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.

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)

Copyright © 1994 Thinking Machines Corporation

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)

sym_tred_gen_eigensystem Given a CM array containing one or more dense Hermitian 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)

Eigensystem analysis of dense real symmetric matrices:

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 ($BL = L^{T}B$). 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

Copyright © 1994 Thinking Machines Corporation

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. Each step of the iterative Arnoldi algorithm uses matrix vector operations to build a projected matrix on a subspace, and solves an eigenvalue problem on the subspace. In the current implementation, we do not exploit parallelism in the subspace computations (the numerical complexity of which scales as n^3 for **gen_arnoldi** and n^2 for **sym_lanczos**, where n is the dimension of the subspace and is supplied in the input parameter *nv*). The projected matrix is stored and processed on the partition manager, whereas the matrix vector operations are performed on the processing nodes. Best performance and scalability are expected when the operations performed on the processing nodes dominate the computation — that is, when the dimension of the subspace is much smaller than that of the original problem.

Communication with the **gen_arnoldi** routine occurs through a *reverse communication* interface. When the routine requires a matrix vector product, it returns a source vector; you must operate on the vector and supply the result in a subsequent **gen_arnoldi** call. No assumption is made regarding the nature or structure of the "matrix vector" operation, which could be, for example, any of the dense or sparse matrix vector operations provided by

Copyright © 1994 Thinking Machines Corporation

CMSSL (arbitrary elementwise sparse, arbitrary block sparse, grid sparse, etc.), or any other operation you provide. Therefore, you may represent the vectors with arrays of any shape, in the manner most natural for your application.

It is important to exploit whatever structure the problem may have when computing this matrix vector product. 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 ($iparam(2) \leq 0$) (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 sys-

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

tems 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

Copyright © 1994 Thinking Machines Corporation

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 Hermitian 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 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.

Copyright © 1994 Thinking Machines Corporation

8.2 Reduction to Tridiagonal Form and Corresponding Basis Transformation

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 15.

8.2.2 Numerical Stability

The routines described in this section are stable.

Copyright © 1994 Thinking Machines Corporation

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**.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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***.

Copyright © 1994 Thinking Machines Corporation

Upon completion of **sym_tred**, each A within $\boldsymbol{A}$ 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 $\boldsymbol{A}$ upon completion of **sym_tred**.

n Scalar integer variable. The number of rows and columns in each Hermitian matrix A within $\boldsymbol{A}$.

row_axis Scalar integer variable. The axis of $\boldsymbol{A}$ that counts the rows of each Hermitian matrix A. (For definitions of row and column axes, see Chapter 1.)

col_axis Scalar integer variable. The axis of $\boldsymbol{A}$ that counts the columns of each Hermitian matrix A. (For definitions of row and column axes, see Chapter 1.)

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 $\boldsymbol{B}$.

ier Scalar integer variable. Return code; set to 0 upon successful return. The following codes indicate errors:

−1 Length of axis *row_axis* of $\boldsymbol{A}$ is < n; must be $\geq n$.

−2 Length of axis *col_axis* of $\boldsymbol{A}$ is < n; must be $\geq n$.

Copyright © 1994 Thinking Machines Corporation

−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 ≤ *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.

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	$T = Q^{H}AQ$ (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.

Copyright © 1994 Thinking Machines Corporation

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 $\boldsymbol{A}$ to real symmetric tridiagonal form. Upon completion of **sym_tred**, each A within $\boldsymbol{A}$ 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 $\boldsymbol{d}$ and $\boldsymbol{e}$.

Basis Transformation. The **sym_to_tridiag** routine transforms the coordinates of the vectors in each B within $\boldsymbol{B}$ 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 $\boldsymbol{B}$ is overwritten with same vectors expressed in the coordinates of the new basis.

NOTES

Distinct Variables. The input CM arrays $\boldsymbol{A}$ and $\boldsymbol{B}$ must be distinct variables. The arrays $\boldsymbol{d}$ and $\boldsymbol{e}$ 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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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 online in the subdirectory

```
tred/cmf/
```

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

Copyright © 1994 Thinking Machines Corporation

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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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 you supply a non-positive *tolerance* value, the routine changes the value internally to *tolerance* $= \varepsilon \, ||T||$, 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

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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 with the same precision as ***d*** and ***e***. Absolute error tolerance for the computed eigenvalues. If you supply a non-positive value, *tolerance* is changed internally to the default value $\varepsilon\|\|T\|\|$, where ε is the machine precision and $\|\|T\|\|$ is the 1-norm of the tridiagonal matrix. In the multiple-instance case, the default tolerance is the smallest tolerance over all the instances. For the accurate extraction of tiny eigenvalues, you may need to set *tolerance* to a smaller value than the default. (For details, see Section 8.3.2.) If the routine uses the default tolerance, then *tolerance* is set to the default value on return. Because *tolerance* may be returned with a new value, it must be a variable, not a constant expression.

