Copyright © 2005 The Scalable Software Infrastructure Project, supported by “Development of Software Infrastructure for Large Scale Scientific Simulation” Team, CREST, JST.
Akira Nishida, CREST team director, JST.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

5 Matrix Storage Formats
5.1 Compressed Sparse Row (CSR) .................................................. 46
  5.1.1 Creating Matrices (for Serial and Multithreaded Environments) .... 46
  5.1.2 Creating Matrices (for Multiprocessing Environment) ............... 47
  5.1.3 Associating Arrays ......................................................... 47
5.2 Compressed Sparse Column (CSC) ............................................ 48
  5.2.1 Creating Matrices (for Serial and Multithreaded Environments) .... 48
  5.2.2 Creating Matrices (for Multiprocessing Environment) ............... 49
  5.2.3 Associating Arrays ......................................................... 49
5.3 Modified Compressed Sparse Row (MSR) .................................. 50
  5.3.1 Creating Matrices (for Serial and Multithreaded Environments) .... 50
  5.3.2 Creating Matrices (for Multiprocessing Environment) ............... 51
  5.3.3 Associating Arrays ......................................................... 51
5.4 Diagonal (DIA) ........................................................................ 52
  5.4.1 Creating Matrices (for Serial Environment) .............................. 52
  5.4.2 Creating Matrices (for Multithreaded Environment) ................... 53
  5.4.3 Creating Matrices (for Multiprocessing Environment) ............... 54
  5.4.4 Associating Arrays ......................................................... 54
5.5 Ellpack-Itpack Generalized Diagonal (ELL) .............................. 55
  5.5.1 Creating Matrices (for Serial and Multithreaded Environments) .... 55
  5.5.2 Creating Matrices (for Multiprocessing Environment) ............... 56
  5.5.3 Associating Arrays ......................................................... 56
5.6 Jagged Diagonal (JAD) .............................................................. 57
  5.6.1 Creating Matrices (for Serial Environment) .............................. 58
  5.6.2 Creating Matrices (for Multithreaded Environment) ................... 59
  5.6.3 Creating Matrices (for Multiprocessing Environment) ............... 60
  5.6.4 Associating Arrays ......................................................... 60
5.7 Block Sparse Row (BSR) .......................................................... 61
  5.7.1 Creating Matrices (for Serial and Multithreaded Environments) .... 61
  5.7.2 Creating Matrices (for Multiprocessing Environment) ............... 62
  5.7.3 Associating Arrays ......................................................... 62
5.8 Block Sparse Column (BSC) ...................................................... 63
  5.8.1 Creating Matrices (for Serial and Multithreaded Environments) .... 63
  5.8.2 Creating Matrices (for Multiprocessing Environment) ............... 64
  5.8.3 Associating Arrays ......................................................... 64
5.9 Variable Block Row (VBR) ....................................................... 65
  5.9.1 Creating Matrices (for Serial and Multithreaded Environments) .... 66
  5.9.2 Creating Matrices (for Multiprocessing Environment) ............... 67
  5.9.3 Associating Arrays ......................................................... 68
5.10 Coordinate (COO) ................................................................. 69
  5.10.1 Creating Matrices (for Serial and Multithreaded Environments) .... 69
  5.10.2 Creating Matrices (for Multiprocessing Environment) ............... 70
  5.10.3 Associating Arrays ......................................................... 70
5.11 Dense (DNS) ......................................................................... 71
  5.11.1 Creating Matrices (for Serial and Multithreaded Environments) .... 71
  5.11.2 Creating Matrices (for Multiprocessing Environment) ............... 72
  5.11.3 Associating Arrays ......................................................... 72
# 6 Functions

## 6.1 Operating Vector Elements
- 6.1.1 `lis_vector_create` .................................................. 74
- 6.1.2 `lis_vector_destroy` ................................................ 74
- 6.1.3 `lis_vector_duplicate` ............................................. 75
- 6.1.4 `lis_vector_set_size` ............................................. 75
- 6.1.5 `lis_vector_get_size` ............................................. 76
- 6.1.6 `lis_vector_get_range` .......................................... 76
- 6.1.7 `lis_vector_set_value` ........................................... 77
- 6.1.8 `lis_vector_get_value` ........................................... 77
- 6.1.9 `lis_vector_set_values` ........................................... 78
- 6.1.10 `lis_vector_get_values` ......................................... 79
- 6.1.11 `lis_vector_scatter` ............................................. 79
- 6.1.12 `lis_vector_gather` ............................................... 80
- 6.1.13 `lis_vector_is_null` ............................................. 80

## 6.2 Operating Matrix Elements
- 6.2.1 `lis_matrix_create` ............................................... 81
- 6.2.2 `lis_matrix_destroy` ............................................. 81
- 6.2.3 `lis_matrix_duplicate` ........................................... 82
- 6.2.4 `lis_matrix_malloc` .............................................. 82
- 6.2.5 `lis_matrix_set_value` ........................................... 83
- 6.2.6 `lis_matrix_assemble` ........................................... 83
- 6.2.7 `lis_matrix_set_size` ........................................... 84
- 6.2.8 `lis_matrix_get_size` ........................................... 84
- 6.2.9 `lis_matrix_get_range` .......................................... 85
- 6.2.10 `lis_matrix_get_nnz` ........................................... 85
- 6.2.11 `lis_matrix_set_type` ........................................... 86
- 6.2.12 `lis_matrix_get_type` ........................................... 86
- 6.2.13 `lis_matrix_set_csr` ........................................... 87
- 6.2.14 `lis_matrix_set_csc` ........................................... 87
- 6.2.15 `lis_matrix_set_csr` ........................................... 88
- 6.2.16 `lis_matrix_set_csc` ........................................... 88
- 6.2.17 `lis_matrix_set_ell` ............................................ 89
- 6.2.18 `lis_matrix_set_jad` ............................................ 89
- 6.2.19 `lis_matrix_set_bsr` ............................................ 90
- 6.2.20 `lis_matrix_set_bsc` ............................................ 90
- 6.2.21 `lis_matrix_set_vbr` ............................................ 91
- 6.2.22 `lis_matrix_set_coo` ............................................ 92
- 6.2.23 `lis_matrix_set_dns` ............................................ 93
- 6.2.24 `lis_matrix_unset` .............................................. 93

## 6.3 Computing with Vectors and Matrices
- 6.3.1 `lis_vector_swap` .................................................. 94
- 6.3.2 `lis_vector_copy` .................................................. 94
- 6.3.3 `lis_vector_axpy` .................................................. 95
- 6.3.4 `lis_vector_xpay` .................................................. 95
- 6.3.5 `lis_vector_axpyz` ................................................ 96
- 6.3.6 `lis_vector_scale` ................................................ 96
- 6.3.7 `lis_vector_pmul` ................................................ 97
- 6.3.8 `lis_vector_pdiv` ................................................ 97
- 6.3.9 `lis_vector_set_all` ............................................. 98
- 6.3.10 `lis_vector_abs` ................................................ 98
- 6.3.11 `lis_vector_reciprocal` ..................................... 99
- 6.3.12 `lis_vector_conjugate` ...................................... 99
References

A File Formats

A.1 Extended Matrix Market Format ................................. 166
A.2 Harwell-Boeing Format ........................................ 167
A.3 Extended Matrix Market Format for Vectors ..................... 168
A.4 PLAIN Format for Vectors ..................................... 168
Changes from Version 1.0

1. Addition: Support for double-double (quadruple) precision operations.
4. Changes:
   (a) Structure of solvers.
   (b) Arguments of the functions `lis_matrix_create()` and `lis_vector_create()`.
   (c) Notation of command line options.

Changes from Version 1.1

2. Addition: Support for 64bit integers.
3. Changes:
   (a) Names of the functions `lis_output_residual_history()` and `lis_get_residual_history()` to `lis_solver_output_rhistory()` and `lis_solver_get_rhistory()`, respectively.
   (b) Origin of the Fortran interfaces `lis_vector_set_value()` and `lis_vector_get_value()` to 1.
   (c) Origin of the Fortran interface `lis_vector_set_size()` to 1.
   (d) Name of the precision flag `-precision` to `-f`.
4. Change: Specification of the function `lis_solve_kernel()` to return the residual computed by the function `lis_solve_execute()`.
5. Changes: Specifications of integer types:
   (a) Replacement: The type of integer in C programs with `LIS_INT`, which is equivalent to `int` by default. If the preprocessor macro `__LONGLONG` is defined, it is replaced with `long long int`.
   (b) Replacement: The type of integer in Fortran programs with `LIS_INTEGER`, which is equivalent to `integer` by default. If the preprocessor macro `__LONGLONG` is defined, it is replaced with `integer*8`.
6. Change: Names of the matrix storage formats CRS (Compressed Row Storage) and CCS (Compressed Column Storage) to CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column), respectively.
7. Change: Names of the functions `lis_get_solvername()`, `lis_get_preconname()`, and `lis_get_esolvername()` to `lis_solver_get_solvername()`, `lis_solver_get_preconname()`, and `lis_esolver_get_esolvername()`, respectively.
Changes from Version 1.2

1. Addition: Support for nmake.
2. Change: Name of the file lis_config_win32.h to lis_config_win.h.
3. Change: Name of the matrix storage format JDS (Jagged Diagonal Storage) to JAD (Jagged Diagonal).
4. Change: Names of the functions lis_fscan_double() and lis_bswap_double() to lis_fscan_scalar() and lis_bswap_scalar(), respectively.

Changes from Version 1.3

1. Addition: Support for long double (quadruple) precision operations.
2. Addition: Support for pointer operations in Fortran.
3. Change: Name of the members residual of the structs LIS_SOLVER and LIS_ESOLVER to rhistory.
4. Change: Names of the members iters and iters2 of the structs LIS_SOLVER and LIS_ESOLVER to iter and iter2, respectively.
5. Change: Names of the functions lis_solver_get_iters(), lis_solver_get_itersex(), lis_esolver_get_iters(), and lis_esolver_get_itersex() to lis_solver_get_iter(), lis_solver_get_iterex(), lis_esolver_get_iter(), and lis_esolver_get_iterex(), respectively.
6. Change: Names of the members *times of the structs LIS_SOLVER and LIS_ESOLVER to *time, respectively.
7. Addition: Member intvalue to the struct LIS_VECTOR.
8. Change: Specifications of the functions lis_output_vector*() and lis_output_mm_vec() to allow integer data.
9. Change: Names of the functions lis_matrix_scaling*() to lis_matrix_scale*(), respectively.
10. Change: Names of the functions lis_array_dot2() and lis_array_invGauss() to lis_array_dot() and lis_array_ge(), respectively.

Changes from Version 1.4

1. Addition: Support for array operations.
3. Change: Specification of the function lis_arrayqr() to return the number of iterations and error of the QR algorithm.
4. Change: Names of the functions lis_array_matvec2() and lis_array_matmat2() to lis_array_matvecns() and lis_array_matmatns(), respectively.
5. Change: Names of the preprocessor macros _LONGLONG and LONGLONG to _LONG_LONG and LONG_LONG, respectively.
Changes from Version 1.5
1. Addition: Support for complex arithmetic.

Changes from Version 1.6
3. Change: Sign of shift for eigensolvers in accordance with custom.

Changes from Version 1.7
2. Names of the flags -ssor_w and -hybrid_w to -ssor_omega and -hybrid_omega, respectively.

Changes from Version 1.8
1. Addition: Support for linear solvers generalized for complex arithmetic.
3. Change: Definition of the Hermition inner product \((x, y) = y^H x\) to \((x, y) = x^H y\).
5. Change: Name of the function lis_matrix_shift_general() to lis_matrix_shift_matrix().

Changes from Version 2.0
1. Addition: Support for operations on specific eigenpairs.
2. Change: Name of the function lis_matrix_shift_general() to lis_matrix_shift_matrix().
1 Introduction

Lis (Library of Iterative Solvers for linear systems, pronounced [lis]) is a parallel software library to solve discretized linear equations

\[ Ax = b \]

and eigenvalue problems

\[ Ax = \lambda Bx \]

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

<table>
<thead>
<tr>
<th>Linear Solvers</th>
<th>Table 1: Linear Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BiCG[4, 5, 6]</td>
<td>BiCR[7]</td>
</tr>
<tr>
<td>CGS[8]</td>
<td>CRS[9]</td>
</tr>
<tr>
<td>BiCGSTAB[10]</td>
<td>BiCRSTAB[9]</td>
</tr>
<tr>
<td>BiCGSTAB(l)[14]</td>
<td>TFQMR[15]</td>
</tr>
<tr>
<td>Jacobi[16]</td>
<td>Orthomin(m)[17]</td>
</tr>
<tr>
<td>Gauss-Seidel[18, 19]</td>
<td>GMRES(m)[20]</td>
</tr>
<tr>
<td>SOR[21, 22]</td>
<td>FGMRES(m)[23]</td>
</tr>
<tr>
<td>IDR(s)[24]</td>
<td>MINRES[25]</td>
</tr>
<tr>
<td>COCG[26]</td>
<td>COCR[27]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eigensolvers</th>
<th>Table 2: Eigensolvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power[28]</td>
<td></td>
</tr>
<tr>
<td>Inverse[29]</td>
<td></td>
</tr>
<tr>
<td>Rayleigh Quotient[30]</td>
<td></td>
</tr>
<tr>
<td>CG[31]</td>
<td></td>
</tr>
<tr>
<td>CR[32]</td>
<td></td>
</tr>
<tr>
<td>Subspace[33]</td>
<td></td>
</tr>
<tr>
<td>Lanczos[34]</td>
<td></td>
</tr>
<tr>
<td>Arnoldi[35]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditioners</th>
<th>Table 3: Preconditioners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi[36]</td>
<td></td>
</tr>
<tr>
<td>SSOR[36]</td>
<td></td>
</tr>
<tr>
<td>ILU(k)[37, 38]</td>
<td></td>
</tr>
<tr>
<td>ILUT[39, 40]</td>
<td></td>
</tr>
<tr>
<td>Crout ILU[40, 41]</td>
<td></td>
</tr>
<tr>
<td>I+S[42]</td>
<td></td>
</tr>
<tr>
<td>SA-AMG[43]</td>
<td></td>
</tr>
<tr>
<td>Hybrid[44]</td>
<td></td>
</tr>
<tr>
<td>SAINV[45]</td>
<td></td>
</tr>
<tr>
<td>Additive Schwarz[46, 47]</td>
<td></td>
</tr>
<tr>
<td>User defined</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix Storage Formats</th>
<th>Table 4: Matrix Storage Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Sparse Row</td>
<td>(CSR)</td>
</tr>
<tr>
<td>Compressed Sparse Column</td>
<td>(CSC)</td>
</tr>
<tr>
<td>Modified Compressed Sparse Row</td>
<td>(MSR)</td>
</tr>
<tr>
<td>Diagonal</td>
<td>(DIA)</td>
</tr>
<tr>
<td>Ellpack-Itpack Generalized Diagonal</td>
<td>(ELL)</td>
</tr>
<tr>
<td>Jagged Diagonal</td>
<td>(JAD)</td>
</tr>
<tr>
<td>Block Sparse Row</td>
<td>(BSR)</td>
</tr>
<tr>
<td>Block Sparse Column</td>
<td>(BSC)</td>
</tr>
<tr>
<td>Variable Block Row</td>
<td>(VBR)</td>
</tr>
<tr>
<td>Coordinate</td>
<td>(COO)</td>
</tr>
<tr>
<td>Dense</td>
<td>(DNS)</td>
</tr>
</tbody>
</table>
2 Installation

This section describes the instructions for installing and testing Lis.

2.1 System Requirements

Installing Lis requires a C compiler. If you wish to use the Fortran interface, a Fortran compiler is needed, and the algebraic multigrid preconditioner requires a Fortran 90 compiler. For parallel computing environments, an OpenMP\cite{89} or MPI-1\cite{83} library is necessary\cite{48, 49}. Lis supports both the Harwell-Boeing\cite{75} and Matrix Market\cite{79} formats for importing and exporting user data. Lis has been tested in the environments listed in Table 5 (see also Table 7).

<table>
<thead>
<tr>
<th>C Compilers</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel C/C++ Compiler 7.0, 8.0, 9.1, 10.1, 11.1, 12.1, 14.0, 16.0, 17.0, 18.0, 19.0, 2021.8.0</td>
<td>Linux, Windows</td>
</tr>
<tr>
<td>IBM XL C/C++ V7.0, 9.0</td>
<td>AIX, Linux</td>
</tr>
<tr>
<td>Sun WorkShop 6, Sun ONE Studio 7, Sun Studio 11, 12</td>
<td>Solaris</td>
</tr>
<tr>
<td>PGI C++ 6.0, 7.1, 10.5, 16.10</td>
<td>Linux</td>
</tr>
<tr>
<td>gcc 3.3, 4.4, 5.4, 6.4, 8.2, 9.3, 10.2, 11.3</td>
<td>Linux, macOS, Windows</td>
</tr>
<tr>
<td>Clang 3.3, 3.4, 3.7, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.1, 12.0</td>
<td>macOS, FreeBSD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran Compilers (Optional)</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Fortran Compiler 8.1, 9.1, 10.1, 11.1, 12.1, 14.0, 16.0, 17.0, 18.0, 19.0, 2021.8.0</td>
<td>Linux, Windows</td>
</tr>
<tr>
<td>IBM XL Fortran V9.1, 11.1</td>
<td>AIX, Linux</td>
</tr>
<tr>
<td>Sun WorkShop 6, Sun ONE Studio 7, Sun Studio 11, 12</td>
<td>Solaris</td>
</tr>
<tr>
<td>PGI Fortran 6.0, 7.1, 10.5, 16.10</td>
<td>Linux</td>
</tr>
<tr>
<td>g77 3.3</td>
<td>Linux, macOS, Windows</td>
</tr>
<tr>
<td>gfortran 4.4, 5.4, 6.4, 8.2, 10.1, 11.3</td>
<td>Linux, macOS, Windows</td>
</tr>
</tbody>
</table>

2.2 Installing on UNIX and Compatible Systems

2.2.1 Extracting Archive

To extract the archive, use the following command, where ($VERSION) represents the version:

```
> unzip lis-($VERSION).zip
```

This will create a directory named lis-($VERSION) along with its subfolders as shown in Figure 1.

