NISTIR 5861

SparseLib++

SparseLib++ v. 1.5

Sparse Matrix Class Library

Reference Guide

Roldan Pozo
Karin A. Remington
U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Applied and Computational Mathematics Division
Gaithersburg, MD 20899

Andrew Lumsdaine
University of Notre Dame

NIST

NISTIR 5861

SparseLib++

SparseLib++ v. 1.5

Sparse Matrix Class Library

Reference Guide

Roldan Pozo
Karin A. Remington
U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Applied and Computational Mathematics Division
Gaithersburg, MD 20899

Andrew Lumsdaine
University of Notre Dame

June 1996

U.S. DEPARTMENT OF COMMERCE
Michael Kantor, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

Contents

1 About SparseLib++

SparseLib++ is a C++ class library for efficient sparse matrix computations across various computational platforms. The software package consists of matrix objects representing several sparse storage formats currently in use (in this release: compressed row, compressed column and coordinate formats), providing basic functionality for managing sparse matrices, together with efficient kernel mathematical operations (e.g. sparse matrix-vector multiply). The Sparse BLAS Toolkit [1] is used to enhance portability and performance across a wide range of computer architectures. Included in the package are various preconditioners commonly used in iterative solvers for linear systems of equations. The focus here is on computational support for iterative methods (see IML++[5]), but the sparse matrix objects presented here can be used in their own right.

The goal of SparseLib++ is to provide the ability to develop and experiment with numerical linear algebra algorithms through the separation of the internal details of a sparse matrix representation from the code that uses it. With this object-oriented approach, the need for separate hand-coded numerical linear algebra routines for each sparse matrix type is reduced.

The SparseLib++ library provides

- double-precision sparse matrix classes for **compressed column**, **compressed row**, and **coordinate** storage formats
- access to matrix elements of any sparse matrix type with the conventional notation `A(i,j)`
- conversion between any two sparse matrix types through simple matrix assignment (e.g. `A = B`)
- computational kernels based on the Sparse BLAS Toolkit interface [1] for maximum efficiency and portability across various hardware platforms
- basic preconditioners useful in iterative methods: incomplete LU (ILU), incomplete cholesky (ICP), and diagonal scaling
- the ability to read and write Harwell-Boeing formatted files and simple text files for matrix input/output
- easy integration with generic Fortran, C arrays together with user-defined C++ matrix packages

2 Sparse Matrix Representations

In the following subsections we describe the underlying data structures for the sparse matrix classes currently supported in SparseLib++. Although the object-oriented paradigm encourages hiding such issues, we have included this section to assist the user in integrating SparseLib++ with Fortran libraries and in matching a storage scheme with their specific problem structure. The SparseLib++ classes use a 0–based indexing internal representation (as opposed to the 1–based indexing representations commonly associated with Fortran, see section 5.1). This should be kept in mind while reading the following subsections.

SparseLib++ classes also use the MV++ matrix/vector classes as internal building blocks. See the *MV++ Reference Manual*[6] for details.

The current version (1.5) supports only double-precision matrices, although it is fairly straightforward to generate `complex` or other user-defined data type sparse matrices from the SparseLib++ source code. (We provide scripts to generate these.) Future releases of SparseLib++ will incorporate fully templated type parameters as template facilities mature in production level C++ compilers.

2.1 Data Storage Formats

To illustrate the various storage formats, we will use the nonsymmetric matrix:

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix} \tag{1}$$

For symmetric matrices, one can store only the upper (or lower) triangular portion of the matrix. The savings in storage is traded for additional bookkeeping. The current release of SparseLib++ does not support symmetric storage, but future releases will incorporate this space-economic feature.

2.2 Coordinate Storage

The most straightforward scheme to denote a sparse matrix simply records each nonzero entry together with its row and column index. Three data structures are used: a `val()` array to hold the floating point values, and row and column index arrays, `row_ind()` and `col_ind()`. The convention is that for each $k \in \{1, \ldots, nz\}$, The value `val(k)` occurs at position `(row_ind(k), col_ind(k))`.

