User's Manual
EigenExa
Version 2.11

i) EigenExa Development Group
Large-scale Parallel Numerical Computing Technology Research Team
ii) Numerical Library development Working Group
FS2020 Architecture Development Team (until FY2020)

RIKEN Center for Computational Science[1,2]

Released for v2.6: November 1, 2020
Revised for v2.6: Feburary 15, 2021
Revised for v2.7: April 1, 2021
Revised for v2.7: June 29, 2021
Revised for v2.8: August 20, 2021
Revised for v2.9: September 24, 2021
Revised for v2.10: October 17, 2021
Revised for v2.11: December 1, 2021

[1]Corresponding author: Toshiyuki Imamura (imamura.toshiyuki@riken.jp)
[2]Contact e-mail address: the EigenExa developer team (EigenExa@ml.riken.jp)

Contents

1 Introduction **7**

1.1 EigenExa and the history of development . . . 7
1.2 Current implementation of EigenExa . . . 7
1.3 License of use and Copyright . . . 10

2 Getting started **13**

2.1 Pre-requisite for EigenExa installation . . . 13
2.2 Obtaining EigenExa . . . 13
2.3 Compile and install procedure . . . 14

3 Quick tutorial **17**

3.1 Standard call . . . 17
3.2 Communicator . . . 18
3.3 Handling counter and index . . . 19
3.4 Corporate with ScaLAPACK . . . 21
3.5 Compiling issues . . . 21

4 API's **23**

4.1 `eigen_init` . . . 23
4.2 `eigen_free` . . . 24
4.3 `eigen_get_blacs_context` . . . 24
4.4 `eigen_sx` . . . 25
4.5 `eigen_s` . . . 26
4.6 `eigen_h` . . . 27
4.7 `eigen_get_version` . . . 28
4.8 `eigen_show_version` . . . 28
4.9 `eigen_get_matdims` . . . 28
4.10 `eigen_memory_internal` . . . 29
4.11 `eigen_get_comm` . . . 30
4.12 `eigen_get_procs` . . . 30
4.13 `eigen_get_id` . . . 30
4.14 `eigen_get_errinfo` . . . 31
4.15 `eigen_loop_start` . . . 31

4.16 eigen_loop_end . . . 33
4.17 eigen_loop_info . . . 33
4.18 eigen_translate_l2g . . . 34
4.19 eigen_translate_g2l . . . 35
4.20 eigen_owner_node . . . 36
4.21 eigen_owner_index . . . 37
4.22 eigen_convert_ID_xy2w . . . 37
4.23 eigen_convert_ID_w2xy . . . 38
4.24 KMATH_EIGEN_GEV . . . 38

5 Other Considerations 41
5.1 Regarding upper/lower compatibilities . . . 41
5.2 Binding with other languages . . . 41
5.3 Behavior on error occurrence . . . 41
5.4 Shared library handling in versions 1.x . . . 42
5.5 Known bugs and the best workaround . . . 42
5.5.1 Numerical reproducibility in Intel compiler and Intel MPI . . . 42

Appendices

Appendix A Algorithm overview 47
A.1 Introduction . . . 47
A.2 Various approaches and related projects . . . 47
A.3 eigen_s . . . 48
A.4 eigen_sx . . . 49
A.5 Differences between eigen_s and eigen_sx . . . 51
A.6 Conclusion . . . 52

Appendix B Release notes 53
B.1 Version 2.11 (December 1, 2021) . . . 53
B.2 Version 2.10 (Octorber 17, 2021) . . . 53
B.3 Version 2.9 (September 24, 2021) . . . 54
B.4 Version 2.8 (August 20, 2021) . . . 54
B.5 Version 2.7 (April 1, 2021) . . . 54
B.6 Version 2.6 (November 1, 2020) . . . 54
B.7 Version 2.5 (August 1, 2019) . . . 54
B.8 Version 2.4b (August 20, 2018) . . . 55
B.9 Version 2.3m (August 20, 2018) . . . 55
B.10 Version 2.4p1 (May 25, 2017) . . . 55
B.11 Version 2.4 (April 18, 2017) . . . 55
B.12 Version 2.3k2 (April 12, 2017) . . . 55
B.13 Version 2.3d (July 07, 2015) . . . 55
B.14 Version 2.3c (April 23, 2015) . . . 56
B.15 Version 2.3b (April 15, 2015) . . . 56

B.16 Version 2.3a (April 14, 2015) . . . 56
B.17 Version 2.3 (April 12, 2015) . . . 56
B.18 Version 2.2d (March 20, 2015) . . . 57
B.19 Version 2.2c (March 10, 2015) . . . 57
B.20 Version 2.2b (October 30, 2014) . . . 57
B.21 Version 2.2a (June 20, 2014) . . . 57
B.22 Version 2.2 (April 10, 2014) . . . 57
B.23 Version 2.1a (Feb 23, 2014) . . . 58
B.24 Version 2.1 (Feb 10, 2014) . . . 58
B.25 Version 2.0 (Dec 13, 2013) . . . 58
B.26 Version 1.3a (Sep 21, 2013) . . . 58
B.27 Version 1.3 (Sep 20, 2013) . . . 58
B.28 Version 1.2 (Sep 17, 2013) . . . 59
B.29 Version 1.1 (Aug 30, 2013) . . . 59
B.30 Version 1.0 (Aug 1, 2013) . . . 59

Acknowledgements **61**

References **63**

Chapter 1

Introduction

1.1 EigenExa and the history of development

EigenExa(/aigen-éksə/) is a high-performance parallel eigenvalue solver. The history of the EigenExa family is beyond a decade and traceable back to EigenES (a code-name but not the official name), which was developed on the world's top-ranked supercomputer, Earth Simulator [1]. The result of EigenES was appreciated in the challenging computational material simulation field, and the authors group was nominated for a Gordon Bell Prize at SC 2006 and it today continues to serve as an eigenvalue solver on large-scale PC clusters [2]. This led to the initiation of EigenK [3, 4] 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 the official launch of the K computer [5, 6] available to the public. EigenExa development still continues, with the underlying objective being to achieve an eigenvalue library scalable to operate on future post-petascale ("exa" ($= 10^{15}$) or "extreme") computer systems.

In the Flagship 2020 project (known as the supercomputer "Fugaku" project) [7], which was conducted by RIKEN Center for Computational Science (R-CCS) from FY2014 to FY2020, EigenExa was thought to be a core part of the numerical calculation library as the system software. RIKEN developed a partly new implementation of EigenExa, version 2.6, which was optimized for the A64FX processor and TofuD interconnect. The R&D and maintenance of EigenExa have been carried out in the Large-scale Parallel Numerical Technology team following the development activities by the FS2020 Project Architecture Development team for the supercomputer "Fugaku".

1.2 Current implementation of EigenExa

As the same as the previous and present releases (version 2.3c, 2.4b, and 2.6 to 2.11), EigenExa provides the simplest function of computing all eigenpairs (eigenvalues paired with their respective eigenvectors) for both standard and generalized eigenvalue problems. In addition to the real symmetric matrices, we have started to provide the 'long-awaited' support of Hermitian matrices in version 2.11. As reported elsewhere [2, 3, 4], EigenExa applies both classical and advanced algorithms in the same basic manner as EigenK, and thereby reduces the required computational time for diagonalization. We analyzed that one of the challenges in the development is the hardware imbalance due to the slowing down of the network performance of the supercomputer after the K computer, while the processor performance has improved accordingly. In fact, communication is the biggest bottleneck in the development of highly parallelized computers, even for recent small-scale supercomputers. It has been recognized even before the development of the Fugaku scale that it cannot be ignored.

In this release (the latest version 2.11 published for the public use on the supercomputer Fugaku), we apply a communication avoidance technique to the householder tridiagonalization together with [15], new process mapping for the 'load balance' of the divide and conquer method. These techniques have made a significant contribution to performance improvement in highly parallel environments.

