Lis User Manual
Version 1.3.33

Cover: Ogata Korin. Irises. c1705. Nezu Museum.

Contents

Changes from Version 1.1

1. Addition: Support for eigensolvers.

2. Changes: Specifications of the following functions:

   (a) Names of the functions `lis_output_residual_history()` and
       `lis_get_residual_history()` to `lis_solver_output_rhistory()` and
       `lis_solver_get_rhistory()`, respectively.

   (b) Origin of the Fortran interfaces `lis_vector_set_value()` and
       `lis_vector_get_value()` to 1.

   (c) Origin of the Fortran interface `lis_vector_set_size()` to 1.

   (d) Name of the precision flag `-precision` to `-f`.

3. Change: Specification of the function `lis_solve_kernel()` to return the residual computed by the
   function `lis_solve_execute()`.

4. Change: Specifications of the integer types:

   (a) Replacement: The type of the integer in the C programs with `LIS_INT`, which is equivalent to
       `int` by default. If the preprossor macro `_LONGLONG` is defined, it is replaced with `long long
       int`.

   (b) Replacement: The type of the integer in the Fortran programs with `LIS_INTEGER`, which is
       equivalent to `integer` by default. If the preprossor macro `LONGLONG` is defined, it is replaced
       with `integer*8`.

5. Change: Names of the matrix storage formats CRS (Compressed Row Storage) and CCS (Compressed Column Storage) to CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column),
   respectively.

6. Change: Names of the functions `lis_get_solvername()`, `lis_get_preconname()`, and
   `lis_get_esolvername()` to `lis_solver_get_solvername()`, `lis_solver_get_preconname()`, and
   `lis_esolver_get_esolvername()`, respectively.

Changes from Version 1.2

1. Change: Name of the file `lis_config_win32.h` to `lis_config_win.h`.

2. Change: Name of the matrix storage format JDS (Jagged Diagonal Storage) to JAD (Jagged
   Diagonal).

3. Change: Names of the functions `lis_fscan_double()` and `lis_bswap_double()` to
   `lis_fscan_scalar()` and `lis_bswap_scalar()`, respectively.

1 Introduction

Lis, a Library of Iterative Solvers for linear systems, is a parallel library for solving the linear equations

$$Ax = b$$

and the standard eigenvalue problems

$$Ax = \lambda x$$

with real sparse matrices, which arise in the numerical solution of partial differential equations, using iterative methods[1]. The solvers available in Lis are listed in Table 1 and 2, and the preconditioners are listed in Table 3. The supported matrix storage formats are listed in Table 4.

Table 1: Linear Solvers

CG[2, 3]	CR[2]
BiCG[4]	BiCR[5]
CGS[6]	CRS[7]
BiCGSTAB[8]	BiCRSTAB[7]
GPBiCG[9]	GPBiCR[7]
BiCGSafe[10]	BiCRSafe[11]
BiCGSTAB(l)[12]	TFQMR[13]
Jacobi[14]	Orthomin(m)[15]
Gauss-Seidel[16, 17]	GMRES(m)[18]
SOR[19, 20]	FGMRES(m)[21]
IDR(s)[22]	MINRES[23]

Table 2: Eigensolvers

Power[24]
Inverse[25]
Approximate Inverse[1]
Rayleigh Quotient[26]
Subspace[27]
Lanczos[28]
CG[29]
CR[30]

Table 3: Preconditioners

Jacobi[31]
SSOR[31]
ILU(k)[32]
ILUT[33, 34]
Crout ILU[34, 35]
I+S[36]
SA-AMG[37]
Hybrid[38]
SAINV[39]
Additive Schwarz[40, 41]
User defined

Table 4: Matrix Storage Formats

Compressed Sparse Row	(CSR)
Compressed Sparse Column	(CSC)
Modified Compressed Sparse Row	(MSR)
Diagonal	(DIA)
Ellpack-Itpack Generalized Diagonal	(ELL)
Jagged Diagonal	(JAD)
Block Sparse Row	(BSR)
Block Sparse Column	(BSC)
Variable Block Row	(VBR)
Coordinate	(COO)
Dense	(DNS)

2 Installation

This section describes the instructions for installing and testing Lis. We assume Lis being installed on a server cluster running Linux operating system.

2.1 System Requirements

The installation of Lis requires a C compiler. The Fortran interface requires a Fortran compiler, and the algebraic multigrid preconditioner requires a Fortran 90 compiler. For parallel computing environments, an OpenMP[74] or MPI-1[73] library is used[42, 43]. Both the Harwell-Boeing[70] and Matrix Market[71] formats are supported to import and export user data. Lis has been tested in the environments listed in Table 5 (see also Table 7).

Table 5: Major Tested Platforms

C Compilers	OS
Intel C/C++ Compiler 7.0, 8.0, 9.1, 10.1, 11.1, Intel C++ Composer XE	Linux Windows
IBM XL C/C++ V7.0, 9.0	AIX Linux
Sun WorkShop 6, Sun ONE Studio 7, Sun Studio 11, 12	Solaris
PGI C++ 6.0, 7.1, 10.5	Linux
gcc 3.3, 4.4	Linux Mac OS X Windows
Microsoft Visual C++ 2008, 2010, 2012	Windows
Fortran Compilers (Optional)	OS
Intel Fortran Compiler 8.1, 9.1, 10.1, 11.1, Intel Fortran Composer XE	Linux Windows
IBM XL Fortran V9.1, 11.1	AIX Linux
Sun WorkShop 6, Sun ONE Studio 7, Sun Studio 11, 12	Solaris
PGI Fortran 6.0, 7.1, 10.5	Linux
g77 3.3 gfortran 4.4	Linux Mac OS X Windows

2.2 Installing on UNIX and Compatible Systems

2.2.1 Extracting Archive

Enter the following command to extract the archive, where (`$VERSION`) represents the version:
```
> gunzip -c lis-($VERSION).tar.gz | tar xvf -
```
This creates a directory `lis-($VERSION)` together with its subfolders as shown in Figure 1.

2.2.2 Configuring Source Tree

In the directory `lis-($VERSION)`, run the following command to configure the source tree:

- default: `> ./configure`

- specify the installation destination: `> ./configure --prefix=<install-dir>`

```
lis-($VERSION)
 + config
 |   configuration files
 + doc
 |   documents
 + include
 |   header files
 + src
 |   source files
 + test
 |   test programs
 + win
     configuration files for Windows systems
```

Figure 1: Files contained in `lis-($VERSION).tar.gz`

Table 6 shows the major options that can be specified for the configuration, and Table 7 shows the major computing environments that can be specified by `TARGET`.

Table 6: Major Configuration Options (see `./configure --help` for the complete list)

`--enable-omp`	Build with OpenMP library
`--enable-mpi`	Build with MPI library
`--enable-fortran`	Build Fortran interface
`--enable-saamg`	Build SA-AMG preconditioner
`--enable-quad`	Build with double double (quadruple) precision support
`--enable-longdouble`	Build with long double (quadruple) precision support
`--enable-longlong`	Build with 64bit integer support
`--enable-debug`	Build with debug mode
`--enable-shared`	Enable dynamic linking
`--enable-gprof`	Enable profiling
`--prefix=<install-dir>`	Specify installation destination
`TARGET=<target>`	Specify computing environment
`CC=<c_compiler>`	Specify C compiler
`CFLAGS=<c_flags>`	Specify options for C compiler
`FC=<fortran90_compiler>`	Specify Fortran 90 compiler
`FCFLAGS=<fc_flags>`	Specify options for Fortran 90 compiler
`LDFLAGS=<ld_flags>`	Specify link options

2.2.3 Compiling

In the directory `lis-($VERSION)`, run the following command to generate the executable files:

> make

To ensure that the library has been built successfully, enter

> make check

in `lis-($VERSION)`. This runs a test script using the executable files created in `lis-($VERSION)/test`, which reads the data of the coefficient matrix and the right-hand side vector from the file `test/testmat.mtx` and solve the linear equation $Ax = b$ by the BiCG method. The result on the SGI Altix 3700 is shown below. Options `--enable-omp` and `--enable-mpi` can be combined.

5

Table 7: Examples of Targets (see `lis-($VERSION)/configure.in` for details)

`<target>`	Equivalent options

`cray_xt3_cross`	`./configure CC=cc FC=ftn CFLAGS="-O3 -B -fastsse -tp k8-64"`
	`  FCFLAGS="-O3 -fastsse -tp k8-64 -Mpreprocess" FCLDFLAGS="-Mnomain"`
	`  ac_cv_sizeof_void_p=8 cross_compiling=yes`
	`  ax_f77_mangling="lower case, no underscore, extra underscore"`

`fujitsu_fx10_cross`	`./configure CC=fccpx FC=frtpx CFLAGS="-Kfast,ocl,preex"`
	`  FCFLAGS="-Kfast,ocl,preex -Cpp -fs" FCLDFLAGS="-mlcmain=main"`
	`  ac_cv_sizeof_void_p=8 cross_compiling=yes`
	`  ax_f77_mangling="lower case, underscore, no extra underscore"`

`hitachi_sr16k`	`./configure CC=cc FC=f90 CFLAGS="-Os -noparallel"`
	`  FCFLAGS="-Oss -noparallel" FCLDFLAGS="-lf90s"`
	`  ac_cv_sizeof_void_p=8`
	`  ax_f77_mangling="lower case, underscore, no extra underscore"`

`ibm_bgl_cross`	`./configure CC=blrts_xlc FC=blrts_xlf90`
	`  CFLAGS="-O3 -qarch=440d -qtune=440 -qstrict"`
	`  FCFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90"`
	`  ac_cv_sizeof_void_p=4 cross_compiling=yes`
	`  ax_f77_mangling="lower case, no underscore, no extra underscore"`

`nec_sx9_cross`	`./configure CC=sxmpic++ FC=sxmpif90 AR=sxar RANLIB=true`
	`  ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes`
	`  ax_f77_mangling="lower case, no underscore, extra underscore"`

Default

```
matrix size = 100 x 100 (460 nonzero entries)
initial vector x = 0
precision : double
solver    : BiCG 2
precon    : none
storage   : CSR
lis_solve : normal end

BiCG: number of iterations   = 15 (double = 15, quad = 0)
BiCG: elapsed time           = 5.178690e-03 sec.
BiCG:    preconditioner      = 1.277685e-03 sec.
BiCG:       matrix creation  = 1.254797e-03 sec.
BiCG:    linear solver       = 3.901005e-03 sec.
BiCG: relative residual 2-norm = 6.327297e-15
```

```
╭─ --enable-omp ─────────────────────────────────────────────────╮
│                                                                 │
│  max number of threads = 32                                     │
│  number of threads = 2                                          │
│  matrix size = 100 x 100 (460 nonzero entries)                  │
│  initial vector x = 0                                           │
│  precision : double                                             │
│  solver    : BiCG 2                                             │
│  precon    : none                                               │
│  storage   : CSR                                                │
│  lis_solve : normal end                                         │
│                                                                 │
│  BiCG: number of iterations    = 15 (double = 15, quad = 0)     │
│  BiCG: elapsed time            = 8.960009e-03 sec.              │
│  BiCG:   preconditioner        = 2.297878e-03 sec.              │
│  BiCG:     matrix creation     = 2.072096e-03 sec.              │
│  BiCG:   linear solver         = 6.662130e-03 sec.              │
│  BiCG: relative residual 2-norm = 6.221213e-15                  │
│                                                                 │
╰─────────────────────────────────────────────────────────────────╯

╭─ --enable-mpi ─────────────────────────────────────────────────╮
│                                                                 │
│  number of processes = 2                                        │
│  matrix size = 100 x 100 (460 nonzero entries)                  │
│  initial vector x = 0                                           │
│  precision : double                                             │
│  solver    : BiCG 2                                             │
│  precon    : none                                               │
│  storage   : CSR                                                │
│  lis_solve : normal end                                         │
│                                                                 │
│  BiCG: number of iterations    = 15 (double = 15, quad = 0)     │
│  BiCG: elapsed time            = 2.911400e-03 sec.              │
│  BiCG:   preconditioner        = 1.560780e-04 sec.              │
│  BiCG:     matrix creation     = 1.459997e-04 sec.              │
│  BiCG:   linear solver         = 2.755322e-03 sec.              │
│  BiCG: relative residual 2-norm = 6.221213e-15                  │
│                                                                 │
╰─────────────────────────────────────────────────────────────────╯
```

2.2.4 Installing

In the directory `lis-($VERSION)`, enter

```
      > make install
```

which copies the files to the destination directory as follows:

```
($INSTALLDIR)
 +bin
 |   +lsolve esolve spmvtest[1-5]
 +include
 |   +lis_config.h lis.h lisf.h
 +lib
 |   +liblis.a
 +share
     +doc/lis examples/lis man
```

`lis_config.h` is the header file required to build the library, and `lis.h` and `lisf.h` are the header files required by the C and Fortran compilers, respectively. `liblis.a` is the library file. To ensure that the library has been installed successfully, enter

```
> make installcheck
```

in `lis-($VERSION)`. This runs a test script using the executable files installed in `examples/lis`. `test1` and `etest5` in `examples/lis` are copied in `($INSTALLDIR)/bin` as `lsolve` and `esolve`, respectively. `examples/lis/spmvtest[1-5]` are also copied in `($INSTALLDIR)/bin`.

2.3 Installing on Windows Systems

Use an appropriate tool to extract the archive `lis-($VERSION).tar.gz`. To use the Microsoft Build Engine, run the following command in the directory `lis-($VERSION)\win` and generate the configuration file `Makefile` (See `configure --help` for details):

```
> configure
```

The default configuration of `Makefile` is defined in `Makefile.in`. To build the library, run the following command in the directory `lis-($VERSION)\win`:

```
> nmake
```

To ensure that the library has been built successfully, enter

```
> nmake check
```

The following command copies the library files to `lis-($VERSION)\lib` and the executable files to `lis-($VERSION)\bin`, respectively:

```
> nmake install
```

To use UNIX compatible environments, follow the instructions in the previous section.

2.4 Testing

2.4.1 test1

`Usage: test1 matrix_filename rhs_setting solution_filename residual_filename [options]`

This program inputs the data of the coefficient matrix from `matrix_filename` and solves the linear equation $Ax = b$ with the solver specified by `options`. It outputs the solution to `solution_filename` in the extended Matrix Market format and the residual history to `residual_filename` in the PLAIN format (see Appendix). The Matrix Market and extended Matrix Market formats are supported for `matrix_filename`. One of the following values can be specified by `rhs_setting`:

0	Use the right-hand side vector b included in the data file
1	Use $b = (1, \dots, 1)^T$
2	Use $b = A \times (1, \dots, 1)^T$
`rhs_filename`	The filename for the right-hand side vector

The PLAIN and Matrix Market formats are supported for `rhs_filename`. `test1f.F` is the Fortran version of `test1.c`.

2.4.2 test2

`Usage: test2 m n matrix_type solution_filename residual_filename [options]`

This program solves the linear equation $Ax = b$, where the coefficient matrix A of size mn is derived from a discretized two dimensional Poisson's equation using the five point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the solution to `solution_filename` in the extended Matrix Market format and the residual history to `residual_filename` in the PLAIN format. The right-hand side vector is set such that the

values of the elements of the solution are 1. The values `m` and `n` represent the numbers of grid points in each dimension.

2.4.3 test3

Usage: `test3 l m n matrix_type solution_filename residual_filename [options]`

This program solves the linear equation $Ax = b$, where the coefficient matrix A of size lmn is derived from a discretized three dimensional Poisson's equation using the seven point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the solution to `solution_filename` in the extended Matrix Market format and the residual history to `residual_filename` in the PLAIN format. The right-hand side vector is set such that the values of the elements of the solution are 1. The values `l`, `m` and `n` represent the numbers of grid points in each dimension.

2.4.4 test4

This program solves the linear equation $Ax = b$ with a specified solver and a preconditioner, where A is a tridiagonal matrix

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

of size 12. The right-hand side vector b is set such that the values of the elements of the solution x are 1. `test4f.F` is the Fortran version of `test4.c`.

2.4.5 test5

Usage: `test5 n gamma [options]`

This program solves a linear equation $Ax = b$, where A is a Toeplitz matrix

$$\begin{pmatrix} 2 & 1 & & & & & \\ 0 & 2 & 1 & & & & \\ \gamma & 0 & 2 & 1 & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & & \gamma & 0 & 2 & 1 \\ & & & & \gamma & 0 & 2 \end{pmatrix}$$

of size n, with the solver specified by `options`. Note that the right-hand vector is set such that the values of the elements of the solution are 1.

2.4.6 etest1

Usage: `etest1 matrix_filename solution_filename residual_filename [options]`

This program inputs the matrix data from `matrix_filename` and solves the eigenvalue problem $Ax = \lambda x$ with the solver specified by `options`. It outputs the associated eigenvector to `solution_filename` in the extended Matrix Market format and the residual history to `residual_filename` in the PLAIN format. The Matrix Market format is supported for `matrix_filename`. `etest1f.F` is the Fortran version of `etest1.c`.

2.4.7 etest2

Usage: `etest2 m n matrix_type solution_filename residual_filename [options]`

This program solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix A of size mn is derived from a discretized two dimensional Helmholtz's equation using the five point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the associated eigenvector to `solution_filename` in the extended Matrix Market format and the residual history to `residual_filename` in the PLAIN format. The values `m` and `m` represent the numbers of grid points in each dimension.

2.4.8 etest3

Usage: `etest3 l m n matrix_type solution_filename residual_filename [options]`

This program solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix A of size lmn is derived from a discretized three dimensional Helmholtz's equation using the seven point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the associated eigenvector to `solution_filename` in the extended Matrix Market format and the residual history to `residual_filename` in the PLAIN format. The values `l`, `m` and `n` represent the numbers of grid points in each dimension.

2.4.9 etest4

Usage: `etest4 n [options]`

This program solves the eigenvalue problem $Ax = \lambda x$ with a specified solver, where A is a tridiagonal matrix

$$
A = \begin{pmatrix}
2 & -1 & & & \\
-1 & 2 & -1 & & \\
& \ddots & \ddots & \ddots & \\
& & -1 & 2 & -1 \\
& & & -1 & 2
\end{pmatrix}
$$

of size $n \times n$. `etest4f.F` is the Fortran version of `etest4.c`.

2.4.10 etest5

Usage: `etest5 matrix_filename eigenvalue_filename eigenvector_filename [options]`

This program inputs the matrix data from `matrix_filename` and solves the eigenvalue problem $Ax = \lambda x$ with the solver specified by `options`. It outputs the eigenvalues specified by `options` to `eigenvalue_filename` and the associated eigenvectors to `eigenvector_filename` in the extended Matrix Market format. The Matrix Market format is supported for `matrix_filename`.

2.4.11 etest6

Usage: `etest6 l m n matrix_type eigenvalue_filename eigenvector_filename [options]`

This program solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix A of size lmn is derived from a discretized three dimensional Helmholtz's equation using the seven point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the eigenvalues specified by `options` to `eigenvalue_filename` and the associated eigenvectors to `eigenvector_filename` in the extended Matrix Market format. The values `l`, `m` and `n` represent the numbers of grid points in each dimension.

2.4.12 spmvtest1

Usage: spmvtest1 n iter [matrix_type]

This program computes the matrix-vector multiply of a discretized one dimensional Laplacian

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

of size n using the three point central difference scheme and a vector $(1, \ldots, 1)^T$. The FLOPS performance is measured as the average of iter iterations. If necessary, one of the following values can be specified by matrix_type:

0	Measure the performance for the available matrix storage formats
1-11	The number of the matrix storage format

2.4.13 spmvtest2

Usage: spmvtest2 m n iter [matrix_type]

This program computes the matrix-vector multiply of a discretized two dimensional Laplacian of size mn using the five point central difference scheme and a vector $(1, \ldots, 1)^T$. The FLOPS performance is measured as the average of iter iterations. If necessary, one of the following values can be specified by matrix_type:

0	Measure the performance for the available matrix storage formats
1-11	The number of the matrix storage format

The values m and n represent the numbers of grid points in each dimension.

2.4.14 spmvtest3

Usage: spmvtest3 l m n iter [matrix_type]

This program computes the matrix-vector multiply of a discretized three dimensional Laplacian of size lmn using the seven point central difference scheme and a vector $(1, \ldots, 1)^T$. The values l, m and n represent the numbers of grid points in each dimension. The FLOPS performance is measured as the average of iter iterations. If necessary, one of the following values can be specified by matrix_type:

0	Measure the performance for the available matrix storage formats
1-11	The number of the matrix storage format

2.4.15 spmvtest4

Usage: spmvtest4 matrix_filename_list iter [block]

This program inputs the matrix data from the files listed in matrix_filename_list, and computes the multiplies of matrices in available matrix storage formats and a vector $(1, \ldots, 1)^T$. The FLOPS performance is measured as the average of iter iterations. The Matrix Market format is supported for matrix_filename_list. If necessary, the block size of the BSR and BSC formats can be specified by block.

2.4.16 spmvtest5

Usage: spmvtest5 matrix_filename matrix_type iter [block]

This program inputs the matrix data from `matrix_filename` and compute the multiply of the matrix with `matrix_type` and a vector $(1, \ldots, 1)^T$. The FLOPS performance is measured as the average of `iter` iterations. The Matrix Market format is supported for `matrix_filename`. If necessary, the block size of the BSR and BSC formats can be specified by `block`.

2.5 Limitations

The current version has the following limitations:

- Preconditioners

  – If a preconditioner other than the Jacobi or SSOR is selected and matrix A is not in the CSR format, a new matrix is created in the CSR format for preconditioning.

  – The SA-AMG preconditioner does not support the BiCG method.

  – The SA-AMG preconditioner does not support the multithreaded environment.

  – The assembly of the matrices in the SAINV preconditioner is not parallelized.

- Matrix storage formats

  – The VBR format does not support the multiprocessing environment.

  – The SA-AMG preconditioner supports only the CSR format.

  – In the multiprocessing environment, the CSR is the only accepted format for user defined arrays.

- Double double (quadruple) precision operations (see Section 4)

  – The Jacobi, Gauss-Seidel, SOR, and IDR(s) methods do not support the quadruple precision operations.

  – The CG and CR methods for the eigenvalue problems do not support the quadruple precision operations.

  – The Jacobi, Gauss-Seidel and SOR methods in the hybrid preconditioner do not support the quadruple precision operations.

  – The I+S and SA-AMG preconditioners do not support the quadruple precision operations.

3 Basic Operations

This section describes how to use the library. A program requires the following statements:

- Initialization

- Matrix creation

- Vector creation

- Solver creation

- Value assignment for matrices and vectors

- Solver assignment

- Solver execution

- Finalization

In addition, it must include one of the following compiler directives:

- C   #include "lis.h"

- Fortran #include "lisf.h"

When Lis is installed in ($INSTALLDIR), `lis.h` and `lisf.h` are located in ($INSTALLDIR)/include.

3.1 Initializing and Finalizing

The functions for initializing and finalizing the execution environment must be called at the top and bottom of the program, respectively, as follows:

```
C
1: #include "lis.h"
2: LIS_INT main(LIS_INT argc, char* argv[])
3: {
4:     lis_initialize(&argc, &argv);
5:     ...
6:     lis_finalize();
7: }
```

```
Fortran
1: #include "lisf.h"
2:     call lis_initialize(ierr)
3:     ...
4:     call lis_finalize(ierr)
```

Initializing

For initializing, the following functions are used:

- C   LIS_INT lis_initialize(LIS_INT* argc, char** argv[])

- Fortran subroutine lis_initialize(LIS_INTEGER ierr)

This function initializes the MPI execution environment, and specifies the options on the command line.

The default type of the integer in the C programs is `LIS_INT`, which is equivalent to `int`. If the preprossor macro `_LONGLONG` is defined, it is replaced with `long long int`. The default type of the integer in the Fortran programs is `LIS_INTEGER`, which is equivalent to `integer`. If the preprocessor macro

LONGLONG is defined, it is replaced with `integer*8`.