2.2.2 Configuring Source Tree

Navigate to the directory lis-($VERSION) and run the following command to configure the source tree:

- default: > ./configure
- specify the installation destination: > ./configure --prefix=<install-dir>
Figure 1: Files contained in lis-($VERSION).zip

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

2.2.3 Compiling

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

```
> make
```

To verify that the library has been built successfully, enter

```
> make check
```

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

```
Default

    /x13 /x10

matrix size = 100 x 100 (460 nonzero entries)
initial vector x : all components set to 0
precision : double
linear solver : BiCG
preconditioner : none
convergence condition : $||b-Ax||_2 \leq 1.0e-12 * ||b-Ax_0||_2$
matrix storage format : CSR
linear solver status : 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 = 6.327297e-15
```
Table 6: Major Configuration Options (see ./configure --help for the complete list)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--enable-omp</td>
<td>Build with OpenMP library</td>
</tr>
<tr>
<td>--enable-mpi</td>
<td>Build with MPI library</td>
</tr>
<tr>
<td>--enable-fortran</td>
<td>Enable FORTRAN 77 compatible interface</td>
</tr>
<tr>
<td>--enable-f90</td>
<td>Enable Fortran 90 compatible interface</td>
</tr>
<tr>
<td>--enable-saamg</td>
<td>Enable SA-AMG preconditioner</td>
</tr>
<tr>
<td>--enable-quad</td>
<td>Enable double-double (quadruple) precision support</td>
</tr>
<tr>
<td>--enable-longdouble</td>
<td>Enable long double (quadruple) precision support</td>
</tr>
<tr>
<td>--enable-longlong</td>
<td>Enable 64bit integer support</td>
</tr>
<tr>
<td>--enable-complex</td>
<td>Enable complex scalar support</td>
</tr>
<tr>
<td>--enable-debug</td>
<td>Enable debugging</td>
</tr>
<tr>
<td>--enable-shared</td>
<td>Enable dynamic linking</td>
</tr>
<tr>
<td>--enable-gprof</td>
<td>Enable profiling</td>
</tr>
<tr>
<td>--disable-test</td>
<td>Disable building test programs</td>
</tr>
<tr>
<td>--prefix=&lt;install-dir&gt;</td>
<td>Specify installation destination</td>
</tr>
<tr>
<td>TARGET=&lt;target&gt;</td>
<td>Specify computing environment</td>
</tr>
<tr>
<td>CC=&lt;c_compiler&gt;</td>
<td>Specify C compiler</td>
</tr>
<tr>
<td>CFLAGS=&lt;c_flags&gt;</td>
<td>Specify options for C compiler</td>
</tr>
<tr>
<td>F77=&lt;f77_compiler&gt;</td>
<td>Specify FORTRAN 77 compiler</td>
</tr>
<tr>
<td>F77FLAGS=&lt;f77_flags&gt;</td>
<td>Specify options for FORTRAN 77 compiler</td>
</tr>
<tr>
<td>FC=&lt;f90_compiler&gt;</td>
<td>Specify Fortran 90 compiler</td>
</tr>
<tr>
<td>FCFLAGS=&lt;f90_flags&gt;</td>
<td>Specify options for Fortran 90 compiler</td>
</tr>
<tr>
<td>LDFLAGS=&lt;ld_flags&gt;</td>
<td>Specify link options</td>
</tr>
</tbody>
</table>

--enable-omp

max number of threads = 32
number of threads = 2
matrix size = 100 x 100 (460 nonzero entries)

initial vector x : all components set to 0
precision : double
linear solver : BiCG
preconditioner : none
convergence condition : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status : 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 = 6.221213e-15
Table 7: Examples of Targets (see lis-$\text{VERSION}$/configure.ac for details)

<table>
<thead>
<tr>
<th>&lt;target&gt;</th>
<th>Equivalent options</th>
</tr>
</thead>
<tbody>
<tr>
<td>cray_xt3_cross</td>
<td>./configure CC=cc FC=f77 CFLAGS=&quot;-O3 -B -fastsse -tp k8-64&quot; FCFLAGS=&quot;-03 -fastsse -tp k8-64 -Mpreprocess&quot; FCLDFLAGS=&quot;-Mnomain&quot; ac_cv_sizeof_void_p=8 cross_compiling=yes ax_f77_mangling=&quot;lower case, no underscore, extra underscore&quot;</td>
</tr>
<tr>
<td>fujitsu_fx10_cross</td>
<td>./configure CC=fccpx FC=frtpx CFLAGS=&quot;-Kfast,ocl,preex&quot; FCFLAGS=&quot;-Kfast,ocl,preex -Cpp -fs&quot; FCLDFLAGS=&quot;-mlcmain=main&quot; ac_cv_sizeof_void_p=8 cross_compiling=yes ax_f77_mangling=&quot;lower case, underscore, no extra underscore&quot;</td>
</tr>
<tr>
<td>hitachi_sr16k</td>
<td>./configure CC=cc FC=f90 CFLAGS=&quot;-Os -noparallel&quot; FCFLAGS=&quot;-Oss -nosequence&quot; FCLDFLAGS=&quot;-lf90s&quot; ac_cv_sizeof_void_p=8 ax_f77_mangling=&quot;lower case, underscore, no extra underscore&quot;</td>
</tr>
<tr>
<td>ibm_bgl_cross</td>
<td>./configure CC=blrta_xlc FC=blrta_xlf90 CFLAGS=&quot;-O3 -qarch=440d -qtune=440 -qstrict&quot; FCFLAGS=&quot;-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90&quot; ac_cv_sizeof_void_p=4 cross_compiling=yes ax_f77_mangling=&quot;lower case, no underscore, no extra underscore&quot;</td>
</tr>
<tr>
<td>intel_mic_cross</td>
<td>./configure CC=icc F77=ifort FC=ifort MPICCC=mipicc MPIFF77=mipifort MPIFC=mipifort CFLAGS=&quot;-mmic&quot; FFLAGS=&quot;-mmic&quot; FCLDFLAGS=&quot;-mmic&quot; LDFLAGS=&quot;-mmic&quot; cross_compiling=yes host=x86_64-pc-linux-gnu host_alias=x86_64-linux-gnu host_cpu=x86_64 host_os=linux-gnu host_vendor=pc target=k1om-mpss-linux-gnu target_alias=k1om-mpss-linux target_cpu=k1om target_os=linux-gnu target_vendor=mpss</td>
</tr>
<tr>
<td>nec_ax9_cross</td>
<td>./configure CC=sxmpic++ FC=sxmpif90 AR=xsar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes ax_f77_mangling=&quot;lower case, no underscore, extra underscore&quot;</td>
</tr>
</tbody>
</table>
---enable-mpi---

number of processes = 2  
matrix size = 100 x 100 (460 nonzero entries)  
initial vector x : all components set to 0  
precision : double  
linear solver : BiCG  
preconditioner : none  
convergence condition : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2  
matrix storage format : CSR  
linear solver status : 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 = 6.221213e-15

2.2.4 Installing

In the directory lis-($VERSION), enter  
> make install 

which copies the files to the destination directory as follows:

($INSTALLDIR)  
+bin  
|  +lsolve esolve esolver gesolve gesolver hpcg_kernel hpcg_spmvtest spmvtest*  
+include  
|  +lis_config.h lis.h lisf.h  
+lib  
|  +liblis.a  
+share  
   +doc/lis examples/lis man  

lis_config.h is the header file required to build the library, and lis.h and lisf.h are the header files required by the C and Fortran compilers, respectively. liblis.a is the library. To ensure that the library has been installed successfully, enter  
> make installcheck  
in lis-($VERSION). This runs a test script using the executable files installed in examples/lis.test1, esolve5, esolve5b, esolve5, test5b, test3b, and spmvtest3b in examples/lis are copied in  
($INSTALLDIR)/bin as lsolve, esolve, esolver, gesolve, gesolver, hpcg_kernel, and hpcg_spmvtest, respectively. examples/lis/spmvtest* are also copied in ($INSTALLDIR)/bin.

To remove the copied files in ($INSTALLDIR), enter  
> make uninstall  
To remove the generated library and executable files in lis-($VERSION), enter  
> make clean  
To remove the configuration files in addition to the other generated files, enter  
> make distclean
2.3 Installing on Windows Systems

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

```bash
> configure.bat
```

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

```bash
> nmake
```

To ensure that the library has been built successfully, enter

```bash
> nmake check
```

The following command copies the library to `($INSTALLDIR)\lib`, the executable files to `($INSTALLDIR)\bin`, the header files to `($INSTALLDIR)\include`, and the PDF documents to `($INSTALLDIR)\doc`, respectively:

```bash
> nmake install
```

To remove the copied files in `($INSTALLDIR)`, enter

```bash
> nmake uninstall
```

To remove the generated library and executables files in `lis-($VERSION)\win`, enter

```bash
> nmake clean
```

To remove the configuration files in addition to the other generated files, enter

```bash
> nmake distclean
```

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

2.4 Testing

Test programs are located in `lis-($VERSION)/test`.

2.4.1 test1

**Usage:** `test1 matrix_filename rhs_setting solution_filename rhistory_filename [options]`

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

- `0` Use the right-hand side vector `b` included in the data file
- `1` Use `b = (1, . . . , 1)^T`
- `2` Use `b = A \times (1, . . . , 1)^T`

`rhs_filename` The filename for the right-hand side vector

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

2.4.2 test2

**Usage:** `test2 m n matrix_type solution_filename rhistory_filename [options]`

This program solves the linear equation `Ax = b`, where the coefficient matrix `A` of size `mn` is a discretized two dimensional Laplacian using the five point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It outputs the solution to `solution_filename` in the extended Matrix Market format and the residual history to
rh**o**r***g****_history**_filename** in the PLAIN format. The right-hand side vector \( b \) is set such that the values of the elements of the solution \( x \) are 1. The values \( m \) and \( n \) represent the numbers of grid points in each dimension. \texttt{test2f.F90} is the Fortran 90 version of \texttt{test2.c}.

2.4.3 test2b

Usage: test2b m n matrix_type solution_filename rhistory_filename [options]

This program solves the linear equation \( Ax = b \), where the coefficient matrix \( A \) of size \( mn \) is a discretized two dimensional Laplacian using the nine point central difference scheme, with the coefficient matrix in the storage format specified by \texttt{matrix_type} and the solver specified by \texttt{options}. It outputs the solution to \texttt{solution_filename} in the extended Matrix Market format and the residual history to \texttt{rhistory_filename} in the PLAIN format. The right-hand side vector \( b \) is set such that the values of the elements of the solution \( x \) are 1. The values \( m \) and \( n \) represent the numbers of grid points in each dimension.

2.4.4 test3

Usage: test3 l m n matrix_type solution_filename rhistory_filename [options]

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

2.4.5 test3b

Usage: test3b l m n matrix_type solution_filename rhistory_filename [options]

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

2.4.6 test3c

Usage: test3c l m n step [options]

This program solves the linear equation \( Ax = b \) for \texttt{step} steps, where the coefficient matrix \( A \) of size \( lmn \) is a discretized three dimensional Laplacian using the seven point central difference scheme, with the coefficient matrix in the CSR format and the solver specified by \texttt{options}. The right-hand side vector \( b \) is set such that the values of the elements of the solution \( x \) are 1. The values of the elements of the matrix and the right-hand side vector are updated at every step. The values \( l \), \( m \) and \( n \) represent the numbers of grid points in each dimension.
2.4.7 test4

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

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

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

2.4.8 test5

Usage: test5 n gamma [options]

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

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

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

2.4.9 test6

Usage: test6 m n

test6.c is the array version of test2.c. This program solves the linear equation $Ax = b$ using the direct method, where the coefficient matrix $A$ of size $mn$ is a discretized two dimensional Laplacian using the five point central difference scheme. The right-hand side vector $b$ is set such that the values of the elements of the solution $x$ are 1. The values $m$ and $n$ represent the numbers of grid points in each dimension. test6f.F90 is the Fortran 90 version of test6.c.

2.4.10 test7

Usage: test7

This program show examples of complex arithmetic. test7f.F is the Fortran version of test7.c.

2.4.11 test8f

Usage: mpiexec -n m test8f

This program solves a nonlinear partial differential equation using the Newton-Raphson method, where the preconditioner is updated separately from the solver (see Subsection 3.7).
2.4.12 etest1

Usage: etest1 matrix_filename evector_filename rhistory_filename [options]

This program inputs the matrix data from matrix_filename and solves the standard eigenvalue problem $Ax = \lambda x$ with the solver specified by options. It outputs the specified eigenvalue to the standard output, the associated eigenvector to evector_filename in the extended Matrix Market format, and the residual history to rhistory_filename in the PLAIN format. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filename. etest1f.F is the Fortran version of etest1.c. See etest5 for obtaining multiple eigenpairs.

2.4.13 getest1

Usage: getest1 matrix_a_filename matrix_b_filename evector_filename rhistory_filename [options]

This program inputs the matrix data from matrix_a_filename and matrix_b_filename, and solves the generalized eigenvalue problem $Ax = \lambda Bx$ with the solver specified by options. It outputs the specified eigenvalue to the standard output, the associated eigenvector to evector_filename in the extended Matrix Market format, and the residual history to rhistory_filename in the PLAIN format. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filenames. See getest5 for obtaining multiple eigenpairs.

2.4.14 etest2

Usage: etest2 m n matrix_type evector_filename rhistory_filename [options]

This program solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix $A$ of size $mn$ is a discretized two dimensional Laplacian 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 specified eigenvalue to the standard output, the associated eigenvector to evector_filename in the extended Matrix Market format, and the residual history to rhistory_filename in the PLAIN format. The values $m$ and $n$ represent the numbers of grid points in each dimension.

2.4.15 etest3

Usage: etest3 l m n matrix_type evector_filename rhistory_filename [options]

This program solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix $A$ of size $lmn$ is a discretized three dimensional Laplacian 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 specified eigenvalue to the standard output, the associated eigenvector to evector_filename in the extended Matrix Market format, and the residual history to rhistory_filename in the PLAIN format. The values $l$, $m$, and $n$ represent the numbers of grid points in each dimension. See etest6 for obtaining multiple eigenpairs.

2.4.16 etest4

Usage: etest4 n [options]

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

\[ A = \begin{bmatrix}
2 & -1 \\
-1 & 2 & -1 \\
& & \ddots & \ddots \\
& & -1 & 2 & -1 \\
& & & & -1 & 2
\end{bmatrix} \]

of size \( n \times n \). \texttt{etest4f.F} is the Fortran version of \texttt{etest4.c}.

2.4.17 \texttt{etest5}

Usage: \texttt{etest5 matrix\_filename evals\_filename evectors\_filename residuals\_filename iters\_filename [options]}

This program inputs the matrix data from \texttt{matrix\_filename} and solves the standard eigenvalue problem \( Ax = \lambda x \) with the solver specified by \texttt{options}. It outputs the specified number of eigenvalues, the number of which is given by option \texttt{-ss}, to \texttt{evals\_filename} and the associated eigenvectors, residual norms, and numbers of iterations to \texttt{evectors\_filename}, \texttt{residuals\_filename}, and \texttt{iters\_filename} respectively in the extended Matrix Market format. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filename.

2.4.18 \texttt{etest5b}

Usage: \texttt{etest5b matrix\_filename evals\_filename [options]}

This program inputs the matrix data from \texttt{matrix\_filename} and solves the standard eigenvalue problem \( Ax = \lambda x \) with the solver specified by \texttt{options}. It outputs the specified number of Ritz values, the number of which is given by option \texttt{-ss}, to \texttt{evals\_filename} in the extended Matrix Market format. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filename.

2.4.19 \texttt{getest5}

Usage: \texttt{getest5 matrix\_a\_filename matrix\_b\_filename evals\_filename evectors\_filename residuals\_filename iters\_filename [options]}

This program inputs the matrix data from \texttt{matrix\_a\_filename} and \texttt{matrix\_b\_filename}, and solves the generalized eigenvalue problem \( Ax = \lambda Bx \) with the solver specified by \texttt{options}. It outputs the specified number of eigenvalues, the number of which is given by option \texttt{-ss}, to \texttt{evals\_filename} and the associated eigenvectors, residual norms, and numbers of iterations to \texttt{evectors\_filename}, \texttt{residuals\_filename}, and \texttt{iters\_filename} respectively in the extended Matrix Market format. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filenames.

2.4.20 \texttt{getest5b}

Usage: \texttt{getest5b matrix\_a\_filename matrix\_b\_filename evals\_filename [options]}

This program inputs the matrix data from \texttt{matrix\_a\_filename} and \texttt{matrix\_b\_filename}, and solves the generalized eigenvalue problem \( Ax = \lambda Bx \) with the solver specified by \texttt{options}. It outputs the specified number of eigenvalues, the number of which is given by option \texttt{-ss}, to \texttt{evals\_filename} in the extended Matrix Market format. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filenames.
2.4.21 etest6

Usage: etest6 l m n matrix_type evals_filename evectors_filename residuals_filename iters_filename [options]

This program solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix $A$ of size $lmn$ is a discretized three dimensional Laplacian 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 specified number of eigenvalues, the number of which is given by option `-ss`, to `evals_filename` and the associated eigenvectors and residual norms to `evectors_filename`, `residuals_filename`, and `iters_filename` respectively in the extended Matrix Market format. The values $l$, $m$ and $n$ represent the numbers of grid points in each dimension.

2.4.22 etest7

Usage: etest7 m n

etest7.c is the array version of etest2.c. This program solves the eigenvalue problem $Ax = \lambda x$ using the QR algorithm, where the coefficient matrix $A$ of size $mn$ is a discretized two dimensional Laplacian using the five point central difference scheme. The values $m$ and $n$ represent the numbers of grid points in each dimension.

2.4.23 spmvtest1

Usage: spmvtest1 n iter [matrix_type]

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

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

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

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

2.4.24 spmvtest2

Usage: spmvtest2 m n iter [matrix_type]

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

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

The values $m$ and $n$ represent the numbers of grid points in each dimension.
2.4.25  spmvtest2b

Usage: spmvtest2b m n iter [matrix_type]

This program computes the matrix-vector multiply of a discretized two dimensional Laplacian of size \(mn\) using the nine 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 \texttt{matrix_type}:

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

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

2.4.26  spmvtest3

Usage: spmvtest3 l m n iter [matrix_type]

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

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

2.4.27  spmvtest3b

Usage: spmvtest3b l m n iter [matrix_type]

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

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

2.4.28  spmvtest4

Usage: spmvtest4 matrix_filename_list iter [block]

This program inputs the matrix data from the files listed in \texttt{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. Both the Matrix Market format and the Harwell-Boeing format are supported for the matrix filenames. If necessary, the block size of the BSR and BSC formats can be specified by \texttt{block}.

2.4.29  spmvtest5

Usage: spmvtest5 matrix_filename matrix_type iter [block]

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

The current version has the following limitations:

- **Matrix storage formats**
  - The VBR format does not support the multiprocessing environment.
  - The SA-AMG preconditioner supports only the CSR format.
  - In the multiprocessing environment, the CSR is the only accepted format for user defined arrays.

- **Double-double (quadruple) precision operations** (see Section 4)
  - The Jacobi, Gauss-Seidel, SOR, IDR(s), COCG, and COCR methods do not support the double-double precision operations.
  - The eigensolvers do not support the double-double precision operations.
  - The Jacobi, Gauss-Seidel and SOR methods in the hybrid preconditioner do not support the double-double precision operations.
  - The I+S and SA-AMG preconditioners do not support the double-double precision operations.
  - The double-double precision operations does not support complex arithmetic.
  - The double-double precision operations cannot be combined with the long-double precision operations.

- **Preconditioners**
  - The algorithm of the ILU(k) preconditioner is based on the localized ILU preconditioning[38], which factorizes the block diagonal elements in parallel. Note that the convergence behavior approaches to that of the Jacobi preconditioner as the number of threads or processes increases.
  - If a preconditioner other than the Jacobi or SSOR is selected and matrix $A$ is not in the CSR format, a new matrix is created in the CSR format for preconditioning.
  - The SA-AMG preconditioner does not support the BiCG method for unsymmetric matrices.
  - The SA-AMG preconditioner does not support multithreading.
  - The SA-AMG preconditioner does not support complex arithmetic.
  - The assembly of the matrices in the SAINV preconditioner is not parallelized.
  - The user defined preconditioner cannot be used.

- **Eigen solvers**
  - To compute complex eigenvalues, complex arithmetic must be enabled. Therefore, when computing eigenvalues of unsymmetric matrices, complex arithmetic must always be enabled.
3 Basic Operations

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

- Initialization
- Matrix creation
- Vector creation
- Solver creation
- Value assignment for matrices and vectors
- Solver assignment
- Solver execution
- Finalization

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

- C       #include "lis.h"
- Fortran #include "lisf.h"

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

3.1 Initializing and Finalizing

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

C

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

Fortran

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

Initializing

For initializing, the following functions are used:

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

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

The default type of the integer in the C programs is LIS_INT, which is equivalent to int. If the preprocessor macro _LONG_LONG is defined, it is replaced with long long int. The default type of the integer in the Fortran programs is LIS_INTEGER, which is equivalent to integer. If the preprocessor
macro LONG_LONG is defined, it is replaced with integer*8.

Finalizing
For finalizing, the following functions are used:

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

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

\[
v = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix}
\]

(3.1)

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

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

C (for serial and multithreaded environments)

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

C (for multiprocessing environment)

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

```fortran
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: do i=1,n
7: call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr)
8: enddo
```

Fortran (for multiprocessing environment)

```fortran
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
```

Creating Vectors

To create vector \( v \), the following functions are used:

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

- Fortran subroutine
  ```fortran
  lis_vector_create(LIS_Comm comm, LIS_VECTOR v, LIS_INTEGER ierr)
  ```

For the example program above, \( \text{comm} \) must be replaced with the MPI communicator. For the serial and multithreaded environments, the value of \( \text{comm} \) is ignored.

Assigning Sizes

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

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

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

Either \( \text{local}_n \) or \( \text{global}_n \) must be provided.

For the serial and multithreaded environments, \( \text{local}_n \) is equal to \( \text{global}_n \). Therefore, both \( \text{lis_vector_set_size}(v,n,0) \) and \( \text{lis_vector_set_size}(v,0,n) \) create a vector of size \( n \).

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

Assigning Values

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

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

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

For the multiprocessing environment, the \( i \)-th row of the global vector must be specified. Either
LIS_INS_VALUE : \( v[i] = value \), or
LIS_ADD_VALUE : \( v[i] = v[i] + value \)

must be provided for flag.

**Duplicating Vectors**

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

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

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

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

**Destroying Vectors**

To destroy the vector, the following functions are used:

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

### 3.3 Operating Matrices

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

\[
A = \begin{pmatrix}
2 & 1 \\
1 & 2 & 1 \\
1 & 2 \\
1 & 2 \\
\end{pmatrix}
\]

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

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

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

Programs to create matrix \( A \) in the CSR format are as follows, where the number of processing elements for the multiprocessor environment is assumed to be two:
C (for serial and multithreaded environments)

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

C (for multiprocessing environment)

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

Fortran (for serial and multithreaded environments)

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

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

Creating Matrices
To create matrix $A$, the following functions are used:

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

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

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

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

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

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

  Either local_n or global_n must be provided.

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

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

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

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

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

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

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

must be provided for the parameter flag.

