Parallel Eigensystem Solver

PeIGS Version 3.0, rev. 0.0

July 5, 1999

George Fann, David Elwood and Richard Littlefield
Pacific Northwest National Laboratory

%

ACKNOWLEDGMENT

This software and its documentation were produced with Government support under Contract Number DE-AC06-76RLO-1830 awarded by the United States Department of Energy. The Government retains a paid-up non-exclusive, irrevocable worldwide license to reproduce, prepare derivative works, perform publicly and display publicly by or for the Government, including the right to distribute to other Government contractors.

1 PeIGS: an overview

PeIGS is a collection of commonly used linear algebra subroutines for computing the eigensystem of the real standard symmetric eigensystem problem $\mathbf{Ax} = \lambda\mathbf{x}$ and the general symmetric eigensystem problem $\mathbf{Ax} = \lambda\mathbf{Bx}$ where $\mathbf{A}$ and $\mathbf{B}$ are dense and real matrices with $\mathbf{B}$ positive definite and λ is an eigenvalue corresponding to eigenvector $\mathbf{x}$. The intended computer architecture is a parallel message passing computer or a **homogeneous** network of workstations. The current code is written in C and Fortran 77 with message passing. Subroutine calls are supported in Fortran 77 and in C. PeIGS was designed for good performance in the regime n/p where n is the size of the matrix and p is the number of available processor.

% The PeIGS program is in the % public domain.

1.1 Problems that PeIGS can Solve

PeIGS can solve problems in linear algebra that are associated with the general symmetric and the standard symmetric eigensystem problems. PeIGS can also handle associated computations such as the Choleski factorization of positive definite matrices in packed storage format and linear matrix equations involving lower and upper triangular matrices in distributed packed row/column storage.

Dense and real matrices are assumed throughout in the computation. Only double precision is currently supported.

1.2 Computers for which PeIGS is Suitable

PeIGS is designed for parallel and distributed memory computers with a high startup time for communication. The IBM SP and the Cray T3s are examples of such parallel computers. Workstations linked together using network message passing software, such as Robert Harrison's *TCGMSG*[1], are also examples of such "parallel computers." PeIGS can be also be used on shared memory computers and on vector computers using the software TCGMSG. PeIGS can also also runs in the MPI environment. For the Cray shmem environment the user must link with the TCGMSG library.

The programming model used by PeIGS is single program multiple data (SPMD).

1.3 BLAS and Floating Point Computations

PeIGS has been written to take advantage of optimized floating point computations by the Basic Linear Algebra Subprograms(BLAS) library[2]. Extremely efficient machine specific BLAS are available for many computers. Such a machine specific BLAS library is necessary for achieving high performance for floating point computations.

[1] Harrison, R.J., "Portable tools and applications for parallel computers," Int. J. Quantum Chem., 40(1991), pp.847-863. TCGMSG can be obtained via anonymous ftp from **ftp.tcg.anl.gov** in the directory **/pub/tcgmsg**

[2] Dongarra J. et. al. A proposal for an Extended Set of Fortran Basic Linear Algebra Subprograms, NAG report TR3/86, NAG Limited, Oxford, 1986. The BLAS library is available from netlib at **netlib2.ornl.gov**

1.4 Algorithms

The numerical algorithms implemented are standard with the exception of a O(n^2) algorithm for finding orthogonal eigenvectors based on the theoretical and experimental work of Vince Fernando, Inderjit Dhillon and Beresford Parlett. Our initial variation implemented was the Berkeley algorithm described in a series of papers by Prof. Beresford Parlett of University of California, Berkeley, Inderjit Dhillon and George Fann[3].

%[4][5].

% In particular, inverse iteration to convergence is performed on all of % the eigenpairs in a cluster in a perfectly parallel fashion. % Orthogonalization is then performed in parallel using all the processors % storing the unconverged eigenvectors of the cluster. Exactly two % iterations of "inverse iteration followed by orthogonalization" are % always done. The key here is that two iterations of modified % Gram-Schmidt are used in the first orthogonalization. This algorithm % usually yields highly orthogonal eigenvectors, but this is not % guaranteed to always be the case since a convergence test is not % currently done.

One feature of PeIGS is that only processors holding some matrix elements participate in the computation. Other processors are free to work on other problems. The PeIGS data layout is a column/row wrapping according to a list of processors marking the processors that own the columns/rows. Most of the algorithms are based on panel-blocked computation strategies with the exception of the choleski factorization and the lower triangular inverse computations.

All of the standard algorithms coded are described in

Anderson, E. et. al. "LAPACK User's Guide,", SIAM, 1992.

Dahlquist, G. et. al.,"Numerical Methods," Prentice Hall, 1974.

Golub, G. and Charles Van Loan, "Matrix Computations,", Johns Hopkins, 1993.

Isaacson, E. and H.B. Keller, "Analysis of Numerical Methods," Wiley, 1966.

Parlett, B., "Symmetric Eigenvalue Problem," Prentice Hall, 1980.

Wilkinson, J.H., "Algebraic Eigenvalue Problem," Oxford, 1988.

1.5 Portability

This package is self-contained with the exception of the processor specific libraries for the BLAS and LAPACK. It is currently intended for cross development and cross compilation using Unix as the host environment and GNU utilites. There is also a Windows NT version available.

The communication calls are made through a set of code wrappers over the basic message passing provided by some of the distributed memory parallel computers and TCGMSG. For the distributed memory computers we do support native message passing provided by the

[3] Inderjit Dhillon, George Fann, Beresford Parlett, siam8th

[4] Fann, G. and % Littlefield, R., "Parallel Inverse Iteration with Reorthogonalization", % Proceeding of the 6th SIAM Conference on Parallel Processing for % Scientific Computing, SIAM, 1993

[5] Fann, G., R. Littlefield, and % D. Elwood, "Performance of a Fully Parallel Dense Real Symmetric Eigensolver % in Quantum Chemistry Applications", in High Performance Computing Symposium % 1995, Proceedings of the 1995 Simulation Multiconference, The Computer % Simulation Society

Intel iPSC_860, and Touchstone DELTA. We also support an interface to TCGMSG and to MPI.

The installation procedure creates a BLAS library and a LAPACK library. The user can replace these libraries with a more efficient optimized library or perhaps newer versions of the BLAS and LAPACK libraries. The user should be aware that there may be changes in the calling syntax of the latest LAPACK that may be incompatible with the LAPACK included here. The LAPACK codes that are included here are dated (Sept. 30, 1994). The subdirectory **/src/f77** contains a number of "fixed" codes originating from LAPACK. These "fixed" codes have the same LAPACK name with a number after their file name (as is the case with their subroutine name) to distinguish them from the original LAPACK codes. **The INSTALL file in the top level of the PeIGS directory supersede the following installation procedure**.

1.6 Support

We currently support the PeIGS library routines that are documented in the next chapter. All of the communication subroutines are supported in the context of providing communication calls used by the linear algebra subroutines. We support the package in the sense that reports of bugs, errors or poor performance will get our immediate attention. We will try to solve your problems and will provide bug fixes.

1.7 Thanks

Our thanks to all who helped to improve this work. In particular, we thank Robert Harrison and Eduardo Apra for assisting us in understanding chemistry problems and their associated spectra. We thank Professor Elizabeth Jessup of University of Colorado and Prof. Ken Bube of University of Washington for advice and strategies for PeIGS2. We thank Prof. Inderjit Dhillon of University of Texas, Austin and Professor Beresford Parlett of University of California, Berkeley for sharing with us their ideas and preliminary codes for the Berkeley algorithm in Prof. Parlett's preprints and Inderjit's thesis. Their work showed us how to reduce orthonormalization costs for large clusters of eigenvalues. All mistakes are our own.

We thank the Free Software Foundation for developing software tools that made our work easier.

We thank the Molecular Science Computing Facility at the Environmental Molecular Sciences Laboratory for providing access to the NWMPP1 and NWECS1. We thank NERSC for proving access to the CRAY-T3E. MSCF is funded by OBER and NERSC is funded by MICS.

The research support of the evolution of version 2 to version 3 of PeIGS was funded by DOE's MICS Grand Challenge in Computational Chemistry. The implementation and maintanence support was provided by EMSL. (PNNL's contract goes here).

2 Installation

This is a short synopsis of how to install PeIGS on a target system. **The user should consult the INSTALL file in the top level PeIGS directory for up to date information regarding the installation procedure.** We currently only support installation on Unix systems with the resources listed below for installing PeIGS. The library codes and the test programs are created using **Makefiles**. All machine specific data is contained in the **DEFS** file in the main PeIGS directory. Each subdirectory has a **Makefile** which "includes" the **DEFS** file. Because the **Makefiles** use **ifeq** conditionals, the GNU make version 3.68 or higher from the Free Software Foundation is required as the **make** utility. If this is not available to the installer then the installer needs to edit **DEFS** and the various **Makefiles** and comment out the **ifeq** and **endif** statements and leave the appropriate lines for the intended system.[1] The **DEFS** file currently has a list of workstations and parallel computers and their corresponding compilation switches. Most of the currently available distributed memory parallel computers are on this list: Cray T3D, IBM SP-1/2, Intel DELTA, Intel Paragon, SGI PowerChallange, Kendall Square Research KSR-2. **Homogeneous** networks of workstations are also support via TCGMSG or MPI.

Thus, the following software resources are needed to make the library:

1. ANSI-C compatible C compiler
2. Fortran 77 compiler
3. GNU make utility, version 3.68 and higher.

We assume that all Fortran callable subroutines have a trailing underscore at the end of their name in the name-space. Thus, we assume that C subroutines call Fortran subroutines using subroutine names with a trailing underscore added to the end of their name, e.g. **f77name_**. We also assume that a Fortran subroutine calls a C subroutine using a subroutine name with a trailing underscore added to the end of their name, e.g. **call cname_**. Users of IBM AIX RS6000 version 3 systems must add the **-qextname** or **-qEXTNAME** options to the **xlf** compiler or to the **f77** compiler. Users of HP9000 HP-UX version 9.0 or higher must add **+ppu** as an option to the HP **f77** compiler. Users of other dialects of Unix should consult their language manuals or system gurus for the appropriate options. A separate makefile has been added for the users of the IBM RS6000 without an installed GNU. The user is advised to read the INSTALL file and the DEFS files in the PeIGS directory.

In many of the calls to C subroutines from Fortran we assume that the C **double** is the same size as the Fortran **double precision**. This is not a serious obstacle since many Fortran compilers and C compilers have **8** byte switches for **real*8** computataions. Also, we assume that the address to double precision numbers is less than or equal to the size of a double precision number. In many of the Fortran to C interface codes we recast some of the double precision scratch spaces as pointers to double precision numbers. The user should also check whether the intended "C" and Fortran compilers are compatible in integer and integer pointers sizes. For example, on machines with 8 bytes for both **int** and **long int** it is important that the Fortran integer also be 8 bytes. This is a common problem when installing PeIGS.

[1] GNU make is available via anonymous ftp from **prep.ai.mit.edu:18.71.0.38** in the directory **/pub/gnu** as **make###.tar.gz**.

In h/blas_lapack.h we have tried to deal with some of the computers, for example, Cray T3D and the KSR2 where the single precision computation is performed in 64 bits of precision.

PeIGS uses machine specific arithmetic constants (obtained from LAPACK's **dlamach** and **slamch**) in a number of places. These constants are defined in h/blas_lapack.h.

You will probably obtain this library as a compressed tar file named **peigsXY.tar.Z**, where X.Y is the version number.

To build PeIGS on a Unix system using **csh**:

– type **make -v** to make sure that you are using the GNU make utility.

– uncompress the file **peigsXY.tar.Z** by typing **uncompress peigsXY.tar.Z** which will create a tar file called **peigsXY.tar**.

– untar the file peigsXY.tar by typing **tar xvf peigsXY.tar**. This will create a directory called **peigsX.Y** with a number of subdirectories and a file called **DEFS**.

– scan through the file **DEFS** using your favorite editor to see if your target parallel computer is supported. Make the appropriate changes, if necessary, for the compilers, directory paths, compiler options and library options. If you want to use MPI, rather than the default of TCGMSG or Intel NX, then follow the directions in **DEFS** about MPI use.

– save the corrected file **DEFS**.

– set the environment variable **TARGET**. For example, for the iPSC_860, the installer would type **setenv TARGET iPSC_860**.

– Check the file **PEIGSDIR/h/blas_lapack.h** to see if the machine parameters are defined for the target computer. If not, then the user must define his own parameters. To do this go to the **example** directory and type "make teslamach" to generate the executable code teslamch. Run teslamch on one node of the intended parallel machine. The code reports the machine arithmetic parameters that need to be inserted into **PEIGSDIR/h/blas_lapack.h**. On machines that use 8 bytes for single precision calculations, e.g. Cray T3D, the user must also insert **CODEOBJ= SINGLE** in the **DEFS** file so that **make** can build the single precision fortran codes in **PEIGSDIR/src/f77**. On other machines the user must set **CODEOBJ= DBLE**.

– type the command **make**. The libraries **libblas.a**, **liblapack.a** and **libpeigs.a** will now be built.

Compiler optimization can cause errors and unexpected results in certain situations. We handle this in the **Makefile**s by having the files that we know do not optimize well compile without optimization. The user is advised that much of our C utility codes fail to function with the GNU **gcc** compiler with the **-O2** optimization. Some of the utility codes also failed to function correctly with the Portland Group's **icc** compiler with the **-O3** optimization option. We are currently trying to remedy this situation. **The user is advised that the latest installation instruction is located in the INSTALL file at the top of the peigs directory.**

3 Content: Code Functionalities and Data Structure

This section lists the subroutines in PeIGS and their functionalities. PeIGS is designed using the bottom up approach. Each subroutine is built up on more basic routines. At the top level are drivers to solve the symmetric eigensystem problems. The second level contains factorization and linear algebra subroutines that are required by the eigensystem solvers. The last level contains internal PeIGS routines and utilites for managing communications, list processing, and error checking. In terms of data structure, the C routines are the most flexible since all the procedure arguments are pointers or one dimensional arrays of pointers. Thus, in many applications, minimal data movements are required to use these subroutines since no copying or data movements are needed. One only needs to set the address to the correct data positions.

