Lis ユーザガイド

バージョン 1.6.24

The Scalable Software Infrastructure Project

http://www.ssisc.org/

表紙: 尾形光琳. 燕子花図. 1705 年頃. 根津美術館.

目 次

バージョン 1.0 からの変更点

1. double-double 型 4 倍精度演算に対応.
2. Fortran コンパイラに対応.
3. Autotools に対応.
4. (a) ソルバの構造を変更.
   (b) 関数 `lis_matrix_create()`, `lis_vector_create()` の引数を変更.
   (c) コマンドラインオプションの記法を変更.

バージョン 1.1 からの変更点

1. 固有値計算に対応.
2. 64 ビット整数型に対応.
3. (a) 関数 `lis_output_residual_history()`, `lis_get_residual_history()` の名称をそれぞれ `lis_solver_output_rhistory()`, `lis_solver_get_rhistory()` に変更.
   (b) Fortran インタフェース `lis_vector_set_value()`, `lis_vector_get_value()` の配列起点を 1 に変更.
   (c) Fortran インタフェース `lis_vector_set_size()` の配列起点を 1 に変更.
   (d) 演算精度に関するオプションの名称を`-precision` から`-f` に変更.
4. 関数 `lis_solve_kernel()` の仕様を `lis_solve_execute()` で計算された残差を返すよう変更.
5. 整数型の仕様を変更.
   (a) C プログラムにおける整数型を `LIS_INT` に変更. `LIS_INT` の既定値は `int`. プリプロセッサマクロ `_LONGLONG` が定義された場合には, `long long int` に置き換えられる.
   (b) Fortran プログラムにおける整数型を `LIS_INTEGER` に変更. `LIS_INTEGER` の既定値は `integer`. プリプロセッサマクロ `LONGLONG` が定義された場合には, `integer*8` に置き換えられる.
6. 行列格納形式 CRS (Compressed Row Storage), CCS (Compressed Column Storage) の名称をそれぞれ CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) に変更.
7. 関数 `lis_get_solvername()`, `lis_get_preconname()`, `lis_get_esolvername()` の名称をそれぞれ `lis_solver_get_solvername()`, `lis_solver_get_preconname()`, `lis_esolver_get_esolvername()` に変更.

バージョン 1.2 からの変更点

1. nmake に対応.
2. ファイル lis_config_win32.h の名称を lis_config_win.h に変更.
3. 行列格納形式 JDS (Jagged Diagonal Storage) の名称を JAD (Jagged Diagonal) に変更.
4. 関数 lis_fscan_double(), lis_bswap_double() の名称をそれぞれ lis_fscan_scalar(), lis_bswap_scalar() に変更.

バージョン 1.3 からの変更点

1. long double 型 4 倍精度演算に対応.
2. Fortran でのポインタ操作に対応.
3. 構造体 LIS_SOLVER, LIS_ESOLVER のメンバ residual の名称を rhistory に変更.
4. 構造体 LIS_SOLVER, LIS_ESOLVER のメンバ iters, iters2 の名称をそれぞれ iter, iter2 に変更.
5. 関数 lis_solver_get_iters(), lis_solver_get_itersex(), lis_esolver_get_iters(), lis_esolver_get_itersex() の名称をそれぞれ lis_solver_get_iter(), lis_solver_get_iterex(), lis_esolver_get_iter(), lis_esolver_get_iterex() に変更.
6. 構造体 LIS_SOLVER, LIS_ESOLVER のメンバ*times の名称をそれぞれ*time に変更.
7. 構造体 LIS_VECTOR にメンバ intvalue を追加.
8. 関数 lis_output_vector*(), lis_output_mm_vec() の仕様を整数値を格納できるよう変更.
9. 関数 lis_matrix_scaling*() の名称をそれぞれ lis_matrix_scale*() に変更.
10. 関数 lis_array_dot2(), lis_array_invGauss() の名称をそれぞれ lis_array_dot(), lis_array_ge() に変更.

バージョン 1.4 からの変更点

1. 配列操作に対応.
2. 前処理とソルバの分離に対応.
3. 関数 lis_array_qr() の仕様を QR 法の反復回数及び誤差を返すよう変更.
4. 関数 lis_array_matvec2(), lis_array_matmat2() の名称をそれぞれ lis_array_matvec_ns(), lis_array_matmat_ns() に変更.
5. プリプロセッサマクロ _LONGLONG, LONGLONG の名称をそれぞれ _LONG__LONG, LONG__LONG に変更.

バージョン 1.5 からの変更点

1. 複素数に対応.

1 はじめに

Lis (Library of Iterative Solvers for linear systems, 発音は [lis]) は, 偏微分方程式の数値計算に現れる疎行列を係数とする線型方程式

$$Ax = b$$

及び標準固有値問題

$$Ax = \lambda x$$

を解くための並列反復法ソフトウェアライブラリである [1]. 対応する線型方程式解法, 固有値解法の一覧を表 1-2, 前処理を表 3 に示す. また行列格納形式の一覧を表 4 に示す.

表 1: 線型方程式解法

CG[2, 3]	CR[2]
BiCG[4]	BiCR[5]
CGS[6]	CRS[7]
BiCGSTAB[8]	BiCRSTAB[7]
GPBiCG[9]	GPBiCR[7]
BiCGSafe[10]	BiCRSafe[11]
BiCGSTAB(l)[12]	TFQMR[13]
Jacobi[14]	Orthomin(m)[15]
Gauss-Seidel[16, 17]	GMRES(m)[18]
SOR[19, 20]	FGMRES(m)[21]
IDR(s)[22]	MINRES[23]
COCG[24]	COCR[25]

表 2: 固有値解法

Power[26]
Inverse[27]
Approximate Inverse[1]
Rayleigh Quotient[28]
CG[29]
CR[30]
Jacobi-Davidson[31]
Subspace[32]
Lanczos[33]
Arnoldi[34]

表 3: 前処理

Jacobi[35]
SSOR[35]
ILU(k)[36, 37]
ILUT[38, 39]
Crout ILU[39, 40]
I+S[41]
SA-AMG[42]
Hybrid[43]
SAINV[44]
Additive Schwarz[45, 46]
ユーザ定義

表 4: 行列格納形式

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

2 導入

本節では, 導入, 検証の手順について述べる.

2.1 システム要件

Lis の導入には C コンパイラが必要である. また, Fortran インタフェースを使用する場合は Fortran コンパイラ, AMG 前処理ルーチンを使用する場合は Fortran 90 コンパイラが必要である. 並列計算環境では, OpenMP ライブラリ [88] または MPI-1 ライブラリ [82] を使用する [47, 48]. データの入出力には, Harwell-Boeing 形式 [74], Matrix Market 形式 [78] が利用可能である. 表 5 に主な動作確認環境を示す (表 7 も参照のこと).

表 5: 主な動作確認環境

C コンパイラ (必須)	OS
Intel C/C++ Compiler 7.0, 8.0, 9.1, 10.1, 11.1, 12.1, 14.0, 16.0	Linux Windows
IBM XL C/C++ V7.0, 9.0	AIX Linux
Sun WorkShop 6, Sun ONE Studio 7, Sun Studio 11, 12	Solaris
PGI C++ 6.0, 7.1, 10.5	Linux
gcc 3.3, 4.4, 5.3	Linux Mac OS X Windows
Microsoft Visual C++ 2008, 2010, 2012, 2013, 2015	Windows
Fortran コンパイラ (オプション)	**OS**
Intel Fortran Compiler 8.1, 9.1, 10.1, 11.1, 12.1, 14.0, 16.0	Linux Windows
IBM XL Fortran V9.1, 11.1	AIX Linux
Sun WorkShop 6, Sun ONE Studio 7, Sun Studio 11, 12	Solaris
PGI Fortran 6.0, 7.1, 10.5	Linux
g77 3.3 gfortran 4.4, 5.3	Linux Mac OS X Windows

2.2 UNIX 及び互換システムへの導入

2.2.1 アーカイブの展開

次のコマンドを入力し, アーカイブを展開する. `($VERSION)` はバージョンを示す.

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

これにより, ディレクトリ `lis-($VERSION)` に図 1 に示すサブディレクトリが作成される.

```
lis-($VERSION)
 + config
 |   設定ファイル
 + doc
 |   説明書
 + graphics
 |   描画用サンプルファイル
 + include
 |   ヘッダファイル
 + src
 |   ソースファイル
 + test
 |   検証プログラム
 + win
     Windows システム用設定ファイル
```

図 1: `lis-($VERSION).tar.gz` のファイル構成

2.2.2 ソースツリーの設定

ディレクトリ `lis-($VERSION)` において次のコマンドを実行し, ソースツリーを設定する.

- 既定の設定を使用する場合 : `> ./configure`
- 導入先を指定する場合 : `> ./configure --prefix=<install-dir>`

表 6 に主な設定オプションを示す. また, 表 7 に `TARGET` として指定できる主な計算機環境を示す.

2.2.3 実行ファイルの生成

ディレクトリ `lis-($VERSION)` において次のコマンドを入力し, 実行ファイルを生成する.

```
> make
```

実行ファイルが正常に生成されたかどうかを確認するには, ディレクトリ `lis-($VERSION)` において次のコマンドを入力し, ディレクトリ `lis-($VERSION)/test` に生成された実行ファイルを用いて検証を行う.

```
> make check
```

このコマンドでは, Matrix Market 形式のファイル `test/testmat.mtx` から行列, ベクトルデータを読み込み, BiCG 法を用いて線型方程式 $Ax = b$ の解を求める. 以下に SGI Altix 3700 上での実行結果を示す. なおオプション`--enable-omp` と`--enable-mpi` は組み合わせて使用することができる.

表 6: 主な設定オプション (一覧は ./configure --help を参照)

--enable-omp	OpenMP ライブラリを使用
--enable-mpi	MPI ライブラリを使用
--enable-fortran	FORTRAN 77 互換インタフェースを使用
--enable-f90	Fortran 90 互換インタフェースを使用
--enable-saamg	SA-AMG 前処理を使用
--enable-quad	double-double 型 4 倍精度演算を使用
--enable-longdouble	long double 型 4 倍精度演算を使用
--enable-longlong	64 ビット整数型を使用
--enable-complex	スカラ型として複素数型を使用
--enable-debug	デバッグモードを使用
--enable-shared	動的リンクを使用
--enable-gprof	プロファイラを使用
--prefix=<install-dir>	導入先を指定
TARGET=<target>	計算機環境を指定
CC=<c_compiler>	C コンパイラを指定
CFLAGS=<c_flags>	C コンパイラオプションを指定
F77=<f77_compiler>	FORTRAN 77 コンパイラを指定
F77FLAGS=<f77_flags>	FORTRAN 77 コンパイラオプションを指定
FC=<f90_compiler>	Fortran 90 コンパイラを指定
FCFLAGS=<f90_flags>	Fortran 90 コンパイラオプションを指定
LDFLAGS=<ld_flags>	リンクオプションを指定

表 7: TARGET の例 (詳細は `lis-($VERSION)/configure.ac` を参照)

<target>	等価なオプション
cray_xt3_cross	./configure CC=cc FC=ftn CFLAGS="-O3 -B -fastsse -tp k8-64" FCFLAGS="-O3 -fastsse -tp k8-64 -Mpreprocess" FCLDFLAGS="-Mnomain" ac_cv_sizeof_void_p=8 cross_compiling=yes ax_f77_mangling="lower case, no underscore, extra underscore"
fujitsu_fx10_cross	./configure CC=fccpx FC=frtpx CFLAGS="-Kfast,ocl,preex" FCFLAGS="-Kfast,ocl,preex -Cpp -fs" FCLDFLAGS="-mlcmain=main" ac_cv_sizeof_void_p=8 cross_compiling=yes ax_f77_mangling="lower case, underscore, no extra underscore"
hitachi_sr16k	./configure CC=cc FC=f90 CFLAGS="-Os -noparallel" FCFLAGS="-Oss -noparallel" FCLDFLAGS="-lf90s" ac_cv_sizeof_void_p=8 ax_f77_mangling="lower case, underscore, no extra underscore"
ibm_bgl_cross	./configure CC=blrts_xlc FC=blrts_xlf90 CFLAGS="-O3 -qarch=440d -qtune=440 -qstrict" FCFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90" ac_cv_sizeof_void_p=4 cross_compiling=yes ax_f77_mangling="lower case, no underscore, no extra underscore"
nec_sx9_cross	./configure CC=sxmpic++ FC=sxmpif90 AR=sxar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes ax_f77_mangling="lower case, no underscore, extra underscore"

既定

```
matrix size = 100 x 100 (460 nonzero entries)

initial vector x      : 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         = 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
```

--enable-omp

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

initial vector x      : 0
precision             : double
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
```

--enable-mpi

```
number of processes = 2
matrix size = 100 x 100 (460 nonzero entries)

initial vector x      : 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 導入

ディレクトリ lis-($VERSION) において次のコマンドを入力し, 導入先のディレクトリにファイルを複製する.

```
> make install
```

これにより, ディレクトリ ($INSTALLDIR) に以下のファイルが複製される.

```
($INSTALLDIR)
 +bin
 |   +lsolve esolve 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 はライブラリを生成する際に, また lis.h は C, lisf.h は Fortran でライブラリを使用する際に必要なヘッダファイルである. liblis.a は生成されたライブラリである. ライブラリが正常に導入されたかどうかを確認するには, ディレクトリ lis-($VERSION) において次のコマンドを入力し, ディレクトリ examples/lis に生成された実行ファイルを用いて検証を行う.

```
> make installcheck
```

examples/lis 下の test1, etest5, test3b, spmvtest3b は, lsolve, esolve, hpcg_kernel, hpcg_spmvtest

の別名で ($INSTALLDIR)/bin に複製される. examples/lis/spmvtest*も, それぞれ ($INSTALLDIR)/bin に複製される.

($INSTALLDIR) に複製されたファイルを削除するには, 次のコマンドを入力する.

```
> make uninstall
```

lis-($VERSION) に生成されたライブラリ, 及び実行ファイルを削除するには, 次のコマンドを入力する.

```
> make clean
```

生成された設定ファイルを合わせて削除するには, 次のコマンドを入力する.

```
> make distclean
```

2.3 Windows システムへの導入

適当なツールを用いてアーカイブを展開した後, Microsoft Build Engine を使用する場合は, ディレクトリ lis-($VERSION)\win において次のコマンドを入力し, 設定ファイル Makefile を生成する (詳細は configure.bat --help を参照).

```
> configure.bat
```

Makefile の既定値は Makefile.in で定義される. 実行ファイルを生成するには, lis-($VERSION)\win において次のコマンドを入力する.

```
> nmake
```

実行ファイルが正常に生成されたかどうかを確認するには, 次のコマンドを入力し, 生成された実行ファイルを用いて検証を行う.

```
> nmake check
```

生成されたライブラリ及び検証プログラムの実行ファイルは, 以下のコマンドにより lis-($VERSION)\lib, lis-($VERSION)\bin にそれぞれ格納される.

```
> nmake install
```

lis-($VERSION)\lib, lis-($VERSION)\bin に複製されたファイルを削除するには, 次のコマンドを入力する.

```
> nmake uninstall
```

lis-($VERSION)\win に生成されたライブラリ, 及び実行ファイルを削除するには, 次のコマンドを入力する.

```
> nmake clean
```

生成された設定ファイルを合わせて削除するには, 次のコマンドを入力する.

```
> nmake distclean
```

UNIX 互換環境を使用する場合は前節を参照のこと.

2.4 検証

検証プログラムは lis-($VERSION)/test に格納される.

2.4.1 test1

ディレクトリ lis-($VERSION)/test において

```
> test1 matrix_filename rhs_setting solution_filename rhistory_filename [options]
```

と入力すると, `matrix_filename` から行列データを読み込み, 線型方程式 $Ax = b$ を `options` で指定された解法で解く．また, 解を拡張 Matrix Market 形式で `solution_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す (付録 A を参照). 入力可能な行列データ形式は Matrix Market 形式, 拡張 Matrix Market 形式である．`rhs_setting` には

0	行列データファイルに含まれる右辺ベクトルを用いる
1	$b = (1, \ldots, 1)^T$ を用いる
2	$b = A \times (1, \ldots, 1)^T$ を用いる
rhs_filename	右辺ベクトルのファイル名

のいずれかを指定できる．`rhs_filename` は PLAIN 形式, Matrix Market 形式に対応する．`test1f.F` は `test1.c` の Fortran 版である．

2.4.2 test2

ディレクトリ `lis-($VERSION)/test` において

```
> test2 m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 2 次元 Laplace 作用素を 5 点中心差分により離散化して得られる次数 mn の行列 A を係数とする線型方程式 $Ax = b$ を, `matrix_type` で指定された行列格納形式, `options` で指定された解法で解く．また, 解を拡張 Matrix Market 形式で `solution_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す．右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される．`m`, `n` は各次元の格子点数である．`test2f.F90` は `test2.c` の Fortran 90 版である．

2.4.3 test2b

ディレクトリ `lis-($VERSION)/test` において

```
> test2b m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 2 次元 Laplace 作用素を 9 点中心差分により離散化して得られる次数 mn の行列 A を係数とする線型方程式 $Ax = b$ を, `matrix_type` で指定された行列格納形式, `options` で指定された解法で解く．また, 解を拡張 Matrix Market 形式で `solution_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す．右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される．`m`, `n` は各次元の格子点数である．

2.4.4 test3

ディレクトリ `lis-($VERSION)/test` において

```
> test3 l m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 3 次元 Laplace 作用素を 7 点中心差分により離散化して得られる次数 lmn の行列 A を係数とする線型方程式 $Ax = b$ を, `matrix_type` で指定された行列格納形式, `options` で指定された解法で解く．また, 解を拡張 Matrix Market 形式で `solution_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す．右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される．`l`, `m`, `n` は各次元の格子点数である．

2.4.5 test3b

ディレクトリ lis-($VERSION)/test において

```
> test3b l m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 3 次元 Laplace 作用素を 27 点中心差分により離散化して得られる次数 lmn の行列 A を係数とする線型方程式 $Ax = b$ を, matrix_type で指定された行列格納形式, options で指定された解法で解く. また, 解を拡張 Matrix Market 形式で solution_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. 右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される. l, m, n は各次元の格子点数である.

2.4.6 test4

線型方程式 $Ax = b$ を指定された解法で解き, 解を標準出力に書き出す. 行列 A は次数 12 の 3 重対角行列

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

である. 右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される. test4f.F は test4.c の Fortran 版である.

2.4.7 test5

ディレクトリ lis-($VERSION)/test において