Finalizing

For finalizing, the following functions are used:

- C   LIS_INT lis_finalize()

- Fortran subroutine lis_finalize(LIS_INTEGER ierr)

3.2   Operating Vectors

Assume that the size of vector v is $global\_n$, and the size of each partial vector stored on $nprocs$ processing elements is $local\_n$. If $global\_n$ is divisible, then $local\_n$ is equal to $global\_n$ / $nprocs$. For example, when vector v is stored on two processing elements, as shown in Equation (3.1), $global\_n$ and $local\_n$ are 4 and 2, respectively.

$$v = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \begin{matrix} \text{PE0} \\ \\ \text{PE1} \end{matrix} \tag{3.1}$$

In the case of creating vector v in Equation (3.1), vector v itself is created for the serial and multi-threaded environments, while the partial vectors are created and stored on a given number of processing elements for the multiprocessing environment.

Programs to create vector v are as follows, where the number of processing elements for the multi-processing environment is assumed to be two:

C (for serial and multithreaded environments)

```
 1: LIS_INT        i,n;
 2: LIS_VECTOR     v;
 3: n = 4;
 4: lis_vector_create(0,&v);
 5: lis_vector_set_size(v,0,n);              /* or lis_vector_set_size(v,n,0); */
 6:
 7: for(i=0;i<n;i++)
 8: {
 9:     lis_vector_set_value(LIS_INS_VALUE,i,(double)i,v);
10:  }
```

C (for multiprocessing environment)

```
 1: LIS_INT        i,n,is,ie;              /* or LIS_INT  i,ln,is,ie; */
 2: LIS_VECTOR     v;
 3: n = 4;                                 /* ln = 2; */
 4: lis_vector_create(MPI_COMM_WORLD,&v);
 5: lis_vector_set_size(v,0,n);            /* lis_vector_set_size(v,ln,0); */
 6: lis_vector_get_range(v,&is,&ie);
 7: for(i=is;i<ie;i++)
 8: {
 9:     lis_vector_set_value(LIS_INS_VALUE,i,(double)i,v);
10:  }
```

```
  ┌─ Fortran (for serial and multithreaded environments) ──────────────────────────┐
  │                                                                                 │
  │   1: LIS_INTEGER    i,n                                                          │
  │   2: LIS_VECTOR     v                                                            │
  │   3: n = 4                                                                       │
  │   4: call lis_vector_create(0,v,ierr)                                           │
  │   5: call lis_vector_set_size(v,0,n,ierr)                                       │
  │   6:                                                                             │
  │   7: do i=1,n                                                                    │
  │   9:     call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr)              │
  │  10: enddo                                                                       │
  └─────────────────────────────────────────────────────────────────────────────────┘

  ┌─ Fortran (for multiprocessing environment) ─────────────────────────────────────┐
  │                                                                                 │
  │   1: LIS_INTEGER    i,n,is,ie                                                    │
  │   2: LIS_VECTOR     v                                                            │
  │   3: n = 4                                                                       │
  │   4: call lis_vector_create(MPI_COMM_WORLD,v,ierr)                               │
  │   5: call lis_vector_set_size(v,0,n,ierr)                                        │
  │   6: call lis_vector_get_range(v,is,ie,ierr)                                     │
  │   7: do i=is,ie-1                                                                │
  │   8:     call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr);              │
  │   9: enddo                                                                       │
  └─────────────────────────────────────────────────────────────────────────────────┘
```

Declaring Variables

As the second line shows, the declaration is stated as follows:

```
LIS_VECTOR     v;
```

Creating Vectors

To create vector v, the following functions are used:

- C   LIS_INT lis_vector_create(LIS_Comm comm, LIS_VECTOR *v)

- Fortran subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOr v, LIS_INTEGER ierr)

For the example program above, `comm` must be replaced with the MPI communicator. For the serial and multithreaded environments, the value of `comm` is ignored.

Assigning Sizes

To assign a size to vector v, the following functions are used:

- C   LIS_INT lis_vector_set_size(LIS_VECTOR v, LIS_INT local_n, LIS_INT global_n)

- Fortran subroutine lis_vector_set_size(LIS_VECTOR v, LIS_INTEGER local_n,
          LIS_INTEGER global_n, LIS_INTEGER ierr)

Either *local_n* or *global_n* must be provided.

For the serial and multithreaded environments, *local_n* is equal to *global_n*. Therefore, both `lis_vector_set_size(v,n,0)` and `lis_vector_set_size(v,0,n)` create a vector of size n.

For the multiprocessing environment, `lis_vector_set_size(v,n,0)` creates a partial vector of size n on each processing element. On the other hand, `lis_vector_set_size(v,0,n)` creates a partial vector of size m_p on processing element p. The values of m_p are determined by the library.

Assigning Values

To assign a value to the i-th element of vector v, the following functions are used:

- C   `LIS_INT lis_vector_set_value(LIS_INT flag, LIS_INT i, LIS_SCALAR value, LIS_VECTOR v)`

- Fortran subroutine `lis_vector_set_value(LIS_INT flag, LIS_INT i, LIS_SCALAR value, LIS_VECTOR v, LIS_INTEGER ierr)`

For the multiprocessing environment, the i-th row of the global vector must be specified. Either

LIS_INS_VALUE : $v[i] = value$, or

LIS_ADD_VALUE : $v[i] = v[i] + value$

must be provided for `flag`.

Duplicating Vectors

To create a vector that has the same information as the existing vector, the following functions are used:

- C   `LIS_INT lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR *vout)`

- Fortran subroutine `lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout, LIS_INTEGER ierr)`

This function does not copy the values of the vector. To copy the values as well, the following functions must be called after the above functions:

- C   `LIS_INT lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst)`

- Fortran subroutine `lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst, LIS_INTEGER ierr)`

Destroying Vectors

To destroy the vector, the following functions are used:

- C   `LIS_INT lis_vector_destroy(LIS_VECTOR v)`

- Fortran subroutine `lis_vector_destroy(LIS_VECTOR v, LIS_INTEGER ierr)`

3.3   Operating Matrices

Assume that the size of matrix A is $global\_n \times global\_n$, and that the size of each row block of matrix A stored on $nprocs$ processing elements is $local\_n \times global\_n$. If $global\_n$ is divisible, then $local\_n$ is equal to $global\_n$ / $nprocs$. For example, when the row block of matrix A is stored on two processing elements, as shown in Equation (3.2), $global\_n$ and $local\_n$ are 4 and 2, respectively.

$$A = \left(\begin{array}{cccc} 2 & 1 & & \\ 1 & 2 & 1 & \\ \hline & 1 & 2 & 1 \\ & & 1 & 2 \end{array} \right) \begin{array}{l} \text{PE0} \\ \\ \text{PE1} \end{array} \tag{3.2}$$

A matrix in a specific storage format can be created in one of the following three ways:

Method 1: Define Arrays in a Specific Storage Format with Library Functions

For creating matrix A in Equation (3.2) in the CSR format, matrix A itself is created for the serial and multithreaded environments, while partial matrices are created and stored on the given number of processing elements for the multiprocessing environment.

Programs to create matrix A in the CSR format are as follows, where the number of processing elements for the multiprocessing environment is assumed to be two:

C (for serial and multithreaded environments)

```
 1: LIS_INT       i,n;
 2: LIS_MATRIX    A;
 3: n = 4;
 4: lis_matrix_create(0,&A);
 5: lis_matrix_set_size(A,0,n);              /* or lis_matrix_set_size(A,n,0); */
 6: for(i=0;i<n;i++) {
 7:     if( i>0   ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0,A);
 8:     if( i<n-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0,A);
 9:     lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
10:  }
11:  lis_matrix_set_type(A,LIS_MATRIX_CSR);
12:  lis_matrix_assemble(A);
```

C (for multiprocessing environment)

```
 1: LIS_INT       i,n,gn,is,ie;
 2: LIS_MATRIX    A;
 3: gn = 4;                                  /* or n=2 */
 4: lis_matrix_create(MPI_COMM_WORLD,&A);
 5: lis_matrix_set_size(A,0,gn);             /* lis_matrix_set_size(A,n,0); */
 6: lis_matrix_get_size(A,&n,&gn);
 7: lis_matrix_get_range(A,&is,&ie);
 8: for(i=is;i<ie;i++) {
 9:     if( i>0    ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0,A);
10:     if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0,A);
11:     lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
12:  }
13:  lis_matrix_set_type(A,LIS_MATRIX_CSR);
14:  lis_matrix_assemble(A);
```

Fortran (for serial and multithreaded environments)

```
 1: LIS_INTEGER   i,n
 2: LIS_MATRIX    A
 3: n = 4
 4: call lis_matrix_create(0,A,ierr)
 5: call lis_matrix_set_size(A,0,n,ierr)
 6: do i=1,n
 7:     if( i>1 ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0d0,A,ierr)
 8:     if( i<n ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0d0,A,ierr)
 9:     call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
10:  enddo
11:  call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
12:  call lis_matrix_assemble(A,ierr)
```

```
  Fortran (for multiprocessing environment)
  1: LIS_INTEGER    i,n,gn,is,ie
  2: LIS_MATRIX     A
  3: gn = 4
  4: call lis_matrix_create(MPI_COMM_WORLD,A,ierr)
  5: call lis_matrix_set_size(A,0,gn,ierr)
  6: call lis_matrix_get_size(A,n,gn,ierr)
  7: call lis_matrix_get_range(A,is,ie,ierr)
  8: do i=is,ie-1
  9:    if( i>1  ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0d0,A,ierr)
 10:    if( i<gn ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0d0,A,ierr)
 11:    call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
 12:  enddo
 13:  call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
 14:  call lis_matrix_assemble(A,ierr)
```

Declaring Variables

As the second line shows, the declaration is stated as:

```
LIS_MATRIX    A;
```

Creating Matrices

To create matrix A, the following functions are used:

- C   LIS_INT lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)

- Fortran subroutine lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, LIS_INTEGER ierr)

comm must be replaced with the MPI communicator. For the serial and multithreaded environments, the value of comm is ignored.

Assigning Sizes

To assign a size to matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_size(LIS_MATRIX A, LIS_INT local_n, LIS_INT global_n)

- Fortran subroutine lis_matrix_set_size(LIS_MATRIX A, LIS_INTEGER local_n,
          LIS_INTEGER global_n, LIS_INTEGER ierr)

Either *local_n* or *global_n* must be provided.

For the serial and multithreaded environments, *local_n* is equal to *global_n*. Therefore, both
lis_matrix_set_size(A,n,0) and lis_matrix_set_size(A,0,n) create a matrix of size $n \times n$.

For the multiprocessing environment, lis_matrix_set_size(A,n,0) creates a partial matrix of size
$n \times N$ on each processing element, where N is the total sum of n. On the other hand, lis_matrix_set_size(A,0,n)
creates a partial matrix of size $m_p \times n$ on processing element p. The values of m_p are determined by the
library.

Assigning Values

To assign a value to the element at the i-th row and the j-th column of matrix A, the following
functions are used:

- C   LIS_INT lis_matrix_set_value(LIS_INT flag, LIS_INT i, LIS_INT j,
          LIS_SCALAR value, LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_value(LIS_INTEGER flag, LIS_INTEGER i,
          LIS_INTEGER j, LIS_SCALAR value, LIS_MATRIX A, LIS_INTEGER ierr)

18

For the multiprocessing environment, the i-th row and the j-th column of the global matrix must be specified. Either

LIS_INS_VALUE : $A(i,j) = value$, or

LIS_ADD_VALUE : $A(i,j) = A(i,j) + value$

must be provided for the parameter `flag`.

Assigning Storage Formats

To assign a storage format to matrix A, the following functions are used:

- C   `LIS_INT lis_matrix_set_type(LIS_MATRIX A, LIS_INT matrix_type)`

- Fortran subroutine `lis_matrix_set_type(LIS_MATRIX A, LIS_INT matrix_type,`
        `LIS_INTEGER ierr)`

where `matrix_type` is `LIS_MATRIX_CSR` when the matrix is created. The following storage formats are supported:

Storage format		matrix_type
Compressed Sparse Row	(CSR)	{LIS_MATRIX_CSR|1}
Compressed Sparse Column	(CSC)	{LIS_MATRIX_CSC|2}
Modified Compressed Sparse Row	(MSR)	{LIS_MATRIX_MSR|3}
Diagonal	(DIA)	{LIS_MATRIX_DIA|4}
Ellpack-Itpack Generalized Diagonal	(ELL)	{LIS_MATRIX_ELL|5}
Jagged Diagonal	(JAD)	{LIS_MATRIX_JAD|6}
Block Sparse Row	(BSR)	{LIS_MATRIX_BSR|7}
Block Sparse Column	(BSC)	{LIS_MATRIX_BSC|8}
Variable Block Row	(VBR)	{LIS_MATRIX_VBR|9}
Coordinate	(COO)	{LIS_MATRIX_COO|10}
Dense	(DNS)	{LIS_MATRIX_DNS|11}

Assembling Matrices

After assigning values and storage formats, the following functions must be called:

- C   `LIS_INT lis_matrix_assemble(LIS_MATRIX A)`

- Fortran subroutine `lis_matrix_assemble(LIS_MATRIX A, LIS_INTEGER ierr)`

`lis_matrix_assemble` assembles A into the storage format specified by `lis_matrix_set_type`.

Destroying Matrices

To destroy the matrix, the following functions are used:

- C   `LIS_INT lis_matrix_destroy(LIS_MATRIX A)`

- Fortran subroutine `lis_matrix_destroy(LIS_MATRIX A, LIS_INTEGER ierr)`

Method 2: Define Arrays in a Specific Storage Format Directly

For creating matrix A in Equation (3.2) in the CSR format, matrix A itself is created for the serial and multithreaded environments, while the partial matrices are created and stored on the given number of processing elements for the multiprocessing environment.

Programs to create matrix A in the CSR format are as follows, where the number of processing elements for the multiprocessing environment is assumed to be two:

```
 1: LIS_INT        i,k,n,nnz;
 2: LIS_INT        *ptr,*index;
 3: LIS_SCALAR     *value;
 4: LIS_MATRIX     A;
 5: n = 4; nnz = 10; k = 0;
 6: lis_matrix_malloc_csr(n,nnz,&ptr,&index,&value);
 7: lis_matrix_create(0,&A);
 8: lis_matrix_set_size(A,0,n);            /* or lis_matrix_set_size(A,n,0); */
 9:
10: for(i=0;i<n;i++)
11: {
12:     if( i>0   ) {index[k] = i-1; value[k] = 1; k++;}
13:     index[k] = i; value[k] = 2; k++;
14:     if( i<n-1 ) {index[k] = i+1; value[k] = 1; k++;}
15:     ptr[i+1] = k;
16:  }
17:  ptr[0] = 0;
18:  lis_matrix_set_csr(nnz,ptr,index,value,A);
19:  lis_matrix_assemble(A);
```

```
 1: LIS_INT        i,k,n,nnz,is,ie;
 2: LIS_INT        *ptr,*index;
 3: LIS_SCALAR     *value;
 4: LIS_MATRIX     A;
 5: n = 2; nnz = 5; k = 0;
 6: lis_matrix_malloc_csr(n,nnz,&ptr,&index,&value);
 7: lis_matrix_create(MPI_COMM_WORLD,&A);
 8: lis_matrix_set_size(A,n,0);
 9: lis_matrix_get_range(A,&is,&ie);
10: for(i=is;i<ie;i++)
11: {
12:     if( i>0   ) {index[k] = i-1; value[k] = 1; k++;}
13:     index[k] = i; value[k] = 2; k++;
14:     if( i<n-1 ) {index[k] = i+1; value[k] = 1; k++;}
15:     ptr[i-is+1] = k;
16:  }
17:  ptr[0] = 0;
18:  lis_matrix_set_csr(nnz,ptr,index,value,A);
19:  lis_matrix_assemble(A);
```

Associating Arrays

To associate the arrays in the CSR format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

Method 3: Read Matrix and Vector Data from External Files

Programs to read matrix A in Equation (3.2) in the CSR format and vector b in Equation (3.1) from an external file are as follows:

C (for serial, multithreaded and multiprocessing environments)

```
1: LIS_MATRIX    A;
2: LIS_VECTOR    b,x;
3: lis_matrix_create(LIS_COMM_WORLD,&A);
4: lis_vector_create(LIS_COMM_WORLD,&b);
5: lis_vector_create(LIS_COMM_WORLD,&x);
6: lis_matrix_set_type(A,LIS_MATRIX_CSR);
7: lis_input(A,b,x,"matvec.mtx");
```

Fortran (for serial, multithreaded and multiprocessing environments)

```
1: LIS_MATRIX    A
2: LIS_VECTOR    b,x
3: call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
4: call lis_vector_create(LIS_COMM_WORLD,b,ierr)
5: call lis_vector_create(LIS_COMM_WORLD,x,ierr)
6: call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
7: call lis_input(A,b,x,'matvec.mtx',ierr)
```

The content of the destination file `matvec.mtx` is:

```
%%MatrixMarket matrix coordinate real general
4 4 10 1 0
1 2  1.0e+00
1 1  2.0e+00
2 3  1.0e+00
2 1  1.0e+00
2 2  2.0e+00
3 4  1.0e+00
3 2  1.0e+00
3 3  2.0e+00
4 4  2.0e+00
4 3  1.0e+00
1  0.0e+00
2  1.0e+00
3  2.0e+00
4  3.0e+00
```

Reading from External Files

To input the matrix data for A from an external file, the following functions are used:

- C   `LIS_INT lis_input_matrix(LIS_MATRIX A, char *filename)`

- Fortran subroutine `lis_input(LIS_MATRIX A,`
  `character filename, LIS_INTEGER ierr)`

`filename` must be replaced with the file path. The following file formats are supported:

- The Matrix Market format

- The Harwell-Boeing format

To read the data for matrix A and vectors b and x from external files, the following functions are used:

- C   `LIS_INT lis_input(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, char *filename)`

- Fortran subroutine `lis_input(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,` `character filename, LIS_INTEGER ierr)`

`filename` must be replaced with the file path. The following file formats are supported:

- The Extended Matrix Market format (extended to allow vector data)

- The Harwell-Boeing format

3.4   Solving Linear Equations

A program to solve the linear equation $Ax = b$ with a specified solver is as follows:

C (for serial, multithreaded and multiprocessing environments)

```
 1: LIS_MATRIX A;
 2: LIS_VECTOR b,x;
 3: LIS_SOLVER solver;
 4:
 5: /* Create matrix and vector */
 6:
 7: lis_solver_create(&solver);
 8: lis_solver_set_option("-i bicg -p none",solver);
 9: lis_solver_set_option("-tol 1.0e-12",solver);
10: lis_solve(A,b,x,solver);
```

Fortran (for serial, multithreaded and multiprocessing environments)

```
 1: LIS_MATRIX A
 2: LIS_VECTOR b,x
 3: LIS_SOLVER solver
 4:
 5: /* Create matrix and vector */
 6:
 7: call lis_solver_create(solver,ierr)
 8: call lis_solver_set_option('-i bicg -p none',solver,ierr)
 9: call lis_solver_set_option('-tol 1.0e-12',solver,ierr)
10: call lis_solve(A,b,x,solver,ierr)
```

Creating Solvers

    To create a solver, the following functions are used:

- C   `LIS_INT lis_solver_create(LIS_SOLVER *solver)`

- Fortran subroutine `lis_solver_create(LIS_SOLVER solver, LIS_INTEGER ierr)`

Specifying Options

    To specify options, the following functions are used:

- C   `LIS_INT lis_solver_set_option(char *text, LIS_SOLVER solver)`

- Fortran subroutine `lis_solver_set_option(character text, LIS_SOLVER solver,` `LIS_INTEGER ierr)`

or

- C   LIS_INT lis_solver_set_optionC(LIS_SOLVER solver)

- Fortran subroutine lis_solver_set_optionC(LIS_SOLVER solver, LIS_INTEGER ierr)

lis_solver_set_optionC is a function that sets the options specified on the command line, and passes them to solver when the program is run.

The table below shows the available command line options, where `-i {cg|1}` means `-i cg` or `-i 1` and `-maxiter [1000]` indicates that `-maxiter` defaults to $1,000$.

Options for Linear Solvers (Default: `-i bicg`)

Solver	Option	Auxiliary Options	
CG	-i {cg|1}		
BiCG	-i {bicg|2}		
CGS	-i {cgs|3}		
BiCGSTAB	-i {bicgstab|4}		
BiCGSTAB(l)	-i {bicgstabl|5}	-ell [2]	The degree l
GPBiCG	-i {gpbicg|6}		
TFQMR	-i {tfqmr|7}		
Orthomin(m)	-i {orthomin|8}	-restart [40]	The restart value m
GMRES(m)	-i {gmres|9}	-restart [40]	The restart value m
Jacobi	-i {jacobi|10}		
Gauss-Seidel	-i {gs|11}		
SOR	-i {sor|12}	-omega [1.9]	The relaxation coefficient ω $(0 < \omega < 2)$
BiCGSafe	-i {bicgsafe|13}		
CR	-i {cr|14}		
BiCR	-i {bicr|15}		
CRS	-i {crs|16}		
BiCRSTAB	-i {bicrstab|17}		
GPBiCR	-i {gpbicr|18}		
BiCRSafe	-i {bicrsafe|19}		
FGMRES(m)	-i {fgmres|20}	-restart [40]	The restart value m
IDR(s)	-i {idrs|21}	-irestart [2]	The restart value s
MINRES	-i {minres|22}		

Options for Preconditioners (Default: `-p none`)

Preconditioner	Option	Auxiliary Options	
None	`-p {none|0}`		
Jacobi	`-p {jacobi|1}`		
ILU(k)	`-p {ilu|2}`	`-ilu_fill [0]`	The fill level k
SSOR	`-p {ssor|3}`	`-ssor_w [1.0]`	The relaxation coefficient ω ($0 < \omega < 2$)
Hybrid	`-p {hybrid|4}`	`-hybrid_i [sor]`	The linear solver
		`-hybrid_maxiter [25]`	The maximum number of iterations
		`-hybrid_tol [1.0e-3]`	The convergence criterion
		`-hybrid_w [1.5]`	The relaxation coefficient ω of the SOR ($0 < \omega < 2$)
		`-hybrid_ell [2]`	The degree l of the BiCGSTAB(l)
		`-hybrid_restart [40]`	The restart values of the GMRES and Orthomin
I+S	`-p {is|5}`	`-is_alpha [1.0]`	The parameter α of $I + \alpha S^{(m)}$
		`-is_m [3]`	The parameter m of $I + \alpha S^{(m)}$
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Select the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \leq \theta^2 |a_{ii}||a_{jj}|$ (symmetric or unsymmetric)
Crout ILU	`-p {iluc|8}`	`-iluc_drop [0.05]`	The drop criterion
		`-iluc_rate [5.0]`	The ratio of the maximum fill-in
ILUT	`-p {ilut|9}`	`-ilut_drop [0.05]`	The drop criterion
		`-ilut_rate [5.0]`	The ratio of the maximum fill-in
Additive Schwarz	`-adds true`	`-adds_iter [1]`	The number of iterations

Other Options

Option	
`-maxiter [1000]`	The maximum number of iterations
`-tol [1.0e-12]`	The convergence criterion
`-print [0]`	The display of the residual
	`-print {none|0}` None
	`-print {mem|1}` Save the residual history
	`-print {out|2}` Display the residual history
	`-print {all|3}` Save the residual history and display it on the screen
`-scale [0]`	The scaling
	(The result will overwrite the original matrix and vectors)
	`-scale {none|0}` No scaling
	`-scale {jacobi|1}` The Jacobi scaling $D^{-1}Ax = D^{-1}b$
	(D represents the diagonal of $A = (a_{ij})$)
	`-scale {symm_diag|2}` The diagonal scaling $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$
	($D^{-1/2}$ represents the diagonal matrix with $1/\sqrt{a_{ii}}$
	as the diagonal)
`-initx_zeros [true]`	The behavior of the initial vector x_0
	`-initx_zeros {false|0}` Given values
	`-initx_zeros {true|1}` All values are set to 0
`-omp_num_threads [t]`	The number of threads
	(`t` represents the maximum number of threads)
`-storage [0]`	The matrix storage format
`-storage_block [2]`	The block size of the BSR and BSC formats
`-f [0]`	The precision of the linear solver
	`-f {double|0}` Double precision
	`-f {quad|1}` Quadruple precision

Solving Linear Equations

To solve the linear equation $Ax = b$, the following functions are used:

- C   `LIS_INT lis_solve(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, LIS_SOLVER solver)`

- Fortran subroutine `lis_solve(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,`   `LIS_SOLVER solver, LIS_INTEGER ierr)`

3.5   Solving Eigenvalue Problems

A program to solve the eigenvalue problem $Ax = \lambda x$ with a specified solver is as follows:

C (for serial, multithreaded and multiprocessing environments)

```
 1: LIS_MATRIX A;
 2: LIS_VECTOR x;
 3: LIS_REAL evalue;
 4: LIS_ESOLVER esolver;
 5:
 6: /* Create matrix and vector */
 7:
 8: lis_esolver_create(&esolver);
 9: lis_esolver_set_option("-e ii -i bicg -p none",esolver);
10: lis_esolver_set_option("-etol 1.0e-12 -tol 1.0e-12",esolver);
11: lis_esolve(A,x,evalue,esolver);
```

```
   ┌─ Fortran (for serial, multithreaded and multiprocessing environments) ──────────┐
   │                                                                                 │
   │   1: LIS_MATRIX A                                                               │
   │   2: LIS_VECTOR x                                                               │
   │   3: LIS_REAL evalue                                                            │
   │   4: LIS_ESOLVER esolver                                                        │
   │   5:                                                                            │
   │   6: /* Create matrix and vector */                                            │
   │   7:                                                                            │
   │   8: call lis_esolver_create(esolver,ierr)                                      │
   │   9: call lis_esolver_set_option('-e ii -i bicg -p none',esolver,ierr)         │
   │  10: call lis_esolver_set_option('-etol 1.0e-12 -tol 1.0e-12',esolver,ierr)    │
   │  11: call lis_esolve(A,x,evalue,esolver,ierr)                                   │
   │                                                                                 │
   └─────────────────────────────────────────────────────────────────────────────┘
```

Creating Eigensolvers

To create an eigensolver, the following functions are used:

- C   `LIS_INT lis_esolver_create(LIS_ESOLVER *esolver)`

- Fortran `subroutine lis_esolver_create(LIS_ESOLVER esolver, LIS_INTEGER ierr)`

Specifying Options

To specify options, the following functions are used:

- C   `LIS_INT lis_esolver_set_option(char *text, LIS_ESOLVER esolver)`

- Fortran `subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver, LIS_INTEGER ierr)`

or

- C   `LIS_INT lis_esolver_set_optionC(LIS_ESOLVER esolver)`

- Fortran `subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, LIS_INTEGER ierr)`

`lis_esolver_set_optionC` is a function that sets the options specified in the command line, and passes them to `esolver` when the program is run.

