UCSparseLib v.1.0 : A library to solve sparse linear systems and an interface module to SEMIYA simulator

Germán Larrazábal
Aurora Camacho
Pablo Guillen

July, 2002

Índice General

1 Introduction **3**

2 Data structure for dense or sparse matrices **4**

3 Modules **5**
3.1 MATRIX 6
3.2 METIS_INTERFACE 7
3.3 FACTORISATIONS 7
3.4 ITERATIVE 7
3.5 AMG 8
3.6 SCHUR 8
3.7 TOOLS 8
3.8 GENERATION 8

4 INTEVEP module **9**

5 Example **15**

6 Future work **15**

7 Conclusions **15**

1 Introduction

Researching on sparse matrix technique has become increasely complex, and this trend is likely to accentuate if only because of necessary growing to design efficient sparse matrix algorithms for modern supercomputers. While there are a number of packages and tools, for performing computations with small dense matrices, there is a lack of any similar tools or in fact, any general-purpose libraries for working with sparse matrices. Already a collection of a few basic programs to perform some elementary and common tasks may be very useful in reducing the typical time to implement and test sparse matrix algorithms. That a common set of routines shared among researchers does not yet exist for sparse matrix computations is rather surprising. Considering the constraing situation in dense matrix computations. The Linpack and Eispack packages developed in the 70's have been of tremendous help in several areas of scientific computing. One might speculate on the number of hours of programming efforts saved worlwide thanks to the widespread availability of these packages. In contrast, it is a frequently case that researchers in sparse matrix computations code their own subroutines for such things as converting the storage mode of a matrix or for reordering a matrix according to a certain permutation. One of the reasons for this situation might be the absence of any standard for sparse matrix computations. For instance, the number of different data structures used to store sparse matrices in various applications is staggering. For the same basic data structure there are often a large number of variations in use.

UCSparseLib v.1.0 is an ANSI C library to solve dense and sparse linear systems. The current version of the library includes the following main functionality:

Matrix Input/Output. Routines to read and write matrices using a simple format. Also, there are routines that transform a matrix in postscript format.

Matrix and Vector operations. Basic operations applied to vectors, Matrix-vector and Matrix-Matrix multiplication, triangular solvers, matrix reordering using METIS interface.

Direct solvers. Complete Factorizations (*Cholesky*, LDL^t and LU).

Iterative solvers. Gauss-Seidel, Jacobi, Conjugate Gradient, GMRES(m), BiCGstab and BiCG.

Preconditioners. Incomplete factorizations ($ICHOL$, $ILDL^t$, ILU, diagonal scaling).

Algebraic Multigrid. AMG with different setup phases: aggregations, red-black coloring and strong connect.

Utility Routines. Timers (wall-clock and CPU time) and memory manage-

ment.

UCSparseLib is easy to use and easy to cut up in pieces. It uses a nearly trivial design with only one external-visible structure. All the C routines in the library start with the prefix `UCSparseLib` and so do the name of the structures and preprocessor macros. Therefore, you should not have any problems using the library together with other libraries. The library is currently sequential.

This version has an interface module to SEMIYA simulator. SEMIYA is a porous fluid flow program developed in INTEVEP S.A. This simulator is developed in FORTRAN 90.

2 Data structure for dense or sparse matrices

One of the difficulties in dense or sparse matrices computations is the the variety of types of matrices that are found in practical applications. The purpose of each of these schemes is gaining efficiency both in terms of memory utilization and arithmetic operations. As a result, many different ways of storing dense and sparse matrices have been devised to take advantage of the structure of the matrices or the specificity of the problem from which they arise. One of the most common storage schemes in use today is Compressed Sparse Row (CSR) scheme. In this scheme all the nonzeros entries are stored row by row in an one-dimensional real array AN together with an array JA containing their column indices and a pointer array which contains the addresses in AN and JA of the beginning of each row. The order of the elements within each row does not matter. Also it is important because of its simplicity is the coordinated storage scheme in which the nonzero entries of AN are stored in any order together with their row and columns indices.

In UCSparseLib a matrix is an object called `TDMatrix`. A `TDMatrix M` is created using following routine:

```
TDMatrixCreate( TDMatrix *M, TMatFormat format, int rows, int cols );
```

where, `M` is a matrix, and it must be declared like `TDMatrix M`. `format` is a storage format of `M`. This format can be `MATRIX_DENSE_R, MATRIX_DENSE_C, MATRIX_SDENSE_R, MATRIX_SDENSE_C, MATRIX_DIAG, MATRIX_CSR, MATRIX_CSC, MATRIX_SCS_R, MATRIX_SCS_C`. And, `rows` and `cols` are the dimension of the matrix.