```
> test5 n gamma [options]
```

と入力すると, 線型方程式 $Ax = b$ を指定された解法で解く. 行列 A は次数 n の Toepliz 行列

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

である. 右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される.

2.4.8 test6

test6.c は test2.c の配列版である. ディレクトリ lis-($VERSION)/test において

```
> test6 m n
```

と入力すると, 2 次元 Laplace 作用素を 5 点中心差分により離散化して得られる次数 mn の行列 A を係数とする線型方程式 $Ax = b$ を直接法で解く. 右辺ベクトル b は解ベクトル x の値がすべて 1 となるよう設定される. m, n は各次元の格子点数である. test6f.F90 は test6.c の Fortran 90 版である.

2.4.9 etest1

ディレクトリ lis-($VERSION)/test において

```
> etest1 matrix_filename evector_filename rhistory_filename [options]
```

と入力すると, matrix_filename から行列データを読み込み, 固有値問題 $Ax = \lambda x$ を options で指定された解法で解いて, 指定された固有値を標準出力に書き出す. また, 対応する固有ベクトルを拡張 Matrix Market 形式で evector_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. 入力可能な行列データ形式は Matrix Market 形式である. etest1f.F は etest1.c の Fortran 版である.

2.4.10 etest2

ディレクトリ lis-($VERSION)/test において

```
> etest2 m n matrix_type evector_filename rhistory_filename [options]
```

と入力すると, 2 次元 Laplace 作用素を 5 点中心差分により離散化して得られる次数 mn の行列 A に関する固有値問題 $Ax = \lambda x$ を, matrix_type で指定された行列格納形式, options で指定された解法で解き, 指定された固有値を標準出力に書き出す. また, 対応する固有ベクトルを evector_filename に, 残差履歴を rhistory_filename に書き出す. m, n は各次元の格子点数である.

2.4.11 etest3

ディレクトリ lis-($VERSION)/test において

```
> etest3 l m n matrix_type evector_filename rhistory_filename [options]
```

と入力すると, 3 次元 Laplace 作用素を 7 点中心差分により離散化して得られる次数 lmn の行列 A に関する固有値問題 $Ax = \lambda x$ を, matrix_type で指定された行列格納形式, options で指定された解法で解き, 指定された固有値を標準出力に書き出す. また, 対応する固有ベクトルを拡張 Matrix Market 形式で evector_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. l, m, n は各次元の格子点数である.

2.4.12 etest4

ディレクトリ lis-($VERSION)/test において

```
> etest4 n [options]
```

と入力すると, 固有値問題 $Ax = \lambda x$ を指定された解法で解き, 指定された固有値を標準出力に書き出す. 行列 A は次数 n の 3 重対角行列

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

である. etest4f.F は etest4.c の Fortran 版である.

2.4.13 etest5

ディレクトリ lis-($VERSION)/test において

```
> etest5 matrix_filename evalues_filename evectors_filename residuals_filename
iters_filename [options]
```

と入力すると, matrix_filename から行列データを読み込み, 固有値問題 $Ax = \lambda x$ を options で指定された解法で解いて, 指定された固有値を標準出力に書き出す. また, 指定された個数の固有値を evalues_filename に, 対応する固有ベクトル, 残差ノルム及び反復回数を evectors_filename, residuals_filename 及び iters_filename に拡張 Matrix Market 形式で書き出す. 入力可能な行列データ形式は Matrix Market 形式である.

2.4.14 etest6

ディレクトリ lis-($VERSION)/test において

```
> etest6 l m n matrix_type evalues_filename evectors_filename residuals_filename
iters_filename [options]
```

と入力すると, 3 次元 Laplace 作用素を 7 点中心差分により離散化して得られる次数 lmn の行列 A に関する固有値問題 $Ax = \lambda x$ を, matrix_type で指定された行列格納形式, options で指定された解法で解き, 指定された固有値を標準出力に書き出す. また, 指定された個数の固有値を evalues_filename に, 対応する固有ベクトル, 残差ノルム及び反復回数を evectors_filename, residuals_filename 及び iters_filename に拡張 Matrix Market 形式で書き出す. l, m, n は各次元の格子点数である.

2.4.15 etest7

etest7.c は etest2.c の配列版である. ディレクトリ lis-($VERSION)/test において

```
> etest7 m n
```

と入力すると, 2 次元 Laplace 作用素を 5 点中心差分により離散化して得られる次数 mn の行列 A に関する固有値問題 $Ax = \lambda x$ を QR 法で解く. m, n は各次元の格子点数である.

2.4.16 spmvtest1

ディレクトリ lis-($VERSION)/test において

```
> spmvtest1 n iter [matrix_type]
```

と入力すると, 1 次元 Laplace 作用素を 3 点中心差分により離散化して得られる次数 n の行列

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

とベクトル $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する.

2.4.17 spmvtest2

ディレクトリ `lis-($VERSION)/test` において

```
> spmvtest2 m n iter [matrix_type]
```

と入力すると, 2 次元 Laplace 作用素を 5 点中心差分により離散化して得られる次数 mn の 5 重対角行列とベクトル $(1, \ldots, 1)^T$ との積を `iter` で指定された回数実行し, FLOPS 値を算出する. 必要なら `matrix_type` により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. `m`, `n` は各次元の格子点数である.

2.4.18 spmvtest2b

ディレクトリ `lis-($VERSION)/test` において

```
> spmvtest2b m n iter [matrix_type]
```

と入力すると, 2 次元 Laplace 作用素を 9 点中心差分により離散化して得られる次数 mn の 9 重対角行列とベクトル $(1, \ldots, 1)^T$ との積を `iter` で指定された回数実行し, FLOPS 値を算出する. 必要なら `matrix_type` により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. `m`, `n` は各次元の格子点数である.

2.4.19 spmvtest3

ディレクトリ `lis-($VERSION)/test` において

```
> spmvtest3 l m n iter [matrix_type]
```

と入力すると, 3 次元 Laplace 作用素を 7 点中心差分により離散化して得られる次数 lmn の 7 重対角行列とベクトル $(1, \ldots, 1)^T$ との積を `iter` で指定された回数実行し, FLOPS 値を算出する. 必要なら `matrix_type` により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. `l`, `m`, `n` は各次元の格子点数である.

2.4.20 spmvtest3b

ディレクトリ lis-($VERSION)/test において

```
> spmvtest3b l m n iter [matrix_type]
```

と入力すると, 3 次元 Laplace 作用素を 27 点中心差分により離散化して得られる次数 lmn の 27 重対角行列とベクトル $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. l, m, n は各次元の格子点数である.

2.4.21 spmvtest4

ディレクトリ lis-($VERSION)/test において

```
> spmvtest4 matrix_filename_list iter [block]
```

と入力すると, matrix_filename_list の示す行列データファイルリストから行列データを読み込み, 各行列とベクトル $(1, \ldots, 1)^T$ との積を実行可能な行列格納形式について iter で指定された回数実行し, FLOPS 値を算出する. 入力可能な行列データ形式は Matrix Market 形式である. 必要なら block により, BSR, BSC 形式のブロックサイズを指定する.

2.4.22 spmvtest5

ディレクトリ lis-($VERSION)/test において

```
> spmvtest5 matrix_filename matrix_type iter [block]
```

と入力すると, matrix_filename の示す行列データファイルから行列データを読み込み, 行列とベクトル $(1, \ldots, 1)^T$ との積を行列格納形式 matrix_type について iter で指定された回数実行し, FLOPS 値を算出する. 入力可能な行列データ形式は Matrix Market 形式である. 必要なら block により, BSR, BSC 形式のブロックサイズを指定する.

2.5 制限事項

現バージョンには以下の制限がある.

- 行列格納形式
  - VBR 形式はマルチプロセス環境では使用できない.
  - CSR 形式以外の格納形式は SA-AMG 前処理では使用できない.
  - マルチプロセス環境において必要な配列を直接定義する場合は, CSR 形式を使用しなければならない. 他の格納形式を使用する場合は, 関数 lis_matrix_convert() を用いて変換を行う.
- double-double 型 4 倍精度演算 (4 節を参照)
  - 線型方程式解法のうち, Jacobi, Gauss-Seidel, SOR, IDR(s) 法では使用できない.

- 固有値解法のうち, CG, CR, Jacobi-Davidson 法では使用できない.
  - Hybrid 前処理での内部反復解法のうち, Jacobi, Gauss-Seidel, SOR 法では使用できない.
  - I+S, SA-AMG 前処理では使用できない.

- long double 型 4 倍精度演算
  - Fortran インタフェースでは使用できない.
  - SA-AMG 前処理では使用できない.

- 前処理
  - Jacobi, SSOR 以外の前処理が選択され, かつ行列 A が CSR 形式でない場合, 前処理作成時に CSR 形式の行列 A が作成される.
  - 非対称線型方程式解法として BiCG 法が選択された場合, SA-AMG 前処理は使用できない.
  - SA-AMG 前処理はマルチスレッド計算には対応していない.
  - SAINV 前処理の前処理行列作成部分は逐次実行される.
  - ユーザ定義前処理は使用できない.

3　基本操作

本節では, ライブラリの使用方法について述べる. プログラムでは, 以下の処理を行う必要がある.

- 初期化処理
- 行列の作成
- ベクトルの作成
- ソルバ (解法の情報を格納する構造体) の作成
- 行列, ベクトルへの値の代入
- 解法の設定
- 求解
- 終了処理

また, プログラムの先頭には以下のコンパイラ指示文を記述しなければならない.

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

`lis.h`, `lisf.h` は, 導入時に `($INSTALLDIR)/include` 下に格納される.

3.1　初期化・終了処理

初期化, 終了処理は以下のように記述する. 初期化処理はプログラムの最初に, 終了処理は最後に実行しなければならない.

C

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

Fortran

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

初期化処理

初期化処理を行うには, 関数

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

- Fortran subroutine lis_initialize(LIS_INTEGER ierr)

を用いる．この関数は, MPI の初期化, コマンドライン引数の取得等の初期化処理を行う．

LIS_INT の既定値 int は, プリプロセッサマクロ _LONG__LONG が定義された場合には long long int に, また LIS_INTEGER の既定値 integer は, プリプロセッサマクロ LONG__LONG が定義された場合には integer*8 に置き換えられる．

終了処理

終了処理を行うには, 関数

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

を用いる．

3.2 ベクトルの操作

ベクトル v の次数を $global\_n$ とする．ベクトル v を $nprocs$ 個のプロセスで行ブロック分割する場合の各部分ベクトルの行数を $local\_n$ とする．$global\_n$ が $nprocs$ で割り切れる場合は $local\_n = global\_n\ /\ nprocs$ となる．例えば, ベクトル v を (3.1) 式のように 2 プロセスで行ブロック分割する場合, $global\_n$ と $local\_n$ はそれぞれ 4 と 2 となる．

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

(3.1) 式のベクトル v を作成する場合, 逐次, マルチスレッド環境ではベクトル v そのものを, マルチプロセス環境では各プロセスにプロセス数で行ブロック分割した部分ベクトルを作成する．

ベクトル v を作成するプログラムは以下のように記述する．ただし, マルチプロセス環境のプロセス数は 2 とする．

C (逐次・マルチスレッド環境)

```
 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 (マルチプロセス環境)

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

Fortran (逐次・マルチスレッド環境)

```
LIS_INTEGER i,n
LIS_VECTOR v
n = 4
call lis_vector_create(0,v,ierr)
call lis_vector_set_size(v,0,n,ierr)

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

Fortran (マルチプロセス環境)

```
LIS_INTEGER i,n,is,ie
LIS_VECTOR v
n = 4
call lis_vector_create(MPI_COMM_WORLD,v,ierr)
call lis_vector_set_size(v,0,n,ierr)
call lis_vector_get_range(v,is,ie,ierr)
do i=is,ie-1
    call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr);
enddo
```

ベクトルの作成

ベクトル v の作成には, 関数

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

を用いる. `comm` には MPI コミュニケータを指定する. 逐次, マルチスレッド環境では `comm` の値は無視される.

次数の設定

次数の設定には, 関数

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

を用いる. $local\_n$ か $global\_n$ のどちらか一方を与えなければならない.

逐次, マルチスレッド環境では, $local\_n$ は $global\_n$ に等しい. したがって, `lis_vector_set_size(v,n,0)` と `lis_vector_set_size(v,0,n)` は, いずれも次数 n のベクトルを作成する.

マルチプロセス環境においては, `lis_vector_set_size(v,n,0)` は各プロセス上に次数 n の部分ベクトルを作成する. 一方, `lis_vector_set_size(v,0,n)` は各プロセス p 上に次数 m_p の部分ベクトルを作成する. m_p はライブラリ側で決定される.

値の代入

ベクトル v の第 i 行に値を代入するには, 関数

- 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)`

を用いる. マルチプロセス環境では, 部分ベクトルの第 i 行ではなく, 全体ベクトルの第 i 行を指定する. `flag` には

`LIS_INS_VALUE` 挿入: $v[i] = value$

`LIS_ADD_VALUE` 加算代入: $v[i] = v[i] + value$

のどちらかを指定する.

ベクトルの複製

既存のベクトルと同じ情報を持つベクトルを作成するには, 関数

- 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)`

を用いる. 第 1 引数 `LIS_VECTOR vin` は `LIS_MATRIX` を指定することも可能である. この関数はベクトルの要素の値は複製しない. 値も複製する場合は, この関数の後に

- 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)`

を呼び出す.

ベクトルの破棄

不要になったベクトルをメモリから破棄するには,

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

を用いる.

3.3 行列の操作

行列 A の次数を $global\_n \times global\_n$ とする．行列 A を $nprocs$ 個のプロセスで行ブロック分割する場合の各ブロックの行数を $local\_n$ とする．$global\_n$ が $nprocs$ で割り切れる場合は $local\_n = global\_n \ / \ nprocs$ となる．例えば，行列 A を (3.2) 式のように 2 個のプロセスで行ブロック分割する場合，$global\_n$ と $local\_n$ はそれぞれ 4 と 2 となる．

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

目的の格納形式の行列を作成するには以下の 3 つの方法がある．

方法 1: ライブラリ関数を用いて目的の格納形式の配列を定義する場合

(3.2) 式の行列 A を CSR 形式で作成する場合，逐次，マルチスレッド環境では行列 A そのものを，マルチプロセス環境では各プロセスにプロセス数で行ブロック分割した部分行列を作成する．

行列 A を CSR 形式で作成するプログラムは以下のように記述する．ただし，マルチプロセス環境のプロセス数は 2 とする．

C (逐次・マルチスレッド環境)

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

C (マルチプロセス環境)

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

Fortran (逐次・マルチスレッド環境)

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

Fortran (マルチプロセス環境)

```
LIS_INTEGER i,n,gn,is,ie
LIS_MATRIX A
gn = 4
call lis_matrix_create(MPI_COMM_WORLD,A,ierr)
call lis_matrix_set_size(A,0,gn,ierr)
call lis_matrix_get_size(A,n,gn,ierr)
call lis_matrix_get_range(A,is,ie,ierr)
do i=is,ie-1
    if( i>1  ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0d0,A,ierr)
    if( i<gn ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0d0,A,ierr)
    call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
enddo
call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
call lis_matrix_assemble(A,ierr)
```

行列の作成

行列 A の作成には, 関数

- 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)`

を用いる. `comm` には MPI コミュニケータを指定する. 逐次, マルチスレッド環境では, `comm` の値は無視される.

次数の設定

次数の設定には, 関数

- 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)`

を用いる. *local_n* か *global_n* のどちらか一方を与えなければならない.

逐次, マルチスレッド環境では, *local_n* は *global_n* に等しい. したがって, `lis_matrix_set_size(A,n,0)` と `lis_matrix_set_size(A,0,n)` は, いずれも次数 $n \times n$ の行列を作成する.

マルチプロセス環境においては, lis_matrix_set_size(A,n,0) は各プロセス上に次数 $n \times N$ の部分行列を作成する. N は n の総和である.
一方, lis_matrix_set_size(A,0,n) は各プロセス p 上に次数 $m_p \times n$ の部分行列を作成する. m_p はライブラリ側で決定される.

値の代入

行列 A の第 i 行第 j 列に値を代入するには, 関数

- 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)

を用いる. マルチプロセス環境では, 全体行列の第 i 行第 j 列を指定する. flag には

LIS_INS_VALUE 挿入: $A[i,j] = value$

LIS_ADD_VALUE 加算代入: $A[i,j] = A[i,j] + value$

のどちらかを指定する.

行列格納形式の設定

行列の格納形式を設定するには, 関数

- 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)

を用いる. 行列作成時の A の matrix_type は LIS_MATRIX_CSR である. 以下に対応する格納形式を示す.

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

行列の組み立て

行列の要素と格納形式を設定した後, 関数

- C　　　`LIS_INT lis_matrix_assemble(LIS_MATRIX A)`

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

を呼び出す. `lis_matrix_assemble` は `lis_matrix_set_type` で指定された格納形式に組み立てられる.

行列の破棄

不要になった行列をメモリから破棄するには,

- C　　　`LIS_INT lis_matrix_destroy(LIS_MATRIX A)`

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

を用いる.

方法 2: 目的の格納形式の配列を直接定義する場合

(3.2) 式の行列 A を CSR 形式で作成する場合, 逐次, マルチスレッド環境では行列 A そのものを, マルチプロセス環境では各プロセスにプロセス数で行ブロック分割した部分行列を作成する.

行列 A を CSR 形式で作成するプログラムは以下のように記述する. ただし, マルチプロセス環境のプロセス数は 2 とする.

C (逐次・マルチスレッド環境)

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

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

C (マルチプロセス環境)

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

配列の関連付け

CSR 形式の配列をライブラリが扱えるよう行列 A に関連付けるには, 関数

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

を用いる. その他の格納形式については 5 節を参照せよ.

方法 3: 外部ファイルから行列, ベクトルデータを読み込む場合

外部ファイルから (3.2) 式の行列 A を CSR 形式で読み込む場合, プログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

```
LIS_MATRIX A;
lis_matrix_create(LIS_COMM_WORLD,&A);
lis_matrix_set_type(A,LIS_MATRIX_CSR);
lis_input_matrix(A,"matvec.mtx");
```

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
LIS_MATRIX A
call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
call lis_input_matrix(A,'matvec.mtx',ierr)
```

Matrix Market 形式による外部ファイル `matvec.mtx` の記述例を以下に示す.

```
%%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
```

外部ファイルから (3.2) 式の行列 A を CSR 形式で, また (3.1) 式のベクトル b を読み込む場合のプログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

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

Fortran (逐次・マルチスレッド・マルチプロセス環境)

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

拡張 Matrix Market 形式による外部ファイル `matvec.mtx` の記述例を以下に示す (付録 A を参照).

```
%%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
```

外部ファイルからの読み込み

外部ファイルから行列 A のデータを読み込むには, 関数

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

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

を用いる. filename にはファイルパスを指定する. 対応するファイル形式は以下の通りである (ファイル形式については付録 A を参照).