Since its development has been initiated in early 2000's, we have taken advantage of various parallel programming languages and libraries, encompassing MPI, OpenMP, high-performance BLAS, and SIMD vectorization with advanced Fortran/C/C++ compiler supports. On the K computer and its successor, Fugaku, EigenExa is expected to open a promised way for high performance computing through the 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 multi-core processors, by OpenMP,
3. High parallelism utilizing BLAS highly optimized by vendors, and
4. SIMD or coarse-grained parallelism utilizing vendor-provided high-performance compilers

The beneficial features of Fortran 90/95/2003 are also actively incorporated into EigenExa. The API of EigenExa is more flexible than that of libraries implemented in Fortran77, and it provides a user-friendly interface, based on modular interfaces and optional parameters. Although data distribution is limited to two-dimensional cyclic decomposition, the processor map can be specified in almost any arbitrary configuration. Compatibility and consistency with the existing numerical computation libraries are guaranteed if the data redistribution function provided by ScaLAPACK is used. Furthermore, EigenExa offers user specification (or omission) for heightened performance, such as block parameters that strongly affect execution performance.

In terms of the library itself's parallel performance, it yields heightened performance by reducing the communication overhead, and it has been shown that in most cases, EigenExa outperforms EigenK, ScaLAPACK, and others of the state-of-the-arts class numerical libraries [4].

Today, EigenExa works on many cutting-edge HPC platforms, including the K computer, the supercomputer Fugaku, and the Fujitsu PRIMEHPC commercial server series, various cluster computers using Intel x86 or AMD64 processors, IBM Blue/Gene Q systems, and the NEC vector computer SX series systems, however, unfortunately, it is not supporting

GPU cluster systems, at the moment. Several reports on EigenExa have been presented at scientific conferences[8, 9, 10, 11, 12, 13, 14, 15, 16, 17], so if interested in the internal implementation, algorithms, and preliminary performance benchmark, the authors hope that the readers refer to them.

This user's manual for EigenExa version 2.11 covers almost the whole spectrum of EigenExa, from installation to actual use, with particular consideration given to installation and compiling, a quick tutorial, the API list, and compatibility with EigenExa 2.3c or prior. It is written and provided with all EigenExa team developers' hope that it will assist many users in achieving efficient parallel simulations.

1.3 License of use and Copyright

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

LICENCE.txt

Note that some of the codes are derivative of LAPACK-3.4.2 and ScaLAPACK-2.0.2, it is subject to their original software license.

[LAPACK 3.4.2]

[ScaLAPACK 2.0.2]

Chapter 2

Getting started

2.1 Pre-requisite for EigenExa installation

Users need the following software packages to compile the EigenExa library. BLAS, LAPACK, ScaLAPACK. Additionally, MPI must be installed on the system prior to compiling EigenExa. 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,
vendor-provided MPI's such as
Intel MPI, Fujitsu MPI, and NEC MPI/SX
- Compilers — Fortran 2003 or 2008 (The features of Fortran 95 and some of Fortran 2003 technical extensions are utilized. Until v2.6, Fortran 90/95 or later was recommended), and C/C++ compilers are required.
GNU compiler 9.0 or later for x86,
Intel compiler 19, or openAPI compiler,
Fujitsu software Technical Computing Suite V4.0,
and other vendor-provided compilers

2.2 Obtaining EigenExa

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

https://www.r-ccs.riken.jp/labs/lpnctrt/projects/eigenexa/

Tarball distribution (tgz or tar.gz) is also provided via this URL. Future planning is in progress for provision of further information on EigenExa. This manual describes the work after downloading the tarball of `EigenExa-2.11`.

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 (tgz or tar.gz) on the working directory and then move to the `EigenExa-2.11` directory

```
% tar zxvf EigenExa-2.11.tar.gz
% cd EigenExa-2.11
```

Environment setting Next, run the `bootstrap` and then `configure` script to generate `Makefile` or other settings automatically based on user's system environment.

```
% ./bootstrap
% ./configure
```

BLAS, LAPACK, and ScaLAPACK are stored in several different directories on each system, and you may have to choose the appropriate one from the several libraries. In that case, you can set the following environment variables appropriately. The available options of the `configure` script can be found in `./configure --help`.

- `FC` fortran compiler for MPI (for example, `mpif90`)
- `CC` C compiler for MPI (for example, `mpicc`)
- `LAPACK_PATH` The directory path information for LAPACKs
- `LAPACK_LIBS` The library information for linking LAPACKs

The cross-compilation option is available by passing `--host=`**hostname**. If you would like to build ARM64v8.2a binaries on the supercomputer Fugaku, please add `--host=login` (here, '**hostname**' is an arbitrary string). If the configure script doesn't work correctly, it may be due to a difference in the software version used in the script. You should begin by executing the `cleanup` script and the `bootstrap` script and then recreate the `configure` script as follows.

```
% ./cleanup
% ./bootstrap
```

make Third, run make. As a result, the static library `libEigenExa.a` and the shared library `libEigenExa.so` are created.

```
% make
```

install Finally, copy the library itself, `libEigenExa.a` (in the case of shared library, `libEigenExa.so`) and several fortran modules (`eigen_libs_mod.mod`, `eigen_libs0_mod.mod`, `eigen_blacs_mod.mod`, `comm_mod.mod`, and `fs_libs_mod.mod`) to the installation sub-directories (`lib/`, and `include/`).

```
% make prefix=(installation directory) install
```

To install manually, e.g. into `/usr/local/lib`, do the following (a backslash at the end of the line ('\') implies a continuation line and is not necessary for the actual input).

```
% cp libEigenExa.a libEigenExa.so eigen_libs_mod.mod \
eigen_libs0_mod.mod \
eigen_blacs_mod.mod comm_mod.mod fs_libs_mod.mod /usr/local/lib/
```

Generalized eigenvalue computation driver routine Until the version 2.4, the generalized eigenvalue driver routine `KMATH_EIGEN_GEV` had to be compiled separately, but since version 2.6, it is imported into EigenExa. In short, once you make it, the driver module `KMATH_EIGEN_GEV.o` for generalized eigenvalues is included in `libEigenExa.a` and `libEigenExa.so`, which enables the generalized eigenvalue calculation at program link time.

Chapter 3

Quick tutorial

3.1 Standard call

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_mod
...
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 only shows a skeletal part and does not actually work, but it is essential to see the typical flow from initialization, array allocation, eigenvalue calculation, to termination procedure.

3.2 Communicator

In the above example, the initialization function `eigen_init()` is invoked without an optional parameter type call. The user can specify to `eigen_init()` the process group, of the form `comm=XXX`, as a communicator to perform the eigenvalue calculation. For example, if you execute eigenvalue calculation in parallel with multiple groups, you can simply pass the communicator created by `MPI_Comm_split()` and so on. Note that `eigen_init()` includes a collective operation and must be called simultaneously by all the processes belonging to the communicator.

Since a different communicator can be specified for each process, a process not participating in the eigenvalue calculation can have `MPI_COMM_NULL` specified for `eigen_init()`, thus having the call skipped for the eigenvalue driver `eigen_sx()` itself. In short, it is possible to perform simultaneous execution of various operations other than `eigen_sx()`, of course, including `eigen_sx()`.

`MPI_Comm_split` and `MPI_COMM_NULL`

```
color = 0; key = my_rank / 4
call MPI_Comm_split( MPI_COMM_WORLD, color, key, comm_new, ierr )
if ( my_rank < 16 ) then
   comm = comm_new
else if ( my_rank < 32 ) then
   comm = MPI_COMM_SELF
else
   comm = MPI_COMM_NULL
endif
call eigen_init( comm )
if ( comm /= MPI_COMM_NULL ) then
   call eigen_sx( .... )
else
   ... (other statements,
         In the specification, exigen_sx returns immediatedly)
endif
```

In EigenExa, processes belonging to a communicator specified by `eigen_init()` are deployed on a two-dimensional process grid. EigenExa is designed to reduce the amount of communication possible by adopting a square-shaped process grid. EigenExa has been developed to enhance user convenience so that the two-dimensional Cartesian adopted by MPI can be specified as comm. In principle, if the geometry of the Cartesian is two-dimensional, EigenExa can be used to compute arbitrary process configurations by calling EigenExa, which can be combined with several types of communicators to perform complex parallel processing. In addition, because the Cartesian process grid is essentially Row-major, the Cartesian process is prioritized in the event of a conflict with order='C' specification. Note that for historical reasons, the default process grid of 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 the 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, a rule for transformation between global and local indices is required when matrix elements are accessed.

