AICS TECHNICAL REPORT

N0. 2015-002

USER'S MANUAL

EIGEN EXA

VERSION 2.3C

BY

EIGEN EXA DEVELOPMENT GROUP

LARGE-SCALE PARALLEL NUMERICAL COMPUTING TECHNOLOGY RESEARCH TEAM

RIKEN ADVANCED INSTITUTE FOR COMPUTATIONAL SCIENCE

SUBMITTED ON 28/08/2015

ACCEPTED ON 04/09/2015

Published and copyrighted by

RIKEN Advanced Institute for Computational Science (AICS)

7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, 650-0047, Japan

User's Manual
EigenExa
Version 2.3c

EigenExa Development Group
Large-scale Parallel Numerical Computing Technology Research Team
RIKEN Advanced Institute for Computational Science

24 June 2015

Contents

1 Introduction **5**
- 1.1 EigenExa and its course of development 5
- 1.2 License for use and copyright 6

2 Before use **7**
- 2.1 Software required for EigenExa installation 7
- 2.2 Obtaining EigenExa 7
- 2.3 Compile and install procedure 7

3 Quick tutorial **9**

4 API **13**
- 4.1 `eigen_init` 13
- 4.2 `eigen_free` 13
- 4.3 `eigen_get_blacs_context` 14
- 4.4 `eigen_sx` 14
- 4.5 `eigen_s` 14
- 4.6 `eigen_get_version` 15
- 4.7 `eigen_show_version` 15
- 4.8 `eigen_get_matdims` 15
- 4.9 `eigen_memory_internal` 16
- 4.10 `eigen_get_comm` 16
- 4.11 `eigen_get_procs` 16
- 4.12 `eigen_get_id` 17
- 4.13 `eigen_loop_start` 17
- 4.14 `eigen_loop_end` 17
- 4.15 `eigen_translate_l2g` 17
- 4.16 `eigen_translate_g2l` 18
- 4.17 `eigen_owner_node` 18
- 4.18 `KMATH_EIGEN_GEV` 18

5 Other key considerations **19**
- 5.1 Caution regarding compatibility 19
- 5.2 Binding with other languages 19
- 5.3 Behavior on error occurrence 19
- 5.4 Shared library handling in versions 1.x 19

A Algorithm overview **21**
A.1 Introduction 21
A.2 Various approaches and related projects 21
A.3 eigen_s 22
A.4 eigen_sx 23
A.5 Differences between eigen_s and eigen_sx 24
A.6 Conclusion 25

Acknowledgements **27**

References **29**

Chapter 1

Introduction

1.1 EigenExa and its course of development

EigenExa is a high-performance numerical eigenvalue solver. It traces its history back to EigenES (code name; no formal name) [1] developed on the Earth Simulator, which attained the rank of number one in the world of such systems in 2002. EigenES was nominated for a Gordon Bell Prize at SC 2006 and today continues to serve as an eigenvalue solver on large-scale PC clusters. This led to the initiation of EigenK [2, 3] development around 2008. The EigenK library became the immediate predecessor of EigenExa, and in August 2013 EigenK was renamed EigenExa and public release was begun, following entry into operation on the K computer[4, 5]. EigenExa development continues, with the underlying objective being to achieve an eigenvalue library scalable to operate on future post-petascale ("exa" or "extreme") computer systems.

In its present release (version 2.3c), EigenExa provides the simplest function of computing all eigenpairs (eigenvalues paired with their respective eigenvectors) for both standard and generalized eigenvalue problems. As reported elsewhere [2, 3], EigenExa applies both classical and advanced algorithms in the same basic manner as EigenK, and thereby reduces the required computation time for diagonalization.

The development of EigenExa includes the utilization of various parallel programming languages and libraries, encompassing MPI, OpenMP, high-performance BLAS, and SIMD vectorized Fortran90 compiler techniques. EigenExa is expected to open the way for high-performance computation through multiple simultaneous functions characterized by the following.

1. Inter-node parallelism in distributed memory architecture, by MPI
2. Parallelism in shared-memory parallel computers and multicore processors, by OpenMP
3. High parallelism utilizing BLAS highly optimized by vendors
4. SIMD or coarse-grained parallelism utilizing vendor-provided high-performance compilers

The good features of Fortran90 are also actively incorporated into EigenExa. The API of EigenExa is more flexible than libraries implemented in Fortran77, and it provides a user-friendly interface, based on modular interfaces and optional parameters. Data distribution is limited to two-dimensional cyclic division, the processor map can be specified in almost any arbitrary configuration, and compatibility and consistency with existing numerical computation libraries are guaranteed if the data redistribution function provided by ScaLAPACK is used. EigenExa also permits user interface specification (or omission) for heightened performance, such as block parameters that strongly affect execution performance.

In the parallel performance of the library itself, it achieves heightened performance by reducing the EigenK communication overhead, and it has been shown that in most cases EigenExa performance exceeds that of EigenK, ScaLAPACK, and others of the highest-level numerical computation libraries [3].

Today, EigenExa is in operation on many HPC platforms, including the K computer and its Fujitsu PRIMEHPC FX10 commercial variant, various cluster computers using Intel x86 series processors, IBM Blue/Gene Q systems, and the NEC vector computer SX series systems. Furthermore, reports on EigenExa have been presented at scientific conferences [6, 7, 8, 9], so if necessary, please refer to them.

This user's manual for EigenExa version 2.3c covers procedures from installation to actual use, with particular consideration given to installation and compiling, a quick tutorial, the API list, and EigenK compatibility. It is written and provided with the hope of all EigenExa team developers that it will be helpful for achieving efficient parallel simulations by many users.

1.2 License for use and copyright

Permission to use KMATH_RANDOM is granted on the basis of the BSD 2-Clause License (found in LICENCE.txt in the library).