This chapter is organized in terms of subroutine and data structure. Section 3.1 summarizes the routines that are supported in this version of PeIGS. Section 3.2. describes how the input data are expected in Fortran and in C. We assume that the reader is familiar with column distribution of data[1].

3.1 Summary

Let us summarize the subroutines and their functions. The following is a short list of the supported routines in PeIGS. These subroutines are protected from possible communication deadlock by enforcing the following rules. In the generalized eigenproblem the set of processors storing the matrix **A**, the matrix **B** and the eigenvectors **Z** must be the same set of processors. In the standard eigenproblem the set of processors storing the matrix **A** and the eigenvectors **Z** must be the same set of processors. This is not the same as the data distribution being the same. It means that the **set** of processors holding one matrix must be the same as the set of processors holding any other matrix.

If you are solving the real symmetric eigensystem problem $\mathbf{Ax} = \lambda \mathbf{x}$ the processors holding the columns of matrix **A**, and the processors holding the eigenvectors **X** are required to be the same set or the processors holding eigenvectors of **X** must be a subset of those holding columns of **A**. This mild restriction is currently necessary to avoid possible race conditions. Using such input restriction and no buffering assumptions PeIGS can be used on strong message typing message passing software (e.g. TCGMSG) – with no message queues or buffers. The user can, of course, use other routines in PeIGS but at their own risk. Some of the unsupported and internal PeIGS routines, with certain data distributions and insufficient message buffer space, may encounter communication deadlock. In this release, we assume that all the matrices **A** and **B** for the general eigensystem problems and the standard symmetric eigensystem problems with matrix of eigenvectors **Z** share the same set of processors for their data distributions.[2]

Level 1 drivers:

- PDSPGV : General real symmetric eigensystem solver
- PDSPEV: Standard real symmetric eigensystem solver

[1] Golub, Gene H. and Charles F. Van Loan, "Matrix Computations," 2nd ed, Johns Hopkins University Press, 1989.

[2] PeIGS no longer support the computation of selected eigenvalue and eigenvector pairs.

- PDSPTRI: Symmetric tri-diagonal eigensystem solver

Level 2 drivers:

- CHOLESKI: submatrix Choleski factorization[3].
- INVERSEL: the inverse of a lower triangular matrix in column distributed format.
- MXM*: a variety of matrix multiplication routines for matrices with different shapes and column/row distributions.
- TRED2: column distributed Householder reduction of a symmetric matrix to tri-diagonal form (only the C version).
- PSTEBZ: a parallel version of LAPACK's DSTEBZ routine for computing the eigenvalues of real tridiagonal matrices using bisection.
- PSTEIN*: a parallel subspace iteration method for finding eigenvectors of symmetric tridiagonal matrices using the our parallel variant of the Dhillon-Parlett algorithm. Computations involving clusters are dynamically scheduled based on spectrum.
- TRESID: a parallel routine for computing the residual of the tridiagonal eigenproblem.
- RESID: a parallel routine for computing the residual of the standard eigenproblem.
- RESIDUAL: a parallel routine for computing the residual of the generalized eigenproblem.
- ORTHO: a parallel routine for testing the orthogonality of eigenvectors
- BORTHO: a parallel routine for testing the B-orthogonality of eigenvectors for the generalized eigenproblem

3.2 Data Layout

From the user's point of view PeIGS' data structure is essentially column(row) wrapping according to a list. By this we mean that a matrix is partitioned into columns (rows) with an integer array **map** describing which processor holds column (row) **i**. For example, **map(i) = p** means that processor **p** stores column (row) **i**. When using MPI this definition is changed slightly. See the subsection on using MPI for more details on this case.

The data layout is used in a compatible way in the PeIGS subroutines. The data structure for each routine can also be used as input to another PeIGS subroutine without remapping. Of course, for optimal performance the user is advised to minimize communication.

The follow subsections describes how to input data from a given processor's point of view. We hope that the examples are informative.

3.2.1 Fortran Data Structure

Let us show how the following 4 by 4 matrix can be **row mapped** on 2 processors (processor 0 and processor 1) in Fortran using our structure. Let **mapA** be the integer array describing processor numbers for the distribution of the matrix.

[3] Geist, G.A. and M.T. Heath, "Matrix Factorization on a Hypercube Multiprocessor", in Hypercube Multiprocessors 1986, SIAM, Philadelphia, 1986, pp. 161-180

```
1   2   3   4
5   6   7   8
9   10  11  12
13  14  15  16
```

If we put row 1 and row 4 on processor 0 and row 2 and row 3 on processor 1, the contents of the array **mapA** on both processors is

mapA : 0 1 1 0

The content of the one dimensional storage array for **matrixA** on processor 0 is

proc 0: matrixA: 1 2 3 4 13 14 15 16

The content of the one dimensional storage array for **matrixA** on processor 1 is

proc 1: matrixA: 5 6 7 8 9 10 11 12

This is the array storage that is expected as input by the Fortran drivers.

For symmetric matrices , we use the packed storage format. Here is an example of a symmetric matrix **A** column decomposed and stored using its lower triangular part.

```
1
5   6
9   10  11
13  14  15  16
```

If we put columns 1 and 4 on processor 0 and columns 2 and 3 on processor 1, the contents of the array **mapA** on both processors are

mapA : 0 1 1 0

The content of the one dimensional storage array for **matrixA** on processor 0 is

proc 0: matrixA: 1 5 9 13 16

The content of the one dimensional storage array for **matrixA** on processor 1 is

proc 1: matrixA: 6 10 14 11 15

This is the array storage format and the mapping format that is expected as input for the symmetric matrices **A** and **B** in the the symmetric eigensystem solvers **PDSPEV**, **PDSPGV**, **PDSPEVX** and **PDSPGVX**.

3.2.2 C Data Structure

The reader can disregard this section if Fortran is the language of choice. Since the C user can call Fortran or C subroutines the information in the last section may be useful to the C programmer.

All of the input parameters for C subroutines in PeIGS are pointers. For the matrix **A** above, the input for the C subroutine requires that the matrix be presented as a one dimenesional array of pointers **ptr**. Each pointer **ptr**[**i**] points to the storage address of the first element of the i-th column owned by the processor. An example is given below.

Again consider the following matrix.

```
1   2   3   4
5   6   7   8
9   10  11  12
13  14  15  16
```

If we distribute columns 1 and 4 on processor 0 and columns 2 and 3 on processor 1, the contents of the array **mapA** on both processors are

$$\mathbf{mapA} : 0\ 1\ 1\ 0$$

Let us use the notation **mem-address(i)** to denote the memory location of the first element in column i. **We assume that the data for column i are stored in consecutive memory locations.**

The content of the one dimensional array of pointers **ptr** on processor 0 is

$$\mathbf{proc0} : \mathbf{ptr} : \mathbf{mem} - \mathbf{addr}(\mathbf{1})\ \ \mathbf{mem} - \mathbf{addr}(\mathbf{4})$$

and, the content of the one dimensional array of pointers **ptr** on processor 1 is is

$$\mathbf{proc1} : \mathbf{ptr} : \mathbf{mem} - \mathbf{addr}(\mathbf{2})\ \ \mathbf{mem} - \mathbf{addr}(\mathbf{3})$$

Notice that **ptr[0]** stores the starting address of column 1 and **ptr[1]** stores the starting address of column 4. This is because column 4 is the second column of **A** that processor 0 owns.

If we regard **ptr[0]** and **ptr[1]** as a C array then the contents of these arrays on processor 0 will be

$$\mathbf{proc0} : \mathbf{ptr}[\mathbf{0}] : 1\ 5\ 9\ 13$$

$$\mathbf{proc0} : \mathbf{ptr}[\mathbf{1}] : 4\ 8\ 12\ 16$$

For symmetric matrices , we use the packed storage format. Here is an example of the input data for a column distributed symmetric matrix stored using only its lower triangular part.

```
1
5   6
9  10   11
13 14   15   16
```

If we put columns 1 and 4 on processor 0 and columns 2 and 3 on processor 1, the contents of the array **mapA** on both processors are

$$\mathbf{mapA} : 0\ 1\ 1\ 0$$

The content of the one dimensional pointer array **ptr** array for **matrixA** on processor 0 is

$$\mathbf{proc0} : \mathbf{ptr} : \mathbf{mem} - \mathbf{addr}(\mathbf{1})\ \ \mathbf{mem} - \mathbf{addr}(\mathbf{16})$$

The content of the one-dimensional pointer array **ptr** on processor 1 is

$$\mathbf{proc1} : \mathbf{ptr} : \mathbf{mem} - \mathbf{addr}(\mathbf{6})\ \ \mathbf{mem} - \mathbf{addr}(\mathbf{11})$$

This is the content of the one-dimensional array of pointers that is expected as input by the C subroutines. In particular, the matrix **A** and **B** in the general symmetric and regular symmetric eigensolvers (**PDSPEV** and **PDSPGV**) are expected in this format as input.[4]

[4] **Note:** In the header files of our code we use the term "double pointer." This term is confusing and the user should substitute this with a " 1-D array of pointers to double precision numbers."

3.2.3 MPI Use

The reader can disregard this section if they are using TCGMSG or Intel NX for communication.

The PeIGS implementation of MPI does NOT support **heterogeneous** networks of computers, but DOES support **homogeneous** networks of computers. In particular, MPI never knows the real data type of the message, a generic MPI_BYTE type is used by PeIGS, hence MPI cannot correctly deal with different binary representations of data on different computers.

PeIGS does not use MPI communicators explicitly. Instead, to use MPI one simply changes the definition of the **map** arrays slightly. In particular, when using MPI **map(i) = p** means that column (row) **i** is stored on the processor which has **rank p** in the **MPI_Comm_World** communicator. Everything else is identical to the non-MPI case. The processors owning a matrix may be part of a communicator other than **MPI_Comm_World**, but the ranks stored in the **map** arrays MUST be the ranks of the processors in the **MPI_Comm_World** communicator. The PeIGS routines then do all communication using the **MPI_Comm_World** communicator, but restricting communication to only those processors listed in one of the **map** arrays passed to the PeIGS routines.

4 Calling Syntax

4.1 Level I Subroutines

The PeIGS package has two main routines for the solution of dense, real symmetric eigensolvers: **PDSPGV** and **PDSPEV** for the generalized and standard eigenproblems. All of these routines can be called from either fortran or C and do extensive checking of input data. See section 6 for more about error handling.

The reduction algorithm, from the general symmetric eigensystem problem to the standard eigensystem used in this section, is to invert the Choleski factor **L** of **B** and perform panel blocked matrix conjugation to arrive at $\mathbf{L^{-1}AL^{-t}}$.

Since the scratch arrays are dependent on the distribution of matrix entries across the processors PeIGS provides a subroutine, FMEMREQ in Fortran and memreq_ in C, which returns the dimensions of the integer, double precision, and pointer arrays that are necessary for the computation. The solver **PDSPEV**, **PDSPEVX**, **PDSPTRI**, **PDSPGV**, and **PDSPGVX** are currently supported.[1]

The requested eigensystem is returned to the user in the following manner. Let's say that the dimension of the matrix is **n** and that the user allocates an array **eval** for the eigenvalues, an array **matrixZ** for the eigenvector and an array **mapZ** for the distribution of the eigenvectors across the processors. PeIGS returns all the requested eigenvalues to all the processors in the list **mapZ**. PeIGS also returns the list **mapZ**, very likely to be different than the input list, which informs the calling routine which processor stores which eigenvector. Suppose that the i-th eigenvector of A is the j-th eigenvector of A stored on the current processor. Then, the eigenpair for the i-th eigenvalue is given by **(eval(i), (matrixZ(j*n+1), matrixZ(j*n+2), ..., matrixZ((j+1)*n)))**.

The C user can replace **matrixZ** above as a one-dimensional array of pointers to **double**, say **vecZ**. Then the i-th eigenpair (in C indexing) is returned as **(eval[i], vecZ[j])** where the i-th eigenvector of **A** is again the j-th eigenvector that is stored on this processor.

PeIGS provides a utility FIL_MAPVEC (F77) and fil_mapvec_(C) which returns the i and j correspondence between the global matrix index i and the local matrix storage index j.

Here's an example of how **FMEMREQ** and **PDSPEV** can be called in Fortran to solve the standard symmetric eigensystem problem $\mathbf{Ax} = \lambda\mathbf{x}$.

Let **mapA** and **mapZ** be integer arrays of dimension **n** which contain the data distribution information. Let **isize** be the parameter holding the integer scratch array size. Let **rsize** be the parameter holding the double precision scratch array size. Let **ptr_size** be the parameter holding the size of the 1 dimensional, double precision array which **PDSPEV** will convert to a scratch array of pointers to double precision numbers. For this example, **mapB** can be empty or just a dummy variable. An integer array **iscratch(1:3*n)** is used for integer scratch space. On output, **FMEMREQ** returns the dimensions of the scratch memory necessary for the calculations in the variables **isize**, **rsize**, **ptr_size**. The parameter **itype = 0** indicates that

[1] We are currently working on reducing the memory requirements for the selected eigensystem problem(see section 7 Bug 1).

PDSPGV or **PDSPGVX** is to be called, **itype** = 1 indicates that **PDSPEV** or **PDSPEVX** is to be called, and **itype** = 2 indicates that **PDSPTRI** is to be called.

