

Lis 1.2.3 ユーザーマニュアル

Copyright (C) 2002-2008 The Scalable Software Infrastructure Project,
supported by "Development of Software Infrastructure for Large Scale Scientific Simula-
tion" Team, CREST, JST.
<http://www.ssisc.org/>

最終更新日 : 2008 年 12 月 4 日

目 次

0	Lis 1.1 からの追加・変更点	1
0.1	追加	1
0.2	変更	1
1	はじめに	2
2	インストール	2
2.1	必要なシステム	2
2.2	ファイルの展開	2
2.3	configure スクリプトの実行	3
2.4	make の実行	3
2.5	テスト	6
2.6	インストール	7
2.7	テストプログラム	7
2.7.1	test1	7
2.7.2	test2	8
2.7.3	test3	8
2.7.4	test4	8
2.7.5	etest1	8
2.7.6	etest2	9
2.7.7	etest3	9
2.7.8	etest4	9
2.8	制限事項	10
3	基本操作	11
3.1	初期化・終了処理	12
3.2	ベクトル	12
3.3	行列	15
3.4	線型方程式の求解	22
3.5	固有値問題の求解	26
3.6	サンプルプログラム	29
3.7	コンパイル・リンク	32
3.8	実行	34
4	4 倍精度演算	35
4.1	4 倍精度演算の実行	35
5	行列格納形式	37
5.1	Compressed Row Storage (CRS)	37
5.1.1	行列の作り方 (逐次、OpenMP)	37
5.1.2	行列の作り方 (MPI)	38
5.1.3	関連する関数	38

5.2	Compressed Column Storage (CCS)	39
5.2.1	行列の作り方 (逐次、OpenMP)	39
5.2.2	行列の作り方 (MPI)	40
5.2.3	関連する関数	40
5.3	Modified Compressed Sparse Row (MSR)	41
5.3.1	行列の作り方 (逐次、OpenMP)	41
5.3.2	行列の作り方 (MPI)	42
5.3.3	関連する関数	42
5.4	Diagonal (DIA)	43
5.4.1	行列の作り方 (逐次)	43
5.4.2	行列の作り方 (OpenMP)	44
5.4.3	行列の作り方 (MPI)	45
5.4.4	関連する関数	45
5.5	Ellpack-Itpack generalized diagonal (ELL)	46
5.5.1	行列の作り方 (逐次、OpenMP)	46
5.5.2	行列の作り方 (MPI)	47
5.5.3	関連する関数	47
5.6	Jagged Diagonal (JDS)	48
5.6.1	行列の作り方 (逐次)	49
5.6.2	行列の作り方 (OpenMP)	50
5.6.3	行列の作り方 (MPI)	51
5.6.4	関連する関数	51
5.7	Block Sparse Row (BSR)	52
5.7.1	行列の作り方 (逐次、OpenMP)	52
5.7.2	行列の作り方 (MPI)	53
5.7.3	関連する関数	53
5.8	Block Sparse Column (BSC)	54
5.8.1	行列の作り方 (逐次、OpenMP)	54
5.8.2	行列の作り方 (MPI)	55
5.8.3	関連する関数	55
5.9	Variable Block Row (VBR)	56
5.9.1	行列の作り方 (逐次、OpenMP)	56
5.9.2	行列の作り方 (MPI)	57
5.9.3	関連する関数	58
5.10	Coordinate (COO)	59
5.10.1	行列の作り方 (逐次、OpenMP)	59
5.10.2	行列の作り方 (MPI)	60
5.10.3	関連する関数	60
5.11	Dense (DNS)	61
5.11.1	行列の作り方 (逐次、OpenMP)	61
5.11.2	行列の作り方 (MPI)	62
5.11.3	関連する関数	62

6	Functions	63
6.1	ベクトル操作	63
6.1.1	lis_vector_create	63
6.1.2	lis_vector_destroy	64
6.1.3	lis_vector_duplicate	64
6.1.4	lis_vector_set_size	65
6.1.5	lis_vector_get_size	66
6.1.6	lis_vector_get_range	66
6.1.7	lis_vector_set_value	67
6.1.8	lis_vector_get_value	67
6.1.9	lis_vector_copy	68
6.1.10	lis_vector_set_all	68
6.1.11	lis_vector_is_null	69
6.2	行列操作	70
6.2.1	lis_matrix_create	70
6.2.2	lis_matrix_destroy	70
6.2.3	lis_matrix_duplicate	71
6.2.4	lis_matrix_malloc	71
6.2.5	lis_matrix_set_value	72
6.2.6	lis_matrix_assemble	72
6.2.7	lis_matrix_set_size	73
6.2.8	lis_matrix_get_size	74
6.2.9	lis_matrix_get_range	74
6.2.10	lis_matrix_set_type	75
6.2.11	lis_matrix_get_type	76
6.2.12	lis_matrix_set_blocksize	76
6.2.13	lis_matrix_convert	77
6.2.14	lis_matrix_copy	77
6.2.15	lis_matrix_get_diagonal	78
6.2.16	lis_matrix_set_crs	79
6.2.17	lis_matrix_set_ccs	79
6.2.18	lis_matrix_set_msr	80
6.2.19	lis_matrix_set_dia	80
6.2.20	lis_matrix_set_ell	81
6.2.21	lis_matrix_set_jds	81
6.2.22	lis_matrix_set_bsr	82
6.2.23	lis_matrix_set_bsc	83
6.2.24	lis_matrix_set_vbr	84
6.2.25	lis_matrix_set_coo	85
6.2.26	lis_matrix_set_dns	85
6.3	ベクトルと行列の計算	86
6.3.1	lis_vector_scale	86

6.3.2	lis_vector_dot	86
6.3.3	lis_vector_nrm2	87
6.3.4	lis_vector_nrm1	87
6.3.5	lis_vector_axpy	88
6.3.6	lis_vector_xpay	88
6.3.7	lis_vector_axpyz	89
6.3.8	lis_matrix_scaling	89
6.3.9	lis_matvec	90
6.3.10	lis_matvect	90
6.4	線型方程式の求解	91
6.4.1	lis_solver_create	91
6.4.2	lis_solver_destroy	91
6.4.3	lis_solver_set_option	92
6.4.4	lis_solver_set_optionC	95
6.4.5	lis_solve	96
6.4.6	lis_solver_get_iters	97
6.4.7	lis_solver_get_itersex	97
6.4.8	lis_solver_get_time	98
6.4.9	lis_solver_get_timeex	98
6.4.10	lis_solver_get_residualnorm	99
6.4.11	lis_solver_get_rhistory	99
6.4.12	lis_solver_get_solver	100
6.4.13	lis_solver_get_solvername	101
6.5	固有値問題の求解	102
6.5.1	lis_esolver_create	102
6.5.2	lis_esolver_destroy	102
6.5.3	lis_esolver_set_option	103
6.5.4	lis_esolver_set_optionC	105
6.5.5	lis_solve	106
6.5.6	lis_esolver_get_iters	107
6.5.7	lis_esolver_get_itersex	107
6.5.8	lis_esolver_get_time	108
6.5.9	lis_esolver_get_timeex	108
6.5.10	lis_esolver_get_residualnorm	109
6.5.11	lis_esolver_get_rhistory	109
6.5.12	lis_esolver_get_evalues	110
6.5.13	lis_esolver_get_evectors	110
6.5.14	lis_esolver_get_esolver	111
6.5.15	lis_esolver_get_esolvername	111
6.6	ファイル入出力	112
6.6.1	lis_input_matrix	112
6.6.2	lis_input_vector	113

6.6.3	lis_input	114
6.6.4	lis_output_vector	115
6.6.5	lis_output	116
6.6.6	lis_solver_output_rhistory	117
6.6.7	lis_esolver_output_rhistory	117
6.7	その他	118
6.7.1	lis_initialize	118
6.7.2	lis_finalize	118
6.7.3	lis_wtime	119
6.7.4	CHKERR	119
参考文献		120
A ファイルフォーマット		122
A.1	拡張 MatrixMarket フォーマット	122
A.2	Harwell-Boeing フォーマット	123
A.3	拡張 MatrixMarket フォーマット (ベクトル用)	124
A.4	PLAIN フォーマット (ベクトル用)	124

0 Lis 1.1 からの追加・変更点

0.1 追加

Lis 1.2 では実行列を対象とする固有値解法が追加されました。

0.2 変更

Lis 1.2 では以下の点が変更されました。

1. ユーザインタフェースの仕様を一部変更

- (a) `lis_output_residual_history()`、`lis_get_residual_history()` をそれぞれ `lis_solver_output_rhistory()`、`lis_solver_get_rhistory()` に変更

1 はじめに

Lis (a Library of Iterative Solvers for linear systems) は大規模疎行列を係数とする線型方程式

$$Ax = b$$

及び固有値問題

$$Ax = \lambda x$$

を解くための反復法ライブラリである。C と Fortran で記述されており、逐次版、OpenMP を使用する共有メモリ並列版、MPI 単独、または MPI、OpenMP を組み合わせて使用する分散メモリ並列版がある。

Lis は以下の特徴を持つ：

- 22 種の線型方程式解法、6 種の固有値解法、10 種の前処理に対応
- 11 種の疎行列格納形式に対応
- 4 倍精度演算に対応
- 逐次、並列ともに共通のインタフェースで処理できる
- 逐次環境から並列環境への移行に際して、プログラムを変更する必要がある（あるいはごくわずかの変更でよい）

Lis では大規模な実行列を係数行列に持つ線型方程式及び固有値問題向けの反復解法が利用可能であり、線型方程式解法については表 1 に示す 22 種、固有値解法については表 2 に示す 6 種、前処理についてはスケーリングの他、表 3 に示す 10 種を実装している。またデータ格納形式については表 4 に示す 11 種に対応している。

2 インストール

本節では、Lis のインストール、テストの手順について述べる。なおここでは Linux クラスタ環境を想定している。

2.1 必要なシステム

Lis のインストールには 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 コンパイラも必要となる。並列計算環境では、OpenMP または MPI-1 を使用する。表 5 の計算環境において動作が確認されている。

2.2 ファイルの展開

次のコマンドを入力して、ファイルを展開する。(\$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) 下に図 1 に示すサブディレクトリが作成される。

表 1: 線型方程式解法

CG	CR
BiCG	BiCR[2]
CGS	CRS[3]
BiCGSTAB	BiCRSTAB[3]
GPBiCG	GPBiCR[3]
BiCGSafe[1]	BiCRSafe[4]
BiCGSTAB(1)	TFQMR
Jacobi	Orthomin(m)
Gauss-Seidel	GMRES(m)
SOR	FGMRES(m)[5]
IDR(s)[13]	MINRES[14]

表 2: 固有値解法

Power Iteration
Inverse Iteration
Approximate Inverse Iteration
Conjugate Gradient[15, 16]
Lanczos Iteration
Subspace Iteration

表 3: 前処理

Jacobi
SSOR
ILU(k)
ILUT[6, 7]
Crout ILU[8, 7]
I+S[9]
SA-AMG[10]
hybrid[11]
SAINV[12]
additive Schwarz
ユーザ定義

表 4: 格納形式

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

2.3 configure スクリプトの実行

次のコマンドを入力して、スクリプトを実行する。

- デフォルトの設定を利用する場合: `>./configure`
- インストール先を指定する場合 : `>./configure --prefix=<install-dir>`

configure に指定できるオプションを表 6 に示す。表 7 に TARGET で指定できる主な計算機環境を示す。

2.4 make の実行

`lis-($VERSION)` ディレクトリにおいて

```
>make
```

と make を実行すればコンパイル可能である。

表 5: 動作確認環境

C コンパイラ (必須)	OS
Intel C/C++ Compiler 7.0, 8.0, 9.0, 9.1, 10.1	Linux
IBM XL C/C++ V7.0	AIX 5.3 Linux
Sun WorkShop 6 update 2, Sun ONE Studio 7 Sun Studio 11	Solaris 9 Solaris 10
GCC 3.3, 4.1	Linux
Fortran コンパイラ (オプション)	OS
Intel Fortran Compiler 8.1, 9.0, 9.1, 10.1	Linux
IBM XL Fortran V9.1	AIX 5.3 Linux
Sun WorkShop 6 update 2, Sun ONE Studio 7 Sun Studio 11	Solaris 9 Solaris 10
g77 3.3 gfortran 4.3 g95 0.91	Linux

```

lis-($VERSION)
  config
  |   make 用のファイル
  include
  |   インクルードファイル
  src
  |   ソースファイル
  test
  |   テストプログラム

```

図 1: lis-(\$VERSION).tar.gz のファイル構成

表 6: configure オプション

--enable-omp	OpenMP を利用
--enable-mpi	MPI を利用
--enable-fortran	Fortran API を利用
--enable-saamg	SA-AMG 前処理を利用
--enable-quad	4 倍精度演算を利用
--prefix=<install-dir>	インストール先を指定
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
FC=<fortran_compiler>	Fortran コンパイラを指定
FCFLAGS=<fc_flags>	Fortran 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: 主な TARGET オプション

<target>	実行される configure スクリプト
cray_xt3	./configure CC=cc FC=ftn CFLAGS="-O3 -B -fastsse -tp k8-64" FCFLAGS="-O3 -fastsse -tp k8-64 -Mpreprocess" FCLDFLAGS="-Mnomain" ac_cv_sizeof_void_p=8 cross_compiling=yes --enable-mpi ax_f77_mangling="lower case, no underscore, extra underscore"
fujitsu_pq	./configure CC=fcc FC=frt ac_cv_sizeof_void_p=8 CFLAGS="-O3 -Kfast" FFLAGS="-O3 -Kfast -Cpp FCFLAGS="-O3 -Kfast -Cpp -Am" --enable-omp --enable-mpi ax_f77_mangling="lower case, underscore, no extra underscore"
hitachi	./configure CC=cc FC=f90 FCLDFLAGS="-lf90s" ac_cv_sizeof_void_p=8 --enable-omp --enable-mpi ax_f77_mangling="lower case, underscore, no extra underscore"
ibm_bg	./configure CC=blrts_xlc FC=blrts_xlf90 CFLAGS="-O3 -qarch=440d -qtune=440 -qstrict -I/bgl/BlueLight/ppcfloor/bglsys/include" FFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F -qfixed=72 -w -I/bgl/BlueLight/ppcfloor/bglsys/include" FCFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90 -w -I/bgl/BlueLight/ppcfloor/bglsys/include" ac_cv_sizeof_void_p=4 cross_compiling=yes --enable-mpi ax_f77_mangling="lower case, no underscore, no extra underscore"
nec_es	./configure CC=esmpic++ FC=esmpif90 AR=esar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes --enable-mpi --enable-omp ax_f77_mangling="lower case, no underscore, extra underscore"
nec_sx9_cross	./configure CC=sxmpic++ FC=sxmpif90 AR=sxar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes --enable-mpi --enable-omp ax_f77_mangling="lower case, no underscore, extra underscore"