- Matrix Market 形式
- Harwell-Boeing 形式

外部ファイルから行列 A とベクトル b, x のデータを読み込むには, 関数

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

を用いる. filename にはファイルパスを指定する. 対応するファイル形式は以下の通りである (ファイル形式については付録 A を参照).

- 拡張 Matrix Market 形式
- Harwell-Boeing 形式

3.4 線型方程式の求解

線型方程式 $Ax = b$ を指定された解法で解く場合, プログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

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

/* 行列とベクトルの作成 */

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

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
LIS_MATRIX A
LIS_VECTOR b,x
LIS_SOLVER solver

/* 行列とベクトルの作成 */

call lis_solver_create(solver,ierr)
call lis_solver_set_option('-i bicg -p none',solver,ierr)
call lis_solver_set_option('-tol 1.0e-12',solver,ierr)
call lis_solve(A,b,x,solver,ierr)
```

ソルバの作成

ソルバ (線型方程式解法の情報を格納する構造体) を作成するには, 関数

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

を用いる.

オプションの設定

線型方程式解法をソルバに設定するには, 関数

- 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)

または

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

を用いる. `lis_solver_set_optionC` は, ユーザプログラム実行時にコマンドラインで指定されたオプションをソルバに設定する関数である.

以下に指定可能なコマンドラインオプションを示す. `-i {cg|1}`は`-i cg` または`-i 1` を意味する. `-maxiter [1000]` は, `-maxiter` の既定値が 1000 であることを意味する.

線型方程式解法に関するオプション (既定値: `-i bicg`)

線型方程式解法	オプション	補助オプション	
CG	`-i {cg|1}`		
BiCG	`-i {bicg|2}`		
CGS	`-i {cgs|3}`		
BiCGSTAB	`-i {bicgstab|4}`		
BiCGSTAB(l)	`-i {bicgstabl|5}`	`-ell [2]`	次数 l
GPBiCG	`-i {gpbicg|6}`		
TFQMR	`-i {tfqmr|7}`		
Orthomin(m)	`-i {orthomin|8}`	`-restart [40]`	リスタート値 m
GMRES(m)	`-i {gmres|9}`	`-restart [40]`	リスタート値 m
Jacobi	`-i {jacobi|10}`		
Gauss-Seidel	`-i {gs|11}`		
SOR	`-i {sor|12}`	`-omega [1.9]`	緩和係数 ω $(0 < \omega < 2)$
BiCGSafe	`-i {bicgsafe|13}`		
CR	`-i {cr|14}`		
BiCR	`-i {bicr|15}`		
CRS	`-i {crs|16}`		
BiCRSTAB	`-i {bicrstab|17}`		
GPBiCR	`-i {gpbicr|18}`		
BiCRSafe	`-i {bicrsafe|19}`		
FGMRES(m)	`-i {fgmres|20}`	`-restart [40]`	リスタート値 m
IDR(s)	`-i {idrs|21}`	`-irestart [2]`	リスタート値 s
IDR(1)	`-i {idr1|22}`		
MINRES	`-i {minres|23}`		
COCG	`-i {cocg|24}`		
COCR	`-i {cocr|25}`		

前処理に関するオプション (既定値: `-p none`)

前処理	オプション	補助オプション	
なし	-p {none|0}		
Jacobi	-p {jacobi|1}		
ILU(k)	-p {ilu|2}	-ilu_fill [0]	フィルインレベル k
SSOR	-p {ssor|3}	-ssor_w [1.0]	緩和係数 ω $(0 < \omega < 2)$
Hybrid	-p {hybrid|4}	-hybrid_i [sor]	線型方程式解法
		-hybrid_maxiter [25]	最大反復回数
		-hybrid_tol [1.0e-3]	収束判定基準
		-hybrid_w [1.5]	SOR の緩和係数 ω $(0 < \omega < 2)$
		-hybrid_ell [2]	BiCGSTAB(l) の次数 l
		-hybrid_restart [40]	GMRES(m), Orthomin(m) の リスタート値 m
I+S	-p {is|5}	-is_alpha [1.0]	$I + \alpha S^{(m)}$ のパラメータ α
		-is_m [3]	$I + \alpha S^{(m)}$ のパラメータ m
SAINV	-p {sainv|6}	-sainv_drop [0.05]	ドロップ基準
SA-AMG	-p {saamg|7}	-saamg_unsym [false]	非対称版の選択 (行列構造は対称とする)
		-saamg_theta [0.05|0.12]	ドロップ基準 $a_{ij}^2 \le \theta^2 \|a_{ii}\|\|a_{jj}\|$ (対称|非対称)
Crout ILU	-p {iluc|8}	-iluc_drop [0.05]	ドロップ基準
		-iluc_rate [5.0]	最大フィルイン数の倍率
ILUT	-p {ilut|9}		
Additive Schwarz	-adds true	-adds_iter [1]	反復回数

その他のオプション

オプション		
-maxiter [1000]	最大反復回数	
-tol [1.0e-12]	収束判定基準 tol	
-tol_w [1.0]	収束判定基準 tol_w	
-print [0]	残差履歴の出力	
	-print {none|0}	残差履歴を出力しない
	-print {mem|1}	残差履歴をメモリに保存する
	-print {out|2}	残差履歴を標準出力に書き出す
	-print {all|3}	残差履歴をメモリに保存し, 標準出力に書き出す
-scale [0]	スケーリングの選択. 結果は元の行列, ベクトルに上書きされる	
	-scale {none|0}	スケーリングなし
	-scale {jacobi|1}	Jacobi スケーリング $D^{-1}Ax = D^{-1}b$ (D は $A = (a_{ij})$ の対角部分)
	-scale {symm_diag|2}	対角スケーリング $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ ($D^{-1/2}$ は対角要素の値が $1/\sqrt{a_{ii}}$ である対角行列)
-initx_zeros [1]	初期ベクトル x_0	
	-initx_zeros {false|0}	与えられた値を使用
	-initx_zeros {true|1}	すべての要素の値を 0 にする
-conv_cond [0]	収束条件	
	-conv_cond {nrm2_r|0}	$\|\|b - Ax\|\|_2 \le tol * \|\|b - Ax_0\|\|_2$
	-conv_cond {nrm2_b|1}	$\|\|b - Ax\|\|_2 \le tol * \|\|b\|\|_2$
	-conv_cond {nrm1_b|2}	$\|\|b - Ax\|\|_1 \le tol_w * \|\|b\|\|_1 + tol$
-omp_num_threads [t]	実行スレッド数 (t は最大スレッド数)	
-storage [0]	行列格納形式	
-storage_block [2]	BSR, BSC 形式のブロックサイズ	
-f [0]	線型方程式解法の精度	
	-f {double|0}	倍精度
	-f {quad|1}	4 倍精度

求解

線型方程式 $Ax = b$ を解くには, 関数

- 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)
```

を用いる.

3.5 固有値問題の求解

固有値問題 $Ax = \lambda x$ を指定された解法で解く場合, プログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

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

/* 行列とベクトルの作成 */

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

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
LIS_MATRIX A
LIS_VECTOR x
LIS_REAL evalue
LIS_ESOLVER esolver

/* 行列とベクトルの作成 */

call lis_esolver_create(esolver,ierr)
call lis_esolver_set_option('-e ii -i bicg -p none',esolver,ierr)
call lis_esolver_set_option('-etol 1.0e-12 -tol 1.0e-12',esolver,ierr)
call lis_esolve(A,x,evalue,esolver,ierr)
```

ソルバの作成

ソルバ (固有値解法の情報を格納する構造体) を作成するには, 関数

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

を用いる.

オプションの設定

固有値解法をソルバに設定するには, 関数

- 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)`

または

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

を用いる. lis_esolver_set_optionC は, ユーザプログラム実行時にコマンドラインで指定されたオプションをソルバに設定する関数である.

以下に指定可能なコマンドラインオプションを示す. -e {pi|1}は-e pi または-e 1 を意味する. -emaxiter [1000] は, -emaxiter の既定値が 1000 であることを意味する.

固有値解法に関するオプション (既定値: -e pi)

固有値解法	オプション	補助オプション	
Power	-e {pi|1}		
Inverse	-e {ii|2}	-i [bicg]	線型方程式解法
Approximate Inverse	-e {aii|3}	-i [bicg]	線型方程式解法
Rayleigh Quotient	-e {rqi|4}	-i [bicg]	線型方程式解法
CG	-e {cg|5}	-i [cg]	線型方程式解法
CR	-e {cr|6}	-i [bicg]	線型方程式解法
Jacobi-Davidson	-e {jd|7}	-i [cg]	線型方程式解法
Subspace	-e {si|8}	-ss [1]	部分空間の大きさ
Lanczos	-e {li|9}	-ss [1]	部分空間の大きさ
Arnoldi	-e {ai|10}	-ss [1]	部分空間の大きさ

前処理に関するオプション (既定値: -p ilu)

前処理	オプション	補助オプション	
なし	-p {none|0}		
Jacobi	-p {jacobi|1}		
ILU(k)	-p {ilu|2}	-ilu_fill [0]	フィルインレベル k
SSOR	-p {ssor|3}	-ssor_w [1.0]	緩和係数 ω $(0 < \omega < 2)$
Hybrid	-p {hybrid|4}	-hybrid_i [sor]	線型方程式解法
		-hybrid_maxiter [25]	最大反復回数
		-hybrid_tol [1.0e-3]	収束判定基準
		-hybrid_w [1.5]	SOR の緩和係数 ω $(0 < \omega < 2)$
		-hybrid_ell [2]	BiCGSTAB(l) の次数 l
		-hybrid_restart [40]	GMRES(m), Orthomin(m) の リスタート値 m
I+S	-p {is|5}	-is_alpha [1.0]	$I + \alpha S^{(m)}$ のパラメータ α
		-is_m [3]	$I + \alpha S^{(m)}$ のパラメータ m
SAINV	-p {sainv|6}	-sainv_drop [0.05]	ドロップ基準
SA-AMG	-p {saamg|7}	-saamg_unsym [false]	非対称版の選択 (行列構造は対称とする)
		-saamg_theta [0.05|0.12]	ドロップ基準 $a_{ij}^2 \le \theta^2 \lvert a_{ii} \rvert \lvert a_{jj} \rvert$ (対称|非対称)
crout ILU	-p {iluc|8}	-iluc_drop [0.05]	ドロップ基準
		-iluc_rate [5.0]	最大フィルイン数の倍率
ILUT	-p {ilut|9}		
Additive Schwarz	-adds true	-adds_iter [1]	反復回数

その他のオプション

オプション		
-emaxiter [1000]	最大反復回数	
-etol [1.0e-12]	収束判定基準	
-eprint [0]	残差履歴の出力	
	-eprint {none|0}	残差履歴を出力しない
	-eprint {mem|1}	残差履歴をメモリに保存する
	-eprint {out|2}	残差履歴を標準出力に書き出す
	-eprint {all|3}	残差履歴をメモリに保存し, 標準出力に書き出す
-ie [ii]	Subspace, Lanczos, Arnoldi 法の内部で使用する固有値解法の指定	
	-ie {pi|1}	Power (Subspace のみ)
	-ie {ii|2}	Inverse
	-ie {aii|3}	Approximate Inverse
	-ie {rqi|4}	Rayleigh Quotient
	-ie {cg|5}	CG (Lanczos のみ)
	-ie {cr|6}	CR (Lanczos, Arnoldi)
	-ie {jd|7}	Jacobi-Davidson (Lanczos のみ)
-shift [0.0]	固有値のシフト量	
-initx_ones [1]	初期ベクトル x_0	
	-initx_ones {false|0}	与えられた値を使用
	-initx_ones {true|1}	すべての要素の値を 1 にする
-omp_num_threads [t]	実行スレッド数	
	t は最大スレッド数	
-estorage [0]	行列格納形式	
-estorage_block [2]	BSR, BSC 形式のブロックサイズ	
-ef [0]	固有値解法の精度	
	-ef {double|0}	倍精度
	-ef {quad|1}	4 倍精度

求解

固有値問題 $Ax = \lambda x$ を解くには, 関数

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

を用いる.

3.6 プログラムの作成

線型方程式 $Ax = b$ を指定された解法で解き, その解を標準出力に書き出すプログラムを以下に示す.

行列 A は次数 12 の 3 重対角行列

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

である. 右辺ベクトル b は解 x がすべて 1 となるよう設定される.

このプログラムはディレクトリ `lis-($VERSION)/test` にある.

検証プログラム: test4.c

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

検証プログラム: test4f.F

```
      implicit none

#include "lisf.h"

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

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

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

      stop
      end
```

3.7 前処理とソルバの分離

前節では, 前処理は lis_solve が呼び出されるたびに更新されていた. 本節では, 前処理とソルバを分離し, 前処理をソルバと切り離して更新する方法について述べる. この方法は, 非線形偏微分方程式を含むある種の問題に対して極めて有効である. 実際, Newton-Raphson 法では反復法が用いられ, 各反復での線型方程式の解をもとに解ベクトルが逐次的に求められる. この機能を実現するためには, 以下の関数を実装する必要がある.

- lis_matrix_psd_set_value: 行列要素の値を再定義する.
- lis_matrix_psd_reset_scale: 行列のスケーリングに関する情報を更新する. この関数, 及び次の関数は, 自明でないスケーリングを行う場合にのみ用いる.
- lis_vector_psd_reset_scale: ベクトルのスケーリングに関する情報を更新する.
- lis_solver_set_matrix: 与えられた行列とソルバを関連付ける. この関数は lis_precon_psd_create より前に呼び出されなければならない.
- lis_precon_psd_create: 選択された前処理のためのデータ構造を生成する.
- lis_precon_psd_update: 前処理を評価する.

この方法には現在以下の制限がある.

- 利用可能な行列格納形式は CSR のみである.
- 利用可能なソルバは GMRES のみである.
- 利用可能な前処理は ILU(k) 及び SA-AMG のみである.

実装例を疑似コードにより示す. 詳細は test8f.F90 を参照のこと.

疑似コード

```
      PROGRAM psd_driver

      implicit none

#include "lisf.h"

      LIS_INTEGER i,n,gn,is,ie,iter,ierr
      LIS_MATRIX A
      LIS_VECTOR b,x
      LIS_SOLVER solver
      REAL :: u(:),du

      CALL lis_initialize(ierr)

     !==================================================================
     ! initialization, only done once
     !==================================================================
      CALL lis_matrix_create(LIS_COMM_WORLD,A,ierr)
      CALL lis_matrix_set_size(A,0,n,ierr)
      CALL lis_matrix_get_size(A,n,gn,ierr)
      CALL lis_matrix_get_range(A,is,ie,ierr)

      CALL UpdateLinearSystem(RHS,LHS)
      DO i=is,ie-1
         DO j=1,gn
             IF (LHS(i,j) exists) THEN
                   CALL lis_matrix_set_value(LIS_INS_VALUE,i,j,LHS(i,j),A,ierr)
             END IF
         END DO
      END DO
      CALL lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
      CALL lis_matrix_assemble(A,ierr)

      CALL lis_vector_duplicate(A,b,ierr)
      CALL lis_vector_duplicate(A,x,ierr)
      DO i=is,ie-1
           CALL lis_vector_set_value(LIS_INS_VALUE,i,RHS(i),b,ierr)
      END DO
      u = u_initial

      CALL lis_solver_create(solver,ierr)
      WRITE(UNIT=options,FMT='(a)') "-p ilu -i gmres -print out -scale none"
      CALL lis_solver_set_option(TRIM(options),solver,ierr)

     !==================================================================
     ! everything up to this point is more or less identical to the standard workflow.
     ! Now comes the preconditioner initialization, and the Newton-Raphson
     ! iteration.
     !==================================================================
     CALL lis_solver_set_matrix(A,solver,ierr)
     CALL lis_precon_psd_create(solver,precon,ierr)
     ! evaluate the preconditioner, at least once . . .
     CALL lis_precon_psd_update(solver,precon,ierr)

```

疑似コード（続き）

```
    DO

        IF (UpdateLHS) THEN
            DO i=is,ie-1
                DO j=1,gn
                    IF (component (i,j) exists) THEN
                        CALL lis_matrix_psd_set_value(LIS_INS_VALUE,i,j,LHS(i,j),A,ierr)
                    END IF
                END DO
             END DO
             CALL lis_matrix_psd_reset_scale(A,ierr)
        END IF

        ! update RHS every iteration
        DO i=is,ie-1
            CALL lis_vector_set_value(LIS_INS_VALUE,i,RHS(i),b,ierr)
        END DO
        CALL lis_vector_psd_reset_scale(A,ierr)

        IF (UpdateLHS) THEN
            CALL lis_precon_psd_update(solver,precon,ierr)
        END IF
        CALL lis_solve_kernel(A,b,x,solver,precon,ierr)
        CALL lis_solver_get_iter(solver,iter,ierr)
        write(*,*) 'number of iterations = ',iter
        CALL lis_vector_print(x,ierr)

        ! update the solution
        DO i=is,ie-1
            CALL lis_vector_get_value(x,i,du,ierr)
            u(i)=u(i)-du
        END DO

        CALL UpdateLinearSystem(RHS,LHS)

        IF (termination criteria satisfied) EXIT

    END DO


     CALL lis_matrix_destroy(A,ierr)
     CALL lis_vector_destroy(b,ierr)
     CALL lis_vector_destroy(x,ierr)
     CALL lis_vector_destroy(u,ierr)
     CALL lis_solver_destroy(solver,ierr)