LICENCE.txt


```
Copyright (C) 2012- 2014 RIKEN.
Copyright (C) 2011- 2012 Toshiyuki Imamura
 Graduate School of Informatics and Engineering,
 The University of Electro-Communications.
Copyright (C) 2011- 2014 Japan Atomic Energy Agency.

-----------------------------------------------------------------------
 Copyright notice is from here
-----------------------------------------------------------------------
  Redistribution  and  use  in  source and binary forms, with or without
  modification,  are  permitted  provided  that the following conditions
  are met:

  * Redistributions  of  source  code  must  retain  the above copyright
    notice,  this  list  of  conditions  and  the  following  disclaimer.
  * Redistributions  in  binary  form must reproduce the above copyright
    notice,  this list of conditions and the following disclaimer in the
    documentation  and/or other materials provided with the distribution.

  THIS  SOFTWARE  IS  PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  ``AS IS''  AND  ANY  EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED  TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A  PARTICULAR  PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL,  EXEMPLARY,  OR  CONSEQUENTIAL  DAMAGES  (INCLUDING,  BUT NOT
  LIMITED  TO,  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA,  OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY  OF  LIABILITY,  WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  OF  THIS  SOFTWARE,  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
-----------------------------------------------------------------------
```

Chapter 2

Before use

2.1 Software required for EigenExa installation

Several software packages are needed to compile the EigenExa library. BLAS, LAPACK, ScaLAPACK, and MPI must be installed in the system before EigenExa is compiled. To present, it has been confirmed that EigenExa can be compiled with the following libraries.

BLAS	Intel MKL, GotoBLAS, OpenBLAS, ATLAS
	Fujitsu SSL II, IBM ESSL, NEC MathKeisan
LAPACK	Version 3.4.0 or later
ScaLAPACK	Version 1.8.0 or later
MPI	MPICH2 version 1.5 or later, MPICH version 3.0.2 or later
	OpenMPI version 1.6.4 or later, MPI/SX

2.2 Obtaining EigenExa

All available information on EigenExa can be obtained at the following URL.

`http://www.aics.riken.jp/labs/lpnctrt/EigenExa.html`

Tarball distribution is also performed via this URL. Planning is in progress for provision of further information on EigenExa.

2.3 Compile and install procedure

Several steps are necessary to compile the EigenExa library. Proceed as described in the following installation guideline.

Decompression and extraction First, unpack the tarball on the working directory and then move to the EigenExa-2.3c directory

```
% tar zxvf EigenExa-2.3c.tgz
% cd EigenExa-2.3c
```

Environment setting Second, edit `Makefile` and `make_inc.xxx` to match the user's environment. For xxx, select and enter the character string corresponding to the compiler to be used, from among the following:

1. `BX900`
2. `Intel` or `Intel.shared`
3. `K_FX10` or `K_FX10.shared`
4. `gcc`
5. `BlueGeneQ`
6. `SX`

The ".shared" suffix is specified in creating a shared library.

make Third, execute `make`. This results in creation of the static library `libEigenExa.a` or the shared library `libEigenExa.so`.

```
% make
```

install Lastly, copy the library itself, `libEigenExa.a` (or `libEigenExa.so` for the shared library), `eigen_libs.mod`, and `eigen_blacs.mod` to the install directory, and end the procedure. To install `/usr/local/lib`, for example,

```
% cp libEigenExa.a eigen_libs.mod eigen_blacs.mod  /usr/local/lib/
```

generalized eigenvalue computation driver routine When the generalized eigenvalue driver routine KMATH_EIGEN_GEV with the same number as the installed version is downloaded from

`http://www.aics.riken.jp/labs/lpnctrt/KMATH_EIGEN_GEV.html`

and followed by make, the `KMATH_EIGEN_GEV.o` driver module for generalized eigenvalues is created. Linking this object when program linking enables computation of generalized eigenvalues. At present, this function is a higher-level independent module used as the EigenExa eigenvalue computation engine. In this manual, it is described in terms of its relation to EigenExa.

Chapter 3

Quick tutorial

The standard benchmark code can be obtained by moving to the working directory and executing 'make benchmark'. The source code components 'main2.F' and 'Makefile' should be useful for code creation. The kernel of main2.f is as follows.

main2.f

```
use MPI
use eigen_libs
...
call MPI_Init_thread( MPI_THREAD_MULTIPLE, i, ierr )
call eigen_init( )

N=10000; mtype=0

call eigen_get_matdims( N, nm, ny )
allocate ( A(nm,ny), Z(nm,ny), w(N) )
call mat_set( N, a, nm, mtype )
call eigen_sx( N, N, a, nm, w, z, nm, m_forward=32, m_backward=128)
deallocate ( A, Z, w )
...
call eigen_free( )
call MPI_Finalize( ierr )
end
```

The above code represents only the framework and does not actually operate, but is sufficient to indicate the overall flow from initialization to sequence assurance to eigenvalue computation to termination procedure.

In the above example, the initialization function `eigen_init()` is executed by a parameter-omitted call. In `eigen_init()`, the group that executes the eigenvalue computation can be specified as a communicator by the form `comm=XXX`. For parallel execution of eigenvalue computation simultaneously by multiple groups, parallel computation is enabled by passing the communicator created by `MPI_Comm_split()`, for example. Note that a constraint exists in the current implementation such that performance of collective operation with respect to the communicator `MPI_COMM_WORLD` within `eigen_init()` requires that `eigen_init()` be called by all processes belonging to `MPI_COMM_WORLD` simultaneously. Since a different communicator can be specified for each individual process, a process not participating in the eigenvalue computation can therefore have `MPI_COMM_NULL` specified for `eigen_init()`, thus having the call skipped for the eigenvalue computation driver `eigen_sx()` itself. In short, it is possible to perform

simultaneous execution of various operations other than `eigen_sx()` processing that include `eigen_sx()`.

MPI_Comm_split and MPI_COMM_NULL

```
if ( my_rank < 10 ) then
   comm = MPI_COMM_SELF
else
   comm = MPI_COMM_NULL
endif
call eigen_init( comm )
call eigen_sx( .... )
```

In EigenExa, processes belonging to a communicator specified by `eigen_init()` are deployed and used in a two-dimensional process grid. EigenExa is designed to utilize a process grid that is nearly square in shape, to reduce communication traffic to the minimum possible. To heighten user convenience, EigenExa has been developed so that the two-dimensional Cartesian grid adopted by MPI can be specified to `comm`. If the Cartesian shape is two dimensional, then in principle EigenExa can be called and computation performed for any process array, thus enabling execution of complicated parallel processing through combination of the above multi-type communicators. Because the Cartesian process grid is essentially row-major, the Cartesian is prioritized in the event of conflict with an `order='C'` specification. For historical reasons, the default process grid in EigenExa is column-major.