3.3 Handling counter and index

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 counter loop structure and the same but translated to local counter loops.

matset(before parallelization)

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

↓↓↓↓↓↓

matset (after parallelization)

```
! Translated local loop program to compute a Frank matrix
use MPI
use eigen_libs_mod

call eigen_loop_info( j_2, j_3, 1, n, 'X' )
call eigen_loop_info( i_2, i_3, 1, n, 'Y' )

do i_1 = i_2, i_3
   i = eigen_translate_l2g( i_1, 'Y' )
   do j_1 = j_2, j_3
      j = eigen_translate_l2g( j_1, 'X' )
      a(j_1, i_1) = DBLE(n+1-Max(n+1-i,n+1-j))
   end do
end do
```

matset (after parallelization using 2.4 or prior)

```
! Translated local loop program to compute a Frank matrix
use MPI
use eigen_libs_mod

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) = DBLE(n+1-Max(n+1-i,n+1-j))
   end do
end do
```

The `eigen_loop_start()` and `eigen_loop_end()` are called to transform the loop range from global to local. The third and fourth parameters specify the global counter values, which present the initial and final values of the loop iteration; besides, the fifth parameter indicates a character that represents the direction of distribution (In the case of 2.4 prior, 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] $\rightarrow$ "x" and [column] $\rightarrow$ "y"(in the case of the overall communicator for all the participating processes, 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 reduced 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 this translation. The second and third parameters should be specified like `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 must be included in the loop), on which returns the same value regardless of whether the process calling the function is the owner process or not.

In knowing the owner process of a given global loop counter, a user can retrieve information by calling `eigen_owner_node()` or `eigen_owner_index()`. The following programs illustrate their use, where the information is used for referring to or broadcasting particular matrix elements, for example, $A(j, i)$.

Broadcast

```
! Broadcast a(j,i)
i_1=eigen_owner_index( i, 'Y' )
j_1=eigen_owner_index( j, 'X' )
if ( i_1 > 0 .and . j_1 > 0 ) then
  v = a(j_1, i_1)
endif
i_0=eigen_owner_node( i, 'Y' )
j_0=eigen_owner_node( j, 'X' )
root=eigen_convert_ID_xy2w( j_0, i_0 )
call MPI_Bcast( v, 1, MPI_DOUBLE_PRECISION, root, TRD_COMM_WORLD, ierr )
```

3.4 Corporate with ScaLAPACK

Furthermore, when progressing to computation together with ScaLAPACK for personnel at advanced levels, the process grid context working as a proxy object between ScaLAPACK and 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 )
```

3.5 Compiling issues

When compiling and 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_mod.mod` 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 GNU-compiler and OSS-based MPI case, the procedure is as follows (note that the library path and names around ScaLAPACK and BLAS vary with the environment).

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

Chapter 4

API's

This section lists the functions in 'eigen_libs_mod.mod' that have been assigned a public attribute. The first three routines are the main drivers, and the others are utility functions. When the user specifies the required modules (basically, eigen_libs_mod.mod) by a USE statement, generic naming rules and parameters having optional attributes attached (written in italics) are available. The parameters attached with an optional attribute can be omitted and can also be specified by TERM='variable' *or* 'constant value' in the Fortran format form.

Floating point number variables use real(8) (complex numbers as well) to distinguish between double-precision and single-precision format, while integers are in principle 32-bit format, in short, integer=integer(4) is supposed except the explicit indication of integer(8).

EigenExa is not a thread-safe implementation. Therefore, the following functions without a note against multi-threading must only be used outside covering the OpenMP, in short, out of the '**OMP region's**'.

4.1 eigen_init

Initializes the functions of EigenExa. Process grid mapping can be specified via the 'comm' and 'order' arguments. Because of this procedure's collective behavior, all processes participating in the EigenExa calculation must call this function simultaneously. Since this function creates up to five sub-communicators, it may consume a lot of internal memory and processing time in massively parallel execution. In addition, we note that the sampling of the communication performance of the sub-communicators requires considerable overhead.

Once invoked the function, you must not call eigen_init() before calling the exit procedure with eigen_free() (described in the next section). For example, if you want to change the base communicator comm, you must call eigen_free() and then use eigen_init() to change the communicator.

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 equivalent to MPI_COMM_NULL, calling the handlers eigen_sx() and eigen_s() results in an immediate return without any internal actions. An inter-communicator cannot be used for comm.

EigenExa internally runs a multi-threaded OpenMP process, but the number of threads must be the same along all the processes. When calling the function, the program will abort if processes with a different number of threads are detected (invoking `MPI_Abort()`). In addition, thread parallelism with the default number of threads, when `eigen_init` is invoked, is only secured, and dynamical changes by `omp_set_num_threads()` in middle stream may result in negative impact on EigenExa internal.

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

1. `integer, optional, intent(IN) :: comm = MPI_COMM_WORLD`
   Base communicator
   When comm is a two-dimensional cartesian, the process map is available.
   Note: default is `MPI_COMM_WORLD`.
2. `character(*), optional, intent(IN) :: order = 'C'`
   `'R'` (Row) or `'C'` (Column)
   Note: default is 'C'. If the grid-major and the cartesian `comm` has
   a conflict on their specification, an appropriate major is taken into account.

4.2 eigen_free

Finalizes the funciton of EigenExa.

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

1. `integer, optional, intent(IN) :: flag = 0`
   The special flag for a timer printer, which is developper-purposed,
   so it should be omitted in normal case. Default is 0.

4.3 eigen_get_blacs_context

Returns the context of ScaLAPACK (BLACS) corresponding to the process grid information specified in EigenExa. This is necessary when exchanging data between EigenExa and ScaLAPACK.

`integer function eigen_get_blacs_context( )`

4.4 eigen_sx

Is the main driver routine of EigenExa to compute the standard real symmetric eigenvalue problem. It computes the eigenpairs via a transformation to a quintuple diagonal matrix. This driver is a collective operation, and all processes belonging to the calling process group 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`
   Dimensions of the matrix and vector to be diagonalized.
2. `integer, intent(IN) :: nvec`
   The number of eigenvectors (eigen-modes) to be computed.
   If positive, eigen modes are computed from the smallest.
   If zero, all eigenvalues without eigenvectors are computed from the smallest.
   (if negative, they are computed from the largest, but not supported currently.)
3. `real(8), intent(INOUT) :: a(1:lda,*)`
   A symmetric matrix to be diagonalized (upper triangular part is available), whose size must be equal or larger than the values obtained by calling `eigen_get_matdims`.
   Array content is destroyed upon the subroutine termination, but
   `a(1, 1)` returns the flops count on exit.
4. `integer, intent(IN) :: lda`
   The leading dimension of array a, which must be equal or larger than
   the value obtained by calling `eigen_get_matdims`.
5. `real(8), intent(OUT) :: w(1:n)`
   The eigenvalues of matrix a stored in the ascending/descending order according to nvec.
6. `real(8), intent(OUT) :: z(1:ldz,*)`
   The eigenvectors of matrix a. The size of array must be equal or larger than
   the values obtained by calling `eigen_get_matdims`.
7. `integer, intent(IN) :: ldz`
   The leading dimension of array z, which must be equal or larger than
   the value obtained by calling `eigen_get_matdims`.
8. `integer, optional, intent(IN) :: m_forward = 48`
   The block factor of the forward Householder transformation (must be even).
   Default is 48.
9. `integer, optional, intent(IN) :: m_backward = 128`
   The backward block factor in the Householder transformation. Default is 128.
10. `character(*), optional, intent(IN) :: mode = 'A'`
    'A' : all eigenvalues and corresponding eigenvectors (default)
    'N' : eigenvalues only
    'X' : add to mode 'A' to improve accuracy.

4.5 eigen_s

Another driver routine of EigenExa to compute the standard real symmetric eigenvalue problem.

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

1. `integer, intent(IN) :: n`
   Dimensions of the matrix and vector to be diagonalized.
2. `integer, intent(IN) :: nvec`
   The number of eigenvectors (eigen-modes) to be computed.
   If positive, eigen modes are computed from the smallest.
   If zero, all eigenvalues without eigenvectors are computed from the smallest.
   (if negative, they are computed from the largest, but not supported currently.)
3. `real(8), intent(INOUT) :: a(1:lda,*)`
   A symmetric matrix to be diagonalized (upper triangular part is available), whose size must be equal or larger than the values obtained by calling `eigen_get_matdims`.
   Array content is destroyed upon the subroutine termination, but
   `a(1, 1)` returns the flops count on exit.
4. `integer, intent(IN) :: lda`
   The leading dimension of array a, which must be equal or larger than
   the value obtained by calling `eigen_get_matdims`.
5. `real(8), intent(OUT) :: w(1:n)`
   The eigenvalues of matrix a stored in the ascending/descending order according to nvec.
6. `real(8), intent(OUT) :: z(1:ldz,*)`
   The eigenvectors of matrix a. The size of array must be equal or larger than
   the values obtained by calling `eigen_get_matdims`.
7. `integer, intent(IN) :: ldz`
   The leading dimension of array z, which must be equal or larger than
   the value obtained by calling `eigen_get_matdims`.
8. `integer, optional, intent(IN) :: m_forward = 48`
   The block factor of the forward Householder transformation (must be even).
   Default is 48.
9. `integer, optional, intent(IN) :: m_backward = 128`
   The backward block factor in the Householder transformation. Default is 128.
10. `character(*), optional, intent(IN) :: mode = 'A'`
    'A' : all eigenvalues and corresponding eigenvectors (default)
    'N' : eigenvalues only
    'X' : add to mode 'A' to improve accuracy.

4.6 eigen_h

Is the driver routine to compute the standard Hermite eigenvalue problem. This driver is a collective operation, and all processes belonging to the calling process group must participate in the call. Note that this driver routine is newly supported from version 2.11.

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

1. `integer, intent(IN) :: n`
   Dimensions of the matrix and vector to be diagonalized.
2. `integer, intent(IN) :: nvec`
   The number of eigenvectors (eigen-modes) to be computed.
   If positive, eigen modes are computed from the smallest.
   If zero, all eigenvalues without eigenvectors are computed from the smallest.
   (if negative, they are computed from the largest, but not supported currently.)
3. `complex(8), intent(INOUT) :: a(1:lda,*)`
   A Hermite matrix to be diagonalized (upper triangular part is available), whose size must be equal or larger than the values obtained by calling `eigen_get_matdims`.
   Array content is destroyed upon the subroutine termination, but
   `DBLE(a(1, 1))` returns the flops count on exit.
4. `integer, intent(IN) :: lda`
   The leading dimension of array a, which must be equal or larger than
   the value obtained by calling `eigen_get_matdims`.
5. `real(8), intent(OUT) :: w(1:n)`
   The eigenvalues of matrix a stored in the ascending/descending order according to nvec.
6. `complex(8), intent(OUT) :: z(1:ldz,*)`
   The eigenvectors of matrix a. The size of array must be equal or larger than
   the values obtained by calling `eigen_get_matdims`.
7. `integer, intent(IN) :: ldz`
   The leading dimension of array z, which must be equal or larger than
   the value obtained by calling `eigen_get_matdims`.
8. `integer, optional, intent(IN) :: m_forward = 48`
   The block factor of the forward Householder transformation (must be even).
   Default is 48.
9. `integer, optional, intent(IN) :: m_backward = 128`
   The backward block factor in the Householder transformation. Default is 128.
10. `character(*), optional, intent(IN) :: mode = 'A'`
    'A' : all eigenvalues and corresponding eigenvectors (default)
    'N' : eigenvalues only
    'X' : add to mode 'A' to improve accuracy.

4.7 eigen_get_version

Returns the version information of EigenExa to the arguments. The format of the argument `version` has been revised since version 2.11: from the one-digit reference to the two-digit representation.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

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

1. `integer, intent(OUT) :: version`
   The version number represented in a six-digit format,
   in which each two-digit refer to major version, minor version, and
   patch level from the upper to lower digits.
2. `character(*), optional, intent(OUT) :: date`
   The release date.
3. `character(*), optional, intent(OUT) :: vcode`
   The code name corresponding to the release version.

4.8 eigen_show_version

Displays the version information of EigenExa on `stdout` (standard output).