     CALL lis_finalize(ierr)

     END PROGRAM psd_driver
```

3.8 実行ファイルの生成

test4.c から実行ファイルを生成する方法について述べる．ディレクトリ lis-($VERSION)/test にある検証プログラム test4.c を SGI Altix 3700 上の Intel C Compiler (icc), Intel Fortran Compiler (ifort) でコンパイルする場合の例を以下に示す．SA-AMG 前処理には Fortran 90 で記述されたコードが含まれるため, SA-AMG 前処理を使用する場合には Fortran 90 コンパイラでリンクしなければならない．また, マルチプロセス環境ではプリプロセッサマクロ USE_MPI が定義されなければならない．64bit 整数型を使用する場合は, C プログラムではプリプロセッサマクロ _LONG__LONG, Fortran プログラムではプリプロセッサマクロ LONG__LONG が定義されなければならない．

逐次環境

コンパイル
```
> icc -c -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -o test4 test4.o -L($INSTALLDIR)/lib -llis
```
リンク (--enable-saamg)
```
> ifort -nofor_main -o test4 test4.o -L($INSTALLDIR)/lib -llis
```

マルチスレッド環境

コンパイル
```
> icc -c -openmp -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis
```
リンク (--enable-saamg)
```
> ifort -nofor_main -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis
```

マルチプロセス環境

コンパイル
```
> icc -c -DUSE_MPI -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi
```
リンク (--enable-saamg)
```
> ifort -nofor_main -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi
```

マルチスレッド・マルチプロセス環境

コンパイル
```
> icc -c -openmp -DUSE_MPI -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi
```
リンク (--enable-saamg)
```
> ifort -nofor_main -openmp -o test4 test4.o -L($INSTALLDIR)/lib -llis -lmpi
```

次に, test4f.F から実行ファイルを生成する方法について述べる．ディレクトリ lis-($VERSION)/test にある検証プログラム test4f.F を SGI Altix 3700 上の Intel Fortran Compiler (ifort) でコンパイルする場合の例を以下に示す．Fortran のユーザプログラムにはコンパイラ指示文が含まれるため, プリプロセッサを使用するようコンパイラに指示しなければならない．ifort の場合は, オプション-fpp が必要である．

逐次環境

コンパイル

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

リンク

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

マルチスレッド環境

コンパイル

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

リンク

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

マルチプロセス環境

コンパイル

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

リンク

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

マルチスレッド・マルチプロセス環境

コンパイル

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

リンク

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

3.9 実行

ディレクトリ `lis-($VERSION)/test` にある検証プログラム `test4` または `test4f` を SGI Altix 3700 上のそれぞれの環境で

逐次環境

```
> ./test4 -i bicgstab
```

マルチスレッド環境

```
> env OMP_NUM_THREADS=2 ./test4 -i bicgstab
```

マルチプロセス環境

```
> mpirun -np 2 ./test4 -i bicgstab
```

マルチスレッド・マルチプロセス環境

```
> mpirun -np 2 env OMP_NUM_THREADS=2 ./test4 -i bicgstab
```

と入力して実行すると, 以下のように解が標準出力に書き出される.

```
initial vector x      : 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-00
     1  1.000000e+00
     2  1.000000e-00
     3  1.000000e+00
     4  1.000000e-00
     5  1.000000e+00
     6  1.000000e+00
     7  1.000000e-00
     8  1.000000e+00
     9  1.000000e-00
    10  1.000000e+00
    11  1.000000e-00
```

3.10 プロセス上での自由度について

MPI プロセスを複数使用する場合には, $global\_n = 0$, $local\_n \geq 0$ であってもよい. すなわち, 一つ以上のプロセスにおいて自由度を 0 とすることができる. ただし, その場合においても $global\_n = 0$ ならば $local\_n$ の総和は 0 より大きくなければならない.

4　4 倍精度演算

反復法の計算では, 丸め誤差の影響によって収束が停滞することがある. 本ライブラリでは, long double 型及び倍精度浮動小数点数を 2 個用いた"double-double"[49, 50] 型の 4 倍精度演算を用いることにより, 収束を改善することが可能である. double-double 型演算では, 浮動小数 a を $a = a.hi + a.lo$, $\frac{1}{2}\mathrm{ulp}(a.hi) \geq |a.lo|$ (上位 $a.hi$ と下位 $a.lo$ は倍精度浮動小数) により定義し, Dekker[51] と Knuth[52] のアルゴリズムに基づいて倍精度の四則演算の組み合わせにより 4 倍精度演算を実現する. double-double 型の演算は一般に Fortran の 4 倍精度演算より高速である [53] が, Fortran の表現形式 [54] では仮数部が 112 ビットであるのに対して, 倍精度浮動小数を 2 個使用するため, 仮数部が 104 ビットとなり, 8 ビット少ない. また, 指数部は倍精度浮動小数と同じ 11 ビットである.

double-double 型演算では, 入力として与えられる行列, ベクトル, 及び出力の解は倍精度である. ユーザプログラムは 4 倍精度変数を直接扱うことはなく, オプションとして 4 倍精度演算を使用するかどうかを指定するだけでよい. なお, Intel 系のアーキテクチャに対しては Streaming SIMD Extensions (SSE) 命令を用いて高速化を行う [55].

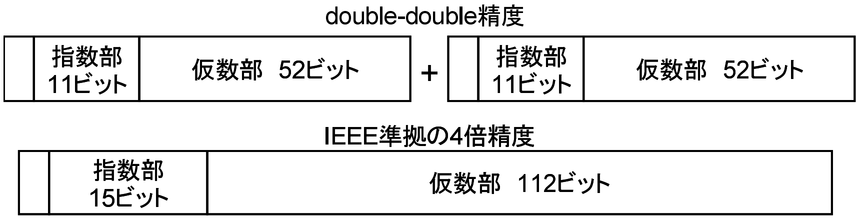

図 2: double-double 型のビット数

4.1　4 倍精度演算の使用

Toepliz 行列

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

に対する線型方程式 $Ax = b$ を指定された解法で解き, 解を標準出力に書き出す検証プログラムが `test5.c` である. 右辺ベクトル b は解 x がすべて 1 となるよう設定される. n は行列 A の次数である. `test5` において,

倍精度の場合

```
> ./test5 200 2.0 -f double
```

または

```
> ./test5 200 2.0
```

と入力して実行すると, 以下の結果が得られる.

```
n = 200, gamma = 2.000000

initial vector x      : 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
```

4 倍精度の場合

```
> ./test5 200 2.0 -f quad
```

と入力して実行すると, 以下の結果が得られる.

```
n = 200, gamma = 2.000000

initial vector x      : 0
precision             : quad
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 = 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 行列格納形式

本節では, ライブラリで使用できる行列の格納形式について述べる. 行列の行 (列) 番号は 0 から始まるものとする. 次数 $n \times n$ の行列 $A = (a_{ij})$ の非零要素数を nnz とする.

5.1 Compressed Sparse Row (CSR)

CSR 形式では, データを 3 つの配列 (`ptr`,`index`,`value`) に格納する.

- 長さ nnz の倍精度配列 `value` は, 行列 A の非零要素の値を行方向に沿って格納する.
- 長さ nnz の整数配列 `index` は, 配列 `value` に格納された非零要素の列番号を格納する.
- 長さ $n+1$ の整数配列 `ptr` は, 配列 `value` と `index` の各行の開始位置を格納する.

5.1.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の CSR 形式での格納方法を図 3 に示す. この行列を CSR 形式で作成する場合, プログラムは以下のように記述する.

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

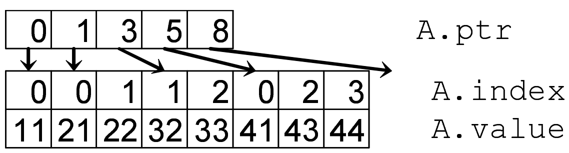

図 3: CSR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.1.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の CSR 形式での格納方法を図 4 に示す. 2 プロセス上にこの行列を CSR 形式で作成する場合, プログラムは以下のように記述する.

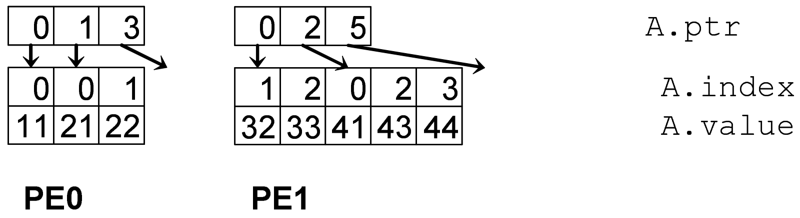

図 4: CSR 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.1.3 関連する関数

配列の関連付け

CSR 形式の配列を行列 A に関連付けるには, 関数

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

を用いる.

5.2 Compressed Sparse Column (CSC)

CSC 形式では, データを 3 つの配列 (`ptr,index,value`) に格納する.

- 長さ nnz の倍精度配列 `value` は, 行列 A の非零要素の値を列方向に沿って格納する.
- 長さ nnz の整数配列 `index` は, 配列 `value` に格納された非零要素の行番号を格納する.
- 長さ $n+1$ の整数配列 `ptr` は, 配列 `value` と `index` の各列の開始位置を格納する.

5.2.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の CSC 形式での格納方法を図 5 に示す. この行列を CSC 形式で作成する場合, プログラムは以下のように記述する.

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

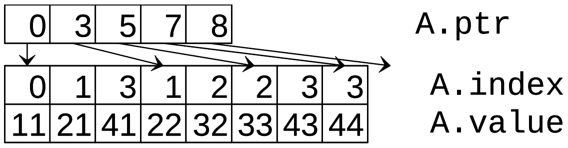

図 5: CSC 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.2.2　行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の CSC 形式での格納方法を図 6 に示す. 2 プロセス上にこの行列を CSC 形式で作成する場合, プログラムは以下のように記述する.

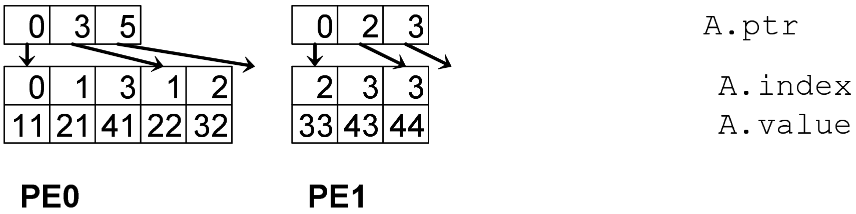

図 6: CSC 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.2.3　関連する関数

配列の関連付け

CSC 形式の配列を行列 A に関連付けるには, 関数

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

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

を用いる.

5.3 Modified Compressed Sparse Row (MSR)

MSR 形式では, データを 2 つの配列 (`index,value`) に格納する. ndz を対角部分の零要素数とする.

- 長さ $nnz + ndz + 1$ の倍精度配列 `value` は, 第 n 要素までは行列 A の対角部分を格納する. 第 $n+1$ 要素は使用しない. 第 $n+2$ 要素からは行列 A の対角部分以外の非零要素の値を行方向に沿って格納する.

- 長さ $nnz + ndz + 1$ の整数配列 `index` は, 第 $n+1$ 要素までは行列 A の非対角部分の各行の開始位置を格納する. 第 $n+2$ 要素からは行列 A の非対角部分の配列 `value` に格納された非零要素の列番号を格納する.

5.3.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の MSR 形式での格納方法を図 7 に示す. この行列を MSR 形式で作成する場合, プログラムは以下のように記述する.

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

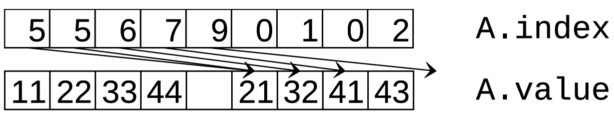

図 7: MSR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.3.2　行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の MSR 形式での格納方法を図 8 に示す. 2 プロセス上にこの行列を MSR 形式で作成する場合, プログラムは以下のように記述する.

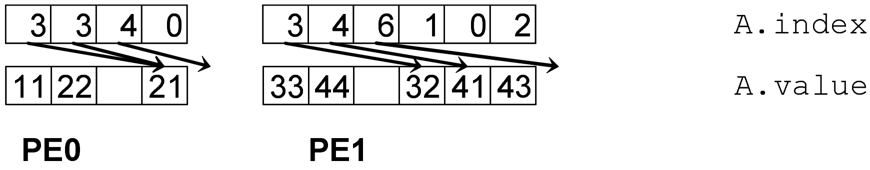

図 8: MSR 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.3.3　関連する関数

配列の関連付け

MSR 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

5.4 Diagonal (DIA)

DIA 形式では, データを 2 つの配列 (index, value) に格納する. nnd を行列 A の非零な対角要素の本数とする.

- 長さ $nnd \times n$ の倍精度配列 value は, 行列 A の非零な対角要素の値を格納する.
- 長さ nnd の整数配列 index は, 主対角要素から各対角要素へのオフセットを格納する.

マルチスレッド環境では以下のように格納する.
データを 2 つの配列 (index, value) に格納する. $nprocs$ をスレッド数とする. nnd_p を行列 A を行ブロック分割した部分行列の非零な対角部分の本数とする. $maxnnd$ を nnd_p の値の最大値とする.

- 長さ $maxnnd \times n$ の倍精度配列 value は, 行列 A を行ブロック分割した部分行列の非零な対角要素の値を格納する.
- 長さ $nprocs \times maxnnd$ の整数配列 index は, 主対角要素から各対角要素へのオフセットを格納する.

5.4.1 行列の作成 (逐次環境)

行列 A の DIA 形式での格納方法を図 9 に示す. この行列を DIA 形式で作成する場合, プログラムは以下のように記述する.

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

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

図 9: DIA 形式のデータ構造 (逐次環境)

逐次環境

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

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

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

5.4.2　行列の作成 (マルチスレッド環境)

2 スレッド上への行列 A の DIA 形式での格納方法を図 10 に示す. 2 スレッド上にこの行列を DIA 形式で作成する場合, プログラムは以下のように記述する.

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

図 10: DIA 形式のデータ構造 (マルチスレッド環境)

マルチスレッド環境

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

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

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

5.4.3 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の DIA 形式での格納方法を図 11 に示す. 2 プロセス上にこの行列を DIA 形式で作成する場合, プログラムは以下のように記述する.

PE0

-1	0			A.index
0	21	11	22	A.value

PE1

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

図 11: DIA 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.4.4 関連する関数

配列の関連付け

DIA 形式の配列を行列 A に関連付けるには, 関数

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

を用いる.

5.5 Ellpack-Itpack Generalized Diagonal (ELL)

ELL 形式では, データを 2 つの配列 (index, value) に格納する. $maxnzr$ を行列 A の各行での非零要素数の最大値とする.

- 長さ $maxnzr \times n$ の倍精度配列 value は, 行列 A の各行の非零要素の値を列方向に沿って格納する. 最初の列は各行の最初の非零要素からなる. ただし, 格納する非零要素がない場合は 0 を格納する.
- 長さ $maxnzr \times n$ の整数配列 index は, 配列 value に格納された非零要素の列番号を格納する. ただし, 第 i 行の非零要素数を nnz とすると index[$nnz \times n + i$] にはその行番号 i を格納する.

5.5.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の ELL 形式での格納方法を図 12 に示す. この行列を ELL 形式で作成する場合, プログラムは以下のように記述する.

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

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

図 12: ELL 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.5.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の ELL 形式での格納方法を図 13 に示す. 2 プロセス上にこの行列を ELL 形式で作成する場合, プログラムは以下のように記述する.

PE0

A.index	0	0	0	1
A.value	11	21	0	22

PE1

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

図 13: ELL 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.5.3 関連する関数

配列の関連付け

ELL 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

5.6 Jagged Diagonal (JAD)

JAD 形式では, 最初に各行の非零要素数の大きい順に行の並び替えを行い, 各行の非零要素を列方向に沿って格納する. JAD 形式では, データを 4 つの配列 (`perm`, `ptr`, `index`, `value`) に格納する. $maxnzr$ を行列 A の各行での非零要素数の最大値とする.

- 長さ n の整数配列 `perm` は, 並び替えた行番号を格納する.
- 長さ nnz の倍精度配列 `value` は, 並び替えられた行列 A の鋸歯状対角要素の値を格納する. 最初の鋸歯状対角要素は各行の第 1 非零要素からなる. 次の鋸歯状対角要素は各行の第 2 非零要素からなる. これを順次繰り返していく.
- 長さ nnz の整数配列 `index` は, 配列 `value` に格納された非零要素の列番号を格納する.
- 長さ $maxnzr+1$ の整数配列 `ptr` は, 各鋸歯状対角要素の開始位置を格納する.

マルチスレッド環境では以下のように格納する.
データを 4 つの配列 (`perm`, `ptr`, `index`, `value`) に格納する. $nprocs$ をスレッド数とする. $maxnzr_p$ を行列 A を行ブロック分割した部分行列の各行での非零要素数の最大値とする. $maxmaxnzr$ は配列 $maxnzr_p$ の値の最大値である.

- 長さ n の整数配列 `perm` は, 行列 A を行ブロック分割した部分行列を並び替えた行番号を格納する.
- 長さ nnz の倍精度配列 `value` は, 並び替えられた行列 A の鋸歯状対角要素の値を格納する. 最初の鋸歯状対角要素は各行の第 1 非零要素からなる. 次の鋸歯状対角要素は各行の第 2 非零要素からなる. これを順次繰り返していく.
- 長さ nnz の整数配列 `index` は, 配列 `value` に格納された非零要素の列番号を格納する.
- 長さ $nprocs \times (maxmaxnzr+1)$ の整数配列 `ptr` は, 行列 A を行ブロック分割した部分行列の各鋸歯状対角要素の開始位置を格納する.

5.6.1 行列の作成 (逐次環境)

行列 A の JAD 形式での格納方法を図 14 に示す. この行列を JAD 形式で作成する場合, プログラムは以下のように記述する.

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

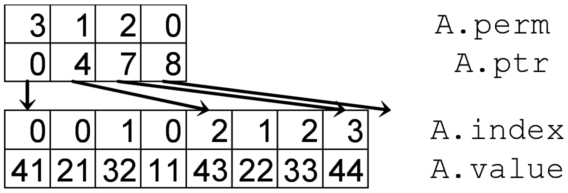

図 14: JAD 形式のデータ構造 (逐次環境)

逐次環境

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

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

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

5.6.2 行列の作成 (マルチスレッド環境)

2 スレッド上への行列 A の JAD 形式での格納方法を図 15 に示す. 2 スレッド上にこの行列を JAD 形式で作成する場合, プログラムは以下のように記述する.