The matrix `A` in (1) could thus be stored with the following arrays:

`val()`	1	2	3	4	5	6	7	8	9	10	11	12
`row_ind()`	0	0	0	1	1	1	2	2	2	3	4	4
`col_ind()`	0	1	4	0	1	2	1	2	4	3	0	4

One should note that the ordering of matrix elements in this format is not fixed, and so the given storage representation for the matrix is not unique. See section 5.2 for details.

2.3 Compressed Row Storage

The compressed row storage format views nonzero elements in each row as a sparse vector, storing pointers to the first element in each row in `row_ptr()`, and nonzero values and their associated column indices in the arrays `val()` and `col_ind()`. An additional element is appended to the `row_ptr()` array specifying the number of nonzero array elements.

The matrix `A` in (1) can be stored in compressed row format with the following arrays:

`row_ptr()`	0	3	6	9	10	12						
`val()`	1	2	3	4	5	6	7	8	9	10	11	12
`col_ind()`	0	1	4	0	1	2	1	2	4	3	0	4

2.4 Compressed Column Storage

The compressed column storage format parallels compressed row storage, but with the roles of rows and columns reversed. The appropriate array entries for the matrix `A` in (1) are as follows.

`col_ptr()`	0	3	6	8	9	12						
`val()`	1	4	11	2	5	7	6	8	10	3	9	12
`col_ind()`	0	1	4	0	1	2	1	2	3	0	2	4

2.5 Creating Sparse Matrices from C Arrays

SparseLib++ sparse matrices can easily be constructed from individual C/C++ vectors. For example, the 5x5 matrix `A` in (1) could be created by:

```
double val[12] = {1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,11.,12.};
int colind[12] = {0, 1, 4, 0, 1, 2, 1, 2, 4,  3,  0,  4 };
int rowptr[6]  = {0,       3,       6,    9, 10,     12 };

CompRow_Mat_double R(5,5,12,val,rowptr,colind);
```

See the class man pages in section A for details of the various constructor parameter lists for each sparse matrix class.

3 Sparse Matrix Operations

3.1 Matrix-Vector Multiplication via Sparse BLAS

One of the core operations in numerical linear algebra is the matrix-vector multiply, which, through operator overloading, can be called in SparseLib++ with the simple `A*x` notation. Since our goal

```
 1        1  1.0000000000000000e+000
 3        1  1.6065109611192374e+000
10        1  1.1329999260086812e+000
11        1  5.5417156097831336e-001
13        1  1.4999424800772864e-001
 2        2  1.0000000000000000e+000
 3        3  1.0000000000000000e+000
13        3 -7.0499387778693664e-002
40        3 -4.0513831677360480e-001
```

Figure 1: An example of a sparse matrix text format: <i>, <j>, <val>.

in creating this class library was to allow not only readable and reusable, but also *efficient* numerical linear algebra routines, we have provided matrix-vector multiplication operations which use the proposed Fortran Sparse BLAS Toolkit [1]. The Toolkit provides a uniform interface to kernel linear algebra routines. With vendor support, using this interface should allow for efficient implementations across platforms by simply linking with appropriate machine-specific BLAS libraries. The current SparseLib++ implementation provides its own sparse BLAS routines which may be used when machine-tuned versions are not available.

3.2 Preconditioners

In solving linear systems $Ax = b$ using iterative techniques, it is often advantageous to precondition the coefficient matrix A to improve convergence. A preconditioner M for A needs to do at most two things: solve the system $Mx = y$, or $M^T x = y$. Common preconditioners are Jacobi preconditioners, where $M = \text{diag}(A)$, or incomplete factorizations (LU, or Cholesky) of A.

In SparseLib++ we provide preconditioner class structures to serve this purpose, and these preconditioners can be used with, for example, the IML++ Iterative Methods Library (see [5]). The current library contains the following preconditioners: `DiagPreconditioner()`, for diagonal preconditioning, `ICPreconditioner()`, for incomplete Cholesky preconditioning, and `ILUPreconditioner()`, for incomplete LU preconditioning. Each preconditioner provides `solve()` and `transpose-solve()` functionality, so they can be used interchangeably in the same base iterative method code if it is templated for a preconditioner. For details on preconditioner construction and use, see the man pages in appendix A.