Copyright © 1994 Thinking Machines Corporation

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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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

Copyright © 1994 Thinking Machines Corporation

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), and this is the default to which *group* is set if you supply a non-positive value. The value of the grouping criterion, however, is 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)
```


Copyright © 1994 Thinking Machines Corporation

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 local dimension for best performance.

Copyright © 1994 Thinking Machines Corporation

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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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 with the same precision as ***d*** and ***e***. Singular values that differ by less than $group||T||\infty$ (where $||T||\infty$ is the infinity norm of the tridiagonal matrix) are grouped together, and their corresponding singular vectors are orthogonalized. If you supply a non-positive value, *group* is changed internally to the default value 10^{-4}. (For details, see Section 8.4.2.) If the routine uses the default grouping criterion, then *group* is set to the default value on return. Because *group* may be returned with a new value, it must be a variable, not a constant expression.

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 same as (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_**

Copyright © 1994 Thinking Machines Corporation

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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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).

Copyright © 1994 Thinking Machines Corporation

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* x *n*. The axes identified by *row_axis* and *col_axis* must have extent *n*.

Copyright © 1994 Thinking Machines Corporation

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. (For definitions of row and column axes, see Chapter 1.)

col_axis — Scalar integer variable. The axis of ***A*** that counts the columns of each Hermitian matrix A. (For definitions of row and column axes, see Chapter 1.)

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 with the same precision as ***d***. Controls the absolute accuracy of the eigenvalues after reduction to tridiagonal form. If you supply a non-positive value, *tolerance* is changed internally to the default value $\varepsilon||T||$, where ε is the machine precision and $||T||$ is the 1-norm of the tridiagonal matrix associated with A. In the multiple-instance case, the default tolerance is the smallest tolerance over all the instances. For the accurate extraction of tiny eigenvalues, you may need to set *tolerance* to a smaller value than the default. (For details, see Section 8.3.2). If the routine uses the default tolerance, then *tolerance* is set to the default value on return. Because *tolerance* may be returned with a new value, it must be a variable, not a constant expression.

Copyright © 1994 Thinking Machines Corporation

group Scalar real variable with the same precision as ***d***. Grouping criterion for eigenvalues after reduction to tridiagonal form. If you supply a non-positive value, *group* is changed internally to the default value 10^{-4}. (For details, see Section 8.4.2.) If the routine uses the default grouping criterion, then *group* is set to the default value on return. Because *group* may be returned with a new value, it must be a variable, not a constant expression.

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*** 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.
1	The rank of ***d*** is not the same as the rank of ***A***.
2	The rank of ***Q*** is not the same as 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

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$AQ = Q * \mathrm{DIAG}(d)$.

EXAMPLES

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