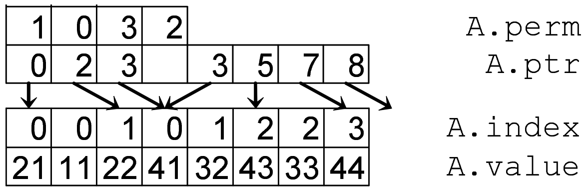

図 15: JAD 形式のデータ構造 (マルチスレッド環境)

マルチスレッド環境

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

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

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

5.6.3 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の JAD 形式での格納方法を図 16 に示す. 2 プロセス上にこの行列を JAD 形式で作成する場合, プログラムは以下のように記述する.

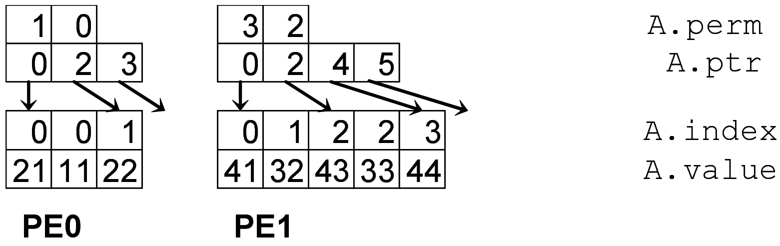

図 16: JAD 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.6.4 関連する関数

配列の関連付け

JAD 形式の配列を行列 A に関連付けるには, 関数

- 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)

を用いる.

5.7 Block Sparse Row (BSR)

BSR 形式では, 行列を $r \times c$ の大きさの部分行列 (ブロックと呼ぶ) に分解する. BSR 形式では, CSR 形式と同様の手順で非零ブロック (少なくとも 1 つの非零要素が存在する) を格納する. $nr = n/r$, $bnnz$ を A の非零ブロック数とする. BSR 形式では, データを 3 つの配列 (`bptr`, `bindex`, `value`) に格納する.

- 長さ $bnnz \times r \times c$ の倍精度配列 `value` は, 非零ブロックの全要素の値を格納する.
- 長さ $bnnz$ の整数配列 `bindex` は, 非零ブロックのブロック列番号を格納する.
- 長さ $nr+1$ の整数配列 `bptr` は, 配列 `bindex` のブロック行の開始位置を格納する.

5.7.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の BSR 形式での格納方法を図 17 に示す. この行列を BSR 形式で作成する場合, プログラムは以下のように記述する.

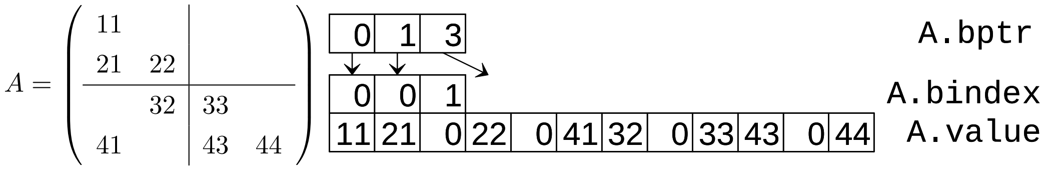

図 17: BSR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.7.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の BSR 形式での格納方法を図 18 に示す. 2 プロセス上にこの行列を BSR 形式で作成する場合, プログラムは以下のように記述する.

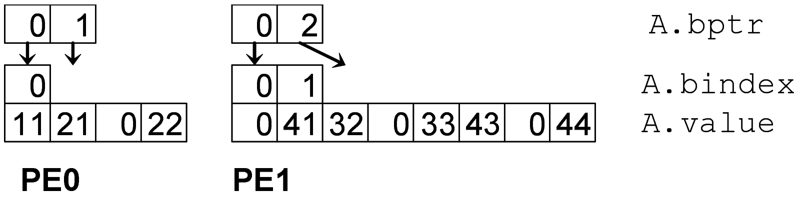

図 18: BSR 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.7.3 関連する関数

配列の関連付け

BSR 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

5.8 Block Sparse Column (BSC)

BSC 形式では, 行列を $r \times c$ の大きさの部分行列 (ブロックと呼ぶ) に分解する. BSC 形式では, CSC 形式と同様の手順で非零ブロック (少なくとも 1 つの非零要素が存在する) を格納する. $nc = n/c$, $bnnz$ を A の非零ブロック数とする. BSC 形式では, データを 3 つの配列 (`bptr`, `bindex`, `value`) に格納する.

- 長さ $bnnz \times r \times c$ の倍精度配列 `value` は, 非零ブロックの全要素の値を格納する.
- 長さ $bnnz$ の整数配列 `bindex` は, 非零ブロックのブロック行番号を格納する.
- 長さ $nc + 1$ の整数配列 `bptr` は, 配列 `bindex` のブロック列の開始位置を格納する.

5.8.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の BSC 形式での格納方法を図 19 に示す. この行列を BSC 形式で作成する場合, プログラムは以下のように記述する.

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

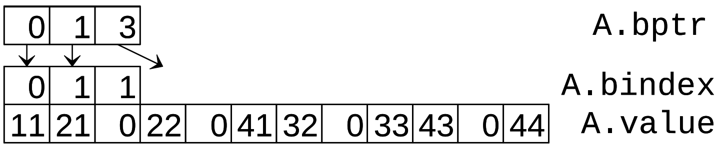

図 19: BSC 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.8.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の BSC 形式での格納方法を図 20 に示す. 2 プロセス上にこの行列を BSC 形式で作成する場合, プログラムは以下のように記述する.

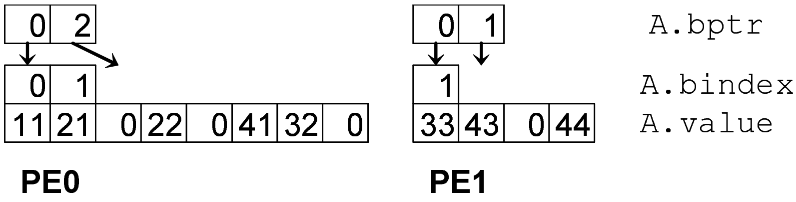

図 20: BSC 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.8.3 関連する関数

配列の関連付け

BSC 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

5.9 Variable Block Row (VBR)

VBR 形式は BSR 形式を一般化したものである. 行と列の分割位置は配列 (`row`, `col`) で与えられる. VBR 形式では, CSR 形式と同様の手順で非零ブロック (少なくとも 1 つの非零要素が存在する) を格納する. nr, nc をそれぞれ行分割数, 列分割数とする. $bnnz$ を A の非零ブロック数, nnz を非零ブロックの全要素数とする. VBR 形式では, データを 6 つの配列 (`bptr`, `bindex`, `row`, `col`, `ptr`, `value`) に格納する.

- 長さ $nr + 1$ の整数配列 `row` は, ブロック行の開始行番号を格納する.
- 長さ $nc + 1$ の整数配列 `col` は, ブロック列の開始列番号を格納する.
- 長さ $bnnz$ の整数配列 `bindex` は, 非零ブロックのブロック列番号を格納する.
- 長さ $nr + 1$ の整数配列 `bptr` は, 配列 `bindex` のブロック行の開始位置を格納する.
- 長さ nnz の倍精度配列 `value` は, 非零ブロックの全要素の値を格納する.
- 長さ $bnnz + 1$ の整数配列 `ptr` は, 配列 `value` の非零ブロックの開始位置を格納する.

5.9.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の VBR 形式での格納方法を図 21 に示す. この行列を VBR 形式で作成する場合, プログラムは以下のように記述する.

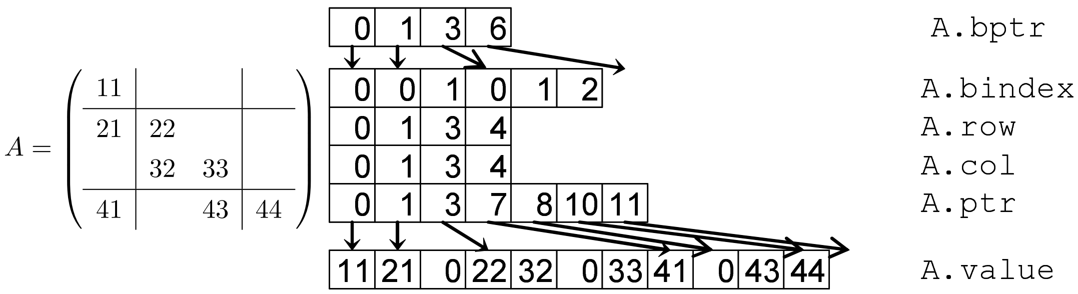

図 21: VBR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.9.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の VBR 形式での格納方法を図 22 に示す. 2 プロセス上にこの行列を VBR 形式で作成する場合, プログラムは以下のように記述する.

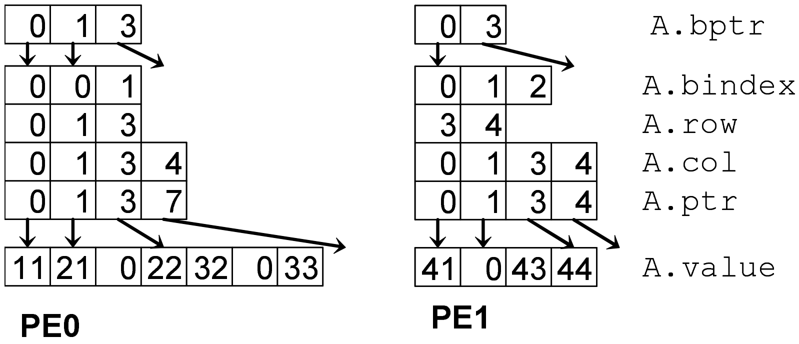

図 22: VBR 形式のデータ構造 (逐次・マルチスレッド環境)

マルチプロセス環境

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

5.9.3 関連する関数

配列の関連付け

VBR 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

5.10 Coordinate (COO)

COO 形式では, データを 3 つの配列 (row, col, value) に格納する.

- 長さ nnz の倍精度配列 value は, 非零要素の値を格納する.
- 長さ nnz の整数配列 row は, 非零要素の行番号を格納する.
- 長さ nnz の整数配列 col は, 非零要素の列番号を格納する.

5.10.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の COO 形式での格納方法を図 23 に示す. この行列を COO 形式で作成する場合, プログラムは以下のように記述する.

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

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

図 23: COO 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.10.2　行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の COO 形式での格納方法を図 24 に示す. 2 プロセス上にこの行列を COO 形式で作成する場合, プログラムは以下のように記述する.

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

図 24: COO 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.10.3　関連する関数

配列の関連付け

COO 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

5.11 Dense (DNS)

DNS 形式では, データを 1 つの配列 (`value`) に格納する.

- 長さ $n \times n$ の倍精度配列 `value` は, 列優先で要素の値を格納する.

5.11.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の DNS 形式での格納方法を図 25 に示す. この行列を DNS 形式で作成する場合, プログラムは以下のように記述する.

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

11	21	0	41	0	22	32	0	A.Value
0	0	33	43	0	0	0	44	

図 25: DNS 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

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

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

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

5.11.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の DNS 形式での格納方法を図 26 に示す. 2 プロセス上にこの行列を DNS 形式で作成する場合, プログラムは以下のように記述する.

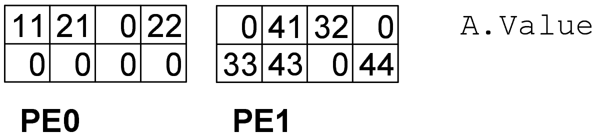

図 26: DNS 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

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

5.11.3 関連する関数

配列の関連付け

DNS 形式の配列を行列 A に関連付けるには, 関数

- 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)`

を用いる.

6 関数

本節では, ユーザが使用できる関数について述べる. 関数の解説は C を基準に記述する. 配列の要素番号は, C では 0 から, Fortran では 1 から始まるものとする. なお, 各ソルバの状態は以下のように定義する.

`LIS_SUCCESS(0)`	正常終了
`LIS_ILL_OPTION(1)`	オプション不正
`LIS_BREAKDOWN(2)`	ブレイクダウン（ゼロ除算）
`LIS_OUT_OF_MEMORY(3)`	メモリ不足
`LIS_MAXITER(4)`	最大反復回数超過
`LIS_NOT_IMPLEMENTED(5)`	未実装
`LIS_ERR_FILE_IO(6)`	ファイル I/O エラー

6.1 ベクトルの操作

ベクトル v の次数を $global\_n$ とする．ベクトル v を $nprocs$ 個のプロセスで行ブロック分割する場合の各ブロックの行数を $local\_n$ とする．$global\_n$ をグローバルな次数, $local\_n$ をローカルな次数と呼ぶ.

6.1.1 lis_vector_create

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

機能

ベクトルを作成する.

入力

LIS_Comm	MPI コミュニケータ

出力

v	ベクトル
ierr	リターンコード

注釈

逐次, マルチスレッド環境では, comm の値は無視される.

6.1.2 lis_vector_destroy

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

機能

不要になったベクトルをメモリから破棄する.

入力

v	メモリから破棄するベクトル

出力

ierr	リターンコード

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)
```

機能

既存のベクトルまたは行列と同じ情報を持つベクトルを作成する.

入力

vin	複製元のベクトルまたは行列

出力

vout	複製先のベクトル
ierr	リターンコード

注釈

vin には LIS_VECOTR または LIS_MATRIX を指定することが可能である. 関数 lis_vector_duplicate は要素の値は複製せず, メモリ領域のみ確保する. 値も複製する場合は, この関数の後に関数 lis_vector_copy を呼び出す.

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)
```

機能

ベクトルの次数を設定する.

入力

v	ベクトル
local_n	ベクトルのローカルな次数
global_n	ベクトルのグローバルな次数

出力

ierr	リターンコード

注釈

local_n か *global_n* のどちらか一方を与えなければならない.

逐次, マルチスレッド環境では, *local_n* は *global_n* に等しい. したがって, `lis_vector_set_size(v,n,0)` と `lis_vector_set_size(v,0,n)` は, いずれも次数 n のベクトルを作成する.

マルチプロセス環境においては, `lis_vector_set_size(v,n,0)` は各プロセス上に次数 n の部分ベクトルを作成する. 一方, `lis_vector_set_size(v,0,n)` は各プロセス p 上に次数 m_p の部分ベクトルを作成する. m_p はライブラリ側で決定される.

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)
```

機能

ベクトル v の次数を得る.

入力

v	ベクトル

出力

local_n	ベクトルのローカルな次数
global_n	ベクトルのグローバルな次数
ierr	リターンコード

注釈

逐次, マルチスレッド環境では, $local\_n$ は $global\_n$ に等しい.

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)
```

機能

部分ベクトル v が全体ベクトルのどこに位置するのかを調べる.

入力

v	部分ベクトル

出力

is	部分ベクトル v の全体ベクトル中での開始位置
ie	部分ベクトル v の全体ベクトル中での終了位置+1
ierr	リターンコード

注釈

逐次, マルチスレッド環境では, ベクトルの次数が n ならば, C 版では $is = 0$, $ie = n$, Fortran 版では $is = 1$, $ie = n + 1$ である.

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)
```

機能

ベクトル v の第 i 行にスカラ値を代入する.

入力

`flag`	`LIS_INS_VALUE` 挿入 : $v[i] = value$
	`LIS_ADD_VALUE` 加算代入: $v[i] = v[i] + value$
`i`	代入する場所
`value`	代入するスカラ値
`v`	ベクトル

出力

`v`	第 i 行にスカラ値が代入されたベクトル
`ierr`	リターンコード

注釈

マルチプロセス環境では, 全体ベクトルの第 i 行を指定する.

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)
```

機能

ベクトル v の第 i 行の値を取得する.

入力

i	取得する場所
v	値を取得するベクトル

出力

value	スカラ値
ierr	リターンコード

注釈

マルチプロセス環境では, 全体ベクトルの第 i 行を指定する.

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)
```

機能

ベクトル v の第 $index[i]$ 行にスカラ値 $value[i]$ を代入する.

入力

flag	LIS_INS_VALUE 挿入 : $v[index[i]] = value[i]$ LIS_ADD_VALUE 加算代入: $v[index[i]] = v[index[i]] + value[i]$
count	代入するスカラ値を格納する配列の要素数
index	代入する場所を格納する配列
value	代入するスカラ値を格納する配列
v	ベクトル

出力

v	第 $index[i]$ 行にスカラ値 $value[i]$ が代入されたベクトル
ierr	リターンコード

注釈

マルチプロセス環境では, 全体ベクトルの第 $index[i]$ 行を指定する.

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)
```

機能

ベクトル v の第 $start+i$ 行の値 $(i=0,1,...,count-1)$ を $value[i]$ に格納する.

入力

start	取得する場所の始点
count	取得するスカラ値の個数
v	値を取得するベクトル

出力

value	取得したスカラ値を格納する配列
ierr	リターンコード

注釈

マルチプロセス環境では, 全体ベクトルの第 $start+i$ 行を指定する.

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)
```

機能

ベクトル v の第 i 行の値 $(i = 0, 1, ..., global\_n - 1)$ を $value[i]$ から取得する.

入力

value	取得するスカラ値を格納する配列

出力

v	値を取得したベクトル
ierr	リターンコード

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)
```

機能

ベクトル v の第 i 行の値 $(i = 0, 1, ..., global\_n - 1)$ を $value[i]$ に格納する.

入力

v	値を取得するベクトル

出力

value	取得したスカラ値を格納する配列
ierr	リターンコード

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)
```

機能

ベクトル v が使用可能かどうかを調べる.

入力

v	ベクトル

出力

ierr	LIS_TRUE 使用可能
	LIS_FALSE 使用不可

6.2 行列の操作

行列 A の次数を $global\_n \times global\_n$ とする．行列 A を $nprocs$ 個のプロセスで行ブロック分割する場合の各部分行列の行数を $local\_n$ とする．$global\_n$ をグローバルな行数, $local\_n$ をローカルな行数と呼ぶ.

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)
```

機能

行列を作成する.

