Lis User Manual

Version 1.2.97

The Scalable Software Infrastructure Project
http://www.ssisc.org/

August 15, 2012

Cover: Ogata Korin, Irises.

Contents

0 Changes from Version 1.1

1. Added the support for the eigensolvers.
2. Changed the specifications of the following functions:
   (a) Changed the names of `lis_output_residual_history()` and `lis_get_residual_history()` to `lis_solver_output_rhistory()` and `lis_solver_get_rhistory()`, respectively.
   (b) Changed the origin of the Fortran interfaces `lis_vector_set_value()` and `lis_vector_get_value()` to 1.
   (c) Changed the origin of the Fortran interface `lis_vector_set_size()` to 1.
   (d) Changed the name of the precision flag `-precision` to `-f`.
3. Changed the specifications of the integer types:
   (a) Replaced the type of integer in the C programs with `LIS_INT`, which is equivalent with `int` by default. If the preprossor macro `_LONGLONG` is defined, it is replaced with `long long int`.
   (b) Replaced the type of integer in the Fortran programs with `LIS_INTEGER`, which is equivalent with `integer` by default. If the preprossor macro `LONGLONG` is defined, it is replaced with `integer*8`.

1 Introduction

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

$$Ax = b$$

and the standard eigenvalue problems

$$Ax = \lambda x$$

with real sparse matrices using the iterative methods. 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	CR
BiCG	BiCR[2]
CGS	CRS[3]
BiCGSTAB	BiCRSTAB[3]
GPBiCG	GPBiCR[3]
BiCGSafe[1]	BiCRSafe[4]
BiCGSTAB(l)	TFQMR
Jacobi	Orthomin(m)
Gauss-Seidel	GMRES(m)
SOR	FGMRES(m)[5]
IDR(s)[13]	MINRES[14]

Table 2: Eigensolvers

Power
Inverse
Approximate Inverse
Rayleigh Quotient
Subspace
Lanczos
CG[18, 19]
CR[20]

Table 3: Preconditioners

Jacobi
SSOR
ILU(k)
ILUT[6, 7]
Crout ILU[8, 7]
I+S[9]
SA-AMG[10]
Hybrid[11]
SAINV[12]
Additive Schwarz
User defined

Table 4: Matrix Storage Formats

Compressed Row Storage	(CRS)
Compressed Column Storage	(CCS)
Modified Compressed Sparse Row	(MSR)
Diagonal	(DIA)
Ellpack-Itpack Generalized Diagonal	(ELL)
Jagged Diagonal	(JDS)
Block Sparse Row	(BSR)
Block Sparse Column	(BSC)
Variable Block Row	(VBR)
Dense	(DNS)
Coordinate	(COO)

2 Installation

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

2.1 System Requirements

Installation of Lis requires a C compiler. The Fortran interface requires a Fortran compiler. The algebraic multigrid preconditioner requires a Fortran 90 compiler. For parallel computing environments, the OpenMP library or the MPI-1 library is used. Lis has been tested on the environments shown 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, 4.3	Linux Mac OS X Windows
Microsoft Visual C++ 2008, 2010, 2012RC	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 4.3, 4.4 g95 0.91	Linux Mac OS X Windows

2.2 Extracting Archive

Enter the following command to extract the archive, where `($VERSION)` represents the version:

```
>gunzip -c lis-($VERSION).tar.gz | tar xvf -
```

It creates a directory `lis-($VERSION)` along with its subfolders as shown in Figure 1.

2.3 Installing on UNIX and Compatible Systems

2.3.1 Configuring Source Tree

Run the following script to configure the source tree:

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

```
lis-($VERSION)
 + config
 |  configuration files
 + include
 |  header files
 + src
 |  source files
 + test
 |  test programs
 + win32
    configuration files for Windows systems
```

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

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

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

`--enable-omp`	Enable the OpenMP library
`--enable-mpi`	Enable the MPI library
`--enable-fortran`	Enable the Fortran interface
`--enable-saamg`	Enable the SA-AMG preconditioner
`--enable-quad`	Enable the quadruple precision operations
`--enable-longlong`	Enable the 64bit integer
`--enable-shared`	Enable the dynamic linking
`--enable-gprof`	Enable the GNU profiler
`--prefix=<install-dir>`	Specify the installation destination
`TARGET=<target>`	Specify the computing environment
`CC=<c_compiler>`	Specify the C compiler
`CFLAGS=<c_flags>`	Specify the options for the C compiler
`FC=<fortran90_compiler>`	Specify the Fortran 90 compiler
`FCFLAGS=<fc_flags>`	Specify the options for the Fortran 90 compiler
`LDFLAGS=<ld_flags>`	Specify the link options

2.3.2 Compiling

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

```
>make
```

To ensure that the library has been successfully built, enter as follows in `lis-($VERSION)`:

```
>make check
```

It 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 writes the solution of the linear equation $Ax = b$ obtained by the BiCG method into `test/sol.txt`, and the residual history into `test/res.txt`. If the values of the elements of the solution are 1, then the result is correct. The result on the SGI Altix 3700 is shown below.

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 --enable-mpi ax_f77_mangling="lower case, no underscore, extra underscore"
fujitsu_fx10_cross	./configure CC=fccpx FC=frtpx CFLAGS="-Kfast,ocl,preex -w" 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 -I/bgl/BlueLight/ppcfloor/bglsys/include" FCFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90 -w -I/bgl/BlueLight/ppcfloor/bglsys/include" ac_cv_sizeof_void_p=4 cross_compiling=yes --enable-mpi ax_f77_mangling="lower case, no underscore, no extra underscore"
nec_es_cross	./configure CC=esmpic++ FC=esmpif90 AR=esar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes --enable-mpi --enable-omp ax_f77_mangling="lower case, no underscore, 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   : CRS
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   : CRS
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   : CRS
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
```

2.3.3 Installing

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

```
>make install
```

It copies the files to the destination directory as follows:

```
$(INSTALLDIR)
 +include
 |   +lis_config.h lis.h lisf.h
 +lib
 |   +liblis.a
 +share
     +doc/lis examples/lis
```

`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 successfully installed, enter as follows in `lis-($VERSION)`:

```
>make installcheck
```

It runs a test script using the executable files installed in `examples/lis`.

2.4 Installing on Windows Systems

Use one of the solution files or project files for the Microsoft Visual Studio in the directory `lis-($VERSION)/win32`. `lis_with_fortran.sln` is the solution file to be used with the Intel Visual Fortran Compiler.

`lis_with_fortran_mpi.sln` is the solution file to be used with the Visual Fortran and MPICH2. The header files are located in `lis-($VERSION)/include`. `lis_config_win32.h` is the header file required to build the library. `lis.h` and `lisf.h` are the header files required by the C and Fortran compilers, respectively. The library files are generated in `lis-($VERSION)/lib`. The executable files of the test programs are generated in `lis-($VERSION)/test`.

2.5 Testing

2.5.1 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` and the residual history to `residual_filename`. The extended Matrix Market format, which is extended to allow vector data, is supported (see Appendix). 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 $b = (1, \ldots, 1)^T$
2	Use $b = A \times (1, \ldots, 1)^T$
`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`.

2.5.2 test2

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

This program solves a discretized two dimensional Poisson equation $Ax = b$ using the five point central difference scheme, with the coefficient matrix A of size mn in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the solution to `solution_filename` and the residual history to `residual_filename`. The right hand side vector is set to make the values of the elements of the solution to be 1. The values `m` and `n` represent the numbers of the grid points in each dimension.

2.5.3 test3

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

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

2.5.4 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 to make the values of the elements of the solution x to be 1. `test4f.F` is the Fortran version of `test4.c`.

2.5.5 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 to make the values of the elements of the solution to be 1.

2.5.6 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` and the residual history to `residual_filename`. The Matrix Market format is supported. `etest1f.F` is the Fortran version of `etest1.c`.

2.5.7 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 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` and the residual history to `residual_filename`. The values `m` and `m` represent the numbers of the grid points in each dimension.

2.5.8 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 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` and the residual history to `residual_filename`. The values `l`, `m` and `n` represent the numbers of the grid points in each dimension.

2.5.9 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`.

2.5.10 etest5

`Usage: etest5 evalue_filename evector_filename`

This program solves the eigenvalue problem $Ax = \lambda x$ with the Subspace method, 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 12×12. It outputs 2 extreme eigenvalues of the smallest magnitude to `evalue_filename` and the associated eigenvectors to `evector_filename` in the extended Matrix Market format (see Appendix).

2.5.11 spmvtest1

`Usage: spmvtest1 n iter [matrix_type]`

This program computes the multiply of a tridiagonal matrix

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

of size n, derived from a discretized one dimensional Poisson equation 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

2.5.12 spmvtest2

`Usage: spmvtest2 m n iter [matrix_type]`

This program computes the multiply of a sparse matrix, derived from a discretized two dimensional Poisson equation 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 the grid points in each dimension.

2.5.13 spmvtest3

`Usage: spmvtest3 l m n iter [matrix_type]`

This program computes the multiply of a sparse matrix, derived from a discretized three dimensional Poisson equation using the seven point central difference scheme, and a vector $(1, \ldots, 1)^T$. The values `l`, `m` and `n` represent the numbers of the 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

2.5.14 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. If necessary, the block size of the BSR and BSC can be specified by `block`.

2.5.15 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. If necessary, the block size of the BSR and BSC can be specified by `block`.

2.6 Restrictions

The current version has the following restrictions:

- Preconditioners
  - If a preconditioner other than the Jacobi or SSOR is selected and the matrix A is not in the CRS format, a new matrix is created in the CRS 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.
- Quadruple precision operations
  - 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.
- Matrix storage formats

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

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 `include` statements:

- `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