The eigenvalues are returned in an array **eval** on all the participating processors (those in **mapA** or **mapZ**) with the eigenvectors returned using a one-dimensional array in the same manner as the input matrices except that it is not in a symmetric format. It is important to note that **mapZ** on exit from the various PeIGS routines will generally be different than it was on entry (i.e., the output array will be a permutation of the input array).

```
      integer isize, rsize, ptrsize, itype, info
      integer iscratch(n)
      integer iscrat(*)
      integer mapA(*), mapZ(*)
      double precision matrixA(*), matrixZ(*), eval(*)
      double precision dblptr(*)
c
c see note on portability
c
      itype = 1
      call memreq( itype, n, mapA, mapB, mapZ, isize, rsize,
     $      ptr_size, iscratch)
c
c make sure enough memory was allocated for the scratch space;
c at least as much as requested in isize, rsize, and ptr_size
c
      call pdspev( n, matrixA, mapA, matrixZ, mapZ, eval, iscrat,
     1  isize, dblptr, ptr_size, scratch, rsize, info )
c
c
c
```

Here's an example of how **PDSPGV** can be called in C. Assume that **mapA[0:n-1]**, **mapB[0:n-1]** and **mapZ[0:n-1]** have been allocated and assigned values.

```
      int itype, indx, info;
      int n, isize, rsize, ptr_size;
      int *mapA, *mapZ, *mapB;
      int *iscratch;

      double **matrixA, **matrixB, **matrixZ;
      double *iptr, *scratch;

      extern void memreq_(), pdspgv();

...
      /*
        input matrixA, matrixB, matrixZ in the manner cited
        in the C input section
      */
...
```

```
/*
        allocate scratch space for the memory routine
*/

   if ((itemp = (int *) malloc( n * sizeof(int))) == NULL ) {
        fprintf(stderr, " ERROR in memory allocation,
               not enough memory for integer scratch space for memreq \n");
       exit(-1);
        }


/*
   itype = 0 specifies memory for the general symmetric eigensystem problem
*/

   itype = 0;
   memreq_ (&itype, &n, mapA, mapB, mapZ, &isize, &rsize, &ptr_size, itemp);
   free(itemp);

   if ( (iscratch = (int *) malloc( isize * sizeof(int))) == NULL ){
      fprintf(stderr, " ERROR in memory allocation,
              not enough memory for integer scratch space \n");
      exit(-1);
   }


   if ( (scratch = (double *) malloc( rsize * sizeof(double))) == NULL ){
      fprintf(stderr, " ERROR in memory allocation,
              not enough memory for double scratch space \n");
      exit(-1);
   }

  if ( (iptr = (double **) malloc( ptr_size * sizeof(double *))) == NULL ){
      fprintf(stderr, " ERROR in memory allocation,
              not enough memory for pointer scratch space \n");
    exit(-1);
  }

/*
  ifact = 1 specifies that matrix B contains B and should be Choleski factored
*/
    ifact = 1;
    pdspgv (&ifact, &n, matrixA, mapA, matrixB, mapB, matrixZ, mapZ, eval,
                 iscratch, &isize, iptr, &ptr_size ,scratch, &rsize, &info);

...
```

4.2 PDSPEV: Real Symmetric Eigensystem Solver

PDSPEV solves the standard real symmetric eigensystem problem: Given a real symmetric matrix **A**, find all its eigen-pairs $(\lambda, \mathbf{x})$ such that $\mathbf{Ax} = \lambda \mathbf{x}$.

In Fortran 77 the calling sequences for PDSPEV and PDSPEVX are:

```
         subroutine pdspev( n, vecA, mapA, vecZ, mapZ, eval, iscratch, iscsize,
      $                    dblptr, ibuffsize, scratch, ssize, info)

        integer     n, mapA, mapZ, iscratch, iscsize, ibuffsize, ssize, info

        double precision  eval(*), scratch(*), vecA(*), vecZ(*), dblptr(*)
c
c The input arguments are the same as in the C code
c pdspev except that one changes the arguments of type
c       "1-D array of pointers to DoublePrecision", i.e.,
c vecA, vecZ and dblptr, to type
c "1-D array of double precision numbers" and store the data in
c       these arrays using the packed storage format described previously.
c       Also, when calling from fortran use FMEMREQ, rather than memreq_,
c       to get the required size of workspace arrays.
c       Finally, the C code describes arrays of length n with indices
c       of 0 to n-1, for fortran usage the indices should be 1 to n.
c
         subroutine pdspevx( ivector, irange, n, vecA, mapA,
      $   lb, ub, ilb, iub, abstol,
      $   meigval, vecZ, mapZ, eval, iscratch, iscsize,
      $   dblptr, ibuffsize, scratch, ssize, info)

        integer     ivector, irange, n, mapA, ilb, iub, meigval,
      $             mapZ, iscratch, iscsize, ibuffsize,
      $             ssize, info

        double precision        lb, ub, abstol, eval(*), scratch(*)
      $                         vecA(*), vecZ(*), dblptr(*)

c
c The input arguments are the same as in the C code
c pdspevx except that one changes the arguments of type
c       "1-D array of pointers to DoublePrecision", i.e.,
c vecA, vecZ and dblptr, to type
c "1-D array of double precision numbers" and store the data in
c       these arrays using the packed storage format described previously.
c       Also, when calling from fortran use FMEMREQ, rather than memreq_,
c       to get the required size of workspace arrays.
c       Finally, the C code describes arrays of length n with indices
c       of 0 to n-1, for fortran usage the indices should be 1 to n.
c
         subroutine pdspevx( ivector, irange, n, vecA, mapA,
     $   lb, ub, ilb, iub, abstol,
     $   meigval, vecZ, mapZ, eval, iscratch, iscsize,
     $   dblptr, ibuffsize, scratch, ssize, info)

       integer      ivector, irange, n, mapA, ilb, iub, meigval,
     $              mapZ, iscratch, iscsize, ibuffsize,
     $              ssize, info

       double precision        lb, ub, abstol, eval(*), scratch(*)
     $                         vecA(*), vecZ(*), dblptr(*)

c
c The input arguments are the same as in the C code
c pdspevx except that one changes the arguments of type
c
```

The calling sequence for PDSPEV in C is:

```
void pdspev(n, vecA, mapA, vecZ, mapZ, eval, iscratch, iscsize,
            dblptr, ibuffsize, scratch, ssize, info)

     Integer            *n, *mapA, *mapZ, *iscratch, *iscsize,
                        *ibuffsize, *ssize, *info;
     DoublePrecision   **vecA, **vecZ, *eval, **dblptr, *scratch;

/*

 *  Our parallel version of LAPACK's dspev.


 *  Purpose
 *  =======

 *  pdspev computes all of the eigenvalues and eigenvectors of a
 *  real symmetric eigenproblem, of the form

 *  A*x=(lambda) * x.

 *  Here A is assumed to be symmetric.
 *  A uses packed storage.

 *  Arguments
 *  =========

 * The current code assumes mapA and mapZ each contain
 * exactly the same set of processor id's, though not necessarily
 * in the same order.

 * All arguments are POINTERS to date of the specified type unless
 * otherwise mentioned.  In particular, INTEGER = (Integer *) and
 * DOUBLE PRECISION = (DoublePrecision *)


 * In the following let:
 *    n      = dimension of matrix A
 *    me     = this processors id (= mxmynd_())
 *    nprocs = number of allocated processors ( = mxnprc_())
 *    nvecsA = number of entries in mapA equal to me
 *             (= count_list( me, mapA, n ))
 *    nvecsZ = number of entries in mapZ equal to me
 *             (= count_list( me, mapZ, n ))

 *-----------------------------------------------------------------

 *  n       (input) INTEGER
 *          The number of rows and columns of the matrices A and B.
 *          N >= 0.

 *  vecA    (input/workspace) array of pointers to DoublePrecision
 *                            = (DoublePrecision **) dimension ( nvecsA )
 *          On entry, vecA[i], i = 0 to nvecsA-1, points to an
 *          array containing the lower triangular part of the i-th
 *          column of A which is owned by this processor.  The
 *          columns of A owned by this processer are determined by mapA
 *          (See below).

 *          On exit, the contents of matrixA are destroyed.

 *  mapA    (input) INTEGER array, dimension (N)
 *          mapA(i) = the id of the processor which owns column i
 *                    of the A matrix, i = 0 to n-1.

 *  vecZ    (output) array of pointers to DoublePrecision (DoublePreci-
sion **)
 *                   dimension ( nvecsZ )
 *          On entry, vecZ[i], i = 0 to nvecsZ-1, should point to an
 *          array of length n.

 *          On exit:

 *            vecZ[i], i = 0 to nvecsZ-1, points to the i-th eigenvector
 *            (as determined by the exit values in mapZ) owned by this
 *            processor.

 *            The eigenvectors are normalized such that: Z'*Z = I.

 *  mapZ    (input/output) INTEGER array, dimension (N)
 *          On entry:

 *          mapZ(i) = the id of a processor which has room for the
 *                    i-th eigenvector, i = 0 to n-1.
 *          On exit:
 *            mapZ(i) = the id of the processor which actually owns the i-
th
 *                      eigenvector, i = 0 to n-1.

 *  eval    (output) DOUBLE PRECISION array, dimension (N)
 *          If INFO = 0, the eigenvalues  of the matrix
 *          in no particular order.

 *  iscratch (workspace) INTEGER array, dimension
 *           Must be >= that returned from utility routine memreq_

 *  iscsize  (input) INTEGER
 *           The number of usable elements in array "iscratch".
*           Must be >= that returned from utility routine memreq_

*  dblptr   (workspace) DoublePrecision pointer to
*           DoublePrecision (DoublePrecision **),
*           dblprt[0] must point to the start of an array consising of
*           pointers to DoublePrecision (DoublePrecision *).
*           Must be >= that returned from utility routine memreq_

*  ibuffsize(input) INTEGER
*           The number of usable elements in array "dblptr".
*           Must be >= that returned from utility routine memreq_

*  scratch  (workspace) DOUBLE PRECISION array, dimension
*           Must be >= that returned from utility routine memreq_

*  ssize    (input) INTEGER
*           The number of usable elements in array "scratch".
*           Must be >= that returned in utility routine memreq_

*  INFO    (output) INTEGER

*          = 0:  successful exit.

*          -50 <= INFO < 0, the -INFOth argument had an illegal value.

*          INFO = -51,      Input data which must be the same on all
*                           processors is NOT the same on all processors
*                           in mapZ.

*          = 1,             Error computing the eigenvalues of the
*                           tridiagonal eigenproblem.  In this case
*                           'pstebz_' returned a non-zero info, whose
*                           value was printed to stderr.

*          = 2,             Error computing the eigenvectors of the
*                           tridiagonal eigenproblem.  In this case
*                           'pstein' returned a non-zero info, whose
*                           value was printed to stderr.

*          Any processor with a negative INFO stops program execution.

*          If -50 <= INFO < 0, then bad data was passed to this routine
*                              and no attempt is made to make sure that
*                              INFO is the same on all processors.

*          All other INFO      INFO should be the same on all processors in
*                              mapA,Z.  If this routine calls a routine
*                              which returns a negative info, then info
*                              may not be the same on all processors.
*                              This, however, should never happen.
*---------------------------------------------------------------------

*/
```

The calling sequence for PDSPEVX in C is:

```
void pdspevx ( ivector, irange, n, vecA, mapA, lb, ub, ilb, iub, abstol,
      meigval, vecZ, mapZ, eval, iscratch, iscsize,
      dblptr, ibuffsize, scratch, ssize, info)
     Integer          *ivector, *irange, *n, *mapA, *ilb, *iub, *meigval;
     Integer          *mapZ, *iscratch, *iscsize, *ibuffsize, *ssize, *info;
     DoublePrecision  *lb, *ub, *abstol, *eval, *scratch;
     DoublePrecision  **vecA, **vecZ, **dblptr;

/*

 *  Our parallel version of LAPACK's dspevx.


 *  Purpose
 *  =======

 *  pdspevx computes some or all of the eigenvalues and, optionally,
 *  eigenvectors of a real symmetric eigenproblem, of the form

 *  A*x=(lambda) * x.

 *  Here A is assumed to be symmetric.
 *  A uses packed storage.

 *  Arguments
 *  =========

 * The current code assumes mapA and mapZ each contain
 * exactly the same set of processor id's, though not necessarily
 * in the same order.

 * All arguments are POINTERS to date of the specified type unless
 * otherwise mentioned.  In particular, INTEGER = (Integer *) and
 * DOUBLE PRECISION = (DoublePrecision *)


 * In the following let:
 *    n      = dimension of matrix A
 *    me     = this processors id (= mxmynd_())
 *    nprocs = number of allocated processors ( = mxnprc_())
 *    nvecsA = number of entries in mapA equal to me
 *             (= count_list( me, mapA, n ))
 *    nvecsZ = number of entries in mapZ equal to me
 *             (= count_list( me, mapZ, n ))

 *-----------------------------------------------------------------
 *  ivector (input) INTEGER
 *          = 0:    Compute eigenvalues only;
 *          = 1:    Compute eigenvalues and eigenvectors.

 *  irange  (input) INTEGER
 *          = 1:   all eigenvalues will be found;
 *          = 2:   all eigenvalues in the half-open interval (lb, ub]
 *                 will be found;
 *          = 3:   the ilb-th through iub-th eigenvalues will be found.


 *  n       (input) INTEGER
 *          The number of rows and columns of the matrices A and B.
 *          N >= 0.

 *  vecA    (input/workspace) array of pointers to DoublePrecision (Dou-
blePrecision **)
 *                            dimension ( nvecsA )
 *          On entry, vecA[i], i = 0 to nvecsA-1, points to an
 *          array containing the lower triangular part of the i-th
 *          column of A which is owned by this processor.  The
 *          columns of A owned by this processer are determined by mapA
 *          (See below).

 *          On exit, the contents of matrixA are destroyed.

 *  mapA    (input) INTEGER array, dimension (N)
 *          mapA(i) = the id of the processor which owns column i
 *                    of the A matrix, i = 0 to n-1.

 *  lb      (input) DOUBLE PRECISION
 *          If IRANGE=2,  the lower bound of the interval to be searched
 *          for eigenvalues.  Not referenced if IRANGE = 1 or 3, but
 *          must not be a pointer to NULL.

 *  ub      (input) DOUBLE PRECISION
 *          If IRANGE=2,  the upper bound of the interval to be searched
 *          for eigenvalues.  Not referenced if IRANGE = 1 or 3, but
 *          must not be a pointer to NULL.

 *  ilb     (input) INTEGER
 *          If IRANGE=3,  the index (from smallest to largest) of the
 *          smallest eigenvalue to be returned.  ilb >= 1.
 *          Not referenced if IRANGE = 1 or 2, but
 *          must not be a pointer to NULL.

 *  iub     (input) INTEGER
 *          If RANGE=3, the index (from smallest to largest) of the
 *          largest eigenvalue to be returned.  min(ilb,N) <= iub <= N.
 *          Not referenced if IRANGE = 1 or 2, but
 *          must not be a pointer to NULL.
 *  abstol  (input) DOUBLE PRECISION
 *          The absolute error tolerance for the eigenvalues.
 *          An approximate eigenvalue is accepted as converged
 *          when it is determined to lie in an interval [a,b]
 *          of width less than or equal to

 *                  ABSTOL + EPS *   max( |a|,|b| ) ,

 *          where EPS is the machine precision.  If ABSTOL is less than
 *          or equal to zero, then  EPS*|T|  will be used in its place,
 *          where |T| is the 1-norm of the tridiagonal matrix obtained
 *          by reducing matrix A to tridiagonal form.

 *          See "Computing Small Singular Values of Bidiagonal Matrices
 *          with Guaranteed High Relative Accuracy," by Demmel and
 *          Kahan, LAPACK Working Note #3.

 *  meigval (output) INTEGER
 *          The total number of eigenvalues found.  0 <= meigval <= N.
 *          If IRANGE = 1,   M = N, and if RANGE = 3,   M = IUB-ILB+1.

 *  vecZ    (output) array of pointers to DoublePrecision (DoublePreci-
sion **)
 *                  dimension ( nvecsZ )
 *          On entry, vecZ[i], i = 0 to nvecsZ-1, should point to an
 *          array of length n.

 *          On exit:

 *            vecZ[i], i = 0 to nvecsZ-1, points to the i-th eigenvector
 *            (as determined by the exit values in mapZ) owned by this
 *            processor.

 *            The eigenvectors are normalized such that: Z'*Z = I.

 *  mapZ    (input/output) INTEGER array, dimension (N)
 *          On entry:

 *          mapZ(i) = the id of a processor which has room for the
 *                    i-th eigenvector, i = 0 to n-1 (current code
 *                    requires this even when computing only some eigenvalues.
 *          On exit:
 *            mapZ(i) = the id of the processor which actually owns the i-
th
 *                      eigenvector, i = 0 to n-1.

 *  eval    (output) DOUBLE PRECISION array, dimension (N)
 *          If INFO = 0, the eigenvalues  of the matrix
*         in no particular order.

*  iscratch (workspace) INTEGER array, dimension
*          Must be >= that returned from utility routine memreq_

*  iscsize  (input) INTEGER
*          The number of usable elements in array "iscratch".
*          Must be >= that returned from utility routine memreq_

*  dblptr   (workspace) DoublePrecision pointer to
*                      DoublePrecision (DoublePrecision **),
*          dblprt[0] must point to the start of an array consising of
*          pointers to DoublePrecision (DoublePrecision *).
*          Must be >= that returned from utility routine memreq_

*  ibuffsize(input) INTEGER
*          The number of usable elements in array "dblptr".
*          Must be >= that returned from utility routine memreq_

*  scratch  (workspace) DOUBLE PRECISION array, dimension
*          Must be >= that returned from utility routine memreq_

*  ssize    (input) INTEGER
*          The number of usable elements in array "scratch".
*          Must be >= that returned in utility routine memreq_

*  INFO    (output) INTEGER

*         = 0:  successful exit.

*         -50 <= INFO < 0, the -INFOth argument had an illegal value.

*         INFO = -51,      Input data which must be the same on all
*                          processors is NOT the same on all processors
*                          in mapZ.

*         = 1,             Error computing the eigenvalues of the
*                          tridiagonal eigenproblem.  In this case
*                          'pstebz_' returned a non-zero info, whose
*                          value was printed to stderr.

*         = 2,             Error computing the eigenvectors of the
*                          tridiagonal eigenproblem.  In this case
*                          'pstein' returned a non-zero info, whose
*                          value was printed to stderr.

*         Any processor with a negative INFO stops program execution.

*         If -50 <= INFO < 0, then bad data was passed to this routine
*                              and no attempt is made to make sure that
*                              INFO is the same on all processors.

*          All other INFO      INFO should be the same on all processors in
*                              mapA,Z.  If this routine calls a routine
*                              which returns a negative info, then info
*                              may not be the same on all processors.
*                              This, however, should never happen.
*---------------------------------------------------------------------

*/
```