入力

LIS_Comm	MPI コミュニケータ

出力

A	行列
ierr	リターンコード

注釈

逐次, マルチスレッド環境では, comm の値は無視される.

6.2.2 lis_matrix_destroy

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

機能

不要になった行列をメモリから破棄する.

入力

A	メモリから破棄する行列

出力

ierr	リターンコード

注釈

関数 lis_matrix_destroy により, 行列 A に関連付けられた配列も破棄される.

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)
```

機能

既存の行列と同じ情報を持つ行列を作成する.

入力

Ain	複製元の行列

出力

Aout	複製先の行列
ierr	リターンコード

注釈

関数 `lis_matrix_duplicate` は行列の要素の値は複製せず, メモリ領域のみ確保する. 値も複製する場合は, この関数の後に関数 `lis_matrix_copy` を呼び出す.

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)
```

機能

行列のメモリ領域を確保する.

入力

A	行列
nnz_row	平均非零要素数
nnz	各行の非零要素数の配列

出力

ierr	リターンコード

注釈

nnz_row または *nnz* のどちらか一方を指定する. この関数は, `lis_matrix_set_value` で効率よく要素に値を代入できるように, あらかじめメモリ領域を確保する.

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)
```

機能

行列 A の第 i 行第 j 列に値を代入する.

入力

flag	LIS_INS_VALUE 挿入 : $A[i,j] = value$ LIS_ADD_VALUE 加算代入: $A[i,j] = A[i,j] + value$
i	行列の行番号
j	行列の列番号
value	代入するスカラ値
A	行列

出力

A	第 i 行第 j 列に値が代入された行列
ierr	リターンコード

注釈

マルチプロセス環境では, 全体行列の第 i 行第 j 列を指定する.

関数 `lis_matrix_set_value` は代入された値を一時的な内部形式で格納するため, `lis_matrix_set_value` を用いた後には関数 `lis_matrix_assemble` を呼び出さなければならない.

大規模な行列に対しては, 関数 `lis_matrix_set_type` の導入を検討すべきである. 詳細については `lis-($VERSION)/test/test2.c` 及び `lis-($VERSION)/test/test2f.F90` を参照のこと.

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)
```

機能

行列をライブラリで使用可能にする.

入力

A	行列

出力

A	使用可能になった行列
ierr	リターンコード

6.2.7 lis_matrix_set_size

```
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)
```

機能

行列の次数を設定する.

入力

A	行列
local_n	行列 A のローカルな行数
global_n	行列 A のグローバルな行数

出力

ierr	リターンコード

注釈

$local\_n$ か $global\_n$ のどちらか一方を与えなければならない.

逐次, マルチスレッド環境では, $local\_n$ は $global\_n$ に等しい. したがって, `lis_matrix_set_size(A,n,0)` と `lis_matrix_set_size(A,0,n)` は, いずれも次数 $n \times n$ の行列を作成する.

マルチプロセス環境においては, `lis_matrix_set_size(A,n,0)` は各プロセス上に次数 $n \times N$ の部分行列を作成する. N は n の総和である.
一方, `lis_matrix_set_size(A,0,n)` は各プロセス p 上に次数 $m_p \times n$ の部分行列を作成する. m_p はライブラリ側で決定される.

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)
```

機能

行列の次数を取得する.

入力

A	行列

出力

local_n	行列 A のローカルな行数
global_n	行列 A のグローバルな行数
ierr	リターンコード

注釈

逐次, マルチスレッド環境では, $local\_n$ は $global\_n$ に等しい.

6.2.9 lis_matrix_get_range

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

機能

部分行列 A が全体行列のどこに位置するのかを調べる.

入力

A	部分行列

出力

is	部分行列 A の全体行列中での開始位置
ie	部分行列 A の全体行列中での終了位置+1
ierr	リターンコード

注釈

逐次, マルチスレッド環境では, 行列の次数が $n \times n$ ならば, C 版では $is = 0$, $ie = n$, Fortran 版では $is = 1$, $ie = n + 1$ である.

6.2.10 lis_matrix_get_nnz

```
C       LIS_INT lis_matrix_get_nnz(LIS_MATRIX A, LIS_INT *nnz)
Fortran subroutine lis_matrix_get_nnz(LIS_MATRIX A, LIS_INTEGER nnz,
         LIS_INTEGER ierr)
```

機能

行列 A の非零要素数を取得する.

入力

A	行列

出力

nnz	行列 A の非零要素数
ierr	リターンコード

注釈

マルチプロセス環境では, 部分行列 A の非零要素数を取得する.

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_INT matrix_type,
         LIS_INTEGER ierr)
```

機能

行列の格納形式を設定する.

入力

A	行列
matrix_type	行列の格納形式

出力

ierr	リターンコード

注釈

行列作成時の A の matrix_type は LIS_MATRIX_CSR である. 以下に matrix_type に指定可能な格納形式を示す.

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

6.2.12 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)
```

機能

行列の格納形式を取得する.

入力

A	行列

出力

matrix_type	行列の格納形式
ierr	リターンコード

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

機能

CSR 形式の配列を行列 A に関連付ける.

入力

`nnz`	非零要素数
`ptr, index, value`	CSR 形式の配列
`A`	行列

出力

`A`	関連付けられた行列

注釈

`lis_matrix_set_csr` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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

機能

CSC 形式の配列を行列 A に関連付ける.

入力

`nnz`	非零要素数
`ptr, index, value`	CSC 形式の配列
`A`	行列

出力

`A`	関連付けられた行列

注釈

`lis_matrix_set_csc` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

6.2.15 lis_matrix_set_msr

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

機能

MSR 形式の配列を行列 A に関連付ける.

入力

nnz	非零要素数
ndz	対角部分の零要素数
index, value	MSR 形式の配列
A	行列

出力

A	関連付けられた行列

注釈

lis_matrix_set_msr を用いた後には, lis_matrix_assemble を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

6.2.16 lis_matrix_set_dia

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

機能

DIA 形式の配列を行列 A に関連付ける.

入力

`nnd`	非零な対角要素の本数
`index, value`	DIA 形式の配列
`A`	行列

出力

`A`	関連付けられた行列

注釈

`lis_matrix_set_dia` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

ELL 形式の配列を行列 A に関連付ける.

入力

`maxnzr`	各行の非零要素数の最大値
`index, value`	ELL 形式の配列
`A`	行列

出力

`A`	関連付けられた行列

注釈

`lis_matrix_set_ell` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

JAD 形式の配列を行列 A に関連付ける.

入力

nnz	非零要素数
maxnzr	各行の非零要素数の最大値
perm, ptr, index, value	JAD 形式の配列
A	行列

出力

A	関連付けられた行列

注釈

`lis_matrix_set_jad` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

BSR 形式の配列を行列 A に関連付ける.

入力

bnr	行ブロックサイズ
bnc	列ブロックサイズ
bnnz	非零ブロック数
bptr, bindex, value	BSR 形式の配列
A	行列

出力

A	関連付けられた行列

注釈

`lis_matrix_set_bsr` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

BSC 形式の配列を行列 A に関連付ける.

入力

`bnr`	行ブロックサイズ
`bnc`	列ブロックサイズ
`bnnz`	非零ブロック数
`bptr`, `bindex`, `value`	BSC 形式の配列
`A`	行列

出力

`A`	関連付けられた行列

注釈

`lis_matrix_set_bsc` を用いた後には, `lis_matrix_assemble` を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

VBR 形式の配列を行列 A に関連付ける.

入力

nnz	非零ブロックの全要素数
nr	行ブロック数
nc	列ブロック数
bnnz	非零ブロック数
row, col, ptr, bptr, bindex, value	VBR 形式の配列
A	行列

出力

A	関連付けられた行列

注釈

lis_matrix_set_vbr を用いた後には, lis_matrix_assemble を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

COO 形式の配列を行列 A に関連付ける.

入力

nnz	非零要素数
row, col, value	COO 形式の配列
A	行列

出力

A	関連付けられた行列

注釈

lis_matrix_set_coo を用いた後には, lis_matrix_assemble を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

DNS 形式の配列を行列 A に関連付ける.

入力

value	DNS 形式の配列
A	行列

出力

A	関連付けられた行列

注釈

lis_matrix_set_dns を用いた後には, lis_matrix_assemble を呼び出さなければならない. また, Fortran 版の配列データは 0 を起点としなければならない.

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)
```

機能

配列の行列 A への関連付けを解除する.

入力

`A`	関連付けられた行列

出力

`A`	関連付けを解除された行列

注釈

`lis_matrix_unset` を用いた後には, `lis_matrix_destroy` を呼び出さなければならない.

6.3 ベクトルと行列を用いた計算

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)
```

機能

ベクトル x, y の要素を交換する.

入力

x, y	ベクトル

出力

x, y	交換後のベクトル
ierr	リターンコード

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)
```

機能

ベクトル x の要素をベクトル y に複製する.

入力

x	複製元のベクトル

出力

y	複製先のベクトル
ierr	リターンコード

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)
```

機能

ベクトル和 $y = \alpha x + y$ を計算する.

入力

alpha	スカラ値
x, y	ベクトル

出力

y	$\alpha x + y$ (ベクトル y の値は上書きされる)
ierr	リターンコード

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)
```

機能

ベクトル和 $y = x + \alpha y$ を計算する.

入力

alpha	スカラ値
x, y	ベクトル

出力

y	$x + \alpha y$ (ベクトル y の値は上書きされる)
ierr	リターンコード

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)
```

機能

ベクトル和 $z = \alpha x + y$ を計算する.

入力

alpha	スカラ値
x, y	ベクトル

出力

z	$\alpha x + y$
ierr	リターンコード

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)
```

機能

ベクトル x の要素を α 倍する.

入力

alpha	スカラ値
x	ベクトル

出力

x	αx (ベクトル x の値は上書きされる)
ierr	リターンコード

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)
```

機能

ベクトル x の要素にベクトル y の対応する要素を掛ける.

入力

x, y	ベクトル

出力

z	計算結果を格納したベクトル
ierr	リターンコード

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)
```

機能

ベクトル x の要素をベクトル y の対応する要素で割る.

入力

x, y	ベクトル

出力

z	計算結果を格納したベクトル
ierr	リターンコード

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)
```

機能

ベクトル x の要素にスカラ値を代入する．

入力

value	代入するスカラ値
x	ベクトル

出力

x	各要素にスカラ値が代入されたベクトル
ierr	リターンコード

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)
```

機能

ベクトル x の要素の絶対値を求める．

入力

x	ベクトル

出力

x	各要素の絶対値を格納したベクトル
ierr	リターンコード

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)
```

機能

ベクトル x の要素の逆数を求める.

入力

x	ベクトル

出力

x	各要素の逆数を格納したベクトル
ierr	リターンコード

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)
```

機能

ベクトル x の要素の共役複素数を求める.

入力

x	ベクトル

出力

x	各要素の共役複素数を格納したベクトル
ierr	リターンコード

6.3.13 lis_vector_shift

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

機能

ベクトル x の要素をシフトさせる.

入力

alpha	シフト量
x	ベクトル

出力

x	各要素のシフト後の値を格納したベクトル
ierr	リターンコード

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)
```

機能

エルミート内積 $y^H x$ を計算する.

入力

x, y	ベクトル

出力

value	エルミート内積
ierr	リターンコード

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)
```

機能

非エルミート内積 $y^T x$ を計算する.

入力

x, y	ベクトル

出力

value	非エルミート内積
ierr	リターンコード

6.3.16 lis_vector_nrm1

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

機能

ベクトル x の 1 ノルムを計算する.

入力

x	ベクトル

出力

value	ベクトルの 1 ノルム
ierr	リターンコード

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)
```

機能

ベクトル x の 2 ノルムを計算する.

入力

x	ベクトル

出力

value	ベクトルの 2 ノルム
ierr	リターンコード

6.3.18 lis_vector_nrmi

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

機能

ベクトル x の無限大ノルムを計算する.

入力

x	ベクトル

出力

value	ベクトルの無限大ノルム
ierr	リターンコード

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)
```

機能

ベクトル x の要素の和を計算する.

入力

x	ベクトル

出力

value	ベクトルの要素の和
ierr	リターンコード

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)
```

機能

BSR, BSC, VBR 形式のブロックサイズ, 分割情報を設定する.

入力

A	行列
bnr	BSR(BSC) 形式の行ブロックサイズ, または VBR 形式の行ブロック数
bnc	BSR(BSC) 形式の列ブロックサイズ, または VBR 形式の列ブロック数
row	VBR 形式の行分割情報の配列
col	VBR 形式の列分割情報の配列

出力

ierr	リターンコード

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)
```

機能

行列 A_{in} を指定した格納形式に変換し，行列 A_{out} に格納する．

入力

Ain	変換元の行列

出力

Aout	指定の格納形式に変換された行列
ierr	リターンコード

注釈

変換先の格納形式の指定は `lis_matrix_set_type` を用いて A_{out} に対して行う．BSR, BSC, VBR 形式のブロックサイズ等の情報は，`lis_matrix_set_blocksize` を用いて A_{out} に対して設定する．

指定された格納形式への変換のうち，以下の表において丸印を付したものは，直接変換される．それ以外のものは，記載される形式を経由した後，指定の格納形式に変換される．表に記載されないものは，CSR 形式を経由した後，指定の格納形式に変換される．

元＼先	CSR	CSC	MSR	DIA	ELL	JAD	BSR	BSC	VBR	COO	DNS
CSR		○	○	○	○	○	○	CSC	○	○	○
COO	○	○	○	CSR	CSR	CSR	CSR	CSC	CSR		CSR

6.3.22 lis_matrix_copy

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

機能

行列 A_{in} の要素を行列 A_{out} に複製する.

入力

Ain	複製元の行列

出力

Aout	複製先の行列
ierr	リターンコード

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)
```

機能

行列和 $B = \alpha A + B$ を計算する.

入力

alpha	スカラ値
A, B	行列

出力

B	$\alpha A + B$ (行列 B の値は上書きされる)
ierr	リターンコード

注釈

行列 A, B は DNS 形式でなければならない.

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)
```

機能

行列和 $B = A + \alpha B$ を計算する.

入力

alpha	スカラ値
A, B	行列

出力

B	$A + \alpha B$ (行列 B の値は上書きされる)
ierr	リターンコード

注釈

行列 A, B は DNS 形式でなければならない.

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)
```

機能

行列和 $C = \alpha A + B$ を計算する.

入力

alpha	スカラ値
A, B	行列

出力

C	$\alpha A + B$
ierr	リターンコード

注釈

行列 A, B, C は DNS 形式でなければならない.

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)
```

機能

行列 A 及びベクトル b のスケーリングを行う.

入力

A	行列
b	ベクトル
action	LIS_SCALE_JACOBI Jacobi スケーリング $D^{-1}Ax = D^{-1}b$ (D は $A = (a_{ij})$ の対角部分) LIS_SCALE_SYMM_DIAG 対角スケーリング $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ ($D^{-1/2}$ は対角要素の値が $1/\sqrt{a_{ii}}$ である対角行列)

出力

A	スケーリングによって得られる行列
b	スケーリングによって得られるベクトル
d	D^{-1} または $D^{-1/2}$ の対角部分を格納したベクトル
ierr	リターンコード

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)
```

機能

行列 A の対角部分をベクトル d に複製する.

入力

A	行列

出力

d	対角要素が格納されたベクトル
ierr	リターンコード

6.3.28 lis_matrix_shift_diagonal

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

機能

行列 A の対角部分をシフトさせる.

入力

alpha	シフト量
A	行列

出力

A	シフト後の行列
ierr	リターンコード

6.3.29 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)
```

機能

行列ベクトル積 $y = Ax$ を計算する.

入力

A	行列
x	ベクトル

出力

y	Ax
ierr	リターンコード

6.3.30 lis_matvect

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

機能

行列ベクトル積 $y = A^T x$ を計算する.

入力

A	行列
x	ベクトル

出力

y	$A^T x$
ierr	リターンコード

6.4 線型方程式の求解

6.4.1 lis_solver_create

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

機能

ソルバ (線型方程式解法の情報を格納する構造体) を作成する.

入力

なし

出力

solver	ソルバ
ierr	リターンコード

注釈

ソルバは線型方程式解法の情報を持つ.

6.4.2 lis_solver_destroy

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

機能

不要になったソルバをメモリから破棄する.

入力

solver	メモリから破棄するソルバ

出力

ierr	リターンコード

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)
```

機能

前処理行列を作成する.

入力

なし

出力

precon	前処理行列
ierr	リターンコード

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)
```

機能

不要になった前処理行列をメモリから破棄する.