### Assigning Storage Formats

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

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

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

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

<table>
<thead>
<tr>
<th>Storage format</th>
<th>matrix_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Sparse Row (CSR)</td>
<td>{LIS_MATRIX_CSR</td>
</tr>
<tr>
<td>Compressed Sparse Column (CSC)</td>
<td>{LIS_MATRIX_CSC</td>
</tr>
<tr>
<td>Modified Compressed Sparse Row (MSR)</td>
<td>{LIS_MATRIX_MSR</td>
</tr>
<tr>
<td>Diagonal (DIA)</td>
<td>{LIS_MATRIX_DIA</td>
</tr>
<tr>
<td>Ellpack-Itpack Generalized Diagonal (ELL)</td>
<td>{LIS_MATRIX_ELL</td>
</tr>
<tr>
<td>Jagged Diagonal (JAD)</td>
<td>{LIS_MATRIX_JAD</td>
</tr>
<tr>
<td>Block Sparse Row (BSR)</td>
<td>{LIS_MATRIX_BSR</td>
</tr>
<tr>
<td>Block Sparse Column (BSC)</td>
<td>{LIS_MATRIX_BSC</td>
</tr>
<tr>
<td>Variable Block Row (VBR)</td>
<td>{LIS_MATRIX_VBR</td>
</tr>
<tr>
<td>Coordinate (COO)</td>
<td>{LIS_MATRIX_COO</td>
</tr>
<tr>
<td>Dense</td>
<td>{LIS_MATRIX_DNS</td>
</tr>
</tbody>
</table>

### Assembling Matrices

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

- **C**
  
  ```c
  LIS_INT lis_matrix_assemble(LIS_MATRIX A)
  ```

- **Fortran subroutine**
  
  ```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**
  
  ```c
  LIS_INT lis_matrix_destroy(LIS_MATRIX A)
  ```

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

### Method 2: Define Arrays in a Specific Storage Format Directly

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

Programs to create matrix \( A \) in the CSR format are as follows, where the number of processing elements for the multiprocessing environment is assumed to be two:
C (for serial and multithreaded environments)

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

C (for multiprocessing environment)

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

Associating Arrays

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

- C
  
  \[
  \text{C} \quad \text{LIS_INT \ lis_matrix_set_csr(LIS_INT nnz, LIS_INT ptr[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)}
  \]

- Fortran subroutine
  
  \[
  \text{Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER ptr(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)}
  \]

Method 3: Read Matrix and Vector Data from External Files
Programs to read matrix \( A \) in Equation (3.2) in the CSR format and vector \( b \) in Equation (3.1) from an external file are as follows:

C (for serial, multithreaded and multiprocessing environments)

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

Fortran (for serial, multithreaded and multiprocessing environments)

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

The content of the destination file \( \text{matvec.mtx} \) is:

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

Reading from External Files

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

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

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

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

- The Matrix Market format
- The Harwell-Boeing format

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

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

Fortran (for serial, multithreaded and multiprocessing environments)

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

Creating Solvers

To create a solver, the following functions are used:

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

Specifying Options

To specify options, the following functions are used:

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

or

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

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

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

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

<table>
<thead>
<tr>
<th>Solver</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>`-i {cg</td>
<td>1}`</td>
</tr>
<tr>
<td>BiCG</td>
<td>`-i {bicg</td>
<td>2}`</td>
</tr>
<tr>
<td>CGS</td>
<td>`-i {cgs</td>
<td>3}`</td>
</tr>
<tr>
<td>BiCGSTAB</td>
<td>`-i {bicgstab</td>
<td>4}`</td>
</tr>
<tr>
<td>BiCGSTAB(l)</td>
<td>`-i {bicgstabl</td>
<td>5}`</td>
</tr>
<tr>
<td>GPBiCG</td>
<td>`-i {gpbicg</td>
<td>6}`</td>
</tr>
<tr>
<td>TFQMR</td>
<td>`-i {tfqmr</td>
<td>7}`</td>
</tr>
<tr>
<td>Orthomin(m)</td>
<td>`-i {orthomin</td>
<td>8}`</td>
</tr>
<tr>
<td>GMRES(m)</td>
<td>`-i {gmres</td>
<td>9}`</td>
</tr>
<tr>
<td>Jacobi</td>
<td>`-i {jacobi</td>
<td>10}`</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>`-i {gs</td>
<td>11}`</td>
</tr>
<tr>
<td>SOR</td>
<td>`-i {sor</td>
<td>12}`</td>
</tr>
<tr>
<td>BiCGSafe</td>
<td>`-i {bicgsafe</td>
<td>13}`</td>
</tr>
<tr>
<td>CR</td>
<td>`-i {cr</td>
<td>14}`</td>
</tr>
<tr>
<td>BiCR</td>
<td>`-i {bicr</td>
<td>15}`</td>
</tr>
<tr>
<td>CRS</td>
<td>`-i {crs</td>
<td>16}`</td>
</tr>
<tr>
<td>BiCRSTAB</td>
<td>`-i {bicrstab</td>
<td>17}`</td>
</tr>
<tr>
<td>GPBiCR</td>
<td>`-i {gpbicr</td>
<td>18}`</td>
</tr>
<tr>
<td>BiCRSafe</td>
<td>`-i {bicrsafe</td>
<td>19}`</td>
</tr>
<tr>
<td>FGMRES(m)</td>
<td>`-i {fgmres</td>
<td>20}`</td>
</tr>
<tr>
<td>IDR(s)</td>
<td>`-i {idrs</td>
<td>21}`</td>
</tr>
<tr>
<td>IDR(1)</td>
<td>`-i {idr1</td>
<td>22}`</td>
</tr>
<tr>
<td>MINRES</td>
<td>`-i {minres</td>
<td>23}`</td>
</tr>
<tr>
<td>COCG</td>
<td>`-i {cocg</td>
<td>24}`</td>
</tr>
<tr>
<td>COCR</td>
<td>`-i {cocr</td>
<td>25}`</td>
</tr>
</tbody>
</table>
## Options for Preconditioners (Default: -p none)

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>-p {none</td>
<td>0}</td>
</tr>
<tr>
<td>Jacobi</td>
<td>-p {jacobi</td>
<td>1}</td>
</tr>
<tr>
<td>ILU(k)</td>
<td>-p {ilu</td>
<td>2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-ssor_omega [1.0] The relaxation coefficient $\omega$ ($0 &lt; \omega &lt; 2$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ILU(k) -p {ilu</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-ssor_omega [1.0] The relaxation coefficient $\omega$ ($0 &lt; \omega &lt; 2$)</td>
</tr>
<tr>
<td>SSOR</td>
<td>-p {ssor</td>
<td>3}</td>
</tr>
<tr>
<td>Hybrid</td>
<td>-p {hybrid</td>
<td>4}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_maxiter [25] The maximum number of iterations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_tol [1.0e-3] The convergence tolerance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_ell [2] The degree $t$ of the BiCGSTAB($l$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_restart [40] The restart values of the GMRES and Orthomin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I+S -p {is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-is_m [3] The parameter $m$ of $I + \alpha S^{(m)}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAINV -p {sainv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-sainv_unsym [false] Select the unsymmetric version (The matrix structure must be symmetric)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-sainv_theta [0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA-AMG -p {saamg</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-saamg_theta [0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Crout ILU -p {iluc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-iluc_rate [5.0] The ratio of the maximum fill-in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ILUT -p {ilut</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-iluc_rate [5.0] The ratio of the maximum fill-in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Additive -adds true The number of iterations</td>
</tr>
<tr>
<td>Schwarz</td>
<td></td>
<td>-adds_iter [1] The number of iterations</td>
</tr>
</tbody>
</table>

---

30
Other Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-maxiter [1000]</td>
<td>The maximum number of iterations</td>
</tr>
<tr>
<td>-tol [1.0e-12]</td>
<td>The convergence tolerance tol</td>
</tr>
<tr>
<td>-tol_w [1.0]</td>
<td>The convergence tolerance tol_w</td>
</tr>
<tr>
<td>-print [0]</td>
<td>The output of the residual history</td>
</tr>
<tr>
<td>-print {none</td>
<td>0}</td>
</tr>
<tr>
<td>-print {mem</td>
<td>1}</td>
</tr>
<tr>
<td>-print {out</td>
<td>2}</td>
</tr>
<tr>
<td>-print {all</td>
<td>3}</td>
</tr>
<tr>
<td>-scale [0]</td>
<td>The scaling</td>
</tr>
<tr>
<td>-scale {none</td>
<td>0}</td>
</tr>
<tr>
<td>-scale {jacobi</td>
<td>1}</td>
</tr>
<tr>
<td></td>
<td>($D$ represents the diagonal of $A = (a_{ij})$)</td>
</tr>
<tr>
<td>-scale {symm_diag</td>
<td>2}</td>
</tr>
<tr>
<td></td>
<td>($D^{-1/2}$ represents the diagonal matrix with $1/\sqrt{a_{ii}}$ as the diagonal)</td>
</tr>
<tr>
<td>-initx_zeros [1]</td>
<td>The behavior of the initial vector $x_0$</td>
</tr>
<tr>
<td>-initx_zeros {false</td>
<td>0}</td>
</tr>
<tr>
<td>-initx_zeros {true</td>
<td>1}</td>
</tr>
<tr>
<td>-conv_cond [0]</td>
<td>The convergence condition</td>
</tr>
<tr>
<td>-conv_cond {nrm2_r</td>
<td>0}</td>
</tr>
<tr>
<td>-conv_cond {nrm2_b</td>
<td>1}</td>
</tr>
<tr>
<td>-conv_cond {nrm1_b</td>
<td>2}</td>
</tr>
<tr>
<td>-omp_num_threads [t]</td>
<td>The number of threads</td>
</tr>
<tr>
<td></td>
<td>(t represents the maximum number of threads)</td>
</tr>
<tr>
<td>-storage [0]</td>
<td>The matrix storage format</td>
</tr>
<tr>
<td>-storage_block [2]</td>
<td>The block size of the BSR and BSC formats</td>
</tr>
<tr>
<td>-f [0]</td>
<td>The precision of the linear solver</td>
</tr>
<tr>
<td>-f {double</td>
<td>0}</td>
</tr>
<tr>
<td>-f {quad</td>
<td>1}</td>
</tr>
</tbody>
</table>

### Solving Linear Equations

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

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

- Fortran subroutine
  
  ```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 standard eigenvalue problem $Ax = \lambda x$ with a specified solver is as follows:
Creating Eigensolvers

To create an eigensolver, the following functions are used:

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

Specifying Options

To specify options, the following functions are used:

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

or

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

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

The table below shows the available command line options, where -e {pi|1} means -e pi or -e 1 and -emaxiter [1000] indicates that -emaxiter defaults to 1,000.
### Options for Eigensolvers (Default: \(-e \ cr\))

<table>
<thead>
<tr>
<th>Eigensolver</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>(-e \ {pi</td>
<td>1})</td>
</tr>
<tr>
<td>Inverse</td>
<td>(-e \ {ii</td>
<td>2} -i \ [bicg])</td>
</tr>
<tr>
<td>Rayleigh Quotient</td>
<td>(-e \ {rqi</td>
<td>3} -i \ [bicg])</td>
</tr>
<tr>
<td>CG</td>
<td>(-e \ {cg</td>
<td>4} -i \ [cg])</td>
</tr>
<tr>
<td>CR</td>
<td>(-e \ {cr</td>
<td>5} -i \ [bicg])</td>
</tr>
<tr>
<td>Subspace</td>
<td>(-e \ {si</td>
<td>6} -ss [1])</td>
</tr>
<tr>
<td>Lanczos</td>
<td>(-e \ {li</td>
<td>7} -ss [1])</td>
</tr>
<tr>
<td>Arnoldi</td>
<td>(-e \ {ai</td>
<td>8} -ss [1])</td>
</tr>
<tr>
<td>Generalized Power</td>
<td>(-e \ {gpi</td>
<td>9} -i \ [bicg])</td>
</tr>
<tr>
<td>Generalized Inverse</td>
<td>(-e \ {gii</td>
<td>10} -i \ [bicg])</td>
</tr>
<tr>
<td>Generalized Rayleigh Quotient</td>
<td>(-e \ {grqi</td>
<td>11} -i \ [bicg])</td>
</tr>
<tr>
<td>Generalized CG</td>
<td>(-e \ {gcg</td>
<td>12} -i \ [cg])</td>
</tr>
<tr>
<td>Generalized CR</td>
<td>(-e \ {gcr</td>
<td>13} -i \ [bicg])</td>
</tr>
<tr>
<td>Generalized Subspace</td>
<td>(-e \ {gsi</td>
<td>14} -ss [1])</td>
</tr>
<tr>
<td>Generalized Lanczos</td>
<td>(-e \ {gli</td>
<td>15} -ss [1])</td>
</tr>
<tr>
<td>Generalized Arnoldi</td>
<td>(-e \ {gai</td>
<td>16} -ss [1])</td>
</tr>
</tbody>
</table>

### Options for Preconditioners (Default: \(-p \ none\))

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>(-p \ {none</td>
<td>0})</td>
</tr>
<tr>
<td>Jacobi</td>
<td>(-p \ {jacobi</td>
<td>1})</td>
</tr>
<tr>
<td>ILU(k)</td>
<td>(-p \ {ilu</td>
<td>2} -ilu_fill [0])</td>
</tr>
<tr>
<td>SSOR</td>
<td>(-p \ {ssor</td>
<td>3} -ssor_omega [1.0])</td>
</tr>
<tr>
<td>Hybrid</td>
<td>(-p \ {hybrid</td>
<td>4} -hybrid_i [sor])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_maxiter [25])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_tol [1.0e-3])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_omega [1.5])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_ell [2])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_restart [40])</td>
</tr>
<tr>
<td>I+S</td>
<td>(-p \ {is</td>
<td>5} -is_alpha [1.0])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-is_m [3])</td>
</tr>
<tr>
<td>SAINV</td>
<td>(-p \ {sainv</td>
<td>6} -sainv_drop [0.05])</td>
</tr>
<tr>
<td>SA-AMG</td>
<td>(-p \ {saamg</td>
<td>7} -saamg_unsym [false])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-saamg_theta [0.05</td>
</tr>
<tr>
<td>Crout ILU</td>
<td>(-p \ {iluc</td>
<td>8} -iluc_drop [0.05])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-iluc_rate [5.0])</td>
</tr>
<tr>
<td>ILUT</td>
<td>(-p \ {ilut</td>
<td>9})</td>
</tr>
<tr>
<td>Additive</td>
<td>(\text{adds true})</td>
<td>(-\text{adds_iter [1]})</td>
</tr>
<tr>
<td>Schwarz</td>
<td>(\text{adds true})</td>
<td>(-\text{adds_iter [1]})</td>
</tr>
</tbody>
</table>
### Other Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-emaxiter [1000]</td>
<td>The maximum number of iterations</td>
</tr>
<tr>
<td>-etol [1.0e-12]</td>
<td>The convergence tolerance</td>
</tr>
<tr>
<td>-eprint [0]</td>
<td>The output of the residual history</td>
</tr>
<tr>
<td>-print {none</td>
<td>0}</td>
</tr>
<tr>
<td>-print {mem</td>
<td>1}</td>
</tr>
<tr>
<td>-print {out</td>
<td>2}</td>
</tr>
<tr>
<td>-print {all</td>
<td>3}</td>
</tr>
<tr>
<td>-ie [ii]</td>
<td>The inner eigensolver used in Subspace, Lanczos, and Arnoldi</td>
</tr>
<tr>
<td>-ige [gii]</td>
<td>The inner eigensolver used in Generalized Subspace, Generalized Lanczos,</td>
</tr>
<tr>
<td></td>
<td>and Generalized Arnoldi</td>
</tr>
<tr>
<td>-shift [0.0]</td>
<td>The amount of the real part of the shift ( \sigma ) to compute ( A - \sigma B )</td>
</tr>
<tr>
<td>-shift_im [0.0]</td>
<td>The amount of the imaginary part of the shift ( \sigma )</td>
</tr>
<tr>
<td>-initx_ones [1]</td>
<td>The behavior of the initial vector ( x_0 )</td>
</tr>
<tr>
<td>-initx_ones {false</td>
<td>0}</td>
</tr>
<tr>
<td></td>
<td>of the function ( \text{lis_esolve()} )</td>
</tr>
<tr>
<td>-initx_ones {true</td>
<td>1}</td>
</tr>
<tr>
<td>-omp_num_threads [t]</td>
<td>The number of threads</td>
</tr>
<tr>
<td></td>
<td>(t represents the maximum number of threads)</td>
</tr>
<tr>
<td>-estorage [0]</td>
<td>The matrix storage format</td>
</tr>
<tr>
<td>-estorage_block [2]</td>
<td>The block size of the BSR and BSC formats</td>
</tr>
<tr>
<td>-ef [0]</td>
<td>The precision of the eigensolver</td>
</tr>
<tr>
<td>-ef {double</td>
<td>0}</td>
</tr>
<tr>
<td>-ef {quad</td>
<td>1}</td>
</tr>
<tr>
<td>-rval [0]</td>
<td>The Ritz values</td>
</tr>
<tr>
<td>-rval {false</td>
<td>0}</td>
</tr>
<tr>
<td>-rval {true</td>
<td>1}</td>
</tr>
</tbody>
</table>

### Solving Eigenvalue Problems

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

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

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

To solve the generalized eigenvalue problem \( Ax = \lambda Bx \), the following functions are used instead:

- C
  ```c
  LIS_INT lis_gesolve(LIS_MATRIX A, LIS_MATRIX B, 
  LIS_VECTOR x, LIS_SCALAR evalue, LIS_ESOLVER esolver)
  ```

- Fortran subroutine
  ```fortran
  lis_gesolve(LIS_MATRIX A, LIS_MATRIX B, 
  LIS_VECTOR x, LIS_SCALAR evalue, LIS_ESOLVER esolver, LIS_INTEGER ierr)
  ```
3.6 Writing Programs

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

\[
\begin{pmatrix}
2 & -1 \\
-1 & 2 & -1 \\
& & \ddots & \ddots \\
& & -1 & 2 & -1 \\
& & & -1 & 2
\end{pmatrix}
\]

of size 12. The the right-hand side vector $b$ is set such that the values of the elements of the solution $x$ are 1. The programs are located in the directory $\text{lis-$(VERSION)$}$/$\text{test}$.
Test program: test4.c

```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_CSR);
    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_iter(solver,&iter);
    printf("number of iterations = %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

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

The previous section represents one of two workflows in Lis. The present workflow represents a decoupling of the preconditioner and the solver (Preconditioner and Solver Decoupled, or PSD), in the sense that the preconditioner can be updated separately from the solver; in the workflow presented in the previous section, the preconditioner is updated with every call to lis_solve. This property (conditional updating of the preconditioner) turns out to be quite useful in solving certain problems, including some nonlinear partial differential equations. For nonlinear PDEs, iterative solution methods are used (e.g., Newton-Raphson) to calculate a solution. In the current context, each Newton-Raphson iteration calculates an incremental improvement to a solution vector via the solution of a linear system of equations. To support this functionality, the following (additional) functions must be used:

- `lis_matrix.psd_set_value`: Re-define the value of an existing matrix component.
- `lis_matrix.psd_reset_scale`: Set the matrix “scaled” status to `false`. Note that this call and the following one are only needed for the case non-trivial scaling.
- `lis_vector.psd_reset_scale`: Set the vector “scaled” status to `false`.
- `lis_solver_set_matrix`: Associate the given matrix and solver. Note that this must be done before the call to `lis_precon.psd_create`.
- `lis_precon.psd_create`: Create data structures associated with the chosen preconditioner.
- `lis_precon.psd_update`: Evaluate the preconditioner.

It is noted that this workflow currently has the following restrictions:

- The only matrix format currently supported is CSR.
- The only solver currently supported is GMRES.
- The only preconditioners currently supported are ILU(k) and SA-AMG.

Error checking has been implemented to prevent any unsupported cases from running.

The following pseudo-code listing presents an example of the workflow. An actual implementation can be seen in test problem test8f.F90.
Pseudo-code example of workflow