4.3 PDSPGV and PDSPGVX: Generalized Symmetric Eigensystem Solver

This is the real general symmetric eigensystem problem: Given a real symmetric matrix $\mathbf{A}$ and a positive definite real symmetric matrix $\mathbf{B}$, find all its eigen-pairs $(\lambda, \mathbf{x})$ such that $\mathbf{Ax} = \lambda\mathbf{Bx}$.

For one processor we use the standard algorithm *(c.f. Wilkinson, J. H., The Algebraic Eigenvalue Problem, page 337-339)*. For multi-processors we perform the reduction to the standard eigensystem problem by computing the inverse of the lower triangular matrix from the Choleski factorization, and then perform matrix conjugation to obtain $\mathbf{L}^T\mathbf{A}\mathbf{L}^{-T}$. **Currently, the multi-processor reduction algorithm is used even when using only one processor**.

In some applications, the matrix **B** may already have been factored and the inverse of the choleski factor is stored in **B**. A parameter **ifact = 0** is used as an input to **pdspgv** to skip the choleski factorization (for serial) and its inverse computations (for parallel). A value **ifact = 1** is used to specify that **B** is not the Choleski factor of **B** or its inverse.

In Fortran 77 the calling sequences for PDSPGV and PDSPGVX are:

```
      subroutine pdspgv (ifact, n, matrixA, mapA,
                         matrixB, mapB, matZ, mapZ, eval,
                         iscratch, iscsize, dblptr, ibuffsz,
                         scratch, rsize, info)

      integer  ifact, n, mapA(*), mapB(*), mapZ(*), iscratch(*), iscsize,
                  ibuffsz, rsize, info;

      double precision  matrixA(*), matrixB(*), matZ(*), eval(*), scratch(*);
      double precision  dblptr(*)

c The input arguments are the same as in the C code
c pdspgv except that one changes the arguments of type
c       "1-D array of pointers to DoublePrecision", i.e.,
c vecA, vecB, vecZ and dblptr, to type
c "1-D array of double precision numbers" and store the data in
c       these arrays using the packed storage format described previously.
c       Also, when calling from fortran use FMEMREQ, rather than memreq_,
c       to get the required size of workspace arrays.
c       Finally, the C code describes arrays of length n with indices
c       of 0 to n-1, for fortran usage the indices should be 1 to n.
c
      subroutine pdspgvx (ifact, ivector, irange, n, matrixA, mapA,
                         matrixB, mapB, matZ, mapZ, eval,
                         iscratch, iscsize, dblptr, ibuffsz,
                         scratch, rsize, info)

      integer  ifact, n, mapA(*), mapB(*), mapZ(*), iscratch(*), iscsize,
                  ibuffsz, rsize, info;

      double precision  matrixA(*), matrixB(*), matZ(*), eval(*), scratch(*);
      double precision  dblptr(*)
```

```
c The input arguments are the same as in the C code
c pdspgvx except that one changes the arguments of type
c       "1-D array of pointers to DoublePrecision", i.e.,
c vecA, vecB, vecZ and dblptr, to type
c "1-D array of double precision numbers" and store the data in
c       these arrays using the packed storage format described previously.
c       Also, when calling from fortran use FMEMREQ, rather than memreq_,
c       to get the required size of workspace arrays.
c       Finally, the C code describes arrays of length n with indices
c       of 0 to n-1, for fortran usage the indices should be 1 to n.
c
```

In C the calling sequence for PDSPGV is:

```
void pdspgv( ifact, n, vecA, mapA, vecB, mapB, vecZ, mapZ,
    eval, iscratch, iscsize, dblptr, ibuffsize, scratch, ssize, info)
     Integer          *ifact, *n, *mapA, *mapB, *mapZ, *iscratch,
                      *iscsize, *ibuffsize, *ssize, *info;
     DoublePrecision  **vecA, **vecB, **vecZ, *eval, **dblptr, *scratch;

  /*

   *  Our parallel version of LAPACK's dspgv.


   *  Purpose
   *  =======

   *  pdspgv computes all of the eigenvalues and eigenvectors of a
   *  real generalized symmetric-definite eigenproblem, of the form

   *  A*x=(lambda)*B*x.

   *  Here A and B are assumed to be symmetric and B is also
   *  positive definite. The matrices A and B use packed storage.

   *  Arguments
   *  =========

   * NOTE: The current code assumes mapA, mapB and mapZ each contain
   *       exactly the same set of processor id's, though not necessarily
   *       in the same order.

   * All arguments are POINTERS to date of the specified type unless
   * otherwise mentioned.  In particular, INTEGER = (Integer *) and
   * DOUBLE PRECISION = (DoublePrecision *)


   * In the following let:
   *    n      = dimension of matrices A and B
   *    me     = this processors id (= mxmynd_())
*    nprocs = number of allocated processors ( = mxnprc_())
*    nvecsA = number of entries in mapA equal to me
*             (= count_list( me, mapA, n ))
*             number of columns of A this processor has
*    nvecsB = number of entries in mapB equal to me
*             (= count_list( me, mapB, n ))
*             number of vectors this processor has
*    nvecsZ = number of entries in mapZ equal to me
*             (= count_list( me, mapZ, n ))

*-----------------------------------------------------------------

*  ifact   (input) INTEGER
*          Specifies whether or not to factor the B matrix.
*          = 0:  Don't factor B, vecB is already the L in B = L*L'.
*                (for serial ) or L**(-1) for parallel
*                which was computed on a previous call to pdspgv.

*          = 1:  Factor B

*  n       (input) INTEGER
*          The number of rows and columns of the matrices A and B.
*          N >= 0.

*  vecA    (input/workspace) array of pointers to DoublePrecision (Dou-
blePrecision **)
*                            dimension ( nvecsA )
*          On entry, vecA[i], i = 0 to nvecsA-1, points to an
*          array containing the lower triangular part of the i-th
*          column of A which is owned by this processor.  The
*          columns of A owned by this processer are determined by mapA
*          (See below).

*          On exit, the contents of matrixA are destroyed.

*  mapA    (input) INTEGER array, dimension (N)
*          mapA(i) = the id of the processor which owns column i
*                    of the A matrix, i = 0 to n-1.

*  vecB    (input/output) array of pointers to DoublePrecision (Dou-
blePrecision **)
*                            dimension ( nvecsB )
*          On entry, vecB[i], i = 0 to nvecsB-1, points to an
*          array containing the lower triangular part of the i-th
*          column of B which is owned by this processor.  The
*          columns of B owned by this processer are determined by mapB
*          (See below).

*          On exit
```

```
   *           Let L be the triangular factor L from the Choleski
   *           factorization B = L*L', then

   *           if (nprocs = 1): vecB contains  L,          same storage as B
   *           else           : vecB contains (L inverse), same storage as B


   *  mapB    (input) INTEGER array, dimension (N)
   *          mapB(i) = the id of the processor which owns column i
   *                    of the B matrix, i = 0 to n-1.

   *  vecZ    (output) array of pointers to DoublePrecision (DoublePre-
cision **)
   *                   dimension ( nvecsZ )
   *          On entry, vecZ[i], i = 0 to nvecsZ-1, should point to an
   *          array of length n.

   *          On exit:

   *            vecZ[i], i = 0 to nvecsZ-1, points to the i-th eigenvector
   *            (as determined by the exit values in mapZ) owned by this
   *            processor.

   *            The eigenvectors are normalized such that: Z'*B*Z = I.

   *  mapZ    (input/output) INTEGER array, dimension (N)
   *          On entry:

   *          mapZ(i) = the id of a processor which has room for the
   *                    i-th eigenvector, i = 0 to n-1.
   *          On exit:
   *            mapZ(i) = the id of the processor which actually owns the i-
th
   *                      eigenvector, i = 0 to n-1.

   *  eval    (output) DOUBLE PRECISION array, dimension (N)
   *          If INFO = 0, the eigenvalues  of the matrix
   *          in no particular order.

   *  iscratch (workspace) INTEGER array, dimension (??)

   *  iscsize  (input) INTEGER
   *           The number of usable elements in array "iscratch".
   *           Must be >= ???.

   *  dblptr   (workspace) DoublePrecision pointer to DoublePrecision (Dou-
blePrecision **),
   *           dblprt[0] must point to the start of an array consising of
   *           ??? pointers to DoublePrecision.
*  ibuffsize(input) INTEGER
*           The number of usable elements in array "dblptr".
*           Must be >= ???.

*  scratch  (workspace) DOUBLE PRECISION array, dimension (??)

*  ssize    (input) INTEGER
*           The number of usable elements in array "scratch".
*           Must be >= ???.

*  INFO    (output) INTEGER

*          = 0:  successful exit.

*          -50 <= INFO < 0, the -INFOth argument had an illegal value.

*          INFO = -51,      Input data which must be the same on all
*                           processors is NOT the same on all processors
*                           in mapZ.

*          = 1,             Error choleski factoring B.  In this case
*                           'choleski' returned a non-zero info, whose
*                           value was printed to stderr.

*          = 2,             Error computing inverse of L, the choleski
*                           factor of B.  In this case
*                           'inverseL' returned a non-zero info, whose
*                           value was printed to stderr.

*          = 3,             Error solving the standard eigenproblem.
*                           In this case
*                           'pdspevx' returned a non-zero info, whose
*                           value was printed to stderr.

*          Any processor with a negative INFO stops program execution.

*          If -50 <= INFO < 0, then bad data was passed to this routine
*                              and no attempt is made to make sure that
*                              INFO is the same on all processors.

*          All other INFO      INFO should be the same on all proces-
sors in
*                              mapA,B,Z.  If this routine calls a routine
*                              which returns a negative info, then info
*                              may not be the same on all processors.
*                              This, however, should never happen.
*    --------------------------------------------------------------
-------
```

```
    */
```

4.4 Level II Subroutines

This section describes the level 2 subroutines. As of this writing we present a brief summary of what each routine does. These routines are called in support of the eigensystem solvers but they can be useful for other calculations too.

Many of the level II routines do little or no checking of input data, hence are not as robust against input errors as the level I routines. Also, some of the routines described in this section **cannot** currently be called directly from fortran (only because the fortran-C interface routines have not yet been tested).

The level II routines which can currently be called from fortran are: bortho, mxm25, mxm5x, mxm88, ortho, pstebz, resid, residual, and tresid.

The memory requirements for the level II routines are desribed in the headers for the various routines. The listed memory requirements are believed to be as large or somewhat larger than what the various routines actually use. However, this has not currently been tested.

Finally, before calling any of PeIGS routines, except the level I routines, one **must call the fortran routine mxinit() (mxinit_() from C)** to initialize the communication package.