The generation of matrix data is performed in `mat_set()`, which is called just before `eigen_sx()`. The matrix data are distributed on the specified two-dimensional process grid in two-dimensional cyclic division style, and are stored in each process as a local array. Because only some of the data are stored for each process, when matrix elements are accessed, a rule for transformation between global and local indices is required.

The following program is an excerpt from `mat_set()` and is presented to compare the program for generation of a Frank matrix with a global counterloop structure and the same but translated to local counterloops.

matset(before)

```
! Global loop program to compute a Frank matrix
do i = 1, n
   do j = 1, n
      a(j, i) = (n+1-Max(n+1-i,n+1-j))*1.0D+00
   end do
end do
```

↓↓↓↓↓↓

matset(after)

```
! Translated local loop program to compute a Frank matrix
use MPI
use eigen_libs
call eigen_get_procs( nnod, x_nnod, y_nnod )
call eigen_get_id   ( inod, x_inod, y_inod )

j_2 = eigen_loop_start( 1, x_nnod, x_inod )
j_3 = eigen_loop_end  ( n, x_nnod, x_inod )
i_2 = eigen_loop_start( 1, y_nnod, y_inod )
i_3 = eigen_loop_end  ( n, y_nnod, y_inod )
do i_1 = i_2, i_3
   i = eigen_translate_l2g( i_1, y_nnod, y_inod )
   do j_1 = j_2, j_3
      j = eigen_translate_l2g( j_1, x_nnod, x_inod )
      a(j_1, i_1) = (n+1-Max(n+1-i,n+1-j))*1.0D+00
   end do
end do
```

The `eigen_loop_start()` and `eigen_loop_end()` are used to transform the loop range. The second and third parameters specify the process number and process ID derived from the communicator, which shows the direction of distribution. In this manual, the correspondence is always [row] → "x" and [column] → "y" (in the case of the communicator for the overall participating process, the "x" and "y" portions are characterless). It is important to note that in EigenExa process IDs are managed as integers, starting with 1. The process ID obtained by the query function `eigen_get_id( )` therefore differs from the MPI rank by 1, and the ID must be reduce by 1 in cases where the MPI rank is required.

In the above program, the local loop counter value is translated to the corresponding global counter value to be used, with `eigen_translate_l2g()` used for the this translation. The second and third parameters should be specified in a manner similar to `eigen_loop_start()`, for example. Conversely, to convert the global counter value to the local counter value, `eigen_translate_g2l()` is used, with the proviso that if the global counter value is viewed as a loop value, then `eigen_translate_g2l()` returns the corresponding local counter value on the process that becomes the owner process (the process where the local counter value corresponding with the global counter value is included in the loop). In order to determine the owner process of the specified global loop counter, `eigen_owner_node()`is used. It should be used when broadcasting in reference to a particular row or column vector value.

Furthermore, when progressing to computation together with ScaLAPACK for personnel at advanced levels, the process grid context used by EigenExa should be obtained via the auxiliary function `eigen_get_blacs_context()`, referring to the `mtype=2` portion of the `mat_set()` function (the following shows the kernel of the `PDTRAN()` call that stores the matrix AS transpose in matrix A).

pdtran

```
! Cooperation with ScaLAPACK
NPROW = x_nnod; NPCOL = y_nnod

ICTXT = eigen_get_blacs_context( )
CALL DESCINIT( DESCA, n, n, 1, 1, 0, 0, ICTXT, nm, INFO )

! A <-- AS^T
CALL PDTRAN( n, n, 1D0, as, 1, 1, DESCA, 1D0, a, 1, 1, DESCA )
```

When compiling, in addition to the use of mpif90, it is necessary to set the path (in most cases, the `-I` option) because of the need to access `eigen_libs` and other modules. To link the EigenExa library, it is also necessary to simultaneously link MPI, OpenMP, ScaLAPACK (if version 1.8 or earlier, then also BLACS), and so forth. In the case of Intel-compiler-based MPI, the procedure is as follows (note that the library names around ScaLAPACK and BLAS vary with the environment).

```
% mpif90 -c a.f -openmp -I/usr/local/include -I/usr/local/lib
% mpif90 -o exe a.o -openmp -L/usr/local/lib -lEigenExa -lscalapack \
  -llapack -lblas
```

Chapter 4

API

This section lists the functions in '`eigen_libs.mod`' that have been assigned a public attribute. The first three routines are the main drivers and the others are utility functions. Parameters having optional attributes attached (written in italics) can be omitted, and can also be specified by TERM=variable *or* constant value in the Fortran format form.

4.1 eigen_init

Initializes the functions of EigenExa. Process grid mapping can be specified via the parameter '`comm`' or '`order`'. This procedure is collective (because collective operation is performed for `MPI_COMM_WORLD` internally in the present version, after which all processes must call this procedure); `comm` can specify a different value for each process group, and when different process groups simultaneously call driver functions (`eigen_sx()` or `eigen_s()`), parallel operations are performed in driver function units. If `comm` is `MPI_COMM_NULL`, calling the handlers `eigen_sx()` and `eigen_s()` results in an immediate return with no internal action. An inter-communicator cannot be used for `comm`.

`subroutine eigen_init(` *`comm`*, *`order`* `)`

1. `integer, optional, intent(IN) :: comm = MPI_COMM_WORLD`
   Base communicator.
   Process grid mapping is enabled when a two-dimensional Cartesian grid is specified for `comm`.
   Note: If omitted, then `MPI_COMM_WORLD`
2. `character*(*), optional, intent(IN) :: order = 'C'`
   Row or Column
   Note: If omitted, then treated as 'C'. If the grid major conflicts with the Cartesian `comm` specification, then 'R' is used.

4.2 eigen_free

Terminates EigenExa functions.

`subsroutine eigen_free(` *`flag`* `)`

1. `integer, optional, intent(IN) :: flag = 0`
   Timer printer flag.
   This parameter is for development, and therefore not ordinarily specified.
   If omitted, then 0.