```fortran
1: PROGRAM psd_driver
2: implicit none
3: #include "lisf.h"
4:
5: LIS_INTEGER i,n,gn,is,ie,iter,ierr
6: LIS_MATRIX A
7: LIS_VECTOR b,x
8: LIS_SOLVER solver
9: REAL :: u(:),du
10:
11: CALL lis_initialize(ierr)
12:
13: !==================================================================
14: ! initialization, only done once
15: !==================================================================
16: CALL lis_matrix_create(LIS_COMM_WORLD,A,ierr)
17: CALL lis_matrix_set_size(A,0,n,ierr)
18: CALL lis_matrix_get_size(A,n,gn,ierr)
19: CALL lis_matrix_get_range(A,is,ie,ierr)
20:
21: CALL UpdateLinearSystem(RHS,LHS)
22: DO i=is,ie-1
23: DO j=1,gn
24: IF (LHS(i,j) exists) THEN
25: CALL lis_matrix_set_value(LIS_INS_VALUE,i,j,LHS(i,j),A,ierr)
26: END IF
27: END DO
28: END DO
29: CALL lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
30: CALL lis_matrix_assemble(A,ierr)
31:
32: CALL lis_vector_duplicate(A,b,ierr)
33: CALL lis_vector_duplicate(A,x,ierr)
34: DO i=is,ie-1
35: CALL lis_vector_set_value(LIS_INS_VALUE,i,RHS(i),b,ierr)
36: END DO
37: u = u_initial
38:
39: CALL lis_solver_create(solver,ierr)
40: WRITE(UNIT=options,FMT='(a)') "-p ilu -i gmres -print out -scale none"
41: CALL lis_solver_set_option(TRIM(options),solver,ierr)
42:
43: !==================================================================
44: ! everything up to this point is more or less identical to the standard workflow.
45: ! Now comes the preconditioner initialization, and the Newton-Raphson
46: ! iteration.
47: !==================================================================
48: CALL lis_solver_set_matrix(A,solver,ierr)
49: CALL lis_precon_psd_create(solver,precon,ierr)
50: ! evaluate the preconditioner, at least once . . .
51: CALL lis_precon_psd_update(solver,precon,ierr)
```

39
Pseudo-code example of workflow (contd.)

55: DO
56:
57: IF (UpdateLHS) THEN
58: DO i=is,ie-1
59: DO j=1,gn
60: IF (component (i,j) exists) THEN
61: CALL lis_matrix_psd_set_value(LIS_INS_VALUE,i,j,LHS(i,j),A,ierr)
62: END IF
63: END DO
64: END DO
65: CALL lis_matrix_psd_reset_scale(A,ierr)
66: END IF
67:
68: ! update RHS every iteration
69: DO i=is,ie-1
70: CALL lis_vector_set_value(LIS_INS_VALUE,i,RHS(i),b,ierr)
71: END DO
72: CALL lis_vector_psd_reset_scale(A,ierr)
73:
74: IF (UpdateLHS) THEN
75: CALL lis_precon_psd_update(solver,precon,ierr)
76: END IF
77: CALL lis_solve_kernel(A,b,x,solver,precon,ierr)
78: CALL lis_solver_get_iter(solver,iter,ierr)
79: write(*,*) 'number of iterations = ',iter
80: CALL lis_vector_print(x,ierr)
81:
82: ! update the solution
83: DO i=is,ie-1
84: CALL lis_vector_get_value(x,i,du,ierr)
85: u(i)=u(i)-du
86: END DO
87:
88: CALL UpdateLinearSystem(RHS,LHS)
89:
90: IF (termination criteria satisfied) EXIT
91:
92: END DO
93:
94:
95: CALL lis_matrix_destroy(A,ierr)
96: CALL lis_vector_destroy(b,ierr)
97: CALL lis_vector_destroy(x,ierr)
98: CALL lis_vector_destroy(u,ierr)
99: CALL lis_solver_destroy(solver,ierr)
100:
101: CALL lis_finalize(ierr)
102:
103: END PROGRAM psd_driver

40
3.8 Compiling and Linking

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

For the serial environment

**Compiling**

```
> icc -c -I($INSTALLDIR)/include test4.c
```

**Linking**

```
> icc -o test4 test4.o -L($INSTALLDIR)/lib -llis
```

**Linking (with SA-AMG)**

```
> ifort -nofor_main -o test4 test4.o -L($INSTALLDIR)/lib -llis
```

For multithreaded environment

**Compiling**

```
> icc -c -openmp -I($INSTALLDIR)/include test4.c
```

**Linking**

```
> icc -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis
```

**Linking (with SA-AMG)**

```
> ifort -nofor_main -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis
```

For multiprocessing environment

**Compiling**

```
> icc -c -DUSE_MPI -I($INSTALLDIR)/include test4.c
```

**Linking**

```
> icc -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi
```

**Linking (with SA-AMG)**

```
> ifort -nofor_main -o test4 test4.o -L($INSTALLDIR)/lib -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 -L($INSTALLDIR)/lib -llis -lmpi
```

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

For serial environment

**Compiling**

```
> ifort -c -fpp -I($INSTALLDIR)/include test4f.F
```

**Linking**

```
> ifort -o test4f test4f.o -L($INSTALLDIR)/lib -llis
```

For serial environment

```
For multithreaded environment

Compiling
> ifort -c -fpp -openmp -I($INSTALLDIR)/include test4.F

Linking
> ifort -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis

For multiprocessing environment

Compiling
> ifort -c -fpp -DUSE_MPI -I($INSTALLDIR)/include test4.F

Linking
> ifort -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi

For multithreaded and multiprocessing environments

Compiling
> ifort -c -fpp -openmp -DUSE_MPI -I($INSTALLDIR)/include test4.F

Linking
> ifort -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi

3.9 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 solution will be returned:

initial vector x : all components set to 0
precision : double
linear solver : BiCGSTAB
preconditioner : none
convergence condition : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status : normal end

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

42
3.10 Zero DOFs on a Process

For the case of multiple MPI processes, it is possible to specify $global_n = 0$ and $local_n \geq 0$. This will allow one or more MPI processes to have zero degrees of freedom (DOFs). However, it is still the case that, for $global_n = 0$, the summation of $local_n$ over all processes should be greater than zero.
4 Quadruple Precision Operations

Double precision operations sometimes require a large number of iterations because of the rounding error. Besides long double precision operations, Lis supports "double-double" precision operations, which are effectively quadruple precision operations achieved by combining two double precision floating point numbers[50, 51]. To use the double-double precision with the same interface as the double precision operations, both the matrix and vectors are assumed to be double precision. Lis also supports the performance acceleration of the double-double precision operations with the SIMD instructions, such as Intel’s Streaming SIMD Extensions (SSE)[56].

4.1 Using Quadruple Precision Operations

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

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

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

**Double precision**

To run the test, enter

> ./test5 200 2.0 -f double

and the following results will be returned:

```
 n = 200, gamma = 2.000000
initial vector x : all components set to 0
precision : double
linear solver : BiCG
preconditioner : none
convergence condition : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
matrix storage format : CSR
linear solver status : normal end
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 = 8.917591e+01
```

**Quadruple precision**

To run the test, enter

> ./test5 200 2.0 -f quad

and the following results will be returned:

```
n = 200, gamma = 2.000000
```
initial vector $x$ : all components set to 0
precision : quad
linear solver : BiCG
preconditioner : none
convergence condition : $\|b-Ax\|_2 \leq 1.0e-12 \times \|b-Ax_0\|_2$
matrix storage format : CSR
linear solver status : 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 = 6.499145e-11
5 Matrix Storage Formats

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

5.1 Compressed Sparse Row (CSR)

The CSR format uses three arrays $\text{ptr}$, $\text{index}$ and $\text{value}$ to store data.

- $\text{value}$ is a double precision array of length $nnz$, which stores the nonzero elements of matrix $A$ along the row.
- $\text{index}$ is an integer array of length $nnz$, which stores the column numbers of the nonzero elements stored in the array $\text{value}$.
- $\text{ptr}$ is an integer array of length $n + 1$, which stores the starting points of the rows of the arrays $\text{value}$ and $\text{index}$.

5.1.1 Creating Matrices (for Serial and Multithreaded Environments)

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

```c
1: LIS_INT n,nnz;
2: LIS_INT *ptr,*index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8;
6: ptr = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
7: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
16: lis_matrix_set_csr(nnz,ptr,index,value,A);
17: lis_matrix_assemble(A);
```

Figure 2: Data structure of CSR format (for serial and multithreaded environments).
5.1.2 Creating Matrices (for Multiprocessing Environment)

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

For multiprocessing environment

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

5.1.3 Associating Arrays

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

- C
  
  ```c
  LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT ptr[], LIS_INT index[],
  LIS_SCALAR value[], LIS_MATRIX A)
  ```

- Fortran subroutine
  
  ```fortran
  subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER ptr(),
  LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
  ```

Figure 3: Data structure of CSR format (for multiprocessing environment).
5.2 Compressed Sparse Column (CSC)

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

- `value` is a double precision array of length `nnz`, which stores the nonzero elements of matrix `A` along the column.
- `index` is an integer array of length `nnz`, which stores the row numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array of length `n + 1`, which stores the starting points of the rows of the arrays `value` and `index`.

5.2.1 Creating Matrices (for Serial and Multithreaded Environments)

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

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

```c
1: LIS_INT n,nnz;
2: LIS_INT *ptr,*index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8;
6: ptr = (LIS_INT *)malloc( (n+1)*sizeof(LIS_INT) );
7: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
16: lis_matrix_set_csc(nnz,ptr,index,value,A);
17: lis_matrix_assemble(A);
```

Figure 4: Data structure of CSC format (for serial and multithreaded environments).
5.2.2 Creating Matrices (for Multiprocessing Environment)

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

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

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

5.2.3 Associating Arrays

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

- C
  ```c
  LIS_INT lis_matrix_set_csc(LIS_INT nnz, LIS_INT row[], LIS_INT index[],
  LIS_SCALAR value[], LIS_MATRIX A)
  ```
- Fortran subroutine
  ```fortran
  subroutine lis_matrix_set_csc(LIS_INTEGER nnz, LIS_INTEGER row(),
  LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
  ```
5.3 Modified Compressed Sparse Row (MSR)

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

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

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

5.3.1 Creating Matrices (for Serial and Multithreaded Environments)

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

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

Figure 6: Data structure of MSR format (for serial and multithreaded environments).
5.3.2 Creating Matrices (for Multiprocessing Environment)

Figure 7 shows how matrix $A$ in Figure 6 is stored in the MSR format on two processing elements. A program to create the matrix in the MSR format on two processing element is as follows:

```
PE1 PE0

0 4 3 3
0 4 3 3
21 22 11
21 22 11

A.index
2 0 1 6 4 3
A.value
43 41 32 44 33
```

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

For multiprocessing environment

1: LIS_INT i,k,n,nnz,ndz,my_rank;
2: LIS_INT *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
6: if( my_rank==0 ) {n = 2; nnz = 3; ndz = 0;}
7: else {n = 2; nnz = 5; ndz = 0;}
8: index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(LIS_INT) );
9: value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
10: lis_matrix_create(MPI_COMM_WORLD,&A);
11: lis_matrix_set_size(A,n,0);
12: if( my_rank==0 ) {
15: else {
17: index[4] = 0; index[5] = 2;
20: lis_matrix_set_msr(nnz,ndz,index,value,A);
21: lis_matrix_assemble(A);
```

5.3.3 Associating Arrays

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

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

- **Fortran subroutine**
  
  ```
  lis_matrix_set_msr(LIS_INTEGER nnz, LIS_INTEGER ndz,
  LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
  ```
5.4 Diagonal (DIA)

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

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

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

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

5.4.1 Creating Matrices (for Serial Environment)

The diagram on the right in Figure 8 shows how matrix \( A \) in Figure 8 is stored in the DIA format. A program to create the matrix in the DIA format is as follows:

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

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

A.value
```

Figure 8: Data structure of DIA format (for serial environment).

```plaintext
1: LIS_INT n,nnd;
2: LIS_INT *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnd = 3;
6: index = (LIS_INT *)malloc( nnd*sizeof(LIS_INT) );
7: value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
8: lis_matrix_create(0,&A);
9: lis_matrix_set_size(A,0,n);
10:
11: index[0] = -3; index[1] = -1; index[2] = 0;
15:
16: lis_matrix_set_dia(nnd,index,value,A);
17: lis_matrix_assemble(A);
```
5.4.2 Creating Matrices (for Multithreaded Environment)

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

```
1: LIS_INT n,maxnnd,nprocs;
2: LIS_INT *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; maxnnd = 3; nprocs = 2;
6: index = (LIS_INT *)malloc( maxnnd*sizeof(LIS_INT) );
7: value = (LIS_SCALAR *)malloc( n*maxnnd*sizeof(LIS_SCALAR) );
8: lis_matrix_create(0,&A);
9: lis_matrix_set_size(A,0,n);
13: lis_matrix_set_dia(maxnnd,index,value,A);
14: lis_matrix_assemble(A);
```

Figure 9: Data structure of DIA format (for multithreaded environment).
### 5.4.3 Creating Matrices (for Multiprocessing Environment)

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

```c
PE0
-1  0
 0 21 11 22
PE1
-3 -1  0
 0 41 32 43 33 44
```

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

For multiprocessing environment

```c
1: LIS_INT i,n,nnd,my_rank;
2: LIS_INT *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
6: if( my_rank==0 ) {n = 2; nnd = 2;}
7: else {n = 2; nnd = 3;}
8: index = (LIS_INT *)malloc( nnd*sizeof(LIS_INT) );
9: value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
10: lis_matrix_create(MPI_COMM_WORLD,&A);
11: lis_matrix_set_size(A,n,0);
12: if( my_rank==0 ) {
13:   index[0] = -1; index[1] = 0;
15: else {
16:   index[0] = -3; index[1] = -1; index[2] = 0;
18:   value[5] = 44;}
19: lis_matrix_set_dia(nnd,index,value,A);
20: lis_matrix_assemble(A);
```

### 5.4.4 Associating Arrays

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

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

- **Fortran subroutine**
  ```fortran
  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 \( \text{maxnzr} \) is the maximum value of the number of nonzero elements in the rows of matrix \( A \).

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

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

5.5.1 Creating Matrices (for Serial and Multithreaded Environments)

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

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

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

```c
1: LIS_INT n, maxnzr;
2: LIS_INT *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; maxnzr = 3;
6: index = (LIS_INT *)malloc( n*maxnzr*sizeof(LIS_INT) );
7: value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
8: lis_matrix_create(0,&A);
9: lis_matrix_set_size(A,0,n);
14: lis_matrix_set_ell(maxnzr,index,value,A);
15: lis_matrix_assemble(A);
```

For serial and multithreaded environments
5.5.2 Creating Matrices (for Multiprocessing Environment)

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

```c
1: LIS_INT i,n,maxnzr,my_rank;
2: LIS_INT *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
6: if( my_rank==0 ) {n = 2; maxnzr = 2;}
7: else {n = 2; maxnzr = 3;}
8: index = (LIS_INT *)malloc( n*maxnzr*sizeof(LIS_INT) );
9: value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
10: lis_matrix_create(MPI_COMM_WORLD,&A);
11: lis_matrix_set_size(A,n,0);
12: if( my_rank==0 ) {
15: else {
17:   index[5] = 3;
20: lis_matrix_set_ell(maxnzr,index,value,A);
21: lis_matrix_assemble(A);
```

5.5.3 Associating Arrays

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

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

- **Fortran subroutine**
  ```fortran
  lis_matrix_set_ell(LIS_INTEGER maxnzr, LIS_INTEGER index(),
  LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
  ```
5.6 Jagged Diagonal (JAD)

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

- \texttt{perm} is an integer array of length \texttt{n}, which stores the sorted row numbers.

- \texttt{value} is a double precision array of length \texttt{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.

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

- \texttt{ptr} is an integer array of length \texttt{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, \texttt{perm, ptr, index, and value}, to store data. Assume that \texttt{nprocs} is the number of threads. $maxnzr_p$ is the number of nonzero diagonal elements of the partial matrix into which the row block of matrix $A$ is divided. $maxmaxnzr$ is the maximum value of $maxnzr_p$.

- \texttt{perm} is an integer array of length \texttt{n}, which stores the sorted row numbers.

- \texttt{value} is a double precision array of length \texttt{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.

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

- \texttt{ptr} is an integer array of length \texttt{nprocs} $\times$ ($maxmaxnzr + 1$), which stores the starting points of the jagged diagonal elements.
5.6.1 Creating Matrices (for Serial Environment)

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

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

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

```c
1: LIS_INT n,nnz,maxnzr;
2: LIS_INT *perm,*ptr,*index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8; maxnzr = 3;
6: perm = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
7: ptr = (LIS_INT *)malloc( (maxnzr+1)*sizeof(LIS_INT) );
8: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
9: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
10: lis_matrix_create(0,&A);
11: lis_matrix_set_size(A,0,n);
14: index[0] = 0; index[1] = 0; index[2] = 1; index[3] = 0;
18: lis_matrix_set_jad(nnz,maxnzr,perm,ptr,index,value,A);
19: lis_matrix_assemble(A);
```
5.6.2 Creating Matrices (for Multithreaded Environment)

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

```c
1: LIS_INT n,nnz,maxmaxnzr,nprocs;
2: LIS_INT *perm,*ptr,*index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8; maxmaxnzr = 3; nprocs = 2;
6: perm = (LIS_INT *)malloc( n*sizeof(LIS_INT) );
7: ptr = (LIS_INT *)malloc( nprocs*(maxmaxnzr+1)*sizeof(LIS_INT) );
8: index = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
9: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
10: lis_matrix_create(0,&A);
11: lis_matrix_set_size(A,0,n);
12: 
14: ptr[0] = 0; ptr[1] = 2; ptr[2] = 3; ptr[3] = 0;
16: index[0] = 0; index[1] = 0; index[2] = 1; index[3] = 0;
20: 
21: lis_matrix_set_jad(nnz,maxmaxnzr,perm,ptr,index,value,A);
22: lis_matrix_assemble(A);
```

Figure 14: Data structure of JAD format (for multithreaded environment).
5.6.3 Creating Matrices (for Multiprocessing Environment)

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

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

5.6.4 Associating Arrays

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

- C
  ```
  LISR_INT lis_matrix_set_jad(LISR_INT nnz, LISR_INT maxnzr, LISR_INT perm[],
  LISR_INT ptr[], LISR_INT index[], LISR_SCALAR value[], LISR_MATRIX A)
  ```

- Fortran subroutine
  ```lis_matrix_set_jad(LISRINTEGER nnz, LISRINTEGER maxnzr, LISRINTEGER perm(), LISRINTEGER ptr(), LISRINTEGER index(), LISR_SCALAR value(), LISR_MATRIX A, LISR_INTEGER ierr)```
5.7 Block Sparse Row (BSR)

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

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

5.7.1 Creating Matrices (for Serial and Multithreaded Environments)

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

```c
1: LIS_INT n,bnr,bnc,nc,bnnz;
2: LIS_INT *bptr,*bindex;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
6: bptr = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
7: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
8: value = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
12: bindex[0] = 0; bindex[1] = 0; bindex[2] = 1;
14: lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
15: lis_matrix_assemble(A);
```

Figure 16: Data structure of BSR format (for serial and multithreaded environments).
5.7.2 Creating Matrices (for Multiprocessing Environment)

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

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

5.7.3 Associating Arrays

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

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

- $value$ is a double precision array of length $bnnz \times r \times c$, which stores the values of the elements of the nonzero blocks.
- $bindex$ is an integer array of length $bnnz$, which stores the block row numbers of the nonzero blocks.
- $bptr$ is an integer array of length $nc + 1$, which stores the starting points of the block columns in the array $bindex$.

5.8.1 Creating Matrices (for Serial and Multithreaded Environments)

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

```c
1: LIS_INT n,bnr,bnc,nc,bnnz;
2: LIS_INT *bptr,*bindex;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
6: bptr = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
7: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
8: value = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
12: bindex[0] = 0; bindex[1] = 1; bindex[2] = 1;
16: lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
17: lis_matrix_assemble(A);
```

Figure 18: Data structure of BSC format (for serial and multithreaded environments).
5.8.2 Creating Matrices (for Multiprocessing Environment)

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

```c
A.value 44 0 43 33
A.binde x
A.value 44 0 43 33
A.binde x
A.bptr 1 0 A.bptr 1 0
0 32 41 0 22 0 21 11
1
2 0
PE1 PE0
```

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

For multiprocessing environment

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

5.8.3 Associating Arrays

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

- C
  ```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
  ```c
  lis_matrix_set_bsc(LIS_INTEGER bnr, LIS_INTEGER bnc,
  LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(),
  LIS_MATRIX A, LIS_INTEGER ierr)
  ```
5.9 Variable Block Row (VBR)

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

- row is an integer array of length \( nr + 1 \), which stores the starting row number of the block rows.
- col is an integer array of length \( nc + 1 \), which stores the starting column number of the block columns.
- bindex is an integer array of length \( bnnz \), which stores the block column numbers of the nonzero blocks.
- bptr is an integer array of length \( nr + 1 \), which stores the starting points of the block rows in the array bindex.
- value is a double precision array of length \( nnz \), which stores the values of the elements of the nonzero blocks.
- ptr is an integer array of length \( bnnz + 1 \), which stores the starting points of the nonzero blocks in the array value.
5.9.1 Creating Matrices (for Serial and Multithreaded Environments)

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

For serial and multithreaded environments

```c
1: LIS_INT n,nnz,nr,nc,bnnz;
2: LIS_INT *row,*col,*ptr,*bptr,*bindex;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 11; bnnz = 6; nr = 3; nc = 3;
6: bptr = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
7: row = (LIS_INT *)malloc( (nr+1)*sizeof(LIS_INT) );
8: col = (LIS_INT *)malloc( (nc+1)*sizeof(LIS_INT) );
9: ptr = (LIS_INT *)malloc( (bnnz+1)*sizeof(LIS_INT) );
10: bindex = (LIS_INT *)malloc( bnnz*sizeof(LIS_INT) );
11: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
12: lis_matrix_create(0,&A);
13: lis_matrix_set_size(A,0,n);
14:
25:
26: lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
27: lis_matrix_assemble(A);
```

Figure 20: Data structure of VBR format (for serial and multithreaded environments).
5.9.2 Creating Matrices (for Multiprocessing Environment)

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

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

For multiprocessing environment

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

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

- C
  
  ```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
  
  ```fortran
  subroutine lis_matrix_set_vbr(LIS_INTEGER nnz, LIS_INTEGER nr,
  LIS_INTEGER nc, LIS_INTEGER bnnz, LIS_INTEGER row(), LIS_INTEGER col(),
  LIS_INTEGER ptr(), LIS_INTEGER bptr(), LIS_INTEGER bindex(),
  LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
  ```
5.10 Coordinate (COO)

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

- **value** is a double precision array of length **nnz**, which stores the values of the nonzero elements.
- **row** is an integer array of length **nnz**, which stores the row numbers of the nonzero elements.
- **col** is an integer array of length **nnz**, which stores the column numbers of the nonzero elements.