```
subroutine eigen_show_version( )
```

4.9 eigen_get_matdims

Returns the recommended array size in EigenExa. The user should dynamically allocate the local array using the array dimensions obtained from this function `(nx,ny)` or larger. The entire matrix is distributed in a 2D-cyclic fashion, in short (CYCLIC,CYCLIC) distribution. The "mode" option allows the users to specify the memory usage and the efficient alignment of the array shape in the order of Minimal<LineAligned<Optimal. The option is available from version 2.6 later.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

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

1. `integer, intent(IN) :: n`
   A dimension of the matrix to be diagonalized.
2. `integer, intent(OUT) :: nx`
   The lower bound or recommended value of the leading dimension of a 2D-array such as `a(:,:)`, `b(:,:)`, and `z(:,:)`.
3. `integer, intent(OUT) :: ny`
   The lower bound value of the second index of a 2D-array such as `a(:,:)`, `b(:,:)`, and `z(:,:)`.
4. `character(*), optional, intent(IN) :: mode = 'O'`
   Option to the memory usage to specify the matrix dimensions.
   `'M'` : Minimal, returns minimal dimensions,
   `'L'` : LineAligned, returns the dimension aligned foe cache line access, and
   `'O'` : Optimal, returns the dimension to avoid cache thrashing (default).

4.10 eigen_memory_internal

This subroutine returns the size of memory that is dynamically allocated internally during EigenExa is called. Users should know the return value of this function and should avoid running out of memory. In version 2.3c, a specification was changed to return an 8-byte integer (`integer(8)`). If the return value is negative (-1 is returned in most cases), the matrix size is too large to represent within the 32bit integer value (`integer(4)`) in the EigenExa library. It alerts the users in advance of the possible risk of overflowing when calculating indices.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

`integer(8) function eigen_memory_internal( n, lda, ldz, m_f, m_b )`

1. `integer, intent(IN) :: n`
   A dimension of the matrix to be diagonalized.
2. `integer, intent(IN) :: lda`
   The leading dimension of array `a`.
3. `integer, intent(IN) :: ldz`
   The leading dimension of array `z`.
4. `integer, intent(IN) :: m_f`
   Householder forward transformation blocksize (must be even)
5. `integer, intent(IN) :: m_b`
   Householder backward transformation blocksize

4.11 eigen_get_comm

Returns the MPI communicators that are generated by the call of `eigen_init()`.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

`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.

4.12 eigen_get_procs

Returns the process information that corresponds to the processes generated by the call of `eigen_init()`.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

`subroutine eigen_get_procs( procs, x_procs, y_procs )`

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

4.13 eigen_get_id

Returns the process ID information that corresponds to the processes generated by the call of `eigen_init()`. Here, the process IDs are managed as integers, starting with 1, and 'the MPI rank' = 'obtained process ID -1' is satisfied.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

```
subroutine eigen_get_id( id, x_id, y_id )
```
1. `integer, intent(OUT) :: id`
   The process ID defined in `comm`.
2. `integer, intent(OUT) :: x_id`
   The process ID defined in `x_comm`.
3. `integer, intent(OUT) :: y_id`
   The process ID defined in `y_comm`.

4.14 eigen_get_errinfo

Returns the internal error status information. If `info` is zero, no error occurred. If non-zero value, some errors happened, and the internal error status has been inherited from eigenvalue solver routines, for example, the DC solvers called internally. The function is just available on version 2.11 or later. Thus, the API is provisional, and the semantics of arguments and the specific meaning of returned value may be revised in future versions.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

```
subroutine eigen_get_errinfo( info )
```
1. `integer, intent(OUT) :: info`
   The error status information
   Curretly (version 2.11), it just reflects the info code from the internal DC routine compatible to `PDSTEDC` in ScaLAPACK.

4.15 eigen_loop_start

Returns the initial loop value in a sense of the local loop structure correspoding to the specified initial value of the global loop. Note that the global loop start value must be greater than or equal to 1. The number of processes in the communicator must be $1 \leq \texttt{inod} \leq \texttt{nnod}$ (or the number of participant processes in the communicator if `pdir` is specified). If `pdir` is not properly specified, 0 is returned.

This function has a generic name interface, and appropriate subfunctions are called according to the arguments.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

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

1. `integer, intent(IN) :: istart`
   The initial value of the global loop.
2. `integer, intent(IN) :: nnod`
   The number of processes.
3. `integer, intent(IN) :: inod`
   The process ID.

`integer function eigen_loop_start( istart, pdir, `*`inod`*` )`

1. `integer, intent(IN) :: istart`
   The initial value of the global loop.
2. `character(*), intent(IN) :: pdir`
   A symbol to specify the communicator; base, row or column.
   `'W'` or `'T'` : `comm`
   `'X'` or `'R'` : `x_comm`
   `'Y'` or `'C'` : `y_comm`
3. `integer, intent(IN), optional :: inod`
   The process ID. Default is the corresponding rank-ID specified by `pdir`.

For example, if you parallelize the following sequential loop organized in the communicator `x_comm`, you may get the local loop range with `eigen_loop_start`, as well as `eigen_loop_end`. The called function `eigen_translate_l2g` in the transformed loop returns a global index value corresponding to the local index value indicated by the local loop counter.