4 File I/O

To enable the use of SparseLib++ with previously generated sparse matrices, and to archive newly created sparse matrices for later use, the SparseLib++ library contains several functions for reading from and writing to plain text files and Harwell-Boeing formatted files. (For a detailed description of the Harwell-Boeing format, see [4].) Note that although the internal storage scheme indexing in SparseLib++ is 0–based, file I/O is performed with the 1–based indexing convention to provide compatibility with external Fortran libraries and other packages treating sparse matrices.

```
1unsymmetric matrix from  pores                                         pores_1
            59             2             12             45             0
rua                       30             30            180             0
(16i5)          (16i5)          (4d20.10)
    1    7   13   21   27   35   41   49   53   59   63   71   77   87   91  101
  105  115  119  127  131  137  140  148  151  159  162  170  173  179  181
    1    2    3    4   11   12    1    2    3    4   11   12    1    2    3    4
    5    6   13   14    3    4    5    6   13   14    3    4    5    6    7    8
   15   16    5    6    7    8   15   16    5    6    7    8    9   10   17   18
    7    8   10   18    7    8    9   10   19   20    9   10   19   20    1    2
   11   12   13   14   21   22   11   12   13   14   21   22    3    4   11   12
   13   14   15   16   23   24   13   14   16   24    5    6   13   14   15   16
   17   18   25   26   15   16   18   26    7    8   15   16   17   18   19   20
   27   28   17   18   20   28    9   10   17   18   19   20   29   30   19   20
   29   30   11   12   21   22   23   24   21   22   24   13   14   21   22   23
   24   25   26   23   24   26   15   16   23   24   25   26   27   28   25   26
   28   17   18   25   26   27   28   29   30   27   28   30   19   20   27   28
   29   30   29   30
  -0.9481011349e+03   -0.7178501646e+07    0.4731272996e+01    0.3574261854e+05
   0.9462545992e+03    0.7134130875e+07    0.2334969309e+05   -0.2461341087e+08
  -0.3005164596e+04    0.1293434629e+08   -0.3680715121e+04    0.6149543185e+07
   0.4731272996e+01    0.3567021095e+05   -0.3120860678e+04   -0.3250082045e+07
   0.1552207555e+02    0.1628780334e+05    0.3104415110e+04    0.3191488210e+07
```

Figure 2: A fragment of an input file from the Harwell-Boeing Sparse Matrix Collection.

4.1 Reading from and Writing to a Plain Text File

```
CompCol_Mat_double A;                  // or CompRow_Mat_double, or Coord_Mat_double
readtxtfile_mat("input_file", A);     // read from file
cout >> A;                            // write to standard out
writetxtfile_mat("output_file", A);   // write to file
```

Using standard output (`cout`), sparse matrices are written in a plain text coordinate format, one nonzero element per line, with each line containing an integer row index, an integer column index and a double value, with spaces separating the three fields. To ensure that the row and column dimensions, M and N, of the matrix can be determined implicitly from written information, we include in the output the `A(M-1,N-1)` matrix element, even if it falls outside of the sparsity pattern. (This convention is used for sparse matrix saves and loads in `MATLAB`.) The functions `readtxtfile_mat()` and `writetxtfile_mat()` provide file access for reading and writing sparse matrix information in this plain coordinate format.

4.2 Reading a Harwell-Boeing Formatted File

```
void readHB_info(const char *filename, int &M, int &N, int &nz, int &nrhs,
                 int verbose = 0)

void readHB_header(FILE *in_file, char *Title, char *Key, char *mat_type,
                   int &Nrow, int &Ncol, int &Nnzero, int &Nrhs, char *Ptrfmt,
                   char *Indfmt, char *Valfmt, char *Rhsfmt, int &Ptrcrd,
                   int &Indcrd, int &Valcrd, int &Rhscrd, char *Rhstype);
```