The table below shows the available command line options, where `-e {pi|1}` means `-e pi` or `-e 1` and `-emaxiter [1000]` indicates that `-emaxiter` defaults to $1,000$.

Options for Eigensolvers (Default: `-e pi`)

Eigensolver	Option	Auxiliary Options	
Power	`-e {pi|1}`		
Inverse	`-e {ii|2}`	`-i [cg]`	The linear solver
Approximate Inverse	`-e {aii|3}`	`-i [cg]`	The linear solver
Rayleigh Quotient	`-e {rqi|4}`	`-i [cg]`	The linear solver
Subspace	`-e {si|5}`	`-ss [2]`	The size of the subspace
		`-m [0]`	The mode number
Lanczos	`-e {li|6}`	`-ss [2]`	The size of the subspace
		`-m [0]`	The mode number
CG	`-e {cg|7}`		
CR	`-e {cr|8}`		

Options for Preconditioners (Default: `-p none`)

Preconditioner	Option	Auxiliary Options	
None	`-p {none|0}`		
Jacobi	`-p {jacobi|1}`		
ILU(k)	`-p {ilu|2}`	`-ilu_fill [0]`	The fill level k
SSOR	`-p {ssor|3}`	`-ssor_w [1.0]`	The relaxation coefficient ω ($0 < \omega < 2$)
Hybrid	`-p {hybrid|4}`	`-hybrid_i [sor]`	The linear solver
		`-hybrid_maxiter [25]`	The maximum number of iterations
		`-hybrid_tol [1.0e-3]`	The convergence criterion
		`-hybrid_w [1.5]`	The relaxation coefficient ω of the SOR ($0 < \omega < 2$)
		`-hybrid_ell [2]`	The degree l of the BiCGSTAB(l)
		`-hybrid_restart [40]`	The restart values of the GMRES and Orthomin
I+S	`-p {is|5}`	`-is_alpha [1.0]`	The parameter α of $I + \alpha S^{(m)}$
		`-is_m [3]`	The parameter m of $I + \alpha S^{(m)}$
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Select the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \leq \theta^2 |a_{ii}||a_{jj}|$ (symmetric or unsymmetric)
Crout ILU	`-p {iluc|8}`	`-iluc_drop [0.05]`	The drop criterion
		`-iluc_rate [5.0]`	The ratio of the maximum fill-in
ILUT	`-p {ilut|9}`	`-ilut_drop [0.05]`	The drop criterion
		`-ilut_rate [5.0]`	The ratio of the maximum fill-in
Additive Schwarz	`-adds true`	`-adds_iter [1]`	The number of iterations

Other Options

Option	
`-emaxiter [1000]`	The maximum number of iterations
`-etol [1.0e-12]`	The convergence criterion
`-eprint [0]`	The display of the residual
	`-eprint {none|0}` None
	`-eprint {mem|1}` Save the residual history
	`-eprint {out|2}` Display the residual history
	`-eprint {all|3}` Save the residual history and display it on the screen
`-ie [ii]`	The inner eigensolver used in the Lanczos and Subspace
	`-ie {pi|1}` The Power (the Subspace only)
	`-ie {ii|2}` The Inverse
	`-ie {aii|3}` The Approximate Inverse
	`-ie {rqi|4}` The Rayleigh Quotient
`-shift [0.0]`	The amount of the shift
`-initx_ones [true]`	The behavior of the initial vector x_0
	`-initx_ones {false|0}` Given values
	`-initx_ones {true|1}` All values are set to 1
`-omp_num_threads [t]`	The number of threads (`t` represents the maximum number of threads)
`-estorage [0]`	The matrix storage format
`-estorage_block [2]`	The block size of the BSR and BSC formats
`-ef [0]`	The precision of the eigensolver
	`-ef {double|0}` Double precision
	`-ef {quad|1}` Quadruple precision

Solving Eigenvalue Problems

To solve the eigenvalue problem $Ax = \lambda x$, the following functions are used:

- C   `LIS_INT lis_esolve(LIS_MATRIX A, LIS_VECTOR x,`
  `LIS_REAL evalue, LIS_ESOLVER esolver)`

- `Fortran subroutine lis_esolve(LIS_MATRIX A, LIS_VECTOR x,`
  `LIS_REAL evalue, LIS_ESOLVER esolver, LIS_INTEGER ierr)`

3.6   Writing Programs

The following are the programs for solving the linear equation $Ax = b$, where matrix A is a tridiagonal matrix

$$
\begin{pmatrix}
2 & -1 & & & & \\
-1 & 2 & -1 & & & \\
& \ddots & \ddots & \ddots & & \\
& & -1 & 2 & -1 \\
& & & -1 & 2
\end{pmatrix}
$$

of size 12. The the right-hand side vector b is set such that the values of the elements of the solution x are 1. The program is located in the directory `lis-($VERSION)/test`.

```
  ┌─ Test program: test4.c ──────────────────────────────────────────
  │
  │    1: #include <stdio.h>
  │    2: #include "lis.h"
  │    3: main(LIS_INT argc, char *argv[])
  │    4: {
  │    5:     LIS_INT i,n,gn,is,ie,iter;
  │    6:     LIS_MATRIX A;
  │    7:     LIS_VECTOR b,x,u;
  │    8:     LIS_SOLVER solver;
  │    9:     n = 12;
  │   10:     lis_initialize(&argc,&argv);
  │   11:     lis_matrix_create(LIS_COMM_WORLD,&A);
  │   12:     lis_matrix_set_size(A,0,n);
  │   13:     lis_matrix_get_size(A,&n,&gn)
  │   14:     lis_matrix_get_range(A,&is,&ie)
  │   15:     for(i=is;i<ie;i++)
  │   16:     {
  │   17:         if( i>0 ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0,A);
  │   18:         if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0,A);
  │   19:         lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
  │   20:     }
  │   21:     lis_matrix_set_type(A,LIS_MATRIX_CSR);
  │   22:     lis_matrix_assemble(A);
  │   23:
  │   24:     lis_vector_duplicate(A,&u);
  │   25:     lis_vector_duplicate(A,&b);
  │   26:     lis_vector_duplicate(A,&x);
  │   27:     lis_vector_set_all(1.0,u);
  │   28:     lis_matvec(A,u,b);
  │   29:
  │   30:     lis_solver_create(&solver);
  │   31:     lis_solver_set_optionC(solver);
  │   32:     lis_solve(A,b,x,solver);
  │   33:     lis_solver_get_iters(solver,&iter);
  │   34:     printf("iter = %d\n",iter);
  │   35:     lis_vector_print(x);
  │   36:     lis_matrix_destroy(A);
  │   37:     lis_vector_destroy(u);
  │   38:     lis_vector_destroy(b);
  │   39:     lis_vector_destroy(x);
  │   40:     lis_solver_destroy(solver);
  │   41:     lis_finalize();
  │   42:     return 0;
  │   43: }
  │ }
  └──────────────────────────────────────────────────────────────────
```

```
  Test program: test4f.F

 1:        implicit none
 2:
 3:#include "lisf.h"
 4:
 5:        LIS_INTEGER      i,n,gn,is,ie,iter,ierr
 6:        LIS_MATRIX       A
 7:        LIS_VECTOR       b,x,u
 8:        LIS_SOLVER       solver
 9:        n  = 12
10:        call lis_initialize(ierr)
11:        call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
12:        call lis_matrix_set_size(A,0,n,ierr)
13:        call lis_matrix_get_size(A,n,gn,ierr)
14:        call lis_matrix_get_range(A,is,ie,ierr)
15:        do i=is,ie-1
16:          if( i>1  ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0d0,
17:        .                                     A,ierr)
18:          if( i<gn ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0d0,
19:        .                                     A,ierr)
20:          call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
21:        enddo
22:        call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
23:        call lis_matrix_assemble(A,ierr)
24:
25:        call lis_vector_duplicate(A,u,ierr)
26:        call lis_vector_duplicate(A,b,ierr)
27:        call lis_vector_duplicate(A,x,ierr)
28:        call lis_vector_set_all(1.0d0,u,ierr)
29:        call lis_matvec(A,u,b,ierr)
30:
31:        call lis_solver_create(solver,ierr)
32:        call lis_solver_set_optionC(solver,ierr)
33:        call lis_solve(A,b,x,solver,ierr)
34:        call lis_solver_get_iters(solver,iter,ierr)
35:        write(*,*) 'iter = ',iter
36:        call lis_vector_print(x,ierr)
37:        call lis_matrix_destroy(A,ierr)
38:        call lis_vector_destroy(b,ierr)
39:        call lis_vector_destroy(x,ierr)
40:        call lis_vector_destroy(u,ierr)
41:        call lis_solver_destroy(solver,ierr)
42:        call lis_finalize(ierr)
43:
44:        stop
45:        end
```

3.7   Compiling and Linking

Provided below is an example `test4.c` located in the directory `lis-($VERSION)/test`, compiled on the SGI Altix 3700 using the Intel C/C++ Compiler (icc). Since the library includes some Fortran 90 codes when the SA-AMG preconditioner is selected, a Fortran 90 compiler must be used for the linking. The preprocessor macro `USE_MPI` must be defined for the multiprocessing environment. The preprocessor macros `_LONGLONG` for C and `LONGLONG` for Fortran must be defined when using the 64bit integer.

For the serial environment

Compiling
```
> icc -c -I($INSTALLDIR)/include test4.c
```
Linking
```
> icc -o test4 test4.o -llis
```
Linking (with SA-AMG)
```
> ifort -nofor_main -o test4 test4.o -llis
```

For multithreaded environment

Compiling
```
> icc -c -openmp -I($INSTALLDIR)/include test4.c
```
Linking
```
> icc -openmp -o test4 test4.o -llis
```
Linking (with SA-AMG)
```
> ifort -nofor_main -openmp -o test4 test4.o -llis
```

For multiprocessing environment

Compiling
```
> icc -c -DUSE_MPI -I($INSTALLDIR)/include test4.c
```
Linking
```
> icc -o test4 test4.o -llis -lmpi
```
Linking (with SA-AMG)
```
> ifort -nofor_main -o test4 test4.o -llis -lmpi
```

For multithreaded and multiprocessing environments

Compiling
```
> icc -c -openmp -DUSE_MPI -I($INSTALLDIR)/include test4.c
```
Linking
```
> icc -openmp -o test4 test4.o -llis -lmpi
```
Linking (with SA-AMG)
```
> ifort -nofor_main -openmp -o test4 test4.o -llis -lmpi
```

Provided below is an example `test4f.F` located in the directory `lis-($VERSION)/test`, compiled on the SGI Altix 3700 using the Intel Fortran Compiler (ifort). Since compiler directives are used in the program, compiler option `-fpp` is specified to use the preprocessor.

For serial environment

Compiling
```
> ifort -c -fpp -I($INSTALLDIR)/include test4f.F
```
Linking
```
> ifort -o test4 test4.o -llis
```

For multithreaded environment

Compiling
```
> ifort -c -fpp -openmp -I($INSTALLDIR)/include test4f.F
```
Linking
```
> ifort -openmp -o test4 test4.o -llis
```

⌐ For multiprocessing environment ──

Compiling
```
> ifort -c -fpp -DUSE_MPI -I($INSTALLDIR)/include test4f.F
```
Linking
```
> ifort -o test4 test4.o -llis -lmpi
```
──

⌐ For multithreaded and multiprocessing environments ─────────────────────────

Compiling
```
> ifort -c -fpp -openmp -DUSE_MPI -I($INSTALLDIR)/include test4f.F
```
Linking
```
> ifort -openmp -o test4 test4.o -llis -lmpi
```
──

3.8 Running

The test programs `test4` and `test4f` in the directory `lis-($VERSION)/test` are run as follows:

For serial environment
```
> ./test4 -i bicgstab
```
For multithreaded environment
```
> env OMP_NUM_THREADS=2 ./test4 -i bicgstab
```
For multiprocessing environment
```
> mpirun -np 2 ./test4 -i bicgstab
```
For multithreaded and multiprocessing environment
```
> mpirun -np 2 env OMP_NUM_THREADS=2 ./test4 -i bicgstab
```

The following results will be returned:

```
precision : double
solver    : BiCGSTAB 4
precon    : none
storage   : CSR
lis_solve : normal end

iter = 6
    0 1.000000e+000
    1 1.000000e+000
    2 1.000000e+000
    3 1.000000e+000
    4 1.000000e+000
    5 1.000000e+000
    6 1.000000e+000
    7 1.000000e+000
    8 1.000000e+000
    9 1.000000e+000
   10 1.000000e+000
   11 1.000000e+000
```

4 Quadruple Precision Operations

Double precision operations sometimes require a large number of iterations because of the rounding error. Besides long double precision operations, Lis supports "double double" precision operations, or quadruple precision operations by combining two double precision floating point numbers[44, 45]. To use the double double precision with the same interface as the double precision operations, both the matrix and vectors are assumed to be double precision. Lis also supports the performance acceleration of the double double precision operations with the SIMD instructions, such as Intel's Streaming SIMD Extensions (SSE)[50].

4.1 Using Quadruple Precision Operations

The test program `test5.c` solves a linear equation $Ax = b$, where A is a Toeplitz matrix

$$\begin{pmatrix} 2 & 1 & & & & & \\ 0 & 2 & 1 & & & & \\ \gamma & 0 & 2 & 1 & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & \gamma & 0 & 2 & 1 \\ & & & \gamma & 0 & 2 \end{pmatrix}.$$

The right-hand vector is set such that the values of the elements of the solution are 1. The value n is the size of matrix A. test5 with option `-f` is run:

Double precision
By entering   > ./test5 200 2.0 -f double
the following results will be returned:

```
n = 200, gamma = 2.000000
initial vector x = 0
precision : double
solver     : BiCG 2
precon     : none
storage    : CSR
lis_solve : LIS_MAXITER(code=4)

BiCG: number of iterations     = 1001 (double = 1001, quad = 0)
BiCG: elapsed time             = 2.044368e-02 sec.
BiCG:    preconditioner        = 4.768372e-06 sec.
BiCG:      matrix creation     = 4.768372e-06 sec.
BiCG:    linear solver         = 2.043891e-02 sec.
BiCG: relative residual 2-norm = 8.917591e+01
```

Quadruple precision
By entering   > ./test5 200 2.0 -f quad
the following results will be returned:

```
n = 200, gamma = 2.000000
initial vector x = 0
precision : quad
solver     : BiCG 2
precon     : none
storage    : CSR
lis_solve : normal end
```

```
BiCG: number of iterations    = 230 (double = 230, quad = 0)
BiCG: elapsed time            = 2.267408e-02 sec.
BiCG:   preconditioner        = 4.549026e-04 sec.
BiCG:      matrix creation    = 5.006790e-06 sec.
BiCG:   linear solver         = 2.221918e-02 sec.
BiCG: relative residual 2-norm = 6.499145e-11
```

5 Matrix Storage Formats

This section describes the matrix storage formats supported by the library. Assume that the matrix row (column) number begins with 0 and that the number of nonzero elements of matrix A of size $n \times n$ is nnz.

5.1 Compressed Sparse Row (CSR)

The CSR format uses three arrays `ptr`, `index` and `value` to store data.

- `value` is a double precision array of length nnz, which stores the nonzero elements of matrix A along the row.

- `index` is an integer array of length nnz, which stores the column numbers of the nonzero elements stored in the array `value`.

- `ptr` is an integer array of length $n + 1$, which stores the starting points of the rows of the arrays `value` and `index`.

5.1.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 2 shows how matrix A in Figure 2 is stored in the CSR format. A program to create the matrix in the CSR format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

| 0 | 1 | 3 | 5 | 8 | | | | A.ptr |

| 0 | 0 | 1 | 1 | 2 | 0 | 2 | 3 | A.index |
| 11 | 21 | 22 | 32 | 33 | 41 | 43 | 44 | A.value |

Figure 2: Data structure of CSR format (for serial and multithreaded environments).

For serial and multithreaded environments

```
 1: LIS_INT      n,nnz;
 2: LIS_INT      *ptr,*index;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; nnz = 8;
 6: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
 7: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: ptr[0] = 0; ptr[1] = 1; ptr[2] = 3; ptr[3] = 5; ptr[4] = 8;
13: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  1;
14: index[4] =  2; index[5] =  0; index[6] =  2; index[7] =  3;
15: value[0] = 11; value[1] = 21; value[2] = 22; value[3] = 32;
16: value[4] = 33; value[5] = 41; value[6] = 43; value[7] = 44;
17:
18:  lis_matrix_set_csr(nnz,ptr,index,value,A);
19:  lis_matrix_assemble(A);
```

5.1.2 Creating Matrices (for Multiprocessing Environment)

Figure 3 shows how matrix A in Figure 2 is stored in the CSR format on two processing elements. A program to create the matrix in the CSR format on two processing elements is as follows:

Figure 3: Data structure of CSR format (for multiprocessing environment).

```
  For multiprocessing environment
  1: LIS_INT       i,k,n,nnz,my_rank;
  2: LIS_INT       *ptr,*index;
  3: LIS_SCALAR    *value;
  4: LIS_MATRIX    A;
  5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  6: if( my_rank==0 ) {n = 2; nnz = 3;}
  7: else            {n = 2; nnz = 5;}
  8: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
  9: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 10: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 11: lis_matrix_create(MPI_COMM_WORLD,&A);
 12: lis_matrix_set_size(A,n,0);
 13: if( my_rank==0 ) {
 14:     ptr[0] = 0; ptr[1] = 1; ptr[2] = 3;
 15:     index[0] =  0; index[1] =  0; index[2] =  1;
 16:     value[0] = 11; value[1] = 21; value[2] = 22;}
 17: else {
 18:     ptr[0] = 0; ptr[1] = 2; ptr[2] = 5;
 19:     index[0] =  1; index[1] =  2; index[2] =  0; index[3] =  2; index[4] =  3;
 20:     value[0] = 32; value[1] = 33; value[2] = 41; value[3] = 43; value[4] = 44;}
 21:  lis_matrix_set_csr(nnz,ptr,index,value,A);
 22:  lis_matrix_assemble(A);
```

5.1.3 Associating Arrays

To associate the arrays in the CSR format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

36

5.2 Compressed Sparse Column (CSC)

The CSS format uses three arrays `ptr`, `index` and `value` to store data.

- `value` is a double precision array of length nnz, which stores the nonzero elements of matrix A along the column.

- `index` is an integer array of length nnz, which stores the row numbers of the nonzero elements stored in the array `value`.

- `ptr` is an integer array of length $n + 1$, which stores the starting points of the rows of the arrays `value` and `index`.

5.2.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 4 shows how matrix A in Figure 4 is stored in the CSC format. A program to create the matrix in the CSC format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

| 0 | 3 | 5 | 7 | 8 | A.ptr

| 0 | 1 | 3 | 1 | 2 | 2 | 3 | 3 | A.index

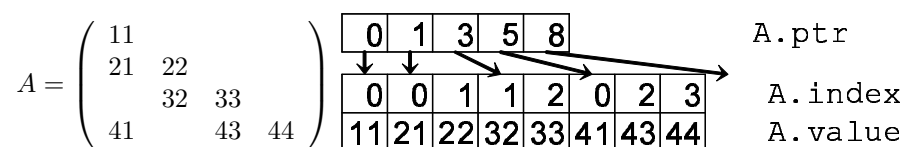| 11 | 21 | 41 | 22 | 32 | 33 | 43 | 44 | A.value

Figure 4: Data structure of CSC format (for serial and multithreaded environments).

For serial and multithreaded environments

```
 1: LIS_INT      n,nnz;
 2: LIS_INT      *ptr,*index;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; nnz = 8;
 6: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
 7: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: ptr[0] = 0; ptr[1] = 3; ptr[2] = 5; ptr[3] = 7; ptr[4] = 8;
13: index[0] =  0; index[1] =  1; index[2] =  3; index[3] =  1;
14: index[4] =  2; index[5] =  2; index[6] =  3; index[7] =  3;
15: value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22;
16: value[4] = 32; value[5] = 33; value[6] = 43; value[7] = 44;
17:
18:  lis_matrix_set_csc(nnz,ptr,index,value,A);
19:  lis_matrix_assemble(A);
```

5.2.2 Creating Matrices (for Multiprocessing Environment)

Figure 5 shows how matrix A in Figure 4 is stored on two processing elements. A program to create the matrix in the CSC format on two processing elements is as follows:
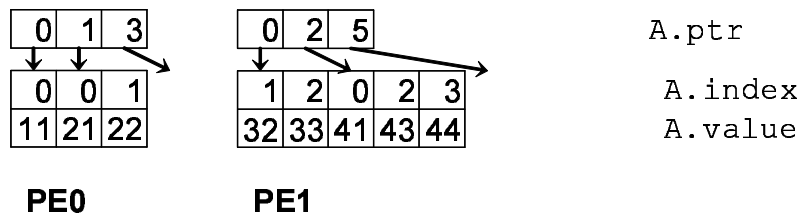

Figure 5: Data structure of CSC format (for multiprocessing environment).

```
 ┌─ For multiprocessing environment ─────────────────────────────
 1: LIS_INT       i,k,n,nnz,my_rank;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3;}
 7: else            {n = 2; nnz = 5;}
 8: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
 9: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
10: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     ptr[0] = 0; ptr[1] = 3; ptr[2] = 5;
15:     index[0] =  0; index[1] =  1; index[2] =  3; index[3] =  1; index[4] =  2;
16:     value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22; value[4] = 32}
17: else {
18:     ptr[0] = 0; ptr[1] = 2; ptr[2] = 3;
19:     index[0] =  2; index[1] =  3; index[2] =  3;
20:     value[0] = 33; value[1] = 43; value[2] = 44;}
21:  lis_matrix_set_csc(nnz,ptr,index,value,A);
22:  lis_matrix_assemble(A);
```

5.2.3 Associating Arrays

To associate the arrays in the CSC format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_csc(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_csc(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

5.3 Modified Compressed Sparse Row (MSR)

The MSR format uses two arrays `index` and `value` to store data. Assume that ndz represents the number of zero elements of the diagonal.