```
do i=J,K
    ....
enddo
```

↓↓↓↓↓↓

```
i_start=eigen_loop_start(J, 'X')
i_end  =eigen_loop_end  (K, 'X')
do i_local=i_start, i_end
  i=eigen_translate_l2g(i_local, 'X')
  ....
enddo
```

4.16 `eigen_loop_end`

Returns the teminal value in a sense of the local loop structure correspoding to the specified terminal value of the global loop. Note that the global loop start value must be greater than or equal to 1. The number of processes in the communicator must be $1 \leq$ `inod` $\leq$ `nnod`(or the number of participant processes in the communicator if `pdir` is specied). If `pdir` is not properly specified, -1 is returned. This function has a generic name interface, and appropriate subfunctions are called according to the arguments.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

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

1. `integer, intent(IN) :: iend`
   The terminal value of the global loop.
2. `integer, intent(IN) :: nnod`
   The number of processes.
3. `integer, intent(IN) :: inod`
   The process ID.

`integer function eigen_loop_end( iend, pdir,` *`inod`* `)`

1. `integer, intent(IN) :: iend`
   The terminal value of the global loop.
2. `character(*), intent(IN) :: pdir`
   A symbol to specify the communicator; base, row or column.
   `'W'` or `'T'` : `comm`
   `'X'` or `'R'` : `x_comm`
   `'Y'` or `'C'` : `y_comm`
3. `integer, intent(IN), optional :: inod`
   The process ID. Default is the corresponding rank-ID specified by `pdir`.

4.17 `eigen_loop_info`

is a combined procedure with `eigen_loop_start` and `eigen_loop_end`, which returns both the initial value and terminal value at the same time. Also, other specific behaviors follow the same as the two functions.

`subroutine eigen_loop_info( istart, iend, lstart, lend, nnod, inod )`

1. `integer, intent(IN) :: istart`
   The initial value of the global loop.

2. `integer, intent(IN) :: iend`
   The terminal value of the global loop.
3. `integer, intent(OUT) :: lstart`
   The initial value of the local loop.
4. `integer, intent(OUT) :: lend`
   The terminal value of the local loop.
5. `integer, intent(IN) :: nnod`
   The number of processes.
6. `integer, intent(IN) :: inod`
   The process ID.

`subroutine eigen_loop_info( istart, iend, lstart, lend, pdir, `*`inod`*` )`

1. `integer, intent(IN) :: istart`
   The initial value of the global loop.
2. `integer, intent(IN) :: iend`
   The terminal value of the global loop.
3. `integer, intent(OUT) :: lstart`
   The initial value of the local loop.
4. `integer, intent(OUT) :: lend`
   The terminal value of the local loop.
5. `character(*), intent(IN) :: pdir`
   A symbol to specify the communicator; base, row or column.
   `'W'` or `'T'` : `comm`
   `'X'` or `'R'` : `x_comm`
   `'Y'` or `'C'` : `y_comm`
6. `integer, intent(IN), optional :: inod`
   The process ID. Default is the corresponding rank-ID specified by `pdir`.

4.18 eigen_translate_l2g

Returns the global index corresponding to the local index value (1 or greater) indicated by the local counter. The number of processes in the communicator must be $1 \leq$`inod`$\leq$`nnod`(or the number of participant processes in the communicator if `pdir` is specified). If `pdir` is not properly specified, -1 is returned. This function has a generic name interface, and appropriate subfunctions are called according to the arguments.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

integer function eigen_translate_l2g(ictr, nnod, inod)
1. integer, intent(IN) :: ictr
   Local counter.
2. integer, intent(IN) :: nnod
   The number of processes.
3. integer, intent(IN):: inod
   The process ID.

integer function eigen_translate_l2g(ictr, pdir, *inod*)
1. integer, intent(IN) :: ictr
   Local counter.
2. character(*), intent(IN) :: pdir
   A symbol to specify the communicator; base, row or column.
   'W' or 'T' : comm
   'X' or 'R' : x_comm
   'Y' or 'C' : y_comm
3. integer, intent(IN), optional :: inod
   The process ID. Default is the corresponding rank-ID specified by pdir.

4.19 eigen_translate_g2l

Returns the local index corresponding to the global index value (1 or greater) indicated by the local counter, whereas it is not certain that the caller process is the owner process. The number of processes in the communicator must be $1 \leq$inod$\leq$nnod(or the number of participant processes in the communicator if pdir is specified). If pdir is not properly specified, -1 is returned. This function has a generic name interface, and appropriate subfunctions are called according to the arguments.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

integer function eigen_translate_g2l(ictr, nnod, inod)
1. integer, intent(IN) :: ictr
   Global counter.
2. integer, intent(IN) :: nnod
   The number of processes.
3. integer, intent(IN) :: inod
   The process ID.

`integer function eigen_translate_g2l( ictr, pdir, *inod* )`

1. `integer, intent(IN) :: ictr`
   Global counter.
2. `character(*), intent(IN) :: pdir`
   A symbol to specify the communicator; base, row or column.
   `'W'` or `'T'` : `comm`
   `'X'` or `'R'` : `x_comm`
   `'Y'` or `'C'` : `y_comm`
3. `integer, intent(IN), optional :: inod`
   The process ID. Default is the corresponding rank-ID specified by `pdir`.

4.20 eigen_owner_node

Returns the owner ID corresponding to the specified global index value (1 or greater). The number of processes in the communicator must be $1 \leq$`inod`$\leq$`nnod`(or the number of participant processes in the communicator if `pdir` is specified). If `pdir` is not properly specified, -1 is returned. This function has a generic name interface, and appropriate subfunctions are called according to the arguments.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

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

1. `integer, intent(IN) :: ictr`
   Global counter.
2. `integer, intent(IN) :: nnod`
   The number of processes.
3. `integer, intent(IN) :: inod`
   The process ID.

`integer function eigen_owner_node( ictr, pdir, *inod* )`

1. `integer, intent(IN) :: ictr`
   Global counter.
2. `character(*), intent(IN) :: pdir`
   A symbol to specify the communicator; base, row or column.
   `'W'` or `'T'` : `comm`
   `'X'` or `'R'` : `x_comm`
   `'Y'` or `'C'` : `y_comm`
3. `integer, intent(IN), optional :: inod`
   The process ID. Default is the corresponding rank-ID specified by `pdir`.

4.21 eigen_owner_index

Returns the corresponding local index if the caller process is the owner of the specified global index value (1 or greater). The number of processes in the communicator must be $1 \le$inod$\le$nnod(or the number of participant processes in the communicator if pdir is specified). If pdir is not properly specified, -1 is returned. This function has a generic name interface, and appropriate subfunctions are called according to the arguments.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

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

1. `integer, intent(IN) :: ictr`
   Global counter.
2. `integer, intent(IN) :: nnod`
   The number of processes.
3. `integer, intent(IN) :: inod`
   The process ID.

`integer function eigen_index_node( ictr, pdir, ` *`inod`* ` )`

1. `integer, intent(IN) :: ictr`
   Global counter.
2. `character(*), intent(IN) :: pdir`
   A symbol to specify the communicator; base, row or column.
   'W' or 'T' : comm
   'X' or 'R' : x_comm
   'Y' or 'C' : y_comm
3. `integer, intent(IN), optional :: inod`
   The process ID. Default is the corresponding rank-ID specified by pdir.

4.22 eigen_convert_ID_xy2w

Converts the 2D process ID to the process ID on the base communicator according to the grid major. It does not check whether the input and return values of the process IDs are within the correct range.

This function is a local operation. Although this function is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

```
integer function eigen_convert_ID_xy2w( xinod, yinod )
```
1. `integer, intent(IN) :: xinod`
   The process ID in `x_comm`
2. `integer, intent(IN) :: yinod`
   The process ID in `y_comm`.

4.23 eigen_convert_ID_w2xy

Converts the process ID on the base communicator to the 2D process ID according to the grid major. It does not check whether the input and return values of the process IDs are within the correct range.

This subroutine is a local operation. Although this subroutine is not thread-safe (because the arguments are specified in a Fortran pointer fashion), it can be called during multi-threading if proper exclusive control of the arguments is provided.