4.3 eigen_get_blacs_context

Returns the ScaLAPACK (BLACS) context corresponding to the process grid information specified by EigenExa.

```
integer function eigen_get_blacs_context( )
```

4.4 eigen_sx

This is the main EigenExa driver routine. Eigenpairs are computed via transformation to a pentadiagonal matrix. This is a collective operation driver; all processes attributed to the process group calling must participate in the call.

```
subroutine eigen_sx( n, nvec, a, lda, w, z, ldz, \
                     m_forward, m_backward, mode )
```

1. `integer, intent(IN) :: n`
   Matrix and vector dimensions
2. `integer, intent(IN) :: nvec`
   Number of computed eigenvectors
   At present, this option is not supported;
   `eigen_sx()` computes all eigenvectors.
3. `real(8), intent(INOUT) :: a(lda,*)`
   Symmetric matrix to be diagonalized
   Array content is destroyed upon subroutine termination but
   the FLOPS count is stored in `a(1, 1)`.
4. `integer, intent(IN) :: lda`
   Leading dimension of array `a`
5. `real(8), intent(OUT) :: w(n)`
   Eigenvalues in ascending order
6. `real(8), intent(OUT) :: z(ldz,*)`
   Orthogonal eigenvectors of matrix `a`
7. `integer, intent(IN) :: ldz`
   Leading dimension of array `z`
8. `integer, optional, intent(IN) :: m_forward = 48`
   Householder transformation blocksize (must be even number);
   48 when omitted
9. `integer, optional, intent(IN) :: m_backward = 128`
   Householder back transformation blocksize; 128 when omitted
10. `character, optional, intent(IN) :: mode = 'A'`
    'A' : all eigenvalues and corresponding eigenvectors (default)
    'N' : eigenvalues only
    'X' : add to mode A to improve eigenvalue precision

4.5 eigen_s

This is the EigenExa driver routine.

```
subroutine eigen_s( n, nvec, a, lda, w, z, ldz, \
                    m_forward, m_backward, mode )
```

1. `integer, intent(IN) :: n`
   Matrix and vector dimensions
2. `integer, intent(IN) :: nvec`
   Number of computed eigenvectors
   At present, this option is not supported;
   `eigen_s()` computes all eigenvectors.
3. `real(8), intent(INOUT) :: a(lda,*)`
   Symmetric matrix to be diagonalized
   Array content is destroyed upon subroutine termination but
   the FLOPS count is stored in `a(1, 1)`.
4. `integer, intent(IN) :: lda`
   Leading dimension of array `a`
5. `real(8), intent(OUT) :: w(n)`
   Eigenvalues in ascending order
6. `real(8), intent(OUT) :: z(ldz,*)`
   Orthogonal eigenvectors of matrix `a`
7. `integer, intent(IN) :: ldz`
   Leading dimension of array `z`
8. `integer, optional, intent(IN) :: m_forward = 48`
   Householder transformation blocksize; 48 when omitted
9. `integer, optional, intent(IN) :: m_backward = 128`
   Householder back transformation blocksize; 128 when omitted
10. `character, optional, intent(IN) :: mode = 'A'`
   'A' : all eigenvalues and corresponding eigenvectors (default)
   'N' : eigenvalues only
   'X' : add to mode A to improve eigenvalue precision

4.6 eigen_get_version

Returns the version information of EigenExa.

```
subroutine eigen_get_version( version, data, vcode )
```

1. `integer, intent(OUT) :: version`
   Version number in three digits.
   The digits indicate major version, minor version, and patch level from the highest.
2. `character, intent(OUT) :: date`
   Release date.
3. `character, intent(OUT) :: vcode`
   Code name of the version.

4.7 eigen_show_version

Print the version information of EigenExa to standard output.

```
subroutine eigen_show_version()
```

4.8 eigen_get_matdims

Returns the array size recommended by EigenExa. It is desirable for the user to dynamically allocate the local array using the array dimensions (nx,ny) obtained by this function or a larger

numeric value. The overall matrix is (CYCLIC, CYCLIC) divided.

```
subroutine eigen_get_matdims( n, nx, ny )
```

1. `integer, intent(IN) :: n`
   Matrix dimensions
2. `integer, intent(OUT) :: nx`
   Lower limit of leading dimensions of arrays `a` and `z`
3. `integer, intent(OUT) :: ny`
   Lower limit of second indices of arrays `a` and `z`

4.9 eigen_memory_internal

This function returns the dynamically allocated internal memory size, while EigenExa is called. The user should ascertain the value returned by this function to ensure avoidance of memory shortage. The type of the return value is changed to `integer(8)` in version 2.3c. If the return value is -1 (a negative value), this funtion warns of possible integer overflow resulting from too large matrix dimension in EigenExa

```
integer(8) function eigen_memory_internal( n, lda, ldz, m1, m0 )
```

1. `integer, intent(IN) :: n`
   Matrix dimensions
2. `integer, intent(IN) :: lda`
   Leading dimension of array `a`
3. `integer, intent(IN) :: ldz`
   Leading dimension of array `z`
4. `integer, intent(IN) :: m1`
   Householder transformation blocksize (must be even number)
5. `integer, intent(IN) :: m0`
   Householder transformation blocksize

4.10 eigen_get_comm

Returns the MPI communicator generated by `eigen_init()`.

```
subroutine eigen_get_comm( comm, x_comm, y_comm )
```

1. `integer, intent(OUT) :: comm`
   Base communicator.
2. `integer, intent(OUT) :: x_comm`
   Row communicator, attribute of all processes with matching row id.
3. `integer, intent(OUT) :: y_comm`
   Column communicator, attribute of all processes with matching column id.

4.11 eigen_get_procs

Returns information on the number of processes related to the communicator generated by `eigen_init()`.

```
subroutine eigen_get_procs( procs, x_procs, y_procs )
```

1. `integer, intent(OUT) :: procs`
   Number of processes in `comm`
2. `integer, intent(OUT) :: x_procs`
   Number of processes in `x_comm`
3. `integer, intent(OUT) :: y_procs`
   Number of processes in `y_comm`

4.12 eigen_get_id

Returns information on the process ID related to the communicator generated by `eigen_init()`. Here, the process ID differs from the MPI rank; it is an integer value starting from 1 in the relation MPI rank = process ID − 1.

```
subroutine eigen_get_id( id, x_id, y_id )
```

1. `integer, intent(OUT) :: id`
   Process ID defined by `comm`
2. `integer, intent(OUT) :: x_id`
   Process ID defined by `x_comm`
3. `integer, intent(OUT) :: y_id`
   Process ID defined by `y_comm`