- `value` is a double precision array of length $nnz + ndz + 1$, which stores the diagonal of matrix A down to the n-th element. The $n + 1$-th element is not used. For the $n + 2$-th and after, the values of the nonzero elements except the diagonal of matrix A are stored along the row.

- `index` is an integer array of length $nnz + ndz + 1$, which stores the starting points of the rows of the off-diagonal elements of matrix A down to the $n + 1$-th element. For the $n + 2$-th and after, it stores the row numbers of the off-diagonal elements of matrix A stored in the array `value`.

5.3.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 6 shows how matrix A is stored in the MSR format. A program to create the matrix in the MSR format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

| 5 | 5 | 6 | 7 | 9 | 0 | 1 | 0 | 2 | A.index
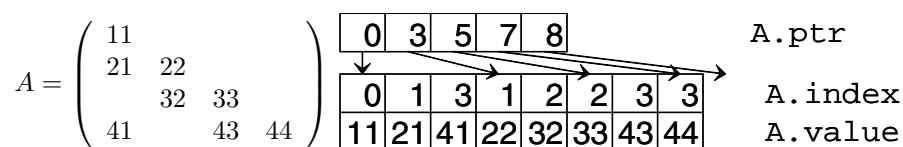
| 11 | 22 | 33 | 44 | | 21 | 32 | 41 | 43 | A.value

Figure 6: Data structure of MSR format (for serial and multithreaded environments).

```
For serial and multithreaded environments
 1: LIS_INT      n,nnz,ndz;
 2: LIS_INT      *index;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; nnz = 8; ndz = 0;
 6: index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(LIS_INT) );
 7: value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] =  5; index[1] =  5; index[2] =  6; index[3] =  7;
12: index[4] =  9; index[5] =  0; index[6] =  1; index[7] =  0; index[8] =  2;
13: value[0] = 11; value[1] = 22; value[2] = 33; value[3] = 44;
14: value[4] =  0; value[5] = 21; value[6] = 32; value[7] = 41; value[8] = 43;
15:
16:  lis_matrix_set_msr(nnz,ndz,index,value,A);
17:  lis_matrix_assemble(A);
```

5.3.2 Creating Matrices (for Multiprocessing Environment)

Figure 7 shows how matrix A in Figure 6 is stored in the MSR format on two processing elements. A program to create the matrix in the MSR format on two processing element is as follows:
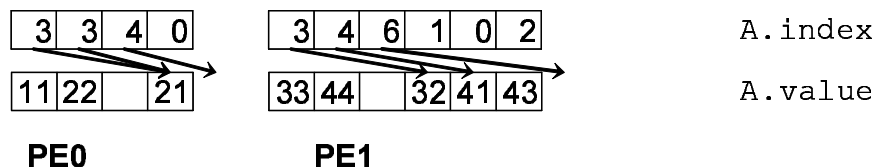
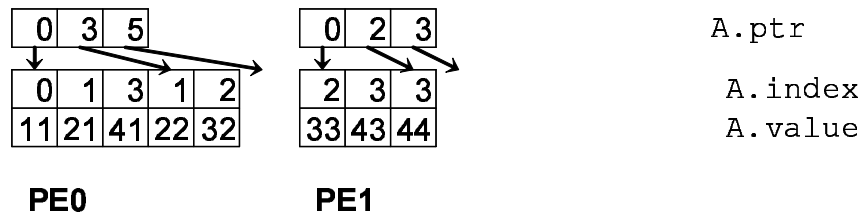Figure 7: Data structure of MSR format (for multiprocessing environment).

```
    ┌─ For multiprocessing environment ──────────────────────────────────────
     1: LIS_INT       i,k,n,nnz,ndz,my_rank;
     2: LIS_INT       *index;
     3: LIS_SCALAR    *value;
     4: LIS_MATRIX    A;
     5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
     6: if( my_rank==0 ) {n = 2; nnz = 3; ndz = 0;}
     7: else             {n = 2; nnz = 5; ndz = 0;}
     8: index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(LIS_INT) );
     9: value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
    10: lis_matrix_create(MPI_COMM_WORLD,&A);
    11: lis_matrix_set_size(A,n,0);
    12: if( my_rank==0 ) {
    13:     index[0] =  3; index[1] =  3; index[2] =  4; index[3] =  0;
    14:     value[0] = 11; value[1] = 22; value[2] =  0; value[3] = 21;}
    15: else {
    16:     index[0] =  3; index[1] =  4; index[2] =  6; index[3] =  1;
    17:     index[4] =  0; index[5] =  2;
    18:     value[0] = 33; value[1] = 44; value[2] =  0; value[3] = 32;
    19:     value[4] = 41; value[5] = 43;}
    20:  lis_matrix_set_msr(nnz,ndz,index,value,A);
    21:  lis_matrix_assemble(A);
```

5.3.3 Associating Arrays

To associate the arrays in the MSR format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_msr(LIS_INT nnz, LIS_INT ndz, LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_msr(LIS_INTEGER nnz, LIS_INTEGER ndz, LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

5.4 Diagonal (DIA)

The DIA format uses two arrays `index` and `value` to store data. Assume that nnd represents the number of nonzero diagonal elements of matrix A.

- `value` is a double precision array of length $nnd \times n$, which stores the values of the nonzero diagonal elements of matrix A.

- `index` is an integer array of length nnd, which stores the offsets from the main diagonal.

For the multithreaded environment, the following modifications have been made: the format uses two arrays `index` and `value` to store data. Assume that $nprocs$ represents the number of threads. nnd_p is the number of nonzero diagonal elements of the partial matrix into which the row block of matrix A is divided. $maxnnd$ is the maximum value nnd_p.

- `value` is a double precision array of length $maxnnd \times n$, which stores the values of the nonzero diagonal elements of matrix A.

- `index` is an integer array of length $nprocs \times maxnnd$, which stores the offsets from the main diagonal.

5.4.1 Creating Matrices (for Serial Environment)

The diagram on the right in Figure 8 shows how matrix A in Figure 8 is stored in the DIA format. A program to create the matrix in the DIA format is as follows:
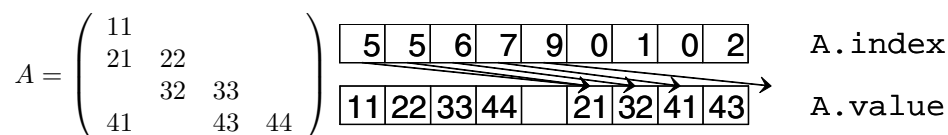
$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

-3	-1	0										A.index
0	0	0	41	0	21	32	43	11	22	33	44	A.value
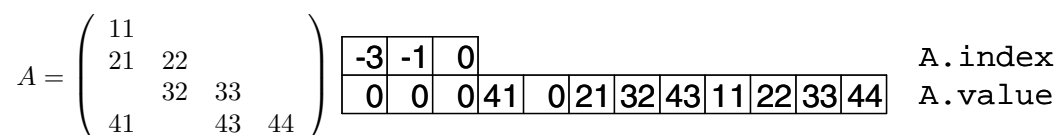
Figure 8: Data structure of DIA format (for serial environment).

```
For serial environment
 1: LIS_INT      n,nnd;
 2: LIS_INT      *index;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; nnd = 3;
 6: index = (LIS_INT *)malloc( nnd*sizeof(LIS_INT) );
 7: value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] = -3; index[1] = -1; index[2] =  0;
12: value[0] =  0; value[1] =  0; value[2] =  0; value[3] = 41;
13: value[4] =  0; value[5] = 21; value[6] = 32; value[7] = 43;
14: value[8] = 11; value[9] = 22; value[10]= 33; value[11]= 44;
15:
16:  lis_matrix_set_dia(nnd,index,value,A);
17:  lis_matrix_assemble(A);
```

5.4.2 Creating Matrices (for Multithreaded Environment)

Figure 9 shows how matrix A in Figure 8 is stored in the DIA format on two threads. A program to create the matrix in the DIA format on two threads is as follows:

-1	0			-3	-1	0					A.index
0	21	11	22			0	41	32	43	33	44	A.value

Figure 9: Data structure of DIA format (for multithreaded environment).

For multithreaded environment

```
 1: LIS_INT       n,maxnnd,nprocs;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; maxnnd = 3; nprocs = 2;
 6: index = (LIS_INT *)malloc( maxnnd*sizeof(LIS_INT) );
 7: value = (LIS_SCALAR *)malloc( n*maxnnd*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] = -1; index[1] =  0; index[2] =  0; index[3] = -3; index[4] = -1; index[5] =  0;
12: value[0] =  0; value[1] = 21; value[2] = 11; value[3] = 22; value[4] =  0; value[5] =  0;
13: value[6] =  0; value[7] = 41; value[8] = 32; value[9] = 43; value[10]= 33; value[11]= 44;
14:
15:  lis_matrix_set_dia(maxnnd,index,value,A);
16:  lis_matrix_assemble(A);
```

5.4.3 Creating Matrices (for Multiprocessing Environment)

Figure 10 shows how matrix A in Figure 8 is stored in the DIA format on two processing elements. A program to create the matrix in the DIA format on two processing elements is as follows:

Figure 10: Data structure of DIA format (for multiprocessing environment).

```
┌─ For multiprocessing environment ──────────────────────────────────────┐
  1: LIS_INT        i,n,nnd,my_rank;
  2: LIS_INT        *index;
  3: LIS_SCALAR     *value;
  4: LIS_MATRIX     A;
  5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  6: if( my_rank==0 ) {n = 2; nnd = 2;}
  7: else             {n = 2; nnd = 3;}
  8: index = (LIS_INT *)malloc( nnd*sizeof(LIS_INT) );
  9: value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
 10: lis_matrix_create(MPI_COMM_WORLD,&A);
 11: lis_matrix_set_size(A,n,0);
 12: if( my_rank==0 ) {
 13:     index[0] = -1; index[1] =  0;
 14:     value[0] =  0; value[1] = 21; value[2] = 11; value[3] = 22;}
 15: else {
 16:     index[0] = -3; index[1] = -1; index[2] =  0;
 17:     value[0] =  0; value[1] = 41; value[2] = 32; value[3] = 43; value[4] = 33;
 18:     value[5] = 44;}
 19:  lis_matrix_set_dia(nnd,index,value,A);
 20:  lis_matrix_assemble(A);
└─────────────────────────────────────────────────────────────────────────┘
```

5.4.4 Associating Arrays

To associate the arrays in the DIA format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_dia(LIS_INT nnd, LIS_INT index[],
  LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_dia(LIS_INTEGER nnd, LIS_INTEGER index(),
  LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

43

5.5 Ellpack-Itpack Generalized Diagonal (ELL)

The ELL format uses two arrays `index` and `value` to store data. Assume that $maxnzr$ is the maximum value of the number of nonzero elements in the rows of matrix A.

- `value` is a double precision array of length $maxnzr \times n$, which stores the values of the nonzero elements of the rows of matrix A along the column. The first column consists of the first nonzero elements of each row. If there is no nonzero elements to be stored, then 0 is stored.

- `index` is an integer array of length $maxnzr \times n$, which stores the column numbers of the nonzero elements stored in the array `value`. If the number of nonzero elements in the i-th row is nnz, then `index[`$nnz \times n + i$`]` stores row number i.

5.5.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 11 shows how matrix A in Figure 11 is stored in the ELL format. A program to create the matrix in the ELL format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

0	0	1	0	0	1	2	2	0	1	2	3	A.index
11	21	32	41	0	22	33	43	0	0	0	44	A.value

Figure 11: Data structure of ELL format (for serial and multithreaded environments).

```
┌─ For serial and multithreaded environments ─────────────────────────────┐
  1: LIS_INT       n,maxnzr;
  2: LIS_INT       *index;
  3: LIS_SCALAR    *value;
  4: LIS_MATRIX    A;
  5: n = 4; maxnzr = 3;
  6: index = (LIS_INT *)malloc( n*maxnzr*sizeof(LIS_INT) );
  7: value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
  8: lis_matrix_create(0,&A);
  9: lis_matrix_set_size(A,0,n);
 10:
 11: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0; index[4] =  0; index[5] =  1;
 12: index[6] =  2; index[7] =  2; index[8] =  0; index[9] =  1; index[10]=  2; index[11]=  3;
 13: value[0] = 11; value[1] = 21; value[2] = 32; value[3] = 41; value[4] =  0; value[5] = 22;
 14: value[6] = 33; value[7] = 43; value[8] =  0; value[9] =  0; value[10]=  0; value[11]= 44;
 15:
 16:  lis_matrix_set_ell(maxnzr,index,value,A);
 17:  lis_matrix_assemble(A);
└─────────────────────────────────────────────────────────────────────────┘
```

44

5.5.2 Creating Matrices (for Multiprocessing Environment)

Figure 12 shows how matrix A in Figure 11 is stored in the ELL format. A program to create the matrix in the ELL format on two processing elements is as follows:

Figure 12: Data structure of ELL format (for multiprocessing environment).

```
  For multiprocessing environment
  1: LIS_INT        i,n,maxnzr,my_rank;
  2: LIS_INT        *index;
  3: LIS_SCALAR     *value;
  4: LIS_MATRIX     A;
  5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  6: if( my_rank==0 ) {n = 2; maxnzr = 2;}
  7: else             {n = 2; maxnzr = 3;}
  8: index = (LIS_INT *)malloc( n*maxnzr*sizeof(LIS_INT) );
  9: value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
 10: lis_matrix_create(MPI_COMM_WORLD,&A);
 11: lis_matrix_set_size(A,n,0);
 12: if( my_rank==0 ) {
 13:     index[0] =  0; index[1] =  0; index[2] =  0; index[3] =  1;
 14:     value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 22;}
 15: else {
 16:     index[0] =  1; index[1] =  0; index[2] =  2; index[3] =  2; index[4] =  2;
 17:     index[5] =  3;
 18:     value[0] = 32; value[1] = 41; value[2] = 33; value[3] = 43; value[4] =  0;
 19:     value[5] = 44;}
 20:  lis_matrix_set_ell(maxnzr,index,value,A);
 21:  lis_matrix_assemble(A);
```

5.5.3 Associating Arrays

To associate an array required by the ELL format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_ell(LIS_INT maxnzr, LIS_INT index[],
  LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_ell(LIS_INTEGER maxnzr, LIS_INTEGER index(),
  LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

45

5.6 Jagged Diagonal (JAD)

The JAD format first sorts the nonzero elements of the rows in decreasing order of size, and then stores them along the column. The JAD format uses four arrays, `perm`, `ptr`, `index`, and `value`, to store data. Assume that $maxnzr$ represents the maximum value of the number of nonzero elements of matrix A.

- `perm` is an integer array of length n, which stores the sorted row numbers.

- `value` is a double precision array of length nnz, which stores the values of the jagged diagonal elements of the sorted matrix A. The first jagged diagonal consists of the values of the first nonzero elements of each row. The next jagged diagonal consists of the values of the second nonzero elements, and so on.

- `index` is an integer array of length nnz, which stores the row numbers of the nonzero elements stored in the array `value`.

- `ptr` is an integer array of length $maxnzr + 1$, which stores the starting points of the jagged diagonal elements.

For the multithreaded environment, the following modifications have been made: the format uses four arrays, `perm`, `ptr`, `index`, and `value`, to store data. Assume that $nprocs$ is the number of threads. $maxnzr_p$ is the number of nonzero diagonal elements of the partial matrix into which the row block of matrix A is divided. $maxmaxnzr$ is the maximum value of $maxnzr_p$.

- `perm` is an integer array of length n, which stores the sorted row numbers.

- `value` is a double precision array of length nnz, which stores the values of the jagged diagonal elements of the sorted matrix A. The first jagged diagonal consists of the values of the first nonzero elements of each row. The next jagged diagonal consist of the values of the second nonzero elements of each row, and so on.

- `index` is an integer array of length nnz, which stores the row numbers of the nonzero elements stored in the array `value`.

- `ptr` is an integer array of length $nprocs \times (maxmaxnzr + 1)$, which stores the starting points of the jagged diagonal elements.

5.6.1 Creating Matrices (for Serial Environment)

The diagram on the right in Figure 13 shows how matrix A in Figure 13 is stored in the JAD format. A program to create the matrix in the JAD format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

3	1	2	0	A.perm
0	4	7	8	A.ptr

0	0	1	0	2	1	2	3	A.index
41	21	32	11	43	22	33	44	A.value
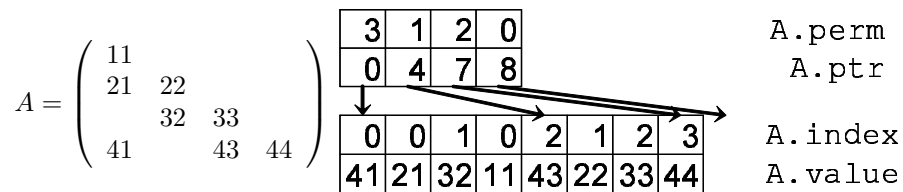
Figure 13: Data structure of JAD format (for serial environment).

For serial environment

```
 1: LIS_INT      n,nnz,maxnzr;
 2: LIS_INT      *perm,*ptr,*index;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; nnz = 8; maxnzr = 3;
 6: perm  = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
 7: ptr   = (LIS_INT *)malloc( (maxnzr+1)*sizeof(LIS_INT) );
 8: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 9: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
10: lis_matrix_create(0,&A);
11: lis_matrix_set_size(A,0,n);
12:
13: perm[0] = 3; perm[1] = 1; perm[2] = 2; perm[3] = 0;
14: ptr[0]  = 0; ptr[1]  = 4; ptr[2]  = 7; ptr[3]  = 8;
15: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0;
16: index[4] =  2; index[5] =  1; index[6] =  2; index[7] =  3;
17: value[0] = 41; value[1] = 21; value[2] = 32; value[3] = 11;
18: value[4] = 43; value[5] = 22; value[6] = 33; value[7] = 44;
19:
20:  lis_matrix_set_jad(nnz,maxnzr,perm,ptr,index,value,A);
21:  lis_matrix_assemble(A);
```

47

5.6.2 Creating Matrices (for Multithreaded Environment)

Figure 14 shows how matrix A in Figure 13 is stored in the JAD format on two threads. A program to create the matrix in the JAD format on two threads is as follows:
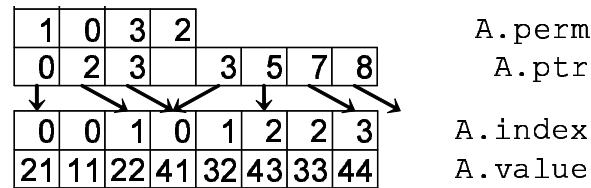
Figure 14: Data structure of JAD format (for multithreaded environment).

```
For multithreaded environment
 1: LIS_INT       n,nnz,maxmaxnzr,nprocs;
 2: LIS_INT       *perm,*ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8; maxmaxnzr = 3; nprocs = 2;
 6: perm  = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
 7: ptr   = (LIS_INT *)malloc( nprocs*(maxmaxnzr+1)*sizeof(LIS_INT) );
 8: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 9: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
10: lis_matrix_create(0,&A);
11: lis_matrix_set_size(A,0,n);
12:
13: perm[0] = 1; perm[1] = 0; perm[2] = 3; perm[3] = 2;
14: ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 3; ptr[3]  = 0;
15: ptr[4]  = 3; ptr[5]  = 5; ptr[6]  = 7; ptr[7]  = 8;
16: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0;
17: index[4] =  1; index[5] =  2; index[6] =  2; index[7] =  3;
18: value[0] = 21; value[1] = 11; value[2] = 22; value[3] = 41;
19: value[4] = 32; value[5] = 43; value[6] = 33; value[7] = 44;
20:
21:  lis_matrix_set_jad(nnz,maxmaxnzr,perm,ptr,index,value,A);
22:  lis_matrix_assemble(A);
```

5.6.3   Creating Matrices (for Multiprocessing Environment)

Figure 15 shows how matrix A in Figure 13 is stored in the JAD format on two processing elements. A program to create the matrix in the JAD format on two processing elements is as follows:

Figure 15: Data structure of JAD format (for multiprocessing environment).

```
 1: LIS_INT      i,n,nnz,maxnzr,my_rank;
 2: LIS_INT      *perm,*ptr,*index;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3; maxnzr = 2;}
 7: else             {n = 2; nnz = 5; maxnzr = 3;}
 8: perm  = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
 9: ptr   = (LIS_INT *)malloc( (maxnzr+1)*sizeof(LIS_INT) );
10: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
11: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
12: lis_matrix_create(MPI_COMM_WORLD,&A);
13: lis_matrix_set_size(A,n,0);
14: if( my_rank==0 ) {
15:     perm[0] = 1; perm[1] = 0;
16:     ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 3;
17:     index[0] =  0; index[1] =  0; index[2] =  1;
18:     value[0] = 21; value[1] = 11; value[2] = 22;}
19: else {
20:     perm[0] = 3; perm[1] = 2;
21:     ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 4; ptr[3]  = 5;
22:     index[0] =  0; index[1] =  1; index[2] =  2; index[3] =  2; index[4] =  3;
23:     value[0] = 41; value[1] = 32; value[2] = 43; value[3] = 33; value[4] = 44;}
24:  lis_matrix_set_jad(nnz,maxnzr,perm,ptr,index,value,A);
25:  lis_matrix_assemble(A);
```

5.6.4   Associating Arrays

To associate an array required by the JAD format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_jad(LIS_INT nnz, LIS_INT maxnzr, LIS_INT perm[], LIS_INT ptr[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_jad(LIS_INTEGER nnz, LIS_INTEGER maxnzr, LIS_INTEGER ptr(),integer index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

49

5.7 Block Sparse Row (BSR)

The BSR format breaks down matrix A into partial matrices called blocks of size $r \times c$. The BSR format stores the nonzero blocks, in which at least one nonzero element exists, in a format similar to that of CSR. Assume that $nr = n/r$ and $nnzb$ are the numbers of nonzero blocks of A. The BSR format uses three arrays `bptr`, `bindex` and `value` to store data.

- `value` is a double precision array of length $nnzb \times r \times c$, which stores the values of the elements of the nonzero blocks.

- `bindex` is an integer array of length $nnzb$, which stores the block column numbers of the nonzero blocks.

- `bptr` is an integer array of length $nr + 1$, which stores the starting points of the block rows in the array `bindex`.

5.7.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 16 shows how matrix A in Figure 16 is stored in the BSR format. A program to create the matrix in the BSR format is as follows:

Figure 16: Data structure of BSR format (for serial and multithreaded environments).

```
 For serial and multithreaded environments
 1: LIS_INT        n,bnr,bnc,nr,nc,bnnz;
 2: LIS_INT        *bptr,*bindex;
 3: LIS_SCALAR     *value;
 4: LIS_MATRIX     A;
 5: n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
 6: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
 7: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
 8: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
13: bindex[0] =  0; bindex[1] =  0; bindex[2] =  1;
14: value[0]  = 11; value[1] = 21; value[2] =   0; value[3] = 22;
15: value[4]  =  0; value[5] = 41; value[6] = 32; value[7] =   0;
16: value[8]  = 33; value[9] = 43; value[10]=  0; value[11]= 44;
17:
18:  lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
19:  lis_matrix_assemble(A);
```

5.7.2 Creating Matrices (for Multiprocessing Environment)

Figure 17 shows how matrix A in Figure 16 is stored in the BSR format on two processing elements. A program to create the matrix in the BSR format on two processing elements is as follows:
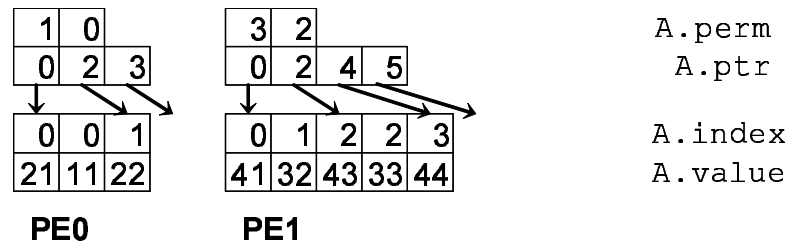

Figure 17: Data structure of BSR format (for multiprocessing environment).

```
 For multiprocessing environment
 1: LIS_INT       n,bnr,bnc,nr,nc,bnnz,my_rank;
 2: LIS_INT       *bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; bnr = 2; bnc = 2; bnnz = 1; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 7: else             {n = 2; bnr = 2; bnc = 2; bnnz = 2; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 8: bptr  = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
 9: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
10: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     bptr[0] = 0; bptr[1] = 1;
15:     bindex[0] =  0;
16:     value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;}
17: else {
18:     bptr[0] = 0; bptr[1] = 2;
19:     bindex[0] =  0; bindex[1] =  1;
20:     value[0]  =  0; value[1]  = 41; value[2] = 32; value[3] =  0;
21:     value[4]  = 33; value[5]  = 43; value[6] =  0; value[7] = 44;}
22:  lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
23:  lis_matrix_assemble(A);
```

5.7.3 Associating Arrays

To associate the arrays in the BSR format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_bsr(LIS_INT bnr, LIS_INT bnc, LIS_INT bnnz, LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_bsr(LIS_INTEGER bnr, LIS_INTEGER bnc, LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

51

5.8 Block Sparse Column (BSC)

The BSC format breaks down matrix A into partial matrices called blocks of size $r \times c$. The BSC format stores the nonzero blocks, in which at least one nonzero element exists, in a format similar to that of CSC. Assume that $nc = n/c$ and $nnzb$ are the numbers of the nonzero blocks of A. The BSC format uses three arrays `bptr`, `bindex` and `value` to store data.