入力

precon	メモリから破棄する前処理行列

出力

ierr	リターンコード

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)
```

機能

線型方程式解法のオプションをソルバに設定する.

入力

`text`	コマンドラインオプション

出力

`solver`	ソルバ
`ierr`	リターンコード

注釈

以下に指定可能なコマンドラインオプションを示す. `-i {cg|1}`は`-i cg`または`-i 1`を意味する. `-maxiter [1000]`は, `-maxiter`の既定値が1000であることを意味する.

線型方程式解法に関するオプション (既定値: `-i bicg`)

線型方程式解法	オプション	補助オプション	
CG	`-i {cg|1}`		
BiCG	`-i {bicg|2}`		
CGS	`-i {cgs|3}`		
BiCGSTAB	`-i {bicgstab|4}`		
BiCGSTAB(l)	`-i {bicgstabl|5}`	`-ell [2]`	次数 l
GPBiCG	`-i {gpbicg|6}`		
TFQMR	`-i {tfqmr|7}`		
Orthomin(m)	`-i {orthomin|8}`	`-restart [40]`	リスタート値 m
GMRES(m)	`-i {gmres|9}`	`-restart [40]`	リスタート値 m
Jacobi	`-i {jacobi|10}`		
Gauss-Seidel	`-i {gs|11}`		
SOR	`-i {sor|12}`	`-omega [1.9]`	緩和係数 ω $(0 < \omega < 2)$
BiCGSafe	`-i {bicgsafe|13}`		
CR	`-i {cr|14}`		
BiCR	`-i {bicr|15}`		
CRS	`-i {crs|16}`		
BiCRSTAB	`-i {bicrstab|17}`		
GPBiCR	`-i {gpbicr|18}`		
BiCRSafe	`-i {bicrsafe|19}`		
FGMRES(m)	`-i {fgmres|20}`	`-restart [40]`	リスタート値 m
IDR(s)	`-i {idrs|21}`	`-irestart [2]`	リスタート値 s
IDR(1)	`-i {idr1|22}`		
MINRES	`-i {minres|23}`		
COCG	`-i {cocg|24}`		
COCR	`-i {cocr|25}`		

前処理に関するオプション (既定値: `-p none`)

前処理	オプション	補助オプション	
なし	-p {none|0}		
Jacobi	-p {jacobi|1}		
ILU(k)	-p {ilu|2}	-ilu_fill [0]	フィルインレベル k
SSOR	-p {ssor|3}	-ssor_w [1.0]	緩和係数 ω $(0 < \omega < 2)$
Hybrid	-p {hybrid|4}	-hybrid_i [sor]	線型方程式解法
		-hybrid_maxiter [25]	最大反復回数
		-hybrid_tol [1.0e-3]	収束判定基準
		-hybrid_w [1.5]	SOR の緩和係数 ω $(0 < \omega < 2)$
		-hybrid_ell [2]	BiCGSTAB(l) の次数 l
		-hybrid_restart [40]	GMRES(m), Orthomin(m) の リスタート値 m
I+S	-p {is|5}	-is_alpha [1.0]	$I + \alpha S^{(m)}$ のパラメータ α
		-is_m [3]	$I + \alpha S^{(m)}$ のパラメータ m
SAINV	-p {sainv|6}	-sainv_drop [0.05]	ドロップ基準
SA-AMG	-p {saamg|7}	-saamg_unsym [false]	非対称版の選択 (行列構造は対称とする)
		-saamg_theta [0.05|0.12]	ドロップ基準 $a_{ij}^2 \leq \theta^2 \lvert a_{ii} \rvert \lvert a_{jj} \rvert$ (対称|非対称)
Crout ILU	-p {iluc|8}	-iluc_drop [0.05]	ドロップ基準
		-iluc_rate [5.0]	最大フィルイン数の倍率
ILUT	-p {ilut|9}		
Additive Schwarz	-adds true	-adds_iter [1]	反復回数

その他のオプション

オプション		
-maxiter [1000]	最大反復回数	
-tol [1.0e-12]	収束判定基準 tol	
-tol_w [1.0]	収束判定基準 tol_w	
-print [0]	残差履歴の出力	
	-print {none|0}	残差履歴を出力しない
	-print {mem|1}	残差履歴をメモリに保存する
	-print {out|2}	残差履歴を標準出力に書き出す
	-print {all|3}	残差履歴をメモリに保存し, 標準出力に書き出す
-scale [0]	スケーリングの選択. 結果は元の行列, ベクトルに上書きされる	
	-scale {none|0}	スケーリングなし
	-scale {jacobi|1}	Jacobi スケーリング $D^{-1}Ax = D^{-1}b$ (D は $A = (a_{ij})$ の対角部分)
	-scale {symm_diag|2}	対角スケーリング $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ ($D^{-1/2}$ は対角要素の値が $1/\sqrt{a_{ii}}$ である対角行列)
-initx_zeros [1]	初期ベクトル x_0	
	-initx_zeros {false|0}	与えられた値を使用
	-initx_zeros {true|1}	すべての要素の値を 0 にする
-conv_cond [0]	収束条件	
	-conv_cond {nrm2_r|0}	$\|\|b - Ax\|\|_2 \leq tol * \|\|b - Ax_0\|\|_2$
	-conv_cond {nrm2_b|1}	$\|\|b - Ax\|\|_2 \leq tol * \|\|b\|\|_2$
	-conv_cond {nrm1_b|2}	$\|\|b - Ax\|\|_1 \leq tol_w * \|\|b\|\|_1 + tol$
-omp_num_threads [t]	実行スレッド数 t は最大スレッド数	
-storage [0]	行列格納形式	
-storage_block [2]	BSR, BSC のブロックサイズ	
-f [0]	線型方程式解法の精度	
	-f {double|0}	倍精度
	-f {quad|1}	4 倍精度

6.4.6 lis_solver_set_optionC

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

機能

ユーザプログラム実行時にコマンドラインで指定された線型方程式解法のオプションをソルバに設定する.

入力

なし

出力

solver	ソルバ
ierr	リターンコード

6.4.7 lis_solve

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

機能

指定された解法で線型方程式 $Ax = b$ を解く.

入力

A	係数行列
b	右辺ベクトル
x	初期ベクトル
solver	ソルバ

出力

x	解
solver	反復回数, 経過時間等の情報
ierr	リターンコード

6.4.8 lis_solve_kernel

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

機能

指定された解法について, 外部で定義された前処理を用いて線型方程式 $Ax = b$ を解く.

入力

A	係数行列
b	右辺ベクトル
x	初期ベクトル
solver	ソルバ
precon	前処理

出力

x	解
solver	反復回数, 経過時間等の情報
ierr	リターンコード

注釈

lis-($VERSION)/src/esolver/lis_esolver_ii.c などを参照のこと. このプログラムでは, lis_solve_kernel を複数回呼び出して最小固有値を計算する.

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)
```

機能

状態をソルバから取得する.

入力

solver	ソルバ

出力

status	状態
ierr	リターンコード

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)
```

機能

反復回数をソルバから取得する.

入力

solver	ソルバ

出力

iter	反復回数
ierr	リターンコード

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)
```

機能

反復回数に関する詳細情報をソルバから取得する.

入力

solver	ソルバ

出力

iter	総反復回数
iter_double	倍精度演算の反復回数
iter_quad	4 倍精度演算の反復回数
ierr	リターンコード

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)
```

機能

経過時間をソルバから取得する.

入力

solver	ソルバ

出力

time	経過時間
ierr	リターンコード

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)
```

機能

経過時間に関する詳細情報をソルバから取得する.

入力

`solver`	ソルバ

出力

`time`	itime と ptime の合計
`itime`	線型方程式解法の経過時間
`ptime`	前処理の経過時間
`p_c_time`	前処理行列作成の経過時間
`p_i_time`	線型方程式解法中の前処理の経過時間
`ierr`	リターンコード

6.4.14 lis_solver_get_residualnorm

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

機能

相対残差ノルム $||b - Ax||_2/||b||_2$ をソルバから取得する.

入力

`solver`	ソルバ

出力

`residual`	相対残差ノルム
`ierr`	リターンコード

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)
```

機能

残差履歴をソルバから取得する.

入力

solver	ソルバ

出力

v	残差履歴が収められたベクトル
ierr	リターンコード

注釈

ベクトル v はあらかじめ関数 `lis_vector_create` で作成しておかなければならない. ベクトル v の次数 n が残差履歴の長さよりも小さい場合は残差履歴の最初から n 個までを取得する.

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)
```

機能

線型方程式解法の番号をソルバから取得する.

入力

solver	ソルバ

出力

nsol	線型方程式解法の番号
ierr	リターンコード

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)
```

機能

前処理の番号をソルバから取得する.

入力

solver	ソルバ

出力

precon_type	前処理の番号
ierr	リターンコード

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)
```

機能

線型方程式解法の番号から解法名を取得する.

入力

nsol	線型方程式解法の番号

出力

name	線型方程式解法名
ierr	リターンコード

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)
```

機能

前処理の番号から前処理名を取得する.

入力

precon_type	前処理の番号

出力

name	前処理名
ierr	リターンコード

6.5 固有値問題の求解

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)
```

機能

ソルバ (固有値解法の情報を格納する構造体) を作成する.

入力

なし

出力

esolver	ソルバ
ierr	リターンコード

注釈

ソルバは固有値解法の情報を持つ.

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)
```

機能

不要になったソルバをメモリから破棄する.

入力

esolver	メモリから破棄するソルバ

出力

ierr	リターンコード

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)
```

機能

固有値解法のオプションをソルバに設定する.

入力

`text`	コマンドラインオプション

出力

`esolver`	ソルバ
`ierr`	リターンコード

注釈

以下に指定可能なコマンドラインオプションを示す. `-e {pi|1}`は`-e pi`または`-e 1`を意味する. `-emaxiter [1000]`は`-emaxiter`の既定値が 1000 であることを意味する.

固有値解法に関するオプション (既定値: `-e pi`)

固有値解法	オプション	補助オプション	
Power	`-e {pi|1}`		
Inverse	`-e {ii|2}`	`-i [bicg]`	線型方程式解法
Approximate Inverse	`-e {aii|3}`	`-i [bicg]`	線型方程式解法
Rayleigh Quotient	`-e {rqi|4}`	`-i [bicg]`	線型方程式解法
CG	`-e {cg|5}`	`-i [cg]`	線型方程式解法
CR	`-e {cr|6}`	`-i [bicg]`	線型方程式解法
Jacobi-Davidson	`-e {jd|7}`	`-i [cg]`	線型方程式解法
Subspace	`-e {si|8}`	`-ss [1]`	部分空間の大きさ
Lanczos	`-e {li|9}`	`-ss [1]`	部分空間の大きさ
Arnoldi	`-e {ai|10}`	`-ss [1]`	部分空間の大きさ

前処理に関するオプション (既定値: `-p ilu`)

前処理	オプション	補助オプション	
なし	-p {none|0}		
Jacobi	-p {jacobi|1}		
ILU(k)	-p {ilu|2}	-ilu_fill [0]	フィルインレベル k
SSOR	-p {ssor|3}	-ssor_w [1.0]	緩和係数 ω $(0 < \omega < 2)$
Hybrid	-p {hybrid|4}	-hybrid_i [sor]	線型方程式解法
		-hybrid_maxiter [25]	最大反復回数
		-hybrid_tol [1.0e-3]	収束判定基準
		-hybrid_w [1.5]	SOR の緩和係数 ω $(0 < \omega < 2)$
		-hybrid_ell [2]	BiCGSTAB(l) の次数 l
		-hybrid_restart [40]	GMRES(m), Orthomin(m) の リスタート値 m
I+S	-p {is|5}	-is_alpha [1.0]	$I + \alpha S^{(m)}$ のパラメータ α
		-is_m [3]	$I + \alpha S^{(m)}$ のパラメータ m
SAINV	-p {sainv|6}	-sainv_drop [0.05]	ドロップ基準
SA-AMG	-p {saamg|7}	-saamg_unsym [false]	非対称版の選択 (行列構造は対称とする)
		-saamg_theta [0.05|0.12]	ドロップ基準 $a_{ij}^2 \le \theta^2 \|a_{ii}\|\|a_{jj}\|$ (対称|非対称)
Crout ILU	-p {iluc|8}	-iluc_drop [0.05]	ドロップ基準
		-iluc_rate [5.0]	最大フィルイン数の倍率
ILUT	-p {ilut|9}		
Additive Schwarz	-adds true	-adds_iter [1]	反復回数

その他のオプション

オプション		
-emaxiter [1000]	最大反復回数	
-etol [1.0e-12]	収束判定基準	
-eprint [0]	残差履歴の出力	
	-eprint {none|0}	残差履歴を出力しない
	-eprint {mem|1}	残差履歴をメモリに保存する
	-eprint {out|2}	残差履歴を標準出力に書き出す
	-eprint {all|3}	残差履歴をメモリに保存し, 標準出力に書き出す
-ie [ii]	Subspace, Lanczos, Arnoldi 法の内部で使用する固有値解法の指定	
	-ie {pi|1}	Power (Subspace のみ)
	-ie {ii|2}	Inverse
	-ie {aii|3}	Approximate Inverse
	-ie {rqi|4}	Rayleigh Quotient
	-ie {cg|5}	CG (Lanczos のみ)
	-ie {cr|6}	CR (Lanczos, Arnoldi)
	-ie {jd|7}	Jacobi-Davidson (Lanczos のみ)
-shift [0.0]	固有値のシフト量	
-initx_ones [1]	初期ベクトル x_0	
	-initx_ones {false|0}	与えられた値を使用
	-initx_ones {true|1}	すべての要素の値を 1 にする
-omp_num_threads [t]	実行スレッド数	
	t は最大スレッド数	
-estorage [0]	行列格納形式	
-estorage_block [2]	BSR, BSC のブロックサイズ	
-ef [0]	固有値解法の精度	
	-ef {double|0}	倍精度
	-ef {quad|1}	4 倍精度

6.5.4 lis_esolver_set_optionC

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

機能

ユーザプログラム実行時にコマンドラインで指定された固有値解法のオプションをソルバに設定する.

入力

なし

出力

esolver	ソルバ
ierr	リターンコード

6.5.5 lis_esolve

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

機能

指定された解法で固有値問題 $Ax = \lambda x$ を解く.

入力

A	係数行列
x	初期ベクトル
esolver	ソルバ

出力

evalue	モード 0 の固有値
x	固有値に対応する固有ベクトル
esolver	反復回数, 経過時間等の情報
ierr	リターンコード

6.5.6 lis_esolver_get_status

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

機能

指定された固有対の状態をソルバから取得する.

入力

esolver	ソルバ

出力

status	状態
ierr	リターンコード

6.5.7 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)
```

機能

指定された固有対の反復回数をソルバから取得する.

入力

esolver	ソルバ

出力

iter	反復回数
ierr	リターンコード

6.5.8 lis_esolver_get_iterex

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

機能

指定された固有対の反復回数に関する詳細情報をソルバから取得する.

入力

esolver	ソルバ

出力

iter	総反復回数
iter_double	倍精度演算の反復回数
iter_quad	4 倍精度演算の反復回数
ierr	リターンコード

6.5.9 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)
```

機能

指定された固有対の経過時間をソルバから取得する.

入力

esolver	ソルバ

出力

time	経過時間
ierr	リターンコード

6.5.10 lis_esolver_get_timeex

```
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 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)
```

機能

指定された固有対の経過時間に関する詳細情報をソルバから取得する.

入力

esolver	ソルバ

出力

time	固有値解法の経過時間
itime	固有値解法中の線型方程式解法の経過時間
ptime	固有値解法中の線型方程式解法前処理の経過時間
p_c_time	前処理行列作成の経過時間
p_i_time	線型方程式解法中の前処理の経過時間
ierr	リターンコード

6.5.11 lis_esolver_get_residualnorm

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

機能

指定された固有対の相対残差ノルム $||\lambda x - Ax||_2/||\lambda x||_2$ をソルバから取得する.

入力

esolver	ソルバ

出力

residual	相対残差ノルム
ierr	リターンコード

6.5.12 lis_esolver_get_rhistory

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

機能

指定された固有対の残差履歴をソルバから取得する.

入力

esolver	ソルバ

出力

v	残差履歴が収められたベクトル
ierr	リターンコード

注釈

ベクトル v はあらかじめ関数 lis_vector_create で作成しておかなければならない. ベクトル v の次数 n が残差履歴の長さよりも小さい場合は残差履歴の最初から n 個までを取得する.

6.5.13 lis_esolver_get_evalues

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

機能

すべての固有値をソルバから取得する.

入力

esolver	ソルバ

出力

v	固有値が納められたベクトル
ierr	リターンコード

注釈

ベクトル v はあらかじめ関数 lis_vector_create で作成しておかなければならない.

6.5.14 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)
```

機能

すべての固有ベクトルをソルバから取得し, 行列 M に格納する.

入力

`esolver`	ソルバ

出力

`M`	COO 形式で固有ベクトルが納められた行列
`ierr`	リターンコード

注釈

行列 M はあらかじめ関数 `lis_matrix_create` で作成しておかなければならない.

6.5.15 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)
```

機能

すべての固有対の相対残差ノルム $||\lambda x - Ax||_2/||\lambda x||_2$ をソルバから取得する.

入力

`esolver`	ソルバ

出力

`v`	相対残差ノルムが納められたベクトル
`ierr`	リターンコード

注釈

ベクトル v はあらかじめ関数 `lis_vector_create` で作成しておかなければならない.

6.5.16 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)
```

機能

すべての固有対の反復回数をソルバから取得する.

入力

esolver	ソルバ

出力

iter	反復回数
ierr	リターンコード

6.5.17 lis_esolver_get_esolver

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

機能

固有値解法の番号をソルバから取得する.

入力

esolver	ソルバ

出力

nsol	固有値解法の番号
ierr	リターンコード

6.5.18 lis_esolver_get_esolvername

```
C       LIS_INT lis_esolver_get_esolvername(LIS_INT esolver, char *name)
Fortran subroutine lis_esolver_get_esolvername(LIS_INTEGER esolver, character name,
         LIS_INTEGER ierr)
```

機能

固有値解法の番号から解法名を取得する.

入力

`nesol`	固有値解法の番号

出力

`name`	固有値解法名
`ierr`	リターンコード

6.6 配列を用いた計算

以下はローカルな処理のための関数であり, 並列化されない. 配列データは 0 を起点とし, 列優先順序で格納される.

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)
```

機能

ベクトル x, y の要素を交換する.

入力

n	ベクトルの次数
x, y	次数 n のベクトル x, y を格納する配列

出力

x, y	交換後の配列
ierr	リターンコード

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)
```

機能

ベクトル x の要素をベクトル y に複製する.

入力

n	ベクトルの次数
x	複製元のベクトル x を格納する配列

出力

y	複製先のベクトル y が格納された配列
ierr	リターンコード

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)
```

機能

ベクトル和 $y = \alpha x + y$ を計算する.

入力

n	ベクトルの次数
alpha	スカラ値
x, y	ベクトル x, y を格納する配列

出力

y	$\alpha x + y$ (ベクトル y の値は上書きされる)
ierr	リターンコード

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)
```

機能

ベクトル和 $y = x + \alpha y$ を計算する.

入力

n	ベクトルの次数
alpha	スカラ値
x, y	ベクトル x, y を格納する配列

出力

y	$x + \alpha y$ (ベクトル y の値は上書きされる)
ierr	リターンコード

6.6.5 lis_array_axpyz

```
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)
```

機能

ベクトル和 $z = \alpha x + y$ を計算する.

入力

n	ベクトルの次数
alpha	スカラ値
x, y	ベクトル x, y を格納する配列

出力

z	$\alpha x + y$
ierr	リターンコード

6.6.6 lis_array_scale

```
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)
```

機能

ベクトル x の要素を α 倍する.

入力

n	ベクトルの次数
alpha	スカラ値
x	ベクトル x を格納する配列

出力

x	αx (ベクトル x の値は上書きされる)
ierr	リターンコード

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)
```

機能

ベクトル x の要素にベクトル y の対応する要素を掛ける.