2.5 テスト

ここでは、make (ライブラリのコンパイル) が正常に行われたかを確認する。lis-(\$VERSION) ディレクトリにおいて

```
>make check
```

とすると lis-(\$VERSION)/test ディレクトリに作成された実行ファイルを用いてテストを実行する。このテストは、MatrixMarket ファイル lis-(\$VERSION)/test/testmat.mtx から行列、ベクトルデータを読み込み、線型方程式 $Ax = b$ を解いた近似解を lis-(\$VERSION)/test/sol.txt にその収束履歴を lis-(\$VERSION)/test/res.txt に書き出す。近似解がすべて 1 ならば正しい結果である。SGI Altix 3700 上での実行結果を以下に示す。表示されている下 4 行は実行環境により値が異なる。

デフォルト

```
100 x 100 matrix  460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER     : BiCG 2
PRECON     : None
STORAGE    : CRS
lis_solve  : normal end

BiCG: iter      = 15 iter_double = 15 iter_quad = 0
BiCG: times     = 5.178690e-03
BiCG: p_times   = 1.277685e-03 (p_c = 1.254797e-03 p_i = 2.288818e-05 )
BiCG: i_times   = 3.901005e-03
BiCG: Residual  = 6.327297e-15
```

--enable-omp

```
Max Procs   = 32
Max Threads = 2
100 x 100 matrix  460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER     : BiCG 2
PRECON     : None
STORAGE    : CRS
lis_solve  : normal end

BiCG: iter      = 15 iter_double = 15 iter_quad = 0
BiCG: times     = 8.960009e-03
BiCG: p_times   = 2.297878e-03 (p_c = 2.072096e-03 p_i = 2.257824e-04 )
BiCG: i_times   = 6.662130e-03
BiCG: Residual  = 6.221213e-15
```

```

--enable-mpi
100 x 100 matrix  460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER      : BiCG 2
PRECON      : None
STORAGE     : CRS
lis_solve   : normal end

BiCG: iter      = 15 iter_double = 15 iter_quad = 0
BiCG: times     = 2.911400e-03
BiCG: p_times   = 1.560780e-04 (p_c = 1.459997e-04 p_i = 1.007831e-05 )
BiCG: i_times   = 2.755322e-03
BiCG: Residual  = 6.221213e-15

```

2.6 インストール

lis-(\$VERSION) ディレクトリにおいて

```
>make install
```

とすると、以下のようにファイルがコピーされる。

```

$(INSTALLDIR)
include
|   lis.h lisf.h
lib
   liblis.a

```

lis.h は C で、lisf.h は Fortran でライブラリを利用するときに必要なヘッダーファイルで、liblis.a はライブラリファイルである。

2.7 テストプログラム

2.7.1 test1

lis-(\$VERSION)/test ディレクトリにおいて

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

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線型方程式 $Ax = b$ を options で指定された解法で解く。また、近似解を result_filename に収束履歴を residual_filename に書き出す。入力可能な行列データフォーマットは MatrixMarket フォーマットである。rhs_setting は

- | | |
|---|-------------------------------------|
| 0 | 行列データファイルに含まれている右辺ベクトルを用いる |
| 1 | $b = (1, \dots, 1)^T$ を用いる |
| 2 | $b = A \times (1, \dots, 1)^T$ を用いる |

rhs_filename

右辺ベクトルのファイル名

が指定できる。rhs_filename は PLAIN フォーマットまたは MatrixMarket フォーマットが利用できる。
test1f.f は test1.c の Fortran 版である。

2.7.2 test2

lis-(\$VERSION)/test ディレクトリにおいて

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

と入力すると、2次元ポアソン方程式を5点中心差分で離散化した行列を係数行列とする線型方程式 $Ax = b$ 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、近似解を result_filename に収束履歴を residual_filename に書き出す。ただし、線型方程式 $Ax = b$ の解ベクトルの値がすべて1となるように右辺ベクトル b を設定している。m と n はそれぞれ垂直方向と水平方向の格子点数である。

2.7.3 test3

線型方程式 $Ax = b$ を指定された解法で解き、近似解を表示する。行列 A は 12×12 の三重対角行列

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

である。右辺ベクトル b は近似解 x がすべて1となるように求めている。

2.7.4 test4

lis-(\$VERSION)/test ディレクトリにおいて

```
>test4 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き、近似解を表示する。行列 A は Toeplitz 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 は近似解 x がすべて1となるように求めている。n は行列 A の次数、gamma は γ である。

2.7.5 etest1

lis-(\$VERSION)/test ディレクトリにおいて

```
>etest1 matrix_filename solution_filename residual_filename [options]
```

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、固有値問題 $Ax = \lambda x$ を options で指定された解法で解き、近似固有値を表示する。また、近似固有ベクトルを result_filename に、収束履歴を residual_filename に書き出す。入力可能な行列データフォーマットは MatrixMarket フォーマットである。etest1f.f は etest1.c の Fortran 版である。

2.7.6 etest2

lis-(\$VERSION)/test ディレクトリにおいて

```
>etest2 m n matrix_type solution_filename residual_filename [options]
```

と入力すると、2次元ポアソン方程式を5点中心差分で離散化した行列を係数行列とする固有値問題 $Ax = \lambda x$ 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、近似固有値を表示する。また、近似固有ベクトルを result_filename に、収束履歴を residual_filename に書き出す。m と n はそれぞれ垂直方向と水平方向の格子点数である。

2.7.7 etest3

lis-(\$VERSION)/test ディレクトリにおいて

```
>etest3 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、近似固有値を表示する。行列 A は次数 n の三重対角行列

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

である。etest3f.f は etest3.c の Fortran 版である。

2.7.8 etest4

lis-(\$VERSION)/test ディレクトリにおいて

```
>etest4 matrix_filename tol [options]
```

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、条件数 options を収束判定基準 tol により計算する。

2.8 制限事項

現在のバージョンには以下のような制限がある。

- 前処理に対する制限
 - Jacobi、SSOR 以外の前処理が選択され、かつ行列 A が CRS 形式でない場合、前処理作成時に CRS 形式の行列 A を作成する
 - BiCG 法を選択した場合、SA-AMG 前処理は非対応
 - OpenMP 環境には SA-AMG 前処理は非対応、SAINV 前処理は前処理行列作成部分は逐次
- 4 倍精度演算に対する制限
 - 線型方程式解法の Jacobi、Gauss-Seidel、SOR、IDR(s) は非対応
 - 固有値解法は非対応
 - hybrid 前処理での内部反復解法の選択に関して、Jacobi、Gauss-Seidel、SOR は非対応
 - I+S、SA-AMG 前処理は非対応
- 行列格納形式に対する制限
 - MPI 環境では、ユーザー自身が目的の格納形式に必要な配列を用意する場合、CRS 形式のみ受付ける。目的の格納形式を利用するには `lis_matrix_convert` を使用して CRS 形式から目的の格納形式へ変換する。

3 基本操作

本節では、ライブラリの利用方法について述べる。線型方程式

$$Ax = b$$

または固有値問題

$$Ax = \lambda x$$

を解くためのプログラムを記述するには以下の処理が必要である。

- 初期化处理
- 行列の作成
- ベクトルの作成
- 線型方程式または固有値解法のためのソルバー（解法の情報を格納する構造体）の作成
- 行列、ベクトルに値を代入
- ソルバーに解法を設定
- 求解
- 終了処理

また、プログラムの先頭に以下の `include` 文を記述しておかなければならない。

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

`lis.h` と `lisf.h` は `$(INSTALLDIR)/include` に存在する。

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、終了処理は最後に必ず実行しなければならない。

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

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

初期化処理

初期化処理を行うには関数

- C `lis_initialize(int* argc, char** argv[])`
- Fortran subroutine `lis_initialize(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。

終了処理

終了処理を行うには関数

- C `int lis_finalize()`
- Fortran subroutine `lis_finalize(integer ierr)`

を用いる。

3.2 ベクトル

ベクトル v の次数を `global_n` とする。ベクトル v を `nprocs` 台のプロセッサで行ブロック分割したとき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 台のプロセッサで行ブロック分割した場合、`global_n` と `local_n` はそれぞれ 4 と 2 となる。

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

(3.1) 式のベクトル v を作成する場合、逐次、OpenMP 版ではベクトル v そのものを、MPI 版では各プロセスにプロセス台数で行ブロック分割した部分ベクトルを作成することとなる。

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、MPI 版のプロセッサ台数は 2 とする。

C(逐次、OpenMP)

```
1: 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(MPI)

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

Fortran(逐次、OpenMP)

```
1: integer i,n
2: LIS_VECTOR v
3: n = 4
4: call lis_vector_create(0,v,ierr)
5: call lis_vector_set_size(v,0,n,ierr)
6:
7: do i=1,n
9:     call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr)
10: enddo
```

Fortran(MPI)

```
1: integer i,n,is,ie
2: LIS_VECTOR v
3: n = 4
4: call lis_vector_create(MPI_COMM_WORLD,v,ierr)
5: call lis_vector_set_size(v,0,n,ierr)
6: call lis_vector_get_range(v,is,ie,ierr)
7: do i=is,ie-1
8:     call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr);
9: enddo
```

変数宣言

2行目のように

```
LIS_VECTOR    v;
```

と宣言する。

ベクトルの作成

ベクトル v の作成は関数

- C `int lis_vector_create(LIS_Comm comm, LIS_VECTOR *vec)`
- Fortran subroutine `lis_vector_create(LIS_Comm comm, LIS_VECTOR vec, integer ierr)`

を用いる。comm には MPI コミュニケータを指定する。逐次、OpenMP 版では comm の値は無視される。

ベクトルサイズの設定

ベクトルサイズの設定は関数

- C `int lis_vector_set_size(LIS_VECTOR vec, int local_n, int global_n)`
- Fortran subroutine `lis_vector_set_size(LIS_VECTOR vec, integer local_n, integer global_n, 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、OpenMP 版では、ベクトルの次数は $\text{local_n} = \text{global_n}$ となる。したがって、`lis_vector_set_size(v,n,0)` と `lis_vector_set_size(v,0,n)` はどちらも次数 n のベクトルを作成することを意味する。

MPI 版では、`lis_vector_set_size(v,n,0)` とすると、各プロセッサ p 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)` とすると各プロセッサ p に次数 m_p の部分ベクトルを作成する。ただし、 m_p の値はライブラリ側で決定される。

要素の代入

ベクトル v の i 行目に要素を代入するには関数

- C `int lis_vector_set_value(int flag, int i, LIS_SCALAR value, LIS_VECTOR v)`
- Fortran subroutine `lis_vector_set_value(int flag, int i, LIS_SCALAR value, LIS_VECTOR v, integer ierr)`

を用いる。MPI 版では、部分ベクトルの i 行目ではなく全体ベクトルの i 行目を指定する。flag には

LIS_INS.VALUE 挿入 : $v(i) = \text{value}$

LIS_ADD.VALUE 加算代入 : $v(i) = v(i) + \text{value}$

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 `int lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR *vout)`
- Fortran subroutine `lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout, 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はコピーしない。要素もコピーしたい場合はこの関数の後に

- C `int lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst)`
- Fortran subroutine `lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst, integer ierr)`

を用いる。

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 `int lis_vector_destroy(LIS_VECTOR v)`
- Fortran subroutine `lis_vector_destroy(LIS_VECTOR vec, integer ierr)`

を用いる。

3.3 行列

係数行列 A の次数を `global_n` \times `global_n` とする。行列 A を `nprocs` 台のプロセッサで行ブロック分割したとき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 台のプロセッサで行ブロック分割した場合、`global_n` と `local_n` はそれぞれ 4 と 2 となる。

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

目的の格納形式の行列を作成するには次の 3 の方法がある:

1. ライブラリ側が目的の格納形式に必要な配列を作成する
2. ユーザが目的の格納形式に必要な配列を用意する (Fortran は未対応)
3. ファイルから行列データを読み込む

パターン 1: ライブラリ側が目的の格納形式に必要な配列を作成する場合

(3.2) 式の行列 A を CRS 形式で作成する場合、逐次、OpenMP 版では行列 A そのものを、MPI 版では各プロセッサにプロセッサ台数で行ブロック分割した部分行列を作成することとなる。

行列 A を CRS 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、MPI 版のプロセッサ台数は 2 とする。

C(逐次、OpenMP)

```

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

```

C(MPI)

```

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

```

Fortran(逐次、OpenMP)

```

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

```

Fortran(MPI)

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

変数宣言

2 行目のように

```
LIS_MATRIX   A;
```

と宣言する。

行列の作成

行列 A の作成は関数

- C `int lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)`
- Fortran subroutine `lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, integer ierr)`

を用いる。comm には MPI コミュニケータを指定する。逐次、OpenMP 版では、comm の値は無視される。

行列のサイズ設定

行列 A のサイズ設定は関数

- C `int lis_matrix_set_size(LIS_MATRIX A, int local_n, int global_n)`
- Fortran subroutine `lis_matrix_set_size(LIS_MATRIX A, integer local_n, integer global_n, 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、OpenMP 版では、行列のサイズは `local_n = global_n` となる。したがって、`lis_matrix_set_size(A,n,0)` と `lis_matrix_set_size(A,0,n)` はともに $n \times n$ のサイズを設定することを意味する。

MPI 版では、`lis_matrix_set_size(A,n,0)` とすると、各プロセッサ p で行列サイズが $n_p \times N$ となるように設定する。ここで、 N は各プロセッサの n_p の総和である。

一方、`lis_matrix_set_size(A,0,n)` とすると各プロセッサ p で行列サイズが $m_p \times n$ となるように設定する。ここで、 m_p は部分行列の行数でこの値はライブラリ側で決定される。

要素の代入

行列 A の i 行 j 列目に要素を代入するには関数

- C `int lis_matrix_set_value(int flag, int i, int j, LIS_SCALAR value, LIS_MATRIX A)`
- Fortran subroutine `lis_matrix_set_value(integer flag, integer i, integer j, LIS_SCALAR value, LIS_MATRIX A, integer ierr)`

を用いる。MPI 版では、部分行列の i 行 j 列目ではなく全体行列の i 行 j 列目を指定する。flag には

LIS_INS_VALUE 挿入 : $A(i,j) = \text{value}$

LIS_ADD_VALUE 加算代入 : $A(i,j) = A(i,j) + \text{value}$

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 `int lis_matrix_set_type(LIS_MATRIX A, int matrix_type)`
- Fortran subroutine `lis_matrix_set_type(LIS_MATRIX A, int matrix_type, integer ierr)`

を用いる。行列作成時に A の matrix_type は LIS_MATRIX_CRCS となっている。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 Row Storage	(CRS)	{LIS_MATRIX_CRCS 1}
Compressed Column Storage	(CCS)	{LIS_MATRIX_CCS 2}
Modified Compressed Sparse Row	(MSR)	{LIS_MATRIX_MSR 3}
Diagonal	(DIA)	{LIS_MATRIX_DIA 4}
Ellpack-Itpack generalized diagonal	(ELL)	{LIS_MATRIX_ELL 5}
Jagged Diagonal	(JDS)	{LIS_MATRIX_JDS 6}
Block Sparse Row	(BSR)	{LIS_MATRIX_BSR 7}
Block Sparse Column	(BSC)	{LIS_MATRIX_BSC 8}
Variable Block Row	(VBR)	{LIS_MATRIX_VBR 9}
Dense	(DNS)	{LIS_MATRIX_DNS 10}
Coordinate	(COO)	{LIS_MATRIX_COO 11}

行列の組み立て

行列の要素をすべて代入したら、必ず関数

- C `int lis_matrix_assemble(LIS_MATRIX A)`
- Fortran subroutine `lis_matrix_assemble(LIS_MATRIX A, 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 `int lis_matrix_destroy(LIS_MATRIX A)`
- Fortran subroutine `lis_matrix_destroy(LIS_MATRIX A, integer ierr)`

を用いる。

パターン 2: ユーザが目的の格納形式に必要な配列を用意する場合

(3.2) 式の行列 A を CRS 形式で作成する場合、逐次、OpenMP 版では行列 A そのものを、MPI 版では各プロセッサにプロセッサ台数で行ブロック分割した部分行列を作成することとなる。

行列 A を CRS 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、MPI 版のプロセッサ台数は 2 とする。

C(逐次、OpenMP)