```
eigen/tred/cmf/
```

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

Copyright © 1994 Thinking Machines Corporation

8.6 Generalized Eigensystem Analysis of Dense Hermitian Matrices

Given a CM array ***A*** containing one or more dense Hermitian 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.

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 $B^{-1/2} = Q_B\ (\lambda_B)^{-1/2}\ Q_B{}^{\mathrm{H}}$.
3. Use **gen_matrix_mult** to compute the Hermitian 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 B is Hermitian, and 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.

Copyright © 1994 Thinking Machines Corporation

8.6.1 Accuracy

Like **sym_tred_eigensystem**, which is described in Section 8.5, **sym_tred_gen_eigensystem** first reduces each 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).

Copyright © 1994 Thinking Machines Corporation

Generalized Eigensystem Analysis of Dense Hermitian Matrices

Given a CM array ***A*** containing one or more dense Hermitian 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 CM array with the same rank, data type, 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.

Copyright © 1994 Thinking Machines Corporation

B CM array of the same rank, shape, data type, and precision as ***A***. For each Hermitian 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 or complex CM array of rank ≥ 2 containing one or more dense Hermitian 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 Hermitian matrix A within ***A***.

row_axis Scalar integer variable. The axis of ***A*** that counts the rows of each Hermitian matrix A. (For definitions of row and column axes, see Chapter 1.)

col_axis Scalar integer variable. The axis of ***A*** that counts the columns of each Hermitian matrix A. (For definitions of row and column axes, see Chapter 1.)

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.

Copyright © 1994 Thinking Machines Corporation

tolerance Scalar real variable with the same precision as ***d***. Controls the absolute accuracy of the eigenvalues after reduction to tridiagonal form. If you supply a non-positive value, *tolerance* is changed internally to the default value $\varepsilon||T||$, where ε is the machine precision and $||T||$ is the 1-norm of the tridiagonal matrix associated with A. In the multiple-instance case, the default tolerance is the smallest tolerance over all the instances. For the accurate extraction of tiny eigenvalues, you may need to set *tolerance* to a smaller value than the default. (For details, see Section 8.3.2). If the routine uses the default tolerance, then *tolerance* is set to the default value on return. Because *tolerance* may be returned with a new value, it must be a variable, not a constant expression.

group Scalar real variable with the same precision as ***d***. Grouping criterion for eigenvalues after reduction to tridiagonal form. If you supply a non-positive value, *group* is changed internally to the default value 10^{-4}. (For details, see Section 8.4.2.) If the routine uses the default grouping criterion, then *group* is set to the default value on return. Because *group* may be returned with a new value, it must be a variable, not a constant expression.

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 dense Hermitian 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

$AQ = BQ * \mathrm{DIAG}(d)$.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$$\text{angle_error}(i) \;<\; \frac{||A||_F * \max(p(n) * \textit{machine_epsilon}, \textit{tolerance})}{\text{gap}(i)}$$

where gap(i) is the absolute difference between eigenvalue(i) and the next nearest eigenvalue.

Copyright © 1994 Thinking Machines Corporation

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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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*.

Copyright © 1994 Thinking Machines Corporation

−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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- The eigenvectors for each matrix A within $\boldsymbol{A}$ are placed in the columns of the corresponding matrix within $\boldsymbol{Q}$. The eigenvectors are sorted to correspond to the order of the eigenvalues returned in $\boldsymbol{d}$. Thus, the eigenvector corresponding to the ith eigenvalue of a matrix in $\boldsymbol{d}$ is returned in column i of the corresponding matrix in $\boldsymbol{Q}$.

NOTES

Argument Values Modified. Since the values of $\boldsymbol{A}$, *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.

Copyright © 1994 Thinking Machines Corporation

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:

$$BL = L^{\mathrm{T}}B$$

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,

$$LV^{(k)} = V^{(k)}T^{(k)} + r^{(k)}e_k{}^{\mathrm{T}} \qquad (1)$$

where $V = [v_1, v_2, ..., v_k]$ has columns orthonormal with respect to B, T is a tridiagonal matrix of order k, and $r^{(k)}e_k{}^{\mathrm{T}}$ 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 fur-

Copyright © 1994 Thinking Machines Corporation

ther. The Rayleigh-Ritz procedure applied to the Lanczos 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 *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* = 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 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 loca-

Copyright © 1994 Thinking Machines Corporation

tion *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*+1:2*k*,...) 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 advan-

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

tage 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.

Copyright © 1994 Thinking Machines Corporation

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. These vectors can have any shape appropriate for the application (see, for instance, the on-line example; the pathname is provided at the end of the man page). 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:

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

```
      .....
      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

Copyright © 1994 Thinking Machines Corporation

```
      ...
      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
```


Copyright © 1994 Thinking Machines Corporation

```
      end if
      go to 10
```

8.8.6 Data Layout Guidelines

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

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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, $BL = L^{\mathrm{T}}B$. 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

Copyright © 1994 Thinking Machines Corporation

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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

'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,

Copyright © 1994 Thinking Machines Corporation

- ***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) $\leq$ 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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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'.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

−7 The length of *work* is not sufficient.

−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).

Copyright © 1994 Thinking Machines Corporation

- $Ax = \lambda Mx$, 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 $ido = 1$,

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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 ($BL = L^{\mathrm{T}}B$), 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,

$$LV^{(k)} = V^{(k)}H^{(k)} + r^{(k)}e_k{}^{\mathrm{T}} \qquad (1)$$

where $V = [v_1, v_2, ..., v_k]$ has columns orthonormal with respect to B, H is a Hessenberg matrix of order k, and $r^{(k)}e_k{}^{\mathrm{T}}$ 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

Copyright © 1994 Thinking Machines Corporation

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 $2k$, 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 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+1:2k,...) are the corresponding

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

eigenvectors. 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 $ritzr(j) = ritzr(j+1) = \alpha$, $ritzi(j) = \beta$, and $ritzi(j+1) = -\beta$. Corresponding eigenvectors are $u+iv$ and $u-iv$, with $u = \mathbf{vec}(k+j, :, ..., :)$ and $v = \mathbf{vec}(k+j+1, :, ..., :)$.

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. These vectors can have any shape appropriate for the application (see, for instance, the on-line example; the pathname is provided at the end of the man page). 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:

Copyright © 1994 Thinking Machines Corporation

```
      ....
      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)
```


Copyright © 1994 Thinking Machines Corporation

```
            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
```


Copyright © 1994 Thinking Machines Corporation

8.9.4 Data Layout Guidelines

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(,)
```

8.9.5 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

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

'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.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

iparam(2) Specifies whether eigenvectors are to be computed, as follows:

iparam(2) $\leq$ 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*.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

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 jth 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.

Copyright © 1994 Thinking Machines Corporation

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 $2k$, 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.

Copyright © 1994 Thinking Machines Corporation

−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.

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation

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.

Copyright © 1994 Thinking Machines Corporation