入力

n	ベクトルの次数
x, y	ベクトル x, y を格納する配列

出力

z	計算結果が格納された配列
ierr	リターンコード

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)
```

機能

ベクトル x の要素をベクトル y の対応する要素で割る.

入力

n	ベクトルの次数
x, y	ベクトル x, y を格納する配列

出力

z	計算結果が格納された配列
ierr	リターンコード

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)
```

機能

ベクトル x の要素にスカラ値を代入する.

入力

n	ベクトルの次数
value	代入するスカラ値
x	ベクトル x を格納する配列

出力

x	各要素にスカラ値が代入された配列
ierr	リターンコード

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)
```

機能

ベクトル x の要素の絶対値を求める.

入力

n	ベクトルの次数
x	ベクトル x を格納する配列

出力

x	各要素の絶対値が格納された配列
ierr	リターンコード

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)
```

機能

ベクトル x の要素の逆数を求める.

入力

n	ベクトルの次数
x	ベクトル x を格納する配列

出力

x	各要素の逆数が格納された配列
ierr	リターンコード

6.6.12 lis_array_shift

```
C       LIS_INT lis_array_shift(LIS_INT n, LIS_SCALAR alpha, LIS_SCALAR x[])
Fortran subroutine lis_array_shift(LIS_INTEGER n, LIS_SCALAR alpha, LIS_SCALAR x(),
         LIS_INTEGER ierr)
```

機能

ベクトル x の要素をシフトさせる.

入力

n	ベクトルの次数
alpha	シフト量
x	ベクトル x を格納する配列

出力

x	各要素のシフト後の値が格納された配列
ierr	リターンコード

6.6.13 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)
```

機能

ベクトル内積 $x^T y$ を計算する.

入力

n	ベクトルの次数
x, y	ベクトル x, y を格納する配列

出力

value	ベクトル内積
ierr	リターンコード

6.6.14 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)
```

機能

ベクトル x の 1 ノルムを計算する.

入力

n	ベクトルの次数
x	ベクトル x を格納する配列

出力

value	ベクトルの 1 ノルム
ierr	リターンコード

6.6.15 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)
```

機能

ベクトル x の 2 ノルムを計算する.

入力

n	ベクトルの次数
x	ベクトル x を格納する配列

出力

value	ベクトルの 2 ノルム
ierr	リターンコード

6.6.16 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)
```

機能

ベクトル x の無限大ノルムを計算する.

入力

n	ベクトルの次数
x	ベクトル x を格納する配列

出力

value	ベクトルの無限大ノルム
ierr	リターンコード

6.6.17 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)
```

機能

ベクトル x の要素の和を計算する.

入力

n	ベクトルの次数
x	ベクトル x を格納する配列

出力

value	ベクトルの要素の和
ierr	リターンコード

6.6.18 lis_array_matvec

```
C       LIS_INT lis_array_matvec(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR x[],
         LIS_SCALAR y[], LIS_INT op)
Fortran subroutine lis_array_matvec(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR x(),
         LIS_SCALAR y(), LIS_INTEGER op, LIS_INTEGER ierr)
```

機能

行列ベクトル積 Ax を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列
x	次数 n のベクトル x を格納する配列
y	次数 n のベクトル y を格納する配列
op	LIS_INS_VALUE 挿入 : $y = Ax$
	LIS_SUB_VALUE 減算代入: $y = y - Ax$

出力

y	y
ierr	リターンコード

6.6.19 lis_array_matvect

```
C       LIS_INT lis_array_matvect(LIS_INT n, LIS_SCALAR a[], LIS_SCALAR x[],
         LIS_SCALAR y[], LIS_INT op)
Fortran subroutine lis_array_matvect(LIS_INTEGER n, LIS_SCALAR a(), LIS_SCALAR x(),
         LIS_SCALAR y(), LIS_INTEGER op, LIS_INTEGER ierr)
```

機能

行列ベクトル積 $A^T x$ を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列
x	次数 n のベクトル x を格納する配列
y	次数 n のベクトル y を格納する配列
op	LIS_INS_VALUE 挿入 : $y = A^T x$
	LIS_SUB_VALUE 減算代入: $y = y - A^T x$

出力

y	y
ierr	リターンコード

6.6.20 lis_array_matvec_ns

```
C       LIS_INT 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)
```

機能

行列 A が正方でない場合に行列ベクトル積 Ax を計算する.

入力

m, n	行列の次数
a	次数 $m \times n$ の行列 A を格納する配列
lda	配列 A の主要次元の次数
x	次数 n のベクトル x を格納する配列
y	次数 m のベクトル y を格納する配列
op	LIS_INS_VALUE 挿入 : $y = Ax$
	LIS_SUB_VALUE 減算代入: $y = y - Ax$

出力

y	y
ierr	リターンコード

6.6.21 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)
```

機能

行列積 AB を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列
b	次数 $n \times n$ の行列 B を格納する配列
c	次数 $n \times n$ の行列 C を格納する配列
op	LIS_INS_VALUE 挿入 : $C = AB$
	LIS_SUB_VALUE 減算代入: $C = C - AB$

出力

c	C
ierr	リターンコード

6.6.22 lis_array_matmat_ns

```
C       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(), LIS_INTEGER lda, LIS_SCALAR b(), LIS_INTEGER ldb,
         LIS_SCALAR c(), LIS_INTEGER ldc, LIS_INTEGER op, LIS_INTEGER ierr)
```

機能

行列 A, B が正方でない場合に積 AB を計算する.

入力

`l, m, n`	行列の次数
`a`	次数 $l \times m$ の行列 A を格納する配列
`lda`	行列 A の主要次元の次数
`b`	次数 $m \times n$ の行列 B を格納する配列
`ldb`	行列 B の主要次元の次数
`c`	次数 $l \times n$ の行列 C を格納する配列
`ldc`	行列 C の主要次元の次数
`op`	`LIS_INS_VALUE` 挿入 : $C = AB$
	`LIS_SUB_VALUE` 減算代入: $C = C - AB$

出力

`c`	C
`ierr`	リターンコード

6.6.23 lis_array_ge

```
C       LIS_INT lis_array_ge(LIS_INT n, LIS_SCALAR a[]
Fortran subroutine lis_array_solve(LIS_INTEGER n, LIS_SCALAR a(), LIS_INTEGER ierr)
```

機能

Gauss の消去法を用いて行列 A の逆を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列

出力

a	A^{-1}
ierr	リターンコード

6.6.24 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)
```

機能

直接法を用いて線型方程式 $Ax = b$ を解く.

入力

n	行列の次数
a	次数 $n \times n$ の係数行列 A を格納する配列
b	次数 n の右辺ベクトル b を格納する配列
w	要素数 $n \times n$ の作業配列

出力

x	解 x
ierr	リターンコード

6.6.25 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)
```

機能

古典 Gram-Schmidt 法を用いて QR 分解 $QR = A$ を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列

出力

q	次数 $n \times n$ の直交行列 Q が格納された配列
r	次数 $n \times n$ の上三角行列 R が格納された配列
ierr	リターンコード

6.6.26 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)
```

機能

修正 Gram-Schmidt 法を用いて QR 分解 $QR = A$ を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列

出力

q	次数 $n \times n$ の直交行列 Q が格納された配列
r	次数 $n \times n$ の上三角行列 R が格納された配列
ierr	リターンコード

6.6.27 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)
```

機能

QR 法を用いて行列 A の固有値を計算する.

入力

n	行列の次数
a	次数 $n \times n$ の行列 A を格納する配列
q	要素数 $n \times n$ の作業配列 Q
r	要素数 $n \times n$ の作業配列 R

出力

a	固有値をブロック対角要素に持つ相似変換後のブロック上三角行列 A が格納された配列
qriter	QR 法の反復回数
qrerr	相似変換後の第 1 劣対角要素 $A(2,1)$ の 2 ノルム
ierr	リターンコード

6.7 ファイルの操作

6.7.1 lis_input

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

機能

外部ファイルから行列, ベクトルデータを読み込む.

入力

filename	ファイル名

出力

A	指定された格納形式の行列
b	右辺ベクトル
x	解ベクトル
ierr	リターンコード

注釈

対応するファイル形式は以下の通りである.

- 拡張 Matrix Market 形式 (ベクトルデータに対応)
- Harwell-Boeing 形式

これらのデータ構造については付録 A を参照せよ.

6.7.2 lis_input_vector

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

機能

外部ファイルからベクトルデータを読み込む.

入力

filename	ファイル名

出力

v	ベクトル
ierr	リターンコード

注釈

対応するファイル形式は

- PLAIN 形式
- 拡張 Matrix Market 形式 (ベクトルデータに対応)

これらのデータ構造については付録 A を参照せよ.

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)
```

機能

外部ファイルから行列データを読み込む.

入力

filename	ファイル名

出力

A	指定された格納形式の行列
ierr	リターンコード

注釈

対応するファイル形式は以下の通りである.

- Matrix Market 形式
- Harwell-Boeing 形式

これらのデータ構造については付録 A を参照せよ.

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)
```

機能

行列, ベクトルデータを外部ファイルに書き込む.

入力

A	行列	
b	右辺ベクトル (ファイルに書き込まない場合は NULL を代入する)	
x	解ベクトル (ファイルに書き込まない場合は NULL を代入する)	
format	ファイル形式	
	LIS_FMT_MM	Matrix Market 形式
filename	ファイル名	

出力

ierr	リターンコード

注釈

ファイル形式のデータ構造は付録 A を参照せよ.

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)
```

機能

ベクトルデータを外部ファイルに書き込む.

入力

v	ベクトル		
format	ファイル形式		
		LIS_FMT_PLAIN	PLAIN 形式
		LIS_FMT_MM	Matrix Market 形式
filename	ファイル名		

出力

ierr	リターンコード

注釈

ファイル形式のデータ構造は付録 A を参照せよ.

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)
```

機能

行列データを外部ファイルに書き込む.

入力

A	行列	
format	ファイル形式	
	LIS_FMT_MM	Matrix Market 形式
filename	ファイル名	

出力

ierr	リターンコード

6.8 その他

6.8.1 lis_initialize

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

機能

MPI の初期化, コマンドライン引数の取得等の初期化処理を行う.

入力

argc	コマンドライン引数の数
argv	コマンドライン引数

出力

ierr	リターンコード

6.8.2 lis_finalize

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

機能

終了処理を行う.

入力

なし

出力

ierr	リターンコード

6.8.3 lis_wtime

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

機能

経過時間を計測する.

入力

なし

出力

ある時点からの経過時間を double 型の値 (単位は秒) として返す.

注釈

処理時間を測定する場合は, 処理の開始時と終了時の時間を `lis_wtime` により測定し, その差を求める.

6.8.4 CHKERR

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

機能

関数が正常に終了したかどうかを判定する.

入力

ierr	リターンコード

出力

なし

注釈

正常に終了していない場合は，`lis_finalize` を実行した後，プログラムを強制終了する.

参考文献

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] H. A. van der Vorst and J. B. M. Melissen. A Petrov-Galerkin Type Method for Solving $Ax = b$, where A is Symmetric Complex. IEEE Transactioins on Magnetics, Vol. 26, No. 2, pp. 706–708, 1990.

[25] T. Sogabe and S. Zhang. A COCR Method for Solving Complex Symmetric Linear Systems. Journal of Computational and Applied Mathematics, Vol. 199, No. 2, pp. 297–303, 2007.

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

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

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

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

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

[31] G. L. G. Sleijpen and H. A. van der Vorst. A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems. SIAM Journal on Matrix Analysis and Applications, Vol. 17, No. 2, pp. 401–425, 1996.

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

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

[34] W. E. Arnoldi. The Principle of Minimized Iterations in the Solution of the Matrix Eigenvalue Problems. Quarterly of Applied Mathematics, Vol. 9, No. 17, pp. 17–29, 1951.

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

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

[37] K. Nakajima, H. Nakamura, and T. Tanahashi. Parallel Iterative Solvers with Localized ILU Preconditioning. Lecture Notes in Computer Science 1225, pp. 342–350, 1997.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[59] D. M. Young. Iterative Solution of Large Linear Systems. Academic Press, 1971.

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

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

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

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

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

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

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

[67] J. H. Wilkinson and C. Reinsch. Handbook for Automatic Computation, Vol. 2: Linear Algebra. Grundlehren Der Mathematischen Wissenschaften, Vol. 186, Springer, 1971.

[68] B. T. Smith, J. M. Boyle, Y. Ikebe, V. C. Klema, and C. B. Moler. Matrix Eigensystem Routines: EISPACK Guide, 2nd ed. Lecture Notes in Computer Science 6, Springer, 1970.

[69] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. Matrix Eigensystem Routines: EISPACK Guide Extension. Lecture Notes in Computer Science 51, Springer, 1972.

[70] J. J. Dongarra, J. R. Bunch, G. B. Moler, and G. M. Stewart. LINPACK Users' Guide. SIAM, 1979.

[71] J. R. Rice and R. F. Boisvert. Solving Elliptic Problems Using ELLPACK. Springer, 1985.

[72] E. Anderson, et al. LAPACK Users' Guide. 3rd ed. SIAM, 1987.

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

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

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

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

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

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

[79] L. S. Blackford, et al. ScaLAPACK Users' Guide. SIAM, 1997.

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

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

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

[83] K. Garatani, H. Nakamura, H. Okuda, and G. Yagawa. GeoFEM: High Performance Parallel FEM for Solid Earth. Lecture Notes in Computer Science 1593, pp. 133–140, Springer, 1999.

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

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

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

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

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

[89] J. Dongarra and M. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

A　ファイル形式

本節では, 本ライブラリで使用できるファイル形式について述べる. なお Harwell-Boeing 形式では, 行列が対称である場合にも上三角及び下三角要素の双方を格納する必要がある.

A.1　拡張 Matrix Market 形式

Matrix Market 形式はベクトルデータの格納に対応していないため, 拡張 Matrix Market 形式では行列とベクトルを合わせて格納できるよう仕様を拡張する. $M \times N$ の行列 $A = (a_{ij})$ の非零要素数を L とする. $a_{ij} = A(I, J)$ とする. ファイル形式を以下に示す.

```
%%MatrixMarket matrix coordinate real general  <-- ヘッダ
%                                              <-+
%                                                | 0 行以上のコメント行
%                                              <-+
M N L B X                                      <-- 行数 列数 非零数 (0 or 1) (0 or 1)
I1 J1 A(I1,J1)                                 <-+
I2 J2 A(I2,J2)                                   | 行番号 列番号 値
 . . .                                           | 配列起点は 1
IL JL A(IL,JL)                                 <-+
I1 B(I1)                                       <-+
I2 B(I2)                                         | B=1 の場合のみ存在する
 . . .                                           | 行番号 値
IM B(IM)                                       <-+
I1 X(I1)                                       <-+
I2 X(I2)                                         | X=1 の場合のみ存在する
 . . .                                           | 行番号 値
IM X(IM)                                       <-+
```

(A.1) 式の行列 A とベクトル b に対するファイル形式を以下に示す.

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

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

A.2 Harwell-Boeing 形式

Harwell-Boeing 形式では, CSC 形式で行列を格納する. value を行列 A の非零要素の値, index を非零要素の行番号, ptr を value と index の各列の開始位置を格納する配列とする. ファイル形式を以下に示す.

```
第 1 行 (A72,A8)
   1 - 72 Title
  73 - 80 Key
第 2 行 (5I14)
   1 - 14 ヘッダを除く総行数
  15 - 28 ptr の行数
  29 - 42 index の行数
  43 - 56 value の行数
  57 - 70 右辺の行数
第 3 行 (A3,11X,4I14)
   1 -  3 行列の種類
            第 1 列: R Real matrix
                   C Complex matrix
                   P Pattern only (非対応)
            第 2 列: S Symmetric (非対応)
                   U Unsymmetric
                   H Hermitian (非対応)
                   Z Skew symmetric (非対応)
                   R Rectangular (非対応)
            第 3 列: A Assembled
                   E Elemental matrices (非対応)
   4 - 14 空白
  15 - 28 行数
  29 - 42 列数
  43 - 56 非零要素数
  57 - 70 0
第 4 行 (2A16,2A20)
   1 - 16 ptr の形式
  17 - 32 index の形式
  33 - 52 value の形式
  53 - 72 右辺の形式
第 5 行 (A3,11X,2I14) 右辺が存在する場合
   1      右辺の種類
            F フルベクトル
            M 行列と同じ形式 (非対応)
   2      初期値が与えられるならば G
   3      解が与えられるならば X
   4 - 14 空白
  15 - 28 右辺の数
  29 - 42 非零要素数
```

(A.1) 式の行列 A とベクトル b に対するファイル形式を以下に示す.

```
1--------10--------20--------30--------40--------50--------60--------70--------80
Harwell-Boeing format sample                                            Lis
            8             1             1             4             2
RUA                       4             4            10             4
(11i7)          (13i6)          (3e26.18)           (3e26.18)
F                         1             0
      1      3      6      9
     1     2     1     2     3     2     3     4     3     4
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
```

```
  2.00000000000000000000E+00  1.00000000000000000000E+00  1.00000000000000000000E+00
  2.00000000000000000000E+00
  0.00000000000000000000E+00  1.00000000000000000000E+00  2.00000000000000000000E+00
  3.00000000000000000000E+00
```

A.3 ベクトル用拡張 Matrix Market 形式

ベクトル用拡張 Matrix Market 形式では, ベクトルデータを格納できるよう Matrix Market 形式の仕様を拡張する. 次数 N のベクトル $b = (b_i)$ に対して $b_i = B(I)$ とする. ファイル形式を以下に示す.

```
%%MatrixMarket vector coordinate real general  <-- ヘッダ
%                                              <-+
%                                                | 0 行以上のコメント行
%                                              <-+
N                                              <-- 行数
I1 B(I1)                                       <-+
I2 B(I2)                                         | 行番号 値
 . . .                                           | 配列起点は 1
IN B(IN)                                       <-+
```

(A.1) 式のベクトル b に対するファイル形式を以下に示す.

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

A.4 ベクトル用 PLAIN 形式

ベクトル用 PLAIN 形式は, ベクトルの値を第 1 要素から順に書き出したものである. 次数 N のベクトル $b = (b_i)$ に対して $b_i = B(I)$ とする. ファイル形式を以下に示す.

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

(A.1) 式のベクトル b に対するファイル形式を以下に示す.

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