```
#include "lis.h"
LIS_INT main(LIS_INT argc, char* argv[])
{
    lis_initialize(&argc, &argv);
    ...
    lis_finalize();
}
```

Fortran

```
#include "lisf.h"
      call lis_initialize(ierr)
     ...
      call lis_finalize(ierr)
```

Initializing

For initializing, the following functions are used:

- `C       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 integer in the C programs is `LIS_INT`, which is equivalent with `int`. If the preprossor macro `_LONGLONG` is defined, it is replaced with `long long int`. The default type of integer in the Fortran programs is `LIS_INTEGER`, which is equivalent with `integer`. If the preprossor 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 the 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 the vector v is stored on two processing elements, as shown in Equation (3.1), *global_n* and *local_n* are 4 and 2, respectively.

$$v = \begin{array}{c} \left(\begin{array}{c} 0 \\ 1 \\ \hline 2 \\ 3 \end{array} \right) \end{array} \begin{array}{l} \mathrm{PE0} \\ \\ \mathrm{PE1} \end{array} \tag{3.1}$$

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

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

C (for serial and multithreaded environments)

```
LIS_INT       i,n;
LIS_VECTOR    v;
n = 4;
lis_vector_create(0,&v);
lis_vector_set_size(v,0,n);             /* or lis_vector_set_size(v,n,0); */

for(i=0;i<n;i++)
{
    lis_vector_set_value(LIS_INS_VALUE,i,(double)i,v);
 }
```

C (for multiprocessing environment)

```
LIS_INT       i,n,is,ie;                /* or LIS_INT  i,ln,is,ie; */
LIS_VECTOR    v;
n = 4;                                  /* ln = 2; */
lis_vector_create(MPI_COMM_WORLD,&v);
lis_vector_set_size(v,0,n);             /* lis_vector_set_size(v,ln,0); */
lis_vector_get_range(v,&is,&ie);
for(i=is;i<ie;i++)
{
    lis_vector_set_value(LIS_INS_VALUE,i,(double)i,v);
 }
```

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 the 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 the 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.

In the case of 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 the 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 the vector v, the following functions are used:

- C `LIS_INT lis_vector_set(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 which 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 the matrix A is *global_n* $\times$ *global_n*, and that the size of each row block of the 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 the 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

In the case of creating the matrix A in Equation (3.2) in the CRS format, the 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 the matrix A in the CRS format are as follows, where the number of the processing elements for the multiprocessing environment is assumed to be two:

C (for serial and multithreaded environments)

```
LIS_INT       i,n;
LIS_MATRIX    A;
n = 4;
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);              /* or lis_matrix_set_size(A,n,0); */
for(i=0;i<n;i++) {
    if( i>0   ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0,A);
    if( i<n-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0,A);
    lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
}
lis_matrix_set_type(A,LIS_MATRIX_CRS);
lis_matrix_assemble(A);
```

C (for multiprocessing environment)

```
LIS_INT       i,n,gn,is,ie;
LIS_MATRIX    A;
gn = 4;                                  /* or n=2 */
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,0,gn);             /* lis_matrix_set_size(A,n,0); */
lis_matrix_get_size(A,&n,&gn);
lis_matrix_get_range(A,&is,&ie);
for(i=is;i<ie;i++) {
    if( i>0    ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0,A);
    if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0,A);
    lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
}
lis_matrix_set_type(A,LIS_MATRIX_CRS);
lis_matrix_assemble(A);
```

Fortran (for serial and multithreaded environments)

```
LIS_INTEGER   i,n
LIS_MATRIX    A
n = 4
call lis_matrix_create(0,A,ierr)
call lis_matrix_set_size(A,0,n,ierr)
do i=1,n
    if( i>1 ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0d0,A,ierr)
    if( i<n ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0d0,A,ierr)
    call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
enddo
call lis_matrix_set_type(A,LIS_MATRIX_CRS,ierr)
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_CRS,ierr)
14:  call lis_matrix_assemble(A,ierr)
```

Declaring Variables

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

```
LIS_MATRIX    A;
```

Creating Matrices

To create the 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 the 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.

In the case of 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 the 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 the 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)`

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 the 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)`

`matrix_type` of A is `LIS_MATRIX_CRS` when the matrix is created. The following storage formats are supported:

Storage format		`matrix_type`
Compressed Row Storage	(CRS)	`{LIS_MATRIX_CRS|1}`
Compressed Column Storage	(CCS)	`{LIS_MATRIX_CCS|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	(JDS)	`{LIS_MATRIX_JDS|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}`
Dense	(DNS)	`{LIS_MATRIX_DNS|10}`
Coordinate	(COO)	`{LIS_MATRIX_COO|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

In the case of creating the matrix A in Equation (3.2) in the CRS format, the 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 the matrix A in the CRS format are as follows, where the number of the processing elements for the multiprocessing environment is assumed to be two:

C (for serial and multithreaded environments)

```
LIS_INT       i,k,n,nnz;
LIS_INT       *ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 10; k = 0;
lis_matrix_malloc_crs(n,nnz,&ptr,&index,&value);
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);              /* or lis_matrix_set_size(A,n,0); */

for(i=0;i<n;i++)
{
    if( i>0   ) {index[k] = i-1; value[k] = 1; k++;}
    index[k] = i; value[k] = 2; k++;
    if( i<n-1 ) {index[k] = i+1; value[k] = 1; k++;}
    ptr[i+1] = k;
 }
 ptr[0] = 0;
 lis_matrix_set_crs(nnz,ptr,index,value,A);
 lis_matrix_assemble(A);
```

C (for multiprocessing environment)

```
LIS_INT       i,k,n,nnz,is,ie;
LIS_INT       *ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 2; nnz = 5; k = 0;
lis_matrix_malloc_crs(n,nnz,&ptr,&index,&value);
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
lis_matrix_get_range(A,&is,&ie);
for(i=is;i<ie;i++)
{
    if( i>0   ) {index[k] = i-1; value[k] = 1; k++;}
    index[k] = i; value[k] = 2; k++;
    if( i<n-1 ) {index[k] = i+1; value[k] = 1; k++;}
    ptr[i-is+1] = k;
 }
 ptr[0] = 0;
 lis_matrix_set_crs(nnz,ptr,index,value,A);
 lis_matrix_assemble(A);
```

Associating Arrays

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

- C `LIS_INT lis_matrix_set_crs(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)`
- Fortran `subroutine lis_matrix_set_crs(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 the matrix A in Equation (3.2) in the CRS format and vector b in Equation (3.1) from an external file are as follows:

C (for serial, multithreaded and multiprocessing environments)

```
LIS_MATRIX    A;
LIS_VECTOR    b,x;
lis_matrix_create(LIS_COMM_WORLD,&A);
lis_vector_create(LIS_COMM_WORLD,&b);
lis_vector_create(LIS_COMM_WORLD,&x);
lis_matrix_set_type(A,LIS_MATRIX_CRS);
lis_input(A,b,x,"matvec.mtx");
```

Fortran (for serial, multithreaded and multiprocessing environments)

```
LIS_MATRIX    A
LIS_VECTOR    b,x
call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
call lis_vector_create(LIS_COMM_WORLD,b,ierr)
call lis_vector_create(LIS_COMM_WORLD,x,ierr)
call lis_matrix_set_type(A,LIS_MATRIX_CRS,ierr)
call lis_input(A,b,x,'matvec.mtx',ierr)
```

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

```
%%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:

- Matrix Market format
- Harwell-Boeing format

To read the data for the 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:

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

```
LIS_MATRIX A;
LIS_VECTOR b,x;
LIS_SOLVER solver;

/* Create matrix and vector */

lis_solver_create(&solver);
lis_solver_set_option("-i bicg -p none",solver);
lis_solver_set_option("-tol 1.0e-12",solver);
lis_solver(A,b,x,solver);
```

Fortran (for serial, multithreaded and multiprocessing environments)

```
LIS_MATRIX A
LIS_VECTOR b,x
LIS_SOLVER solver

/* Create matrix and vector */

call lis_solver_create(solver,ierr)
call lis_solver_set_option('-i bicg -p none',solver,ierr)
call lis_solver_set_option('-tol 1.0e-12',solver,ierr)
call lis_solver(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 which sets the options specified on the command line, and pass 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 the 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 the preconditioner of the $I + \alpha S^{(m)}$ type
		`-is_m [3]`	The parameter m of the preconditioner of the $I + \alpha S^{(m)}$ type
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Selects the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \leq \theta^2 \lvert a_{ii} \rvert \lvert a_{jj} \rvert$ (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 the iterations

Other Options

Option		
`-maxiter [1000]`	The maximum number of the 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 the threads (`t` represents the maximum number of the threads)	
`-storage [0]`	The matrix storage format	
`-storage_block [2]`	The block size of the BSR and BSC	
`-f [0]`	The precision of the linear solvers	
	`-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)