```
subroutine eigen_convert_ID_w2xy( inod, xinod, yinod )
```
1. `integer, intent(IN) :: inod`
   Process ID in the Base communicator.
2. `integer, intent(OUT) :: xinod`
   The process ID in `x_comm`.
3. `integer, intent(OUT) :: yinod`
   The process ID in `y_comm`.

4.24 KMATH_EIGEN_GEV

Is a generalized eigenvalue computing driver routine that uses EigenExa as the internal eigenvalue computing engine. Since version 2.6, this function is officially bundled in the EigenExa library. A user has to make a separate package in a case prior to 2.4. In this driver, `eigen_sx` is called to compute the eigenpairs via transformation to a pentadiagonal matrix as $Cy = \lambda y$, where $B = X_B \Lambda_B X_B^T \Rightarrow Y_B = X_B \Lambda_B^{1/2} \Rightarrow C = Y_B^{-1} A Y_B^{-T}, y = Y_B^T x$. The constraints on the driver are similar to those on `eigen_sx`.

```
subroutine kmath_eigen_gev( n, nvec, a, lda, b, ldb, w, z, ldz )
```
1. `integer, intent(IN) :: n`
   Dimensions of the matrix and vector to be solved.
2. `integer, intent(IN) :: nvec`
   The number of eigenvectors (eigenmodes) to be computed.
   If positive, eigenmodes are computed from the smallest.

If zero, all eigenvalues without eigenvectors are computed from the smallest.
(if negative, they are computed from the largest, but not supported currently.)

3. `real(8), intent(INOUT) :: a(1:lda,*)`
   A target matrix pencil $(A - \lambda B)$;
   matrix A is real symmetric (upper triangular part is available), whose size must be equal or larger than the values obtained by calling `eigen_get_matdims`.
   Array content is destroyed upon the subroutine termination.
4. `integer, intent(IN) :: lda`
   The leading dimension of array `a`, which must be equal or larger than the value obtained by calling `eigen_get_matdims`.
5. `real(8), intent(INOUT) :: b(1:ldb,*)`
   A target matrix pencil $(A - \lambda B)$;
   matrix B is real symmetric (upper triangular part is available), whose size must be equal or larger than the values obtained by calling `eigen_get_matdims`.
   The matrix for transformation to the standard eigenvalue problem is stored upon subroutine termination.
6. `integer, intent(IN) :: ldb`
   The leading dimension of array `b`, which must be equal or larger than the value obtained by calling `eigen_get_matdims`.
7. `real(8), intent(OUT) :: w(1:n)`
   The eigenvalues of the matrix pencil $(A - \lambda B)$ stored in the ascending/descending order according to nvec .
8. `real(8), intent(OUT) :: z(1:ldz,*)`
   The eigenvectors of the generalized eigenproblem with B-orthogonal.
   The size of array must be equal or larger than the values obtained by calling `eigen_get_matdims`.
9. `integer, intent(IN) :: ldz`
   The leading dimension of array `z`, which must be equal or larger than the value obtained by calling `eigen_get_matdims`.

Chapter 5

Other Considerations

5.1 Regarding upper/lower compatibilities

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

Updating EigenExa from version 2.3 to 2.4, module management and naming rules have been changed. Therefore, users using versions 2.3 or earlier should be careful when updating to higher versions.

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 [18], 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 several conditions are met; for example, a linked subordinate library such as BLAS or LAPACK produces an error, or memory allocation in EigenExa got failed. In short, these error occurrence do not resume the main caller routine but terminate the main program.

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

EigenExa@ml.riken.jp).

5.4 Shared library handling in versions 1.x

In former versions (1.x), Shared libraries were not supported, because at the time of their development it was not possible to guarantee complete and collision-free resolution of function names when shared libraries are being used (with specific 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 be performed solely at the user's responsibility.

EigenExa versions 2.x and later are both static- and shared-library capable, a development achieved with the technical cooperation of former Team Leader Toshiyuki Maeda and other members of the HPC Usability Research Team at the RIKEN Advanced Institute for Computational Science in 2015. Eventually, from version 2.4 both libraries are built in the make phase in default. When executing, always remember to make the appropriate settings for the environment variables (such as `LD_LIBRARY_PATH`).

5.5 Known bugs and the best workaround

5.5.1 Numerical reproducibility in Intel compiler and Intel MPI

Some MPI implementations adopt non-reproducible algorithms for collective communication, especially for reduction operations, including arithmetic operations, in order to improve communication performance. The Intel MPI [19] clearly states that bit-level reproducibility of real number reduction operations (e.g., `MPI_SUM`) is not ensured due to the adoption of network topology-aware algorithms. EigenExa includes several groups of processes that perform redundant calculations. Since the bit-level consistency between these process groups is required, the operation of EigenExa may become abnormal if numerical reproducibility fails. Therefore, we try to avoid such a situation in our internal implementation as much as possible. However, the current implementation of the algorithm does not solve the essential part of the problem.

In order to solve such reproducibility problems, we recommend two options. First for the runtime options, Intel MPI provides the environment variable `I_MPI_CBWR`. EigenExa handles the additional runtime attributes to the communicator to achieve the same functionality as if the environment variable were specified internally. Nevertheless, this feature may not work correctly if the user specifies the communication algorithm explicitly. Therefore, when using Intel MPI, it is recommended that the environment variable `I_MPI_CBWR` be set to 2.[1]