```

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

```

C(MPI)

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

配列の関連付け

ユーザー自身が作成した CRS 形式に必要な配列をライブラリが扱えるように行列 A に関連付けるには関数

- C `int lis_matrix_set_crs(int nnz, int *row, int *index, LIS_SCALAR *value, LIS_MATRIX A)`
- Fortran 未対応

を用いる。そのほかの格納形式については第 5 節を参照せよ。

パターン 3: ファイルから行列、ベクトルデータを読み込む場合

(3.2) 式の行列 A を CRS 形式でファイルから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、OpenMP、MPI)

```
1: LIS_MATRIX   A;
3: lis_matrix_create(LIS_COMM_WORLD,&A);
6: lis_matrix_set_type(A,LIS_MATRIX_CRS);
7: lis_input_matrix(A,"matvec.mtx");
```

Fortran(逐次、OpenMP、MPI)

```
1: LIS_MATRIX   A
3: call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
6: call lis_matrix_set_type(A,LIS_MATRIX_CRS,ierr)
7: call lis_input_matrix(A,'matvec.mtx',ierr)
```

読み込むファイル `matvec.mtx` の中身は以下になっている。これは MatrixMarket フォーマットである。

```
%%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

```

(3.2) 式の行列 A を CRS 形式で、また (3.1) 式のベクトル b をファイルから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、OpenMP、MPI)

```

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

```

Fortran(逐次、OpenMP、MPI)

```

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

```

読み込むファイル `matvec.mtx` の中身は以下のようになっている。これは拡張 MatrixMarket フォーマットである (Appendix 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

```

ファイルからの読み込み

行列 A にファイルからデータを読み込むには関数

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

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

を用いる。filename には読み込むファイルのパスを指定する。読み込むことができるファイルフォーマットは以下のとおりである。ファイルフォーマットについては Appendix A を参照。

- MatrixMarket フォーマット
- Harwell-Boeing フォーマット

行列 A とベクトル b、x にファイルからデータを読み込むには関数

- C 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, integer ierr)

を用いる。filename には読み込むファイルのパスを指定する。読み込むことができるファイルフォーマットは以下のとおりである。ファイルフォーマットについては Appendix A を参照。

- MatrixMarket フォーマット (ベクトルデータも読み込めるように拡張)
- Harwell-Boeing フォーマット

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プログラムは以下のように記述する。

C(逐次、OpenMP、MPI)

```
1: LIS_MATRIX A;
2: LIS_VECTOR b,x;
3: LIS_SOLVER solver;
4:
5: /* 行列とベクトルの作成 */
6:
7: lis_solver_create(&solver);
8: lis_solver_set_option("-i bicg -p none",solver);
9: lis_solver_set_option("-tol 1.0e-12",solver);
10: lis_solve(A,b,x,solver);
```

Fortran(逐次、OpenMP、MPI)

```
1: LIS_MATRIX A
2: LIS_VECTOR b,x
3: LIS_SOLVER solver
4:
5: /* 行列とベクトルの作成 */
6:
7: call lis_solver_create(solver,ierr)
8: call lis_solver_set_option('-i bicg -p none',solver,ierr)
9: call lis_solver_set_option('-tol 1.0e-12',solver,ierr)
10: call lis_solve(A,b,x,solver,ierr)
```

ソルバーの作成

ソルバー（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 `int lis_solver_create(LIS_SOLVER *solver)`
- Fortran subroutine `lis_solver_create(LIS_SOLVER solver, 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ーに設定するには関数

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

または

- C `int lis_solver_set_optionC(LIS_SOLVER solver)`
- Fortran subroutine `lis_solver_set_optionC(LIS_SOLVER solver, integer ierr)`

を用いる。`lis_solver_set_optionC` はユーザー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ー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` のデフォルト値が 1000 であることを意味する。

線型方程式解法の指定 デフォルト: -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] l の値
GPBiCG	-i {gpbicg 6}	
TFQMR	-i {tfqmr 7}	
Orthomin(m)	-i {orthomin 8}	-restart [40] リスタート m の値
GMRES(m)	-i {gmres 9}	-restart [40] リスタート m の値
Jacobi	-i {jacobi 10}	
Gauss-Seidel	-i {gs 11}	
SOR	-i {sor 12}	-omega [1.9] 緩和係数 ω の値 ($0 < \omega < 2$)
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 m の値
IDR(s)	-i {idrs 21}	-restart [40] リスタート s の値
MINRES	-i {minres 22}	

前処理の指定 デフォルト: -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 法の緩和係数 ω ($0 < \omega < 2$) -hybrid_ell [2] BiCGSTAB(l) 法の l の値 -hybrid_restart [40] GMRES, Orthomin のリスタート値
I+S	-p {is 5}	-is_alpha [1.0] $I + \alpha S^{(m)}$ 型前処理のパラメータ α -is_m [3] $I + \alpha S^{(m)}$ 型前処理のパラメータ 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)
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}	-ilut_drop [0.05] ドロップ基準 -ilut_rate [5.0] 最大フィルイン数の倍率
additive Schwarz	-adds true	-adds_iter [1] 繰り返し回数

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
-print [0]	残差の画面表示 -print {none 0} 何もしない -print {mem 1} 収束履歴をメモリに保存する -print {out 2} 収束履歴を画面に表示する -print {all 3} 収束履歴をメモリに保存し画面に表示する
-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 $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ $D^{-1/2}$ は対角要素に $1/\sqrt{a_{ii}}$ を持つ対角行列
-initx_zeros [true]	初期ベクトル x_0 の振舞い -initx_zeros {false 0} 与えられた値を使用 -initx_zeros {true 1} すべての要素を 0 にする
-omp_num_threads [t]	実行スレッド数 t は最大スレッド数

演算精度の指定 デフォルト: -precision double

精度	オプション	補助オプション
倍精度	-precision {double 0}	
4 倍精度	-precision {quad 1}	

求解

線型方程式 $Ax = b$ を解くには、関数

- C `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, integer ierr)`

を用いる。

3.5 固有値問題の求解

固有値問題 $Ax = \lambda x$ を指定の解法で解くプログラムは以下のように記述する。

C(逐次、OpenMP、MPI)

```
1: LIS_MATRIX A;
2: LIS_VECTOR x;
3: LIS_REAL eval;
4: LIS_ESOLVER esolver;
5:
6: /* 行列とベクトルの作成 */
7:
8: lis_esolver_create(&esolver);
9: lis_esolver_set_option("-e ii -i bicg -p none",esolver);
10: lis_esolver_set_option("-etol 1.0e-12 -tol 1.0e-12",esolver);
11: lis_solve(A,x,eval,esolver);
```

Fortran(逐次、OpenMP、MPI)

```
1: LIS_MATRIX A
2: LIS_VECTOR x
3: LIS_REAL eval
4: LIS_ESOLVER esolver
5:
6: /* 行列とベクトルの作成 */
7:
8: call lis_esolver_create(esolver,ierr)
9: call lis_esolver_set_option('-e ii -i bicg -p none',esolver,ierr)
10: call lis_esolver_set_option('-etol 1.0e-12 -tol 1.0e-12',esolver,ierr)
11: call lis_solve(A,x,eval,esolver,ierr)
```

ソルバーの作成

ソルバー（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 `int lis_esolver_create(LIS_ESOLVER *esolver)`

- Fortran subroutine lis_esolver_create(LIS_ESOLVER esolver, integer ierr)

を用いる。

オプションの設定

固有値解法をソルバーに設定するには関数

- C `int lis_esolver_set_option(char *text, LIS_ESOLVER esolver)`
- Fortran subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver,
 integer ierr)

または

- C `int lis_esolver_set_optionC(LIS_ESOLVER esolver)`
- Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, integer ierr)

を用いる。lis_esolver_set_optionC はユーザー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ーに設定する関数である。

オプションの設定

固有値解法をソルバーに設定するには関数

- C `int lis_esolver_set_option(char *text, LIS_ESOLVER esolver)`
- Fortran subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver,
 integer ierr)

または

- C `int lis_esolver_set_optionC(LIS_ESOLVER esolver)`
- Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, integer ierr)

を用いる。lis_esolver_set_optionC はユーザー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ー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 [10000] は-emaxiter のデフォルト値が 10000 であることを意味する。

固有値解法の指定 デフォルト: -e pi

固有値解法	オプション	補助オプション	
Power Iteration	-e {pi 1}		
Inverse Iteration	-e {ii 2}	-i [bicg]	線型方程式解法
Approximate Inverse Iteration	-e {aii 3}		
Conjugate Gradient	-e {cg 4}		
Lanczos Iteration	-e {li 5}	-ss [10]	部分空間の大きさ
Subspace Iteration	-e {si 6}	-ss [10]	部分空間の大きさ

前処理の指定 デフォルト: -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 法の緩和係数 ω ($0 < \omega < 2$) -hybrid_ell [2] BiCGSTAB(l) 法の l の値 -hybrid_restart [40] GMRES, Orthomin のリスタート値
I+S	-p {is 5}	-is_alpha [1.0] $I + \alpha S^{(m)}$ 型前処理のパラメータ α -is_m [3] $I + \alpha S^{(m)}$ 型前処理のパラメータ 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)
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}	-ilut_drop [0.05] ドロップ基準 -ilut_rate [5.0] 最大フィルイン数の倍率
additive Schwarz	-adds true	-adds_iter [1] 繰り返し回数

その他のオプション

オプション	
-emaxiter [10000]	最大反復回数
-etol [1.0e-5]	収束判定基準
-eprint [0]	残差の画面表示 -eprint {none 0} 何もしない -eprint {mem 1} 収束履歴をメモリに保存する -eprint {out 2} 収束履歴を画面に表示する -eprint {all 3} 収束履歴をメモリに保存し画面に表示する
-ie [ii]	Lanczos Iteration, Subspace Iteration の内部で使用する固有値解法の指定 -ie {pi 1} Power Iteration (Subspace Iteration のみ) -ie {ii 2} Inverse Iteration -ie {aii 3} Approximate Inverse Iteration
-shift [0.0]	固有値のシフト量
-initx_ones [true]	初期ベクトル x_0 の振舞い -initx_ones {false 0} 与えられた値を使用 -initx_ones {true 1} すべての要素を 1 にする
-omp_num_threads [t]	実行スレッド数 t は最大スレッド数

求解

固有値問題 $Ax = \lambda x$ を解くには関数

- C `int lis_solve(LIS_MATRIX A, LIS_VECTOR x,
 LIS_REAL eval, LIS_ESOLVER solver)`
- Fortran `subroutine lis_solve(LIS_MATRIX A, LIS_VECTOR x,
 LIS_ESOLVER solver, integer ierr)`

を用いる。

3.6 サンプルプログラム

線型方程式 $Ax = b$ を指定された解法で解き、その近似解を表示するプログラムを以下に示す。

行列 A は 12×12 の三重対角行列

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

である。右辺ベクトル b は近似解 x がすべて 1 となるように求めている。

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

テストプログラム: test3.c

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

テストプログラム: test3f.f

```

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

```

3.7 コンパイル・リンク

test3.c のユーザプログラムをコンパイル、リンクする方法について述べる。lis-(\$VERSION)/test ディレクトリにあるテストプログラム test3.c を SGI Altix 3700 上の Intel C/C++ Compiler 8.1 (icc)、Intel Fortran Compiler 8.1 (ifort) でコンパイルする場合の例を以下に示す。Lis ライブラリのインストール時に SA-AMG 前処理を利用するよう選択 (--enable-saamg) した場合は Fortran90 のコードが含まれるため、リンクは Fortran90 コンパイラで行わなければならない。また、MPI 環境では USE_MPI マクロを指定しなければならない。

逐次

コンパイル

```
>icc -c -I$(INSTALLDIR)/include test3.c
```

リンク

```
>icc -o test3 test3.o -llis
```

リンク (--enable-saamg)

```
>ifort -nofor_main -o test3 test3.o -llis
```

OpenMP

コンパイル

```
>icc -c -openmp -I$(INSTALLDIR)/include test3.c
```

リンク

```
>icc -openmp -o test3 test3.o -llis
```

リンク (--enable-saamg)

```
>ifort -nofor_main -openmp -o test3 test3.o -llis
```

MPI

コンパイル

```
>icc -c -DUSE_MPI -I$(INSTALLDIR)/include test3.c
```

リンク

```
>icc -o test3 test3.o -llis -lmpi
```

リンク (--enable-saamg)

```
>ifort -nofor_main -o test3 test3.o -llis -lmpi
```

OpenMP + MPI

コンパイル

```
>icc -c -openmp -DUSE_MPI -I$(INSTALLDIR)/include test3.c
```

リンク

```
>icc -openmp -o test3 test3.o -llis -lmpi
```

リンク (--enable-saamg)

```
>ifort -nofor_main -openmp -o test3 test3.o -llis -lmpi
```

次に、test3f.f のユーザープログラムをコンパイル・リンクする方法について述べる。lis-(\$VERSION)/test ディレクトリにあるテストプログラム test3f.f を SGI Altix 3700 上の Intel Fortran Compiler 8.1 (ifort) でコンパイルする場合の例を以下に示す。Fortran のユーザープログラムには#include 文が用いられているため、プリプロセッサを利用するようコンパイラオプションを指定しなければならない。ifort の場合のオプションは-fpp である。