- `value` is a double precision array of length $nnzb \times r \times c$, which stores the values of the elements of the nonzero blocks.

- `bindex` is an integer array of length $nnzb$, which stores the block row numbers of the nonzero blocks.

- `bptr` is an integer array of length $nc + 1$, which stores the starting points of the block columns in the array `bindex`.

5.8.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 18 shows how matrix A in Figure 18 is stored in the BSC format. A program to create the matrix in the BSC format is as follows:
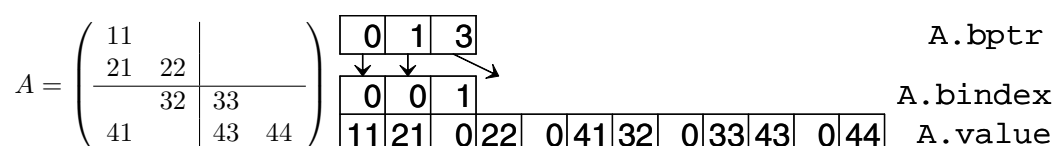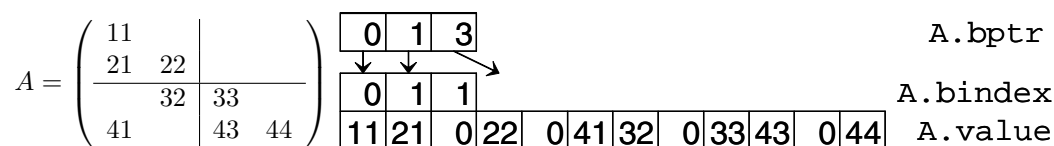

Figure 18: Data structure of BSC format (for serial and multithreaded environments).

```
For serial and multithreaded environments
 1: LIS_INT      n,bnr,bnc,nr,nc,bnnz;
 2: LIS_INT      *bptr,*bindex;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
 6: bptr  = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
 7: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
 8: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
13: bindex[0] =  0; bindex[1] =  1; bindex[2] =  1;
14: value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
15: value[4]  =  0; value[5] = 41; value[6] = 32; value[7] =  0;
16: value[8]  = 33; value[9] = 43; value[10]=  0; value[11]= 44;
17:
18:  lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
19:  lis_matrix_assemble(A);
```

5.8.2 Creating Matrices (for Multiprocessing Environment)

Figure 19 shows how matrix A in Figure 18 is stored in the BSC format on two processing elements. A program to create the matrix in the BSC format on two processing elements is as follows:
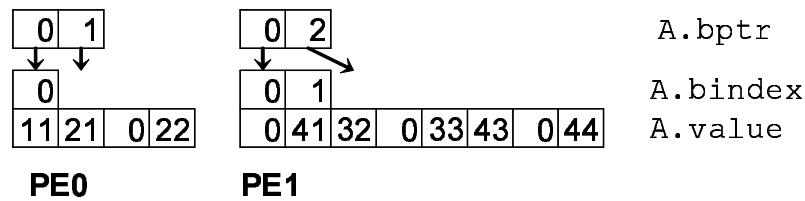

Figure 19: Data structure of BSC format (for multiprocessing environment).

```
   For multiprocessing environment
 1: LIS_INT        n,bnr,bnc,nr,nc,bnnz,my_rank;
 2: LIS_INT        *bptr,*bindex;
 3: LIS_SCALAR     *value;
 4: LIS_MATRIX     A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; bnr = 2; bnc = 2; bnnz = 2; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 7: else             {n = 2; bnr = 2; bnc = 2; bnnz = 1; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 8: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
 9: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
10: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     bptr[0] = 0; bptr[1] = 2;
15:     bindex[0] =  0; bindex[1] =  1;
16:     value[0]  = 11; value[1]  = 21; value[2] =  0; value[3] = 22;
17:     value[4]  =  0; value[5]  = 41; value[6] = 32; value[7] =  0;}
18: else {
19:     bptr[0] = 0; bptr[1] = 1;
20:     bindex[0] =  1;
21:     value[0]  = 33; value[1]  = 43; value[2] =  0; value[3] = 44;}
22:   lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
23:   lis_matrix_assemble(A);
```

5.8.3 Associating Arrays

To associate the arrays in the BSC format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_bsc(LIS_INT bnr, LIS_INT bnc, LIS_INT bnnz,
  LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_bsc(LIS_INTEGER bnr, LIS_INTEGER bnc,
  LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(),
  LIS_MATRIX A, LIS_INTEGER ierr)

5.9 Variable Block Row (VBR)

The VBR format is the generalized version of the BSR format. The division points of the rows and columns are given by the arrays `row` and `col`. The VBR format stores the nonzero blocks (the blocks in which at least one nonzero element exists) in a format similar to that of CSR. Assume that nr and nc are the numbers of row and column divisions, respectively, and that $nnzb$ denotes the number of nonzero blocks of A, and nnz denotes the total number of elements of the nonzero blocks. The VBR format uses six arrays, `bptr`, `bindex`, `row`, `col`, `ptr`, and `value`, to store data.

- `row` is an integer array of length $nr + 1$, which stores the starting row number of the block rows.

- `col` is an integer array of length $nc + 1$, which stores the starting column number of the block columns.

- `bindex` is an integer array of length $nnzb$, which stores the block column numbers of the nonzero blocks.

- `bptr` is an integer array of length $nr + 1$, which stores the starting points of the block rows in the array `bindex`.

- `value` is a double precision array of length nnz, which stores the values of the elements of the nonzero blocks.

- `ptr` is an integer array of length $nnzb + 1$, which stores the starting points of the nonzero blocks in the array `value`.

5.9.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 20 shows how matrix A in Figure 20 is stored in the VBR format. A program to create the matrix in the VBR format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

| 0 | 1 | 3 | 6 | | | A.bptr |

0	0	1	0	1	2	A.bindex
0	1	3	4			A.row
0	1	3	4			A.col
0	1	3	7	8	10	11	A.ptr

| 11 | 21 | 0 | 22 | 32 | 0 | 33 | 41 | 0 | 43 | 44 | A.value |

Figure 20: Data structure of VBR format (for serial and multithreaded environments).

For serial and multithreaded environments

```
 1: LIS_INT       n,nnz,nr,nc,bnnz;
 2: LIS_INT       *row,*col,*ptr,*bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 11; bnnz = 6; nr = 3; nc = 3;
 6: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
 7: row    = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
 8: col    = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
 9: ptr    = (LIS_INT *)malloc( (bnnz+1)*sizeof(LIS_INT) );
10: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
11: value  = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
12: lis_matrix_create(0,&A);
13: lis_matrix_set_size(A,0,n);
14:
15: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3; bptr[3] = 6;
16: row[0]  = 0; row[1]  = 1; row[2]  = 3; row[3] = 4;
17: col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
18: bindex[0] =  0; bindex[1] =  0; bindex[2] =  1; bindex[3] =  0;
19: bindex[4] =  1; bindex[5] =  2;
20: ptr[0]     = 0; ptr[1]     = 1; ptr[2]     = 3; ptr[3]     = 7;
21: ptr[4]     = 8; ptr[5]     = 10; ptr[6]     = 11;
22: value[0]  = 11; value[1]  = 21; value[2]  =  0; value[3]  = 22;
23: value[4]  = 32; value[5]  =  0; value[6]  = 33; value[7]  = 41;
24: value[8]  =  0; value[9]  = 43; value[10] = 44;
25:
26:  lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
27:  lis_matrix_assemble(A);
```

5.9.2 Creating Matrices (for Multiprocessing Environment)

Figure 21 shows how matrix *A* in Figure 20 is stored in the VBR format on two processing elements. A program to create the matrix in the VBR format on two processing elements is as follows:
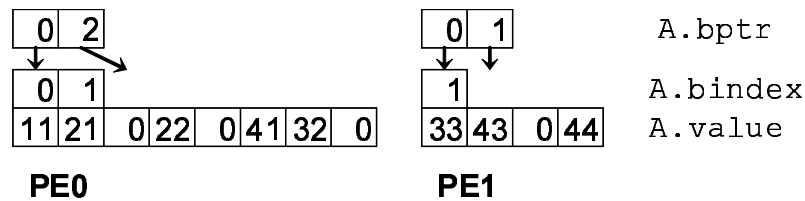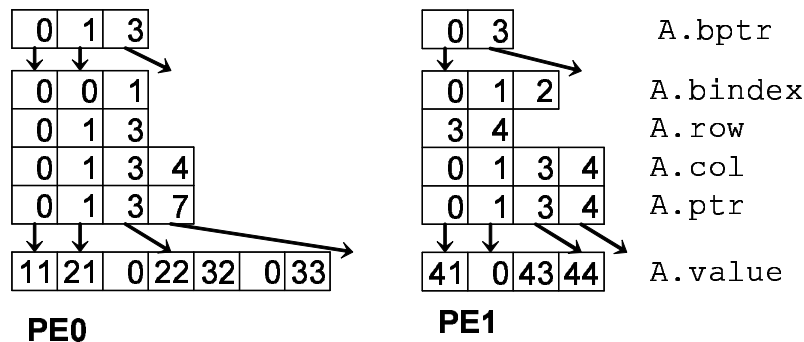

Figure 21: Data structure of VBR format (for multiprocessing environment).

```
 1: LIS_INT      n,nnz,nr,nc,bnnz,my_rank;
 2: LIS_INT      *row,*col,*ptr,*bptr,*bindex;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 7; bnnz = 3; nr = 2; nc = 3;}
 7: else             {n = 2; nnz = 4; bnnz = 3; nr = 1; nc = 3;}
 8: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
 9: row    = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
10: col    = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
11: ptr    = (LIS_INT *)malloc( (bnnz+1)*sizeof(LIS_INT) );
12: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
13: value  = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
14: lis_matrix_create(MPI_COMM_WORLD,&A);
15: lis_matrix_set_size(A,n,0);
16: if( my_rank==0 ) {
17:     bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
18:     row[0]  = 0; row[1]  = 1; row[2]  = 3;
19:     col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
20:     bindex[0] =  0; bindex[1] =  0; bindex[2] =  1;
21:     ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  7;
22:     value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
23:     value[4]  = 32; value[5] =  0; value[6] = 33;}
24: else {
25:     bptr[0] = 0; bptr[1] = 3;
26:     row[0]  = 3; row[1]  = 4;
27:     col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
28:     bindex[0] =  0; bindex[1] =  1; bindex[2] =  2;
29:     ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  4;
30:     value[0]  = 41; value[1] =  0; value[2]  = 43; value[3]  = 44;}
31:   lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
32:   lis_matrix_assemble(A);
```

5.9.3 Associating Arrays

To associate the arrays in the VBR format with matrix A, the following functions are used:

- C   `LIS_INT lis_matrix_set_vbr(LIS_INT nnz, LIS_INT nr, LIS_INT nc, LIS_INT bnnz, LIS_INT row[], LIS_INT col[], LIS_INT ptr[], LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)`

- `Fortran subroutine lis_matrix_set_vbr(LIS_INTEGER nnz, LIS_INTEGER nr, LIS_INTEGER nc, LIS_INTEGER bnnz, LIS_INTEGER row(), LIS_INTEGER col(), LIS_INTEGER ptr(), LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)`

5.10 Coordinate (COO)

The COO format uses three arrays `row`, `col` and `value` to store data.

- `value` is a double precision array of length nnz, which stores the values of the nonzero elements.

- `row` is an integer array of length nnz, which stores the row numbers of the nonzero elements.

- `col` is an integer array of length nnz, which stores the column numbers of the nonzero elements.

5.10.1 Creating Matrices (for Serial and Multithreaded Environments)

The diagram on the right in Figure 22 shows how matrix A in Figure 22 is stored in the COO format. A program to create the matrix in the COO format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

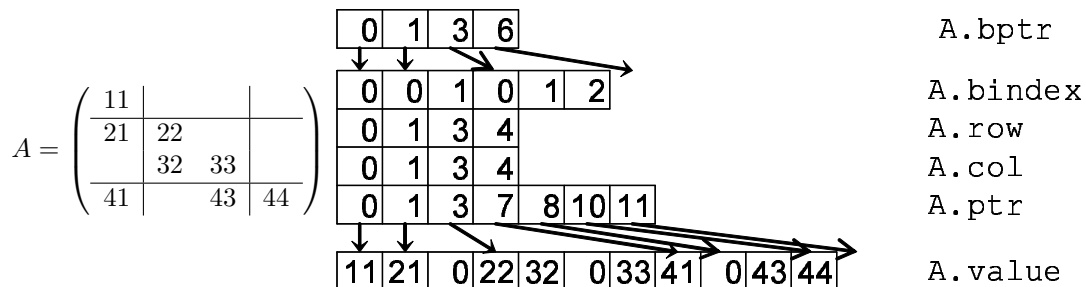| 0	1	3	1	2	2	3	3	A.row
0	0	0	1	1	2	2	3	A.col
11	21	41	22	32	33	43	44	A.value

Figure 22: Data structure of COO format (for serial and multithreaded environments).

```
For serial and multithreaded environments
 1: LIS_INT      n,nnz;
 2: LIS_INT      *row,*col;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: n = 4; nnz = 8;
 6: row   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 7: col   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: row[0] = 0; row[1] = 1; row[2] = 3; row[3] = 1;
13: row[4] = 2; row[5] = 2; row[6] = 3; row[7] = 3;
14: col[0] = 0; col[1] = 0; col[2] = 0; col[3] = 1;
15: col[4] = 1; col[5] = 2; col[6] = 2; col[7] = 3;
16: value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22;
17: value[4] = 32; value[5] = 33; value[6] = 43; value[7] = 44;
18:
19:  lis_matrix_set_coo(nnz,row,col,value,A);
20:  lis_matrix_assemble(A);
```

5.10.2 Creating Matrices (for Multiprocessing Environment)

Figure 23 shows how matrix A in Figure 22 is stored in the COO format on two processing elements. A program to create the matrix in the COO format on two processing elements is as follows:

0	1	1		3	2	2	3	3	A.row
0	0	1		0	1	2	2	3	A.col
11	21	22		41	32	33	43	44	A.value

PE0　　　　**PE1**

Figure 23: Data structure of COO format (for multiprocessing environment).

```
For multiprocessing environment

 1: LIS_INT      n,nnz,my_rank;
 2: LIS_INT      *row,*col;
 3: LIS_SCALAR   *value;
 4: LIS_MATRIX   A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3;}
 7: else            {n = 2; nnz = 5;}
 8: row   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
 9: col   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
10: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     row[0] = 0; row[1] = 1; row[2] = 1;
15:     col[0] = 0; col[1] = 0; col[2] = 1;
16:     value[0] = 11; value[1] = 21; value[2] = 22;}
17: else {
18:     row[0] = 3; row[1] = 2; row[2] = 2; row[3] = 3; row[4] = 3;
19:     col[0] = 0; col[1] = 1; col[2] = 2; col[3] = 2; col[4] = 3;
20:     value[0] = 41; value[1] = 32; value[2] = 33; value[3] = 43; value[4] = 44;}
21:  lis_matrix_set_coo(nnz,row,col,value,A);
22:  lis_matrix_assemble(A);
```

5.10.3 Associating Arrays

To associate the arrays in the COO format with matrix A, the following functions are used:

- C　　　　LIS_INT lis_matrix_set_coo(LIS_INT nnz, LIS_INT row[], LIS_INT col[],
  LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_coo(LIS_INTEGER nnz, LIS_INTEGER row(),
  LIS_INTEGER col(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

5.11 Dense (DNS)

The DNS format uses one array `value` to store data.

- `value` is a double precision array of length $n \times n$, which stores the values of the elements with priority given to the columns.

5.11.1 Creating Matrices (for Serial and Multithreaded Environments)

The right diagram in Figure 24 shows how matrix A in Figure 24 is stored in the DNS format. A program to create the matrix in the DNS format is as follows:

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

11	21	0	41	0	22	32	0
0	0	33	43	0	0	0	44

A.Value

Figure 24: Data structure of DNS format (for serial and multithreaded environments).

For serial and multithreaded environments

```
 1: LIS_INT      n;
 2: LIS_SCALAR   *value;
 3: LIS_MATRIX   A;
 4: n = 4;
 5: value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
 6: lis_matrix_create(0,&A);
 7: lis_matrix_set_size(A,0,n);
 8:
 9: value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 41;
10: value[4] =  0; value[5] = 22; value[6] = 32; value[7] =  0;
11: value[8] =  0; value[9] =  0; value[10]= 33; value[11]= 43;
12: value[12]=  0; value[13]=  0; value[14]=  0; value[15]= 44;
13:
14:  lis_matrix_set_dns(value,A);
15:  lis_matrix_assemble(A);
```

5.11.2 Creating Matrices (for Multiprocessing Environment)

Figure 25 shows how matrix A in Figure 24 is stored in the DNS format on two processing elements. A program to create the matrix in the DNS format on two processing elements is as follows:

11	21	0	22
0	0	0	0

0	41	32	0
33	43	0	44

A.Value

PE0   **PE1**

Figure 25: Data structure of DNS format (for multiprocessing environment).

```
For multiprocessing environment
 1: LIS_INT       n,my_rank;
 2: LIS_SCALAR    *value;
 3: LIS_MATRIX    A;
 4: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 5: if( my_rank==0 ) {n = 2;}
 6: else              {n = 2;}
 7: value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(MPI_COMM_WORLD,&A);
 9: lis_matrix_set_size(A,n,0);
10: if( my_rank==0 ) {
11:     value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 22;
12:     value[4] =  0; value[5] =  0; value[6] =  0; value[7] =  0;}
13: else {
14:     value[0] =  0; value[1] = 41; value[2] = 32; value[3] =  0;
15:     value[4] = 33; value[5] = 43; value[6] =  0; value[7] = 44;}
16:  lis_matrix_set_dns(value,A);
17:  lis_matrix_assemble(A);
```

5.11.3 Associating Arrays

To associate the arrays in the DNS format with matrix A, the following functions are used:

- C   LIS_INT lis_matrix_set_dns(LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_dns(LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

6 Functions

This section describes the functions which can be employed by the user. The return codes of the functions in C and the values of `ierr` in Fortran are:

`LIS_SUCCESS(0)`	Normal termination
`LIS_ILL_OPTION(1)`	Illegal option
`LIS_BREAKDOWN(2)`	Breakdown
`LIS_OUT_OF_MEMORY(3)`	Out of working memory
`LIS_MAXITER(4)`	Maximum number of iterations
`LIS_NOT_IMPLEMENTED(5)`	Not implemented
`LIS_ERR_FILE_IO(6)`	File I/O error

6.1 Operating Vector Elements

Assume that the size of vector v is *global_n* and that the size of the partial vectors stored on *nprocs* processing elements is *local_n*. *global_n* and *local_n* are called the global size and the local size, respectively.

6.1.1 lis_vector_create

```
C       LIS_INT lis_vector_create(LIS_Comm comm, LIS_VECTOR *v)
Fortran subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR v, LIS_INTEGER ierr)
```

Description
    Create vector v.

Input

LIS_Comm	The MPI communicator

Output

v	The vector
ierr	The return code

Note
    For the serial and multithreaded environments, the value of `comm` is ignored.

6.1.2 lis_vector_destroy

```
C       LIS_INT lis_vector_destroy(LIS_VECTOR v)
Fortran subroutine lis_vector_destroy(LIS_VECTOR v, LIS_INTEGER ierr)
```

Description
    Destroy vector v.

Input

v	The vector to be destroyed

Output

ierr	The return code

6.1.3 lis_vector_duplicate

```
C        LIS_INT lis_vector_duplicate(void *vin, LIS_VECTOR *vout)
Fortran subroutine lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout,
         LIS_INTEGER ierr)
```

Description

Create vector v_{out}, which has the same information as v_{in}.

Input

vin	The source vector or matrix

Output

vout	The destination vector
ierr	The return code

Note

The function `lis_vector_duplicate` does not copy the values, but allocates only the memory. To copy the values as well, the function `lis_vector_copy` must be called after this function.

6.1.4 lis_vector_set_size

```
C        LIS_INT lis_vector_set_size(LIS_VECTOR v, LIS_INT local_n,
         LIS_INT global_n)
Fortran subroutine lis_vector_set_size(LIS_VECTOR v, LIS_INTEGER local_n,
         LIS_INTEGER global_n, LIS_INTEGER ierr)
```

Description

Assign the size of vector v.

Input

v	The vector
local_n	The size of the partial vector
global_n	The size of the global vector

Output

ierr	The return code

Note

Either *local_n* or *global_n* must be provided.

For the serial and multithreaded environments, *local_n* is equal to *global_n*. Therefore, both `lis_vector_set_size(v,n,0)` and `lis_vector_set_size(v,0,n)` create a vector of size n.

For the multiprocessing environment, `lis_vector_set_size(v,n,0)` creates a partial vector of size n on each processing element. On the other hand, `lis_vector_set_size(v,0,n)` creates a partial vector of size m_p on processing element p. The values of m_p are determined by the library.

6.1.5   lis_vector_get_size

```
C        LIS_INT lis_vector_get_size(LIS_VECTOR v, LIS_INT *local_n,
C         LIS_INT *global_n)
Fortran subroutine lis_vector_get_size(LIS_VECTOR v, LIS_INTEGER local_n,
         LIS_INTEGER global_n, LIS_INTEGER ierr)
```

Description
  Get the size of vector v.

Input

  v   The vector

Output

  local_n   The size of the partial vector

  global_n   The size of the global vector

  ierr   The return code

Note
  For the serial and multithreaded environments, $local\_n$ is equal to $global\_n$.

6.1.6   lis_vector_get_range

```
C        LIS_INT lis_vector_get_range(LIS_VECTOR v, LIS_INT *is, LIS_INT *ie)
Fortran subroutine lis_vector_get_range(LIS_VECTOR v, LIS_INTEGER is,
         LIS_INTEGER ie, LIS_INTEGER ierr)
```

Description
  Get the location of the partial vector v in the global vector.

Input

  v   The partial vector

Output

  is   The location where the partial vector v starts in the global vector

  ie   The location where the partial vector v ends in the global vector

  ierr   The return code

Note
  For the serial and multithreaded environments, a vector of size n results in $is = 0$ and $ie = n$.

6.1.7 lis_vector_set_value

```
C        LIS_INT lis_vector_set_value(LIS_INT flag, LIS_INT i, LIS_SCALAR value,
         LIS_VECTOR v)
Fortran subroutine lis_vector_set_value(LIS_INTEGER flag, LIS_INTEGER i,
         LIS_SCALAR value, LIS_VECTOR v, LIS_INTEGER ierr)
```

Description

Assign the scalar value to the i-th row of vector v.