5.10.1 Creating Matrices (for Serial and Multithreaded Environments)

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

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>col</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
1: LIS_INT n,nnz;
2: LIS_INT *row,*col;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8;
6: row = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
7: col = (LIS_INT *)malloc( nnz*sizeof(LIS_INT) );
8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
14: lis_matrix_set_coo(nnz,row,col,value,A);
15: lis_matrix_assemble(A);
```

For serial and multithreaded environments
5.10.2 Creating Matrices (for Multiprocessing Environment)

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

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

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

5.10.3 Associating Arrays

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

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

- Fortran subroutine
  
  ```fortran
  lis_matrix_set_coo(LIS_INTEGER nnz, LIS_INTEGER row(),
  LIS_INTEGER col(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
  ```
5.11 Dense (DNS)

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

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

5.11.1 Creating Matrices (for Serial and Multithreaded Environments)

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

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

A. Value

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

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

```c
1: LIS_INT n;
2: LIS_SCALAR *value;
3: LIS_MATRIX A;
4: n = 4;
5: value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
6: lis_matrix_create(0,&A);
7: lis_matrix_set_size(A,0,n);
8:
13:
14: lis_matrix_set_dns(value,A);
15: lis_matrix_assemble(A);
```
5.11.2 Creating Matrices (for Multiprocessing Environment)

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

```c
1: LIS_INT n, my_rank;
2: LIS_SCALAR *value;
3: LIS_MATRIX A;
4: MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
5: if (my_rank == 0) {n = 2;}
6: else {n = 2;}
7: value = (LIS_SCALAR *)malloc(n*n*sizeof(LIS_SCALAR));
8: lis_matrix_create(MPI_COMM_WORLD, &A);
9: lis_matrix_set_size(A, n, 0);
10: if (my_rank == 0) {
13: else {
16: lis_matrix_set_dns(value, A);
17: lis_matrix_assemble(A);
```

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

5.11.3 Associating Arrays

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

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

- **Fortran subroutine**
  
  ```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 statuses of the solvers are defined as follows:

- **LIS_SUCCESS(0)**: Normal termination
- **LIS_ILL_OPTION(1)**: Illegal option
- **LIS_BREAKDOWN(2)**: Breakdown (division by zero)
- **LIS_OUT_OF_MEMORY(3)**: Out of working memory
- **LIS_MAXITER(4)**: Maximum number of iterations
- **LIS_NOT_IMPLEMENTED(5)**: Not implemented
- **LIS_ERR_FILE_IO(6)**: File I/O error
6.1 Operating Vector Elements

Assume that the size of vector \( v \) is \( \text{global}_n \) and that the size of the partial vectors stored on \( nprocs \) processing elements is \( \text{local}_n \). \( \text{global}_n \) and \( \text{local}_n \) are called the global size and the local size, respectively.

6.1.1 lis_vector_create

<table>
<thead>
<tr>
<th>Language</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>LIS_INT lis_vector_create(LIS_Comm comm, LIS_VECTOR *v)</td>
<td>Create vector ( v ).</td>
</tr>
<tr>
<td>Fortran</td>
<td>subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR v, LIS_INTEGER ierr)</td>
<td>Create vector ( v ).</td>
</tr>
</tbody>
</table>

**Input**
- \( \text{comm} \): The MPI communicator

**Output**
- \( v \): The vector
- \( \text{ierr} \): The return code

**Note**
For the serial and multithreaded environments, the value of \( \text{comm} \) is ignored.

6.1.2 lis_vector_destroy

<table>
<thead>
<tr>
<th>Language</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>LIS_INT lis_vector_destroy(LIS_VECTOR v)</td>
<td>Destroy vector ( v ).</td>
</tr>
<tr>
<td>Fortran</td>
<td>subroutine lis_vector_destroy(LIS_VECTOR v, LIS_INTEGER ierr)</td>
<td>Destroy vector ( v ).</td>
</tr>
</tbody>
</table>

**Input**
- \( v \): The vector to be destroyed

**Output**
- \( \text{ierr} \): The return code
6.1.3 lis_vector_duplicate

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

Description
Create vector \(v_{\text{out}}\), which has the same information as \(v_{\text{in}}\).

Input
vin       The source vector or matrix

Output
vout      The destination vector
ierr      The return code

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

6.1.4 lis_vector_set_size

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

Description
Assign the size of vector \(v\).

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

Output
ierr    The return code

Note
Either local_n or global_n must be provided.
For the serial and multithreaded environments, local_n is equal to global_n. Therefore, both
lis_vector_set_size(v,n,0) and lis_vector_set_size(v,0,n) create a vector of size \(n\).
For the multiprocessing environment, lis_vector_set_size(v,n,0) creates a partial vector of size \(n\)
on each processing element. On the other hand, lis_vector_set_size(v,0,n) creates a partial vector
of size \(m_p\) on processing element \(p\). The values of \(m_p\) are determined by the library.
6.1.5 lis_vector_get_size

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

Description
Get the size of vector v.

Input
v       The vector

Output
local_n  The size of the partial vector
global_n The size of the global vector
ierr     The return code

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

6.1.6 lis_vector_get_range

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

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

Input
v       The partial vector

Output
is      The location where the partial vector v starts in the global vector
ie      The location where the partial vector v ends in the global vector
ierr    The return code

Note
For the serial and multithreaded environments, a vector of size n results in is = 0 and ie = n in the C version, and is = 1 and ie = n + 1 in the Fortran version.
6.1.7 lis_vector_set_value

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

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

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

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

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

6.1.8 lis_vector_get_value

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

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

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

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

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

C

    LIS_INT lis_vector_set_values(LIS_INT flag, LIS_INT count,
    LIS_INT index[], LIS_SCALAR value[], LIS_VECTOR v)

Fortran subroutine lis_vector_set_values(LIS_INTEGER flag, LIS_INTEGER count,
    LIS_INTEGER index(), LIS_SCALAR value(), LIS_VECTOR v, LIS_INTEGER ierr)

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

Input
flag
    LIS_INS_VALUE : v[index[i]] = value[i]
    LIS_ADD_VALUE : v[index[i]] = v[index[i]] + value[i]

count
    The number of elements in the array that stores the scalar values to be assigned

index
    The array that stores the location where the scalar values are assigned

value
    The array that stores the scalar values to be assigned

v
    The vector

Output
v
    The vector with scalar value[i] assigned to the index[i]-th row

ierr
    The return code

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

C    LIS_INT lis_vector_get_values(LIS_VECTOR v, LIS_INT start, LIS_INT count,
                        LIS_SCALAR value[])

Fortran subroutine lis_vector_get_values(LIS_VECTOR v, LIS_INTEGER start,
                        LIS_INTEGER count, LIS_SCALAR value(), LIS_INTEGER ierr)

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

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

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

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

6.1.11 lis_vector_scatter

C    LIS_INT lis_vector_scatter(LIS_SCALAR value[], LIS_VECTOR v)

Fortran subroutine lis_vector_scatter(LIS_SCALAR value(), LIS_VECTOR v,
                        LIS_INTEGER ierr)

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

Input
value             The array that stores the scalar values to be assigned

Output
v                 The vector
ierr              The return code

Note
For the multiprocessing environment, this function is implemented by MPI_Scatterv.
6.1.12 lis_vector_gather

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

Description
Get scalar value[i] of the i-th row of vector v, where i = 0, 1, ..., global_n - 1.

Input
v              The source vector

Output
value          The array that stores the scalar values
ierr           The return code

Note
For the multiprocessing environment, this function is implemented by MPI_Allgatherv.

6.1.13 lis_vector_is_null

C       LIS_INT lis_vector_is_null(LIS_VECTOR v)
Fortran subroutine lis_vector_is_null(LIS_VECTOR v,LIS_INTEGER ierr)

Description
Determine if vector v is available.

Input
v              The vector

Output
ierr           The return code

   LIS_TRUE            Available
   LIS_FALSE           Not available
6.2 Operating Matrix Elements

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

6.2.1 lis_matrix_create

C
LIS_INT lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)

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

Description
Create matrix $A$.

Input
comm The MPI communicator

Output
A The matrix
ierr The return code

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

6.2.2 lis_matrix_destroy

C
LIS_INT lis_matrix_destroy(LISMATRIX A)

Fortran subroutine lis_matrix_destroy(LIS_MATRIX A, LIS_INTEGER ierr)

Description
Destroy matrix $A$.

Input
A The matrix to be destroyed

Output
ierr The return code

Note
The function lis_matrix_destroy frees the memory for the set of arrays associated with matrix $A$. 
6.2.3 lis_matrix_duplicate

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

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

Input
   Ain          The source matrix

Output
   Aout         The destination matrix
   ierr         The return code

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

6.2.4 lis_matrix_malloc

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

Description
Allocate memory for matrix $A$.

Input
   A           The matrix
   nnz_row     The average number of nonzero elements in each row
   nnz         The array of numbers of nonzero elements in each row

Output
   ierr        The return code

Note
Either nnz_row or nnz must be provided.
This function allocates memory for the function lis_matrix_set_value.
6.2.5  lis_matrix_set_value

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

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

Input
flag LIS_INS_VALUE : \(A[i, j] = value\)
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 \texttt{lis\_matrix\_set\_value} stores the assigned value in a temporary internal format. Therefore, after \texttt{lis\_matrix\_set\_value} is called, the function \texttt{lis\_matrix\_assemble} must be called.

For large matrices, the introduction of the function \texttt{lis\_matrix\_set\_type} should be considered. See \texttt{lis-\$VERSION)/test/test2.c} and \texttt{lis-\$VERSION)/test/test2f.F90} for details.

6.2.6  lis_matrix_assemble

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

Description
Assemble matrix \(A\) into the specified storage format.

Input
A The matrix

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

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

Description
Assign the size of matrix A.

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

Output
ierr The return code

Note
Either local_n or global_n must be provided.
For the serial and multithreaded environments, local_n is equal to global_n. Therefore, both
lis_matrix_set_size(A,n,0) and lis_matrix_set_size(A,0,n) create a matrix of size \( n \times n \).
For the multiprocessing environment, lis_matrix_set_size(A,n,0) creates a partial matrix of size 
\( n \times N \) on each processing element, where \( N \) is the total sum of \( n \). On the other hand,
lis_matrix_set_size(A,0,n) creates a partial matrix of size \( m_p \times n \) on processing element \( p \). The
values of \( m_p \) are determined by the library.

6.2.8 lis_matrix_get_size

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

Description
Get the size of matrix A.

Input
A The matrix

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

Note
For the serial and multithreaded environments, local_n is equal to global_n.
6.2.9 lis_matrix_get_range

```c
C    LIS_INT lis_matrix_get_range(LIS_MATRIX A, LIS_INT *is, LIS_INT *ie)
```

Fortran subroutine:
```
Fortran subroutine lis_matrix_get_range(LIS_MATRIX A, LIS_INTEGER is,
LIS_INTEGER ie, LIS_INTEGER ierr)
```

**Description**
Get the location of partial matrix \( A \) in the global matrix.

**Input**
- \( A \) The partial matrix

**Output**
- \( is \) The location where partial matrix \( A \) starts in the global matrix
- \( ie \) The location where 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 \( ie = n \) in the C version, and \( is = 1 \) and \( ie = n + 1 \) in the Fortran version.

6.2.10 lis_matrix_get_nnz

```c
C    LIS_INT lis_matrix_get_nnz(LIS_MATRIX A, LIS_INT *nnz)
```

Fortran subroutine:
```
Fortran subroutine lis_matrix_get_nnz(LIS_MATRIX A, LIS_INTEGER nnz,
LIS_INTEGER ierr)
```

**Description**
Get the number of nonzero elements of matrix \( A \).

**Input**
- \( A \) The matrix

**Output**
- \( nnz \) The number of nonzero elements
- \( ierr \) The return code

**Note**
For the multiprocessing environment, this function gets the number of nonzero elements of partial matrix \( A \).
6.2.11 lis_matrix_set_type

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

Description
   Assign the storage format.

Input
   A          The matrix
   matrix_type The storage format

Output
   ierr      The return code

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

<table>
<thead>
<tr>
<th>Storage format</th>
<th>matrix_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Sparse Row</td>
<td>LIS_MATRIX_CSR</td>
</tr>
<tr>
<td>Compressed Sparse Column</td>
<td>LIS_MATRIX_CSC</td>
</tr>
<tr>
<td>Modified Compressed Sparse Row</td>
<td>LIS_MATRIX_MSR</td>
</tr>
<tr>
<td>Diagonal</td>
<td>LIS_MATRIX_DIA</td>
</tr>
<tr>
<td>Ellpack-Itpack Generalized Diagonal</td>
<td>LIS_MATRIX_ELL</td>
</tr>
<tr>
<td>Jagged Diagonal</td>
<td>LIS_MATRIX_JAD</td>
</tr>
<tr>
<td>Block Sparse Row</td>
<td>LIS_MATRIX_BSR</td>
</tr>
<tr>
<td>Block Sparse Column</td>
<td>LIS_MATRIX_BSC</td>
</tr>
<tr>
<td>Variable Block Row</td>
<td>LIS_MATRIX_VBR</td>
</tr>
<tr>
<td>Coordinate</td>
<td>LIS_MATRIX_COO</td>
</tr>
<tr>
<td>Dense</td>
<td>LIS_MATRIX_DNS</td>
</tr>
</tbody>
</table>

6.2.12 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
### 6.2.13 lis_matrix_set_csr

```c
LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT ptr[], LIS_INT index[],
LIS_SCALAR value[], LIS_MATRIX A)
```

**Fortran subroutine**

```fortran
Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER ptr(),
LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

**Description**

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

**Input**

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

**Output**

- `A` The matrix associated with the arrays

**Note**

After `lis_matrix_set_csr` is called, the function `lis_matrix_assemble` must be called. Array indexing must be zero-origin in the Fortran version.

### 6.2.14 lis_matrix_set_csc

```c
LIS_INT lis_matrix_set_csc(LIS_INT nnz, LIS_INT ptr[], LIS_INT index[],
LIS_SCALAR value[], LIS_MATRIX A)
```

**Fortran subroutine**

```fortran
Fortran subroutine lis_matrix_set_csc(LIS_INTEGER nnz, LIS_INTEGER ptr(),
LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)
```

**Description**

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

**Input**

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

**Output**

- `A` The matrix associated with the arrays

**Note**

After `lis_matrix_set_csc` is called, the function `lis_matrix_assemble` must be called. Array indexing must be zero-origin in the Fortran version.
6.2.15 lis_matrix_set_msr

\[\text{C} \quad \text{LIS_INT lis_matrix_set_msr(LIS_INT nnz, LIS_INT ndz, LIS_INT index[],}
\]
\[\text{LIS_SCALAR value[], LIS_MATRIX A)}\]

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

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

Input
\begin{itemize}
  \item \textit{nnz} The number of nonzero elements
  \item \textit{ndz} The number of nonzero elements in the diagonal
  \item \textit{index, value} The arrays in the MSR format
  \item \textit{A} The matrix
\end{itemize}

Output
\begin{itemize}
  \item \textit{A} The matrix associated with the arrays
\end{itemize}

Note
After \textit{lis_matrix_set_msr} is called, the function \textit{lis_matrix_assemble} must be called. Array indexing must be zero-origin in the Fortran version.

6.2.16 lis_matrix_set_dia

\[\text{C} \quad \text{LIS_INT lis_matrix_set_dia(LIS_INT nnd, LIS_INT index[],}
\]
\[\text{LIS_SCALAR value[], LIS_MATRIX A)}\]

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

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

Input
\begin{itemize}
  \item \textit{nnd} The number of nonzero diagonal elements
  \item \textit{index, value} The arrays in the DIA format
  \item \textit{A} The matrix
\end{itemize}

Output
\begin{itemize}
  \item \textit{A} The matrix associated with the arrays
\end{itemize}

Note
After \textit{lis_matrix_set_dia} is called, the function \textit{lis_matrix_assemble} must be called. Array indexing must be zero-origin in the Fortran version.
6.2.17 lis_matrix_set_ell

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

Description

Associate the arrays in the ELL format with matrix A.

Input

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

Output

A            The matrix associated with the arrays

Note

After lis_matrix_set_ell is called, the function lis_matrix_assemble must be called. Array
indexing must be zero-origin in the Fortran version.

6.2.18 lis_matrix_set_jad

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

Description

Associate the arrays in the JAD format with matrix A.

Input

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

Output

A            The matrix associated with the arrays

Note

After lis_matrix_set_jad is called, the function lis_matrix_assemble must be called. Array
indexing must be zero-origin in the Fortran version.
6.2.19 lis_matrix_set_bsr

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

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

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

Output
A       The matrix associated with the arrays

Note
After lis_matrix_set_bsr is called, the function lis_matrix_assemble must be called. Array indexing must be zero-origin in the Fortran version.

6.2.20 lis_matrix_set_bsc

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

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

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

Output
A       The matrix associated with the arrays

Note
After lis_matrix_set_bsc is called, the function lis_matrix_assemble must be called. Array indexing must be zero-origin in the Fortran version.
6.2.21 lis_matrix_set_vbr

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

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

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

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

Output
A The matrix associated with the arrays

Note
After lis_matrix_set_vbr is called, the function lis_matrix_assemble must be called. Array indexing must be zero-origin in the Fortran version.
6.2.22 lis_matrix_set_coo

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

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

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

Input

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

Output

A       The matrix associated with the arrays

Note
After lis_matrix_set_coo is called, the function lis_matrix_assemble must be called. Array
indexing must be zero-origin in the Fortran version.
6.2.23 lis_matrix_set_dns

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

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

Input
value             The array in the DNS format
A                 The matrix

Output
A                 The matrix associated with the array

Note
After lis_matrix_set_dns is called, the function lis_matrix_assemble must be called. Array
indexing must be zero-origin in the Fortran version.

6.2.24 lis_matrix_unset

C     LIS_INT lis_matrix_unset(LIS_MATRIX A)
Fortran subroutine lis_matrix_unset(LIS_MATRIX A, LIS_INTEGER ierr)

Description
Unassociate the arrays from matrix A without deallocating memory.

Input
A                 The matrix associated with the arrays

Output
A                 The unassociated matrix

Note
After lis_matrix_unset is called, the function lis_matrix_destroy must be called.
6.3 Computing with Vectors and Matrices

6.3.1 lis_vector_swap