逐次

コンパイル

```
>ifort -c -fpp -I$(INSTALLDIR)/include test3f.f
```

リンク

```
>ifort -o test3 test3.o -llis
```

OpenMP

コンパイル

```
>ifort -c -fpp -openmp -I$(INSTALLDIR)/include test3f.f
```

リンク

```
>ifort -openmp -o test3 test3.o -llis
```

MPI

コンパイル

```
>ifort -c -fpp -DUSE_MPI -I$(INSTALLDIR)/include test3f.f
```

リンク

```
>ifort -o test3 test3.o -llis -lmpi
```

OpenMP + MPI

コンパイル

```
>ifort -c -fpp -openmp -DUSE_MPI -I$(INSTALLDIR)/include test3f.f
```

リンク

```
>ifort -openmp -o test3 test3.o -llis -lmpi
```

3.8 実行

lis-(\$VERSION)/test ディレクトリにあるテストプログラム test3.c または test3f.f を SGI Altix 3700 上で

逐次

```
>./test3 -i bicgstab
```

OpenMP

```
>env OMP_NUM_THREADS=2 ./test3 -i bicgstab
```

MPI

```
>mpirun -np 2 ./test3 -i bicgstab
```

OpenMP + MPI

```
>mpirun -np 2 env OMP_NUM_THREADS=2 ./test3 -i bicgstab
```

と入力して実行すると以下のように表示される。

```
Initial vector x = 0
PRECISION : DOUBLE
SOLVER     : BiCGSTAB 4
PRECON     : None
STORAGE    : CRS
lis_solve is normal end
iter = 6
  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
```


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 [17]。本ライブラリでは、倍精度浮動小数点数を 2 個用いた”double-double”[18, 19] 型の 4 倍精度演算により、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 $a = a.hi + a.lo$, $\frac{1}{2}\text{ulp}(a.hi) \geq |a.lo|$ (上位 $a.hi$ と下位 $a.lo$ は倍精度浮動小数) により定義し、Dekker[20] と Knuth[21] 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 倍精度演算を実現している。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 [22] が、Fortran の表現形式 [23] 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 個利用しているため、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。

本ライブラリでは、入力として与えられる係数行列 A 、右辺 b 、初期値 x_0 、出力の解は倍精度としている。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 倍精度演算を利用するかどうかを指定するだけでよい。なお、x86 系の CPU に対しては SSE2 命令を用いて高速化を行っている [24]。

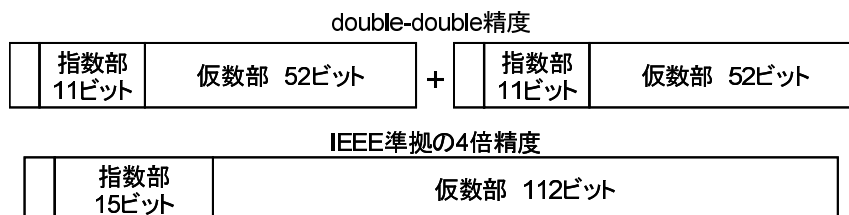


図 2: double-double 精度のビット数.

4.1 4倍精度演算の実行

Toeplitz 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、近似解を表示するテストプログラムが test4.c である。右辺ベクトル b は近似解 x がすべて 1 となるように求めている。n は行列 A の次数、gamma は γ である。test4.c を SGI Altix 3700(CPU: Itanium2) 上で

逐次・倍精度

```
>./test4 200 2.0 -precision double
```

または

```
>./test4 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
SOLVER    : BiCG 2
PRECON    : None
STORAGE   : CRS
lis_solve is LIS_MAXITER(code=4)
BiCG: iter      = 1001 iter_double = 1001 iter_quad = 0
BiCG: times     = 2.044368e-02
BiCG: p_times   = 4.768372e-06 (p_c = 4.768372e-06 p_i = 0.000000e+00 )
BiCG: i_times   = 2.043891e-02
BiCG: Residual  = 8.917591e+01
```

逐次・4倍精度

```
>./test4 200 2.0 -precision 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
SOLVER    : BiCG 2
PRECON    : None
STORAGE   : CRS
lis_solve is normal end
BiCG: iter      = 230 iter_double = 0 iter_quad = 230
BiCG: times     = 2.267408e-02
BiCG: p_times   = 4.549026e-04 (p_c = 5.006790e-06 p_i = 4.498959e-04 )
BiCG: i_times   = 2.221918e-02
BiCG: Residual  = 6.499145e-11
```

5 行列格納形式

本節では、ライブラリで使え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。 $n \times n$ 行列 $A = (a_{ij})$ の非零要素数を nnz とする。なお、Fortran の場合は、ユーザが目的の格納形式の配列を用意してライブラリで利用することはできない。

5.1 Compressed Row Storage (CRS)

CRS 形式は 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 行列の作り方 (逐次、OpenMP)

図 3 の行列 A を CRS 形式で格納すると図 3 の右図のようになる。この行列を CRS 形式で作成する場合のプログラムは以下のように記述する。

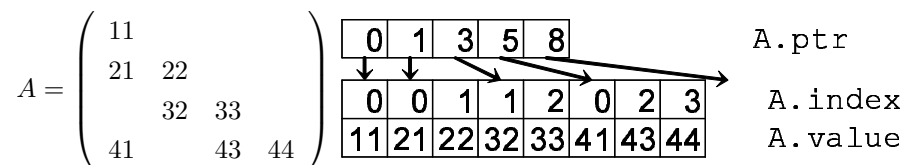


図 3: Data structures of CRS.

逐次、OpenMP

```

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

```

5.1.2 行列の作り方 (MPI)

図 3 の行列 A を 2 プロセッサ上に CRS 形式で格納すると図 4 のようになる。この行列を 2 プロセッサ上の CRS 形式で作成する場合のプログラムは以下のように記述する。

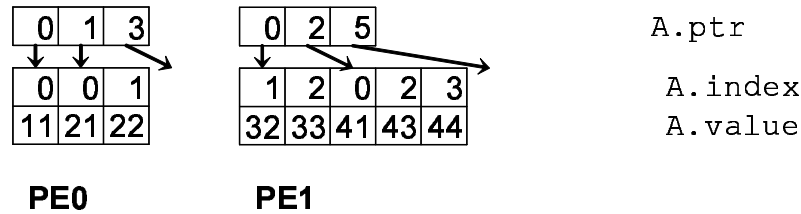


図 4: Data structures of CRS.

MPI

```

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

```

5.1.3 関連する関数

配列の関連付け

CRS 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_crs(int nnz, int *row, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.2 Compressed Column Storage (CCS)

CCS 形式は 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 行列の作り方 (逐次、OpenMP)

図 5 の行列 A を CCS 形式で格納すると図 5 の右図のようになる。この行列を CCS 形式で作成する場合のプログラムは以下のように記述する。

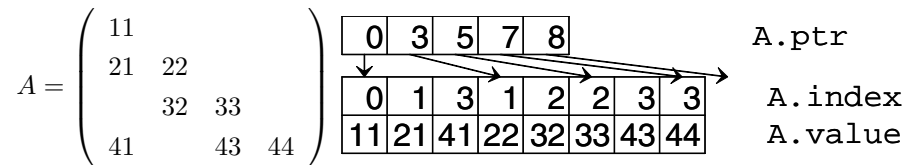


図 5: Data structures of CCS.

逐次、OpenMP

```

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

```

5.2.2 行列の作り方 (MPI)

図 5 の行列 A を 2 プロセッサ上に CCS 形式で格納すると図 6 のようになる。この行列を 2 プロセッサ上の CCS 形式で作成する場合のプログラムは以下のように記述する。

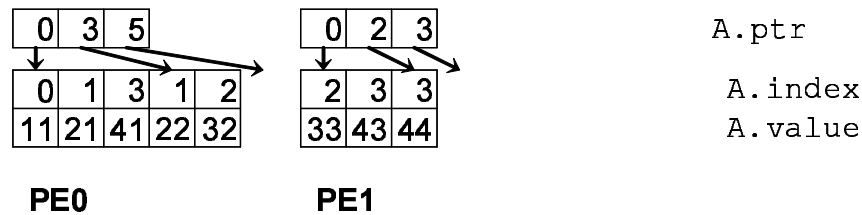


図 6: Data structures of CCS.

MPI

```

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

```

5.2.3 関連する関数

配列の関連付け

CCS 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_ccs(int nnz, int *row, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.3 Modified Compressed Sparse Row (MSR)

MSR 形式は CRS 形式を修正したものである。その違いは対角部分を分けて格納しているところである。MSR 形式は 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 n 番目までは行列 A の対角部分を格納する。 $n + 1$ 番目の要素は使用しない。 $n + 2$ 番目からは行列 A の対角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 $n + 1$ 番目までは行列 A の非対角部分の各行の開始位置を格納する。 $n + 2$ 番目からは行列 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 行列の作り方 (逐次、OpenMP)

図 7 の行列 A を MSR 形式で格納すると図 7 の右図のようになる。この行列を MSR 形式で作成する場合のプログラムは以下のように記述する。

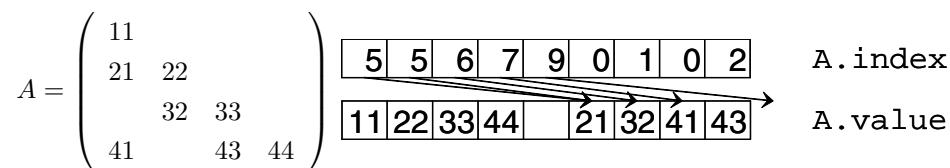


図 7: Data structures of MSR.

逐次、OpenMP

```

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

```

5.3.2 行列の作り方 (MPI)

図 7 の行列 A を 2 プロセッサ上に MSR 形式で格納すると図 8 のようになる。この行列を 2 プロセッサ上の MSR 形式で作成する場合のプログラムは以下のように記述する。

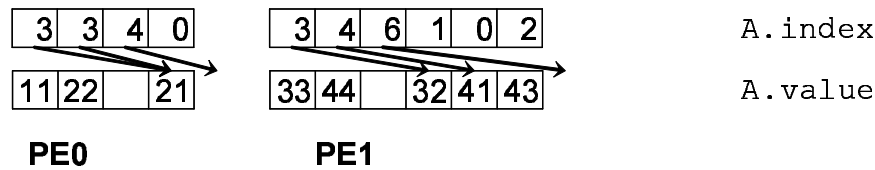


図 8: Data structures of MSR.

MPI

```

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

```

5.3.3 関連する関数

配列の関連付け

MSR 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_msr(int nnz, int ndz, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.4 Diagonal (DIA)

DIA は 2 つの配列 (index, value) に格納する。 nnd を行列 A の非零な対角の本数とする。

- 長さ $nnd \times n$ の倍精度配列 value は行列 A の非零な対角を格納する。
- 長さ nnd の整数配列 index は主対角から各対角へのオフセットを格納する。

OpenMP 版では以下のように修正している。

DIA は 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を格納する。
- 長さ $nprocs \times maxnnd$ の整数配列 index は主対角から各対角へのオフセットを格納する。

5.4.1 行列の作り方 (逐次)

図 9 の行列 A を DIA 形式で格納すると図 9 の右図のようになる。この行列を DIA 形式で作成する場合のプログラムは以下のように記述する。

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix} \quad \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline -3 & -1 & 0 & & & & & & & & & & \\ \hline 0 & 0 & 0 & 41 & 0 & 21 & 32 & 43 & 11 & 22 & 33 & 44 & \\ \hline \end{array} \quad \begin{array}{l} \text{A.index} \\ \text{A.value} \end{array}$$

図 9: Data structures of DIA.

逐次

```

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

```

図 9 の行列 A を 2 スレッドで DIA 形式で格納すると図 10 のようになる。この行列を 2 スレッドで DIA 形式で作成する場合のプログラムは以下のように記述する。

Figure 10: Data structures of DIA.

```

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

```

5.4.3 行列の作り方 (MPI)

図 9 の行列 A を 2 プロセッサ上に DIA 形式で格納すると図 11 のようになる。この行列を 2 プロセッサ上の DIA 形式で作成する場合のプログラムは以下のように記述する。

<table><tr><td>-1</td><td>0</td></tr><tr><td>0</td><td>21</td></tr></table>	-1	0	0	21	<table><tr><td>-3</td><td>-1</td><td>0</td></tr><tr><td>0</td><td>41</td><td>32</td></tr></table>	-3	-1	0	0	41	32	<table><tr><td>11</td><td>22</td></tr><tr><td>43</td><td>33</td></tr></table>	11	22	43	33	<table><tr><td>44</td></tr></table>	44	A.index
-1	0																		
0	21																		
-3	-1	0																	
0	41	32																	
11	22																		
43	33																		
44																			
PE0	PE1			A.value															

図 11: Data structures of DIA.

MPI

```

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

```

5.4.4 関連する関数

配列の関連付け

DIA 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_dia(int nnd, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.5 Ellpack-Itpack generalized diagonal (ELL)

ELL は 2 つの配列 (index, value) に格納する。maxnzs を行列 A の各行での非零要素数の最大値とする。

- 長さ $\text{maxnzs} \times n$ の倍精度配列 value は行列 A の各行の非零要素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数からなる。ただし、格納する非零要素数がない場合は 0 を格納する。
- 長さ $\text{maxnzs} \times n$ 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 行目の非零要素数を nnz_i とすると $\text{index}[\text{nnz}_i * n + i]$ にはその行番号 i を格納する。

5.5.1 行列の作り方 (逐次、OpenMP)

図 12 の行列 A を ELL 形式で格納すると図 12 の右図のようになる。この行列を ELL 形式で作成する場合のプログラムは以下のように記述する。

$$A = \begin{pmatrix} 11 & & & & \\ 21 & 22 & & & \\ & 32 & 33 & & \\ & & & 43 & 44 \\ 41 & & & & \end{pmatrix} \quad \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 0 & 0 & 1 & 2 & 2 & 0 & 1 & 2 & 3 \\ \hline 11 & 21 & 32 & 41 & & 0 & 22 & 33 & 43 & & 0 & 44 \\ \hline \end{array} \quad \begin{array}{l} \text{A.index} \\ \text{A.value} \end{array}$$

図 12: Data structures of ELL.