Input

flag	LIS_INS_VALUE : $v[i] = value$
	LIS_ADD_VALUE : $v[i] = v[i] + value$
i	The location where the value is assigned
value	The scalar value to be assigned
v	The vector

Output

v	The vector with the scalar value assigned to the i-th row
ierr	The return code

Note

For the multiprocessing environment, the i-th row of the global vector must be specified instead of the i-th row of the partial vector.

6.1.8 lis_vector_get_value

```
C        LIS_INT lis_vector_get_value(LIS_VECTOR v, LIS_INT i, LIS_SCALAR *value)
Fortran subroutine lis_vector_get_value(LIS_VECTOR v, LIS_INTEGER i,
         LIS_SCALAR value, LIS_INTEGER ierr)
```

Description

Get the scalar value of the i-th row of vector v.

Input

i	The location where the value is assigned
v	The source vector

Output

value	The value of the i-th row
ierr	The return code

Note

For the multiprocessing environment, the i-th row of the global vector must be specified.

6.1.9　lis_vector_set_values

```
C        LIS_INT lis_vector_set_values(LIS_INT flag, LIS_INT count,
          LIS_INT index[], LIS_SCALAR value[], LIS_VECTOR v)
Fortran subroutine lis_vector_set_values(LIS_INTEGER flag, LIS_INTEGER count,
          LIS_INTEGER index(), LIS_SCALAR value(), LIS_VECTOR v, LIS_INTEGER ierr)
```

Description

　　Assign scalar $value[i]$ to the $index[i]$-th row of vector v, where $i = 0, 1, ..., count - 1$.

Input

flag	LIS_INS_VALUE : $v[index[i]] = value[i]$
	LIS_ADD_VALUE : $v[index[i]] = v[index[i]] + value[i]$
count	The number of elements in the array that stores the scalar values to be assigned
index	The array that stores the location where the scalar values are assigned
value	The array that stores the scalar values to be assigned
v	The vector

Output

v	The vector with scalar $value[i]$ assigned to the $index[i]$-th row
ierr	The return code

Note

　　For the multiprocessing environment, the $index[i]$-th row of the global vector must be specified instead of the $index[i]$-th row of the partial vector.

6.1.10 lis_vector_get_values

```
C        LIS_INT lis_vector_get_values(LIS_VECTOR v, LIS_INT start, LIS_INT count,
            LIS_SCALAR value[])
Fortran subroutine lis_vector_get_values(LIS_VECTOR v, LIS_INTEGER start,
            LIS_INTEGER count, LIS_SCALAR value(), LIS_INTEGER ierr)
```

Description

Get scalar $value[i]$ of the $start + i$-th row of vector v, where $i = 0, 1, ..., count - 1$.

Input

start	The starting location
count	The number of values to get
v	The source vector

Output

value	The array to store the scalar values
ierr	The return code

Note

For the multiprocessing environment, the $start + i$-th row of the global vector must be specified.

6.1.11 lis_vector_scatter

```
C        LIS_INT lis_vector_scatter(LIS_SCALAR value[], LIS_VECTOR v)
Fortran subroutine lis_vector_scatter(LIS_SCALAR value(), LIS_VECTOR v,
            LIS_INTEGER ierr)
```

Description

Assign scalar $value[i]$ to the i-th row of vector v, where $i = 0, 1, ..., global\_n - 1$.

Input

value	The array that stores the scalar values to be assigned

Output

v	The vector
ierr	The return code

Note

6.1.12 lis_vector_gather

```
C       LIS_INT lis_vector_gather(LIS_VECTOR v, LIS_SCALAR value[])
Fortran subroutine lis_vector_gather(LIS_VECTOR v, LIS_SCALAR value(),
        LIS_INTEGER ierr)
```

Description

Get scalar $value[i]$ of the i-th row of vector v, where $i = 0, 1, ..., global\_n - 1$.

Input

v   The source vector

Output

value   The array that stores the scalar values

ierr   The return code

Note

6.1.13 lis_vector_copy

```
C       LIS_INT lis_vector_copy(LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_vector_copy(LIS_VECTOR x, LIS_VECTOR y, LIS_INTEGER ierr)
```

Description

Copy the values of the vector elements.

Input

x   The source vector

Output

y   The destination vector

ierr   The return code

6.1.14 lis_vector_set_all

```
C       LIS_INT lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR v)
Fortran subroutine lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR v,
        LIS_INTEGER ierr)
```

Description

Assign the scalar value to the elements of vector v.

Input

value   The scalar value to be assigned

v   The vector

Output

v   The vector with the value assigned to the elements

ierr   The return code

6.2 Operating Matrix Elements

Assume that the size of matrix A is $global\_n \times global\_n$ and that the size of each partial matrix stored on $nprocs$ processing elements is $local\_n \times global\_n$. Here, $global\_n$ and $local\_n$ are called the number of rows of the global matrix and the number of rows of the partial matrix, respectively.

6.2.1 lis_matrix_create

```
C       LIS_INT lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)
Fortran subroutine lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, LIS_INTEGER ierr)
```

Description
   Create matrix A.

Input

LIS_Comm   The MPI communicator

Output

A   The matrix

ierr   The return code

Note
   For the serial and multithreaded environments, the value of `comm` is ignored.

6.2.2 lis_matrix_destroy

```
C       LIS_INT lis_matrix_destroy(LIS_MATRIX A)
Fortran subroutine lis_matrix_destroy(LIS_MATRIX A, LIS_INTEGER ierr)
```

Description
   Destroy matrix A.

Input

A   The matrix to be destroyed

Output

ierr   The return code

6.2.3 lis_matrix_duplicate

```
C       LIS_INT lis_matrix_duplicate(LIS_MATRIX Ain, LIS_MATRIX *Aout)
Fortran subroutine lis_matrix_duplicate(LIS_MATRIX Ain, LIS_MATRIX Aout,
        LIS_INTEGER ierr)
```

Description

   Create matrix A_{out} which has the same information as A_{in}.

Input

   Ain   The source matrix

Output

   Aout   The destination matrix

   ierr   The return code

Note

   The function `lis_matrix_duplicate` does not copy the values of the elements of the matrix, but allocates only the memory. To copy the values of the elements as well, the function `lis_matrix_copy` must be called after this function.

6.2.4 lis_matrix_malloc

```
C       LIS_INT lis_matrix_malloc(LIS_MATRIX A, LIS_INT nnz_row, LIS_INT nnz[])
Fortran subroutine lis_matrix_malloc(LIS_MATRIX A, LIS_INTEGER nnz_row,
        LIS_INTEGER nnz[], LIS_INTEGER ierr)
```

Description

   Allocate the memory for matrix A.

Input

   A   The matrix

   nnz_row   The average number of nonzero elements

   nnz   The array of numbers of nonzero elements in each row

Output

   ierr   The return code

Note

   Either `nnz_row` or `nnz` must be provided.

6.2.5   lis_matrix_set_value

```
C       LIS_INT lis_matrix_set_value(LIS_INT flag, LIS_INT i, LIS_INT j,
         LIS_SCALAR value, LIS_MATRIX A)
Fortran subroutine lis_matrix_set_value(LIS_INTEGER flag, LIS_INTEGER i,
         LIS_INTEGER j, LIS_SCALAR value, LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Assign the scalar value to the (i, j)-th element of matrix A.

Input

flag	LIS_INS_VALUE : $A[i,j] = value$
	LIS_ADD_VALUE : $A[i,j] = A[i,j] + value$
i	The row number of the matrix
j	The column number of the matrix
value	The value to be assigned
A	The matrix

Output

A	The matrix
ierr	The return code

Note

For the multiprocessing environment, the i-th row and the j-th column of the global matrix must be specified.

The function `lis_matrix_set_value` stores the assigned value in a temporary internal format. Therefore, after `lis_matrix_set_value` is called, the function `lis_matrix_assemble` must be called.

6.2.6   lis_matrix_assemble

```
C       LIS_INT lis_matrix_assemble(LIS_MATRIX A)
Fortran subroutine lis_matrix_assemble(LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Assemble matrix A into the specified storage format.

Input

A	The matrix

Output

A	The matrix assembled into the specified storage format
ierr	The return code

6.2.7 lis_matrix_set_size

```
C        LIS_INT lis_matrix_set_size(LIS_MATRIX A, LIS_INT local_n,
         LIS_INT global_n)
Fortran subroutine lis_matrix_set_size(LIS_MATRIX A, LIS_INTEGER local_n,
         LIS_INTEGER global_n, LIS_INTEGER ierr)
```

Description

Assign the size of matrix A.

Input

A	The matrix
local_n	The number of rows of the partial matrix
global_n	The number of rows of the global matrix

Output

ierr	The return code

Note

Either *local_n* or *global_n* must be provided.

For the serial and multithreaded environments, *local_n* is equal to *global_n*. Therefore, both `lis_matrix_set_size(A,n,0)` and `lis_matrix_set_size(A,0,n)` create a matrix of size $n \times n$.

For the multiprocessing environment, `lis_matrix_set_size(A,n,0)` creates a partial matrix of size $n \times N$ on each processing element, where N is the total sum of n. On the other hand, `lis_matrix_set_size(A,0,n)` creates a partial matrix of size $m_p \times n$ on processing element p. The values of m_p are determined by the library.

6.2.8 lis_matrix_get_size

```
C        LIS_INT lis_matrix_get_size(LIS_MATRIX A, LIS_INT *local_n,
         LIS_INT *global_n)
Fortran subroutine lis_matrix_get_size(LIS_MATRIX A, LIS_INTEGER local_n,
         LIS_INTEGER global_n, LIS_INTEGER ierr)
```

Description

Get the size of matrix A.

Input

A	The matrix

Output

local_n	The number of rows of the partial matrix
global_n	The number of rows of the global matrix
ierr	The return code

Note

For the serial and multithreaded environments, *local_n* is equal to *global_n*.

6.2.9 lis_matrix_get_range

```
C        LIS_INT lis_matrix_get_range(LIS_MATRIX A, LIS_INT *is, LIS_INT *ie)
Fortran subroutine lis_matrix_get_range(LIS_MATRIX A, LIS_INTEGER is,
         LIS_INTEGER ie, LIS_INTEGER ierr)
```

Description

Get the location of the partial matrix A in the global matrix.

Input

A   The partial matrix

Output

is   The location where the partial matrix A starts in the global matrix

ie   The location where the partial matrix A ends in the global matrix

ierr   The return code

Note

For the serial and multithreaded environments, a matrix of $n \times n$ results in $is = 0$ and $ie = n$.

6.2.10   lis_matrix_set_type

```
C       LIS_INT lis_matrix_set_type(LIS_MATRIX A, LIS_INT matrix_type)
Fortran subroutine lis_matrix_set_type(LIS_MATRIX A, LIS_INTEGER matrix_type,
        LIS_INTEGER ierr)
```

Description
  Assign the storage format.

Input

A   The matrix

matrix_type   The storage format

Output

ierr   The return code

Note
  matrix_type of A is LIS_MATRIX_CSR when the matrix is created. The table below shows the available storage formats for matrix_type.

Storage format		matrix_type
Compressed Sparse Row	(CSR)	{LIS_MATRIX_CSR|1}
Compressed Sparse Column	(CSC)	{LIS_MATRIX_CSC|2}
Modified Compressed Sparse Row	(MSR)	{LIS_MATRIX_MSR|3}
Diagonal	(DIA)	{LIS_MATRIX_DIA|4}
Ellpack-Itpack Generalized Diagonal	(ELL)	{LIS_MATRIX_ELL|5}
Jagged Diagonal	(JAD)	{LIS_MATRIX_JAD|6}
Block Sparse Row	(BSR)	{LIS_MATRIX_BSR|7}
Block Sparse Column	(BSC)	{LIS_MATRIX_BSC|8}
Variable Block Row	(VBR)	{LIS_MATRIX_VBR|9}
Coordinate	(COO)	{LIS_MATRIX_COO|10}
Dense	(DNS)	{LIS_MATRIX_DNS|11}

6.2.11   lis_matrix_get_type

```
C       LIS_INT lis_matrix_get_type(LIS_MATRIX A, LIS_INT *matrix_type)
Fortran subroutine lis_matrix_get_type(LIS_MATRIX A, LIS_INTEGER matrix_type,
        LIS_INTEGER ierr)
```

Description
  Get the storage format.

Input

A   The matrix

Output

matrix_type   The storage format

ierr   The return code

6.2.12 lis_matrix_set_blocksize

```
C       LIS_INT lis_matrix_set_blocksize(LIS_MATRIX A, LIS_INT bnr, LIS_INT bnc,
        LIS_INT row[], LIS_INT col[])
Fortran subroutine lis_matrix_set_blocksize(LIS_MATRIX A, LIS_INTEGER bnr,
        LIS_INTEGER bnc, LIS_INTEGER row[], LIS_INTEGER col[], LIS_INTEGER ierr)
```

Description
Assign the block size of the BSR, BSC, and VBR formats.

Input

A	The matrix
bnr	The row block size of the BSR (BSC) format or the number of row blocks of the VBR format
bnc	The olumn block size of the BSR (BSC) format or the number of column blocks of the VBR format
row	The array of the row division information about the VBR format
col	The array of the column division information about the VBR format

Output

ierr	The return code

6.2.13 lis_matrix_convert

```
C       LIS_INT lis_matrix_convert(LIS_MATRIX Ain, LIS_MATRIX Aout)
Fortran subroutine lis_matrix_convert(LIS_MATRIX Ain, LIS_MATRIX Aout,
        LIS_INTEGER ierr)
```

Description
Convert matrix A_{in} into A_{out} of the format specified by lis_matrix_set_type.

Input

Ain	The source matrix

Output

Aout	The destination matrix
ierr	The return code

Note
The storage format of A_{out} is set by `lis_matrix_set_type`. The block size of the BSR, BSC, and VBR formats is set by `lis_matrix_set_blocksize`.

The conversions indicated by 1 in the table below are performed directly, and the others are performed via the indicated formats. The conversions with no indication are performed via the CSR format.

Src \Dst	CSR	CSC	MSR	DIA	ELL	JAD	BSR	BSC	VBR	COO	DNS
CSR		1	1	1	1	1	1	CSC	1	1	1
COO	1	1	1	CSR	CSR	CSR	CSR	CSC	CSR		CSR

6.2.14 lis_matrix_copy

```
C       LIS_INT lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout)
Fortran subroutine lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout,
         LIS_INTEGER ierr)
```

Description
  Copy the values of the matrix elements.

Input

Ain   The source matrix

Output

Aout   The destination matrix

ierr   The return code

6.2.15 lis_matrix_get_diagonal

```
C       LIS_INT lis_matrix_get_diagonal(LIS_MATRIX A, LIS_VECTOR d)
Fortran subroutine lis_matrix_get_diagonal(LIS_MATRIX A, LIS_VECTOR d,
         LIS_INTEGER ierr)
```

Description
  Store the diagonal elements of matrix A to vector d.

Input

A   The matrix

Output

d   The vector that stores the diagonal elements of the matrix

ierr   The return code

6.2.16 lis_matrix_set_csr

```
C        LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT ptr[], LIS_INT index[],
         LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER row(),
         LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description
Associate the arrays in the CSR format with matrix A.

Input

nnz	The number of nonzero elements
ptr, index, value	The arrays in the CSR format
A	The matrix

Output

A	The matrix associated with the arrays

Note
After `lis_matrix_set_csr` is called, the function `lis_matrix_assemble` must be called.

6.2.17 lis_matrix_set_csc

```
C        LIS_INT lis_matrix_set_csc(LIS_INT nnz, LIS_INT ptr[], LIS_INT index[],
         LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_csc(LIS_INTEGER nnz, LIS_INTEGER row(),
         LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description
Associate the arrays in the CSC format with matrix A.

Input

nnz	The number of nonzero elements
ptr, index, value	The arrays in the CSC format
A	The matrix

Output

A	The matrix associated with the arrays

Note
After `lis_matrix_set_csc` is called, the function `lis_matrix_assemble` must be called.

6.2.18 lis_matrix_set_msr

```
C        LIS_INT lis_matrix_set_msr(LIS_INT nnz, LIS_INT ndz, LIS_INT index[],
           LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_msr(LIS_INTEGER nnz, LIS_INTEGER ndz,
           LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Associate the arrays in the MSR format with matrix A.

Input

nnz	The number of nonzero elements
ndz	The number of nonzero elements in the diagonal
index, value	The arrays in the MSR format
A	The matrix

Output

A	The matrix associated with the arrays

Note

After `lis_matrix_set_msr` is called, the function `lis_matrix_assemble` must be called.

6.2.19 lis_matrix_set_dia

```
C        LIS_INT lis_matrix_set_dia(LIS_INT nnd, LIS_INT index[],
           LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_dia(LIS_INTEGER nnd, LIS_INTEGER index(),
           LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Associate the arrays in the DIA format with matrix A.

Input

nnd	The number of nonzero diagonal elements
index, value	The arrays in the DIA format
A	The matrix

Output

A	The matrix associated with the arrays

Note

After `lis_matrix_set_dia` is called, the function `lis_matrix_assemble` must be called.

6.2.20 lis_matrix_set_ell

```
C        LIS_INT lis_matrix_set_ell(LIS_INT maxnzr, LIS_INT index[],
         LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_ell(LIS_INTEGER maxnzr,
         LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A,
         LIS_INTEGER ierr)
```

Description
   Associate the arrays in the ELL format with matrix A.

Input

maxnzr	The maximum number of nonzero elements in each row
index, value	The arrays in the ELL format
A	The matrix

Output

A	The matrix associated with the arrays

Note
   After lis_matrix_set_ell is called, the function lis_matrix_assemble must be called.

6.2.21 lis_matrix_set_jad

```
C        LIS_INT lis_matrix_set_jad(LIS_INT nnz, LIS_INT maxnzr, LIS_INT perm[],
         LIS_INT ptr[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_jad(LIS_INTEGER nnz, LIS_INTEGER maxnzr,
         LIS_INTEGER ptr(), LIS_INTEGER index(), LIS_SCALAR value(),
         LIS_MATRIX A, LIS_INTEGER ierr)
```

Description
   Associate the arrays in the JAD format with matrix A.

Input

nnz	The number of nonzero elements
maxnzr	The maximum number of nonzero elements in each row
perm, ptr, index, value	The arrays in the JAD format
A	The matrix

Output

A	The matrix associated with the arrays

Note
   After lis_matrix_set_jad is called, the function lis_matrix_assemble must be called.

6.2.22 lis_matrix_set_bsr

```
C        LIS_INT lis_matrix_set_bsr(LIS_INT bnr, LIS_INT bnc, LIS_INT bnnz,
         LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_bsr(LIS_INTEGER bnr, LIS_INTEGER bnc,
         LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(),
         LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Associate the arrays in the BSR format with matrix A.

Input

bnr	The row block size
bnc	The column block size
bnnz	The number of nonzero blocks
bptr, bindex, value	The arrays in the BSR format
A	The matrix

Output

A	The matrix associated with the arrays

Note

After `lis_matrix_set_bsr` is called, the function `lis_matrix_assemble` must be called.

6.2.23 lis_matrix_set_bsc

```
C        LIS_INT lis_matrix_set_bsc(LIS_INT bnr, LIS_INT bnc, LIS_INT bnnz,
         LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_bsc(LIS_INTEGER bnr, LIS_INTEGER bnc,
         LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(),
         LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Associate the arrays in the BSC format with matrix A.

Input

bnr	The row block size
bnc	The column block size
bnnz	The number of nonzero blocks
bptr, bindex, value	The arrays in the BSC format
A	The matrix

Output

A	The matrix associated with the arrays

Note

After `lis_matrix_set_bsc` is called, the function `lis_matrix_assemble` must be called.

6.2.24 lis_matrix_set_vbr

```
C       LIS_INT lis_matrix_set_vbr(LIS_INT nnz, LIS_INT nr, LIS_INT nc,
         LIS_INT bnnz, LIS_INT row[], LIS_INT col[], LIS_INT ptr[],
         LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[],
         LIS_MATRIX A)
Fortran subroutine lis_matrix_set_vbr(LIS_INTEGER nnz, LIS_INTEGER nr,
         LIS_INTEGER nc, LIS_INTEGER bnnz, LIS_INTEGER row(),
         LIS_INTEGER col(), LIS_INTEGER ptr(), LIS_INTEGER bptr(),
         LIS_INTEGER bindex(), LIS_SCALAR value(), LIS_MATRIX A,
         LIS_INTEGER ierr)
```

Description

Associate the arrays in the VBR format with matrix A.

Input

nnz	The number of nonzero elements
nr	The number of row blocks
nc	The number of column blocks
bnnz	The number of nonzero blocks
row, col, ptr, bptr, bindex, value	The arrays in the VBR format
A	The matrix

Output

A	The matrix associated with the arrays

Note

After `lis_matrix_set_vbr` is called, the function `lis_matrix_assemble` must be called.