4.4.1 Choleski Factorization of Positive Definite Symmetric Matrix

This subroutine performs a submatrix Choleski factorization of a symmetric positive matrix with a column wrap using a list of processors. This code originated from Mike Heath *(Geist, G. A., and M. T. Heath, "Matrix Factorization on a Hypercube Multiprocessor," in Hypercube Multiprocessors 1986, SIAM, Philadelphia, 1986, pp. 161-180).* We modified it for our purposes and are fully responsible for any bugs.

The C syntax is:

```
void choleski( n, vecA, mapA, iscratch, scratch, info )

     Integer            *n, *mapA, *iscratch, *info;

     DoublePrecision         *scratch;

     DoublePrecision        **vecA;

   /*
    *  ==================================================================


    *  The original code is due to Mike Heath.  George Fann
    *  bastardized it for our code.  We thank Mike
    *  for his generosity.  Any bugs and problems are introduced
    *  by us.
   *  Currently, this procedure only handles the case where the
   *  lower triangular part of A is distributed to processors by columns.


   *  Purpose
   *  =======

   *  choleski computes the Cholesky factorization of a real symmetric
   *  positive definite matrix A in packed storage format.

   *  The factorization has the form
   *     A = L  * L'
   *  where L is lower triangular.  On output the matrix A is overwrit-
ten by
   *   the Choleski factor L.


   *  Arguments
   *  =========

   *  In the following let:

   *     me     = The id of this processor

   *     nvecsA = The number of columns of A owned by this processor.
   *              In particular, nvecsA = count_list(me, mapA, n).

   *     nprocsA = Number of distinct processor id's in "mapA".  In
   *               particular, nprocs = reduce_list( n, mapA, iscratch )

   *  n       (input) (Integer *) scaler
   *          The order of the matrix A.  n >= 0.

   *  vecA    (input/output) (DoublePrecision **) array, length (nvecsA)
   *          On entry, the portion of the lower triangular part of the
   *          symmetric matrix A owned by this processor.

   *          vecA[i] points to an array containing the lower triangular
   *          part of the i-th column of A which is owned by this
   *          processor, i = 0 to nvecsA.

   *          On exit, if *info = 0, the factor L from the Cholesky
   *          factorization A = L*L^(T).

   *  mapA    (input) (Integer *) array, length (*n)
   *          mapA[i] = the id of the processor which owns column i of A,
   *          i = 0 to *n - 1.

   *  iscratch  (integer workspace) (Integer *) array, length (2 n + 1)
*  scratch   (DoublePrecision workspace) (DoublePrecision *) array, length (n+1)

*            scratch must contain at least bufsiz bytes (see cmbbrf.h)


*  info      (output) (Integer *) scaler
*            = 0: successful exit
*            < 0: if info = -k, the k-th argument had an illegal value
*            > 0: if info = k, 1<= k <= n, the leading minor of order
*                 k is not positive definite, and the factorization
*                 could not be completed.

*  =================================================================
*/
```

4.4.2 Inverse of Lower Triangular Matrix

The subroutine **inverseL** computes the inverse of a lower triangular matrix in column wrapped form(according to a list). On output the input matrix is overwritten with its output.

The C syntax is:

```
void inverseL ( msize, col, map, iwork, buffer, info)
     Integer *msize, *map, *iwork, *info;
     DoublePrecision **col, *buffer;

     /*
       This routine computes the inverse of a lower triangular matrix.
       On output, the lower triangular matrix is overwritten by its inverse.
       The matrix is assume to be column wrapped and distributed according
       to the array map.  Thus, map[i] = processor id which holds column
       i of the matrix.

       Arguments:

       msize = (input) integer, size of the matrix

       col = (input/1-D array of pointers to DoublePrecision numbers)
     col[j] -> the address of the j-th vector that this processor owns

       map = (input/integer array) integer array, length n, such that for
              the i-th column,
     map[i] = real processor id holding this column

       iscratch = (scratch/integer) integer scratch space of size nprocs

       buffer = (scratch) DoublePrecision precision array

       info  = (output/integer)
```

```
                  if info == 0 normal
                  if info == i > 0 then the i-th diagonal element
                ( in Fortran notation ) is smaller than
                        dlamch("s"), the safe inverse limit: overflow
                        may have occurred.
        */
```

4.4.3 Matrix Multiplication

There are several different matrix multiplication routines depending on how the data is distributed and the shape of the matrix. The Fortran 77 calling syntax is given in very little detail. Most of the detail and descriptions of the routines are included with the C calling syntax.

The Fortran 77 syntax is:

```
        SUBROUTINE mxm25 ( n1, n2, rowQ, mapQ, m, colW, mapW, colZ, iwork, work)
        INTEGER          n1, n2, mapQ(*), m, mapW(*), iwork(*)
        Double Precision rowQ(*), colW(*), colZ(*), work(*)
        SUBROUTINE mxm5x( n, rowU, mapU, m, colF, mapF, iscratch, scratch)
        Integer          n, mapU(*), mapF(*),  m, iscratch(*)
        Double Precision colF(*), rowU(*), scratch(*)
        SUBROUTINE mxm88 ( n, colQ, mapQ, m, colW, mapW, iwork, work )
        Integer          n, mapQ(*), m, mapW(*), iwork(*)
        Double Precision colQ(*), colW(*), work(*)

   C  Note that the C routine mxm88 has one more argument, iptr, than the
   C  Fortran 77 call syntax listed here.
```

The input arguments for the above routines are the same as in the C codes listed below except that one changes the arguments of type "1-D array of pointers to DoublePrecision", i.e., rowQ, colW, etc, to type "1-D array of double precision numbers" and store the data in these arrays using the packed storage format described previously. The C code describes arrays of length n with indices of 0 to n-1, for fortran usage the indices should be 1 to n.

The double precision work arrays for the Fortran 77 codes must be somewhat larger than listed below in the C calling syntax. In particular, let ME, be the id of a processor, and let nvecsX be the number of entries in array mapX equal to ME, where X = Q,W,U, or F. Then, on processor ME the extra memory requirements for the Fortran 77 routines are:

- MXM25: increase size of work by nvecsQ+nvecsW+nvecsZ
- MXM5x: increase size of scratch by nvecsU+nvecsF+2
- MXM88: increase size of work by nvecsQ+2*nvecsW+3

The C syntax is:

```
   void mxm25 ( n1, n2, rowQ, mapQ, m, colW, mapW, colZ, iwork, work)
        Integer *n1, *n2, *mapQ, *m, *mapW, *iwork;
        DoublePrecision **rowQ, **colW, **colZ, *work;

   /***************************************************************
    *
    * Subroutine mxm25
```

```
*
  This subroutine does the matrix multiplication Z <- Q*W

  where matrix Q is a n1 x n2 general matrix in packed storage format,
        distributed by rows,

        matrix W is a general n2 x m matrix in packed storage format,
        distributed by columns.

        and matrix Z is the n1 x m product Q*W, distributed the same as W.

  In this version we assume that Q and W share the same processors
  ( perhaps different data distributions )

  It is ok to have colZ = colW.  However, in this
  case each colZ[i]=colW[i] must point to a vector of
  length = MAX( n1, n2 ).

  ARGUMENTS
  ---------
  In the following:

    INTEGER          = "pointer to Integer"
    DOUBLE PRECISION = "pointer to DoublePrecision"

    me     = this processor's id (= mxmynd_())
    nprocs = number of allocated processors ( = mxnprc_())
    nvecsW = number of entries in mapW equal to me
                 (= count_list( me, mapW, m ))
    nvecsQ = number of entries in mapQ equal to me
                 (= count_list( me, mapQ, n1 ))
    nvecsQ_max = maximum number of entries in mapQ equal to i,
                 i = 0 to nprocs-1
                 (= max over i of count_list( i, mapQ, n1 ))
    sDP    = sizeof( DoublePrecision )

  n1 ..... (input) INTEGER
           The number of rows in matrix Q

  n2 ..... (input) INTEGER
           The number of columns in matrix Q
           and the number of rows in matrix W

  rowQ ... (input) array of pointers to DoublePrecision,
                   length (nvecsQ)
           The part of matrix Q owned by this processer stored
           in packed format, i.e., rowQ[i] points to the start
           of the i-th row of Q
           owned by this processor, i = 0 to nvecsQ-1.
```

```
   mapQ ... (input) INTEGER array, length (n1)
            The i-th row of Q is owned by processor
            mapQ[i], i = 0 to n1-1.

   m ...... (input) INTEGER
            The number of columns in W.

   colW ... (input) array of pointers to DoublePrecision,
                               length (nvecsW)

            The part of matrix W owned by this processer stored
            in packed format by columns, i.e., colW[i] points
            to the first element of the i-th column of W
            owned by this processor, i = 0 to nvecsW-1.

   mapW ... (input) INTEGER array, length (m)
            The i-th column of W is
            owned by processor mapW[i], i = 0 to m-1.

   colZ ... (output) array of pointers to DoublePrecision,
                         length (nvecsW)

             The result matrix Z = Q * W stored identical to W.

             It is ok to have colZ = colW.  However, in this
             case each colZ[i]=colW[i] must point to a vector of
             length = MAX( n1, n2 ).

   iwork .. (workspace) INTEGER array, length ( 3 * n1 + 2 * m )

   work ... (workspace) DOUBLE PRECISION array,
                        length ( maximum ( n1 + m, (nvecsW + 2 * nvecsQ_max) * n2 )
*/

void mxm5x( n, rowU, mapU, m, colF, mapF, iscratch, scratch)
     Integer *n, *mapU, *mapF,  *m, *iscratch;
     DoublePrecision **colF, **rowU, *scratch;

/***************************************************************
 *
 * Subroutine mxm5x
 *
   This subroutine does the matrix multiplication F <- U * F

   where U is a n x n upper trianguler matrix in packed storage format,
   and row distributed,
   and matrix F is a general n x m matrix distributed by columns.

   ARGUMENTS
```

```
---------
In the following:

  INTEGER          = "pointer to Integer"
  DOUBLE PRECISION = "pointer to DoublePrecision"

  me     = this processor's id (= mxmynd_())
  nprocs = number of allocated processors ( = mxnprc_())
  nvecsU = number of entries in mapU equal to me
           (= count_list( me, mapU, n ))
  neleU_max = maximum number of elements of U owned by any
              processor.
              (= max over i of ci_size_( i, n, mapU) )
  nvecsF = number of entries in mapF equal to me
           (= count_list( me, mapF, m ))
  sDP    = sizeof( DoublePrecision )

n ...... (input) INTEGER
         The number of rows and columns of U, and the number
         of rows of F.

rowU ... (input) array of pointers to DoublePrecision,
                 length (nvecsU)
         The part of matrix U owned by this processer stored
         in packed format, i.e., rowU[i] points to the diagonal
         element of the i-th row of U
         owned by this processor, i = 0 to nvecsU-1.

mapU ... (input) INTEGER array, length (n)
         The i-th row of U is
         owned by processor mapU[i], i = 0 to n-1.

m ...... (input) INTEGER
         m is the number of columns in F.

colF ... (input/output) array of pointers to DoublePrecision,
                        length (nvecsF)

         On Entry:
           The part of matrix F owned by this processer stored
           in packed format by columns, i.e., colF[i] points
           to the first element of the i-th column of F
           owned by this processor, i = 0 to nvecsF-1.

         On Exit:
           The result matrix U * F stored in the same manner as
           F on entry.

mapF ... (input) INTEGER array, length (m)
```

```
                The i-th column of F is
                owned by processor mapF[i], i = 0 to m-1.

      iscratch .. (workspace) INTEGER array, length ( 3*n+2*m +1 )

      scratch.... (workspace) DOUBLE PRECISION array,
                          length ( maximum ( n+m, mxlbuf_()/sDP + 1,
                                             (nvecsF * n + 2 * neleU_max )
   */

   void mxm88 ( n, colQ, mapQ, m, colW, mapW, iwork, work, iptr)
        Integer *n, *mapQ, *m, *mapW, *iwork;
        DoublePrecision **colQ, **colW, *work, **iptr;

   /***************************************************************
    *
    * Subroutine mxm88
    *
      This subroutine does the matrix multiplication W <- Q * W

      where Q is a n x n symmetric matrix in packed storage format,
      and row (or equivalently column) distributed,
      and matrix W is a general n x m matrix distributed by columns.

      ARGUMENTS
      ---------
      In the following:

        INTEGER          = "pointer to Integer"
        DOUBLE PRECISION = "pointer to DoublePrecision"

        me     = this processor's id (= mxmynd_())
        nprocs = number of allocated processors ( = mxnprc_())
        nvecsW = number of entries in mapW equal to me
                     (= count_list( me, mapW, m ))
        nvecsQ = number of entries in mapQ equal to me
                     (= count_list( me, mapQ, n ))
        sDP    = sizeof( DoublePrecision )

      n ...... (input) INTEGER
               n is the dimension of the symmetric matrix Q

      colQ ... (input) array of pointers to DoublePrecision,
                       length (nvecsQ)
               The part of matrix Q owned by this processer stored
               in packed format, i.e., colQ[i] points to the diagonal
               element of the i-th column (or equivalently row) of Q
               owned by this processor, i = 0 to nvecsQ-1.

      mapQ ... (input) INTEGER array, length (n)
```

```
                The i-th column (or equivalently row) of Q is
                owned by processor mapQ[i], i = 0 to n-1.

       m ...... (input) INTEGER
                m is the number of columns in W.

       colW ... (input/output) array of pointers to DoublePrecision,
                               length (nvecsW)

                On Entry:
                  The part of matrix W owned by this processer stored
                  in packed format by columns, i.e., colW[i] points
                  to the first element of the i-th column of W
                  owned by this processor, i = 0 to nvecsW-1.

                On Exit:
                  The result matrix Q * W stored in the same manner as
                  W on entry.

       mapW ... (input) INTEGER array, length (m)
                The i-th column of W is
                owned by processor mapW[i], i = 0 to m-1.


       iwork .. (workspace) INTEGER array, length ( 2*(n+m)+nvecsW+nvecsQ )

       work ... (workspace) DOUBLE PRECISION array,
                            length ( maximum ( (nvecsW+1)*n, mxlbuf_()/sDP + 1 )

       iptr ... (workspace ) array of pointers to DoublePrecision,
                             length (nvecsW)

    */
```