逐次、OpenMP

```

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

```

図 12 の行列 A を 2 プロセッサ上に ELL 形式で格納すると図 13 のようになる。この行列を 2 プロセッサ上の ELL 形式で作成する場合のプログラムは以下のように記述する。

Figure 13: Data structures of ELL.

```

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

```

配列の関連付け

- `int lis_matrix_set_ell(int maxnzs, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

47

5.6 Jagged Diagonal (JDS)

JDS 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。JDS は 4 つの配列 (perm, ptr, index, value) に格納する。 $maxn_zr$ 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 のぎざぎざ対角を格納する。最初のぎざぎざ対角は各行の最初の非零要素数からなる。次のぎざぎざ対角は各行の 2 番目の非零要素数からなる。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 $maxn_zr + 1$ の整数配列 ptr は各ぎざぎざ対角の開始位置を格納する。

OpenMP 版では以下のように修正を行っている。

JDS は 4 つの配列 (perm, ptr, index, value) に格納する。 $nprocs$ をスレッド数とする。 $maxn_zr_p$ 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。 $maxmaxn_zr$ は配列 $maxn_zr_p$ 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 のぎざぎざ対角を格納する。最初のぎざぎざ対角は各行の最初の非零要素数からなる。次のぎざぎざ対角は各行の 2 番目の非零要素数からなる。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 $nprocs \times (maxmaxn_zr + 1)$ 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ぎざぎざ対角の開始位置を格納する。

5.6.1 行列の作り方 (逐次)

図 14 の行列 A を JDS 形式で格納すると図 14 の右図のようになる。この行列を JDS 形式で作成する場合のプログラムは以下のように記述する。

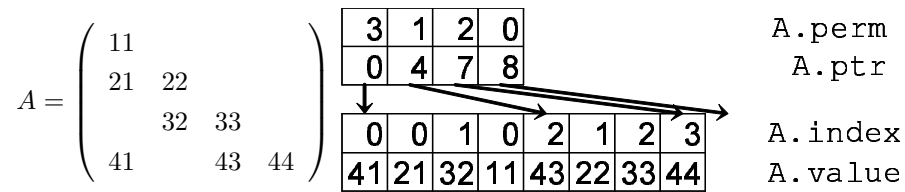


図 14: Data structures of JDS.

逐次

```

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

```

5.6.2 行列の作り方 (OpenMP)

図 14 の行列 A を 2 スレッドで JDS 形式で格納すると図 15 のようになる。この行列を 2 スレッドで JDS 形式で作成する場合のプログラムは以下のように記述する。

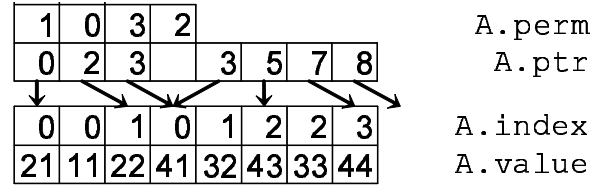


図 15: Data structures of JDS.

OpenMP

```

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

```


5.6.3 行列の作り方 (MPI)

図 14 の行列 A を 2 プロセッサ上に JDS 形式で格納すると図 16 のようになる。この行列を 2 プロセッサ上の JDS 形式で作成する場合のプログラムは以下のように記述する。

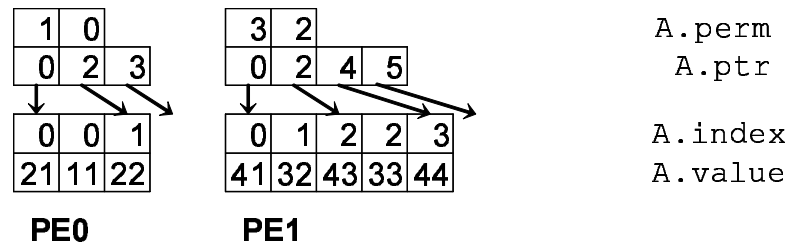


図 16: Data structures of JDS.

MPI

```

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

```

5.6.4 関連する関数

配列の関連付け

JDS 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_jds(int nnz, int maxnzs, int *perm, int *ptr, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.7 Block Sparse Row (BSR)

BSR 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 は CRS 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。 $nr = n/r$ 、 $nnzb$ を A の非零ブロック数とする。BSR は 3 つの配列 (bptr, bindex, value) に格納する。

- 長さ $nnzb \times r \times c$ の倍精度配列 value は非零ブロックの全要素を格納する。
- 長さ $nnzb$ 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 行列の作り方 (逐次、OpenMP)

図 17 の行列 A を BSR 形式で格納すると図 17 の右図のようになる。この行列を BSR 形式で作成する場合のプログラムは以下のように記述する。

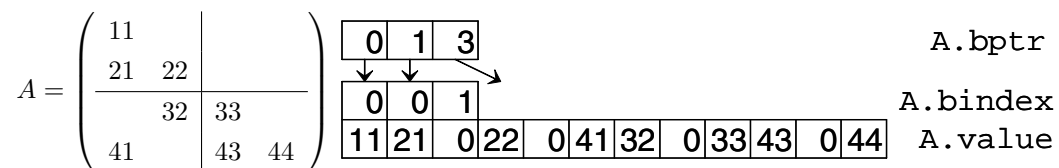


図 17: Data structures of BSR.

逐次、OpenMP

```

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

```

5.7.2 行列の作り方 (MPI)

図 17 の行列 A を 2 プロセッサ上に BSR 形式で格納すると図 18 のようになる。この行列を 2 プロセッサ上の BSR 形式で作成する場合のプログラムは以下のように記述する。

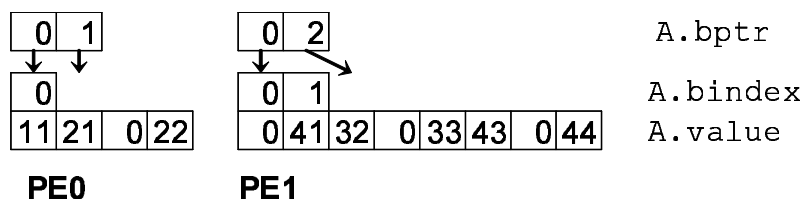


図 18: Data structures of BSR.

MPI

```

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

```

5.7.3 関連する関数

配列の関連付け

BSR 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_bsr(int bnr, int bnc, int bnnz, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.8 Block Sparse Column (BSC)

BSC 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 は CCS 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。 $nc = n/c$ 、 $nnzb$ を A の非零ブロック数とする。BSC は 3 つの配列 (bptr, bindex, value) に格納する。

- 長さ $nnzb \times r \times c$ の倍精度配列 value は非零ブロックの全要素を格納する。
- 長さ $nnzb$ 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 行列の作り方 (逐次、OpenMP)

図 19 の行列 A を BSC 形式で格納すると図 19 の右図のようになる。この行列を BSC 形式で作成する場合のプログラムは以下のように記述する。

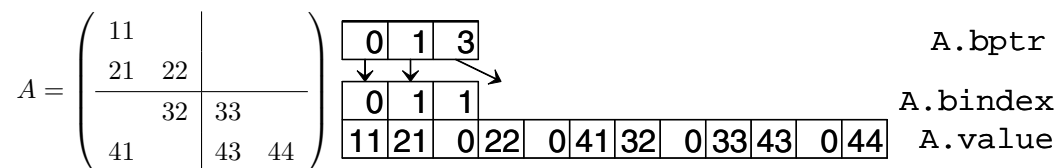


図 19: Data structures of BSC.

逐次、OpenMP

```

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

```

5.8.2 行列の作り方 (MPI)

図 19 の行列 A を 2 プロセッサ上に BSC 形式で格納すると図 20 のようになる。この行列を 2 プロセッサ上の BSC 形式で作成する場合のプログラムは以下のように記述する。

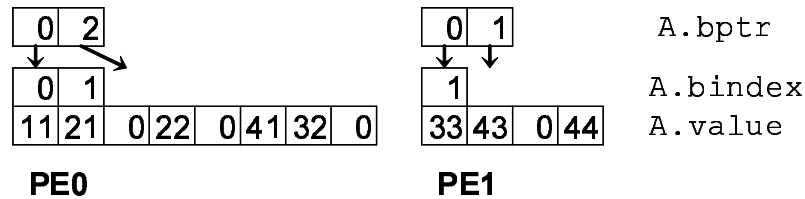


図 20: Data structures of BSC.

MPI

```

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

```

5.8.3 関連する関数

配列の関連付け

BSC 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_bsc(int bnr, int bnc, int bnnz, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)`

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 は CRS 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。nnzb 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。BSR は 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 nnzb 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を格納する。
- 長さ nnzb + 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 行列の作り方 (逐次、OpenMP)

図 21 の行列 A を VBR 形式で格納すると図 21 の右図のようになる。この行列を VBR 形式で作成する場合のプログラムは以下のように記述する。

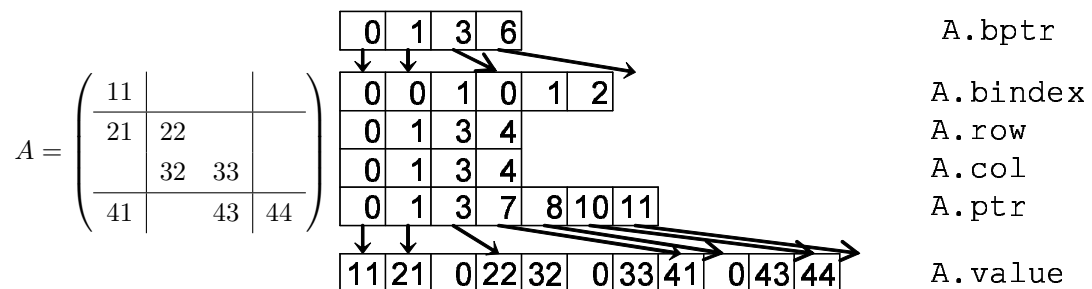


図 21: Data structures of VBR.

逐次、OpenMP

```

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

```

5.9.2 行列の作り方 (MPI)

図 21 の行列 A を 2 プロセッサ上に VBR 形式で格納すると図 22 のようになる。この行列を 2 プロセッサ上の VBR 形式で作成する場合のプログラムは以下のように記述する。

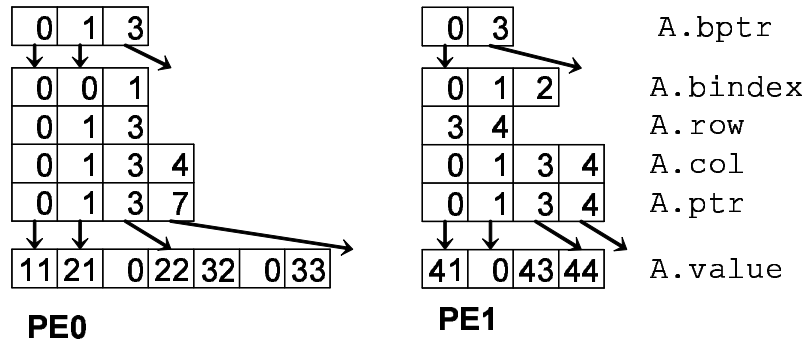


図 22: Data structures of VBR.

MPI

```

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

```

5.9.3 関連する関数

配列の関連付け

VBR 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_vbr(int nnz, int nr, int nc, int bnnz, int *row, int *col, int *ptr, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.10 Coordinate (COO)

COO は 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 行列の作り方 (逐次、OpenMP)

図 23 の行列 A を COO 形式で格納すると図 23 の右図のようになる。この行列を COO 形式で作成する場合のプログラムは以下のように記述する。

$$A = \begin{pmatrix} 11 & & & & & & & \\ 21 & 22 & & & & & & \\ & 32 & 33 & & & & & \\ 41 & & 43 & 44 & & & & \end{pmatrix} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 3 & 1 & 2 & 2 & 3 & 3 \\ \hline 0 & 0 & 0 & 1 & 1 & 2 & 2 & 3 \\ \hline 11 & 21 & 41 & 22 & 32 & 33 & 43 & 44 \\ \hline \end{array} \quad \begin{array}{l} A.\text{row} \\ A.\text{col} \\ A.\text{value} \end{array}$$

図 23: Data structures of COO.

逐次、OpenMP

```

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

```

5.10.2 行列の作り方 (MPI)

図 23 の行列 A を 2 プロセッサ上に COO 形式で格納すると図 24 のようになる。この行列を 2 プロセッサ上の COO 形式で作成する場合のプログラムは以下のように記述する。

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

図 24: Data structures of COO.

MPI

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

5.10.3 関連する関数

配列の関連付け

COO 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_coo(int nnz, int *row, int *col, LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

5.11 Dense (DNS)

DNS は 1 つの配列 (value) に格納する。

- 長さ $n \times n$ の倍精度配列 value は列優先で要素を格納する。

5.11.1 行列の作り方 (逐次、OpenMP)

図 25 の行列 A を DNS 形式で格納すると図 25 の右図のようになる。この行列を DNS 形式で作成する場合のプログラムは以下のように記述する。

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 11 & 21 & 0 & 41 & 0 & 22 & 32 & 0 \\ \hline 0 & 0 & 33 & 43 & 0 & 0 & 0 & 44 \\ \hline \end{array} \quad A.Value$$

図 25: Data structures of DNS.

逐次、OpenMP

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

5.11.2 行列の作り方 (MPI)

図 25 の行列 A を 2 プロセッサ上に DNS 形式で格納すると図 26 のようになる。この行列を 2 プロセッサ上の DNS 形式で作成する場合のプログラムは以下のように記述する。

11	21	0	22	0	41	32	0	A.Value
0	0	0	0	33	43	0	44	
PE0				PE1				

図 26: Data structures of DNS.

