Points to the starting location of the real part of the Ritz values array, *ritzr*, in *work*.

ipntr(7) Points to the starting location of the imaginary part of the Ritz values array, *ritzi*, in *work*.

ipntr(8) Points to the starting location of the error bounds array, *bounds*, in *work*.

w Real CM array with rank one greater than that of ***resid***. The first axis must have extent at least 3 and must be serial. The remaining axes must match the axes of ***resid*** in order of declaration, extents, and layout. This array is used internally.

work Real one-dimensional front-end array of length *lwork*. If *iparam*(2) is greater than 0, the eigenvectors of the final Hessenberg matrix (see *ipntr*(5)) are returned in the first k^2 locations of *work*, stored by columns. When the *j*th and (*j*+1)st Ritz values are the conjugate pair $\alpha+i\beta$ and $\alpha-i\beta$, the corresponding eigenvectors are $u+iv$ and $u-iv$, with u in the $(k+j)$th column and v in the $(k+j+1)$st column.

lwork Scalar integer variable. Supply the declared dimension of *work*. Must be at least $3nv^2 + 6nv$.

info Scalar integer variable. The input value affects the initial residual vector, as follows:

- If *info* = 0, ***resid*** is set to a random initial residual vector internally.
- If *info* is not 0, you must supply the initial residual vector in ***resid***.

On return, *info* contains one of the following error codes:

0 Normal exit.

1 All possible eigenvalues of the operator *L* have been found. *nconv* = *iparam*(5) is equal to the size of the invariant subspace spanning the operator *L*.

2 The eigenvectors are requested but there is not enough space in ***vec*** to carry out the computation because *nconv* > *k*. To obtain the eigenvectors, rerun with *k* equal to *nconv* = *iparam*(5).

-2 *k* must be positive.

-3 *nv* must be greater than *k* (or 2*k*, when eigenvectors are requested), and less than or equal to the size of the eigenproblem.

-4 The maximum number of Arnoldi update iterations must be greater than zero.

-5 *which* must be one of the following: 'LM', 'SM', 'LR', 'SR', 'LI', 'SI'.

-6 *type* must be 'I' or 'G'.

-7 The length of *work* is not sufficient.

-8 Error return from the LAPACK Hessenberg eigenvalue calculation.

-9 Starting vector is zero.

-9999 Maximum number of Arnoldi update iterations have occurred.

setup One-dimensional integer array of length 3. Internal variable. When you call **gen_arnoldi** or **deallocate_gen_arnoldi_setup**, supply the values returned by **gen_arnoldi_setup**.

ier Scalar integer variable. Set to 0 upon successful return. Upon return from **gen_arnoldi_setup**, may contain the following error codes:

-1 The first dimension of ***vec*** or ***w*** is not declared **:serial**.

-2 The serial dimension of ***vec*** or ***w*** has extent less than *nv* or 3, respectively.

-3 (rank ***vec***), (rank ***w***), and (rank ***resid*** + 1) are not equal.

-4 The sections of ***vec*** and ***w*** containing the vectors and indexed by the first dimension do not have the same shape as ***resid***.

DESCRIPTION

Intended Use. The **gen_arnoldi** routine solves the following eigenproblems:

- $Ax = \lambda x$, A symmetric, $L = A$, $B = I$.
- $Ax = \lambda Mx$, M symmetric positive definite, $L = M^{-1}A$, $B = M$.
- $Ax = \lambda Mx$, M symmetric semi-definite, $L = \mathrm{Re}\{(A-\sigma M)^{-1}M\}$, $B = M$ (shift-invert mode, in real arithmetic). If $Lx = \mu x$ and $\overline{\sigma}$ denotes the complex conjugate of σ, then $\mu = 1/2\ [\ 1/(\lambda - \sigma) + 1/(\lambda - \overline{\sigma})\ ]$.
- $Ax = \lambda Mx$, M symmetric semi-definite, $L = \mathrm{Im}\{(A-\sigma M)^{-1}M\}$, $B = M$ (shift-invert mode, in real arithmetic). If $Lx = \mu x$, then $\mu = 1/2i\ [\ 1/(\lambda - \sigma) - 1/(\lambda - \overline{\sigma})]$.

The third and fourth modes above provide the same enhancement for eigenvalues close to the (complex) shift σ. However, as λ goes to infinity, the operator L in the fourth mode dampens the eigenvalues more strongly than does L as defined in the third mode.

Setup and Deallocation. To use **gen_arnoldi**, follow these steps:

1. Call **gen_arnoldi_setup**.

   This routine generates three setup IDs and returns them in the array *setup* of length 3. You must supply this *setup* array in all subsequent **gen_arnoldi** and **deallocate_gen_arnoldi_setup** calls associated with this setup call.