4.4.4 TRED2: Householder reduction of a symmetric matrix to tridiagonal form

Let **A** be a symmetric matrix stored in packed. Then subroutine TRED2 computes the Householder transformation matrix Q and the tridiagonal matrix **T** such that $\mathbf{A} = \mathbf{QTQ^T}$

This code originates from Shirish Chinchalkar. We have modified it to suit our purpose. We are responsible for introducing any bugs or errors.

```
int tred2 (n, vecA, mapA, Q, mapQ, diag, upperdiag, iwork, work )
     double **vecA,                /* matrix to be reduced */
       **Q,                 /* Householder matrix */
       *diag,              /* diagonal elements of tri-diagonal matrix */
       *upperdiag,         /* upper diagonal elements of tri-diagonal ma-
trix */
       *work;            /* Householder vector (temp space of size n dou-
bles) */
```

```
    int   *n,                 /* problem size */
      *mapA,
      *mapQ,
      *iwork;         /* integer scratch work space */

 /*

TRED2 : reduction of a real symmetric matrix to tri-diagonal form
        Uses Householder reductions.

INPUT:
   n     ->  integer , matrix size.

   **A   ->  (1-D array of pointers to doubles) input matrix to be reduced.

   mapA  ->  (integer array/ input) integer array of length n
                      distribution the

  **Q    ->  (1-D array of pointers to doubles) Householder matrix to be stored
                       in packed storage format as discussed in section 3.2.
                       currently assumes that mapA[i] = mapQ[i] for all i

  mapQ        ->  (integer array/ input) integer array of length n
                      distribution the same as mapA

   OUTPUT:

   diag[0:n-1]         ->  diagonal elements of the tri-diagonal matrix
                       obtained from A
   upperdiag[1:n-1]    ->  upper diagonal elements of the tri-diagonal matrix
                       obtained from A. upperdiag[0] is set to 0.0 and
                       upperdiag[1] to upperdiag[n-1] contains the desired
                       result

   **Q    ->  Householder matrix ( generated by Householder reflections)
              distributed by rows(or columns) in a wrap fashion (compact)
              same distribution as mapA

   mapQ       (integer array of length n) current returns a copy of mapA,
              same distribution as matrixA

  */
```

4.4.5 PSTEBZ eigenvalues of real symmetric tridiagonal matrix

Given a real symmetric tridiagonal matrix **T**. The subroutine **PSTEBZ** computes the specified eigenvalues of **T** by bisection using Sturm sequence. The syntax used is not quit that of current LAPACK dstebz (notice that the e-array is different).

```
      subroutine pstebz( job, n, lb, ub, jjjlb, jjjub, abstol, d, e,
     $                   mapZ, neigval, nsplit, eval, iblock, isplit,
     $                   work, iwork, info)

      Integer            job, n, jjjlb, jjjub, mapZ(*), neigval, nsplit,
     $                   iblock(*), isplit(*), iwork(*), info
      Double Precision   lb, ub, abstol, d(*), e(*), eval(*), work(*)
c
c     Arguments are the same as for the C call sequence, but any
c     reference to "pointer to data type" should be interpreted as
c     "array of data type".  Also, C routine indexing "from 0 to k-1" should
c     be interpreted as indexing "from 1 to k" in fortran.
c

void pstebz_( job, n, lb, ub, jjjlb, jjjub, abstol, d, e, mapZ, neigval,
     nsplit, eval, iblock, isplit, work, iwork, info)

     Integer            *job, *n, *jjjlb, *jjjub, *mapZ, *neigval, *nsplit,
                    *iblock, *isplit, *iwork, *info;
     DoublePrecision         *lb, *ub, *abstol, *d, *e, *eval, *work;

/*
 *  Parallel version of LAPACK's DSTEBZ.

 *  This routine is directly callable from both FORTRAN and C.
 *  The documentation below always uses FORTRAN array indexing,
 *  i.e., 1 to N, rather then C array indexing, i.e.,  0 to N-1.
 *  This should be kept in mind when calling this routine from C.
 *  Also when calling this routine from C
 *    INTEGER (array)           means "pointer to Integer" and
 *    DOUBLE PREICISION (array) means "pointer to DoublePrecision"


 *  Purpose
 *  =======

 *  PSTEBZ computes the eigenvalues of a symmetric tridiagonal
 *  matrix T.  The user may ask for all eigenvalues, all eigenvalues
 *  in the half-open interval (LB, UB], or the JJJLB-th through JJJUB-th
 *  eigenvalues.

 *  To avoid overflow, the matrix must be scaled so that its
 *  largest element is no greater than overflow**(1/2) *
 *  underflow**(1/4) in absolute value, and for greatest
 *  accuracy, it should not be much smaller than that.

 *  See W. Kahan "Accurate Eigenvalues of a Symmetric Tridiagonal
 *  Matrix", Report CS41, Computer Science Dept., Stanford
 *  University, July 21, 1966.
*  Arguments
*  =========


*  JOB     (input) INTEGER
*          = 1  : ("All")   all eigenvalues will be found.
*          = 2  : ("Value") all eigenvalues in the half-open interval
*                           (LB, UB] will be found.
*          = 3  : ("Index") the JJJLB-th through JJJUB-th eigenvalues (of the
*                           entire matrix) will be found.

*  N       (input) INTEGER
*          The order of the tridiagonal matrix T.  N >= 0.

*  LB      (input) DOUBLE PRECISION
*  UB      (input) DOUBLE PRECISION
*          If JOB=2, the lower and upper bounds of the interval to
*          be searched for eigenvalues.  Eigenvalues less than or equal
*          to LB, or greater than UB, will not be returned.  LB < UB.
*          Not referenced if JOB = 1 or 3.

*  JJJLB   (input) INTEGER
*  JJJUB   (input) INTEGER
*          If JOB=3, the indices (in ascending order) of the
*          smallest and largest eigenvalues to be returned.
*          1 <= JJJLB <= JJJUB <= N, if N > 0.
*          Not referenced if JOB = 1 or 2.

*  ABSTOL  (input) DOUBLE PRECISION
*          The absolute tolerance for the eigenvalues.  An eigenvalue
*          (or cluster) is considered to be located if it has been
*          determined to lie in an interval whose width is ABSTOL or
*          less.  If ABSTOL is less than or equal to zero, then ULP*|T|
*          will be used, where |T| means the 1-norm of T.

*          Eigenvalues will be computed most accurately when ABSTOL is
*          set to twice the underflow threshold 2*DLAMCH('S'), not zero.

*  D       (input) DOUBLE PRECISION array, dimension (N)
*          The n diagonal elements of the tridiagonal matrix T.

*  E       (input) DOUBLE PRECISION array, dimension (N)
*          The first element of E, E(1), is junk, the rest of E, E(2:N),
*          contains the (n-1) off-diagonal elements of the tridiagonal
*          matrix T.

*  MAPZ    (input) INTEGER array, dimension (N)
*          A list of the ids of the processors which are to participate
*          in the eigenvalue computation.
*  NEIGVAL (output) INTEGER
*          The actual number of eigenvalues found. 0 <= NEIGVAL <= N.

*  NSPLIT  (output) INTEGER
*          The number of diagonal blocks in the matrix T.
*          1 <= NSPLIT <= N.

*  EVAL    (output) DOUBLE PRECISION array, dimension (N)
*          On exit, the first NEIGVAL elements of EVAL will contain the
*          eigenvalues.  (PSTEBZ may use the remaining N-NEIGVAL elements as
*          workspace.)
*          The eigenvalues will be grouped by split-off block (see IBLOCK,
*          ISPLIT) and ordered from smallest to largest within the block.

*  IBLOCK  (output) INTEGER array, dimension (N)
*          At each row/column j where E(j) is zero or small, the
*          matrix T is considered to split into a block diagonal
*          matrix.  On exit, if INFO = 0, IBLOCK(i) specifies to which
*          block (from 1 to the number of blocks) the eigenvalue EVAL(i)
*          belongs.  (DSTEBZ may use the remaining N-NEIGVAL elements as
*          workspace.)

*  ISPLIT  (output) INTEGER array, dimension (N)
*          The splitting points, at which T breaks up into submatrices.
*          The first submatrix consists of rows/columns 1 to ISPLIT(1),
*          the second of rows/columns ISPLIT(1)+1 through ISPLIT(2),
*          etc., and the NSPLIT-th consists of rows/columns
*          ISPLIT(NSPLIT-1)+1 through ISPLIT(NSPLIT)=N.
*          (Only the first NSPLIT elements will actually be used, but
*          since the user cannot know a priori what value NSPLIT will
*          have, N words must be reserved for ISPLIT.)

*  WORK    (workspace) DOUBLE PRECISION array, dimension ( )

*  IWORK   (workspace) INTEGER array, dimension ( )

*  INFO    (output) INTEGER

*          PSTEBZ attempts to return the same INFO on all processors in MAPZ.
*          Currently, however, if the input data is invalid, -50 < INFO < 0,
*          then INFO will be different on different processors.

*          = 0:   successful exit

*          < 0 &
*          > -50: if INFO = -i, the i-th argument had an illegal value

*          = -51: Then the input data was not the same on all processors in MAPZ
```

```
 *          > 0:   some or all of the eigenvalues failed to converge or
 *                 were not computed, or they were computed incorrectly:

 *                 =1: routine DSTEBZ3 returned a non-zero info.  Meaning
 *                     that on one or more processors some or all of the
 *                     eigenvalues failed to converge or were not computed
 *                     In this case the processor with a non-zero info from
 *                     DSETBZ3 prints and error message to stderr.

 *                 =2: NSPLIT and/or ISPLIT were not the same on all processors

 *                 =3: Relative processor i computed an eigenvalue larger than
 *                     the smallest eigenvalue computed by relative pro-
cessor i+1
 *                     for some i.  This should not occur using the cur-
rent algorithms.

 *                 =4: The number of eigenvalues found in a block of T is bigger
 *                     then the dimension of the block.


 *               In theory INFO = 2, 3, or 4 should never occur.  If they do occur
 *               then DSTEBZ3 failed to correctly compute the requested eigenvalues.

 *               Note that DSTEBZ3 is a modification of LAPACK's DSTEBZ.  The
 *               modifications had to be made to avoid gettings INFOs like 3 and 4.

 */
```

4.4.6 PSTEIN: eigenvectors of real symmetric tridiagonal matrix

Given a real symmetric tridiagonal matrix **T** and some of its eigenvalues. The subroutine **PSTEBZ** computes the specified eigenvectors of **T**. The syntax used is not quit that of current LAPACK dstein (notice that the e-array is different).

```
void pstein ( n, dd, ee, meigval, eval, iblock, nsplit, isplit,
     mapZ, vecZ, ddwork, iiwork, ppiwork, info )

     Integer            *n, *meigval, *iblock, *nsplit, *isplit, *mapZ,
                        *iiwork, *info, **ppiwork;
     DoublePrecision        *dd, *ee, *eval, **vecZ, *ddwork;

  /*

   * Driver to compute eigenvectors of a symmetric tridiagonal matrix.
   * Basically a parallel version of LAPACK's DSTEIN.

   *  n ......... The dimension of the matrix.
   *  dd ....... The diagonal of the matrix.

   *  ee ....... The off-diagonal of the matrix.  ee[0] is junk, the
   *             actual off-diagonal starts at ee[1].

   *  meigval ... (input) Integer
   *             The number of eigenvalues found

   *  eval ...... (input) DoublePrecision array, length (neigval)
   *             The actual eigenvalues.  Sorted by split of block,
   *             and within a block sorted by value.

   *  iblock .... (input) Integer array, length (neigval)
   *             Array from pstebz.

   *  nsplit .... (input) Integer
   *             The number of split points in the tridiagonal matrix.
   *             From pstebz.

   *  isplit .... (input) Integer array, length (nsplit)
   *             Array from pstebz.

   *  mapZ    (input/output) INTEGER array, dimension (meigval)
   *         On entry:

   *         mapZ(i) = the id of a processor which has room for the
   *                   i-th eigenvector, i = 0 to meigval-1
   *         On exit:
   *           mapZ(i) = the id of the processor which actually owns the i-
th
   *                     eigenvector, i = 0 to meigval-1.

   *         The value of mapZ on exit may be different then on entry.

   *  vecZ    (output) array of pointers to DoublePrecision (DoublePre-
cision **)
   *                  dimension ( nvecsZ )
   *         On entry:
   *            vecZ[i], i = 0 to nvecsZ-1, should point to an array of length n.

   *         On exit:

   *           vecZ[i], i = 0 to nvecsZ-1, points to the i-th eigenvector
   *           (as determined by the exit values in mapZ) owned by this
   *           processor.

   *           The eigenvectors are normalized such that: Z'* Z = I.
   *  ddwork .... (workspace) DoublePrecision array
   *  iiwork .... (workspace) Integer array,
   *  ppiwork ... (workspace) array of pointers to Integer,


   *  INFO ... (output) INTEGER

   *          PSTEIN attempts to return the same INFO on all processors in MAPZ.
   *          Currently, however, if the input data is invalid, -50 < INFO < 0,
   *          then INFO will be different on different processors.

   *          = 0:   successful exit

   *          -50 <= INFO < 0:
   *                 Then the (-INFO)-th argument had an illegal value

   *          = -51: Then the input data was not the same on all proces-
sors in MAPZ

   *          0 < INFO <= MEIGVAL:
   *                 Then the INFO-th eigenvector failed to converge, but all
   *                 lower numbered eigenvectors did converge.

   *          N < INFO:
   *                 Then the residual,
   *                 res == ( max_i |T z_i-lamba_i z_i | /( eps |T|) ),
   *                 for the tridiagonal eigenproblem was excessively large.
   *                 In particular, INFO = N + (Integer) log10( res )
   */
```

4.4.7 Residual and Orthogonality checks

This section describes routines for computing residuals for the various eigenproblems, and for testing the orthogonality of computed eigenvectors. The Fortran 77 calling syntax is given in very little detail. Most of the detail and descriptions of the routines are included with the C calling syntax.