4.13 eigen_loop_start

Returns the loop starting value in the local loop structure corresponding to the specified global loop starting value.

```
integer function eigen_loop_start( istart, nnod, inod )
```

1. `integer, intent(IN) :: istart`
   Global loop starting value
2. `integer, intent(IN) :: nnod`
   Process number
3. `integer, intent(IN) :: inod`
   Process ID

4.14 eigen_loop_end

Returns the terminal value of the loop in the local loop structure corresponding to the terminal value of the specified global loop.

```
integer function eigen_loop_end( iend, nnod, inod )
```

1. `integer, intent(IN) :: istart`
   Global loop terminal value
2. `integer, intent(IN) :: nnod`
   Process number
3. `integer, intent(IN) :: inod`
   Process ID

4.15 eigen_translate_l2g

```
integer function eigen_translate_l2g( ictr, nnod, inod )
```

1. `integer, intent(IN) :: ictr`
   Local counter
2. `integer, intent(IN) :: nnod`
   Process number
3. `integer, intent(IN):: inod`
   Process ID

4.16 eigen_translate_g2l

```
integer function eigen_translate_g2l( ictr, nnod, inod )
```

1. `integer, intent(IN) :: ictr`
   Global counter
2. `integer, intent(IN) :: nnod`
   Process number
3. `integer, intent(IN) :: inod`
   Process ID

4.17 eigen_owner_node

Returns owner process ID corresponding to the specified global loop counter value.

```
integer function eigen_owner_node( ictr, nnod, inod )
```

1. `integer, intent(IN) :: ictr`
   Global loop counter
2. `integer, intent(IN) :: nnod`
   Process number
3. `integer, intent(IN) :: inod`
   Process ID

4.18 KMATH_EIGEN_GEV

This is a generalized eigenvalue computing driver routine that uses EigenExa as the eigenvalue computing engine. In this driver, `eigen_sx` is called to compute the eigenpairs via transformation to a pentadiagonal matrix. The constraints on this driver are similar to those on `eigen_sx`. Use of this driver routine requires linking `KMATH_EIGEN_GEV.o` together with EigenExa itself.

```
subroutine kmath_eigen_gev( n, a, lda, b, ldb, w, z, ldz )
```

1. `integer, intent(IN) :: n`
   Matrix and vector dimensions
2. `real(8), intent(INOUT) :: a(lda,*)`
   Matrix A for computing the pencil $(A - \lambda B)$
   The array content is destroyed upon subroutine termination.
3. `integer, intent(IN) :: lda`
   Leading dimension of array `a`
4. `real(8), intent(INOUT) :: b(ldb,*)`
   Matrix B for computing the pencil $(A - \lambda B)$
   The matrix for transformation to the standard eigenvalue problem is stored upon subroutine termination.
5. `integer, intent(IN) :: ldb`
   Leading dimension of array `b`
6. `real(8), intent(OUT) :: w(n)`
   Eigenvalues in ascending order
7. `real(8), intent(OUT) :: z(ldz,*)`
   B-orthogonal eigenvectors of the generalized eigenvalue problem
8. `integer, intent(IN) :: ldz`
   Leading dimension of array `z`

Chapter 5

Other key considerations

5.1 Caution regarding compatibility

As the successor to EigenK, EigenExa has inherited many of its functions. Complete compatibility between the two is not guaranteed, however, since their internal implementations differ in certain details. These are mainly differences in function and variable naming rules and in common domain management methods. For the same reason, simultaneous linking of EigenExa and EigenK is not recommended.

5.2 Binding with other languages

The method for calling EigenExa from a language other than Fortran90 is highly dependent on the user's environment. For further information, refer to "Language bindings" and "Method of linking to multiple programming languages" in the compiler manual. Information of reference may also be found in the "Python binding of EigenExa" project [10], which enabled calling from the Python language.

5.3 Behavior on error occurrence

During initialization, EigenExa checks that it is being executed under appropriate conditions, but error detection is not performed during execution. In some cases, forced library termination may occur if a linked subordinate library such as BLAS or LAPACK produces an error.

Information on bug discoveries is essential for improvement of library quality. On discovery of any bug, please be sure to report it to the developer email address listed on the public website of the library.

5.4 Shared library handling in versions 1.x

Shared libraries were not supported in former versions (1.x), because at the time of their development it was not possible to guarantee complete, collision-free resolution of function names when shared libraries are being used (with certain versions of gcc, abnormal shutdown occurred without resolution of function names at execution). When version 1.x is to be used as a shared library, this must therefore be performed solely at the user's responsibility.

EigenExa versions 2.x and later are shared-library capable, a development achieved with the technical cooperation of Team Leader Toshiyuki Maeda and other members of the HPC

Usability Research Team at the RIKEN Advanced Institute for Computational Science. As noted in the description of the library build, one should select the appropriate make.inc, and when executing, always remember to make the appropriate settings for the environment variables (such as `LD_LIBRARY_PATH`).

Appendix A

Algorithm overview

A.1 Introduction

Appendix A provides an overview of the eigenvalue computation algorithms used in EigenExa, with the main focus on ontlines of algorithms that two driver routines (eigen_s and eigen_sx) use and differences between them. Both routines are designed to meet the underlying EigenExa objective of computing all eigenvalues and eigenvectors of real symmetric dense matrices. For general details about eigenvalue computation algorithms for dense matrices, refer to sources such as [11, 12, 13, 14, 15].

A.2 Various approaches and related projects

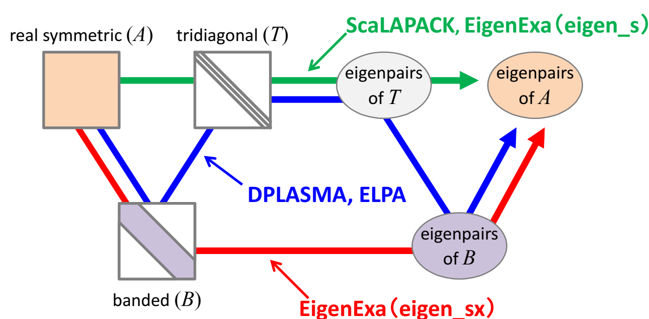


Figure A.1: Various approaches to eigenvalue computation for real symmetric dense matrices.