```
LIS_MATRIX A;
LIS_VECTOR x;
LIS_REAL evalue;
LIS_ESOLVER esolver;

/* Create matrix and vector */

lis_esolver_create(&esolver);
lis_esolver_set_option("-e ii -i bicg -p none",esolver);
lis_esolver_set_option("-etol 1.0e-12 -tol 1.0e-12",esolver);
lis_esolve(A,x,evalue,esolver);
```

Fortran (for serial, multithreaded and multiprocessing environments)

```
LIS_MATRIX A
LIS_VECTOR x
LIS_REAL evalue
LIS_ESOLVER esolver

/* Create matrix and vector */

call lis_esolver_create(esolver,ierr)
call lis_esolver_set_option('-e ii -i bicg -p none',esolver,ierr)
call lis_esolver_set_option('-etol 1.0e-12 -tol 1.0e-12',esolver,ierr)
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 which sets the options specified on the command line, and pass 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 [bicg]`	The linear solver
Approximate Inverse	`-e {aii|3}`		
Rayleigh Quotient	`-e {rqi|4}`	`-i [bicg]`	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 the 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 the preconditioner of the $I + \alpha S^{(m)}$ type
		`-is_m [3]`	The parameter m of the preconditioner of the $I + \alpha S^{(m)}$ type
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Selects the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \leq \theta^2 \lvert a_{ii} \rvert \lvert a_{jj} \rvert$ (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 the iterations

Other Options

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

Solving Eivenvalue 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 the 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 to make the values of the elements of the solution x is 1. The program is located in the directory `lis-($VERSION)/test`.

Test program: test4.c

```
#include <stdio.h>
#include "lis.h"
main(LIS_INT argc, char *argv[])
{
    LIS_INT i,n,gn,is,ie,iter;
    LIS_MATRIX A;
    LIS_VECTOR b,x,u;
    LIS_SOLVER solver;
    n = 12;
    lis_initialize(&argc,&argv);
    lis_matrix_create(LIS_COMM_WORLD,&A);
    lis_matrix_set_size(A,0,n);
    lis_matrix_get_size(A,&n,&gn)
    lis_matrix_get_range(A,&is,&ie)
    for(i=is;i<ie;i++)
    {
        if( i>0 ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0,A);
        if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0,A);
        lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
    }
    lis_matrix_set_type(A,LIS_MATRIX_CRS);
    lis_matrix_assemble(A);

    lis_vector_duplicate(A,&u);
    lis_vector_duplicate(A,&b);
    lis_vector_duplicate(A,&x);
    lis_vector_set_all(1.0,u);
    lis_matvec(A,u,b);

    lis_solver_create(&solver);
    lis_solver_set_optionC(solver);
    lis_solve(A,b,x,solver);
    lis_solver_get_iters(solver,&iter);
    printf("iter = %d\n",iter);
    lis_vector_print(x);
    lis_matrix_destroy(A);
    lis_vector_destroy(u);
    lis_vector_destroy(b);
    lis_vector_destroy(x);
    lis_solver_destroy(solver);
    lis_finalize();
    return 0;
}
}
```

Test program: test4f.F

```
      implicit none

#include "lisf.h"

      LIS_INTEGER      i,n,gn,is,ie,iter,ierr
      LIS_MATRIX        A
      LIS_VECTOR        b,x,u
      LIS_SOLVER        solver
      n  = 12
      call lis_initialize(ierr)
      call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
      call lis_matrix_set_size(A,0,n,ierr)
      call lis_matrix_get_size(A,n,gn,ierr)
      call lis_matrix_get_range(A,is,ie,ierr)
      do i=is,ie-1
        if( i>1  ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0d0,
     .                                        A,ierr)
        if( i<gn ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0d0,
     .                                        A,ierr)
        call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
      enddo
      call lis_matrix_set_type(A,LIS_MATRIX_CRS,ierr)
      call lis_matrix_assemble(A,ierr)

      call lis_vector_duplicate(A,u,ierr)
      call lis_vector_duplicate(A,b,ierr)
      call lis_vector_duplicate(A,x,ierr)
      call lis_vector_set_all(1.0d0,u,ierr)
      call lis_matvec(A,u,b,ierr)

      call lis_solver_create(solver,ierr)
      call lis_solver_set_optionC(solver,ierr)
      call lis_solve(A,b,x,solver,ierr)
      call lis_solver_get_iters(solver,iter,ierr)
      write(*,*) 'iter = ',iter
      call lis_vector_print(x,ierr)
      call lis_matrix_destroy(A,ierr)
      call lis_vector_destroy(b,ierr)
      call lis_vector_destroy(x,ierr)
      call lis_vector_destroy(u,ierr)
      call lis_solver_destroy(solver,ierr)
      call lis_finalize(ierr)

      stop
      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 8.1 (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.

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 8.1 (ifort). Since an `include` statement is used in the program, the 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   : CRS
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. Lis supports "double-double", or quadruple precision operations by combining two double precision floating point numbers[15, 16]. To use the quadruple 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 quadruple precision operations with the SIMD instructions, such as Intel's Streaming SIMD Extensions (SSE)[24].

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 to make the values of the elements of the solution to be 1. The value n is the size of the matrix A. `test5` with `-f` option is run as follows:

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   : CRS
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   : CRS
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 the nonzero elements of the matrix A of size $n \times n$ is nnz.

5.1 Compressed Row Storage (CRS)

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

- `value` is a double precision array with a length of nnz, which stores the nonzero elements of the matrix A along the row.
- `index` is an integer array with a length of nnz, which stores the column numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array with a length of $n+1$, 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 right diagram in Figure 2 shows how the matrix A in Figure 2 is stored in the CRS format. A program to create the matrix in the CRS format is as follows:

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

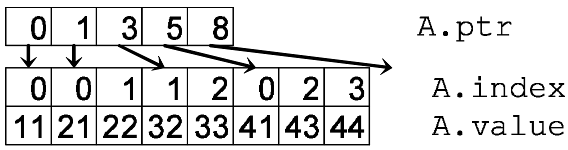

Figure 2: Data structure of CRS format (for serial and multithreaded environments).

For serial and multithreaded environments

```
LIS_INT       n,nnz;
LIS_INT       *ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 8;
ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

ptr[0] = 0; ptr[1] = 1; ptr[2] = 3; ptr[3] = 5; ptr[4] = 8;
index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  1;
index[4] =  2; index[5] =  0; index[6] =  2; index[7] =  3;
value[0] = 11; value[1] = 21; value[2] = 22; value[3] = 32;
value[4] = 33; value[5] = 41; value[6] = 43; value[7] = 44;

 lis_matrix_set_crs(nnz,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.1.2 Creating Matrices (for Multiprocessing Environment)

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

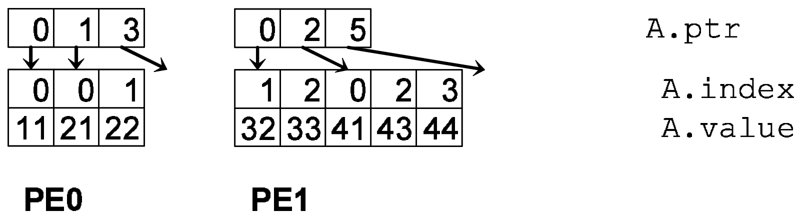

Figure 3: Data structure of CRS format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       i,k,n,nnz,my_rank;
LIS_INT       *ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnz = 3;}
else             {n = 2; nnz = 5;}
ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    ptr[0] = 0; ptr[1] = 1; ptr[2] = 3;
    index[0] =  0; index[1] =  0; index[2] =  1;
    value[0] = 11; value[1] = 21; value[2] = 22;}
else {
    ptr[0] = 0; ptr[1] = 2; ptr[2] = 5;
    index[0] =  1; index[1] =  2; index[2] =  0; index[3] =  2; index[4] =  3;
    value[0] = 32; value[1] = 33; value[2] = 41; value[3] = 43; value[4] = 44;}
 lis_matrix_set_crs(nnz,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.1.3 Associating Arrays

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

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

5.2 Compressed Column Storage (CCS)

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

- `value` is a double precision array with a length of nnz, which stores the values of the nonzero elements of the matrix A along the column.
- `index` is an integer array with a length of nnz, which stores the row numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array with a length of $n+1$, 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 right diagram in Figure 4 shows how the matrix A in Figure 4 is stored in the CCS format. A program to create the matrix in the CCS format is as follows:

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

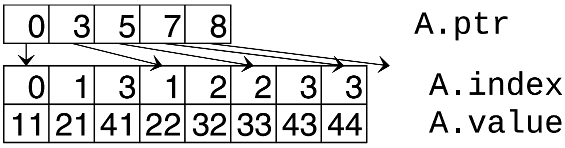

Figure 4: Data structure of CCS format (for serial and multithreaded environments).