MPI

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

5.11.3 関連する関数

配列の関連付け

DNS 形式に必要な配列を行列 A に関連付けるには関数

- `int lis_matrix_set_dns(LIS_SCALAR *value, LIS_MATRIX A)`

を用いる。

6 Functions

本節では、ユーザーが使用できる関数について述べる。関数の解説は C を基準に記述している。配列は C では 0 オリジン、Fortran では 1 オリジンである。なお、C での各関数の戻り値、Fortran での `ierr` の値は以下のようにになっている。

戻り値

<code>LIS_SUCCESS(0)</code>	正常終了
<code>LIS_ILL_OPTION(1)</code>	オプションが不正
<code>LIS_BREAKDOWN(2)</code>	ブレイクダウン
<code>LIS_OUT_OF_MEMORY(3)</code>	メモリ不足
<code>LIS_MAXITER(4)</code>	最大反復回数までに収束しなかった
<code>LIS_NOT_IMPLEMENTED(5)</code>	まだ実装されていない
<code>LIS_ERR_FILE_IO(6)</code>	ファイル I/O エラー

6.1 ベクトル操作

ベクトル v の次数を `global_n` とする。ベクトル v を `nprocs` 台のプロセッサで行ブロック分割したとき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      int lis_vector_create(LIS_Comm comm, LIS_VECTOR *vec)
Fortran subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR vec, integer ierr)
```

機能

ベクトル v を作成する。

入力

<code>LIS_Comm</code>	MPI コミュニケータ
-----------------------	-------------

出力

<code>vec</code>	ベクトル
<code>ierr</code>	リターンコード

注釈

逐次、OpenMP 版では、`comm` の値は無視される。

6.1.2 lis_vector_destroy

```
C      int lis_vector_destroy(LIS_VECTOR vec)
Fortran subroutine lis_vector_destroy(LIS_VECTOR vec, 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

vec 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      int lis_vector_duplicate(void *vin, LIS_VECTOR *vout)
Fortran subroutine lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout,
                                         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。lis_vector_duplicate 関数は値はコピーされず、領域のみ確保される。値もコピーしたい場合はこの関数の後に lis_vector_copy 関数を用いる。

6.1.4 lis_vector_set_size

```
C      int lis_vector_set_size(LIS_VECTOR vec, int local_n, int global_n)
Fortran subroutine lis_vector_set_size(LIS_VECTOR vec, integer local_n,
      integer global_n, integer ierr)
```

機能

ベクトルのサイズを設定する

入力

vec	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。この関数はベクトルのサイズを、逐次、OpenMP 版では、ベクトルの次数は local_n = global_n となる。したがって、lis_vector_set_size(v,n,0) と lis_vector_set_size(v,0,n) はどちらも次数 n のベクトルを作成することを意味する。

MPI 版では、lis_vector_set_size(v,n,0) とすると、各プロセッサ p に次数 n_p の部分ベクトルを作成する。一方、lis_vector_set_size(v,0,n) とすると各プロセッサ p に次数 m_p の部分ベクトルを作成する。ただし、 m_p の値はライブラリ側で決定される。

```
C      int lis_vector_get_size(LIS_VECTOR v, int *local_n, int *global_n)
Fortran subroutine lis_vector_get_size(LIS_VECTOR v, integer local_n,
      integer global_n, integer ierr)
```

```
C      int lis_vector_get_range(LIS_VECTOR v, int *is, int *ie)
Fortran subroutine lis_vector_get_range(LIS_VECTOR v, integer is, integer ie,
      integer ierr)
```


6.1.7 lis_vector_set_value

```
C      int lis_vector_set_value(int flag, int i, LIS_SCALAR value, LIS_VECTOR v)
Fortran subroutine lis_vector_set_value(integer flag, integer i, LIS_SCALAR value,
      LIS_VECTOR v, integer ierr)
```

機能

ベクトル v の i 行目にスカラー値 $value$ 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	代入するスカラー値
v	代入されるベクトル

出力

v	i 行目にスカラー値 $value$ が代入されたベクトル
ierr	リターンコード

注釈

MPI 版では、部分ベクトルの i 行目ではなく全体ベクトルの i 行目を指定する。

6.1.8 lis_vector_get_value

```
C      int lis_vector_get_value(LIS_VECTOR v, int i, LIS_SCALAR *value)
Fortran subroutine lis_vector_get_value(LIS_VECTOR v, integer i, LIS_SCALAR value,
      integer ierr)
```

機能

ベクトル v の i 行目の値を $value$ に取得する。

入力

i	取得する場所
v	値を取得するベクトル

出力

value	スカラー値
ierr	リターンコード

注釈

MPI 版では、部分ベクトルの i 行目ではなく全体ベクトルの i 行目を指定する。

6.1.9 lis_vector_copy

```
C      int lis_vector_copy(LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_vector_copy(LIS_VECTOR x, LIS_VECTOR y, integer ierr)
```

機能

ベクトルの値をコピーする。 $y \leftarrow x$

入力

x コピー元ベクトル

出力

y コピー先ベクトル

ierr リターンコード

6.1.10 lis_vector_set_all

```
C      int lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR x)
Fortran subroutine lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR x, integer ierr)
```

機能

ベクトルのすべての要素にスカラー値 value を代入する。

入力

value 代入するスカラー値

v 代入されるベクトル

出力

v すべての要素に value が代入されたベクトル

ierr リターンコード

```
C      int lis_vector_is_null(LIS_VECTOR v)
Fortran subroutine lis_vector_is_null(LIS_VECTOR v, integer ierr)
```

ベクトル v が利用可能かどうか調べる。

V ベクトル

ierr	LIS_TRUE 利用可能
	LIS_FALSE 利用不可

6.2 行列操作

行列 A の次数を `global_n` \times `global_n` とする。行列 A を `nprocs` 台のプロセッサで行ブロック分割したとき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      int lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)
Fortran subroutine lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, integer ierr)
```

機能

行列 A 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、OpenMP 版では、`comm` の値は無視される。

6.2.2 `lis_matrix_destroy`

```
C      int lis_matrix_destroy(LIS_MATRIX A)
Fortran subroutine lis_matrix_destroy(LIS_MATRIX A, 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

```
C      int lis_matrix_duplicate(LIS_MATRIX Ain, LIS_MATRIX *Aout)
Fortran subroutine lis_matrix_duplicate(LIS_MATRIX Ain, LIS_MATRIX Aout,
      integer ierr)
```

既存の行列と同じ情報を持つ行列を作成する。

Ain 複製元の行列

Aout 複製先の行列

ierr	リターンコード
------	---------

`lis_matrix_duplicate` 関数は行列の要素の値はコピーされず、領域のみ確保される。要素の値もコピーしたい場合は `lis_matrix_copy` 関数を用いる。

```
C      int lis_matrix_malloc(LIS_MATRIX A, int nnz_row, int nnz[])
Fortran subroutine lis_matrix_malloc(LIS_MATRIX A, integer nnz_row, integer nnz[,
    integer ierr)
```

行列の領域を確保する。

A	行列
nnz_row	平均非零要素数
nnz	各行の非零要素数の配列

ierr	リターンコード
------	---------

`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` で効率よく要素を代入できるように、あらかじめ領域を確保する。

6.2.5 lis_matrix_set_value

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

機能

行列 A の i 行 j 列目に要素を代入する。

入力

flag	LIS_INS_VALUE 挿入 : $A(i,j) = \text{value}$ LIS_ADD_VALUE 加算代入 : $A(i,j) = A(i,j) + \text{value}$
i	行列の行番号
j	行列の列番号
value	代入するスカラー値
A	行列

出力

A	i 行 j 列目に要素が代入された行列
ierr	リターンコード

注釈

MPI 版では、部分行列の i 行 j 列目ではなく全体行列の i 行 j 列目を指定する。

lis_matrix_set_value 関数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 を用いた後には必ず lis_matrix_assemble 関数を呼び出さなければならない。

6.2.6 lis_matrix_assemble

```
C      int lis_matrix_assemble(LIS_MATRIX A)
Fortran subroutine lis_matrix_assemble(LIS_MATRIX A, integer ierr)
```

機能

行列をライブラリで利用可能状態にする。

入力

A	行列
---	----

出力

A	利用可能状態になった行列
ierr	リターンコード

6.2.7 lis_matrix_set_size

```
int lis_matrix_set_size(LIS_MATRIX A, int local_n, int global_n)
Fortran subroutine lis_matrix_set_size(LIS_MATRIX A, integer local_n,
    integer global_n, integer ierr)
```

機能

行列のサイズを設定する。

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、OpenMP 版では、行列のサイズは local_n = global_n となる。したがって、lis_matrix_set_size(A,n,0) と lis_matrix_set_size(A,0,n) はともに $n \times n$ のサイズを設定することを意味する。

MPI 版では、lis_matrix_set_size(A,n,0) とすると、各プロセッサ p で行列サイズが $n_p \times N$ となるように設定する。ここで、 N は各プロセッサの n_p の総和である。

一方、lis_matrix_set_size(A,0,n) とすると各プロセッサ p で行列サイズが $m_p \times n$ となるように設定する。ここで、 m_p は部分行列の行数でこの値はライブラリ側で決定される。

[illegible]

行列のサイズを取得する。

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
ierr	リターンコード

逐次、OpenMP 版では、`local_n = global_n` となる

```
C      int lis_matrix_get_range(LIS_MATRIX A, int *is, int *ie)
Fortran subroutine lis_matrix_get_range(LIS_MATRIX A, integer is, integer ie,
      integer ierr)
```

部分行列 A が全体行列のどこに存在しているのかを調べる。

A 部分行列

is	部分行列 A が全体行列で存在する開始行番号
ie	部分行列 A が全体行列で存在する終了行番号+1
ierr	リターンコード

逐次、OpenMP 版では、 $n \times n$ 行列では $is = 0$ 、 $ie = n$ となる。

6.2.10 lis_matrix_set_type

```
C      int lis_matrix_set_type(LIS_MATRIX A, int matrix_type)
Fortran subroutine lis_matrix_set_type(LIS_MATRIX A, int matrix_type, 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

行列作成時に A の matrix_type は LIS_MATRIX_CRS となっている。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 Row Storage	(CRS)	LIS_MATRIX_CRS
Compressed Column Storage	(CCS)	LIS_MATRIX_CCS
Modified Compressed Sparse Row	(MSR)	LIS_MATRIX_MSR
Diagonal	(DIA)	LIS_MATRIX_DIA
Ellpack-Itpack generalized diagonal	(ELL)	LIS_MATRIX_ELL
Jagged Diagonal	(JDS)	LIS_MATRIX_JDS
Block Sparse Row	(BSR)	LIS_MATRIX_BSR
Block Sparse Column	(BSC)	LIS_MATRIX_BSC
Variable Block Row	(VBR)	LIS_MATRIX_VBR
Dense	(DNS)	LIS_MATRIX_DNS
Coordinate	(COO)	LIS_MATRIX_COO

```
C      int lis_matrix_get_type(LIS_MATRIX A, int *matrix_type)
Fortran subroutine lis_matrix_get_type(LIS_MATRIX A, integer matrix_type,
                                     integer ierr)
```

ierr	リターンコード
------	---------

```
C      int lis_matrix_set_blocksize(LIS_MATRIX A, int bnr, int bnc, int row[],
      int col[])

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

ierr	リターンコード
------	---------

6.2.13 lis_matrix_convert

```
C      int lis_matrix_convert(LIS_MATRIX Ain, LIS_MATRIX Aout)
Fortran subroutine lis_matrix_convert(LIS_MATRIX Ain, LIS_MATRIX Aout, integer ierr)
```

機能

行列 A_{in} を指定の格納形式に変換した行列 A_{out} を作成する。

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

ier リターンコード

注釈

変換する格納形式の指定は lis_matrix_set_type を用いて Aout に設定する。BSR、BSC、VBR のブロックサイズ等の情報はユーザが lis_matrix_set_blocksize を用いて Aout に設定する。

元の行列から指定の格納形式への変換で以下の表の となっている経路は直接変換され、それ以外は記載されている形式を経由してから指定の格納形式に変換される。また、表に記載されていない経路は CRS を経由してから指定の格納形式に変換される。

元 \ 先	CRS	CCS	MSR	DIA	ELL	JDS	BSR	BSC	VBR	DNS	COO
CRS								CCS			
COO				CRS	CRS	CRS	CRS	CCS	CRS	CRS	

6.2.14 lis_matrix_copy

```
C      int lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout)
Fortran subroutine lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout, integer ierr)
```

機能

行列の要素をコピーする。

入力

Ain コピー元の行列

出力

Aout コピー先の行列

ier リターンコード

```
C      int lis_matrix_get_diagonal(LIS_MATRIX A, LIS_VECTOR d)
Fortran subroutine lis_matrix_get_diagonal(LIS_MATRIX A, LIS_VECTOR d, integer ierr)
```

行列 A の対角部分をベクトル d にコピーする。

A 行列

d 対角要素が格納されたベクトル

ierr	リターンコード
------	---------

6.2.16 lis_matrix_set_crs

```
C      int lis_matrix_set_crs(int nnz, int *ptr, int *index, LIS_SCALAR *value,
                             LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した CRS 形式に必要な配列を行列 A に関連付ける。

入力

nnz	非零要素数
ptr, index, value	CRS 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_crs を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.17 lis_matrix_set_ccs

```
C      int lis_matrix_set_ccs(int nnz, int *ptr, int *index, LIS_SCALAR *value,
                             LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した CCS 形式に必要な配列を行列 A に関連付ける。

入力

nnz	非零要素数
ptr, index, value	CCS 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_ccs を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.18 lis_matrix_set_msr

```
C      int lis_matrix_set_msr(int nnz, int ndz, int *index, LIS_SCALAR *value,
                             LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した MSR 形式に必要な配列を行列 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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.19 lis_matrix_set_dia

```
C      int lis_matrix_set_dia(int nnd, int *index, LIS_SCALAR *value,
                             LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した DIA 形式に必要な配列を行列 A に関連付ける。

入力

nnd	非零な対角の本数
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() を用いた後には必ず lis_matrix_assemble() を呼び出さなければならない。

6.2.20 lis_matrix_set_ell

```
C      int lis_matrix_set_ell(int maxnzs, int index[], LIS_SCALAR value[],
                             LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した ELL 形式に必要な配列を行列 A に関連付ける。

入力

maxnzs	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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.21 lis_matrix_set_jds

```
C      int lis_matrix_set_jds(int nnz, int maxnzs, int *perm, int *ptr, int *index,
                             LIS_SCALAR *value, LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した JDS 形式に必要な配列を行列 A に関連付ける。

入力

nnz	非零要素数
maxnzs	各行の非零要素数の最大値
perm, ptr, index, value	JDS 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_jds を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.22 lis_matrix_set_bsr

```
C      int lis_matrix_set_bsr(int bnr, int bnc, int bnnz, int *bptr, int *bindex,
        LIS_SCALAR value[], LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した BSR 形式に必要な配列を行列 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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.23 lis_matrix_set_bsc

```
C      int lis_matrix_set_bsc(int bnr, int bnc, int bnnz, int *bptr, int *bindex,
                             LIS_SCALAR *value, LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した BSC 形式に必要な配列を行列 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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.24 lis_matrix_set_vbr

```
C      int lis_matrix_set_vbr(int nnz, int nr, int nc, int bnnz, int *row,
                             int *col, int *ptr, int *bptr, int *bindex, LIS_SCALAR *value,
                             LIS_MATRIX A)
```

Fortran 未対応

機能

ユーザー自身が作成した VBR 形式に必要な配列を行列 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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.25 lis_matrix_set_coo

```
C      int lis_matrix_set_coo(int nnz, int row[], int col[], LIS_SCALAR value[],
                             LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した COO 形式に必要な配列を行列 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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.2.26 lis_matrix_set_dns

```
C      int lis_matrix_set_dns(LIS_SCALAR value[], LIS_MATRIX A)
Fortran 未対応
```

機能

ユーザー自身が作成した DNS 形式に必要な配列を行列 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 を用いた後には必ず lis_matrix_assemble を呼び出さなければならない。

6.3 ベクトルと行列の計算

6.3.1 lis_vector_scale

```
C      int lis_vector_scale(LIS_SCALAR alpha, LIS_VECTOR x)
Fortran subroutine lis_vector_scale(LIS_SCALAR alpha, LIS_VECTOR x, integer ierr)
```

機能

ベクトルのすべての値を alpha 倍する。 $x \leftarrow \alpha x$

入力

alpha	スカラー値
x	alpha 倍するベクトル

出力

x	すべての要素が alpha 倍されたベクトル
ierr	リターンコード

6.3.2 lis_vector_dot

```
C      int lis_vector_dot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR *val)
Fortran subroutine lis_vector_dot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR val,
                                   integer ierr)
```

機能

内積を計算する。 $val \leftarrow x^T y$

入力

x	ベクトル
y	ベクトル

出力

val	内積の値
ierr	リターンコード