Let us begin with a brief introduction to the basic aspects of the eigenvalue computing procedures that is usually applied to real symmetric dense matrices. Textbooks on general matrix computation describe an approach based on tridiagonalization of the input matrix (the green path in Fig. A.1), which is used in ScaLAPACK [16] (and LAPACK). In the first step of this approach (tridiagonalization), however, the performance is limited by memory bandwidth, and therefore is expected to be not sufficiently high on recent computer systems.

This problem led to two development projects, ELPA [17] and DPLASMA [18], which employ an approach based on two-stage tridiagonalization via a banded matrix (the blue path in Fig. A.1). In this tridiagonalization, the dominant cost arises during the first stage, in transformation from dense to band. The byte/flop ratio required in this transformation is smaller

than that in direct tridiagonalization, which means the improvement of effective performance. But the eigenvector back-transformation process also requires two stages (basically doubling the cost). Since high-performance implementation in the first stage of the back-transformation (from T to B) is currently difficult, its cost becomes extremely large in obtaining a large number of eigenvectors.

These situations led to the development and provision of two routines for EigenExa. One (eigen_s) applies an approach based on the conventional (one-stage) tridiagonalization (the green path in Fig. A.1). The other (eigen_sx) applies an approach in which the eigenvalues and eigenvectors of a banded matrix are computed directly (the red path in Fig. A.1). The following sections describe these two approaches in a little more detail.

A.3 eigen_s

As noted above, the eigen_s routine in EigenExa applies an approach based on the conventional (one-stage) tridiagonalization, which is used in ScaLAPACK and other libraries. More specifically, it obtains solutions to the eigenvalue problem $A\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i$ $(i = 1, \ldots, N)$ through the following three steps:

1. Tridiagonalization of the input matrix by Householder transformations: $Q^\top AQ \to T$
2. Computation of the eigenvalues and eigenvectors of a tridiagonal matrix by the divide-and-conquer method: $T\boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i$
3. Back transformation of the eigenvectors: $Q\boldsymbol{y}_i \to \boldsymbol{x}_i$

In step 1, the Householder transformations act from both sides

$$H_{N-2}^\top \cdots H_1^\top A H_1 \cdots H_{N-2} \to T, \quad H_i = I - \boldsymbol{u}_i \beta_i \boldsymbol{u}_i^\top \tag{A.1}$$

with each column (row) of the input matrix transformed in turn to a tridiagonal matrix (Fig. A.2(a)). Here, we chose the position of the variable *beta* in the equations for its correspondence with the description further below. The computation of the transform by each Householder transformation is usually performed by using the symmetry of A, as

$$(I - \boldsymbol{u}\beta\boldsymbol{u}^\top)^\top A(I - \boldsymbol{u}\beta\boldsymbol{u}^\top) = A - \boldsymbol{u}\boldsymbol{v}^\top - \boldsymbol{v}\boldsymbol{u}^\top, \quad \boldsymbol{v} = (\boldsymbol{w} - \frac{1}{2}\boldsymbol{u}\beta^\top(\boldsymbol{w}^\top\boldsymbol{u}))\beta, \quad \boldsymbol{w} = A\boldsymbol{u}. \tag{A.2}$$

Furthermore, the Dongarra's method makes it possible to apply a number of transformations to the matrix in the form of matrix-matrix multiplications at a time:

$$(I - \boldsymbol{u}_K\beta_K\boldsymbol{u}_K{}^\top)^\top \cdots (I - \boldsymbol{u}_1\beta_1\boldsymbol{u}_1{}^\top)^\top A(I - \boldsymbol{u}_1\beta_1\boldsymbol{u}_1{}^\top) \cdots (I - \boldsymbol{u}_K\beta_K\boldsymbol{u}_K{}^\top) = A - UV^\top - VU^\top. \tag{A.3}$$

However, matrix-vector multiplications, whose performance is limited by the memory bandwidth, still remain (for obtaining the matrix V) and is therefore a major bottleneck for high-performance computing.

In the second step, the divide-and-conquer method proposed by Cuppen [19] is applied to compute the eigenvalues and eigenvectors of the tridiagonal matrix. As shown in Fig. A.2(b), a tridiagonal matrix can be decomposed into a block diagonal matrix and a rank one perturbation. The underlying idea of the method is computing the eigenvalue decomposition of the tridiagonal matrix efficiently by using the eigenvalue decomposition of the block diagonal matrix (and recursively apply this idea to the block diagonal matrix).

In the third step, back transformation of the eigenvectors is performed by applying the Householder transformations obtained in the first step to the eigenvectors of the tridiagonal

matrix in the reverse order. Since a number of Householder transformations can be aggregated into a convenient form (compact-WY representation):

$$H_1 \cdots H_K = (I - \boldsymbol{u}_1\beta_1\boldsymbol{u}_1^\top)\cdots(I - \boldsymbol{u}_K\beta_K\boldsymbol{u}_K^\top) \to I - USU^\top, \quad U = [\boldsymbol{u}_1 \cdots \boldsymbol{u}_K] \tag{A.4}$$

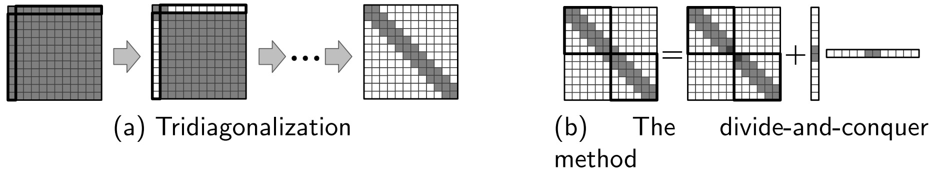
at low cost (only for computing a small matrix S), the back transformation is usually computed via matrix-matrix multiplications (Level-3 BLAS):

$$H_1 \cdots H_{N-2}Y = (I - U_1S_1U_1^\top)\cdots(I - U_MS_MU_M^\top)Y \to X, \tag{A.5}$$

where

$$Y = [\boldsymbol{y}_1 \cdots \boldsymbol{y}_N], \quad X = [\boldsymbol{x}_1 \cdots \boldsymbol{x}_N]. \tag{A.6}$$

For this reason, this step is expected to achieve high performance.



(a) Tridiagonalization (b) The divide-and-conquer method

Figure A.2: Schematic of eigenvalue computation by eigen_s.