```
% export I_MPI_CBWR=2 (Bash)
% setenv I_MPI_CBWR 2 (C shell)
```

Besides, the second option for the compilation step is recommended. The user has to specify no options to the environment variables (`CFLAGS` and `FFLAGS`) of the `configure`

[1] However, the countermeasure does not guarantee perfect bitwise numerical reproducibility, even on other numerical libraries or user's source segments.

script that may affect accuracy, for example, weaker options than `-fp-model=strict`, to avoid compiler-level numerical optimization problems. Note that the `configure` script automatically adds the option `-fp-model=strict` if the Intel compiler is detected.

Appendices

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 outlines 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 [20, 21, 22, 23, 24, 25, 26, 27, 28].

A.2 Various approaches and related projects

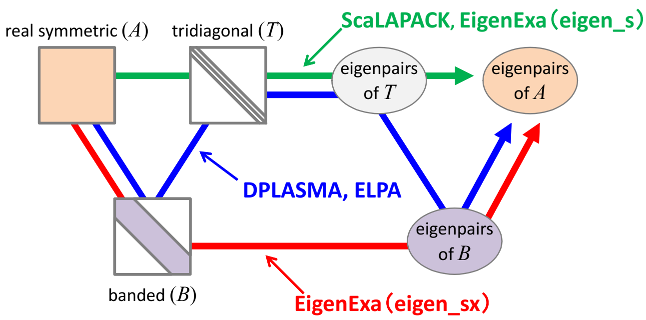


Figure A.1: Various approaches to eigenvalue computation for real symmetric dense matrices.

Let us begin with a brief introduction to the essential 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 [29] (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 [30] and DPLASMA [31], 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 the transformation from dense to band. The byte/flop ratio required in this transformation is smaller than that in a 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 enormous 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, remain (for obtaining the matrix V) and is, therefore, a significant bottleneck for high-performance computing.

In the second step, the divide-and-conquer method proposed by Cuppen [32] 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):

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

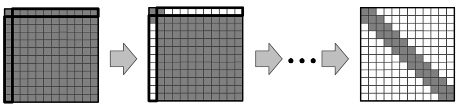
at low cost (only for computing a small matrix S), the back transformation is usually computed via matrix-matrix multiplications (Level-3 BLAS):

$$H_1 \cdots H_{N-2} Y = (I - U_1 S_1 U_1^\top) \cdots (I - U_M S_M U_M^\top) Y \to X, \tag{A.5}$$

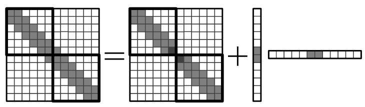
where

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

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:

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

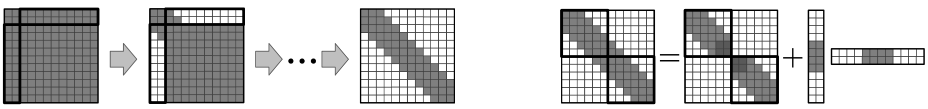
to transform every two columns (two rows) of the input matrix into a pentadiagonal matrix (Fig. A.3(a)), where

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

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 principle of the divide-and-conquer method for a tridiagonal matrix twice and compute the eigenvalues and eigenvectors of the pentadiagonal matrix [33].

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_s`, 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-/Penta-diagonalization

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 about the same (at least for the highest term) for the two routines; the 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 similar reasons, the amount of data transferred 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 work 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 more considerable 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 significant problem in 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 did 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 in 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.

Appendix B

Release notes

The release history of EigenExa is listed in `ReleaseNotes.txt`.

B.1 Version 2.11 (December 1, 2021)

- **[Upgrade]** Experimental support for Hermite solver, namely, `eigen_h`.
- **[Serious]** Fix the numerical error happened to be included in v2.10, where the tall-skinny QR coded in `eigen_prd_t4x.F` of `eigen_sx` was too sensitive to treat tiny values and forced-double truncation. But, it might depend on the compiler version and the code generator.
- Modify pointer attribution to 'allocatable' to avoid automatic deallocation on the exit of callee routines.
- Some code modifications are applied to pass strong debugging tools with respect to Fortran 95 and some of Fortran 2003 extensions.
- Fix some bugs, for exmaple, missing private attribution to some variables for OpenMP.

B.2 Version 2.10 (Octorber 17, 2021)

- **[Serious]** Bug fix for violation of the result of allreduce in DC. It happened very rarely when data to be transferred was shorter than the number of processes participating.
- **[Serious]** Bug fix for inconsistent API interpretation of DLAED4, when K is less than or equal to 2. This bug happened when a lot of deflations are carried out, and sub-matrices are shrunk tiny as 1 or 2. So, it is infrequent to see.
- **[Serious]** Bug fix for non-deterministic behavior of the DC branch, which happened if an uninitialized variable referred in the brach condition, and is affected by the side-effects of other modules, etc. It was fixed when 2.9 was released but noted in the release.

- Reduce the internal data capacity in the TRD and DC routines.
- Fix the installation of Fortran modules.

B.3 Version 2.9 (September 24, 2021)

- Modify the flops count precise in DC kernels.
- Modify trbak not to multiply D^{-1} and `TRSM`.
- Add enable/disable-switch for building the shared library.
- Modify to detect the memory allocation fault.

B.4 Version 2.8 (August 20, 2021)

- **[Minor]** Modify the DC kernel to reduce intermediate buffer storage.
- Bug fix on a t1 loop structure
- Updated the error check routine
- Fixed on Makefile to add the missing fortran module.

B.5 Version 2.7 (April 1, 2021)

- **[Minor]** Modify the compilation rules corresponding to static/shared libraries defined in `src/Makefile.am`.
- Performance tweak with a modification of the compilation options not to use -fPIC when build a static library.
- License document is packed as an independent file (the license notice was stated in User's manual for version 2.6).

B.6 Version 2.6 (November 1, 2020)

- **[Upgrade]** This version applies a communication avoidance technique to the householder tridiagonalization together with, new process mapping for the load balance of the divide and conquer method.

B.7 Version 2.5 (August 1, 2019)

- **[Internal]** Refine the data distribution in the divide and conquer algorithm routine.
- This version is only internal management version and not published.

B.8 Version 2.4b (August 20, 2018)

- **[Serious]** Bug fix for incorrect data redistribution, which might violate allocated memory. The bug might have happened in the case that the number of processes, $P = P_x * P_y$, is large, and P_x and P_y are not equal but nearly equal.
- This version is for only bug fix for the serious one.

B.9 Version 2.3m (August 20, 2018)

- **[Serious]** Bug fix for incorrect data redistribution, which might violate allocated memory. The bug might have happened in the case that the number of processes, $P = P_x * P_y$, is large, and P_x and P_y are not equal but nearly equal.
- This version is for only bug fix for the serious one.

B.10 Version 2.4p1 (May 25, 2017)

- **[Serious]** Bug fix for incorrect data redistribution in `eigen_s`.
- Major change with Autoconf -and- Automake framework.

B.11 Version 2.4 (April 18, 2017)

- **[Upgrade]** Major change with Autoconf -and- Automake framework

B.12 Version 2.3k2 (April 12, 2017)

- Communication Avoiding algorithms to the `eigen_s` driver.
- The optional argument `nvec` is available, which specifies the number of eigenvectors to be computed from the smallest. This version does not employ the special algorithm to reduce the computational cost. It only drops off the unnecessary eigenmodes in the backtransformation.

B.13 Version 2.3d (July 07, 2015)

- Tuned up the parameters according to target architectures.
- Introduce a sort routine in bisect.F and bisect2.F for eigenvalues.
- Modify the algorithm to create reflector vectors in `eigen_prd_t4x.F`

- Modify the matrix setting routine to load the mtx (Matrix Market) format file via both `'A.mtx'` and `'B.mtx'`.
- Re-format the source code by the fortran-mode of emacs and extra rules.

B.14 Version 2.3c (April 23, 2015)

- Fix bug on flops count of `eigen_s` which returned incorrect value due to missing initialization in `dc2.F`.
- This bug is found in version 2.3a and version 2.3b.
- Minor change on timer routines.
- Minor change on broadcast algorithm in `comm.F`.

B.15 Version 2.3b (April 15, 2015)

- Minor change to manage the real constants.
- Minor change to use Level 1 and 2 BLAS routines.
- Minor change to preserve invalid or oversized matrices.
- Minor change of Makefile to allow '-j' option.

B.16 Version 2.3a (April 14, 2015)

- Minor change on thread parallelization of `eigen_s`.
- Minor change of the API's for timer routines.
- Fix the unexpected optimization of rounding errors in `eigen_dcx()`.

B.17 Version 2.3 (April 12, 2015)

- Bug fix on the benchmark program.
- Refine the race condition in the backtransformation routine.
- Introduce Communication Avoiding algorithms to the `eigen_s` driver.