For serial and multithreaded environments

```
LIS_INT       n,nnz;
LIS_INT       *ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 8;
ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

ptr[0] = 0; ptr[1] = 3; ptr[2] = 5; ptr[3] = 7; ptr[4] = 8;
index[0] =  0; index[1] =  1; index[2] =  3; index[3] =  1;
index[4] =  2; index[5] =  2; index[6] =  3; index[7] =  3;
value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22;
value[4] = 32; value[5] = 33; value[6] = 43; value[7] = 44;

 lis_matrix_set_ccs(nnz,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.2.2 Creating Matrices (for Multiprocessing Environment)

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

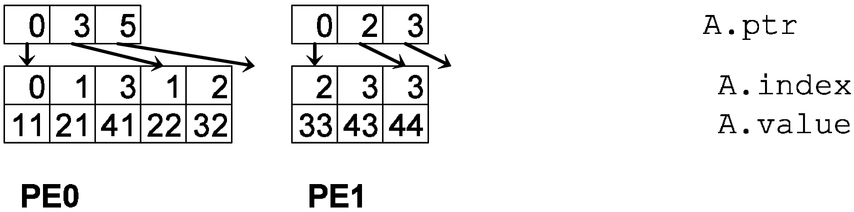

Figure 5: Data structure of CCS format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       i,k,n,nnz,my_rank;
LIS_INT       *ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnz = 3;}
else             {n = 2; nnz = 5;}
ptr   = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    ptr[0] = 0; ptr[1] = 3; ptr[2] = 5;
    index[0] =  0; index[1] =  1; index[2] =  3; index[3] =  1; index[4] =  2;
    value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22; value[4] = 32}
else {
    ptr[0] = 0; ptr[1] = 2; ptr[2] = 3;
    index[0] =  2; index[1] =  3; index[2] =  3;
    value[0] = 33; value[1] = 43; value[2] = 44;}
 lis_matrix_set_ccs(nnz,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.2.3 Associating Arrays

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

- C `LIS_INT lis_matrix_set_ccs(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)`
- Fortran `subroutine lis_matrix_set_ccs(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 the zero elements of the diagonal.

- `value` is a double precision array with a length of $nnz + ndz + 1$, which stores the diagonal of the 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 the matrix A are stored along the row.
- `index` is an integer array with a length of $nnz + ndz + 1$, which stores the starting points of the rows of the off-diagonal elements of the 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 the matrix A stored in the array `value`.

5.3.1 Creating Matrices (for Serial and Multithreaded Environments)

The right diagram 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:

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

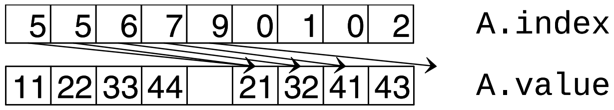

Figure 6: Data structure of MSR format (for serial and multithreaded environments).

For serial and multithreaded environments

```
LIS_INT       n,nnz,ndz;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 8; ndz = 0;
index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

index[0] =  5; index[1] =  5; index[2] =  6; index[3] =  7;
index[4] =  9; index[5] =  0; index[6] =  1; index[7] =  0; index[8] =  2;
value[0] = 11; value[1] = 22; value[2] = 33; value[3] = 44;
value[4] =  0; value[5] = 21; value[6] = 32; value[7] = 41; value[8] = 43;

 lis_matrix_set_msr(nnz,ndz,index,value,A);
 lis_matrix_assemble(A);
```

5.3.2 Creating Matrices (for Multiprocessing Environment)

Figure 7 shows how the 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:

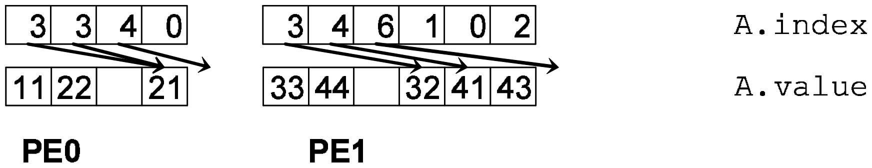

Figure 7: Data structure of MSR format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       i,k,n,nnz,ndz,my_rank;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnz = 3; ndz = 0;}
else             {n = 2; nnz = 5; ndz = 0;}
index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    index[0] =  3; index[1] =  3; index[2] =  4; index[3] =  0;
    value[0] = 11; value[1] = 22; value[2] =  0; value[3] = 21;}
else {
    index[0] =  3; index[1] =  4; index[2] =  6; index[3] =  1;
    index[4] =  0; index[5] =  2;
    value[0] = 33; value[1] = 44; value[2] =  0; value[3] = 32;
    value[4] = 41; value[5] = 43;}
 lis_matrix_set_msr(nnz,ndz,index,value,A);
 lis_matrix_assemble(A);
```

5.3.3 Associating Arrays

To associate the arrays in the MSR format with the 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 the nonzero diagonal elements of the matrix A.

- `value` is a double precision array with a length of $nnd \times n$, which stores the values of the nonzero diagonal elements of the matrix A.
- `index` is an integer array with a length of 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 the threads. nnd_p is the number of the nonzero diagonal elements of the partial matrix into which the row block of the matrix A is divided. $maxnnd$ is the maximum value nnd_p.

- `value` is a double precision array with a length of $maxnnd \times n$, which stores the values of the nonzero diagonal elements of the matrix A.
- `index` is an integer array with a length of $nprocs \times maxnnd$, which stores the offsets from the main diagonal.

5.4.1 Creating Matrices (for Serial Environment)

The right diagram in Figure 8 shows how the 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

```
LIS_INT       n,nnd;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnd = 3;
index = (LIS_INT *)malloc( nnd*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

index[0] = -3; index[1] = -1; index[2] =  0;
value[0] =  0; value[1] =  0; value[2] =  0; value[3] = 41;
value[4] =  0; value[5] = 21; value[6] = 32; value[7] = 43;
value[8] = 11; value[9] = 22; value[10]= 33; value[11]= 44;

 lis_matrix_set_dia(nnd,index,value,A);
 lis_matrix_assemble(A);
```

5.4.2 Creating Matrices (for Multithreaded Environment)

Figure 9 shows how the 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

```
LIS_INT       n,maxnnd,nprocs;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; maxnnd = 3; nprocs = 2;
index = (LIS_INT *)malloc( maxnnd*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( n*maxnnd*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

index[0] = -1; index[1] =  0; index[2] =  0; index[3] = -3; index[4] = -1; index[5] =  0;
value[0] =  0; value[1] = 21; value[2] = 11; value[3] = 22; value[4] =  0; value[5] =  0;
value[6] =  0; value[7] = 41; value[8] = 32; value[9] = 43; value[10]= 33; value[11]= 44;

 lis_matrix_set_dia(maxnnd,index,value,A);
 lis_matrix_assemble(A);
```

5.4.3 Creating Matrices (for Multiprocessing Environment)

Figure 10 shows how the 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:

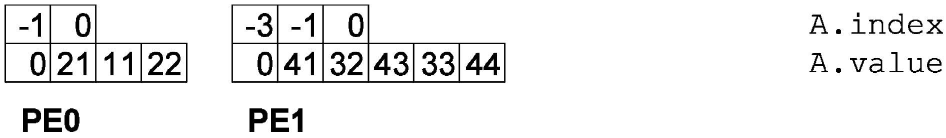

Figure 10: Data structure of DIA format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       i,n,nnd,my_rank;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnd = 2;}
else             {n = 2; nnd = 3;}
index = (LIS_INT *)malloc( nnd*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    index[0] = -1; index[1] =  0;
    value[0] =  0; value[1] = 21; value[2] = 11; value[3] = 22;}
else {
    index[0] = -3; index[1] = -1; index[2] =  0;
    value[0] =  0; value[1] = 41; value[2] = 32; value[3] = 43; value[4] = 33;
    value[5] = 44;}
 lis_matrix_set_dia(nnd,index,value,A);
 lis_matrix_assemble(A);
```

5.4.4 Associating Arrays

To associate the arrays in the DIA format with the 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)`

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 the nonzero elements in the rows of the matrix A.

- `value` is a double precision array with a length of $maxnzr \times n$, which stores the values of the nonzero elements of the rows of the 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 with a length of $maxnzr \times n$, which stores the column numbers of the nonzero elements stored in the array `value`. If the number of the 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 right diagram in Figure 11 shows how the 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

```
LIS_INT       n,maxnzr;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; maxnzr = 3;
index = (LIS_INT *)malloc( n*maxnzr*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0; index[4] =  0; index[5] =  1;
index[6] =  2; index[7] =  2; index[8] =  0; index[9] =  1; index[10]=  2; index[11]=  3;
value[0] = 11; value[1] = 21; value[2] = 32; value[3] = 41; value[4] =  0; value[5] = 22;
value[6] = 33; value[7] = 43; value[8] =  0; value[9] =  0; value[10]=  0; value[11]= 44;

 lis_matrix_set_ell(maxnzr,index,value,A);
 lis_matrix_assemble(A);
```

5.5.2 Creating Matrices (for Multiprocessing Environment)

Figure 12 shows how the 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:

0	0	0	1		1	0	2	2	2	3	A.index
11	21	0	22		32	41	33	43	0	44	A.value
PE0					**PE1**						

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

For multiprocessing environment