6.2.25 lis_matrix_set_coo

```
C       LIS_INT lis_matrix_set_coo(LIS_INT nnz, LIS_INT row[], LIS_INT col[],
         LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_coo(LIS_INTEGER nnz, LIS_INTEGER row(),
         LIS_INTEGER col(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

Description

Associate the arrays in the COO format with matrix A.

Input

nnz	The number of nonzero elements
row, col, value	The arrays in the COO format
A	The matrix

Output

A	The matrix associated with the arrays

Note

After `lis_matrix_set_coo` is called, the function `lis_matrix_assemble` must be called.

6.2.26 lis_matrix_set_dns

```
C        LIS_INT lis_matrix_set_dns(LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_dns(LIS_SCALAR value(), LIS_MATRIX A,
         LIS_INTEGER ierr)
```

Description
  Associate the array in the DNS format with matrix A.

Input

  value   The array in the DNS format

  A   The matrix

Output

  A   The matrix associated with the array

Note
  After `lis_matrix_set_dns` is called, the function `lis_matrix_assemble` must be called.

6.3 Operating Vectors and Matrices

6.3.1 lis_vector_scale

```
C       LIS_INT lis_vector_scale(LIS_SCALAR alpha, LIS_VECTOR x)
Fortran subroutine lis_vector_scale(LIS_SCALAR alpha, LIS_VECTOR x,
        LIS_INTEGER ierr)
```

Description

Multiply vector x by scalar α.

Input

alpha	The scalar value
x	The vector

Output

x	αx (vector x is overwritten)
ierr	The return code

6.3.2 lis_vector_dot

```
C       LIS_INT lis_vector_dot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR *val)
Fortran subroutine lis_vector_dot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR val,
        LIS_INTEGER ierr)
```

Description

Calculate the inner product $x^T y$.

Input

x	The vector
y	The vector

Output

val	The inner product
ierr	The return code

6.3.3 lis_vector_nrm1

```
C        LIS_INT lis_vector_nrm1(LIS_VECTOR x, LIS_SCALAR *val)
Fortran subroutine lis_vector_nrm1(LIS_VECTOR x, LIS_SCALAR val, LIS_INTEGER ierr)
```

Description
Calculate the 1-norm of vector x.

Input

x	The vector

Output

val	The 1-norm of the vector
ierr	The return code

6.3.4 lis_vector_nrm2

```
C        LIS_INT lis_vector_nrm2(LIS_VECTOR x, LIS_SCALAR *val)
Fortran subroutine lis_vector_nrm2(LIS_VECTOR x, LIS_SCALAR val, LIS_INTEGER ierr)
```

Description
Calculate the 2-norm of vector x.

Input

x	The vector

Output

val	The 2-norm of the vector
ierr	The return code

6.3.5 lis_vector_nrmi

```
C        LIS_INT lis_vector_nrmi(LIS_VECTOR x, LIS_SCALAR *val)
Fortran subroutine lis_vector_nrmi(LIS_VECTOR x, LIS_SCALAR val, LIS_INTEGER ierr)
```

Description
Calculate the infinity norm of vector x.

Input

x	The vector

Output

val	The infinity norm of the vector
ierr	The return code

6.3.6 lis_vector_axpy

```
C        LIS_INT lis_vector_axpy(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_vector_axpy(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y,
         LIS_INTEGER ierr)
```

Description

Calculate the sum of the vectors $y = \alpha x + y$.

Input

alpha	The scalar value
x, y	The vectors

Output

y	$\alpha x + y$ (vector y is overwritten)
ierr	The return code

6.3.7 lis_vector_xpay

```
C        LIS_INT lis_vector_xpay(LIS_VECTOR x, LIS_SCALAR alpha, LIS_VECTOR y)
Fortran subroutine lis_vector_xpay(LIS_VECTOR x, LIS_SCALAR alpha, LIS_VECTOR y,
         LIS_INTEGER ierr)
```

Description

Calculate the sum of the vectors $y = x + \alpha y$.

Input

alpha	The scalar value
x, y	The vectors

Output

y	$x + \alpha y$ (vector y is overwritten)
ierr	The return code

6.3.8 lis_vector_axpyz

```
C         LIS_INT lis_vector_axpyz(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y,
          LIS_VECTOR z)
Fortran subroutine lis_vector_axpyz(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y,
          LIS_VECTOR z, LIS_INTEGER ierr)
```

Description

Calculate the sum of the vectors $z = \alpha x + y$.

Input

alpha	The scalar value
x, y	The vectors

Output

z	$x + \alpha y$
ierr	The return code

6.3.9 lis_matrix_scaling

```
C         LIS_INT lis_matrix_scaling(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR d,
          LIS_INT action)
Fortran subroutine lis_matrix_scaling(LIS_MATRIX A, LIS_VECTOR b,
          LIS_VECTOR d, LIS_INTEGER action, LIS_INTEGER ierr)
```

Description

Scale matrix A.

Input

A	The matrix
b	The vector
action	LIS_SCALE_JACOBI : Jacobi scaling $D^{-1}Ax = D^{-1}b$, where D represents the diagonal of $A = (a_{ij})$
	LIS_SCALE_SYMM_DIAG : Diagonal scaling $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$, where $D^{-1/2}$ represents a diagonal matrix with $1/\sqrt{a_{ii}}$ as the diagonal

Output

d	The vector that stores the diagonal elements of D^{-1} or $D^{-1/2}$
ierr	The return code

6.3.10 lis_matvec

```
C       LIS_INT lis_matvec(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_matvec(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y,
        LIS_INTEGER ierr)
```

Description
  Calculate the matrix vector product $y = Ax$.

Input

A	The matrix
x	The vector

Output

y	Ax
ierr	The return code

6.3.11 lis_matvect

```
C       LIS_INT lis_matvect(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_matvect(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y,
        LIS_INTEGER ierr)
```

Description
  Calculate the transposed matrix vector product $y = A^T x$.

Input

A	The matrix
x	The vector

Output

y	$A^T x$
ierr	The return code

6.4 Solving Linear Equations

6.4.1 lis_solver_create

```
C       LIS_INT lis_solver_create(LIS_SOLVER *solver)
Fortran subroutine lis_solver_create(LIS_SOLVER solver, LIS_INTEGER ierr)
```

Description
  Create the solver.

Input

  None

Output

  solver   The solver

  ierr   The return code

Note
  solver has the information on the solver, the preconditioner, etc.

6.4.2 lis_solver_destroy

```
C       LIS_INT lis_solver_destroy(LIS_SOLVER solver)
Fortran subroutine lis_solver_destroy(LIS_SOLVER solver, LIS_INTEGER ierr)
```

Description
  Destroy the solver.

Input

  solver   The solver to be destroyed

Output

  ierr   The return code

6.4.3 lis_solver_set_option

```
C       LIS_INT lis_solver_set_option(char *text, LIS_SOLVER solver)
Fortran subroutine lis_solver_set_option(character text, LIS_SOLVER solver,
        LIS_INTEGER ierr)
```

Description

Set the options for the solver.

Input

text   The command line options

Output

solver   The solver

ierr   The return code

Note

The table below shows the available command line options, where `-i {cg|1}` means `-i cg` or `-i 1` and `-maxiter [1000]` indicates that `-maxiter` defaults to $1,000$.

Options for Linear Solvers (Default: `-i bicg`)

Solver	Option	Auxiliary Options	
CG	-i {cg|1}		
BiCG	-i {bicg|2}		
CGS	-i {cgs|3}		
BiCGSTAB	-i {bicgstab|4}		
BiCGSTAB(l)	-i {bicgstabl|5}	-ell [2]	The degree l
GPBiCG	-i {gpbicg|6}		
TFQMR	-i {tfqmr|7}		
Orthomin(m)	-i {orthomin|8}	-restart [40]	The restart value m
GMRES(m)	-i {gmres|9}	-restart [40]	The restart value m
Jacobi	-i {jacobi|10}		
Gauss-Seidel	-i {gs|11}		
SOR	-i {sor|12}	-omega [1.9]	The relaxation coefficient ω $(0 < \omega < 2)$
BiCGSafe	-i {bicgsafe|13}		
CR	-i {cr|14}		
BiCR	-i {bicr|15}		
CRS	-i {crs|16}		
BiCRSTAB	-i {bicrstab|17}		
GPBiCR	-i {gpbicr|18}		
BiCRSafe	-i {bicrsafe|19}		
FGMRES(m)	-i {fgmres|20}	-restart [40]	The restart value m
IDR(s)	-i {idrs|21}	-irestart [2]	The restart value s
MINRES	-i {minres|22}		

Options for Preconditioners (Default: `-p none`)

Preconditioner	Option	Auxiliary Options	
None	`-p {none|0}`		
Jacobi	`-p {jacobi|1}`		
ILU(k)	`-p {ilu|2}`	`-ilu_fill [0]`	The fill level k
SSOR	`-p {ssor|3}`	`-ssor_w [1.0]`	The relaxation coefficient ω $(0 < \omega < 2)$
Hybrid	`-p {hybrid|4}`	`-hybrid_i [sor]`	The linear solver
		`-hybrid_maxiter [25]`	The maximum number of iterations
		`-hybrid_tol [1.0e-3]`	The convergence criterion
		`-hybrid_w [1.5]`	The relaxation coefficient ω of the SOR $(0 < \omega < 2)$
		`-hybrid_ell [2]`	The degree l of the BiCGSTAB(l)
		`-hybrid_restart [40]`	The restart values of the GMRES and Orthomin
I+S	`-p {is|5}`	`-is_alpha [1.0]`	The parameter α of $I + \alpha S^{(m)}$
		`-is_m [3]`	The parameter m of $I + \alpha S^{(m)}$
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Select the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \leq \theta^2 |a_{ii}||a_{jj}|$ (symmetric or unsymmetric)
Crout ILU	`-p {iluc|8}`	`-iluc_drop [0.05]`	The drop criterion
		`-iluc_rate [5.0]`	The ratio of the maximum fill-in
ILUT	`-p {ilut|9}`	`-ilut_drop [0.05]`	The drop criterion
		`-ilut_rate [5.0]`	The ratio of the maximum fill-in
Additive Schwarz	`-adds true`	`-adds_iter [1]`	The number of iterations

Other Options

Option	
`-maxiter [1000]`	The maximum number of iterations
`-tol [1.0e-12]`	The convergence criterion
`-print [0]`	The display of the residual
	`-print {none|0}` None
	`-print {mem|1}` Save the residual history
	`-print {out|2}` Display the residual history
	`-print {all|3}` Save the residual history and display it on the screen
`-scale [0]`	The scaling
	(The result will overwrite the original matrix and vectors)
	`-scale {none|0}` No scaling
	`-scale {jacobi|1}` The Jacobi scaling $D^{-1}Ax = D^{-1}b$
	(D represents the diagonal of $A = (a_{ij})$)
	`-scale {symm_diag|2}` The diagonal scaling $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$
	($D^{-1/2}$ represents the diagonal matrix with $1/\sqrt{a_{ii}}$
	as the diagonal)
`-initx_zeros [true]`	The behavior of the initial vector x_0
	`-initx_zeros {false|0}` Given values
	`-initx_zeros {true|1}` All values are set to 0
`-omp_num_threads [t]`	The number of threads
	(`t` represents the maximum number of threads)
`-storage [0]`	The matrix storage format
`-storage_block [2]`	The block size of the BSR and BSC formats
`-f [0]`	The precision of the linear solver
	`-f {double|0}` Double precision
	`-f {quad|1}` Quadruple precision

6.4.4 lis_solver_set_optionC

```
C        LIS_INT lis_solver_set_optionC(LIS_SOLVER solver)
Fortran subroutine lis_solver_set_optionC(LIS_SOLVER solver, LIS_INTEGER ierr)
```

Description

Set the options for the solver on the command line.

Input

None

Output

solver	The solver
ierr	The return code

6.4.5 lis_solve

```
C        LIS_INT lis_solve(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
          LIS_SOLVER solver)
Fortran subroutine lis_solve(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
          LIS_SOLVER solver, LIS_INTEGER ierr)
```

Description

Solve the linear equation $Ax = b$ with the specified solver.

Input

A	The coefficient matrix
b	The right-hand side vector
x	The initial vector
solver	The solver

Output

x	The solution
solver	The number of iterations, the execution time, etc.
ierr	The return code

6.4.6 lis_solve_kernel

```
C        LIS_INT lis_solve_kernel(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
          LIS_SOLVER solver, LIS_PRECON, precon)
Fortran subroutine lis_solve_kernel(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
          LIS_SOLVER solver, LIS_PRECON precon, LIS_INTEGER ierr)
```

Description

Solve the linear equation $Ax = b$ with the specified solver and the predefined preconditioner.

Input

A	The coefficient matrix
b	The right-hand side vector
x	The initial vector
solver	The solver
precon	The preconditioner

Output

x	The solution
solver	The number of iterations, the execution time, etc.
ierr	The return code

Note

See lis-($VERSION)/src/esolver/lis_esolver_ii.c, which computes the smallest eigenvalue by calling lis_solve_kernel multiple times, for example.

6.4.7 lis_solver_get_status

```
C        LIS_INT lis_solver_get_status(LIS_SOLVER solver, LIS_INT *status)
Fortran subroutine lis_solver_get_status(LIS_SOLVER solver, LIS_INTEGER status,
         LIS_INTEGER ierr)
```

Description
  Get the status from the solver.

Input

  solver   The solver

Output

  status   The status

  ierr   The return code

6.4.8 lis_solver_get_iters

```
C        LIS_INT lis_solver_get_iters(LIS_SOLVER solver, LIS_INT *iters)
Fortran subroutine lis_solver_get_iters(LIS_SOLVER solver, LIS_INTEGER iters,
         LIS_INTEGER ierr)
```

Description
  Get the number of iterations from the solver.

Input

  solver   The solver

Output

  iters   The number of iterations

  ierr   The return code

6.4.9 lis_solver_get_itersex

```
C       LIS_INT lis_solver_get_itersex(LIS_SOLVER solver, LIS_INT *iters,
          LIS_INT *iters_double, LIS_INT *iters_quad)
Fortran subroutine lis_solver_get_itersex(LIS_SOLVER solver, LIS_INTEGER iters,
          LIS_INTEGER iters_double, LIS_INTEGER iters_quad, LIS_INTEGER ierr)
```

Description

Get the detailed information on the number of iterations from the solver.

Input

solver	The solver

Output

iters	The number of iterations
iters_double	The number of double precision iterations
iters_quad	The number of quadruple precision iterations
ierr	The return code

6.4.10 lis_solver_get_time

```
C       LIS_INT lis_solver_get_time(LIS_SOLVER solver, double *times)
Fortran subroutine lis_solver_get_time(LIS_SOLVER solver, real*8 times,
          LIS_INTEGER ierr)
```

Description

Get the execution time from the solver.

Input

solver	The solver

Output

times	The time in seconds of the execution
ierr	The return code

6.4.11 lis_solver_get_timeex

```
C       LIS_INT lis_solver_get_timeex(LIS_SOLVER solver, double *times,
          double *itimes, double *ptimes, double *p_c_times, double *p_i_times)
Fortran subroutine lis_solver_get_timeex(LIS_SOLVER solver, real*8 times,
          real*8 itimes, real*8 ptimes, real*8 p_c_times, real*8 p_i_times,
          LIS_INTEGER ierr)
```

Description
  Get the detailed information on the execution time from the solver.

Input

solver	The solver

Output

times	The total time in seconds
itimes	The time in seconds of the iterations
ptimes	The time in seconds of the preconditioning
p_c_times	The time in seconds of the creation of the preconditioner
p_i_times	The time in seconds of the iterations in the preconditioner
ierr	The return code

6.4.12 lis_solver_get_residualnorm

```
C       LIS_INT lis_solver_get_residualnorm(LIS_SOLVER solver, LIS_REAL *residual)
Fortran subroutine lis_solver_get_residualnorm(LIS_SOLVER solver,
          LIS_REAL residual, LIS_INTEGER ierr)
```

Description
  Calculate the relative residual norm $||b - Ax||_2/||b||_2$ from the solution x.

Input

solver	The solver

Output

residual	The relative residual norm $||b - Ax||_2/||b||_2$
ierr	The return code

6.4.13   lis_solver_get_rhistory

```
C       LIS_INT lis_solver_get_rhistory(LIS_SOLVER solver, LIS_VECTOR v)
Fortran subroutine lis_solver_get_rhistory(LIS_SOLVER solver, LIS_VECTOR v,
         LIS_INTEGER ierr)
```

Description

  Store the residual norm history of the solver.

Input

  solver   The solver

Output

  v   The vector

  ierr   The return code

Note

  Vector v must be created in advance with the function `lis_vector_create`. When vector v is shorter than the residual history, it stores the residual history in order to vector v.

6.4.14 lis_solver_get_solver

```
C        LIS_INT lis_solver_get_solver(LIS_SOLVER solver, LIS_INT *nsol)
Fortran subroutine lis_solver_get_solver(LIS_SOLVER solver, LIS_INTEGER nsol,
         LIS_INTEGER ierr)
```

Description
  Get the solver number from the solver.

Input

  solver   The solver

Output

  nsol   The solver number

  ierr   The return code

6.4.15 lis_solver_get_precon

```
C        LIS_INT lis_solver_get_precon(LIS_SOLVER solver, LIS_INT *precon_type)
Fortran subroutine lis_solver_get_precon(LIS_SOLVER solver, LIS_INTEGER precon_type,
         LIS_INTEGER ierr)
```

Description
  Get the preconditioner number from the solver.

Input

  solver   The solver

Output

  precon_type   The preconditioner number

  ierr   The return code

6.4.16 lis_solver_get_solvername

```
C       LIS_INT lis_solver_get_solvername(LIS_INT nsol, char *name)
Fortran subroutine lis_solver_get_solvername(LIS_INTEGER nsol, character name,
        LIS_INTEGER ierr)
```

Description

Get the solver name from the solver number.

Input

nsol   The solver number

Output

name   The solver name

ierr   The return code

6.4.17 lis_solver_get_preconname

```
C       LIS_INT lis_solver_get_preconname(LIS_INT precon_type, char *name)
Fortran subroutine lis_solver_get_preconname(LIS_INTEGER precon_type,
        character name, LIS_INTEGER ierr)
```

Description

Get the preconditioner name from the preconditioner number.

Input

precon_type   The preconditioner number

Output

name   The preconditioner name

ierr   The return code

6.5 Solving Eigenvalue Problems

6.5.1 lis_esolver_create

```
C       LIS_INT lis_esolver_create(LIS_ESOLVER *esolver)
Fortran subroutine lis_esolver_create(LIS_ESOLVER esolver, LIS_INTEGER ierr)
```

Description
 Create the eigensolver.

Input

 None

Output

esolver	The eigensolver
ierr	The return code

Note
 esolver has the information on the eigensolver, the preconditioner, etc.

6.5.2 lis_esolver_destroy

```
C       LIS_INT lis_esolver_destroy(LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_destroy(LIS_ESOLVER esolver, LIS_INTEGER ierr)
```

Description
 Destroy the eigensolver.

Input

esolver	The eigensolver to be destoyed

Output

ierr	The return code

6.5.3 lis_esolver_set_option

```
C        LIS_INT lis_esolver_set_option(char *text, LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver,
         LIS_INTEGER ierr)
```

Description

   Set the options for the eigensolver.

Input

   text   The command line options

Output

   esolver   The eigensolver

   ierr   The return code

Note

   The table below shows the available command line options, where `-e {pi|1}` means `-e pi` or `-e 1` and `-emaxiter [1000]` indicates that `-emaxiter` defaults to $1,000$.

Options for Eigensolvers (Default: `-e pi`)

Eigensolver	Option	Auxiliary Options	
Power	`-e {pi|1}`		
Inverse	`-e {ii|2}`	`-i [cg]`	The linear solver
Approximate Inverse	`-e {aii|3}`	`-i [cg]`	The linear solver
Rayleigh Quotient	`-e {rqi|4}`	`-i [cg]`	The linear solver
Subspace	`-e {si|5}`	`-ss [2]`	The size of the subspace
		`-m [0]`	The mode number
Lanczos	`-e {li|6}`	`-ss [2]`	The size of the subspace
		`-m [0]`	The mode number
CG	`-e {cg|7}`		
CR	`-e {cr|8}`		

<div align="center">

Options for Preconditioners (Default: `-p none`)

</div>

Preconditioner	Option	Auxiliary Options	
None	`-p {none|0}`		
Jacobi	`-p {jacobi|1}`		
ILU(k)	`-p {ilu|2}`	`-ilu_fill [0]`	The fill level k
SSOR	`-p {ssor|3}`	`-ssor_w [1.0]`	The relaxation coefficient ω $(0 < \omega < 2)$
Hybrid	`-p {hybrid|4}`	`-hybrid_i [sor]`	The linear solver
		`-hybrid_maxiter [25]`	The maximum number of iterations
		`-hybrid_tol [1.0e-3]`	The convergence criterion
		`-hybrid_w [1.5]`	The relaxation coefficient ω of the SOR $(0 < \omega < 2)$
		`-hybrid_ell [2]`	The degree l of the BiCGSTAB(l)
		`-hybrid_restart [40]`	The restart values of the GMRES and Orthomin
I+S	`-p {is|5}`	`-is_alpha [1.0]`	The parameter α of $I + \alpha S^{(m)}$
		`-is_m [3]`	The parameter m of $I + \alpha S^{(m)}$
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Select the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \leq \theta^2 \|a_{ii}\|\|a_{jj}\|$ (symmetric or unsymmetric)
Crout ILU	`-p {iluc|8}`	`-iluc_drop [0.05]`	The drop criterion
		`-iluc_rate [5.0]`	The ratio of the maximum fill-in
ILUT	`-p {ilut|9}`	`-ilut_drop [0.05]`	The drop criterion
		`-ilut_rate [5.0]`	The ratio of the maximum fill-in
Additive Schwarz	`-adds true`	`-adds_iter [1]`	The number of iterations

<div align="center">

Other Options

</div>

Option	
`-emaxiter [1000]`	The maximum number of iterations
`-etol [1.0e-12]`	The convergence criterion
`-eprint [0]`	The display of the residual
	`-eprint {none|0}` None
	`-eprint {mem|1}` Save the residual history
	`-eprint {out|2}` Display the residual history
	`-eprint {all|3}` Save the residual history and display it on the screen
`-ie [ii]`	The inner eigensolver used in the Lanczos and Subspace
	`-ie {pi|1}` The Power (the Subspace only)
	`-ie {ii|2}` The Inverse
	`-ie {aii|3}` The Approximate Inverse
	`-ie {rqi|4}` The Rayleigh Quotient
`-shift [0.0]`	The amount of the shift
`-initx_ones [true]`	The behavior of the initial vector x_0
	`-initx_ones {false|0}` Given values
	`-initx_ones {true|1}` All values are set to 1
`-omp_num_threads [t]`	The number of threads (`t` represents the maximum number of threads)
`-estorage [0]`	The matrix storage format
`-estorage_block [2]`	The block size of the BSR and BSC formats
`-ef [0]`	The precision of the eigensolver
	`-ef {double|0}` Double precision
	`-ef {quad|1}` Quadruple precision

6.5.4   lis_esolver_set_optionC

```
C        LIS_INT lis_esolver_set_optionC(LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, LIS_INTEGER ierr)
```

Description

   Set the options for the eigensolver on the command line.

Input

   None

Output

esolver	The eigensolver
ierr	The return code

6.5.5   lis_esolve

```
C        LIS_INT lis_esolve(LIS_MATRIX A, LIS_VECTOR x,
         LIS_REAL evalue, LIS_ESOLVER esolver)
Fortran subroutine lis_esolve(LIS_MATRIX A, LIS_VECTOR x,
         LIS_REAL evalue, LIS_ESOLVER esolver, LIS_INTEGER ierr)
```

Description

   Solve the eigenvalue problem $Ax = \lambda x$ with the specified eigensolver.

Input

A	The matrix
x	The initial vector
esolver	The eigensolver

Output

evalue	The eigenvalue of the mode specified by option -m [0]
x	The associated eigenvector
esolver	The number of iterations, the execution time, etc.
ierr	The return code

6.5.6 lis_esolver_get_status

```
C       LIS_INT lis_esolver_get_status(LIS_ESOLVER esolver, LIS_INT *status)
Fortran subroutine lis_esolver_get_status(LIS_ESOLVER esolver, LIS_INTEGER status,
        LIS_INTEGER ierr)
```

Description

Get the status from the eigensolver.

Input

esolver   The eigensolver

Output

status   The status

ierr   The return code

6.5.7 lis_esolver_get_iters

```
C       LIS_INT lis_esolver_get_iters(LIS_ESOLVER esolver, LIS_INT *iters)
Fortran subroutine lis_esolver_get_iters(LIS_ESOLVER esolver, LIS_INTEGER iters,
        LIS_INTEGER ierr)
```

Description

Get the number of iterations from the eigensolver.

Input

esolver   The eigensolver

Output

iters   The number of iterations

ierr   The return code

6.5.8 lis_esolver_get_itersex

```
C       LIS_INT lis_esolver_get_itersex(LIS_ESOLVER esolver, LIS_INT *iters,
          LIS_INT *iters_double, LIS_INT *iters_quad)
Fortran subroutine lis_esolver_get_itersex(LIS_ESOLVER esolver, LIS_INTEGER iters,
          LIS_INTEGER iters_double, LIS_INTEGER iters_quad, LIS_INTEGER ierr)
```

Description

Get the detailed information on the number of iterations from the eigensolver.

Input

esolver	The eigensolver

Output

iters	The number of iterations
iters_double	The number of double precision iterations
iters_quad	The number of quadruple precision iterations
ierr	The return code

6.5.9 lis_esolver_get_time

```
C       LIS_INT lis_esolver_get_time(LIS_ESOLVER esolver, double *times)
Fortran subroutine lis_esolver_get_time(LIS_ESOLVER esolver, real*8 times,
          LIS_INTEGER ierr)
```

Description

Get the execution time from the eigensolver.

Input

esolver	The eigensolver

Output

times	The time in seconds of the execution
ierr	The return code

6.5.10 lis_esolver_get_timeex

```
C       LIS_INT lis_esolver_get_timeex(LIS_ESOLVER esolver, double *times,
        double *itimes, double *ptimes, double *p_c_times, double *p_i_times)
Fortran subroutine lis_esolver_get_timeex(LIS_ESOLVER esolver, real*8 times,
        real*8 itimes, real*8 ptimes, real*8 p_c_times, real*8 p_i_times,
        LIS_INTEGER ierr)
```

Description

Get the detailed information on the execution time from the eigensolver.

Input

esolver	The eigensolver

Output

times	The total time in seconds
itimes	The time in seconds of the iterations
ptimes	The time in seconds of the preconditioning
p_c_times	The time in seconds of the creation of the preconditioner
p_i_times	The time in seconds of the iterations in the preconditioner
ierr	The return code

6.5.11 lis_esolver_get_residualnorm

```
C       LIS_INT lis_esolver_get_residualnorm(LIS_ESOLVER esolver,
        LIS_REAL *residual)
Fortran subroutine lis_esolver_get_residualnorm(LIS_ESOLVER esolver,
        LIS_REAL residual, LIS_INTEGER ierr)
```

Description

Calculate the relative residual norm $||\lambda x - Ax||_2/\lambda$ from the eigenvector x.

Input

esolver	The eigensolver

Output

residual	The relative residual norm $||\lambda x - Ax||_2/\lambda$
ierr	The return code

6.5.12 lis_esolver_get_rhistory