The `readHB_info` function opens and reads the numerical header information from the specified Harwell-Boeing file and returns the number of rows and columns in the stored matrix (`M` and `N`), the number of nonzeros in the matrix (`nz`), and the number of right-hand-sides stored along with the matrix (`nrhs`). The optional `verbose` parameter, if set to 1, sends to standard output more detailed information from the header, including title, etc. To retrieve and save to variables all descriptive header information from the file (such as "title" or "key'), the `readHB_header` is available.

```
void readHB_mat(const char *filename, Coord_Mat_double &A)

void readHB_mat(const char *filename, CompCol_Mat_double &A)

void readHB_mat(const char *filename, CompRow_Mat_double &A)
```

The `readHB_mat` function opens and reads the specified file, interpreting its contents as a sparse matrix stored in the Harwell/Boeing standard format and creating a sparse matrix object of the type indicated in the calling sequence.

```
void readHB_rhs(const char *filename, MV_Vector_double &b, int j=0)

void readHB_rhs(const char *filename, MV_ColMat_double &B)
```

The first `readHB_rhs` function opens and reads the specified file, returning a right-hand-side vector `b`. If the file provides a matrix of right-hand-sides, (that is, a sequence of right-hand-side vectors), the optional argument `j` can be used to indicate which right-hand-side is desired. The default reads the first stored right-hand-side (the `0`th column of the right-hand-side matrix). The second form of the `readHB_rhs` function is used to read in an entire multiple right-hand-side matrix, assigning it to `B`.

4.3 Writing a Sparse Matrix in Harwell-Boeing Format

```
void writeHB(const char *filename, const Coord_Mat_double A,
          const char * Title, const char * Key )
```

```
void writeHB(const char *filename, const CompRow_Mat_double A,
             const char * Title, const char * Key )

void writeHB(const char *filename, const CompCol_Mat_double A,
             const char * Title, const char * Key )

void writeHB(const char *filename, const Coord_Mat_double A)

void writeHB(const char *filename, const CompRow_Mat_double A)

void writeHB(const char *filename, const CompCol_Mat_double A)
```

The `writeHB` function opens the named file and writes the specified matrix to that file in Harwell-Boeing format. If the `Title` and `Key` arguments are not supplied, default values are used to indicate that SparseLib++ generated the file.

5 Programming Considerations

5.1 Integrating with Fortran

Since SparseLib++ matrices, like C and C++ arrays, have 0–based indexing, some care must be used to ensure compatibility with Fortran subroutines. Note that this is an issue only with sparse matrices, since the specific row/column index values are part of the **data**.

There are several ways to handle this issue: (1) one could create a copy of the matrix with each row and/or column index incremented by one, (2) make SparseLib++ matrices explicitly 1-based (or use arbitrary bases), or (3) modify the 0-based arguments to 1-based subroutines.

Solution (1) is easy to implement but rather expensive in practice, requiring an extra copy of the matrix as well as a $O(nz)$ algorithm to update the indices. Solution (2) seems reasonable, particularly for Fortran programmers, but makes SparseLib++ incompatible with native C arrays. There are several other inconsistencies brought up by violating the basic indexing scheme of C. Solution (3) can sometimes be used, but is not a universal solution. For example, the Fortran Sparse BLAS matrix-vector multiply routine ($y \leftarrow \alpha Ax + \beta y$) for a Coordinate storage sparse matrix and 0-based C/C++ array can be called as:

```
    DCOOMM_("N", M, N, 1, alpha, A, &x(1), ldb, beta, &y(1), N,
 work, iwork);
```

Similarly, for a more general matrix-matrix multiply , we can write ($C \leftarrow \alpha AB + \beta C$)

```
   DCOOMM_("N", M, N, K, alpha, A, &B(i,j)+ 1, ldb, beta, &C(k,l)+1, ldc,
```

```
work, iwork);
```

This scheme makes assumptions about how the 0-based parameters are used internally, and therefore its correctness cannot be guaranteed.

Note, also, that an expression such as "`&x(1)`" assumes a vector has at least two elements. If this causes an error when bounds checking is turned on, the equivalent expression "`&x[0]+1`" can be used to circumvent the bounds checking in this case.