```
C      int lis_vector_nrm2(LIS_VECTOR x, LIS_REAL *val)
Fortran subroutine lis_vector_nrm2(LIS_VECTOR x, LIS_REAL val, integer ierr)
```

入力

出力

ierr	リターンコード
------	---------

```
C      int lis_vector_nrm1(LIS_VECTOR x, LIS_REAL *val)
Fortran subroutine lis_vector_nrm1(LIS_VECTOR x, LIS_REAL val, integer ierr)
```

入力

出力

ierr	リターンコード
------	---------

6.3.5 lis_vector_axpy

```
C      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,
                                   integer ierr)
```

機能

$y \leftarrow \alpha x + y$ を計算する。

入力

alpha	スカラー値
x, y	ベクトル

出力

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

6.3.6 lis_vector_xpay

```
C      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,
                                   integer ierr)
```

機能

$y \leftarrow x + \alpha y$ を計算する。

入力

alpha	スカラー値
x, y	ベクトル

出力

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

6.3.7 lis_vector_axpyz

```
C      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, integer ierr)
```

機能

$z \leftarrow \alpha x + y$ を計算する。

入力

alpha	スカラー値
x, y	ベクトル

出力

z	$x + \alpha y$ の計算結果
ierr	リターンコード

6.3.8 lis_matrix_scaling

```
C      int lis_matrix_scaling(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR d, int action)
Fortran subroutine lis_matrix_scaling(LIS_MATRIX A, LIS_VECTOR b,
                                   LIS_VECTOR d, integer action, integer ierr)
```

機能

行列のスケーリングを行う。

入力

A	スケーリングを行う行列
b	スケーリングを行うベクトル
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 $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ 、 $D^{-1/2}$ は対角要素に $1/\sqrt{a_{ii}}$ を持つ対角行列

出力

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

6.3.9 lis_matvec

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

機能

行列ベクトル積 $y \leftarrow Ax$ を行う。

入力

A	行列
x	ベクトル

出力

y	ベクトル
---	------

6.3.10 lis_matvect

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

機能

転置行列ベクトル積 $y \leftarrow A^T x$ を行う。

入力

A	行列
x	ベクトル

出力

y	ベクトル
---	------

6.4 線型方程式の求解

6.4.1 lis_solver_create

```
C      int lis_solver_create(LIS_SOLVER *solver)
Fortran subroutine lis_solver_create(LIS_SOLVER solver, integer ierr)
```

機能

ソルバー（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	ソルバー
ierr	リターンコード

注釈

ソルバーは線型方程式解法の情報を持つ。

6.4.2 lis_solver_destroy

```
C      int lis_solver_destroy(LIS_SOLVER solver)
Fortran subroutine lis_solver_destroy(LIS_SOLVER solver, integer ierr)
```

機能

不要になったソルバー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ー
--------	---------------

出力

ierr	リターンコード
------	---------

6.4.3 lis_solver_set_option

```
C      int lis_solver_set_option(char *text, LIS_SOLVER solver)
Fortran subroutine lis_solver_set_option(character text, LIS_SOLVER solver,
      integer ierr)
```

機能

線型方程式解法のオプションをソルバーに設定する。

入力

text	コマンドラインオプション
------	--------------

出力

solver	ソルバー
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 のデフォルト値が 1000 であることを意味する。

線型方程式解法の指定 デフォルト: -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] l の値
GPBiCG	-i {gpbicg 6}	
TFQMR	-i {tfqmr 7}	
Orthomin(m)	-i {orthomin 8}	-restart [40] リスタート m の値
GMRES(m)	-i {gmres 9}	-restart [40] リスタート m の値
Jacobi	-i {jacobi 10}	
Gauss-Seidel	-i {gs 11}	
SOR	-i {sor 12}	-omega [1.9] 緩和係数 ω の値 ($0 < \omega < 2$)
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 m の値
IDR(s)	-i {idrs 21}	-restart [40] リスタート s の値
MINRES	-i {minres 22}	

前処理の指定 デフォルト: -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 法の緩和係数 ω ($0 < \omega < 2$) -hybrid_ell [2] BiCGSTAB(l) 法の l の値 -hybrid_restart [40] GMRES, Orthomin のリスタート値
I+S	-p {is 5}	-is_alpha [1.0] $I + \alpha S^{(m)}$ 型前処理のパラメータ α -is_m [3] $I + \alpha S^{(m)}$ 型前処理のパラメータ 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)
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}	-ilut_drop [0.05] ドロップ基準 -ilut_rate [5.0] 最大フィルイン数の倍率
additive Schwarz	-adds true	-adds_iter [1] 繰り返し回数

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
-print [0]	残差の画面表示 -print {none 0} 何もしない -print {mem 1} 収束履歴をメモリに保存する -print {out 2} 収束履歴を画面に表示する -print {all 3} 収束履歴をメモリに保存し画面に表示する
-scale [0]	スケーリングの選択。スケーリング結果は元の行列、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 $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ $D^{-1/2}$ は対角要素に $1/\sqrt{a_{ii}}$ を持つ対角行列
-initx_zeros [true]	初期ベクトル x_0 の振舞い -initx_zeros {false 0} 与えられた値を使用 -initx_zeros {true 1} すべての要素を 0 にする
-omp_num_threads [t]	実行スレッド数 t は最大スレッド数

演算精度 デフォルト: -precision double		
精度	オプション	補助オプション
倍精度	-precision {double 0}	
4 倍精度	-precision {quad 1}	

6.4.4 lis_solver_set_optionC

```
C      int lis_solver_set_optionC(LIS_SOLVER solver)
Fortran subroutine lis_solver_set_optionC(LIS_SOLVER solver, integer ierr)
```

機能

ユーザー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ーに設定する。

入力

なし

出力

solver

ソルバー

ierr

リターンコード

6.4.5 lis_solve

```
C      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, integer ierr)
```

機能

指定された解法で線型方程式 $Ax = b$ を解く。ソルバーに与えられた出力は `lis_solver_get_iters`、`lis_solver_get_time`、`lis_solver_get_residualnorm` で取得する。

入力

A	係数行列
b	右辺ベクトル
x	初期ベクトル
solver	ソルバー

出力

x	近似解
solver	反復回数、計算時間等の情報
ierr	リターンコード

6.4.6 lis_solver_get_iters

```
C      int lis_solver_get_iters(LIS_SOLVER solver, int *iters)
Fortran subroutine lis_solver_get_iters(LIS_SOLVER solver, integer iters,
                                         integer ierr)
```

機能

ソルバーから反復回数を取得する。

入力

solver	ソルバー
--------	------

出力

iters	反復回数
ierr	リターンコード

6.4.7 lis_solver_get_itersex

```
C      int lis_solver_get_itersex(LIS_SOLVER solver, int *iters,
                                int *iters_double, int *iters_quad)
Fortran subroutine lis_solver_get_itersex(LIS_SOLVER solver, integer iters,
                                         integer iters_double, integer iters_quad, integer ierr)
```

機能

ソルバーから反復回数を取得する。

入力

solver	ソルバー
--------	------

出力

iters	総反復回数
iters_double	倍精度演算の反復回数
iters_quad	4倍精度演算の反復回数
ierr	リターンコード

6.4.8 lis_solver_get_time

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

機能

ソルバーから計算時間を取得する。

入力

solver	ソルバー
--------	------

出力

times	経過時間
ierr	リターンコード

6.4.9 lis_solver_get_timeex

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

機能

ソルバーから計算時間を取得する。

入力

solver	ソルバー
--------	------

出力

times	itimes と ptimes の合計
itimes	線型方程式解法の経過時間
ptimes	前処理の経過時間
p_c_times	前処理行列作成の経過時間
p_i_times	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

6.4.10 lis_solver_get_residualnorm

```
C      int lis_solver_get_residualnorm(LIS_SOLVER solver, LIS_REAL *residual)
Fortran subroutine lis_solver_get_residualnorm(LIS_SOLVER solver,
      LIS_REAL residual, integer ierr)
```

機能

ソルバーから近似解 x で再計算した $b - Ax$ の 2 ノルムを取得する。

入力

solver	ソルバー
--------	------

出力

residual	$b - Ax$ の 2 ノルム
ierr	リターンコード

6.4.11 lis_solver_get_rhistory

```
C      int lis_solver_get_rhistory(LIS_SOLVER solver, LIS_VECTOR v)
Fortran subroutine lis_solver_get_rhistory(LIS_SOLVER solver,
      LIS_VECTOR v, integer ierr)
```

機能

ソルバーから収束履歴を取得する。

入力

なし

出力

v	収束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 `lis_vector_create` 関数で作成しておかなければならない。ベクトル v の次数 n が収束履歴の長さよりも小さい場合は収束履歴の最初から n 個までを取得する。

```
C      int lis_solver_get_solver(LIS_SOLVER solver, int *nsol)
Fortran subroutine lis_solver_get_solver(LIS_SOLVER solver, integer nsol,
      integer ierr)
```

ソルバーから選択されている線型方程式解法の番号を取得する。

solver ソルバー

nsol 線型方程式解法の番号

ierr	リターンコード
------	---------

線型方程式解法の番号は以下の通りである。

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

6.4.13 lis_solver_get_solvername

```
C      int lis_get_solvername(int nsol, char *name)
Fortran subroutine lis_get_solvername(integer nsol, character name, 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

6.5 固有値問題の求解

6.5.1 lis_esolver_create

```
C      int lis_esolver_create(LIS_ESOLVER *esolver)
Fortran subroutine lis_esolver_create(LIS_ESOLVER esolver, integer ierr)
```

機能

ソルバー（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	ソルバー
ierr	リターンコード

注釈

ソルバーは固有値解法の情報を持つ。

6.5.2 lis_esolver_destroy

```
C      int lis_esolver_destroy(LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_destroy(LIS_ESOLVER esolver, integer ierr)
```

機能

不要になったソルバー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ー
---------	---------------

出力

ierr	リターンコード
------	---------

```
C      int lis_esolver_set_option(char *text, LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver,
      integer ierr)
```

固有値解法のオプションをソルバーに設定する。

text コマンドラインオプション

esolver ソルバー

ierr	リターンコード
------	---------

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 [10000] は-emaxiter のデフォルト値が 10000 であることを意味する。

固有値解法の指定 デフォルト: -e pi			
固有値解法	オプション	補助オプション	
Power Iteration	-e {pi} 1}		
Inverse Iteration	-e {ii} 2}	-i [bicg]	線型方程式解法
Approximate Inverse Iteration	-e {aii} 3}		
Conjugate Gradient	-e {cg} 4}		
Lanczos Iteration	-e {li} 5}	-ss [10]	部分空間の大きさ
Subspace Iteration	-e {si} 6}	-ss [10]	部分空間の大きさ

前処理の指定 デフォルト: -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 法の緩和係数 ω ($0 < \omega < 2$) -hybrid_ell [2] BiCGSTAB(l) 法の l の値 -hybrid_restart [40] GMRES, Orthomin のリスタート値
I+S	-p {is 5}	-is_alpha [1.0] $I + \alpha S^{(m)}$ 型前処理のパラメータ α -is_m [3] $I + \alpha S^{(m)}$ 型前処理のパラメータ 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)
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}	-ilut_drop [0.05] ドロップ基準 -ilut_rate [5.0] 最大フィルイン数の倍率
additive Schwarz	-adds true	-adds_iter [1] 繰り返し回数

その他のオプション

オプション	
-emaxiter [10000]	最大反復回数
-etol [1.0e-5]	収束判定基準
-eprint [0]	残差の画面表示 -eprint {none 0} 何もしない -eprint {mem 1} 収束履歴をメモリに保存する -eprint {out 2} 収束履歴を画面に表示する -eprint {all 3} 収束履歴をメモリに保存し画面に表示する
-ie [ii]	Lanczos Iteration, Subspace Iteration の内部で使用する固有値解法の指定 -ie {pi 1} Power Iteration (Subspace Iteration のみ) -ie {ii 2} Inverse Iteration -ie {aii 3} Approximate Inverse Iteration
-shift [0.0]	固有値のシフト量
-initx_ones [true]	初期ベクトル x_0 の振舞い -initx_ones {false 0} 与えられた値を使用 -initx_ones {true 1} すべての要素を 1 にする
-omp_num_threads [t]	実行スレッド数 t は最大スレッド数

6.5.4 lis_esolver_set_optionC

```
C      int lis_esolver_set_optionC(LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, integer ierr)
```

機能

ユーザー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ーに設定する。

入力

なし

出力

esolver	ソルバー
ier	リターンコード

6.5.5 lis_solve

```
C      int lis_solve(LIS_MATRIX A, LIS_VECTOR x,
                   LIS_REAL evalue, LIS_ESOLVER esolver)
Fortran subroutine lis_solve(LIS_MATRIX A, LIS_VECTOR x,
                             LIS_ESOLVER esolver, integer ierr)
```

機能

指定された解法で固有値問題 $Ax = \lambda x$ を解く。ソルバーに与えられた出力は `lis_esolver_get_iters`、`lis_esolver_get_time`、`lis_esolver_get_evalues`、`lis_esolver_get_evectors`、`lis_esolver_get_residualnorm` で取得する。

入力

A	係数行列
x	初期ベクトル
esolver	ソルバー

出力

evalue	近似固有値
x	近似固有ベクトル
esolver	反復回数、計算時間等の情報
ierr	リターンコード

6.5.6 lis_esolver_get_iters

```
C      int lis_esolver_get_iters(LIS_ESOLVER esolver, int *iters)
Fortran subroutine lis_esolver_get_iters(LIS_ESOLVER esolver, integer iters,
      integer ierr)
```

機能

ソルバーから反復回数を取得する。

入力

esolver	ソルバー
---------	------

出力

iters	反復回数
ierr	リターンコード

6.5.7 lis_esolver_get_itersex

```
C      int lis_esolver_get_itersex(LIS_ESOLVER esolver, int *iters)
Fortran subroutine lis_esolver_get_itersex(LIS_ESOLVER esolver, integer iters,
      integer ierr)
```

機能

ソルバーから反復回数を取得する。

入力

esolver	ソルバー
---------	------

出力

iters	総反復回数
iters_double	倍精度演算の反復回数
iters_quad	4 倍精度演算の反復回数
ierr	リターンコード

6.5.8 lis_esolver_get_time

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

機能

ソルバーから計算時間を取得する。

入力

esolver	ソルバー
---------	------

出力

times	経過時間
ierr	リターンコード

6.5.9 lis_esolver_get_timeex

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

機能

ソルバーから計算時間を取得する。

入力

esolver	ソルバー
---------	------

出力

times	固有値解法の経過時間
itimes	固有値解法中の線型方程式解法の経過時間
ptimes	固有値解法中の線型方程式解法前処理の経過時間
p_c_times	前処理行列作成の経過時間
p_i_times	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