2. Call **gen_arnoldi** iteratively, as described under **Reverse Communication Interface**, below.

   You can use the same *setup* array to solve more than one eigenproblem sequentially, as long as the array geometries are the same. You can also have more than one setup active at a time.

3. Call **deallocate_gen_arnoldi_setup**.

   This routine deallocates the memory associated with the three setup IDs.

Returned Eigenvalues and Eigenvectors. Upon successful final return,

- The real parts of the k desired eigenvalues are located in the first k locations of *ritzr*. The argument *ipntr*(6) points to the starting location of the *ritzr* array within *work*.

- The imaginary parts of the k desired eigenvalues are located in the first k locations of *ritzi*. The argument *ipntr*(7) points to the starting location of the *ritzi* array within *work*.

- If eigenvectors are requested (*iparam*(2) > 0), the corresponding eigenvectors are returned in *vec*(k+1:2k, :, ..., :). In the case of complex conjugate pairs, the eigenvalue with positive imaginary part is always first. Hence, if the jth and (j+1)st eigenvalues are the conjugate pair $\alpha+i\beta$ and $\alpha-i\beta$, then *ritzr*(j) = *ritzr*(j+1) = α, *ritzi*(j) = β, and *ritzi*(j+1) = $-\beta$. Corresponding eigenvectors are $u+iv$ and $u-iv$, with u = *vec*(k+j, :, ..., :) and v = *vec*(k+j+1, :, ..., :).

Reverse Communication Interface. The aim of the reverse communication interface is to isolate the matrix-vector operations from the k-step Arnoldi code. Such operations are performed by routines you supply, on data structures which are the most natural to the problem at hand. To this end, you must call **gen_arnoldi** iteratively. It returns control to the calling routine whenever the action of operators L or B on vectors is required. The reverse communication flag, *ido*, which must be 0 on input to the first call to **gen_arnoldi**, dictates which operator is to be applied. For standard eigenvalue problems, there is no distinction between *ido* = 1 and *ido* = -1. In both cases, the operation $y = Lx$ is required, where x and y are the source and destination vectors, ***src*** and ***dst***, respectively. For the generalized eigenvalue problem, the operation $y = Lx$ is always done in two steps, since L is a product of operators. The only difference between *ido* = 1 and *ido* = -1 is that when *ido* = 1, the product Bx is already available in ***src1*** and need not be computed, whereas it must be computed explicitly when *ido* = -1. The value *ido* = -1 is returned by **gen_arnoldi** at the first iteration to force the starting vector into the range of L (see reference 17 in Section 8.10). For generalized eigenproblems, **gen_arnoldi** also returns the value *ido* = 2, calling for the operation $y = Bx$ to be executed.

NOTES

Use of Array *w*. Do not use the CM array ***w*** as temporary workspace.

Data Layout. The CM arrays ***resid***, ***src***, ***src1***, ***dst***, ***vec***, and ***w*** must adhere to several constraints with regard to shape and layout. Arrays ***resid***, ***src***, ***src1***, and ***dst*** each contain a vector, while ***vec*** and ***w*** are collections of vectors. You may represent each vector with an array of arbitrary dimension, in the manner that is the most natural with respect to the matrix-vector operations. *The product of the axis extents of the arrays representing the vectors must be equal to the size of the eigenproblem.* Arrays ***resid***, ***src***, ***src1***, and ***dst*** must have the same shape and layout. Furthermore, ***vec*** and ***w*** must each have

an extra (instance) axis, which must be the first axis and must have extent at least *nv* in *vec* and at least 3 in *w*. This axis must be made local to a processing element so that the vectors, which have identical shape and layout, are "stacked up" in memory. This is accomplished by declaring the instance axis **:serial** in the calling program using a **CMF$LAYOUT** directive.

For example, in the one-dimensional case where the size of the eigenproblem is *n*, array declarations would be as follows:

```
      real vec(nv,n),w(3,n),resid(n),src(n),src1(n),dst(n)
CMF$LAYOUT vec(:serial,),w(:serial,),resid()
CMF$LAYOUT src(),src1(),dst()
```

In the two-dimensional case where the size of the problem is *n1* * *n2* = *n*, the array declarations would be

```
      real vec(nv,n1,n2),w(3,n1,n2),resid(n1,n2)
      real src(n1,n2),src1(n1.n2),dst(n1,n2)
CMF$LAYOUT vec(:serial,,),w(:serial,,),resid(,)
CMF$LAYOUT src(,),src1(,),dst(,)
```

On-Line Example. The on-line **gen_arnoldi** example is taken from the aeronautical industry. The so-called Tolosa matrix comes from the Aerospatiale Aircraft Division in Toulouse, France. It is part of the Harwell-Boeing collection (see reference 18 in Section 8.10). The eigenvalues with largest imaginary part are of interest to engineers. The matrix is very sparse with a block structure and is of order N=90+5k where k is an integer greater than 1. In the example, we choose k=782; hence, N=4000. The default of normality, which grows exponentially with N for this matrix, accounts for the discrepancy between the estimated bounds and the actual residuals (see reference 19). For the location of the on-line example, see below.