```
LIS_INT       i,n,maxnzr,my_rank;
LIS_INT       *index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; maxnzr = 2;}
else             {n = 2; maxnzr = 3;}
index = (LIS_INT *)malloc( n*maxnzr*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    index[0] =  0; index[1] =  0; index[2] =  0; index[3] =  1;
    value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 22;}
else {
    index[0] =  1; index[1] =  0; index[2] =  2; index[3] =  2; index[4] =  2;
    index[5] =  3;
    value[0] = 32; value[1] = 41; value[2] = 33; value[3] = 43; value[4] =  0;
    value[5] = 44;}
 lis_matrix_set_ell(maxnzr,index,value,A);
 lis_matrix_assemble(A);
```

5.5.3 Associating Arrays

To associate an array required by the ELL format with the 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)`

5.6 Jagged Diagonal (JDS)

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

- `perm` is an integer array with a length of n, which stores the sorted row numbers.
- `value` is a double precision array with a length of 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 with a length of nnz, which stores the row numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array with a length of $maxnzr + 1$, 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 the threads. $maxnzr_p$ is the number of the nonzero diagonal elements of the partial matrix into which the row block of the matrix A is divided. $maxmaxnzr$ is the maximum value of $maxnzr_p$.

- `perm` is an integer array with a length of n, which stores the sorted row numbers.
- `value` is a double precision array with a length of 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 with a length of nnz, which stores the row numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array with a length of $nprocs \times (maxmaxnzr + 1)$, which stores the starting points of the jagged diagonal elements.

5.6.1 Creating Matrices (for Serial Environment)

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

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

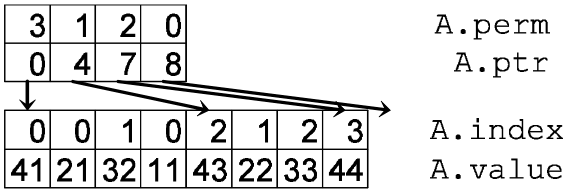

Figure 13: Data structure of JDS format (for serial environment).

For serial environment

```
LIS_INT       n,nnz,maxnzr;
LIS_INT       *perm,*ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 8; maxnzr = 3;
perm  = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
ptr   = (LIS_INT *)malloc( (maxnzr+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

perm[0] = 3; perm[1] = 1; perm[2] = 2; perm[3] = 0;
ptr[0]  = 0; ptr[1]  = 4; ptr[2]  = 7; ptr[3]  = 8;
index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0;
index[4] =  2; index[5] =  1; index[6] =  2; index[7] =  3;
value[0] = 41; value[1] = 21; value[2] = 32; value[3] = 11;
value[4] = 43; value[5] = 22; value[6] = 33; value[7] = 44;

 lis_matrix_set_jds(nnz,maxnzr,perm,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.6.2 Creating Matrices (for Multithreaded Environment)

Figure 14 shows how the matrix A in Figure 13 is stored in the JDS format on two threads. A program to create the matrix in the JDS format on two threads is as follows:

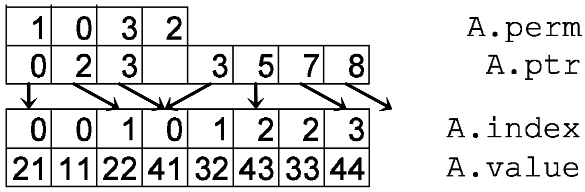

Figure 14: Data structure of JDS format (for multithreaded environment).

For multithreaded environment

```
LIS_INT       n,nnz,maxmaxnzr,nprocs;
LIS_INT       *perm,*ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 8; maxmaxnzr = 3; nprocs = 2;
perm  = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
ptr   = (LIS_INT *)malloc( nprocs*(maxmaxnzr+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

perm[0] = 1; perm[1] = 0; perm[2] = 3; perm[3] = 2;
ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 3; ptr[3]  = 0;
ptr[4]  = 3; ptr[5]  = 5; ptr[6]  = 7; ptr[7]  = 8;
index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0;
index[4] =  1; index[5] =  2; index[6] =  2; index[7] =  3;
value[0] = 21; value[1] = 11; value[2] = 22; value[3] = 41;
value[4] = 32; value[5] = 43; value[6] = 33; value[7] = 44;

 lis_matrix_set_jds(nnz,maxmaxnzr,perm,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.6.3 Creating Matrices (for Multiprocessing Environment)

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

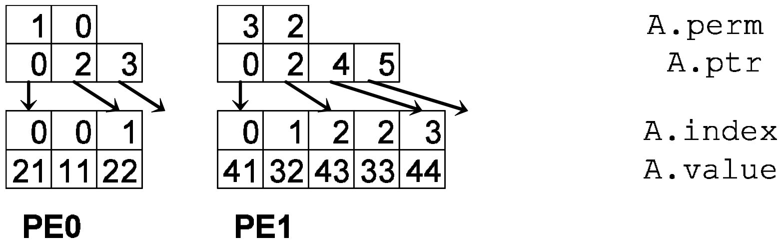

Figure 15: Data structure of JDS format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       i,n,nnz,maxnzr,my_rank;
LIS_INT       *perm,*ptr,*index;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnz = 3; maxnzr = 2;}
else             {n = 2; nnz = 5; maxnzr = 3;}
perm  = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
ptr   = (LIS_INT *)malloc( (maxnzr+1)*sizeof(LIS_INT) );
index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    perm[0] = 1; perm[1] = 0;
    ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 3;
    index[0] =  0; index[1] =  0; index[2] =  1;
    value[0] = 21; value[1] = 11; value[2] = 22;}
else {
    perm[0] = 3; perm[1] = 2;
    ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 4; ptr[3]  = 5;
    index[0] =  0; index[1] =  1; index[2] =  2; index[3] =  2; index[4] =  3;
    value[0] = 41; value[1] = 32; value[2] = 43; value[3] = 33; value[4] = 44;}
 lis_matrix_set_jds(nnz,maxnzr,perm,ptr,index,value,A);
 lis_matrix_assemble(A);
```

5.6.4 Associating Arrays

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

- `C       LIS_INT lis_matrix_set_jds(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_jds(LIS_INTEGER nnz, LIS_INTEGER maxnzr, LIS_INTEGER ptr(),integer index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)`

5.7 Block Sparse Row (BSR)

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

- `value` is a double precision array with a length of $nnzb \times r \times c$, which stores the values of the elements of the nonzero blocks.
- `bindex` is an integer array with a length of $nnzb$, which stores the block column numbers of the nonzero blocks.
- `bptr` is an integer array with a length of $nr + 1$, which stores the starting points of the block rows in the array `bindex`.

5.7.1 Creating Matrices (for Serial and Multithreaded Environments)

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

$$A = \left(\begin{array}{cc|cc} 11 & & & \\ 21 & 22 & & \\ \hline & 32 & 33 & \\ 41 & & 43 & 44 \end{array}\right)$$

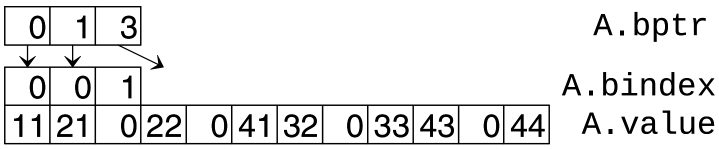

Figure 16: Data structure of BSR format (for serial and multithreaded environments).

For serial and multithreaded environments

```
LIS_INT       n,bnr,bnc,nr,nc,bnnz;
LIS_INT       *bptr,*bindex;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
bindex[0] =  0; bindex[1] =  0; bindex[2] =  1;
value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
value[4]  =  0; value[5] = 41; value[6] = 32; value[7] =  0;
value[8]  = 33; value[9] = 43; value[10]=  0; value[11]= 44;

 lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
 lis_matrix_assemble(A);
```

5.7.2 Creating Matrices (for Multiprocessing Environment)

Figure 17 shows how the 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:

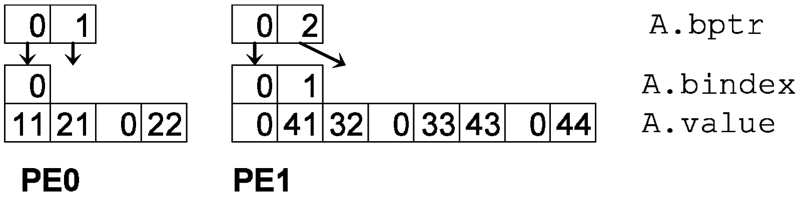

Figure 17: Data structure of BSR format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       n,bnr,bnc,nr,nc,bnnz,my_rank;
LIS_INT       *bptr,*bindex;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; bnr = 2; bnc = 2; bnnz = 1; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
else             {n = 2; bnr = 2; bnc = 2; bnnz = 2; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    bptr[0] = 0; bptr[1] = 1;
    bindex[0] =  0;
    value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;}
else {
    bptr[0] = 0; bptr[1] = 2;
    bindex[0] =  0; bindex[1] =  1;
    value[0]  =  0; value[1]  = 41; value[2] = 32; value[3] =  0;
    value[4]  = 33; value[5]  = 43; value[6] =  0; value[7] = 44;}
 lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
 lis_matrix_assemble(A);
```

5.7.3 Associating Arrays

To associate the arrays in the BSR format with the 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)`

5.8 Block Sparse Column (BSC)

The BSC format breaks down the matrix A into partial matrices called blocks, with a size of $r \times c$. The BSC format stores the nonzero blocks, in which at least one nonzero element exists, in the similar format as the CCS. 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 with a length of $nnzb \times r \times c$, which stores the values of the elements of the nonzero blocks.
- `bindex` is an integer array with a length of $nnzb$, which stores the block row numbers of the nonzero blocks.
- `bptr` is an integer array with a length of $nc+1$, which stores the starting points of the block columns in the array `bindex`.

5.8.1 Creating Matrices (for Serial and Multithreaded Environments)

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

$$A = \left(\begin{array}{cc|cc} 11 & & & \\ 21 & 22 & & \\ \hline & 32 & 33 & \\ 41 & & 43 & 44 \end{array}\right)$$

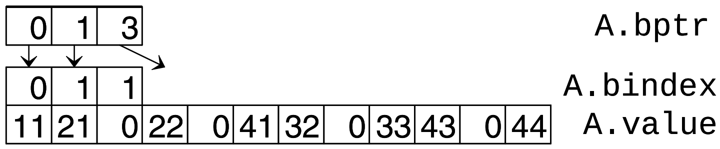

Figure 18: Data structure of BSC format (for serial and multithreaded environments).

For serial and multithreaded environments

```
LIS_INT       n,bnr,bnc,nr,nc,bnnz;
LIS_INT       *bptr,*bindex;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
bptr   = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
bindex[0] =  0; bindex[1] =  1; bindex[2] =  1;
value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
value[4]  =  0; value[5] = 41; value[6] = 32; value[7] =  0;
value[8]  = 33; value[9] = 43; value[10]=  0; value[11]= 44;

 lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
 lis_matrix_assemble(A);
```

5.8.2 Creating Matrices (for Multiprocessing Environment)

Figure 19 shows how the 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:

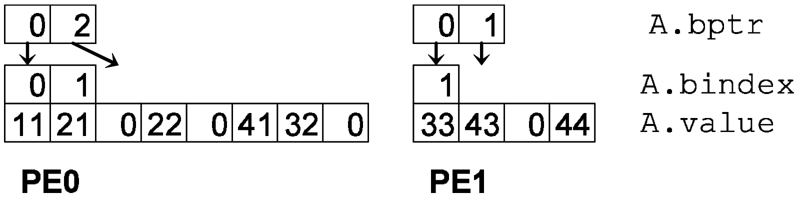

Figure 19: Data structure of BSC format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       n,bnr,bnc,nr,nc,bnnz,my_rank;
LIS_INT       *bptr,*bindex;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; bnr = 2; bnc = 2; bnnz = 2; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
else             {n = 2; bnr = 2; bnc = 2; bnnz = 1; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    bptr[0] = 0; bptr[1] = 2;
    bindex[0] =  0; bindex[1] =  1;
    value[0]  = 11; value[1]  = 21; value[2] =  0; value[3] = 22;
    value[4]  =  0; value[5]  = 41; value[6] = 32; value[7] =  0;}
else {
    bptr[0] = 0; bptr[1] = 1;
    bindex[0] =  1;
    value[0]  = 33; value[1]  = 43; value[2] =  0; value[3] = 44;}
 lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
 lis_matrix_assemble(A);
```

5.8.3 Associating Arrays

To associate the arrays in the BSC format with the 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 the similar format as the CRS. Assume that nr and nc are the numbers of row and column divisions, respectively, and that $nnzb$ denotes the number of the nonzero blocks of A, and nnz denotes the total number of the 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 with a length of $nr + 1$, which stores the starting row number of the block rows.
- `col` is an integer array with a length of $nc + 1$, which stores the starting column number of the block columns.
- `bindex` is an integer array with a length of $nnzb$, which stores the block column numbers of the nonzero blocks.
- `bptr` is an integer array with a length of $nr + 1$, which stores the starting points of the block rows in the array `bindex`.
- `value` is a double precision array with a length of nnz, which stores the values of the elements of the nonzero blocks.
- `ptr` is an integer array with a length of $nnzb + 1$, which stores the starting points of the nonzero blocks in the array `value`.

5.9.1 Creating Matrices (for Serial and Multithreaded Environments)

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

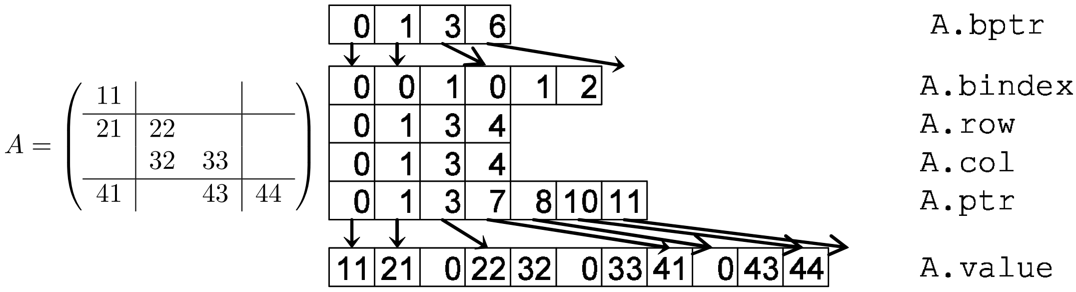

Figure 20: Data structure of VBR format (for serial and multithreaded environments).

For serial and multithreaded environments

```
LIS_INT       n,nnz,nr,nc,bnnz;
LIS_INT       *row,*col,*ptr,*bptr,*bindex;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 11; bnnz = 6; nr = 3; nc = 3;
bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
row    = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
col    = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
ptr    = (LIS_INT *)malloc( (bnnz+1)*sizeof(LIS_INT) );
bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
value  = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

bptr[0] = 0; bptr[1] = 1; bptr[2] = 3; bptr[3] = 6;
row[0]  = 0; row[1]  = 1; row[2]  = 3; row[3] = 4;
col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
bindex[0] =  0; bindex[1] =  0; bindex[2] =  1; bindex[3] =  0;
bindex[4] =  1; bindex[5] =  2;
ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  7;
ptr[4]    =  8; ptr[5]    = 10; ptr[6]    = 11;
value[0]  = 11; value[1]  = 21; value[2]  =  0; value[3]  = 22;
value[4]  = 32; value[5]  =  0; value[6]  = 33; value[7]  = 41;
value[8]  =  0; value[9]  = 43; value[10] = 44;

 lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
 lis_matrix_assemble(A);
```

5.9.2 Creating Matrices (for Multiprocessing Environment)

Figure 21 shows how the 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:

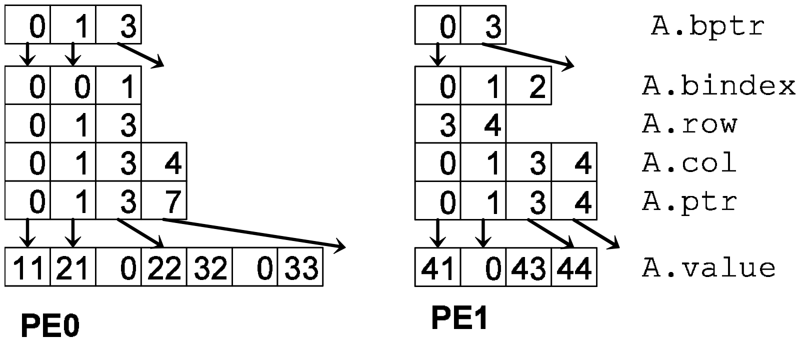

Figure 21: Data structure of VBR format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       n,nnz,nr,nc,bnnz,my_rank;
LIS_INT       *row,*col,*ptr,*bptr,*bindex;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnz = 7; bnnz = 3; nr = 2; nc = 3;}
else             {n = 2; nnz = 4; bnnz = 3; nr = 1; nc = 3;}
bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
row    = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
col    = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
ptr    = (LIS_INT *)malloc( (bnnz+1)*sizeof(LIS_INT) );
bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
value  = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
    row[0]  = 0; row[1]  = 1; row[2]  = 3;
    col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
    bindex[0] =  0; bindex[1] =  0; bindex[2] =  1;
    ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  7;
    value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
    value[4]  = 32; value[5] =  0; value[6] = 33;}
else {
    bptr[0] = 0; bptr[1] = 3;
    row[0]  = 3; row[1]  = 4;
    col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
    bindex[0] =  0; bindex[1] =  1; bindex[2] =  2;
    ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  4;
    value[0]  = 41; value[1]  =  0; value[2]  = 43; value[3]  = 44;}
 lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
 lis_matrix_assemble(A);
```

5.9.3 Associating Arrays

To associate the arrays in the VBR format with the 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 with a length of nnz, which stores the values of the nonzero elements.
- `row` is an integer array with a length of nnz, which stores the row numbers of the nonzero elements.
- `col` is an integer array with a length of nnz, which stores the column numbers of the nonzero elements.

5.10.1 Creating Matrices (for Serial and Multithreaded Environments)

The right diagram in Figure 22 shows how the 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