```
C       LIS_INT lis_esolver_get_rhistory(LIS_ESOLVER esolver, LIS_VECTOR v)
Fortran subroutine lis_esolver_get_rhistory(LIS_ESOLVER esolver, LIS_VECTOR v,
        LIS_INTEGER ierr)
```

Description

Store the residual norm history of the eigensolver.

Input

esolver   The eigensolver

Output

v   The vector

ierr   The return code

Note

Vector v must be created in advance with the function `lis_vector_create`. When vector v is shorter than the residual history, it stores the residual history in order to vector v.

6.5.13 lis_esolver_get_evalues

```
C       LIS_INT lis_esolver_get_evalues(LIS_ESOLVER esolver, LIS_VECTOR v)
Fortran subroutine lis_esolver_get_evalues(LIS_ESOLVER esolver,
        LIS_VECTOR v, LIS_INTEGER ierr)
```

Description
 Store the eigenvalues in vector v.

Input

 esolver   The eigensolver

Output

 v   The vector which stores the eigenvalues

 ierr   The return code

Note
 Vector v must be created in advance with the function `lis_vector_create`.

6.5.14 lis_esolver_get_evectors

```
C       LIS_INT lis_esolver_get_evectors(LIS_ESOLVER esolver, LIS_MATRIX A)
Fortran subroutine lis_esolver_get_evectors(LIS_ESOLVER esolver,
        LIS_MATRIX A, LIS_INTEGER ierr)
```

Description
 Store the eigenvectors in matrix A.

Input

 esolver   The eigensolver

Output

 A   The matrix in the CSR format which stores the eigenvectors

 ierr   The return code

Note
 Matrix A must be created in advance with the function `lis_matrix_create`.

6.5.15 lis_esolver_get_esolver

```
C         LIS_INT lis_esolver_get_esolver(LIS_ESOLVER esolver, LIS_INT *nesol)
Fortran subroutine lis_esolver_get_esolver(LIS_ESOLVER esolver,
          LIS_INTEGER nesol, LIS_INTEGER ierr)
```

Description
  Get the eigensolver number from the eigensolver.

Input

esolver   The eigensolver

Output

nesol   The eigensolver number

ierr   The return code

6.5.16 lis_esolver_get_esolvername

```
C         LIS_INT lis_esolver_get_esolvername(LIS_INT nesol, char *ename)
Fortran subroutine lis_esolver_get_esolvername(LIS_INTEGER nesol, character ename,
          LIS_INTEGER ierr)
```

Description
  Get the eigensolver name from the eigensolver number.

Input

nesol   The eigensolver number

Output

name   The eigensolver name

ierr   The return code

6.6 Operating External Files

6.6.1 lis_input

```
C       LIS_INT lis_input(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, char *filename)
Fortran subroutine lis_input(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
        character filename, LIS_INTEGER ierr)
```

Description
    Read the matrix and vector data from the external file.

Input

filename	The source file

Output

A	The matrix in the specified storage format
b	The right-hand side vector
x	The solution
ierr	The return code

Note
    The following file formats are supported:

- The extended Matrix Market format (extended to allow vector data)

- The Harwell-Boeing format

6.6.2 lis_input_vector

```
C       LIS_INT lis_input_vector(LIS_VECTOR v, char *filename)
Fortran subroutine lis_input_vector(LIS_VECTOR v, character filename,
        LIS_INTEGER ierr)
```

Description
    Read the vector data from the external file.

Input

filename	The source file

Output

v	The vector
ierr	The return code

Note
    The following file formats are supported:

- The PLAIN format

- The extended Matrix Market format (extended to allow vector data)

6.6.3 lis_input_matrix

```
C       LIS_INT lis_input_matrix(LIS_MATRIX A, char *filename)
Fortran subroutine lis_input_matrix(LIS_MATRIX A, character filename,
        LIS_INTEGER ierr)
```

Description
   Read the matrix data from the external file.

Input

filename	The source file

Output

A	The matrix in the specified storage format
x	The solution
ierr	The return code

Note
   The following file formats are supported:

   • The Matrix Market format

   • The Harwell-Boeing format

6.6.4 lis_output

```
C       LIS_INT lis_output(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
        LIS_INT format, char *filename)
Fortran subroutine lis_output(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
        LIS_INTEGER format, character filename, LIS_INTEGER ierr)
```

Description
   Write the matrix and vector data into the external file.

Input

A	The matrix
b	The right-hand side vector (If no vector is written to the external file, then NULL must be input.)
x	The solution (If no vector is written to the external file, then NULL must be input.)
format	The file format
	LIS_FMT_MM The Matrix Market format
filename	The destination file

Output

ierr	The return code

6.6.5 lis_output_vector

```
C       LIS_INT lis_output_vector(LIS_VECTOR v, LIS_INT format, char *filename)
Fortran subroutine lis_output_vector(LIS_VECTOR v, LIS_INTEGER format,
        character filename, LIS_INTEGER ierr)
```

Description
   Write the vector data into the external file.

Input

v	The vector
format	The file format

	LIS_FMT_PLAIN	The PLAIN format
	LIS_FMT_MM	The Matrix Market format

filename	The destination file

Output

ierr	The return code

6.6.6 lis_output_matrix

```
C       LIS_INT lis_output_matrix(LIS_MATRIX A, LIS_INT format, char *filename)
Fortran subroutine lis_output_matrix(LIS_MATRIX A, LIS_INTEGER format,
        character filename, LIS_INTEGER ierr)
```

Description
   Write the matrix data into the external file.

Input

A	The matrix
format	The file format

	LIS_FMT_MM	The Matrix Market format

filename	The destination file

Output

ierr	The return code

6.7   Other Functions

6.7.1   lis_initialize

```
C       LIS_INT lis_initialize(LIS_INT* argc, char** argv[])
Fortran subroutine lis_initialize(LIS_INTEGER ierr)
```

Description

Initialize the execution environment.

Input

argc   The number of command line arguments

argv   The command line argument

Output

ierr   The return code

6.7.2   lis_finalize

```
C       LIS_INT lis_finalize()
Fortran subroutine lis_finalize(LIS_INTEGER ierr)
```

Description

Finalize the execution environment.

Input

None

Output

ierr   The return code

6.7.3 lis_wtime

```
C       double lis_wtime()
Fortran function lis_wtime()
```

Description
Measure the elapsed time.

Input

None

Output
The elapsed time in seconds from the given point is returned as the double precision number.

Note
To measure the processing time, call `lis_wtime` to get the starting time, call it again to get the ending time, and calculate the difference.

6.7.4 CHKERR

```
C       void CHKERR(LIS_INT ierr)
Fortran subroutine CHKERR(LIS_INTEGER ierr)
```

Description
Check the value of the return code.

Input

`ierr`	The return code

Output

None

Note
If the value of the return code is not 0, it calls `lis_finalize` and terminates the program.

References

[1] A. Nishida. Experience in Developing an Open Source Scalable Software Infrastructure in Japan. Lecture Notes in Computer Science 6017, pp. 87-98, Springer, 2010.

[2] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. Journal of Research of the National Bureau of Standards, Vol. 49, No. 6, pp. 409–436, 1952.

[3] C. Lanczos. Solution of Linear Equations by Minimized Iterations. Journal of Research of the National Bureau of Standards, Vol. 49, No. 1, pp. 33–53, 1952.

[4] R. Fletcher. Conjugate Gradient Methods for Indefinite Systems. Lecture Notes in Mathematics 506, pp. 73–89, Springer, 1976.

[5] T. Sogabe, M. Sugihara, and S. Zhang. An Extension of the Conjugate Residual Method to Nonsymmetric Linear Systems. Journal of Computational and Applied Mathematics, Vol. 226, No. 1, pp. 103–113, 2009.

[6] P. Sonneveld. CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, Vol. 10, No. 1, pp. 36–52, 1989.

[7] K. Abe, T. Sogabe, S. Fujino, and S. Zhang. A Product-Type Krylov Subspace Method Based on Conjugate Residual Method for Nonsymmetric Coefficient Matrices (in Japanese). IPSJ Transactions on Advanced Computing Systems, Vol. 48, No. SIG8(ACS18), pp. 11–21, 2007.

[8] H. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, Vol. 13, No. 2, pp. 631–644, 1992.

[9] S. Zhang. Generalized Product-Type Methods Preconditionings Based on Bi-CG for Solving Nonsymmetric Linear Systems. SIAM Journal on Scientific Computing, Vol. 18, No. 2, pp. 537–551, 1997.

[10] S. Fujino, M. Fujiwara, and M. Yoshida. A Proposal of Preconditioned BiCGSafe Method with Safe Convergence. Proceedings of The 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, CD-ROM, 2005.

[11] S. Fujino and Y. Onoue. Estimation of BiCRSafe Method Based on Residual of BiCR Method (in Japanese). IPSJ SIG Technical Report, 2007-HPC-111, pp. 25–30, 2007.

[12] G. L. G. Sleijpen, H. A. van der Vorst, and D. R. Fokkema. BiCGstab(l) and Other Hybrid Bi-CG Methods. Numerical Algorithms, Vol. 7, No. 1, pp. 75–109, 1994.

[13] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. SIAM Journal on Scientific Computing, Vol. 14, No. 2, pp. 470–482, 1993.

[14] K. R. Biermann. Eine unveröffentlichte Jugendarbeit C. G. J. Jacobi über wiederholte Funktionen. Journal für die reine und angewandte Mathematik, Vol. 207, pp. 996-112, 1961.

[15] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational Iterative Methods for Nonsymmetric Systems of Linear Equations. SIAM Journal on Numerical Analysis, Vol. 20, No. 2, pp. 345–357, 1983.

[16] C. F. Gauss. Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem. Perthes et Besser, 1809.

[17] L. Seidel. Über ein Verfahren, die Gleichungen, auf welche die Methode der kleinsten Quadrate führt, sowie lineäre Gleichungen überhaupt, durch successive Annäherung aufzulösen. Abhandlungen der Bayerischen Akademie, Vol. 11, pp. 81–108, 1873.

[18] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, Vol. 7, No. 3, pp. 856–869, 1986.

[19] D. Young. Iterative Methods for Solving Partial Difference Equations of Elliptic Type. Doctoral Thesis, Harvard University, 1950.

[20] S. P. Frankel. Convergence Rates of Iterative Treatments of Partial Differential Equations. Mathematical Tables and Other Aids to Computation, Vol. 4, No. 30, pp. 65–75, 1950.

[21] Y. Saad. A Flexible Inner-outer Preconditioned GMRES Algorithm. SIAM Journal on Scientific and Statistical Computing, Vol. 14, No. 2, pp. 461–469, 1993.

[22] P. Sonnerveld and M. B. van Gijzen. IDR(s): a Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations. SIAM Journal on Scientific Computing, Vol. 31, No. 2, pp. 1035–1062, 2008.

[23] C. C. Paige and M. A. Saunders. Solution of Sparse Indefinite Systems of Linear Equations. SIAM Journal on Numerical Analysis, Vol. 12, No. 4, pp. 617–629, 1975.

[24] R. von Mises and H. Pollaczek-Geiringer. Praktische Verfahren der Gleichungsauflösung. Zeitschrift für Angewandte Mathematik und Mechanik, Vol. 9, No. 2, pp. 152–164, 1929.

[25] H. Wielandt. Beiträge zur mathematischen Behandlung komplexer Eigenwertprobleme, Teil V: Bestimmung höherer Eigenwerte durch gebrochene Iteration. Bericht B 44/J/37, Aerodynamische Versuchsanstalt Göttingen, 1944.

[26] J. W. S. Rayleigh. Some General Theorems relating to Vibrations. Proceedings of the London Mathematical Society, Vol. 4, No. 1, pp. 357–368, 1873.

[27] H. R. Rutishauser. Computational Aspects of F. L. Bauser's Simultaneous Iteration Method. Numerische Mathematik, Vol. 13, No. 1, pp. 4–13, 1969.

[28] C. Lanczos. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. Journal of Research of the National Bureau of Standards, Vol. 45, No. 4, pp. 255–282, 1950.

[29] A. V. Knyazev. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM Journal on Scientific Computing, Vol. 23, No. 2, pp. 517–541, 2001.

[30] E. Suetomi and H. Sekimoto. Conjugate Gradient Like Methods and Their Application to Eigenvalue Problems for Neutron Diffusion Equation. Annals of Nuclear Energy, Vol. 18, No. 4, pp. 205–227, 1991.

[31] O. Axelsson. A Survey of Preconditioned Iterative Methods for Linear Systems of Equations. BIT, Vol. 25, No. 1, pp. 166–187, 1985.

[32] I. Gustafsson. A Class of First Order Factorization Methods. BIT, Vol. 18, No. 2, pp. 142–156, 1978.

[33] Y. Saad. ILUT: A Dual Threshold Incomplete LU Factorization. Numerical Linear Algebra with Applications, Vol. 1, No. 4, pp. 387–402, 1994.

[34] Y. Saad, et al. ITSOL: ITERATIVE SOLVERS Package.
http://www-users.cs.umn.edu/~saad/software/ITSOL/.

[35] N. Li, Y. Saad, and E. Chow. Crout Version of ILU for General Sparse Matrices. SIAM Journal on Scientific Computing, Vol. 25, No. 2, pp. 716–728, 2003.

[36] T. Kohno, H. Kotakemori, and H. Niki. Improving the Modified Gauss-Seidel Method for Z-matrices. Linear Algebra and its Applications, Vol. 267, pp. 113–123, 1997.

[37] A. Fujii, A. Nishida, and Y. Oyanagi. Evaluation of Parallel Aggregate Creation Orders : Smoothed Aggregation Algebraic Multigrid Method. High Performance Computational Science and Engineering, pp. 99–122, Springer, 2005.

[38] K. Abe, S. Zhang, H. Hasegawa, and R. Himeno. A SOR-base Variable Preconditioned CGR Method (in Japanese). Transactions of the JSIAM, Vol. 11, No. 4, pp. 157–170, 2001.

[39] R. Bridson and W. P. Tang. Refining an Approximate Inverse. Journal of Computational and Applied Mathematics, Vol. 123, No. 1-2, pp. 293–306, 2000.

[40] T. Chan and T. Mathew. Domain Decomposition Algorithms. Acta Numerica, Vol. 3, pp. 61–143, 1994.

[41] M. Dryja and O. B. Widlund. Domain Decomposition Algorithms with Small Overlap. SIAM Journal on Scientific Computing, Vol. 15, No. 3, pp. 604–620, 1994.

[42] H. Kotakemori, H. Hasegawa, and A. Nishida. Performance Evaluation of a Parallel Iterative Method Library using OpenMP. Proceedings of the 8th International Conference on High Performance Computing in Asia Pacific Region, pp. 432–436, IEEE, 2005.

[43] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance Evaluation of Parallel Sparse Matrix-Vector Products on SGI Altix 3700. Lecture Notes in Computer Science 4315, pp. 153–163, Springer, 2008.

[44] D. H. Bailey. A Fortran-90 Double-Double Library. http://crd-legacy.lbl.gov/~dhbailey/mpdist/.

[45] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for Quad-Double Precision Floating Point Arithmetic. Proceedings of the 15th Symposium on Computer Arithmetic, pp. 155–162, 2001.

[46] T. Dekker. A Floating-Point Technique for Extending the Available Precision. Numerische Mathematik, Vol. 18, No. 3, pp. 224–242, 1971.

[47] D. E. Knuth. The Art of Computer Programming: Seminumerical Algorithms, Vol. 2. Addison-Wesley, 1969.

[48] D. H. Bailey. High-Precision Floating-Point Arithmetic in Scientific Computation. Computing in Science and Engineering, Vol. 7, No. 3, pp. 54–61, IEEE, 2005.

[49] Intel Fortran Compiler for Linux Systems User's Guide, Vol I. Intel Corporation, 2004.

[50] H. Kotakemori, A. Fujii, H. Hasegawa, and A. Nishida. Implementation of Fast Quad Precision Operation and Acceleration with SSE2 for Iterative Solver Library (in Japanese). IPSJ Transactions on Advanced Computing Systems, Vol. 1, No. 1, pp. 73–84, 2008.

[51] R. Courant and D. Hilbert. Methods of Mathematical Physics. Wiley-VCH, 1989.

[52] C. Lanczos. The Variational Principles of Mechanics, 4th Edition. University of Toronto Press, 1970.

[53] J. H. Wilkinson. The Algebraic Eigenvalue Problem. Oxford University Press, 1988.

[54] G. H. Golub and C. F. Van Loan. Matrix Computations, 3rd Edition. The Johns Hopkins University Press, 1996.

[55] Y. Saad. Numerical Methods for Large Eigenvalue Problems. Halsted Press, 1992.

[56] Y. Saad. Iterative Methods for Sparse Linear Systems. Second Edition. SIAM, 2003.

[57] Y. Saad. SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations, Version 2, 1994. http://www-users.cs.umn.edu/~saad/software/SPARSKIT/.

[58] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. SIAM, 1991.

[59] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A Sparse Matrix Library in C++ for High Performance Architectures. Proceedings of the Second Object Oriented Numerics Conference, pp. 214–218, 1992.

[60] R. Barrett, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, 1994.

[61] Z. Bai, et al. Templates for the Solution of Algebraic Eigenvalue Problems. SIAM, 2000.

[62] A. Greenbaum. Iterative Methods for Solving Linear Systems. SIAM, 1997.

[63] S. Balay, et al. PETSc Users Manual. Technical Report ANL-95/11, Argonne National Laboratory, 2004.

[64] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems. ACM Transactions on Mathematical Software, Vol. 31, No. 3, pp. 351–362, 2005.

[65] R. D. Falgout, J. E. Jones, and U. M. Yang. The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. Lecture Notes in Computational Science and Engineering 51, pp. 209–236, Springer, 2006.

[66] R. S. Tuminaro, et al. Official Aztec User's Guide, Version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, 1999.

[67] M. A. Heroux, et al. An Overview of the Trilinos Project. ACM Transactions on Mathematical Software, Vol. 31, No. 3, pp. 397–423, 2005.

[68] R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly-Restarted Arnoldi Methods. SIAM, 1998.

[69] R. Bramley and X. Wang. SPLIB: A Library of Iterative Methods for Sparse Linear System. Technical Report, Department of Computer Science, Indiana University, 1995.

[70] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I). Technical Report TR/PA/92/86, CERFACS, 1992.

[71] R. F. Boisvert, et al. The Matrix Market Exchange Formats: Initial Design. Technical Report NISTIR 5935, National Institute of Standards and Technology, 1996.

[72] A. Geist, et al. PVM: Parallel Virtual Machine. MIT Press, 1994.

[73] W. Gropp, E. Lusk, and A. Skjellum. Using MPI, 2nd Edition: Portable Parallel Programming with the Message-Passing Interface. MIT Press, 1999.

[74] B. Chapman, G. Jost, and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007.

A File Formats

This section describes the file formats available for the library.

A.1 Extended Matrix Market Format

The Matrix Market format does not support the vector data. The extended Matrix Market format is the extension of the Matrix Market format to handle the matrix and vector data. Assume that the number of nonzero elements of matrix $A = (a_{ij})$ of size $M \times N$ is L and that $a_{ij} = A(I, J)$. The format is as follows:

```
%%MatrixMarket matrix coordinate real general  <-- Header
%                                              <-+
%                                               | Comment lines with 0 or more lines
%                                              <-+
M N L B X                                      <-- Numbers of rows, columns, and
I1 J1 A(I1,J1)                                 <-+    nonzero elements (0 or 1) (0 or 1)
I2 J2 A(I2,J2)                                  | Row and column number values
 . . .                                          | The index is one origin
IL JL A(IL,JL)                                 <-+
I1 B(I1)                                        <-+
I2 B(I2)                                         | Exists only when B=1
 . . .                                           | Row number value
IM B(IM)                                        <-+
I1 X(I1)                                         <-+
I2 X(I2)                                         | Exists only when X=1
 . . .                                           | Row number value
IM X(IM)                                         <-+
```

The extended Matrix Market format for matrix A and vector b in Equation (A.1) is as follows:

$$A = \begin{pmatrix} 2 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & 2 \end{pmatrix} \qquad b = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \tag{A.1}$$

```
%%MatrixMarket matrix coordinate real general
4 4 10 1 0
1 2  1.00e+00
1 1  2.00e+00
2 3  1.00e+00
2 1  1.00e+00
2 2  2.00e+00
3 4  1.00e+00
3 2  1.00e+00
3 3  2.00e+00
4 4  2.00e+00
4 3  1.00e+00
1  0.00e+00
2  1.00e+00
3  2.00e+00
4  3.00e+00
```

A.2 Harwell-Boeing Format

The Harwell-Boeing format stores the matrix in the CSC format. Assume that the array `value` stores the values of the nonzero elements of matrix A, the array `index` stores the row indices of the nonzero

elements and the array `ptr` stores pointers to the top of each column in the arrays `value` and `index`. The format is as follows:

```
Line 1 (A72,A8)
   1 - 72 Title
  73 - 80 Key
Line 2 (5I14)
   1 - 14 Total number of lines excluding header
  15 - 28 Number of lines for ptr
  29 - 42 Number of lines for index
  43 - 56 Number of lines for value
  57 - 70 Number of lines for right-hand side vectors
Line 3 (A3,11X,4I14)
   1 -  3 Matrix type
           Col.1: R Real matrix
                  C Complex matrix (Not supported)
                  P Pattern only (Not supported)
           Col.2: S Symmetric
                  U Unsymmetric
                  H Hermitian (Not supported)
                  Z Skew symmetric (Not supported)
                  R Rectangular (Not supported)
           Col.3: A Assembled
                  E Elemental matrices (Not supported)
   4 - 14 Blank space
  15 - 28 Number of rows
  29 - 42 Number of columns
  43 - 56 Number of nonzero elements
  57 - 70 0
Line 4 (2A16,2A20)
   1 - 16 Format for ptr
  17 - 32 Format for index
  33 - 52 Format for value
  53 - 72 Format for right-hand side vectors
Line 5 (A3,11X,2I14) Only presents if there are right-hand side vectors
   1      right-hand side vector type
            F for full storage
            M for same format as matrix (Not supported)
   2      G if a starting vector is supplied
   3      X if an exact solution is supplied
   4 - 14 Blank space
  15 - 28 Number of right-hand side vectors
  29 - 42 Number of nonzero elements
```

The Harwell-Boeing format for matrix A and vector b in Equation (A.1) is as follows:

```
1--------10--------20--------30--------40--------50--------60--------70--------80
Harwell-Boeing format sample                                          Lis
            8             1             1             4             2
RUA                       4             4            10             4
(11i7)          (13i6)          (3e26.18)          (3e26.18)
F                             1             0
      1     3     6     9
      1     2     1     2     3     2     3     4     3     4
  2.000000000000000000E+00   1.000000000000000000E+00   1.000000000000000000E+00
  2.000000000000000000E+00   1.000000000000000000E+00   1.000000000000000000E+00
  2.000000000000000000E+00   1.000000000000000000E+00   1.000000000000000000E+00
  2.000000000000000000E+00
```

```
  0.000000000000000000E+00   1.000000000000000000E+00   2.000000000000000000E+00
  3.000000000000000000E+00
```

A.3   Extended Matrix Market Format for Vectors

The extended Matrix Market format for vectors is the extension of the Matrix Market format to handle the vector data. Assume that vector $b = (b_i)$ is a vector of size N and that $b_i = B(I)$. The format is as follows:

```
%%MatrixMarket vector coordinate real general  <--  Header
%                                              <-+
%                                                | Comment lines with 0 or more lines
%                                              <-+
N                                              <-- Number of rows
I1 B(I1)                                       <-+
I2 B(I2)                                          | Row number value
 . . .                                            | The index is one origin
IN B(IN)                                       <-+
```

The extended Matrix Market format for vector b in Equation (A.1) is as follows:

```
%%MatrixMarket vector coordinate real general
4
1  0.00e+00
2  1.00e+00
3  2.00e+00
4  3.00e+00
```

A.4   PLAIN Format for Vectors

The PLAIN format for vectors is designed to write vector values in order. Assume that vector $b = (b_i)$ is a vector of size N and that b_i is equal to $B(I)$. The format is as follows:

```
B(1)                                           <-+
B(2)                                              | Vector value
 . . .                                            |
B(N)                                           <-+
```

The PLAIN format for vector b in Equation (A.1) is as follows:

```
0.00e+00
1.00e+00
2.00e+00
3.00e+00
```