Acknowledgments. The **gen_arnoldi** routine is a CM Fortran adaptation for the CM of a Fortran77 code written by D. Sorensen and P. Vu at the Center for Research on Parallel Computation, Rice University (see reference 12 in Section 8.10). The portions of the code operating on front-end arrays make use of LAPACK (see reference 18 and BLAS routines which have been integrated so that **gen_arnoldi** is self-contained.

We thank S.Godet-Thobie at CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique) for providing us with the Tolosa matrix and the routines to build it, which are included in the on-line example.

EXAMPLES

Sample CM Fortran code that uses the **gen_arnoldi** routine can be found on-line in the subdirectory

```
eigen/arnoldi/cmf/
```

of a CMSSL examples directory whose location is site-specific.

8.10 References

For further information about reduction to tridiagonal form, see the following references:

1. Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 2d ed. Baltimore: Johns Hopkins University Press, 1989.

2. Martin, R. S., C. Reinsch, and J. H. Wilkinson. Householder Tridiagonalization of a Symmetric Matrix. Contribution II/2 in *Handbook for Automatic Computation: Linear Algebra*. Springer Verlag, 1971.

For information relevant to the **sym_tridiag_eigenvalues** routine, see

3. Barth, W., R. S. Martin, and J. H. Wilkinson. Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection. Contribution II/5 in *Handbook for Automatic Computation: Linear Algebra*. Springer Verlag, 1971.

For information relevant to **sym_tridiag_eigenvectors**, see

4. Peters, G. and J. H. Wilkinson. The calculation of specified eigenvectors by inverse iteration. Contribution II/18 in *Handbook for Automatic Computation: Linear Algebra*. Springer Verlag, 1971.

5. Demmel, J. Comments on inverse iteration on the CM-2. Private communication, September 1991.

6. Jessup, E. R. and I. C. F. Ipsen. Improving the accuracy of inverse iteration. *SIAM J. Sci. Stat. Comput.* **13** (1992): 550–72.

The following references provide information about the Jacobi rotations method:

7. Golub, G.H., and C.F. Van Loan. *Matrix Computations*, 2d ed. Baltimore: Johns Hopkins University Press, 1989. Pp. 444–59.

8. Modi, J. J. *Parallel Algorithms and Matrix Computation*. Oxford University Press, 1988. Pp. 154–95.

9. Mascarenhas, W. *On the Convergence of the Jacobi Method for Arbitrary Orderings*. Ph.D. dissertation, MIT, 1991.

10. Schroff, G., and R. Schreiber. On the Convergence of the Cyclic Jacobi Method for Parallel Block Orderings. *SIAM J. Matrix Anal. Appl.* **10** (1989).

11. Luk, F.T., and H. Park. A Proof of Convergence for Two Parallel Jacobi SVD Algorithms. *IEEE Transactions on Computers* **38**, no. 6 (1989): 806-11.

For information about the Lanczos algorithm described in this chapter, see the following references:

12. Sorensen, D. Implicit application of polynomial filters in a *k*-step Arnoldi method. *SIAM J. Matr. Anal. Apps.* **13** (1992): 357-85.

13. Ericsson, T. and A. Ruhe. The spectral transformation Lanczos method for the numerical solution of large sparse generalized symmetric eigenvalue problem. *Math. Comp.* **35** (1980): 1251-68.

14. Nour-Omid, B., B. N. Parlett, T. Ericsson, and P. S. Jensen. How to implement the spectral transformation. *Math. Comp.* **48** (1987): 663-73.

15. Parlett, B. N. and B. Nour-Omid. Towards a black box Lanczos program. *Comp. Phys. Com.* **53** (1989): 169-79.

16. Anderson, E., 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, *SIAM*.

For further information about the Arnoldi algorithm described in this chapter, see references 12, 14, and 16, as well as the following references:

17. Parlett, B. N. and Y. Saad. Complex Shift and Invert Strategies for Real Matrices. *Linear Algebra and its Applications*, **88/89** (1987): 575-95.

18. Duff, I., R. Grimes, and J. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection.* CERFACS Report TR-PA 9286.

19. Godet-Thobie, S. *Large Highly Non-Normal Eigenvalue Problems.* Ph.D. Thesis, Université de Paris-9-Dauphine, December 1992.

Index

Numbers

A

B

C

D

E

F

G

L

M

N

O

P

Q

R

S

T

U

V