6.5.10 lis_esolver_get_residualnorm

```
C      int lis_esolver_get_residualnorm(LIS_ESOLVER esolver, LIS_REAL *residual)
Fortran subroutine lis_esolver_get_residualnorm(LIS_ESOLVER esolver,
        LIS_REAL residual, integer ierr)
```

機能

ソルバーから近似固有ベクトル x で再計算した $(\lambda x - Ax)/\lambda$ の 2 ノルムを取得する。

入力

esolver	ソルバー
---------	------

出力

residual	$(\lambda x - Ax)/\lambda$ の 2 ノルム
ierr	リターンコード

6.5.11 lis_esolver_get_rhistory

```
C      int lis_esolver_get_rhistory(LIS_ESOLVER esolver, LIS_VECTOR v)
Fortran subroutine lis_esolver_get_rhistory(LIS_ESOLVER esolver,
        LIS_VECTOR v, integer ierr)
```

機能

収束履歴をベクトルに格納する。

入力

なし

出力

v	収束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 `lis_vector_create` 関数で作成しておかなければならない。ベクトル v の次数 n が収束履歴の長さよりも小さい場合は収束履歴の最初から n 個までを取得する。

```
C      int lis_esolver_get_evalues(LIS_ESOLVER esolver, LIS_VECTOR v)
Fortran subroutine lis_esolver_get_evalues(LIS_ESOLVER esolver,
      LIS_VECTOR v, integer ierr)
```

```
C      int lis_esolver_get_evecors(LIS_ESOLVER esolver, LIS_VECTOR *v)
Fortran subroutine lis_esolver_get_evecors(LIS_ESOLVER esolver,
      LIS_VECTOR v, integer ierr)
```

```
C      int lis_esolver_get_esolver(LIS_ESOLVER esolver, int *nsol)
Fortran subroutine lis_esolver_get_esolver(LIS_ESOLVER esolver, integer nsol,
      integer ierr)
```

ソルバーから選択されている固有値解法の番号を取得する。

esolver ソルバー

nsol	固有値解法の番号
------	----------

ierr	リターンコード
------	---------

固有値解法の番号は以下の通りである。

解法	番号
Power Iteration	1
Inverse Iteration	2
Approximate Inverse Iteration	3
Conjugate Gradient	4
Lanczos Iteration	5
Subspace Iteration	6

```
C      int lis_get_esolvername(int esolver, char *name)
Fortran subroutine lis_get_esolvername(integer esolver, character name,
      integer ierr)
```

固有値解法の番号から解法名を取得する。

nesol 固有値解法の番号

name	固有値解法名
------	--------

ierr	リターンコード
------	---------

6.6 ファイル入出力

6.6.1 lis_input_matrix

```
C      int lis_input_matrix(LIS_MATRIX A, char *filename)
Fortran subroutine lis_input_matrix(LIS_MATRIX A, character filename,
                                   integer ierr)
```

機能

ファイルから行列データを読み込む。

入力

filename	読み込むファイルのファイル名
----------	----------------

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

対応しているファイルフォーマットは以下のとおりである：

- MatrixMarket フォーマット
- Harwell-Boeing フォーマット

これらのデータ構造は付録 A を参照せよ。

6.6.2 lis_input_vector

```
C      int lis_input_vector(LIS_VECTOR v, char *filename)
Fortran subroutine lis_input_vector(LIS_VECTOR v, character filename, integer ierr)
```

機能

ファイルからベクトルデータを読み込む。

入力

filename	読み込むファイルのファイル名
----------	----------------

出力

v	ベクトル
ierr	リターンコード

注釈

対応しているファイルフォーマットは

- PLAIN フォーマット
- MM フォーマット

これらのデータ構造は付録 A を参照せよ。

6.6.3 lis_input

```
C      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, integer ierr)
```

機能

ファイルから行列、ベクトルデータを読み込む。

入力

filename	読み込むファイルのファイル名
----------	----------------

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

対応しているファイルフォーマットは以下のとおりである：

- MatrixMarket フォーマット（ベクトルデータも読み込めるように拡張）
- Harwell-Boeing フォーマット

これらのデータ構造は付録 A を参照せよ。

6.6.4 lis_output_vector

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

機能

ベクトルデータをファイルに書き込む。

入力

v	ベクトル	
format	ファイルフォーマット	
	LIS_FMT_PLAIN	PLAIN フォーマット
	LIS_FMT_MM	MM フォーマット
filename	書き込むファイルのファイル名	

出力

ierr	リターンコード
------	---------

注釈

ファイルフォーマットのデータ構造は付録 A を参照せよ。

6.6.5 lis_output

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

機能

行列、ベクトルデータを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ときは NULL
x	解ベクトル。ファイルに書き込まないときは NULL
format	ファイルフォーマット
	LIS_FMT_MM MatrixMarket フォーマット
filename	書き込むファイルのファイル名

出力

ierr	リターンコード
------	---------

注釈

ファイルフォーマットのデータ構造は付録 A を参照せよ。

6.6.6 lis_solver_output_rhistory

```
C      int lis_solver_output_rhistory(LIS_SOLVER solver, char *filename)
Fortran subroutine lis_solver_output_rhistory(LIS_SOLVER solver,
      character filename)
```

機能

線型方程式解法の収束履歴をファイルに書き出す。

入力

solver	ソルバー
filename	書き込むファイルのファイル名

出力

ierr	リターンコード
------	---------

6.6.7 lis_esolver_output_rhistory

```
C      int lis_esolver_output_rhistory(LIS_ESOLVER esolver, char *filename)
Fortran subroutine lis_esolver_output_rhistory(LIS_ESOLVER esolver,
      character filename)
```

機能

固有値解法の収束履歴をファイルに書き出す。

入力

esolver	ソルバー
filename	書き込むファイルのファイル名

出力

ierr	リターンコード
------	---------

6.7 その他

6.7.1 lis_initialize

```
C      int lis_initialize(int* argc, char** argv[])
Fortran subroutine lis_initialize(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
------	---------

6.7.2 lis_finalize

```
C      void lis_finalize()
Fortran subroutine lis_finalize(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
------	---------

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

経過時間を計測する。

なし

ある時点からの時間経過を double 型の値（単位は秒）として返す。

処理時間を測定したい場合は、時間の測定を開始する直前と終了した直後の時間を `lis_wtime` により測定し、その差を求める。

```
C      void CHKERR(int err)
Fortran subroutine CHKERR(integer err)
```

実行した関数がエラーかどうか判断する。

err リターンコード

なし

エラーならば `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] 藤野清次, 藤原牧, 吉田正浩. 準残差の最小化に基づく積型 BiCG 法. 日本計算工学会論文集, 2005.
<http://save.k.u-tokyo.ac.jp/jsces/trans/trans2005/No20050028.pdf>.
- [2] 曾我部知広, 杉原正顕, 張紹良. 共役残差法の非対称行列への拡張. 日本応用数理学会論文誌, Vol. 15, No. 3, pp. 445–460, 2005.
- [3] 阿部邦美, 曾我部知広, 藤野清次, 張紹良. 非対称行列用共役残差法に基づく積型反復解法. 情報処理学会論文誌, Vol. 48, No. SIG8(ACS18), pp. 11–21, 2007.
- [4] 藤野清次, 尾上勇介. BiCR 法の残差をベースにした BiCRSafe 法の収束性評価. 情報処理学会研究報告, 2007-HPC-111, pp. 25–30, 2007.
- [5] Y. Saad. A Flexible Inner-outer Preconditioned GMRES Algorithm. SIAM J. Sci. Stat. Comput., Vol. 14, pp. 461–469, 1993.
- [6] Y. Saad. ILUT: a dual threshold incomplete LU factorization. Numerical Linear Algebra with Applications, Vol. 1, No. 4, pp. 387–402, 1994.
- [7] ITSOL: ITERATIVE SOLVERS package
<http://www-users.cs.umn.edu/~saad/software/ITSOL/index.html>
- [8] N. Li, Y. Saad and E. Chow. Crout version of ILU for general sparse matrices. SIAM J. Sci. Comput., Vol. 25, pp. 716–728, 2003.
- [9] Toshiyuki Kohno, Hisashi Kotakemori and Hiroshi Niki. Improving the Modified Gauss-Seidel Method for Z-matrices. Linear Algebra and its Applications, Vol. 267, pp. 113–123, 1997.
- [10] A. Fujii, A. Nishida, and Y. Oyanagi. Evaluation of Parallel Aggregate Creation Orders : Smoothed Aggregation Algebraic Multigrid Method. High Performance Computational Science And Engineering, pp. 99–122, Springer, 2005.
- [11] 阿部邦美, 張紹良, 長谷川秀彦, 姫野龍太郎. SOR 法を用いた可変的前処理付き一般化共役残差法. 日本応用数理学会論文誌, Vol. 11, No. 4, pp. 157–170, 2001.
- [12] R. Bridson and W.-P. Tang. Refining an approximate inverse. J. Comput. Appl. Math., Vol. 123, pp. 293–306, 2000.
- [13] P. Sonneveld and M. B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. TR 07-07, Math. Anal., Delft Univ. of Tech., 2007.
http://ta.twi.tudelft.nl/TWA_Reports/07-07.pdf.
- [14] A. Greenbaum. Iterative Methods for Solving Linear Systems. SIAM, 1997.
- [15] A. V. Knyazev. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM J. Sci. Comput., Vol. 23, No. 2, pp. 517–541, 2001.
- [16] 西田晃. 大規模対称固有値問題への前処理付共役勾配法の適用と評価. 日本応用数理学会年会予稿集, pp. 326–327, 2003.

- [17] H. Hasegawa. Utilizing the Quadruple-Precision Floating-Point Arithmetic Operation for the Krylov Subspace Methods. the 8th SIAM Conference on Applied Linear Algebra, 2003.
- [18] D. H. Bailey. A fortran-90 double-double library. <http://www.nersc.gov/~dhbailey/mpdist/mpdist.html>.
- [19] 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.
- [20] T. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, vol.18 pp. 224–242, 1971.
- [21] D. E. Knuth. The Art of Computer Programming: Seminumerical Algorithms, vol.2. Addison-Wesley, 1969.
- [22] D. H. Bailey. High-Precision Floating-Point Arithmetic in Scientific Computation. *Computing in Science and Engineering*, Volume 7, Issue 3, pp. 54–61, IEEE, 2005.
- [23] Intel Fortran Compiler User’s Guide Vol I.
- [24] 小武守恒, 藤井昭宏, 長谷川秀彦, 西田晃. 反復法ライブラリ向け 4 倍精度演算の実装と SSE2 を用いた高速化. *情報処理学会論文誌「コンピューティングシステム」*, Vol. 1, No. 1, pp. 73–84, 2008.
- [25] R. Barrett, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, 1994.
- [26] Z. Bai, et al. Templates for the Solution of Algebraic Eigenvalue Problems. SIAM, 2000.
- [27] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations, version 2, June 1994. <http://www.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html>.
- [28] S. Balay, et al. PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratory, August 2004.
- [29] R. S. Tuminaro, et al. Official Aztec user’s guide, version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, November 1999.
- [30] R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK Users’ Guide: Solution of Large-scale Eigenvalue Problems with implicitly-restarted Arnoldi Methods. SIAM, 1998.
- [31] R. Bramley and X. Wang. SPLIB: A library of iterative methods for sparse linear system. Technical report, Indiana University–Bloomington, 1995.
- [32] Matrix Market. <http://math.nist.gov/MatrixMarket>.

A ファイルフォーマット

本節では、本ライブラリで利用できるファイルフォーマットについて述べる。

A.1 拡張 MatrixMarket フォーマット

MatrixMarket フォーマット [32] は、ベクトルデータを格納できない。本ライブラリはベクトルを格納できるように拡張している。 $M \times N$ 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。ファイル構造は以下ようになる。

```
%%MatrixMarket matrix coordinate real general <-- ヘッダー
%
%                                <--+
%                                | 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
. . .                          | インデックスは 1-base
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 に対する拡張 MatrixMarket ファイルは以下ようになる。

$$A = \begin{pmatrix} 2 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & 2 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \quad (\text{A.1})$$

```
%%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 フォーマット

Harwell-Boeing フォーマットは CCS 形式で行列を入出力する。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。ファイル構造は以下になる。

```

1 行目 (A72,A8)
  1 - 72 Title
  73 - 80 Key
2 行目 (5I14)
  1 - 14 ヘッダーを除く総行数
  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
3 行目 (A3,11X,4I14)
  1 - 3 行列の種類
    1 列目: R Real matrix
          C Complex matrix (非サポート)
          P Pattern only (非サポート)
    2 列目: S Symmetric
          U Unsymmetric
          H Hermitian (非サポート)
          Z Skew symmetric (非サポート)
          R Rectangular (非サポート)
    3 列目: A Assembled
          E Elemental matrices (非サポート)

  4 - 14 空白
  15 - 28 行数
  29 - 42 列数
  43 - 56 非零要素数
  57 - 70 0
4 行目 (2A16,2A20)
  1 - 16 ptr のフォーマット
  17 - 32 index のフォーマット
  33 - 52 value のフォーマット
  53 - 72 右辺のフォーマット
5 行目 (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 行列と同じフォーマット (非サポート)
  2 初期値が与えられているならば G
  3 解が与えられているならば 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 Harwell-Boeing ファイルは以下になる。

```

1-----10-----20-----30-----40-----50-----60-----70-----80
Harwell-Boeing format sample
                        8          1          4          2          Lis
RUA                      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

```

```

2.0000000000000000E+00  1.0000000000000000E+00  1.0000000000000000E+00
2.0000000000000000E+00  1.0000000000000000E+00  1.0000000000000000E+00
2.0000000000000000E+00  1.0000000000000000E+00  1.0000000000000000E+00
2.0000000000000000E+00
0.0000000000000000E+00  1.0000000000000000E+00  2.0000000000000000E+00
3.0000000000000000E+00

```

A.3 拡張 MatrixMarket フォーマット (ベクトル用)

MatrixMarket フォーマット [32] でベクトルデータを格納できるように拡張している。次数 N のベクトル $b = (b_i)$ に対して $b_i = B(I)$ とする。ファイル構造は以下のようになる。

```

%%MatrixMarket vector coordinate real general  <-- ヘッダー
%
%                                     | 0 行以上のコメント行
%
%                                     <--+
N                                     <-- 行数
I1 B(I1)                               <--+
I2 B(I2)                               | 行番号 値
. . .                                 | インデックスは 1-base
IN B(IN)                              <--+

```

(A.1) 式のベクトル b に対する拡張 MatrixMarket ファイルは以下のようになる。

```

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

```

A.4 PLAIN フォーマット (ベクトル用)

PLAIN フォーマットはベクトルの値を始めから順番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。ファイル構造は以下のようになる。

```

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

```

(A.1) 式のベクトル b に対する PLAIN ファイルは以下のようになる。

```

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

```