5.2 Ordering of Sparse Matrix Elements

Many of the sparse matrix formats do not require a unique ordering of their elements. Most obvious in this category is coordinate storage, but other (more structured) formats share this ambiguity. The compressed row storage format, for example, assumes no explicit ordering among the elements in each row, but there are varying levels of ordering which may be present, and advantages from each level. No assumed ordering is the most general; ordering diagonal elements from each row first within each row is convenient for scaling and Jacobi preconditioners; ordering within rows by column index is probably the most natural and provides for most efficient addressing. Although the current release of SparseLib++ does not assume ordering, future releases will be equipped to order the elements to suit user requirements.

6 References

[1] S. Carney, M. A. Heroux, G. Li, and K. Wu, *A Revised Proposal for a Sparse BLAS Toolkit*, Army High Performance Computing Research Center Technical Report 94-034, June 1994.

[2] J. J. Dongarra, R. Pozo, and D. Walker, *LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra*, Proceedings of Supercomputing '93, Portland, Oregon, November 1993.

[3] J. J. Dongarra, A. Lumsdaine, R. Pozo, K. A. Remington, *A Sparse Matrix Library in C++ for High Performance Architectures,* Proceedings of the Second Annual Object-Oriented Numerics Conference, 1994, pp. 214-218.

[4] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Soft., 15, 1, pp. 1–14, 1989.

[5] J. J. Dongarra, A. Lumsdaine, R. Pozo, and K. A. Remington, *IML++ Iterative Methods Library Reference Guide*, `http://gams.nist.gov/acmd/Staff/RPozo/sparselib++.html`, 1994.

[6] R. Pozo, *MV++ Matrix / Vector Classes Reference Guide*, 1994. Available via anonymous ftp from `gams.nist.gov:~ftp/pub/pozo/src`.

A Class Man Pages

Name

Coord_Mat_double

Declaration

#include "coord.h"
class **Coord_Mat_double**

Description

A coordinate formatted sparse matrix class. Access to matrix elements as `A(i,j)` are 0–based (i.e. `A(0,0)` is the first element).

The nonzero matrix elements and their indices are stored in three vectors, with `val()` holding the element values, `row_ind()` holding the row indices, and `col_ind()` holding the column indices. For example, a Coord_Mat_double object specifying the matrix

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

may have the following vectors in internal storage:

`val()`	1	2	3	4	5	6	7	8	9	10	11	12
`row_ind()`	0	0	0	1	1	1	2	2	2	3	4	4
`col_ind()`	0	1	4	0	1	2	1	2	4	3	0	4

Constructors/Destructors

Coord_Mat_double ( )

Construct a null `0` × `0` matrix.

Coord_Mat_double (const Coord_Mat_double &C)

Construct a copy of the coordinate matrix `C`.

Coord_Mat_double (const CompCol_Mat_double &C)

Construct a coordinate matrix from a given compressed column representation.

Coord_Mat_double (const CompRow_Mat_double &C)

Construct a coordinate matrix from a given compressed row representation.

Coord_Mat_double (int M, int N, int nz, double *val, int *r, int *c)

Construct a coordinate matrix of size `M` × `N`, with `nz` nonzeros, using the values given in `val[]`, and row and column indices given in `r[]` and `c[]`. The vectors `r` and `c` are assumed to be **0–based**.

~Coord_Mat_double ( )

Matrix destructor.

Access and Information

double **operator** (int i, int j)

Return $A(i, j)$. (Returns zero if matrix element not found in sparse structure.)

double& set (int i, int j)

Assign $A(i, j)$ a value. Reports an error to stderr and exits the program if the assignment violates the sparsity structure of A (i.e. causes fill-in). For dynamically growing a sparse matrix, use an appropriate class.

double& val (int i)

Return the `i`th element of the nonzero value storage vector.

int row_ind (int i)

Return the row index of the element stored in `val(i)`.

int col_ind (int i)

Return the column index of the element stored in `val(i)`.

int dim (int i)

Return the size of the matrix along dimension `i`.

int size (int i)

Return the size of the matrix along dimension `i`. (Same as `dim()`.)

int NumNonzeros ( )

Return the number of nonzeros in the matrix.