```
LIS_INT       n,nnz;
LIS_INT       *row,*col;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4; nnz = 8;
row   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
col   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

row[0] = 0; row[1] = 1; row[2] = 3; row[3] = 1;
row[4] = 2; row[5] = 2; row[6] = 3; row[7] = 3;
col[0] = 0; col[1] = 0; col[2] = 0; col[3] = 1;
col[4] = 1; col[5] = 2; col[6] = 2; col[7] = 3;
value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22;
value[4] = 32; value[5] = 33; value[6] = 43; value[7] = 44;

 lis_matrix_set_coo(nnz,row,col,value,A);
 lis_matrix_assemble(A);
```

5.10.2 Creating Matrices (for Multiprocessing Environment)

Figure 23 shows how the 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:

PE0				PE1					
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

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

For multiprocessing environment

```
LIS_INT       n,nnz,my_rank;
LIS_INT       *row,*col;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2; nnz = 3;}
else             {n = 2; nnz = 5;}
row   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
col   = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    row[0] = 0; row[1] = 1; row[2] = 1;
    col[0] = 0; col[1] = 0; col[2] = 1;
    value[0] = 11; value[1] = 21; value[2] = 22;}
else {
    row[0] = 3; row[1] = 2; row[2] = 2; row[3] = 3; row[4] = 3;
    col[0] = 0; col[1] = 1; col[2] = 2; col[3] = 2; col[4] = 3;
    value[0] = 41; value[1] = 32; value[2] = 33; value[3] = 43; value[4] = 44;}
 lis_matrix_set_coo(nnz,row,col,value,A);
 lis_matrix_assemble(A);
```

5.10.3 Associating Arrays

To associate the arrays in the COO format with the 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 with a length of $n \times n$, 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 the 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	A.Value
0	0	33	43	0	0	0	44	

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

For serial and multithreaded environments

```
LIS_INT       n;
LIS_SCALAR    *value;
LIS_MATRIX    A;
n = 4;
value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
lis_matrix_create(0,&A);
lis_matrix_set_size(A,0,n);

value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 41;
value[4] =  0; value[5] = 22; value[6] = 32; value[7] =  0;
value[8] =  0; value[9] =  0; value[10]= 33; value[11]= 43;
value[12]=  0; value[13]=  0; value[14]=  0; value[15]= 44;

 lis_matrix_set_dns(value,A);
 lis_matrix_assemble(A);
```

5.11.2 Creating Matrices (for Multiprocessing Environment)

Figure 25 shows how the 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:

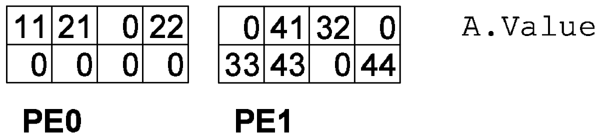

Figure 25: Data structure of DNS format (for multiprocessing environment).

For multiprocessing environment

```
LIS_INT       n,my_rank;
LIS_SCALAR    *value;
LIS_MATRIX    A;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
if( my_rank==0 ) {n = 2;}
else             {n = 2;}
value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
lis_matrix_create(MPI_COMM_WORLD,&A);
lis_matrix_set_size(A,n,0);
if( my_rank==0 ) {
    value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 22;
    value[4] =  0; value[5] =  0; value[6] =  0; value[7] =  0;}
else {
    value[0] =  0; value[1] = 41; value[2] = 32; value[3] =  0;
    value[4] = 33; value[5] = 43; value[6] =  0; value[7] = 44;}
 lis_matrix_set_dns(value,A);
 lis_matrix_assemble(A);
```

5.11.3 Associating Arrays

To associate the arrays in the DNS format with the 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 as follows:

`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 the 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 the 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 the 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 the vector v_{out} which has the same information as v_{in}

Input

`vin`	The source vector

Output

`vout`	The destination vector
`ierr`	The return code

Note

The function `lis_vector_duplicate` does not copy the values, but only allocates 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 the 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.

In the case of 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 the 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 the 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

In the case of 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 next 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 the 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 destination 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 value of the i-th row of the vector v

Input

`i`	The location where the value is assigned
`v`	The destination 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 the scalar values `value[`i`]` to the `index[`i`]`-th rows of the vector v

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 the elements of the array which stores the scalar values to be assigned
`index`	The array which stores the location where the scalar values are assigned
`value`	The array which stores the scalar values to be assigned
`v`	The destination vector

Output

`v`	The vector with the 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 the scalar values of the $start+i$-th row of the vector v, where $i = 0, 1, ..., count - 1$

Input

`start`	The starting location
`count`	The number of the values to get
`v`	The destination vector

Output

`value`	The vector 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 the scalar values of the i-th row of the vector v, where $i = 0, 1, ..., global\_n - 1$

Input

`value`	The array which stores the scalar values to be assigned

Output

`v`	The destination 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 the scalar values of the i-th row of the vector v, where $i = 0, 1, ..., global\_n - 1$

Input

`v`	The source vector

Output

`value`	The vector to store 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 x)
Fortran subroutine lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR x,
         LIS_INTEGER ierr)
```

Description

Assign the scalar *value* to the all elements of the vector v

Input

`value`	The scalar value to be assigned
`v`	The destination vector

Output

`v`	The vector with the *value* assigned to the all elements
`ierr`	The return code

6.2 Operating Matrix Elements

Assume that the size of the 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 the rows of the global matrix and the number of the 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 the matrix A

Input

`LIS_Comm`	The MPI communicator

Output

`A`	The matrix
`ierr`	The return code

Note

For the sequential and the 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 the 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 the matrix A_{out} which has the same information as the original 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 only allocates 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 the matrix A

Input

`A`	The matrix
`nnz_row`	The average number of the nonzero elements
`nnz`	The array of numbers of the 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 the 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 the 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 the matrix A

Input

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

Output

`ierr`	The return code

Note

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

In the case of 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 the 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 the matrix A

Input

`A`	The matrix

Output

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

Note

In case of 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 next 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_CRS` when the matrix is created. The table below shows the available storage formats for `matrix_type`.

Storage format		`matrix_type`
Compressed Row Storage	(CRS)	`{LIS_MATRIX_CRS|1}`
Compressed Column Storage	(CCS)	`{LIS_MATRIX_CCS|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	(JDS)	`{LIS_MATRIX_JDS|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}`
Dense	(DNS)	`{LIS_MATRIX_DNS|10}`
Coordinate	(COO)	`{LIS_MATRIX_COO|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

Input

`A`	The matrix
`bnr`	The row block size of the BSR (BSC) format or the number of the row blocks of the VBR format
`bnc`	The olumn block size of the BSR (BSC) format or the number of the 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 the 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 the A_{out} is set by `lis_matrix_set_type`. The block size of the BSR, BSC, and VBR is set by `lis_matrix_set_blocksize`.

The conversions indicated by one in the table below are performed directly, and the other ones are performed via the indicated formats. The conversions with no indication are performed via the CRS format.

Src \Dst	CRS	CCS	MSR	DIA	ELL	JDS	BSR	BSC	VBR	DNS	COO
CRS		1	1	1	1	1	1	CCS	1	1	1
COO	1	1	1	CRS	CRS	CRS	CRS	CCS	CRS	CRS	

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 the matrix A to the vector d

Input

`A`	The matrix

Output

`d`	The vector which stores the diagonal elements of the matrix
`ierr`	The return code

6.2.16 lis_matrix_set_crs

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

Description

Associate the arrays in the CRS format with the matrix A

Input

`nnz`	The number of nonzero elements
`ptr, index, value`	The rrays in the CRS format
`A`	The matrix

Output

`A`	The matrix associated with the arrays

Note

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

6.2.17 lis_matrix_set_ccs

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

Description

Associate the arrays in the CCS format with the matrix A

Input

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

Output

`A`	The matrix associated with the arrays

Note

After `lis_matrix_set_ccs` 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 the matrix A

Input

`nnz`	The number of the nonzero elements
`ndz`	The number of the 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 the matrix A

Input

`nnd`	The number of the 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 the matrix A

Input

`maxnzr`	The maximum number of the 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_jds

```
C       LIS_INT lis_matrix_set_jds(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_jds(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 JDS format with the matrix A

Input

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

Output

`A`	The matrix associated with the arrays

Note

After `lis_matrix_set_jds` 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 the matrix A

Input

`bnr`	The row block size
`bnc`	The column block size
`bnnz`	The number of the 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 the matrix A

Input

`bnr`	The row block size
`bnc`	The column block size
`bnnz`	The number of the 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 the matrix A

Input

`nnz`	The number of the all nonzero elements
`nr`	The number of the row blocks
`nc`	The number of the column blocks
`bnnz`	The number of the 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 the matrix A

Input

`nnz`	The number of the 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 the 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 the vector x by the scalar α

Input

`alpha`	The scalar value α
`x`	The vector to be multiplied

Output

`x`	The vector multiplied by α
`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 value
`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 the 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 the 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 the 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 vector sum $y = \alpha x + y$

Input

`alpha`	The scalar value
`x, y`	The vectors

Output

`y`	$\alpha x + y$ (the 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 vector sum $y = x + \alpha y$

Input

`alpha`	The scalar value
`x, y`	The vectors

Output

`y`	$x + \alpha y$ (the 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 vector sum $z = \alpha x + y$

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 the 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 which stores the diagonal elements of D^{-1} or $D^{-1/2}$
`ierr`	The return code

6.3.10 lis_matvec