The Fortran 77 syntax is:

```
      SUBROUTINE tresid( n, m, d, e, colZ, mapZ, eval, iwork, work, res, info)
      Integer          n, m, mapZ(*), iwork(*), info
      DoublePrecision d(*), e(*), colZ(*), eval(*), work(*), res

      SUBROUTINE resid( n, colA, mapA, m, colZ, mapZ, eval, iwork, work,
                        res, info)
      Integer          n, mapA(*), m, mapZ(*), iwork(*), info
      DoublePrecision colA(*), colZ(*), eval(*), work(*), res

C  Note that the C routine resid has one more argument, ibuffptr, than the
C  Fortran 77 call syntax listed here.
      SUBROUTINE residual( n, colA, mapA, colB, mapB, m, colZ, mapZ, eval,
                         iwork, work, res, info)
     Integer         n, mapA(*), mapB(*), m, mapZ(*), iwork(*), info
     DoublePrecision colA(*), colB(*), colZ(*), eval(*), work(*), res

C  Note that the C routine residual has one more argument, ibuffptr, than the
C  Fortran 77 call syntax listed here.
     SUBROUTINE ortho( n, m, colZ, mapZ, iwork, work, ort, info)
     Integer n, m, mapZ(*), iwork(*), info
     DoublePrecision colZ(*), work(*),  ort

C  Note that the C routine ortho has one more argument, ibuffptr, than the
C  Fortran 77 call syntax listed here.
     SUBROUTINE bortho( n, colB, mapB, m, colZ, mapZ, iwork, work, ort, info)
     Integer         n, mapB(*), m, mapZ(*), iwork(*), info
     DoublePrecision colB(*), colZ(*), work(*),  ort

C  Note that the C routine bortho has one more argument, ibuffptr, than the
C  Fortran 77 call syntax listed here.
```

The input arguments for the above routines are the same as in the C codes listed below except that one changes the arguments of type "1-D array of pointers to DoublePrecision", i.e., rowQ, colW, etc, to type "1-D array of double precision numbers" and store the data in these arrays using the packed storage format described previously. The C code describes arrays of length n with indices of 0 to n-1, for fortran usage the indices should be 1 to n.

The work arrays for the Fortran 77 codes must be somewhat larger than listed below in the C calling syntax. In particular, let ME, be the id of a processor, and let nvecsX be the number of entries in array mapX equal to ME, where X = Q,W,U, or F. Then, on processor ME the extra memory requirements for the Fortran 77 routines are:

- TRESID: increase size of work by nvecsZ+1
- RESID: increase size of work by nvecsA+3*nvecsZ+4
- RESIDUAL: increase size of work by nvecsA+nvecsB+4*nvecsZ+6
- ORTHO: increase size of work by 2*nvecsZ+2
- BORTHO: increase size of work by nvecsB+4*nvecsZ+5

The C syntax is:

```
void tresid( n, m, d, e, colZ, mapZ, eval, iwork, work, res, info)
     Integer *n, *m, *mapZ, *iwork, *info;
     DoublePrecision *d, *e, **colZ, *eval, *work, *res;

/*
        this subroutine computes the residual

        res = max_(i) | T z_(i) - \lambda_(i) z_(i) |/( | T | * ulp )

        where T is an n-by-n  tridiagonal matrix,
         ( \lambda_(i) , z_(i) ) is a standard eigen-pair, and
        ULP = (machine precision) * (machine base)
```

```
    |T| is the 1-norm of T
    |T z_(i) .... | is the infinity-norm
    res is reasonable if it is of order about 50 or less.

Arguments
---------
In the following:

  INTEGER          = "pointer to Integer"
  DOUBLE PRECISION = "pointer to DoublePrecision"

  me     = this processor's id (= mxmynd_())
  nprocs = number of allocated processors ( = mxnprc_())
  nvecsZ = number of entries in mapZ equal to me
                (= count_list( me, mapZ, n ))
  sDP    = sizeof( DoublePrecision )


n....... (input) INTEGER
         dimension of the matrix T

m....... (input) INTEGER
         number of eigenvalues/eigenvectors

d....... (input) DOUBLE PRECISION array, length (n)
         diagonal of T

e....... (input) DOUBLE PRECISION array, length (n)
         e[0] = junk,
         e[1:n-1] = sub-diagonal of T
                  = super-diagonal of T

colZ ... (input) array of pointers to DoublePrecision,
                 length (nvecsZ)
         The part of matrix Z owned by this processer, stored
         in packed format, i.e., colZ[i] points to the start
         of the i-th column of matrix Z owned by this
         processor, i = 0 to nvecsZ-1.

mapZ ... (input) INTEGER array, length (m)
         The i-th column of matrix Z is owned by processor
         mapZ[i], i = 0 to m-1.

eval.... (input) DOUBLE PRECISION array, length (m)
         the eigenvalues

iwork... (workspace) INTEGER array, length(m)

work.... (workspace) DOUBLE PRECISION array,
                     length( mxlbuf_() / sDP + 1 )

   res..... (output) INTEGER
            the residual described above.

   info.... (output) INTEGER
            = 0, not currently used
*/

void resid( n, colA, mapA, m, colZ, mapZ, eval, ibuffptr, iwork, work,
            res, info)
     Integer *n, *mapA, *m, *mapZ, *iwork, *info;
     DoublePrecision **colA, **colZ, *eval, **ibuffptr, *work, *res;

/*
   this subroutine computes the residual

   res = max_(i) | A z_(i) - \lambda_(i) z_(i) |/( | A | * ulp )

   where
   A is an n-by-n symmetric matrix, in packed storage format,
   column (or equivalently row) distributed

   (lambda_i, z_i) is a standard eigen-pair of A and
   Z is an n-by-m matrix of eigenvectors, in packed storage format,
   column distributed

   ULP = (machine precision) * (machine base)

   |A z_(i) ... | is the infinity-norm,
   |A| is the 1-norm of A,
   res is reasonable if it is of order about 50 or less.

   Arguments
   ---------
   In the following:

     INTEGER          = "pointer to Integer"
     DOUBLE PRECISION = "pointer to DoublePrecision"

     me     = this processor's id (= mxmynd_())
     nprocs = number of allocated processors ( = mxnprc_())
     nvecsA = number of entries in mapA equal to me
                  (= count_list( me, mapA, n ))
     nvecsZ = number of entries in mapZ equal to me
                  (= count_list( me, mapZ, m ))
     sDP    = sizeof( DoublePrecision )


   n....... (input) INTEGER
```

```
         dimension of the matrix A

colA ... (input) array of pointers to DoublePrecision,
                 length (nvecsA)
         The part of matrix A owned by this processer stored
         in packed format, i.e., colA[i] points to the diagonal
         element of the i-th column (or equivalently row) of A
         owned by this processor, i = 0 to nvecsA-1.

mapA ... (input) INTEGER array, length (n)
         The i-th column (or equivalently row) of A is
         owned by processor mapA[i], i = 0 to n-1.

m....... (input) INTEGER
         number of columns of Z, i.e. # of eigenvalues/eigenvectors

colZ ... (input) array of pointers to DoublePrecision,
                 length (nvecsZ)
         The part of matrix Z owned by this processer, stored
         in packed format, i.e., colZ[i] points to the start
         of the i-th column of matrix Z owned by this
         processor, i = 0 to nvecsZ-1.

mapZ ... (input) INTEGER array, length (m)
         The i-th column of matrix Z is owned by processor
         mapZ[i], i = 0 to m-1.

eval.... (input) DOUBLE PRECISION array, length (m)
         the eigenvalues

ibuffptr (workspace) array of pointers to DoublePrecision,
                     length (2 * nvecsZ + 1)

iwork... (workspace) INTEGER array,
                     length( m + maximum( nprocs, n+nvecsA, i_mxm88 ))
                     where i_mxm88 = 2*(n+m)+nvecsA+nvecsZ

work.... (workspace) DOUBLE PRECISION array,
                     length( nvecsZ * n + maximum( d_mxm88,
                                          n + 1 + mxlbuf_() / sDP + 1 )
                     where d_mxm88 = maximum ( (nvecsZ+1)*n,
                                           mxlbuf_()/sDP + 1 )

res..... (output) DOUBLE PRECISION
         the residual described above.

info.... (output) INTEGER
         = 0, not currently used
```

```
 */

void residual( n, colA, mapA, colB, mapB, m, colZ, mapZ, eval,
                ibuffptr, iwork, work, res, info)
     Integer *n, *mapA, *mapB, *m, *mapZ, *iwork, *info;
     DoublePrecision **colA, **colB, **colZ, *eval, **ibuffptr, *work, *res;

/*

   This subroutine computes the residual

   res = max_(i) | A z_(i) - \lambda_(i) B z_(i) |/( | A | * ulp )

   where

   A is an n-by-n symmetric matrix, in packed storage format,
   column (or equivalently row) distributed

   B is an n-by-n symmetric matrix, in packed storage format,
   column (or equivalently row) distributed

   (lambda_i, z_i) is a generalized eigen-pair and
   Z is an n-by-m matrix of eigenvectors, in packed storage format,
   column distributed

   ULP = (machine precision) * (machine base)

   |A z_(i) ... | is the infinity-norm,
   |A| is the 1-norm of A,

   res is reasonable if it is of order about 50 or less.

   Arguments
   ---------
   In the following:

     INTEGER          = "pointer to Integer"
     DOUBLE PRECISION = "pointer to DoublePrecision"

     me     = this processor's id (= mxmynd_())
     nprocs = number of allocated processors ( = mxnprc_())
     nvecsA = number of entries in mapA equal to me
              (= count_list( me, mapA, n ))
     nvecsB = number of entries in mapB equal to me
              (= count_list( me, mapB, n ))
     nvecsZ = number of entries in mapZ equal to me
              (= count_list( me, mapZ, m ))
     sDP    = sizeof( DoublePrecision )
```

```
n....... (input) INTEGER
         size of the symmetric matrices A and B,
         and the number of rows in Z.

colA ... (input) array of pointers to DoublePrecision,
                 length (nvecsA)
         The part of matrix A owned by this processer stored
         in packed format, i.e., colA[i] points to the diagonal
         element of the i-th column (or equivalently row) of A
         owned by this processor, i = 0 to nvecsA-1.

mapA ... (input) INTEGER array, length (n)
         The i-th column (or equivalently row) of A is
         owned by processor mapA[i], i = 0 to n-1.

colB ... (input) array of pointers to DoublePrecision,
                 length (nvecsB)
         The part of matrix B owned by this processer stored
         in packed format, i.e., colB[i] points to the diagonal
         element of the i-th column (or equivalently row) of B
         owned by this processor, i = 0 to nvecsB-1.

mapB ... (input) INTEGER array, length (n)
         The i-th column (or equivalently row) of B is
         owned by processor mapB[i], i = 0 to n-1.

m....... (input) INTEGER
         number of columns of Z, i.e., # of eigenvalues/eigenvectors

colZ ... (input) array of pointers to DoublePrecision,
                 length (nvecsZ)
         The part of matrix Z owned by this processer, stored
         in packed format, i.e., colZ[i] points to the start
         of the i-th column of matrix Z owned by this
         processor, i = 0 to nvecsZ-1.

mapZ ... (input) INTEGER array, length (m)
         The i-th column of matrix Z is owned by processor
         mapZ[i], i = 0 to m-1.

eval.... (input) DOUBLE PRECISION array, length (m)
         the eigenvalues

ibuffptr (workspace) array of pointers to DoublePrecision,
                     length (3 * nvecsZ + 3)

iwork... (workspace) INTEGER array,
                     length( m + nvecsA+nvecsB+nvecsZ+
                             MAX( i_mxm88, nprocs, nvecsA + n )
```

```
                         where i_mxm88 = 2*(n+m)+nvecsA+nvecsZ+nvecsB

   work.... (workspace) DOUBLE PRECISION array,
                         length( 2*nvecsZ * n + maximum( d_mxm88,
                                                n + 1 + mxlbuf_() / sDP + 1 )
                         where d_mxm88 = maximum ( (nvecsZ+1)*n,
                                                  mxlbuf_()/sDP + 1 )

   res..... (output) INTEGER
            the residual described above.

   info.... (output) INTEGER
            = 0, not currently used

 */

void ortho( n, m, colZ, mapZ, ibuffptr, iwork, work, ort, info)
     Integer *n, *m, *mapZ, *iwork, *info;
     DoublePrecision **colZ, **ibuffptr, *work,  *ort;

/*

   This subroutine computes the infinity-norm measure:

   ort = max_i | (Z^t.Z)_i - I_i | / ULP,

   for the standard symmetric eigensystem problem where

     Z is N-by-M
     I is M-by-M.
     Z_i is an eigenvector,
     (Z^t.Z)_i is the i-th column of Z^t.Z
     I_i is the i-th column of the m-by-m identity matrix
     ULP = (machine precision) * (machine base)
     |.| is the infinity-norm.

   res is reasonable if it is of order about 50 or less.


   MUST have M <= N.  If M > N then program exits.
   This is not a limitation of this subroutine as M > N
   implies the columns of Z are linearly dependent, which
   implies "ort" will always be large in this case.

   Arguments
   ---------
   In the following:

     INTEGER          = "pointer to Integer"
     DOUBLE PRECISION = "pointer to DoublePrecision"
```

```
      me     = this processor's id (= mxmynd_())
      nprocs = number of allocated processors ( = mxnprc_())
      nvecsZ = number of entries in mapZ equal to me
                   (= count_list( me, mapZ, m ))
      nvecsZ_max = maximum number of entries in mapZ equal to i,
                   i = 0 to nprocs-1
                   (= max over i of count_list( i, mapZ, m ))
      sDP    = sizeof( DoublePrecision )


    n....... (input) INTEGER
             Number of rows in Z

    m....... (input) INTEGER
             number of columns in Z (i.e., # of
             eigenvalues/eigenvectors).
             Must have m <= n.

    colZ ... (input) array of pointers to DoublePrecision,
                     length (nvecsZ)
             The part of matrix Z owned by this processer, stored
             in packed format, i.e., colZ[i] points to the start
             of the i-th column of matrix Z owned by this
             processor, i = 0 to nvecsZ-1.

    mapZ ... (input) INTEGER array, length (m)
             The i-th column of matrix Z is owned by processor
             mapZ[i], i = 0 to m-1.

    ibuffptr (workspace) array of pointers to DoublePrecision,
                         length(nvecsZ)

    iwork... (workspace) INTEGER array, length( 7 * m )

    work.... (workspace) DOUBLE PRECISION array,
                         length( nvecsZ * n + maximum( d_mxm25,
                                                 mxlbuf_() / sDP + 1 )
                         where d_mxm25 = maximum ( 2*m,
                                                (nvecsZ + 2*nvecsZ_max)*n )

    ort..... (output) INTEGER
             the residual described above.

    info.... (output) INTEGER
             = 0, not currently used

   */

  void b_ortho ( n, colB, mapB, m, colZ, mapZ, ibuffptr, iwork, work, ort, info)
```

```
      Integer *n, *mapB, *m, *mapZ, *iwork, *info;
      DoublePrecision **colB, **colZ, **ibuffptr, *work,  *ort;

/*
   This subroutine computes the  infinity-norm measure:

   ort = max_i | (Z^t.B.Z)_i - I_i | / ULP,

   for the generalized symmetric eigensystem problem where

     Z is N-by-M and distributed by columns
     B is N-by-N symmetric and in packed storage
     I is M-by-M.
     Z_i is an eigenvector,
     (Z^t.B.Z)_i is the i-th column of Z^t.B.Z
     I_i is the i-th column of the m-by-m identity matrix
     ULP = (machine precision) * (machine base)
     |.| is the infinity-norm.

   res is reasonable if it is of order about 50 or less.

   MUST have M <= N.  If M > N then program exits.
   This is not a limitation of this subroutine as M > N
   implies the columns of Z are linearly dependent, which
   implies "ort" will always be large in this case.

   Arguments
   ---------
   In the following:

     INTEGER          = "pointer to Integer"
     DOUBLE PRECISION = "pointer to DoublePrecision"

     me     = this processor's id (= mxmynd_())
     nprocs = number of allocated processors ( = mxnprc_())
     nvecsZ = number of entries in mapZ equal to me
                  (= count_list( me, mapZ, n ))
     nvecsZ_max = maximum number of entries in mapZ equal to i,
                  i = 0 to nprocs-1
                  (= max over i of count_list( i, mapZ, n ))
     sDP    = sizeof( DoublePrecision )


   n....... (input) INTEGER
            Number of rows and columns in B, and
            the number of rows in Z

   colB ... (input) array of pointers to DoublePrecision,
                    length (nvecsB)
                  The part of matrix B owned by this processer stored
                  in packed format, i.e., colB[i] points to the diagonal
                  element of the i-th column (or equivalently row) of B
                  owned by this processor, i = 0 to nvecsB-1.

         mapB ... (input) INTEGER array, length (n)
                  The i-th column (or equivalently row) of B is
                  owned by processor mapB[i], i = 0 to n-1.

         m....... (input) INTEGER
                  number of columns in Z (i.e., # of
                  eigenvalues/eigenvectors).
                  Must have m <= n.

         colZ ... (input) array of pointers to DoublePrecision,
                          length (nvecsZ)
                  The part of matrix Z owned by this processer, stored
                  in packed format, i.e., colZ[i] points to the start
                  of the i-th column of matrix Z owned by this
                  processor, i = 0 to nvecsZ-1.

         mapZ ... (input) INTEGER array, length (m)
                  The i-th column of matrix Z is owned by processor
                  mapZ[i], i = 0 to m-1.

         ibuffptr (workspace) array of pointers to DoublePrecision,
                              length( 3*nvecsZ + 1)

         iwork... (workspace) INTEGER array,
                              length( n+m + MAX(nprocs, 3*n+2*m+nvecsB+nvecsZ )

         work.... (workspace) DOUBLE PRECISION array,
                              length( nvecsZ * (n+m) + maximum( d_mxm25, d_mxm88,
                                                                mxlbuf_() / sDP + 1 )
                              where d_mxm25 = maximum( n+m,
                                                       (nvecsZ + 2*nvecsZ_max)*n )
                                    d_mxm88 = maximum ( (nvecsZ+1)*n,
                                                        mxlbuf_()/sDP + 1 )

         ort..... (output) INTEGER
                  the residual described above.

         info.... (output) INTEGER
                  = 0, not currently used
      */
```