```c
LIS_INT lis_vector_swap(LIS_VECTOR x, LIS_VECTOR y)
```

Fortran subroutine:
```
lis_vector_swap(LIS_VECTOR x, LIS_VECTOR y, LIS_INTEGER ierr)
```

Description
Swap the values of the vector elements.

Input
- `x, y` The source vectors

Output
- `x, y` The destination vectors
- `ierr` The return code

6.3.2 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
6.3.3  lis_vector_axpy

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

Description
Calculate the sum of the vectors \( y = \alpha x + y \).

Input
\begin{itemize}
  \item \textbf{alpha} \quad The scalar value
  \item \textbf{x}, \textbf{y} \quad The vectors
\end{itemize}

Output
\begin{itemize}
  \item \textbf{y} \quad \alpha x + y \ (vector \ y \ is \ overwritten)
  \item \textbf{ierr} \quad The return code
\end{itemize}

6.3.4  lis_vector_xpay

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

Description
Calculate the sum of the vectors \( y = x + \alpha y \).

Input
\begin{itemize}
  \item \textbf{alpha} \quad The scalar value
  \item \textbf{x}, \textbf{y} \quad The vectors
\end{itemize}

Output
\begin{itemize}
  \item \textbf{y} \quad x + \alpha y \ (vector \ y \ is \ overwritten)
  \item \textbf{ierr} \quad The return code
\end{itemize}
6.3.5 lis_vector_axpyz

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

Description
Calculate the sum of the vectors \( z = \alpha x + y \).

Input
alpha       The scalar value
x, y        The vectors

Output
z           \( \alpha x + y \)
ierr        The return code

6.3.6 lis_vector_scale

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

Description
Multiply vector \( x \) by scalar \( \alpha \).

Input
alpha       The scalar value
x            The vector

Output
x            \( \alpha x \) (vector \( x \) is overwritten)
ierr        The return code
6.3.7 lis_vector_pmul

C     LIS_INT lis_vector_pmul(LIS_VECTOR x, LIS_VECTOR y, LIS_VECTOR z)
Fortran subroutine lis_vector_pmul(LIS_VECTOR x, LIS_VECTOR y, LIS_VECTOR z,
     LIS_INTEGER ierr)

Description
Multiply each element of vector x by the corresponding element of y.

Input
x, y     The vectors

Output
z     The vector that stores the multiplied elements of x
ierr     The return code

6.3.8 lis_vector_pdiv

C     LIS_INT lis_vector_pdiv(LIS_VECTOR x, LIS_VECTOR y, LIS_VECTOR z)
Fortran subroutine lis_vector_pdiv(LIS_VECTOR x, LIS_VECTOR y, LIS_VECTOR z,
     LIS_INTEGER ierr)

Description
Divide each element of vector x by the corresponding element of y.

Input
x, y     The vectors

Output
z     The vector that stores the divided elements of x
ierr     The return code
6.3.9  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 elements of vector x.

Input
value   The scalar value to be assigned
x       The vector

Output
x       The vector with the value assigned to the elements
ierr    The return code

6.3.10 lis_vector_abs

C     LIS_INT lis_vector_abs(LIS_VECTOR x)
Fortran subroutine lis_vector_abs(LIS_VECTOR x, LIS_INTEGER ierr)

Description
Get the absolute values of the elements of vector x.

Input
x       The vector

Output
x       The vector that stores the absolute values
ierr    The return code
6.3.11 lis_vector_reciprocal

C      LIS_INT lis_vector_reciprocal(LIS_VECTOR x)
Fortran subroutine lis_vector_reciprocal(LIS_VECTOR x, LIS_INTEGER ierr)

Description
Get the reciprocal values of the elements of vector x.

Input
x       The vector

Output
x       The vector that stores the reciprocal values
ierr    The return code

6.3.12 lis_vector_conjugate

C      LIS_INT lis_vector_conjugate(LIS_VECTOR x)
Fortran subroutine lis_vector_conjugate(LIS_VECTOR x, LIS_INTEGER ierr)

Description
Get the conjugate complex values of the elements of vector x.

Input
x       The vector

Output
x       The vector that stores the conjugate complex values
ierr    The return code
6.3.13    lis_vector_shift

C    LIS_INT lis_vector_shift(LIS_SCALAR sigma, LIS_VECTOR x)
Fortran subroutine lis_vector_shift(LIS_SCALAR sigma, LIS_VECTOR x,
    LIS_INTEGER ierr)

Description
    Shift vector \( x \).

Input
    sigma    The amount of the shift
    x        The vector

Output
    x        The vector that stores the shifted elements \( x_i - \sigma \)
    ierr     The return code


6.3.14 lis_vector_dot

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

Description
Calculate the Hermitian inner product $x^H y$.

Input
x, y            The vectors

Output
value           The inner product
ierr            The return code

6.3.15 lis_vector_nhdot

C    LIS_INT lis_vector_nhdot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR *value)
Fortran subroutine lis_vector_nhdot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR value,
                                   LIS_INTEGER ierr)

Description
Calculate the non-Hermitian inner product $x^T y$.

Input
x, y            The vectors

Output
value           The inner product
ierr            The return code
6.3.16  lis_vector_nrm1

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

Description
Calculate the 1-norm of vector x.

Input
x                The vector

Output
value             The 1-norm of the vector
ierr              The return code

6.3.17  lis_vector_nrm2

C    LIS_INT lis_vector_nrm2(LIS_VECTOR x, LIS_REAL *value)
Fortran subroutine lis_vector_nrm2(LIS_VECTOR x, LIS_REAL value, LIS_INTEGER ierr)

Description
Calculate the 2-norm of vector x.

Input
x                The vector

Output
value             The 2-norm of the vector
ierr              The return code
6.3.18 lis_vector_nrm1

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

Description
Calculate the infinity norm of vector x.

Input
x The vector

Output
value The infinity norm of the vector
ierr The return code

6.3.19 lis_vector_sum

C LIS_INT lis_vector_sum(LIS_VECTOR x, LIS_SCALAR *value)
Fortran subroutine lis_vector_sum(LIS_VECTOR x, LIS_SCALAR value, LIS_INTEGER ierr)

Description
Calculate the sum of the elements of vector x.

Input
x The vector

Output
value The sum of the vector elements
ierr The return code
6.3.20  lis_matrix_set_blocksize

```c
LIS_INT lis_matrix_set_blocksize(LIS_MATRIX A, LIS_INT bnr, LIS_INT bnc,
LIS_INT row[], LIS_INT col[])
```

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

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

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

**Output**
- **ierr**  The return code
6.3.21 lis_matrix_convert

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

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

Input

$A_{in}$ The source matrix

Output

$A_{out}$ The destination matrix

ierr The return code

Note
The storage format of $A_{out}$ is set by lis_matrix_set_type. The block size of the BSR, BSC, and VBR formats is set by lis_matrix_set_blocksize. See lis-($VERSION)/test/test2.c and lis-($VERSION)/test/test2f.F90.

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

<table>
<thead>
<tr>
<th>Src</th>
<th>CSR</th>
<th>CSC</th>
<th>MSR</th>
<th>DIA</th>
<th>ELL</th>
<th>JAD</th>
<th>BSR</th>
<th>BSC</th>
<th>VBR</th>
<th>COO</th>
<th>DNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CSC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COO</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CSR</td>
<td>CSR</td>
<td>CSR</td>
<td>CSR</td>
<td>CSR</td>
<td>CSR</td>
<td>CSR</td>
<td>CSR</td>
</tr>
</tbody>
</table>

105
6.3.22 lis_matrix_copy

```c
LIS_INT lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout)
```

Fortran subroutine lime_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

6.3.23 lis_matrix_axpy

```c
LIS_INT lis_matrix_axpy(LIS_SCALAR alpha, LIS_MATRIX A, LIS_MATRIX B)
```

Fortran subroutine lis_matrix_axpy(LIS_SCALAR alpha, LIS_MATRIX A, LIS_MATRIX B,
LIS_INTEGER ierr)

**Description**
Calculate the sum of the matrices \( B = \alpha A + B \).

**Input**
- **alpha** The scalar value
- **A, B** The matrices

**Output**
- **B** \( \alpha A + B \) (matrix \( B \) is overwritten)
- **ierr** The return code

Matrices \( A, B \) must be in the DNS format.
6.3.24 lis_matrix_xpay

C LIS_INT lis_matrix_xpay(LIS_SCALAR alpha, LIS_MATRIX A, LIS_MATRIX B)
Fortran subroutine lis_matrix_xpay(LIS_SCALAR alpha, LIS_MATRIX A, LIS_MATRIX B,
LIS_INTEGER ierr)

Description
Calculate the sum of the matrices \( B = A + \alpha B \).

Input
\begin{itemize}
\item \textit{alpha} \quad The scalar value
\item \textit{A, B} \quad The matrices
\end{itemize}

Output
\begin{itemize}
\item \textit{B} \quad \( A + \alpha B \) (matrix \( B \) is overwritten)
\item \textit{ierr} \quad The return code
\end{itemize}

Note
Matrices \( A, B \) must be in the DNS format.

6.3.25 lis_matrix_axpyz

C LIS_INT lis_matrix_axpyz(LIS_SCALAR alpha, LIS_MATRIX A, LIS_MATRIX B,
LIS_MATRIX C)
Fortran subroutine lis_matrix_axpyz(LIS_SCALAR alpha, LIS_MATRIX A, LIS_MATRIX B,
LIS_MATRIX C, LIS_INTEGER ierr)

Description
Calculate the sum of the matrices \( C = \alpha A + B \).

Input
\begin{itemize}
\item \textit{alpha} \quad The scalar value
\item \textit{A, B} \quad The matrices
\end{itemize}

Output
\begin{itemize}
\item \textit{C} \quad \( \alpha A + B \)
\item \textit{ierr} \quad The return code
\end{itemize}

Note
Matrices \( A, B, \) and \( C \) must be in the DNS format.
6.3.26 lis_matrix_scale

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

Description
Scale matrix $A$ and vector $b$.

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

6.3.27 lis_matrix_get_diagonal

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

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

Input
- $A$ : The matrix

Output
- $d$ : The vector that stores the diagonal elements of the matrix
- ierr : The return code

108
### 6.3.28 lis_matrix_shift_diagonal

C    LIS_INT lis_matrix_shift_diagonal(LIS_MATRIX A, LIS_SCALAR sigma)
Fortran subroutine lis_matrix_shift_diagonal(LIS_MATRIX A, LIS_SCALAR sigma, 
   LIS_INTEGER ierr)

**Description**
Shift the diagonal elements of matrix $A$.

**Input**
- **sigma** The amount of the shift
- **A** The matrix

**Output**
- **A** The shifted matrix $A - \sigma E$
- **ierr** The return code

### 6.3.29 lis_matrix_shift_matrix

C    LIS_INT lis_matrix_shift_matrix(LIS_MATRIX A, LIS_MATRIX B, 
   LIS_SCALAR sigma)
Fortran subroutine lis_matrix_shift_matrix(LIS_MATRIX A, LIS_MATRIX B, 
   LIS_SCALAR sigma, LIS_INTEGER ierr)

**Description**
Shift the matrix elements of matrix $A$.

**Input**
- **sigma** The amount of the shift
- **A** The matrix
- **B** The matrix

**Output**
- **A** The shifted matrix $A - \sigma B$
- **ierr** The return code
6.3.30 lis_matvec

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

Description
Calculate the matrix-vector product \( y = Ax \).

Input
A     The matrix
x     The vector

Output
y     \( Ax \)
ierr   The return code

6.3.31 lis_matvech

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

Description
Calculate the matrix-vector product \( y = A^H x \).

Input
A     The matrix
x     The vector

Output
y     \( A^H x \)
ierr   The return code
6.4 Solving Linear Equations

6.4.1 lis_solver_create

\begin{verbatim}
C     LIS_INT lis_solver_create(LIS_SOLVER *solver)
Fortran subroutine lis_solver_create(LIS_SOLVER solver, LIS_INTEGER ierr)
\end{verbatim}

**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

\begin{verbatim}
C     LIS_INT lis_solver_destroy(LIS_SOLVER solver)
Fortran subroutine lis_solver_destroy(LIS_SOLVER solver, LIS_INTEGER ierr)
\end{verbatim}

**Description**
Destroy the solver.

**Input**

- **solver** The solver to be destroyed

**Output**

- **ierr** The return code
6.4.3  lis_precon_create

C    LIS_INT lis_precon_create(LIS_SOLVER solver, LIS_PRECON *precon)
Fortran subroutine lis_precon_create(LIS_SOLVER solver, LIS_PRECON precon,
       LIS_INTEGER ierr)

Description
   Create the preconditioner.

Input
   None

Output
   solver       The solver
   precon       The preconditioner
   ierr         The return code

6.4.4  lis_precon_destroy

C    LIS_INT lis_precon_destroy(LIS_PRECON precon)
Fortran subroutine lis_precon_destroy(LIS_PRECON precon, LIS_INTEGER ierr)

Description
   Destroy the preconditioner.

Input
   precon       The preconditioner to be destroyed

Output
   ierr         The return code
6.4.5  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.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>-i {cg</td>
<td>1}</td>
</tr>
<tr>
<td>BiCG</td>
<td>-i {bicg</td>
<td>2}</td>
</tr>
<tr>
<td>CGS</td>
<td>-i {cgs</td>
<td>3}</td>
</tr>
<tr>
<td>BiCGSTAB</td>
<td>-i {bicgstab</td>
<td>4}</td>
</tr>
<tr>
<td>BiCGSTAB(l)</td>
<td>-i {bicgstab</td>
<td>5}</td>
</tr>
<tr>
<td>GPBiCG</td>
<td>-i {gpbicg</td>
<td>6}</td>
</tr>
<tr>
<td>TFQMR</td>
<td>-i {tfqmr</td>
<td>7}</td>
</tr>
<tr>
<td>Orthomin(m)</td>
<td>-i {orthomin</td>
<td>8}</td>
</tr>
<tr>
<td>GMRES(m)</td>
<td>-i {gmres</td>
<td>9}</td>
</tr>
<tr>
<td>Jacobi</td>
<td>-i {jacobi</td>
<td>10}</td>
</tr>
<tr>
<td>Gauss-Seidel</td>
<td>-i {gs</td>
<td>11}</td>
</tr>
<tr>
<td>SOR</td>
<td>-i {sor</td>
<td>12}</td>
</tr>
<tr>
<td>BiCGSafe</td>
<td>-i {bicgsafe</td>
<td>13}</td>
</tr>
<tr>
<td>CR</td>
<td>-i {cr</td>
<td>14}</td>
</tr>
<tr>
<td>BiCR</td>
<td>-i {bicr</td>
<td>15}</td>
</tr>
<tr>
<td>CRS</td>
<td>-i {crs</td>
<td>16}</td>
</tr>
<tr>
<td>BiCRSTAB</td>
<td>-i {bicrstab</td>
<td>17}</td>
</tr>
<tr>
<td>GPBiCR</td>
<td>-i {gpbicr</td>
<td>18}</td>
</tr>
<tr>
<td>BiCRSafe</td>
<td>-i {bicrsafe</td>
<td>19}</td>
</tr>
<tr>
<td>FGMRES(m)</td>
<td>-i {fgmres</td>
<td>20}</td>
</tr>
<tr>
<td>IDR(s)</td>
<td>-i {idrs</td>
<td>21}</td>
</tr>
<tr>
<td>IDR(1)</td>
<td>-i {idr1</td>
<td>22}</td>
</tr>
<tr>
<td>MINRES</td>
<td>-i {minres</td>
<td>23}</td>
</tr>
<tr>
<td>COCG</td>
<td>-i {cocg</td>
<td>24}</td>
</tr>
<tr>
<td>COCR</td>
<td>-i {coccr</td>
<td>25}</td>
</tr>
</tbody>
</table>
### Options for Preconditioners (Default: -p none)

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>-p {none</td>
<td>0}</td>
</tr>
<tr>
<td>Jacobi</td>
<td>-p {jacobi</td>
<td>1}</td>
</tr>
<tr>
<td>ILU(k)</td>
<td>-p {ilu</td>
<td>2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The fill level k</td>
</tr>
<tr>
<td>SSOR</td>
<td>-p {ssor</td>
<td>3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The relaxation coefficient ω (0 &lt; ω &lt; 2)</td>
</tr>
<tr>
<td>Hybrid</td>
<td>-p {hybrid</td>
<td>4}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The linear solver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_maxiter [25]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The maximum number of iterations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_tol [1.0e-3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The convergence tolerance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_omega [1.5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The relaxation coefficient ω of the SOR (0 &lt; ω &lt; 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_ell [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The degree l of the BiCGSTAB(l)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-hybrid_restart [40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The restart values of the GMRES and Orthomin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-is_alpha [1.0]</td>
</tr>
<tr>
<td>I+S</td>
<td>-p {is</td>
<td>5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-is_m [3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parameter $m$ of $I + αS^{(m)}$</td>
</tr>
<tr>
<td>SAINV</td>
<td>-p {sainv</td>
<td>6}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The drop criterion</td>
</tr>
<tr>
<td>SA-AMG</td>
<td>-p {saamg</td>
<td>7}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Select the unsymmetric version (The matrix structure must be symmetric)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-saamg_theta [0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The drop criterion $a_{ij}^2 \leq θ^2</td>
</tr>
<tr>
<td>Crout ILU</td>
<td>-p {iluc</td>
<td>8}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The drop criterion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-iluc_rate [5.0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The ratio of the maximum fill-in</td>
</tr>
<tr>
<td>ILUT</td>
<td>-p {ilut</td>
<td>9}</td>
</tr>
<tr>
<td>Additive</td>
<td>-add true</td>
<td>-adds_iter [1]</td>
</tr>
<tr>
<td>Schwarz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Other Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-maxiter [1000]</code></td>
<td>The maximum number of iterations</td>
</tr>
<tr>
<td><code>-tol [1.0e-12]</code></td>
<td>The convergence tolerance <code>tol</code></td>
</tr>
<tr>
<td><code>-tol_w [1.0]</code></td>
<td>The convergence tolerance <code>tol_w</code></td>
</tr>
<tr>
<td><code>-print [0]</code></td>
<td>The output of the residual history</td>
</tr>
<tr>
<td>-print {none</td>
<td>0}</td>
</tr>
<tr>
<td>-print {mem</td>
<td>1}</td>
</tr>
<tr>
<td>-print {out</td>
<td>2}</td>
</tr>
<tr>
<td>-print {all</td>
<td>3}</td>
</tr>
<tr>
<td><code>-scale [0]</code></td>
<td>The scaling (The result will overwrite the original matrix and vectors)</td>
</tr>
<tr>
<td>-scale {none</td>
<td>0}</td>
</tr>
<tr>
<td>-scale {jacobi</td>
<td>1}</td>
</tr>
<tr>
<td>-scale {symm_diag</td>
<td>2}</td>
</tr>
<tr>
<td><code>-initx_zeros [1]</code></td>
<td>The behavior of the initial vector $x_0$</td>
</tr>
<tr>
<td>-initx_zeros {false</td>
<td>0}</td>
</tr>
<tr>
<td>-initx_zeros {true</td>
<td>1}</td>
</tr>
<tr>
<td><code>-conv_cond [0]</code></td>
<td>The convergence condition</td>
</tr>
<tr>
<td>-conv_cond {nrm2_r</td>
<td>0}</td>
</tr>
<tr>
<td>-conv_cond {nrm2_b</td>
<td>1}</td>
</tr>
<tr>
<td>-conv_cond {nrm1_b</td>
<td>2}</td>
</tr>
<tr>
<td><code>-omp_num_threads [t]</code></td>
<td>The number of threads (t represents the maximum number of threads)</td>
</tr>
<tr>
<td><code>-storage [0]</code></td>
<td>The matrix storage format</td>
</tr>
<tr>
<td><code>-storage_block [2]</code></td>
<td>The block size of the BSR and BSC formats</td>
</tr>
<tr>
<td><code>-f [0]</code></td>
<td>The precision of the linear solver</td>
</tr>
<tr>
<td>-f {double</td>
<td>0}</td>
</tr>
<tr>
<td>-f {quad</td>
<td>1}</td>
</tr>
</tbody>
</table>
6.4.6 lis_solver_set_optionC

```plaintext
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
```plaintext
 solver       The solver
 ierr         The return code
```

6.4.7 lis_solve

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

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

Note
If the option `-initx_zeros {false|0}` is specified, the initial vector is given by the argument $x$.
Otherwise, all the components of the initial vector are set to 0.

This function returns 0 if the status of the solver (solver->retcode) is LIS_BREAKDOWN or LIS_MAXITER.
See also the function lis_solver_get_status().
6.4.8 lis_solve_kernel

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

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

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

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

**Note**
See lis-($VERSION$)/src/esolver/lis_esolver_ii.c, which computes the smallest eigenvalue by calling `lis_solve_kernel` multiple times, for example.
This function returns 0 if the status of the solver (`solver->retcode`) is `LIS_BREAKDOWN` or `LIS_MAXITER`.
See also the function `lis_solver_get_status()`.
6.4.9  lis_solver_get_status

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

Description
  Get the status from the solver.

Input
  solver       The solver

Output
  status      The status
  ierr        The return code

Note
  This function returns the status of the solver (solver->retcode).

6.4.10 lis_solver_get_iter

C    LIS_INT lis_solver_get_iter(LIS_SOLVER solver, LIS_INT *iter)
Fortran subroutine lis_solver_get_iter(LIS_SOLVER solver, LIS_INTEGER iter,
                             LIS_INTEGER ierr)

Description
  Get the number of iterations from the solver.

Input
  solver       The solver

Output
  iter        The number of iterations
  ierr        The return code
6.4.11  lis_solver_get_iterex

C     LIS_INT lis_solver_get_iterex(LIS_SOLVER solver, LIS_INT *iter,
                  LIS_INT *iter_double, LIS_INT *iter_quad)
Fortran subroutine lis_solver_get_iterex(LIS_SOLVER solver, LIS_INTEGER iter,
                  LIS_INTEGER iter_double, LIS_INTEGER iter_quad, LIS_INTEGER ierr)

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

Input
solver          The solver

Output
iter            The number of iterations
iter_double     The number of double precision iterations
iter_quad       The number of double-double (quadruple) precision iterations
ierr            The return code

6.4.12  lis_solver_get_time

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

Description
Get the execution time from the solver.

Input
solver          The solver

Output
time            The time in seconds of the execution
ierr            The return code
6.4.13 lis_solver_get_timeex

C
LIS_INT lis_solver_get_timeex(LIS_SOLVER solver, double *time,
   double *itime, double *ptime, double *p_c_time, double *p_i_time)

Fortran subroutine lis_solver_get_timeex(LIS_SOLVER solver, real*8 time,
   real*8 itime, real*8 ptime, real*8 p_c_time, real*8 p_i_time,
   LIS_INTEGER ierr)

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

Input
solver
The solver

Output
time
The total time in seconds
itime
The time in seconds of the iterations
ptime
The time in seconds of the preconditioning
p_c_time
The time in seconds of the creation of the preconditioner
p_i_time
The time in seconds of the iterations in the preconditioner
ierr
The return code

6.4.14 lis Solver get residual norm

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
Get the relative residual norm ||b - Ax||_2/||b||_2 from the solver.