Standard Output

friend ostream& operator<< (ostream &os, const Coord_Mat_double &C)

Print nonzero matrix elements one per line in the format $<i>$ $< j >$ $< val >$.

See also

CompRow_Mat_double, CompCol_Mat_double, MV_Vector (MV++), SparseLib++ File I/O

Name

CompRow_Mat_double

Declaration

#include "comprow.h"
class **CompRow_Mat_double**

Description

A compressed row formatted sparse matrix class. Access to matrix elements as `A(i,j)` are 0–based (i.e. `A(0,0)` is the first element).

The nonzero elements and index information are stored in three vectors, with `val()` holding the values of the nonzero elements, `row_ptr()` holding pointers to the first element in each row, and `col_ind()` holding a column index for each of the elements in `val()`. An additional element is appended to the `row_ptr()` array specifying the number of nonzero array elements. For example, a CompRow_Mat_double object specifying the matrix

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

may have the following vectors in internal storage:

`row_ptr()`	0	3	6	9	10	12						
`val()`	1	2	3	4	5	6	7	8	9	10	11	12
`col_ind()`	0	1	4	0	1	2	1	2	4	3	0	4

Constructors/Destructors

CompRow_Mat_double ( )

Construct a null `0` × `0` matrix.

CompRow_Mat_double (const CompRow_Mat_double &R)

Create a copy of the compressed row matrix `R`.

CompRow_Mat_double (const CompCoord_Mat_double &R)

Construct a compressed row matrix from a given coordinate representation.

CompRow_Mat_double (const CompCol_Mat_double &R)

Construct a compressed row matrix from a given compressed column representation.

CompRow_Mat_double (int M, int N, int nz, double *val, int *r, int *c)

Construct a compressed row matrix of size `M` × `N`, with `nz` nonzeros, using the values given in `val[]`, and row pointers and column indices given in `r[]` and `c[]`. The vectors `r` and `c` are assumed to be **0–based**.

~CompRow_Mat_double ( )

Matrix destructor.

Access and Information

double **operator** (int i, int j)

Return $A(i,j)$. (Returns zero if matrix element not found in sparse structure.)

double& set (int i, int j)

Assign $A(i,j)$ a value. Reports an error to stderr and exits the program if the assignment violates the sparsity structure of A (i.e. causes fill-in). For dynamically growing a sparse matrix, use an appropriate class.

double& val (int i)

Return the `ith` element of the nonzero value storage vector.

int row_ptr (int i)

Return the row pointer associated with row `i`.

int col_ind (int j)

Return the column index of the element stored in `val(j)`.

int dim (int i)

Return the size of the matrix along dimension `i`.

int size (int i)

Return the size of the matrix along dimension `i`. (Same as `dim()`.)

int NumNonzeros ( )

return the number of nonzeros in the matrix.

Standard Output

friend **ostream& operator<<** (ostream &os,
const CompRow_Mat_double &R)

Print matrix elements one per line in the format $<i> < j > < val >$.

See also CompCol_Mat_double, Coord_Mat_double, MV_Vector (MV++),
SparseLib++ File I/O

Name

CompCol_Mat_double

Declaration

#include "compcol.h"
class **CompCol_Mat_double**

Description

A compressed column formatted sparse matrix class. Access to matrix elements as `A(i,j)` are 0–based (i.e. `A(0,0)` is the first element).

The nonzero elements and index information are stored in three vectors: `val()` holding the values of the nonzero elements, `col_ptr()` holding pointers to the first element in each column, and `row_ind()` holding a row index for each of the elements in `val()`. An additional element is appended to the `col_ptr()` array specifying the number of nonzero array elements. For example, a CompCol_Mat_double object specifying the matrix

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

may have the following vectors in internal storage:

`col_ptr()`	0	3	6	8	9	12						
`val()`	1	4	11	2	5	7	6	8	10	3	9	12
`row_ind()`	0	1	4	0	1	2	1	2	3	0	2	4

Constructors/Destructors

CompCol_Mat_double ( )

Construct a null `0` × `0` matrix.

CompCol_Mat_double (const CompCol_Mat_double &C)

Create a copy of the compressed column matrix `C`.