5 Utilites

PeIGS has a few utilites that may assist you in allocating scratch space, find array index of matrix elements stored in 1-D arrays.

5.1 FMEMREQ and memreq_: Scratch Memory Size

Since scratch space may vary depending on data distribution of the matrices, PeIGS provides a subroutine **FMEMREQ** and **memreq_** for determining on each processor the amount of scratch storage space that is required to complete the computation safely.

Currently, **FMEMREQ** and **memreq_** only provide the scratch memory space requirment information for the general, the standard, and the tridiagonal symmetric eigensystem drivers (i.e., the level I routines).

The F77 calling syntax for FMEMREQ is:

```
          subroutine FMEMREQ ( type, n, mapA, mapB, mapZ, isize,
       $     rsize, ptrsize, iwork)
          integer type, n, isize, rsize, ptrsize
          integer mapA(*), mapB(*), mapZ(*), iwork(*)
c
c    input:
c    ------
c    type (input/integer)
c         return memory requirements for:
c         = 0    the generalized eigensystem problem (pdspgvx or pdspgv )
c         = 1    the standard eigensystem problem (pdspevx or pdspev )
c         = 2    the tri-diagonal standard eigensystem problem (pdsptri )

c    n: size of the A and B matrices

c   mapA : integer array of size n;
c   mapA[i], 0< i <= n, is the processor holding this vector

c     mapB : integer array of size n;
c     mapB[i], 0< i <= n, is the processor holding this vector
c              not used unless type = 0.

c    mapZ : integer array of size n
c    mapZ[i], 0< i <= n, is the processor holding this vector

c    output:

c    isize = size of integer workspace required
c    rsize = size of Double precision workspace required
c    ptr_size = size of secondary Double precision workspace required
c               this will be converted to an array of
c               "pointers to double precision"
c
c    iscratch ... integer scratch space of length 3*n
```

The C calling syntax for **memreq_** is:

```
void memreq_(type, n, mapA, mapB, mapZ, isize, rsize, ptr_size, iscratch )
     Integer *type, *n, *mapA, *mapB, *mapZ, *isize, *rsize, *ptr_size, *iscratch;

/*
    this subroutine computes the memory requirements for this processor to
    safely setup the eigensystem problem.  On output, isize, risize, and
    ptr_size
    are the sizes ( in the respective data types ) of the data needed
    by the eigensystem
    program.  It performs the basic error checks and assumes that the user has
    input the correct information into type, n, mapA, mapB, mapZ

    input:
                 memory for

   *type = 0    the generalized eigensystem problem (pdspgvx or pdspgv )
         = 1    the standard eigensystem problem (pdspevx or pdspev )
         = 2    the tri-diagonal standard eigensystem problem (pdsptri )

     n: size of the matrix

     mapA : integer array of size n;
     mapA[i], 0<= i < n, is the processor holding this vector

     mapB : integer array of size n;
     mapB[i], 0<= i < n, is the processor holding this vector
              not used unless type = 0

    mapZ : integer array of size n
    mapZ[i], 0<= i < n, is the processor holding this vector

    output:

    isize = size of integer workspace required
    rsize = size of Double precision workspace required
    ptr_size= size of array of "pointer to DoublePrecision" workspace

    iscratch ... integer scratch space of length 3*n

    at this point one can allocate the required memory using C
    or using off-sets from Fortran arrays
*/
```

5.2 General Utilities

There are 3 general utilities that may be of use. In F77: **CI_ENTRY**, **CI_SIZE**, **FIL_MAPVEC**. In C, **ci_entry**, **ci_size_**, and **fil_mapvec_**.

Their use is listed below for the fortran version. The analogous description applies to the C version.

- CI_ENTRY :returns index a(i,j) in Fortran for a 1-D data array storing the lower triangular part of a symmetric matrix
- CI_SIZE: returns the total memory required for "standard" column wrapping of a symmetric matrix
- FIL_MAPVEC: from map construct a correspondance mapvec: the real column/row index that the matrix on this processor owns

5.2.1 CI_ENTRY and ci_entry_

```
          integer function ci_entry (me, n, i, j, map)
          integer me, n, i, j, map(*)
c
c    PeIGS utility routine
c
c    this routine returns the F77 index of the a(i,j)
c    for a symmetric matrix with i >= j
c    on processor me stored using a 1-D array
c
c    if (processor = me)
c    doesn't own this element -1 is returned
c
c    Argument:
c
c    me        = (input/integer) processor id
c
c    n          = (input/integer) dimension of the matrix
c
c    i           = (input/integer) the row index of the element a[j][i]
c
c    j           = (input/integer) the column index of the element a[j][i]
c
c    map      = (input/integer array ) integer array of length n
c    map[j] = the processor id holding column j
c
int ci_entry (me, n, i, j, map)
     int *me, *n, *i, *j, *map;
  /*
    PeIGS utility routine

    this routine returns the C index of the a(i,j)
    for a symmetric matrix with i >= j
    on processor me stored using a 1-D array

    if (processor = me)
    doesn't own this element -1 is returned
    Argument:

    me        = (input/integer) processor id

    n          = (input/integer) dimension of the matrix

    i           = (input/integer) the row index of the element a[j][i]

    j           = (input/integer) the column index of the element a[j][i]

    map      = (input/integer array ) integer array of length n
    map[j] = the processor id holding column j

    */
        integer function ci_size_ (me, n, map)
        integer me, n, map(*)
c
c    Fortran and C callable ( not very portable )
c
c    returns the total number of double precision
c    storage location required for storing the lower
c    triangular part of a symmetric matrix distributed
c    according the map
c
c    me        = (input/integer) processor id
c
c    n          = (input/integer) dimension of the matrix and also
c                  the length of the array map
c
c    map(*)   = (input/integer array) length n array of processor ids
c
int ci_size_ (me, n, map)
     int *me, *n, *map;
  /*
    Fortran and C callable

    returns the total number of double precision
    storage location required for storing the lower
    triangular part of a symmetric matrix distributed
    according the map

    me        = (input/integer) processor id

    n          = (input/integer) dimension of the matrix and also
    the length of the array map

    map(*)   = (input/integer array) length n array of processor ids

    */
```

```
        integer function fil_mapvec ( me, n, map, mapvec)
        integer me, n, map, mapvec(*)
c
c    from the map array construct a shorter
c    array with information about the vectors stored on this processor
c
c       may be compatibility problems between C and fortran
c
c    mapvec(i) = j
c    i = the i-th vector stored on this processor
c    j = the real column/row indx of the i-th vector
c
c    returns the number of vectors on this processor given
c    by map and also the mapvec list
c
c    argument:
c
c    me         = (input/integer) node id
c
c    n           = (input/integer) dimension of the matrix
c
c    map       = (input/integer array) distribution of the columns of the matrix
c                                length n
c
c    mapvec  = (output/integer array) length n
c

int fil_mapvec_ ( me, n, map, mapvec)
     int *me, *n, *map, *mapvec;
  /*
    from the map array construct a shorter
    array with information about the vectors stored on this processor

    mapvec[i] = j
    i = the i-th vector stored on this processor
    j = the real column/row indx of the i-th vector

    returns the number of vectors on this processor given
    by map and also the mapvec list

    argument:

    me         = (input/integer) node id

    n           = (input/integer) dimension of the matrix

    map       = (input/integer array) distribution of the columns of the matrix
                                length n

    mapvec  = (output/integer array) length n
    */

void sfnorm( n, colA, mapA, norm, iwork, work, info)
     Integer *n, *mapA, *iwork, *info;
     DoublePrecision **colA, *work, *norm;

  /*
    computes the F-norm of a symmetric n-by-n matrix A, where

    F-norm(A) = sqrt( sum_i,j a_i,j^2 )

    n       = size of the matrix
    colA    = DoublePrecision pointer to the columns
    mapA    = distribution of the columns
    iwork   = integer scratch space
    work    = DoublePrecision precision scratch space

    */
```

6 Error Handling

Since the PeIGS routines are designed to work with lists of processors, and this list of processors may not be all the processors allocated – there may two PeIGS routines running on two different sets of processors simultaneously – global checking of data does not work. Instead, the set of processor ids appearing in mapA, mapB, and/or mapZ are used to determine a "master" processor list. PeIGS then tries to have all processors in this "master" list return the same INFO whenever possible.

All of the level 1 and **some of the level 2 routines** do extensive checking of the input data. In particular, they make sure that the input data on a processor is "vaild". If any input data is invalid on a processor, then that processor prints a negative value of INFO to the standard error output unit and terminates program execution. In this case other processors are not informed of the error and may continue program execution. If some processors have invalid data and some have valid data, then some of the processors will continue running and will have to be killed by the user.

If all of the input data on each processor is valid, then the processors in the "master" processor list compare their data to make sure it is consistent across the processors. If the input data is not consistent on all processors, then the processors with incosisent data print a negative INFO and an error message to the standard error output unit indicating the nature of the problem to standard error. With inconsistent data all processors in the "master" list try to return the same value of INFO and terminate program execution together. However, with some types of data inconsistencies it is not possible to exit gracefully and some processors will continue running. These processors must be killed by the user.

All PeIGS routines return a positive INFO error code if something goes wrong in the computation. The value of INFO should be the same on all participating processors and can be used to determine what went wrong. Program execution is not stopped for positive INFOs.

7 BUGS

In this chapter we describe some possible problems associated with our implementation of the PeIGS eigensolver.

1) If the user wishes to solve only for a fraction of the eigen-pairs the current code asks for (but needs only the minimum memory space) the same memory space as for a full eigen-system solve to satisfy the error checking routines and to prevent race condition. We are working on elimanating this restriction.

2) The inverse iteration with modified Gram Schmidt current does a fixed number of combined iterations. This process can theoretically fail to converge to an eigenvector on a degenerate eigen-sub-space.

3) The error reporting is being done on all processors. One should just make processor 0 report errors.

4) The C codes calling C code needs a fortran driver so that the mxsubs.cpp codes get initialized correctly...SUN common block is ...

5) When solving a tridiagonal problem with n = 2053 on four processors of an Intel PARAGON we sporadically get large residuals. The problem is in mxm25, the eigenvector back-transformation. We have not been able to find any problems with the PeIGS code, and believe that the errors occuring here are caused by a system problem having to do with the use of "virtual" memory on the nodes (this problem exceeds the physical memory available on a node).

We appreciate any advice and reports of bugs. We ask the user to mail us all of the input parameters and the parallel (or serial) environment (as well as) operating system version number.

There are likely to be unknown bugs in the code and we strongly recommend that the user run the residual and orthogonality checking routines to check the results of computations whenever practical.

Index

Table of Contents