In eigen_s, the first and third steps are newly implemented from scratch with appropriate thread parallelization, whereas the second step is almost ported from the ScaLAPACK code.

A.4 eigen_sx

The other driver routine provided in EigenExa, namely eigen_sx, is based on an approach that employs direct computation of the eigenvalues and eigenvectors of a banded matrix. At present, for the reason mentioned in the last section, a pentadiagonal matrix is selected as the banded matrix. More specifically, the eigenvalue problem is solved in the following three steps.

1. Pentadiagonalization of the input matrix by block version of Householder transformations: $\tilde{Q}^\top A\tilde{Q} \to B$
2. Computation of the eigenvalues and eigenvectors of a pentadiagonal matrix by the divide-and-conquer method: $B\boldsymbol{y}_i = \lambda_i\boldsymbol{y}_i$
3. Back transformation of the eigenvectors: $\tilde{Q}\boldsymbol{y}_i \to \boldsymbol{x}_i$

In the first step, the block version of Householder transformations are applied to the input matrix from both sides:

$$\tilde{H}_{N/2-1}^\top \cdots \tilde{H}_1^\top A\tilde{H}_1 \cdots \tilde{H}_{N/2-1} \to P, \quad \tilde{H}_i = I - \tilde{\boldsymbol{u}}_i\tilde{\beta}_i\tilde{\boldsymbol{u}}_i^\top \tag{A.7}$$

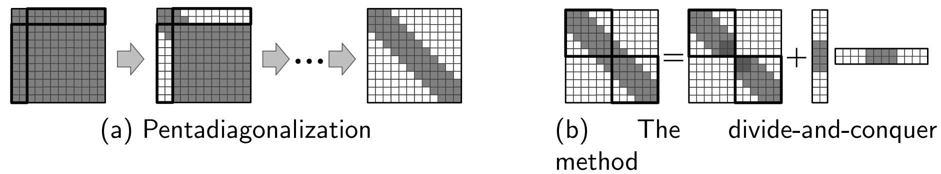
to transform every two columns (two rows) of the input matrix into a pentadiagonal matrix (Fig. A.3(a)), where

$$\tilde{\boldsymbol{u}}_i = [\boldsymbol{u}_1^{(i)}\ \boldsymbol{u}_1^{(i)}], \quad \tilde{\beta}_i = \begin{pmatrix} \beta_{11}^{(i)} & \beta_{12}^{(i)} \\ \beta_{21}^{(i)} & \beta_{22}^{(i)} \end{pmatrix}. \tag{A.8}$$

There is no difference excepting the form of $\tilde{H}$ between Eqs. (A.1) and (A.7), so that the procedure of the pentadiagonalization is the same as the tridiagonalization; the Dongarra's method can similarly be applied. The performance bottleneck thus resides in the part of computing $A\tilde{\boldsymbol{u}}$.

In the second step, as shown in Fig. A.3(b), the pentadiagonal matrix is decomposed into a block diagonal matrix and a rank two perturbation. By treating the rank two perturbation as two rank one perturbations, we apply the principal of the divide-and-conquer method for a tridiagonal matrix twice and compute the eigenvalues and eigenvectors of the pentadiagonal matrix [20].

In the third step, the block version of Householder transformations obtained in the first step are applied to the eigenvectors of the pentadiagonal matrix in the reverse order, which is essentially the same as in the case of tridiagonalization. The block version of Householder transformations can also be aggregated in a form with matrices as in Eq. A.4, and matrix-matrix multiplication can therefore be used in this step, which promises that this step easily achieves high performance.



(a) Pentadiagonalization (b) The divide-and-conquer method

Figure A.3: Schematic of eigenvalue computation by eigen_sx.

In eigen_sx, as in eigen_s, steps 1 and 3 are newly implemented with appropriate thread parallelization, whereas step 2 is a simple extended implementation of the ScaLAPACK code for a pentadiagonal matrix.

A.5 Differences between eigen_s and eigen_sx

As described in A.3 and A.4, eigen_s and eigen_sx comprise three similar steps and are nearly the same in computation procedures. Particularly in the step of the back transformation of the eigenvectors, there is no essential difference between them. In this section, we mention the main differences in the first and second steps between them.

Tri/Pentadiagonalization

The essential difference between eigen_s and eigen_sx in this step is that eigen_s processes a single vector, whereas eigen_sx processes two vectors together, e.g.

$$\boldsymbol{w} = A\boldsymbol{u} \text{ (in eigen_s)} \quad \rightarrow \quad [\boldsymbol{w}_1\ \boldsymbol{w}_2] = A[\boldsymbol{u}_1\ \boldsymbol{u}_2] \text{ (in eigen_sx)}. \tag{A.9}$$

In the overall step, the total number of floating-point operations is a about the same (at least for the highest term) for the two routines; amount per operation required in eigen_sx in about twice that in eigen_s, but the number of operations in eigen_sx is about half that in eigen_s because the former deals with two columns at an operation. For the similar reasons, the amount of the data transfered among distributed processes is almost the same for the two.

The first difference that becomes evident between the two is in the effective performance of the floating-point operations (in particular, in matrix-vector multiplications). The data of matrix A can be reused when computing $A[\boldsymbol{u}_1\ \boldsymbol{u}_2]$, whereas it cannot when computing $A\boldsymbol{u}$. This means that the required byte/flop ratio in the former is lower than that in the latter. As a result, the effect of limitation by memory bandwidth is smaller in the former than in the latter (theoretically by about half), which indicates the increasing effective performance in eigen_sx.

The second difference that becomes evident is in the communication latency, arising from the difference in communication frequencies. Although the data amount per communication is larger

in eigen_sx than in eigen_s, the frequency of communications is lower (by about half). The feature of lower communication frequency (Communication-Avoidance) is a very substantial difference, especially in cases of massively parallel computing, due to the fact that communication latency has become a major problem on recent systems.

In short, eigen_sx exhibits clear advantages over eigen_s in terms of both the performance of floating-point operations and the latency cost of communication. In cases where the problem size relative to the number of processes is sufficiently large (with the time for floating-point operations thus dominant), the advantage in effective performance is significant. On the other hand, in cases where the number of processes is large (with communication time thus dominant), the advantage in the latency cost is therefore significant. In total, eigen_sx is expected to achieve higher performance than eigen_s.