CompCol_Mat_double (const CompCoord_Mat_double &C)

Construct a compressed column matrix from a given coordinate representation.

CompCol_Mat_double (const CompRow_Mat_double &C)

Construct a compressed column matrix from a given compressed row representation.

CompCol_Mat_double (int M, int N, int nz, double *val, int *r, int *c)

Construct a compressed column matrix of size `M` × `N`, with `nz` nonzeros, using the values given in `val[]`, and row indices and column pointers given in `r[]` and `c[]`. The vectors `r` and `c` are assumed to be **0–based**.

~CompCol_Mat_double ( )

Matrix destructor.

Access and Information

double **operator** (int i, int j)

Return $A(i,j)$. (Returns zero if matrix element not found in sparse structure.)

double& set (int i, int j)

Assign $A(i,j)$ a value. Reports an error to stderr and exits the program if the assignment violates the sparsity structure of A (i.e. causes fill-in). For dynamically growing a sparse matrix, use an appropriate class.

double& val (int i)

Return the `i`th element of the nonzero value storage vector.

int row_ind (int i)

Return the row index of the element stored in `val(i)`.

int col_ptr (int j)

Return the column pointer associated with column `j`.

int dim (int i)

Return the size of the matrix along dimension `i`.

int size (int i)

Return the size of the matrix along dimension `i`. (Same as `dim()`.)

int NumNonzeros ( )

return the number of nonzeros in the matrix.

Standard Output

friend ostream& operator<< (ostream &os,
const CompCol_Mat_double &C)

Print nonzero matrix elements to standard output, one per line, in the format $<i>$ $<j>$ $<val>$.

See also CompRow_Mat_double, Coord_Mat_double, MV_Vector (MV++), SparseLib++ File I/O

Name

DiagPreconditioner — Diagonal Preconditioner Class

Declaration

#include "diagpre.h"

class **DiagPreconditioner**

Description

Implements diagonal preconditioning for use with IML++ iterative methods. Presently, implementations for SparseLib++ compressed row and compressed column matrix formats (*CompCol_Mat_double* and *CompRow_Mat_double*, respectively) have been provided. In general, users of IML++ will not need to access member functions of this class except to create an instance of a preconditioner.

Constructors/Destructors

DiagPreconditioner (const CompCol_Mat_double& A)

Construct a diagonal preconditioner from the matrix A.

DiagPreconditioner (const CompRow_Mat_double& A)

Construct a diagonal preconditioner from the matrix A.

~DiagPreconditioner (void)

Reclaim memory space.

Member Functions

Vector_double **solve** (const Vector_double& b)const

Perform the preconditioning, that is, return the solution of the linear system with the preconditioner and the vector b.

Vector_double **trans_solve** (const Vector_double& b)const

Perform the transpose preconditioning, that is, return the solution of the linear system with the transposed preconditioner and the vector b. For the diagonal preconditioner (which is trivially symmetric), this is the same as the **solve()** member function.

Example

For examples of the use of this class, see the examples provided with the descriptions of the IML++ iterative method functions.

See Also

SparseLib++

Name ICPreconditioner — Incomplete Cholesky Preconditioner

Declaration #include "icpre.h"

class **ICPreconditioner**

Description Implements incomplete Cholesky preconditioning for use with IML++ iterative methods. Presently, implementations for SparseLib++ compressed column and compressed row matrix formats (*CompCol_Mat_double* and *CompRow_Mat_double*, respectively) have been provided. In general, users of IML++ will not need to access member functions of this class except to create an instance of a preconditioner. The matrix A must be symmetric positive definite.

The present implementation of incomplete Cholesky factorization does not create any fill for structural zero elements (i.e., it is an implementation of IC(0)), but it does modify non-zero elements.

Constructors/Destructors

ICPreconditioner (const CompCol_Mat_double& A)

Construct an incomplete Cholesky preconditioner from the matrix A.

ICPreconditioner (const CompRow_Mat_double& A)

Construct an incomplete Cholesky preconditioner from the matrix A.

~**ICPreconditioner** (void)

Reclaim memory space.