B.18 Version 2.2d (March 20, 2015)

- Bug fix on the timer print part in `trbakwy4.F` not to do zero division.
- Modify the synchronization point in `eigen_s`.
- Modify thread parallelization in `eigen_dc2()` and `eigen_dcx()`.

B.19 Version 2.2c (March 10, 2015)

- Bug fix on the benchmark program.
- Add the `make_inc` file for an NEC SX platform.

B.20 Version 2.2b (October 30, 2014)

- Introduce new API to query the current version.
- Introduce the constant `eigen_NB`=64, which refers to the block size for cooperative work with the ScaLAPACK routines.
- Correct the requred array size in `eigen_mat_dims()`.
- Improve the performance of test matrix generator routine `mat_set()`.
- Add the listing option of test matrices in `eigenexa_benchmark/`.

B.21 Version 2.2a (June 20, 2014)

- Fix minor bug of `Makefile`, `miscC.c` and etc for BG/Q.
- Modify the initialization process not to use invalid communicators.
- Comment out the calling `BLACS_EXIT()` in `eigen_free()`.

B.22 Version 2.2 (April 10, 2014)

- Arrange the structure of source directory.
- Reversion of the DC routines back to version 1.3a to avoid bug.
- Hack `miscC.c` to be called from IBM BG/Q.
- Fix bug on the benchmark program for exceptional case of `MPI_COMM_NULL`.
- Fix bug on `eigen_s` with splitted communicator.

- Update machine depended configuration files.
- Experimental support of building a shared library

B.23 Version 2.1a (Feb 23, 2014)

- Fix bug on the benchmark program.

B.24 Version 2.1 (Feb 10, 2014)

- Fix bug on `eigen_sx`: it gave wrong results when N=3.
- Modify the bisect2 by a pivoting algorithm.
- Update the test program '`eigenexa_benchmark`' in order to check accuracy with several test matrices and computing modes.
- Tune performance for K computer and Fujitsu FX10 platforms.
- Add `make_inc` file for a BlueGeneQ platform, but it is not official support, just an experimental.

B.25 Version 2.0 (Dec 13, 2013)

- **[Upgrade]** Add `eigen_s`, which adopts the conventional 1-stage algorithm.
- Add optional modes to compute only eigenvalues and to improve the accuracy of eigenvalues.
- Modify to support a thread mode with any number of threads.
- Tune performance for K computer and Fujitsu FX10 platforms.

B.26 Version 1.3a (Sep 21, 2013)

- Fix bug on synchronization mechanism of `eigen_trbakwyx()`.

B.27 Version 1.3 (Sep 20, 2013)

- Fix bug on `eigen_init()` in initialization with `MPI_Cart`'s or `MPI_COMM_NULL`'s.
- Add test programs to check several process patterns.

B.28 Version 1.2 (Sep 17, 2013)

- Fix bug on benchmark code in making a random seed.
- Modify to support upto 64-thread running.

B.29 Version 1.1 (Aug 30, 2013)

- Fix bug on data-redistribution row vector to column vector when $P = p * q$ and p and q have common divisor except themselves.
- Optimize data redistribution algorithm in `dc_redist[12].F`.

B.30 Version 1.0 (Aug 1, 2013)

- **[Release]** This is the first release
- Standard eigenvalue problem for a dense symmetric matrix by a novel one-stage algorithm

Acknowledgements

We would like to thank all the members of RIKEN Center for Computational Science (R-CCS) and all of the EigenExa development group members for their dedicated supports. We received many feedbacks from EigenExa users. We could not release EigenExa without the efforts of the members above mentioned.

The EigenExa project has been supported by the following funds and awarded by HPC projects.

1. JST CREST, “Development of System Software Technologies for post-Peta Scale High Performance Computing, ” (FY2011–FY2015), and partly collabrated with the ESSEX-II project in SPPEXA-phase2 (2016-18)

2. MEXT KAKENHI, Kiabn-B, Grant Numbers 21300013 (FY2012), Kiban-A 23240005 (FY2011–FY2013), Kiban-B 15H02709 (FY2015–FY2017), Kiban-B 19H04127 (FY2019–FY2021).

3. HPCI System computers, General Use Projects (the K computer), project ID hp140069 (FY2014), project ID hp120170 (FY2012–2013)

4. Computational resources supported by the Grant-in-aid for the K computer, so called 「「京」調整高度化枠」 in Japanese originally, poroject ID ra000005 (FY2012–2019)

5. FX10 supercomputer system, "Large-Scale HPC Challenge" awarded from Information Technology Center, The University of Tokyo (FY2013)

6. Flagship 2020 (post-K) project (FY2014–2020)

7. Computational resources for the pre-operation of the supercomputer Fugaku, so called 「「富岳」共用前評価環境」 in Japanese originally, poroject ID ra000006 (FY2020)

8. ‘Startup Preparation Project’ for Fugaku Preliminary Use Projects in FY 2020, so called 「「富岳」試行的利用課題（利用準備課題）」 in Japanese originally, project ID hp200263 (FY2020), and hp200313 (FY2021 H1)

9. Computational resources of the supercomputer Fugaku provided by the RIKEN Center for Computatioal Science, namely 「高度化・利用拡大枠」 in Japanese, poroject ID ra000006 (FY2021-)

The EigenK project, the predecessor of the EigenExa project, was also supported by the following fund. We would like to express our deepest gratitude to our great co-works at the Center for Computational Science and e-Systems, Japan Atomic Energy Agency (CCSE-JAEA).

10. 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] 今村俊幸, 「T2Kスパコンにおける固有値ソルバの開発」, 東京大学スーパーコンピューティングニュース, Vol.11, No.6, pp.12-32 (2009).
http://www.cc.u-tokyo.ac.jp/support/press/news/VOL11/No6/200911imamura.pdf

[3] 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) .

[4] 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.

[5] What is K? | RIKEN Center for Computational Science,
http://www.r-ccs.riken.jp/en/k-computer/about/

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

[7] スパーコンピュータ「富岳」プロジェクト (Supercomputer Fugaku project),
https://www.r-ccs.riken.jp/jp/fugaku/project.html (日本語) https://www.r-ccs.riken.jp/en/fugaku/project (English)

[8] 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).
https://www.r-ccs.riken.jp/labs/lpnctrt/assets/img/EPASA2014_dense_poster_ImamuraT_only.pdf

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

[10] 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.

[11] 深谷猛, 今村俊幸, 「FX10 4800ノードを用いた密行列向け固有値ソルバ EigenExa の性能評価」, 東京大学スーパーコンピューティングニュース, Vol.16 No.3, pp.20-27 (2014). http://www.cc.u-tokyo.ac.jp/support/press/news/VOL16/No3/09_User201405-1.pdf

[12] 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.

[13] Takeshi Fukaya, and Toshiyuki Imamura, "Performance Evaluation of the Eigen Exa Eigensolver on Oakleaf-FX: Tridiagonalization Versus Pentadiagonalization", Proceedings of the Parallel and Distributed Processing Symposium Workshop (IPDPSW, PDSEC 2015), p.960-969, doi:10.1109/IPDPSW.2015.128, 2015.

[14] Yusuke Hirota, Daichi Mukunoki, Susumu Yamada, Narimasa Sasa, Toshiyuki Imamura, and Masahiko Machida, "Performance of Quadruple Precision Eigenvalue Solver Libraries QPEigenK & QPEigenG on the K Computer", poster presentation, ISC2016, Best poster award in 'HPC in Asia'.

[15] Toshiyuki Imamura, Takeshi Fukaya, Yusuke Hirota, Susumu Yamada, and Masahiko Machida: "CAHTR: Communication-Avoiding Householder Tridiagonalization", Proceedings of ParCo2015, Advances in Parallel Computing, Vol.27: Parallel Computing: On the Road to Exascale, p.381-390, doi:10.3233/978-1-61499-621-7-381, 2016

[16] Yusuke Hirota, and Toshiyuki Imamura, "Performance Analysis of a Dense Eigenvalue Solver on the K Computer", Proceedings of the 36th JSST Annual International Conference on Simulation Technology, Oct. 2017.

[17] Takeshi Fukaya, Toshiyuki Imamura, and Yusaku Yamamoto, "A Case Study on Modeling the Performance of Dense Matrix Computation: Tridiagonalization in the EigenExa Eigensolver on the K Computer". Proceedings of the Parallel and Distributed Processing Symposium Workshop (IPDPSW, iWAPT 2018), pp. 1113-1122, doi:10.1109/IPDPSW.2018.00171, 2018

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

[19] Intel MPI Library Developer Reference for Linux OS, Developer Reference, especially, algorithm selection method and further descriptions about collective communication algorithms are described. https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/environment-variable-reference/i-mpi-adjust-family-environment-variables.html

[20] 一般社団法人 日本計算工学会 編, 長谷川秀彦, 今村俊幸, 山田進, 櫻井鉄也, 荻田武史, 相島健助, 木村欣司, 中村佳正 著 計算力学レクチャーコース 「固有値計算と特異値計算」, 丸善出版 (2019)

[21] 杉原正顯, 室田一雄, 「線形計算の数理」, 岩波書店 (2009).

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

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

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

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

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

[27] 山本有作, 「密行列固有値解法の最近の発展(I) : Multiple Relatively Robust Representationsアルゴリズム」, 日本応用数理学会論文誌, Vol.15, No.2, pp.181–208 (2005).

[28] 山本有作, 「密行列固有値解法の最近の発展(II) : マルチシフトQR法」, 日本応用数理学会論文誌, Vol.16, No.4, pp.507–534 (2006).

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

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

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

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

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