The divide-and-conquer method

It is clear that eigen_sx requires more cost (both for floating-point operations and communications) than eigen_s because the former deals with a rank two perturbation whereas the latter deals with a rank one perturbation. In eigen_sx, a rank two perturbation is dealt with two rank one perturbations. The computational cost for the first rank one perturbation can be reduced by exploiting the structure of the matrix (i.e. block diagonal), however such benefit done not exist in the computation for the second rank one perturbation; the latter cost is about twice that of the former cost. The resulting cost required in eigen_sx increases by a factor of about three.

In the divide-and-conquer method, one can reduce the cost substantially by the technique known as "deflation". The number of opportunities in which deflation can be applied varies with the problem. In addition, it is different even for the same input matrix between routines via tridiagonalization and pentadiagonalization. Therefore, it is not easy to derive a theoretical estimation of the difference between the costs of the two routines.

A.6 Conclusion

In this chapter, we gave an overview of the algorithms employed in the two routines, eigen_s and eigen_sx, provided in EigenExa and explained their main differences. Increasing the bandwidth of the banded matrix generally involves the trade-off; it is advantageous in the first step (the transformation step), but disadvantageous in the second step (the divide-and-conquer method). Taking this trade-off into account, we deem that the pentadiagonal matrix is appropriate on today's systems. With increasing performance of future systems and improved implementation of the divide-and-conquer method (our ScaLAPACK-based implementation seems to be rarely suitable for current systems), banded matrix with larger band width (e.g. heptadiagonal) may prove promising. By contrast, the use of conventional tridiagonalization (eigen_s) might in some circumstances be the best choice. We hope that an understanding of the information in this appendix will help users to select the routine that is most appropriate to their application.

Acknowledgements

We would like to express our deepest gratitude to all the members of the EigenExa development group for their sincere contribution to developing EigenExa. We also grateful the support from the RIKEN Advanced Institute for Computational Science (AICS). In addition, we appreciate the users of EigenExa for their bug reports and feedback that is beneficial to further quality improvement of EigenExa. Without people and support above mentioned, EigenExa could have not been released.

The EigenExa project has been supported by the following funds.

- JST CREST, "Development of System Software Technologies for post-Peta Scale High Performance Computing" (FY2011–FY2015).
- MEXT KAKENHI, Grant Numbers 21300013 (FY2012), 23240005 (FY2011–FY2013), and 15H02709 (FY2015–FY2017).

The EigenExa project has used computational resources of the K computer provided by the RIKEN AICS through the folowing projects.

- HPCI System Research project, project IDs hp140069 (FY2014) and hp140069 (FY2012–FY2013).
- Computational resources for enhancement, Project ID ra000005 (FY2013–).

The EigenK project, the predecessor of the EigenExa project, was supported by the following fund.

- JST CREST, "High Performance Computing for Multi-Scale and Multi-Physics Phenomena" (FY2006–FY2012).

References

[1] S. Yamada, T. Imamura, T. Kano and M. Machida, "High-Performance Computing for Exact Numerical Approaches to Quantum Many-Body Problems on the Earth Simulator", Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06), November 2006. Tampa USA.
`http://doi.acm.org/10.1145/1188455.1188504`

[2] T. Imamura, S. Yamada and M. Machida, "Development of a High Performance Eigensolver on the Peta-Scale Next Generation Supercomputer System", Progress in Nuclear Science and Technology, the Atomic Energy Society of Japan, Vol. 2, pp.643–650 (2011) .

[3] T. Imamura, S. Yamada and M. Machida, "Eigen-K: high performance eigenvalue solver for symmetric matrices developed for K computer", 7th International Workshop on. Parallel Matrix Algorithms and Applications (PMAA2012), June 2012, London UK.

[4] What is K? — RIKEN Advanced Institute for Computational Science,
`http://www.aics.riken.jp/en/k-computer/about/`

[5] K computer — Fujitsu Global,
`http://www.fujitsu.com/global/about/businesspolicy/tech/k/`

[6] T. Imamura and Y. Yamamoto, "CREST: Dense Eigen-Engine Groups", International Workshop on Eigenvalue Problems: Algorithms; Software and Applications, in Petascale Computing (EPASA 2014), Tsukuba, March 7–9, 2014 (poster).
`http://www.aics.riken.jp/labs/lpnctrt/EPASA2014_dense_poster_ImamuraT_only.pdf`

[7] T. Imamura, "The EigenExa Library – High Performance & Scalable Direct Eigensolver for Large-Scale Computational Science", HPC in Asia, Leipzig, Germany, June 22–26, 2014.

[8] T. Imamura, Y. Hirota, T. Fukaya, S. Yamada and M. Machida, "EigenExa: high performance dense eigensolver, present and future", 8th International Workshop on Parallel Matrix Algorithms and Applications (PMAA14), Lugano, Switzerland, July 2–4, 2014.

[9] T. Fukaya and T. Imamura, "Performance evaluation of the EigenExa eigensolver on the Oakleaf-FX supercomputing system", Annual Meeting on Advanced Computing System and Infrastructure (ACSI) 2015, Tsukuba, January 26–28, 2015.

[10] Python binding of EigenExa, HPC Usability Research Team, RIKEN AICS, `http://www.hpcu.aics.riken.jp/`

[11] B. Parlett, "The Symmetric Eigenvalue Problem", SIAM (1987).

[12] J. Demmel, "Applied Numerical Linear Algebra", SIAM (1997).

[13] L. Trefethen and D. Bau, III, "Numerical Liner Algebra", SIAM (1997).

[14] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, "Numerical Recipes: The Art of Scientific Computing", 3rd ed., Cambridge University Press (2007).

[15] G. Golub and C. Van Loan, "Matrix Computations", 4th ed., Johns Hopkins University Press (2012).

[16] ScaLAPACK, `http://www.netlib.org/scalapack/`

[17] ELPA, `http://elpa.rzg.mpg.de/`

[18] DPLASMA, `http://icl.cs.utk.edu/dplasma/`

[19] J. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem", Numer. Math, Vol.36, pp.177–195 (1981).

[20] P. Arbenz, "Divide and conquer algorithms for the bandsymmetric eigenvalue problem", Parallel Computing, Vol.18, No.10, pp.1105–1128 (1992).