Member Functions Vector_double **solve** (const Vector_double& b)const

Perform the preconditioning, that is, return the solution of the linear system with the preconditioner and the vector b.

Vector_double **trans_solve** (const Vector_double& b)const

Perform the transpose preconditioning, that is, return the solution of the linear system with the transposed preconditioner and the vector b. For the IC preconditioner (which is symmetric), this is the same as the **solve()** member function.

Example The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with CG. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file *test.dat*. An initial guess of 0 is made for the solution and the system is solved using CG and an incomplete Cholesky preconditioner.

```
#include <iostream.h>
#include <stdlib.h>
#include "cg.h"
#include "icpre.h"
#include "compcol1.h"
#include "iohb.h"
#include "vector.h"
#include "blas1.h"

int main()
{
  double tol = 1.e-6;
  int result, maxit = 150;

  CompCol_Mat_double A;                     // Create a matrix
  readHB("test.hb", A);                     // Read matrix data

  Vector_double x(A.dim(1), 0.0), b;        // Solution, rhs vectors
  readHB("test.hb", b);                     // Read rhs data

  ICPreconditioner D(A);       // IC preconditioner

  result = CG(A, x, b, D, maxit, tol);      // Solve system with CG

  cout << "CG flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;
  cout << x;

  return 0;
}
```

See Also

CG
SparseLib++

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

J. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

Name

ILUPreconditioner — Incomplete LU Preconditioner

Declaration

#include "ilupre.h"

class **CompCol_ILUPreconditioner**
class **CompRow_ILUPreconditioner**

Description

Implement incomplete LU preconditioning for use with IML++ iterative methods. Presently, implementations for SparseLib++ compressed row and compressed column matrix formats (*CompCol_Mat_double* and *CompRow_Mat_double*, respectively) have been provided. In general, users of IML++ will not need to access member functions of this class except to create an instance of a preconditioner.

This implementation of incomplete LU factorization does not create any fill for structural zero elements (i.e., it is an implementation of ILU(0)), but it does modify non-zero elements.

Constructors/Destructors

CompCol_ILUPreconditioner (const CompCol_Mat_double& A)

Construct an incomplete LU preconditioner from the matrix A.

~**CompCol_ILUPreconditioner** (void)

Reclaim memory space.

CompRow_ILUPreconditioner (const CompRow_Mat_double& A)

Construct an incomplete LU preconditioner from the matrix A.

~**CompRow_ILUPreconditioner** (void)

Reclaim memory space.

Member Functions

Vector_double **solve** (const Vector_double& b)const

Perform the preconditioning, that is, return the solution of the linear system with the preconditioner and the vector b.

Vector_double **trans_solve** (const Vector_double& b)const

Perform the transpose preconditioning, that is, return the solution of the linear system with the transposed preconditioner and the vector b.

Example

The following example program uses IML++ in conjunction with SparseLib++ to solve a linear system with **GMRES**. The program reads in a matrix and right-hand side stored in Harwell-Boeing format from the file *test.dat*. An initial guess of 0 is made for the solution and the system is solved using **GMRES** and an incomplete LU preconditioner.

```
#include <iostream.h>
#include <stdlib.h>
#include "gmres.h"
#include "ilupre.h"
#include "compcol1.h"
#include "iohb.h"
#include "vector.h"
#include "blas1.h"

int main()
{
  double tol = 1.e-6;
  int result, m = 32, maxit = 150;

  CompCol_Mat_double A;                         // Create a matrix
  readHB("test.hb", A);                         // Read matrix data

  Matrix_double H(m+1, m, 0.0);                 // H matrix
  Vector_double x(A.dim(1), 0.0), b;           // Solution, rhs vectors
  readHB("test.hb", b);                         // Read rhs data

  CompCol_ILUPreconditioner D(A);         // ILU preconditioner

  result = GMRES(A, x, b, D, H, m, maxit, tol); // Solve system with GMRES

  cout << "GMRES flag = " << result << endl;
  cout << "iterations performed: " << maxit << endl;
  cout << "tolerance achieved  : " << tol << endl;
  cout << x;

  return 0;
}
```

See Also

GMRES
SparseLib++

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

J. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.