Input
solver
The solver

Output
residual
The relative residual norm ||b - Ax||_2/||b||_2
ierr
The return code
6.4.15  lis_solver_get_rhistory

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

Description
    Get the residual history from the solver.

Input
    solver                            The solver

Output
    v                                 The vector
    ierr                              The return code

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

```c
LIS_INT lis_solver_get_solver(LIS_SOLVER solver, LIS_INT *nsol)
```

Fortran subroutine:
```
lis_solver_get_solver(LIS_SOLVER solver, LIS_INTEGER nsol, LIS_INTEGER ierr)
```

**Description**
Get the solver number from the solver.

**Input**
- `solver`: The solver

**Output**
- `nsol`: The solver number
- `ierr`: The return code

6.4.17 lis_solver_get_precon

```c
LIS_INT lis_solver_get_precon(LIS_SOLVER solver, LIS_INT *precon_type)
```

Fortran subroutine:
```
lis_solver_get_precon(LIS_SOLVER solver, LIS_INTEGER precon_type, LIS_INTEGER ierr)
```

**Description**
Get the preconditioner number from the solver.

**Input**
- `solver`: The solver

**Output**
- `precon_type`: The preconditioner number
- `ierr`: The return code
6.4.18  lis_solver_get_solvername

```c
LIS_INT lis_solver_get_solvername(LIS_INT nsol, char *name)
```

Fortran subroutine
```
lis_solver_get_solvername(LIS_INTEGER nsol, character name, LIS_INTEGER ierr)
```

**Description**
Get the solver name from the solver number.

**Input**
- `nsol` The solver number

**Output**
- `name` The solver name
- `ierr` The return code

6.4.19  lis_solver_get_preconname

```c
LIS_INT lis_solver_get_preconname(LIS_INT precon_type, char *name)
```

Fortran subroutine
```
lis_solver_get_preconname(LIS_INTEGER precon_type, character name, LIS_INTEGER ierr)
```

**Description**
Get the preconditioner name from the preconditioner number.

**Input**
- `precon_type` The preconditioner number

**Output**
- `name` The preconditioner name
- `ierr` The return code
6.5 Solving Eigenvalue Problems

6.5.1 lis_esolver_create

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

Description
Create the eigensolver.

Input
None

Output
esolver          The eigensolver
ierr             The return code

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

6.5.2 lis_esolver_destroy

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

Description
Destroy the eigensolver.

Input
esolver          The eigensolver to be destroyed

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.

<table>
<thead>
<tr>
<th>Eigensolver</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>-e {pi</td>
<td>1}</td>
</tr>
<tr>
<td>Inverse</td>
<td>-e {ii</td>
<td>2}</td>
</tr>
<tr>
<td>Rayleigh Quotient</td>
<td>-e {rq</td>
<td>3}</td>
</tr>
<tr>
<td>CG</td>
<td>-e {cg</td>
<td>4}</td>
</tr>
<tr>
<td>CR</td>
<td>-e {cr</td>
<td>5}</td>
</tr>
<tr>
<td>Subspace</td>
<td>-e {si</td>
<td>6}</td>
</tr>
<tr>
<td>Lanczos</td>
<td>-e {li</td>
<td>7}</td>
</tr>
<tr>
<td>Arnoldi</td>
<td>-e {ai</td>
<td>8}</td>
</tr>
<tr>
<td>Generalized Power</td>
<td>-e {gpi</td>
<td>9}</td>
</tr>
<tr>
<td>Generalized Inverse</td>
<td>-e {gii</td>
<td>10}</td>
</tr>
<tr>
<td>Generalized Rayleigh Quotient</td>
<td>-e {grqi</td>
<td>11}</td>
</tr>
<tr>
<td>Generalized CG</td>
<td>-e {gcg</td>
<td>12}</td>
</tr>
<tr>
<td>Generalized CR</td>
<td>-e {gcr</td>
<td>13}</td>
</tr>
<tr>
<td>Generalized Subspace</td>
<td>-e {gai</td>
<td>14}</td>
</tr>
<tr>
<td>Generalized Lanczos</td>
<td>-e {gli</td>
<td>15}</td>
</tr>
<tr>
<td>Generalized Arnoldi</td>
<td>-e {gai</td>
<td>16}</td>
</tr>
</tbody>
</table>
# Options for Preconditioners (Default: `-p none`)

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Option</th>
<th>Auxiliary Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>`-p {none</td>
<td>0}`</td>
</tr>
<tr>
<td>Jacobi</td>
<td>`-p {jacobi</td>
<td>1}`</td>
</tr>
<tr>
<td>ILU(k)</td>
<td>`-p {ilu</td>
<td>2}`</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-ssor_omega [1.0]</code> The relaxation coefficient $\omega$ (0 &lt; $\omega$ &lt; 2)</td>
</tr>
<tr>
<td>SSOR</td>
<td>`-p {ssor</td>
<td>3}`</td>
</tr>
<tr>
<td>Hybrid</td>
<td>`-p {hybrid</td>
<td>4}`</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-hybrid_maxiter [25]</code> The maximum number of iterations</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-hybrid_tol [1.0e-3]</code> The convergence tolerance</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-hybrid_ell [2]</code> The degree $l$ of the BiCGSTAB(l)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-hybrid_restart [40]</code> The restart values of the GMRES and Orthomin</td>
</tr>
<tr>
<td>I+S</td>
<td>`-p {is</td>
<td>5}`</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-is_m [3]</code> The parameter $m$ of $I + \alpha S^{(m)}$</td>
</tr>
<tr>
<td>SAINV</td>
<td>`-p {sainv</td>
<td>6}`</td>
</tr>
<tr>
<td>SA-AMG</td>
<td>`-p {saamg</td>
<td>7}`</td>
</tr>
<tr>
<td></td>
<td></td>
<td>`-saamg_theta [0.05</td>
</tr>
<tr>
<td>Crout ILU</td>
<td>`-p {iluc</td>
<td>8}`</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-iluc_rate [5.0]</code> The ratio of the maximum fill-in</td>
</tr>
<tr>
<td>ILUT</td>
<td>`-p {ilut</td>
<td>9}`</td>
</tr>
<tr>
<td>Additive</td>
<td><code>-adds true</code></td>
<td><code>-adds_iter [1]</code> The number of iterations</td>
</tr>
<tr>
<td>Schwarz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>-emaxiter [1000]</td>
<td>The maximum number of iterations</td>
<td></td>
</tr>
<tr>
<td>-etol [1.0e-12]</td>
<td>The convergence tolerance</td>
<td></td>
</tr>
<tr>
<td>-eprint [0]</td>
<td>The output of the residual history</td>
<td></td>
</tr>
<tr>
<td>-eprint {none</td>
<td>0}</td>
<td>None</td>
</tr>
<tr>
<td>-eprint {mem</td>
<td>1}</td>
<td>Save the residual history</td>
</tr>
<tr>
<td>-eprint {out</td>
<td>2}</td>
<td>Output it to the standard output</td>
</tr>
<tr>
<td>-eprint {all</td>
<td>3}</td>
<td>Save the residual history and output it to the standard output</td>
</tr>
<tr>
<td>-ie [ii]</td>
<td>The inner eigensolver used in Subspace, Lanczos, and Arnoldi</td>
<td></td>
</tr>
<tr>
<td>-ige [gii]</td>
<td>The inner eigensolver used in Generalized Subspace, Generalized Lanczos, and Generalized Arnoldi</td>
<td></td>
</tr>
<tr>
<td>-shift [0.0]</td>
<td>The amount of the real part of the shift $\sigma$ to compute $A - \sigma B$</td>
<td></td>
</tr>
<tr>
<td>-shift_im [0.0]</td>
<td>The amount of the imaginary part of the shift $\sigma$</td>
<td></td>
</tr>
<tr>
<td>-initx_ones [1]</td>
<td>The behavior of the initial vector $x_0$</td>
<td></td>
</tr>
<tr>
<td>-initx_ones {false</td>
<td>0}</td>
<td>The components are given by the argument $x$ of the function lis_esolve()</td>
</tr>
<tr>
<td>-initx_ones {true</td>
<td>1}</td>
<td>All the components are set to 1</td>
</tr>
<tr>
<td>-omp_num_threads [t]</td>
<td>The number of threads</td>
<td></td>
</tr>
<tr>
<td>(t represents the maximum number of threads)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-estorage [0]</td>
<td>The matrix storage format</td>
<td></td>
</tr>
<tr>
<td>-estorage_block [2]</td>
<td>The block size of the BSR and BSC formats</td>
<td></td>
</tr>
<tr>
<td>-ef [0]</td>
<td>The precision of the eigensolver</td>
<td></td>
</tr>
<tr>
<td>-ef {double</td>
<td>0}</td>
<td>Double precision</td>
</tr>
<tr>
<td>-ef {quad</td>
<td>1}</td>
<td>Double-double (quadruple) precision</td>
</tr>
<tr>
<td>-rval [0]</td>
<td>The Ritz values</td>
<td></td>
</tr>
<tr>
<td>-rval {false</td>
<td>0}</td>
<td>The eigenpairs are computed based on the Ritz values</td>
</tr>
<tr>
<td>-rval {true</td>
<td>1}</td>
<td>Only the Ritz values are computed</td>
</tr>
</tbody>
</table>
6.5.4 lis_esolver_set_optionC

<table>
<thead>
<tr>
<th>C</th>
<th>LIS_INT lis_esolver_set_optionC(LIS_ESOLVER esolver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, LIS_INTEGER ierr)</td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>C</th>
<th>LIS_INT lis_esolve(LIS_MATRIX A, LIS_VECTOR x,</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIS_SCALAR evalue, LIS_ESOLVER esolver)</td>
<td></td>
</tr>
<tr>
<td>Fortran subroutine lis_esolve(LIS_MATRIX A, LIS_VECTOR x,</td>
<td></td>
</tr>
<tr>
<td>LIS_SCALAR evalue, LIS_ESOLVER esolver, LIS_INTEGER ierr)</td>
<td></td>
</tr>
</tbody>
</table>

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

**Input**
A The matrix
x The initial vector
esolver The eigensolver

**Output**
evalue The default eigenvalue
x The associated eigenvector
esolver The number of iterations, the execution time, etc.
ierr The return code

**Note**
The default eigenvalue is the eigenvalue of mode 0. In the following sections, the default eigenpair means the eigenvalue of mode 0 and the associated eigenvector.

If the option `-initx ones {false|0}` is specified, the initial vector is given by the argument x. Otherwise, all the components of the initial vector are set to 1.

This function returns 0 if the status of the eigensolver (esolver->retcode) is LIS_MAXITER. See also the function lis_esolver_get_status().
6.5.6 lis_gesolve

C    LIS_INT lis_gesolve(LIS_MATRIX A, LIS_MATRIX B, 
                        LIS_VECTOR x, LIS_SCALAR eval, LIS_ESOLVER esolver)
Fortran subroutine lis_gesolve(LIS_MATRIX A, LIS_MATRIX B, 
                              LIS_VECTOR x, LIS_SCALAR eval, LIS_ESOLVER esolver, LIS_INTEGER ierr)

Description
Solve the generalized eigenvalue problem $Ax = \lambda Bx$ with the specified eigensolver.

Input
A    The matrix
B    The matrix
x    The initial vector
esolver    The eigensolver

Output
eval    The default eigenvalue
x    The associated eigenvector
esolver    The number of iterations, the execution time, etc.
ierr    The return code

Note
If the option -initx.ones [false|0] is specified, the initial vector is given by the argument x. Otherwise, all the components of the initial vector are set to 1.
This function returns 0 if the status of the eigensolver (esolver->retcode) is LIS_MAXITER. See also the function lis_esolver_get_status().
6.5.7 lis_esolver_get_status

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

Description
Get the status from the eigensolver.

Input
esolver            The eigensolver

Output
status             The status
ierr               The return code

Note
This function returns the status of the eigensolver (esolver->retcode).

6.5.8 lis_esolver_get_iter

C     LIS_INT lis_esolver_get_iter(LIS_ESOLVER esolver, LIS_INT *iter)
Fortran subroutine lis_esolver_get_iter(LIS_ESOLVER esolver, LIS_INTEGER iter,
                                         LIS_INTEGER ierr)

Description
Get the number of iterations for the default eigenpair from the eigensolver.

Input
esolver            The eigensolver

Output
iter               The number of iterations
ierr               The return code
6.5.9 lis_esolver_get_iterex

C    LIS_INT lis_esolver_get_iterex(LIS_ESOLVER esolver, LIS_INT *iter,
                  LIS_INT *iter_double, LIS_INT *iter_quad)
Fortran subroutine lis_esolver_get_iterex(LIS_ESOLVER esolver, LIS_INTEGER iter,
                  LIS_INTEGER iter_double, LIS_INTEGER iter_quad, LIS_INTEGER ierr)

Description
Get the detailed information on the number of iterations for the default eigenpair from the eigensolver.

Input
- esolver: The eigensolver

Output
- iter: The number of iterations
- iter_double: The number of double precision iterations
- iter_quad: The number of double-double (quadruple) precision iterations
- ierr: The return code

6.5.10 lis_esolver_get_time

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

Description
Get the execution time for the default eigenpair from the eigensolver.

Input
- esolver: The eigensolver

Output
- time: The time in seconds of the execution
- ierr: The return code
### 6.5.11 lis_esolver_get_timeex

```c
C
LIS_INT lis_esolver_get_timeex(LIS_ESOLVER esolver, double *time,
   double *itime, double *ptime, double *p_c_time, double *p_i_time)
```

Fortran subroutine:
```fortran
Fortran subroutine lis_esolver_get_timeex(LIS_ESOLVER esolver, real*8 time,
   real*8 itime, real*8 ptime, real*8 p_c_time, real*8 p_i_time,
   LIS_INTEGER ierr)
```

**Description**
Get the detailed information on the execution time for the default eigenpair from the eigensolver.

**Input**
- `esolver`: The eigensolver

**Output**
- `time`: The total time in seconds
- `itime`: The time in seconds of the iterations
- `ptime`: The time in seconds of the preconditioning
- `p_c_time`: The time in seconds of the creation of the preconditioner
- `p_i_time`: The time in seconds of the iterations in the preconditioner
- `ierr`: The return code

### 6.5.12 lis_esolver_get_residualnorm

```c
C
LIS_INT lis_esolver_get_residualnorm(LIS_ESOLVER esolver,
   LIS_REAL *residual)
```

Fortran subroutine:
```fortran
Fortran subroutine lis_esolver_get_residualnorm(LIS_ESOLVER esolver,
   LIS_REAL residual, LIS_INTEGER ierr)
```

**Description**
Get the relative residual norm $\frac{\|\lambda x - (B^{-1})Ax\|_2}{\|\lambda x\|_2}$ for the default eigenpair from the eigensolver.

**Input**
- `esolver`: The eigensolver

**Output**
- `residual`: The relative residual norm $\frac{\|\lambda x - Ax\|_2}{\|\lambda x\|_2}$
- `ierr`: The return code
6.5.13 lis_esolver_get_rhistory

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

Description
Get the residual history for the default eigenpair from the eigensolver.

Input
esolver     The eigensolver

Output
v     The vector
ierr     The return code

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

6.5.14 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
Get all the eigenvalues from the eigensolver.

Input
esolver     The eigensolver

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

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

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

Description
   Get all the eigenvectors from the eigensolver.

Input
   esolver        The eigensolver

Output
   M              The matrix in the COO format which stores the eigenvectors
   ierr           The return code

Note
   Matrix M must be created in advance with the function lis_matrix_create. The i-th eigenvector is stored on the i-th column of the matrix M. See lis-($\$VERSION)/test/etest5.c.

6.5.16 lis_esolver_get_residualnorms

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

Description
   Get the relative residual norms $||\lambda x - (B^{-1})Ax||_2/||\lambda x||_2$ of all the eigenpairs from the eigensolver.

Input
   esolver        The eigensolver

Output
   v              The vector which stores the residual norms
   ierr           The return code

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

C     LIS_INT lis_esolver_get_iters(LIS_ESOLVER esolver, LIS_VECTOR v)
Fortran subroutine lis_esolver_get_iter(LIS_ESOLVER esolver,
   LIS_VECTOR v, LIS_INTEGER ierr)

Description
Get the numbers of iterations of all the eigenpairs from the eigensolver.

Input
esolver       The eigensolver

Output
v             The vector which stores the numbers of iterations
ierr          The return code

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

6.5.18 lis_esolver_get_specific_evalue

C     LIS_INT lis_esolver_get_specific_evalue(LIS_ESOLVER esolver,
   LIS_INT mode, LIS_SCALAR *evalue)
Fortran subroutine lis_esolver_get_specific_evalue(LIS_ESOLVER esolver,
   LIS_INT mode, LIS_SCALAR evalue, LIS_INTEGER ierr)

Description
Get the specified eigenvalue from the eigensolver.

Input
esolver       The eigensolver
mode          The mode number of the eigenvalue

Output
evalue        The eigenvalue
ierr          The return code
### 6.5.19 lis_esolver_get_specific_evector

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LIS_INT lis_esolver_get_specific_evector(LIS_ESOLVER esolver, LIS_INT mode, LIS_VECTOR x)</td>
<td></td>
</tr>
</tbody>
</table>

Fortran subroutine `lis_esolver_get_specific_evector(LIS_ESOLVER esolver, LIS_INT mode, LIS_VECTOR x, LIS_INTEGER ierr)`

**Description**
Get the specified eigenvector from the eigensolver.

**Input**
- `esolver` The eigensolver
- `mode` The mode number of the eigenvector

**Output**
- `x` The eigenvector
- `ierr` The return code

### 6.5.20 lis_esolver_get_specific_residualnorm

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LIS_INT lis_esolver_get_specific_residualnorm(LIS_ESOLVER esolver, LIS_INT mode, LIS_REAL *residual)</td>
<td></td>
</tr>
</tbody>
</table>

Fortran subroutine `lis_esolver_get_specific_residualnorm(LIS_ESOLVER esolver, LIS_INT mode, LIS_REAL residual, LIS_INTEGER ierr)`

**Description**
Get the relative residual norm $\|\lambda x - (B^{-1})Ax\|_2/\|\lambda x\|_2$ of the specified eigenpair from the eigensolver.

**Input**
- `esolver` The eigensolver
- `mode` The mode number of the eigenpair

**Output**
- `residual` The residual norm
- `ierr` The return code
6.5.21 lis_esolver_get_specific_iter

C    LIS_INT lis_esolver_get_specific_iter(LIS_ESOLER esolver, LIS_INT mode, LIS_INT *iter)
Fortran subroutine lis_esolver_get_specific_iter(LIS_ESOLER esolver, LIS_INT mode, LIS_INT iter, LIS_INTEGER ierr)

**Description**
Get the numbers of iterations of the specified eigenpair from the eigensolver.

**Input**
esolver  The eigensolver
mode     The mode number of the eigenpair

**Output**
iter     The numbers of iterations
ierr     The return code

6.5.22 lis_esolver_get_esolver

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

**Description**
Get the eigensolver number from the eigensolver.

**Input**
esolver  The eigensolver

**Output**
nesol    The eigensolver number
ierr     The return code

6.5.23 lis_esolver_get_esolvername

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

**Description**
Get the eigensolver name from the eigensolver number.

**Input**
nesol    The eigensolver number

**Output**
ename    The eigensolver name
ierr     The return code
6.6 Computing with Arrays

The following functions, which are not parallelized, are for local processing. Array data are stored in column-major order. Array indexing is zero-origin. See lis-$(VERSION)/test/test6.c and lis-$(VERSION)/test/test6f.F90.

6.6.1 lis_array_swap

C LIS_INT lis_array_swap(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR y[])
Fortran subroutine lis_array_swap(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR y(),
LIS_INTEGER ierr)

Description
Swap the values of the vector elements.

Input
\n\n  n The size of the vectors
  x, y The source arrays that store vectors x, y of size n

Output
  x, y The destination arrays
  ierr The return code

6.6.2 lis_array_copy

C LIS_INT lis_array_copy(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR y[])
Fortran subroutine lis_array_copy(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR y(),
LIS_INTEGER ierr)

Description
Copy the values of the vector elements.