```
C       void lis_matvec(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_matvec(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
```

Description

Calculate the matrix vector product $y = Ax$

Input

A	The matrix
x	The vector

Output

y	Ax

6.3.11 lis_matvect

```
C       void lis_matvect(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_matvect(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
```

Description

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

Input

A	The matrix
x	The vector

Output

y	$A^T x$

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 the 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 the preconditioner of the $I + \alpha S^{(m)}$ type
		`-is_m [3]`	The parameter m of the preconditioner of the $I + \alpha S^{(m)}$ type
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Selects 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 the iterations

Other Options

Option		
`-maxiter [1000]`	The maximum number of the 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 the threads	
	(`t` represents the maximum number of the threads)	
`-storage [0]`	The matrix storage format	
`-storage_block [2]`	The block size of the BSR and BSC	
`-f [0]`	The precision of the linear solvers	
	`-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, 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 the 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 number of iterations
`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 number of iterations from the solver

Input

`solver`	The solver

Output

`iters`	The number of the iterations
`iters_double`	The number of the double precision iterations
`iters_quad`	The number of the 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 execution time from the solver

Input

`solver`	The solver

Output

`times`	The total time in seconds
`itimes`	The time in seconds of the iteration
`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 iteration 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 redidual 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(VECTOR v)
Fortran subroutine lis_solver_get_rhistory(LIS_VECTOR v, LIS_INTEGER ierr)
```

Description

Store the residual norm history of the solver

Input

None

Output

`v`	The vector
`ierr`	The return code

Note

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

Note

The number of the solver is as follows:

Solver	Number	Solver	Number
CG	1	SOR	12
BiCG	2	BiCGSafe	13
CGS	3	CR	14
BiCGSTAB	4	BiCR	15
BiCGSTAB(l)	5	CRS	16
GPBiCG	6	BiCRSTAB	17
TFQMR	7	GPBiCR	18
Orthomin(m)	8	BiCRSafe	19
GMRES(m)	9	FGMRES(m)	20
Jacobi	10	IDR(s)	21
Gauss-Seidel	11	MINRES	22

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

Note

The number of the preconditioner is as follows:

Preconditioner	Number
none	0
Jacobi	1
ILU(k)	2
SSOR	3
Hybrid	4
I+S	5
SAINV	6
SA-AMG	7
Crout ILU	8
ILUT	9

6.4.16 lis_get_solvername

```
C       LIS_INT lis_get_solvername(LIS_INT nsol, char *name)
Fortran subroutine lis_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_get_preconname

```
C       LIS_INT lis_get_preconname(LIS_INT precon_type, char *name)
Fortran subroutine lis_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 [bicg]`	The linear solver
Approximate Inverse	`-e {aii|3}`		
Rayleigh Quotient	`-e {rqi|4}`	`-i [bicg]`	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 the 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 the preconditioner of the $I + \alpha S^{(m)}$ type
		`-is_m [3]`	The parameter m of the preconditioner of the $I + \alpha S^{(m)}$ type
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	The drop criterion
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	Selects the unsymmetric version (The matrix structure must be symmetric)
		`-saamg_theta [0.05|0.12]`	The drop criterion $a_{ij}^2 \le \theta^2 \|a_{ii}\|\|a_{jj}\|$ (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 the iterations

Other Options

Option	
`-emaxiter [1000]`	The maximum number of the 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 the threads
	(`t` represents the maximum number of the threads)
`-estorage [0]`	The matrix storage format
`-estorage_block [2]`	The block size of the BSR and BSC
`-ef [0]`	The precision of the eigensolvers
	`-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 the `-m [0]` option
`x`	The associated eigenvector
`esolver`	The number of the 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 number of the iterations
`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 the 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 number of iterations from the eigensolver

Input

`esolver`	The eigensolver

Output

`iters`	The number of the iterations
`iters_double`	The number of the double precision iterations
`iters_quad`	The number of the 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 execution time from the eigensolver

Input

`esolver`	The eigensolver

Output

`times`	The total time in seconds
`itimes`	The time in seconds of the iteration
`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 iteration 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 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(VECTOR v)
Fortran subroutine lis_esolver_get_rhistory(LIS_VECTOR v, LIS_INTEGER ierr)
```

Description

Store the residual norm history of the eigensolver

Input

None

Output

`v`	The vector
`ierr`	The return code

Note

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

Input

`esolver`	The eigensolver

Output

`v`	The vector which stores the eigenvalues
`ierr`	The return code

Note

The 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 the matrix A

Input

`esolver`	The eigensolver

Output

`A`	The matrix in the CRS format which stores the eigenvectors
`ierr`	The return code

Note

The 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

Note

The number of the eigensolver is as follows:

Eigensolver	Number
Power	1
Inverse	2
Approximate Inverse	3
Rayleigh Quotient	4
Subspace	5
Lanczos	6
CG	7
CR	8

6.5.16 lis_get_esolvername

```
C       LIS_INT lis_get_esolvername(LIS_INT nesol, char *ename)
Fortran subroutine lis_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 supported file formats are shown below:

- The 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 formats are supported:

- The PLAIN format
- The Matrix Market format

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, LIS_VECTOR x,
         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 supported file formats are shown below:

- The Matrix Market format (extended to allow vector data)
- 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 path, 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_ format, character path,
         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 the command line arguments
`argv`	The command line argument

Output

`ierr`	The return code

6.7.2 lis_finalize

```
C       void 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.

References

[1] S. Fujino, M. Fujiwara and M. Yoshida. BiCGSafe method based on minimization of associate residual (in Japanese). Transactions of JSCES, Paper No.20050028, 2005. http://save.k.u-tokyo.ac.jp/jsces/trans/trans2005/No20050028.pdf.

[2] T. Sogabe, M. Sugihara and S. Zhang. An Extension of the Conjugate Residual Method for Solving Nonsymmetric Linear Systems(in Japanese). Transactions of the Japan Society for Industrial and Applied Mathematics, Vol. 15, No. 3, pp. 445–460, 2005.

[3] 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.

[4] 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.

[5] Y. Saad. A Flexible Inner-outer Preconditioned GMRES Algorithm. SIAM J. Sci. Stat. Comput., Vol. 14, pp. 461–469, 1993.

[6] Y. Saad. ILUT: a dual threshold incomplete *LU* factorization. Numerical linear algebra with applications, Vol. 1, No. 4, pp. 387–402, 1994.

[7] ITSOL: ITERATIVE SOLVERS package
http://www-users.cs.umn.edu/˜saad/software/ITSOL/index.html.

[8] N. Li, Y. Saad and E. Chow. Crout version of ILU for general sparse matrices. SIAM J. Sci. Comput., Vol. 25, pp. 716–728, 2003.

[9] 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.

[10] 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.

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

[12] R. Bridson and W.-P. Tang. Refining an approximate inverse. J. Comput. Appl. Math., Vol. 123, pp. 293–306, 2000.

[13] 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 J. Sci. Comput., Vol. 31, Issue 2, pp. 1035–1062, 2008.

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

[15] D. H. Bailey. A fortran-90 double-double library. http://www.nersc.gov/˜dhbailey/mpdist /mpdist.html.

[16] 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.

[17] T. Dekker. A floating-point technique for extending the available precision. Numerische Mathematik, vol.18 pp. 224–242, 1971.

[18] A. V. Knyazev. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM J. Sci. Comput., Vol. 23, No. 2, pp. 517–541, 2001.

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

[20] 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.

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

[22] D. H. Bailey. High-Precision Floating-Point Arithmetic in Scientific Computation. Computing in Science and Engineering, Volume 7, Issue 3, pp. 54–61, IEEE, 2005.

[23] Intel Fortran Compiler User's Guide Vol I.

[24] 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.

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

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

[27] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations, version 2, June 1994. http://www.cs.umn.edu/˜saad/software/SPARSKIT/sparskit.html.

[28] S. Balay, et al. PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratory, August 2004.

[29] R. S. Tuminaro, et al. Official Aztec user's guide, version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, November 1999.

[30] 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.

[31] R. Bramley and X. Wang. SPLIB: A library of iterative methods for sparse linear system. Technical report, Indiana University–Bloomington, 1995.

[32] Matrix Market. http://math.nist.gov/MatrixMarket.

A File Formats

This section describes the file formats available for the library.

A.1 Extended Matrix Market Format

The Matrix Market format[32] 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 the nonzero elements of the 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 the matrix A and the 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 inputs and outputs the matrix in the CCS storage format. Assume that the array `value` stores the values of the nonzero elements of the 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 the matrix A and the 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 the 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 the 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 the 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 the vector b in Equation (A.1) is as follows:

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