Input
\n\n  n The size of the vectors
  x The source array that stores vector x

Output
  y The destination array
  ierr The return code
6.6.3  lis_array_axpy

C     LIS_INT lis_array_axpy(LIS_INT n, LIS_SCALAR alpha, LIS_SCALAR x[],
                       LIS_SCALAR y[])
Fortran subroutine lis_array_axpy(LIS_INTEGER n, LIS_SCALAR alpha, LIS_SCALAR x(),
                       LIS_SCALAR y(), LIS_INTEGER ierr)

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

Input
\begin{itemize}
  \item $n$ The size of the vectors
  \item $\alpha$ The scalar value
  \item $x, y$ The arrays that store vectors $x, y$
\end{itemize}

Output
\begin{itemize}
  \item $y$ $\alpha x + y$ (vector $y$ is overwritten)
  \item $ierr$ The return code
\end{itemize}

6.6.4  lis_array_xpay

C     LIS_INT lis_array_xpay(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR alpha,
                       LIS_SCALAR y[])
Fortran subroutine lis_array_xpay(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR alpha,
                       LIS_SCALAR y(), LIS_INTEGER ierr)

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

Input
\begin{itemize}
  \item $n$ The size of the vectors
  \item $\alpha$ The scalar value
  \item $x, y$ The arrays that store vectors $x, y$
\end{itemize}

Output
\begin{itemize}
  \item $y$ $x + \alpha y$ (vector $y$ is overwritten)
  \item $ierr$ The return code
\end{itemize}
6.6.5 lis_array_axpyz

```c
C   LIS_INT lis_array_axpyz(LIS_INT n, LIS_SCALAR alpha, LIS_SCALAR x[],
   LIS_SCALAR y[], LIS_SCALAR z[])
Fortran subroutine lis_array_axpyz(LIS_INTEGER n, LIS_SCALAR alpha, LIS_SCALAR x(),
   LIS_SCALAR y(), LIS_SCALAR z(), LIS_INTEGER ierr)
```

**Description**
Calculate the sum of the vectors \( z = \alpha x + y \).

**Input**
- \( n \)
  - The size of the vectors
- \( \alpha \)
  - The scalar value
- \( x, y \)
  - The arrays that store vectors \( x, y \)

**Output**
- \( z \)
  - \( \alpha x + y \)
- \( ierr \)
  - The return code

6.6.6 lis_array_scale

```c
C   LIS_INT lis_array_scale(LIS_INT n, LIS_SCALAR alpha, LIS_SCALAR x[])
Fortran subroutine lis_array_scale(LIS_INTEGER n, LIS_SCALAR alpha, LIS_SCALAR x(),
   LIS_INTEGER ierr)
```

**Description**
Multiply vector \( x \) by scalar \( \alpha \).

**Input**
- \( n \)
  - The size of the vector
- \( \alpha \)
  - The scalar value
- \( x \)
  - The array that stores vector \( x \)

**Output**
- \( x \)
  - \( \alpha x \) (vector \( x \) is overwritten)
- \( ierr \)
  - The return code
6.6.7 lis_array_pmul

C    LIS_INT lis_array_pmul(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR y[],
           LIS_SCALAR z[])
Fortran subroutine lis_array_pmul(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR y(),
           LIS_SCALAR z(), LIS_INTEGER ierr)

Description
Multiply each element of vector \( x \) by the corresponding element of \( y \).

Input
\n  \n  \textbf{n} \quad \text{The size of the vectors}
  \textbf{x, y} \quad \text{The arrays that store vectors } x, y

Output
\n  \textbf{z} \quad \text{The array that stores the multiplied elements of } x
  \textbf{ierr} \quad \text{The return code}

6.6.8 lis_array_pdiv

C    LIS_INT lis_array_pdiv(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR y[],
           LIS_SCALAR z[])
Fortran subroutine lis_array_pdiv(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR y(),
           LIS_SCALAR z(), LIS_INTEGER ierr)

Description
Divide each element of vector \( x \) by the corresponding element of \( y \).

Input
\n  \n  \textbf{n} \quad \text{The size of the vectors}
  \textbf{x, y} \quad \text{The arrays that store vectors } x, y

Output
\n  \textbf{z} \quad \text{The array that stores the divided elements of } x
  \textbf{ierr} \quad \text{The return code}
6.6.9  lis_array_set_all

C      LIS_INT lis_array_set_all(LIS_INT n, LIS_SCALAR value, LIS_SCALAR x[])
Fortran subroutine lis_array_set_all(LIS_INTEGER n, LIS_SCALAR value,
                                   LIS_SCALAR x(), LIS_INTEGER ierr)

Description
Assign the scalar value to the elements of vector x.

Input

n            The size of the vector
value        The scalar value to be assigned
x            The array that stores vector x

Output

x            The array with the value assigned to the elements
ierr         The return code

6.6.10 lis_array_abs

C      LIS_INT lis_array_abs(LIS_INT n, LIS_SCALAR x[])
Fortran subroutine lis_array_abs(LIS_INTEGER n, LIS_SCALAR x(), LIS_INTEGER ierr)

Description
Get the absolute values of the elements of vector x.

Input

n            The size of the vector
x            The array that stores vector x

Output

x            The array that stores the absolute values
ierr         The return code
6.6.11 lis_array_reciprocal

C    LIS_INT lis_array_reciprocal(LIS_INT n, LIS_SCALAR x[])
Fortran subroutine lis_array_reciprocal(LIS_INTEGER n, LIS_SCALAR x(),
   LIS_INTEGER ierr)

Description
   Get the reciprocal values of the elements of vector \( x \).

Input
   \( n \)  The size of the vector
   \( x \)  The array that stores vector \( x \)

Output
   \( x \)  The array that stores the reciprocal values
   \( ierr \)  The return code

6.6.12 lis_array_conjugate

C    LIS_INT lis_array_conjugate(LIS_INT n, LIS_SCALAR x[])
Fortran subroutine lis_array_conjugate(LIS_INTEGER n, LIS_SCALAR x(),
   LIS_INTEGER ierr)

Description
   Get the conjugate values of the elements of vector \( x \).

Input
   \( n \)  The size of the vector
   \( x \)  The array that stores vector \( x \)

Output
   \( x \)  The array that stores the conjugate values
   \( ierr \)  The return code
6.6.13 lis_array_shift

C     LIS_INT lis_array_shift(LIS_INT n, LIS_SCALAR sigma, LIS_SCALAR x[])
Fortran subroutine lis_array_shift(LIS_INTEGER n, LIS_SCALAR sigma, LIS_SCALAR x(), LIS_INTEGER ierr)

Description
Shift vector x.

Input
n    The size of the vector
sigma The amount of the shift
x     The array that stores vector x

Output
x     The array that stores the shifted elements $x_i - \sigma$
ierr  The return code

6.6.14 lis_array_dot

C     LIS_INT lis_array_dot(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR y[], LIS_SCALAR *value)
Fortran subroutine lis_array_dot(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR y(), LIS_SCALAR value, LIS_INTEGER ierr)

Description
Calculate the Hermitian inner product $x^H y$ of vectors $x, y$.

Input
n    The size of the vectors
x, y  The arrays that store vectors $x, y$

Output
value The Hermitian inner product
ierr  The return code

144
6.6.15 lis_array_nhdot

C     LIS_INT lis_array_nhdot(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR y[],
               LIS_SCALAR *value)
Fortran subroutine lis_array_nhdot(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR y(),
               LIS_SCALAR value, LIS_INTEGER ierr)

Description
Calculate the non-Hermitian inner product $x^T y$ of vectors $x, y$.

Input
  n            The size of the vectors
  x, y         The arrays that store vectors $x, y$

Output
  value        The non-Hermitian inner product
  ierr         The return code

6.6.16 lis_array_nrm1

C     LIS_INT lis_array_nrm1(LIS_INT n, LIS_SCALAR x[], LIS_REAL *value)
Fortran subroutine lis_array_nrm1(LIS_INTEGER n, LIS_SCALAR x(), LIS_REAL value,
               LIS_INTEGER ierr)

Description
Calculate the 1-norm of vector $x$.

Input
  n            The size of the vector
  x            The array that stores vector $x$

Output
  value        The 1-norm of the vector
  ierr         The return code
6.6.17 lis_array_nrm2

```c
LIS_INT lis_array_nrm2(LIS_INT n, LIS_SCALAR x[], LIS_REAL *value)
```

**Fortran subroutine**
```
lis_array_nrm2(LIS_INTEGER n, LIS_SCALAR x(), LIS_REAL value,
       LIS_INTEGER ierr)
```

**Description**
Calculate the 2-norm of vector $x$.

**Input**
- $n$ The size of the vector
- $x$ The array that stores vector $x$

**Output**
- $value$ The 2-norm of the vector
- $ierr$ The return code

6.6.18 lis_array_nrmi

```c
LIS_INT lis_array_nrmi(LIS_INT n, LIS_SCALAR x[], LIS_REAL *value)
```

**Fortran subroutine**
```
lis_array_nrmi(LIS_INTEGER n, LIS_SCALAR x(), LIS_REAL value,
       LIS_INTEGER ierr)
```

**Description**
Calculate the infinity norm of vector $x$.

**Input**
- $n$ The size of the vector
- $x$ The array that stores vector $x$

**Output**
- $value$ The infinity norm of the vector
- $ierr$ The return code
6.6.19  lis_array_sum

C     LIS_INT lis_array_sum(LIS_INT n, LIS_SCALAR x[], LIS_SCALAR *value)
Fortran subroutine lis_array_sum(LIS_INTEGER n, LIS_SCALAR x(), LIS_SCALAR value,
                             LIS_INTEGER ierr)

Description
    Calculate the sum of the elements of vector \( x \).

Input
    n          The size of the vector
    x          The array that stores vector \( x \)

Output
    value      The sum of the vector elements
    ierr       The return code
6.6.20 lis_array_matvec

Description
Calculate the matrix-vector product \( Ax \).

Input
- \( n \) The size of the matrix and vectors
- \( a \) The array that stores matrix \( A \) of size \( n \times n \)
- \( x \) The array that stores vector \( x \) of size \( n \)
- \( y \) The array that stores vector \( y \) of size \( n \)
- \( op \) LIS_INS_VALUE : \( y = Ax \)
  LIS_SUB_VALUE : \( y = y - Ax \)

Output
- \( y \) The return code

6.6.21 lis_array_matvech

Description
Calculate the matrix-vector product \( A^H x \).

Input
- \( n \) The size of the matrix and vectors
- \( a \) The array that stores matrix \( A \) of size \( n \times n \)
- \( x \) The array that stores vector \( x \) of size \( n \)
- \( y \) The array that stores vector \( y \) of size \( n \)
- \( op \) LIS_INS_VALUE : \( y = A^H x \)
  LIS_SUB_VALUE : \( y = y - A^H x \)

Output
- \( y \) The return code
6.6.22 lis_array_matvec_ns

C  LIS_INTEGER lis_array_matvec_ns(LIS_INT m, LIS_INT n, LIS_SCALAR a[],
   LIS_INT lda, LIS_SCALAR x[], LIS_SCALAR y[], LIS_INT op)
Fortran subroutine lis_array_matvec_ns(LIS_INTEGER m, LIS_INTEGER n, LIS_SCALAR a()
   LIS_INTEGER lda, LIS_SCALAR x(), LIS_SCALAR y(), LIS_INTEGER op,
   LIS_INTEGER ierr)

Description
Calculate the matrix-vector product $Ax$, where matrix $A$ is not square.

Input

m, n    The sizes of the matrix and vectors
a       The array that stores matrix $A$ of size $m \times n$
lda     The size of the leading dimension of array $A$
x       The array that stores vector $x$ of size $n$
y       The array that stores vector $y$ of size $m$
op
   LIS_INS_VALUE : $y = Ax$
   LIS_SUB_VALUE : $y = y - Ax$

Output

y       $y$
ierr    The return code
6.6.23  lis_array_matmat

C  LIS_INT lis_array_matmat(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR b[],
   LIS_SCALAR c[], LIS_INT op)
Fortran subroutine lis_array_matmat(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR b(),
   LIS_SCALAR c(), LIS_INTEGER op, LIS_INTEGER ierr)

Description
Calculate the matrix-matrix product $AB$.

Input
- $n$  The size of the matrices
- $a$  The array that stores matrix $A$ of size $n \times n$
- $b$  The array that stores matrix $B$ of size $n \times n$
- $c$  The array that stores matrix $C$ of size $n \times n$
- $op$  LIS_INS_VALUE : $C = AB$
- LIS_SUB_VALUE : $C = C - AB$

Output
- $c$  $C$
- $ierr$  The return code
### 6.6.24 lis_array_matmat_ns

```c
#include <lis.h>

LIS_INT lis_array_matmat_ns(LIS_INT l, LIS_INT m, LIS_INT n,
                          LIS_SCALAR a[], LIS_INT lda, LIS_SCALAR b[], LIS_INT ldb, LIS_SCALAR c[],
                          LIS_INT ldc, LIS_INT op)
```

| Fortran subroutine lis_array_matmat_ns(LIS_INTEGER l, LIS_INTEGER m, LIS_INTEGER n,
| LIS_SCALAR a(), LISINTEGER lda, LIS_SCALAR b(), LISINTEGER ldb,
| LIS_SCALAR c(), LISINTEGER ldc, LISINTEGER op, LISINTEGER ierr) |

#### Description

Calculate the matrix-matrix product $AB$, where matrices $A, B$ are not square.

#### Input

- $m, n$  
  The sizes of the matrices
- $a$  
  The array that stores matrix $A$ of size $l \times m$
- $lda$  
  The size of the leading dimension of array $A$
- $b$  
  The array that stores matrix $B$ of size $m \times n$
- $ldb$  
  The size of the leading dimension of array $B$
- $c$  
  The array that stores matrix $C$ of size $l \times n$
- $ldc$  
  The size of the leading dimension of array $C$
- $op$  
  LIS_INS_VALUE: $C = AB$
  LIS_SUB_VALUE: $C = C - AB$

#### Output

- $c$  
  $C$
- $ierr$  
  The return code
6.6.25 lis_array_ge

C    LIS_INT lis_array_ge(LIS_INT n, LIS_SCALAR a[])
Fortran subroutine lis_array_ge(LIS_INTEGER n, LIS_SCALAR a(), LIS_INTEGER ierr)

Description
  Calculate the inverse of matrix $A$ with the Gaussian elimination.

Input
  
  n           The size of the matrix
  a           The array that stores matrix $A$ of size $n \times n$

Output
  
  a           The inverse $A^{-1}$
  ierr        The return code

6.6.26 lis_array_solve

C    LIS_INT lis_array_solve(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR b[],
                             LIS_SCALAR x[], LIS_SCALAR w[])
Fortran subroutine lis_array_solve(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR b(),
                                 LIS_SCALAR x(), LIS_SCALAR w(), LIS_INTEGER ierr)

Description
  Solve the linear equation $Ax = b$ with the direct method.

Input
  
  n           The size of the matrix
  a           The array that stores coefficient matrix $A$ of size $n \times n$
  b           The array that stores right-hand side vector $b$ of size $n$
  w           The work array of size $n \times n$

Output
  
  x           The array that stores solution $x$
  ierr        The return code

152
6.6.27 lis_array_cgs

C     LIS_INT lis_array_cgs(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR q[],
             LIS_SCALAR r[])
Fortran subroutine lis_array_cgs(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR q(),
             LIS_SCALAR r(), LIS_INTEGER ierr)

Description
Calculate the QR factorization $QR = A$ with the classical Gram-Schmidt process.

Input
n      The size of the matrices
a      The array that stores matrix $A$ of size $n \times n$

Output
q      The array that stores orthogonal matrix $Q$ of size $n \times n$
r      The array that stores upper-triangular matrix $R$ of size $n \times n$
ierr   The return code

6.6.28 lis_array_mgs

C     LIS_INT lis_array_mgs(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR q[],
             LIS_SCALAR r[])
Fortran subroutine lis_array_mgs(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR q(),
             LIS_SCALAR r(), LIS_INTEGER ierr)

Description
Calculate the QR factorization $QR = A$ with the modified Gram-Schmidt process.

Input
n      The size of the matrices
a      The array that stores matrix $A$ of size $n \times n$

Output
q      The array that stores orthogonal matrix $Q$ of size $n \times n$
r      The array that stores upper-triangular matrix $R$ of size $n \times n$
ierr   The return code
6.6.29  lis_array_qr

C        LIS_INT lis_array_qr(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR q[],
                        LIS_SCALAR r[], LIS_INT *qriter, LIS_REAL *qrerr)
Fortran subroutine lis_array_qr(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR q(),
                        LIS_SCALAR r(), LIS_INTEGER qriter, LIS_REAL qrerr, LIS_INTEGER ierr)

Description
Calculate the eigenvalues of matrix A with the QR algorithm.

Input
n            The size of the matrices
a            The array that stores symmetric matrix A of size n x n
q            The work array of size n x n
r            The work array of size n x n

Output
a            The array that stores the block upper-triangular matrix with eigenvalues in the block diagonal elements after similarity transformation
qriter       The number of iterations of the QR algorithm
qrerr        The 2-norm of the first subdiagonal element A(2,1) after similarity transformation
ierr         The return code
6.7 Operating External Files

6.7.1 lis_input

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

Fortran subroutine

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

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

**Input**
- `filename` The source file

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

**Note**
The following file formats are supported:

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

6.7.2 lis_input_vector

```c
LIS_INT lis_input_vector(LIS_VECTOR v, char *filename)
```

Fortran subroutine

```fortran
subroutine lis_input_vector(LIS_VECTOR v, character filename, 
                            LIS_INTEGER ierr)
```

**Description**
Read the vector data from the external file.

**Input**
- `filename` The source file

**Output**
- `v` The vector
- `ierr` The return code

**Note**
The following file formats are supported:

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

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

Description
Read the matrix data from the external file.

Input
- filename The source file

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

Note
The following file formats are supported:
- The Matrix Market format
- The Harwell-Boeing format

6.7.4 lis_output

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

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

Input
- A The matrix
- b The right-hand side vector
- x The solution
- format The file format
  - LIS_FMT_MM The Matrix Market format
- filename The destination file

Output
- ierr The return code

Note
In the C version, NULL can be input if the vector should not be written to the external file.
6.7.5 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

6.7.6 lis_output_matrix

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

Description
Write the matrix data into the external file.

Input
A              The matrix
format          The file format
                  LIS_FMT.MM            The Matrix Market format
filename        The destination file

Output
ierr            The return code
6.8 Other Functions

6.8.1 lis_initialize

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

Description
Initialize the execution environment.

Input
argc       The number of command line arguments
argv       The command line argument

Output
ierr       The return code

6.8.2 lis_finalize

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

Description
Finalize the execution environment.

Input
None

Output
ierr       The return code
6.8.3  

**lis_wtime**

C     double lis_wtime()
Fortran real*8 lis_wtime()

**Description**

Measure the elapsed time.

**Input**

None

**Output**

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

**Note**

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

6.8.4  

**CHKERR**

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

**Description**

Check the value of the return code.

**Input**

ierr  The return code

**Output**

None

**Note**

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

C   LIS_INT lis_printf(LIS_Comm comm, const char *mess, ...)

Description
Print message on processing element 0.

Input
comm    The MPI communicator

Output
mess    The message

Note
The string ‘%D’ is replaced with ‘%lld’ if LIS_INT is long long int, and is replaced with ‘%d’ if it is int.
For the serial and multithreaded environments, the value of comm is ignored.
References


A File Formats

This section describes the file formats available for the library. Note that both the upper and lower triangular entries need to be stored irrespective of whether the matrix is symmetric or not in the Harwell-Boeing Format.

A.1 Extended Matrix Market Format

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

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

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

$$
A = \begin{pmatrix}
2 & 1 \\
1 & 2 & 1 \\
1 & 2 & 1 & 2
\end{pmatrix}, \quad b = \begin{pmatrix}
0 \\
1 \\
2 \\
3
\end{pmatrix}
$$

(A.1)
A.2 Harwell-Boeing Format

The Harwell-Boeing format stores the matrix in the CSC format. Assume that the array value stores the values of the nonzero elements of matrix A, the array index stores the row indices of the nonzero elements and the array ptr stores pointers to the top of each column in the arrays value and index. The format is as follows:

Line 1 (A72,A8)
  1 - 72 Title
  73 - 80 Key
Line 2 (5I14)
  1 - 14 Total number of lines excluding header
  15 - 28 Number of lines for ptr
  29 - 42 Number of lines for index
  43 - 56 Number of lines for value
  57 - 70 Number of lines for right-hand side vectors
Line 3 (A3,11X,4I14)
  1 - 3 Matrix type
     Col.1:
       R Real matrix
       C Complex matrix
       P Pattern only (Not supported)
     Col.2:
       S Symmetric (Not supported)
       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,4I14) Only presents if there are right-hand side vectors
  1 right-hand side vector type
     1 F for full storage
     2 M for same format as matrix (Not supported)
     3 G if a starting vector is supplied
     4 X if an exact solution is supplied
  4 - 14 Blank space
  15 - 28 Number of right-hand side vectors
  29 - 42 Number of nonzero elements

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

1--------10--------20--------30--------40--------50--------60--------70--------80
Harwell-Boeing format sample Lis
  8 1 1 4 2
RUA
(11i7)    (13i6)    (3e26.18)    (3e26.18)
A.3 Extended Matrix Market Format for Vectors

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

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

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

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

A.4 PLAIN Format for Vectors

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

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

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

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

168