TDMatrix object is defined how:

```
typedef struct DMatrix_t *TDMatrix;
  struct DMatrix_t
  {
    TMatFormat format;  /* DENSE, CSR, CSC, DIAG */
    int    nn;          /* IA dimension */
```

```
    int    mm;          /* SDENSE_R, DENSE_R, SCSR or CSR -> nbcols;*/
                        /* SDENSE_C, DENSE_C, SCSC or CSC -> nbrows */
    size_t nnz;         /* AN/JA dimension */
    TDSparseVec *ia;    /* Point to the begin of each row/col */
    int         *ja;    /* Col/Row of each AN element */
    double      *an;    /* No null entries of the matrix */
    Tcomplex    *zan;   /* No null entries of the matrix complex case */
    double      *invd;  /* Inverse of the diagonal */
    Tcomplex    *zinvd; /* Inverse of the diagonal complex case */
    TLinkSparseVec *transpose; /* Linked list to access the transpose matrix */
    int   nblocks;     /* For internal debuging */
    int   share_ja;    /* TRUE if ja is shared (allocated by the user) */
    int   share_an;    /* TRUE if an is shared (allocated by the user) */
  };
```

An important component of the TDMatrix object is the TDSparseVec structure. This structure represent a sparse vector and it is:

```
typedef struct {
  int nz ;     /* no nulls elements */
  int diag;    /*index of the diagonal element */
  int *id;     /*col/row values */
  double *val; /* row/col elements */
} TDSparseVec;
```

To access the `i` row of the `TDmatrix M`, we use the following macro:

```
For_TDMatrix_Row( M, i, row, mode ) {
  .
  .
  .
}
```

where,
`row` is a `TDSparseVec` and `mode` can be: `ACCESS_READ, ACCESS_WRITE` and `ACCESS_RW`.

Also, we have macros to access columns or row/col if the user doesn't know the matrix format. The macros are: `For_TDMatrix_Col` and `For_TDMatrix_RC`, respectively.

3 Modules

In this section, we present a general overview of each module of UCSparseLib library. UCSparseLib has 8 modules.

3.1 MATRIX

In MATRIX module there are routines to create, manipulate and delete a TDMatrix object. The following routines are available:

1. TDMatrixCreate: Allocate space for a TDMatrix struct and initialise the object with null values.

2. TDMatrixInitFromOUT: Allocate space for a TDMatrix struct and initialise the object with the dense matrix defined an. The memory area pointed by an is defined as shared depending on the shared parameter. If the matrix is sparse, the rows/columns are shorted in increasing order, modifing the original position of each entry.

3. TDMatrixDelete: Free memory space used by a TDMatrix object. This space is freed depending on how it was allocated, i.e., in a single block or in multiple blocks. For the single block the space is freed if it is not shared with other application.

4. TDMatrixSetNbRC: If the matrix is a DENSE_R or CSR, set the number of columns, in the other case (DENSE_C or CSC) set the number of rows.

5. TDMatrixSetNbnz: Set the number of no nulls. It is the AN/JA dimension.

6. TDMatrixSetFormat: Set the type of the format.

7. TDMatrixSetInvDiag: Set the inverse diagonal in an array.

8. TDMatrixNbRows: Give the number of rows.

9. TDMatrixNbCols: Give the number of columns.

10. TDMatrixNbNz: Give the number of no nulls elements.

11. TDMatrixFormat: Give the format.

12. TDMatrixAllRowsNz: Put the number of the stored elements of each row in an array.

13. TDMatrixAllColsNz: Put the number of the stored elements of each column in an array.

14. TDMatrixInvd: Returns a pointer to the inverse of the diagonal.

15. TDMatrixDup: Allocate space for the TDMatrix and initialise the object copying the matrix.

16. TDMatrixIdentity: Sets the input matrix as the identity matrix.

17. TDMatrixPermute: permute the input matrix.

3.2 METIS_INTERFACE

This module is used for reordering and partitioning sparse matrices. UCSparseLib uses METIS package in these cases.

3.3 FACTORISATIONS

In this module there is a set of routines that can factorise a matrix using the LU, LDL^t, Cholesky, Gill-Murray and QR algorithms. The factorisations can be done within or without pivoting. Moreover, there is a routine to compute an incomplete LU, LDL^t or Choleski factorisations. Also, there are routines to solve different kinds of triangular systems. The permutations arrays could be NULL if there is not any permutation. Moreover, there are routines used after a QR factorisations.

3.4 ITERATIVE

In this module there are routines to use TDMatrix objects and vectors. Also, many iterative methods are implemented like: GMRES(m), Conjugate Gradient, Stabilised BiConjugate Gradiente and BiConjugate Gradient. These methods can be preconditioned. The preconditioners implemented are: incomplete factorisations and diagonal scaling. It module also contains the symmetric Lanczos method. Moreover, many relax methods have been implemented such as Weigh Jacobi, Gauss-Seidel an SOR.

3.5 AMG

In this module many Algebraic MultiGrid methods have been implemented such as methods based on aggregations, red-black coloring and strong connect between nodes. There have been implemented many solutions schemes like V_cicle, W_cicle and F_cicle. At this time, the AMG can be executed stand-alone, in future versions the AMG can be used as a preconditioner.

3.6 SCHUR

In this module there is a parallel iterative sparse solver based on the schur complement system.

3.7 TOOLS

In this module there are basic routines to integer and double precision vectors, memory management, timers, traces and routines to send and receive message using MPI or PVM.

3.8 GENERATION

In this module, an elliptical equation is discretizated and with this is generated only one matrix with no nulls elements, those are storaged and the same ones are used by other modules. This module is for generating proof matrices of different dimentions.

The following figure shows the structure of the library:

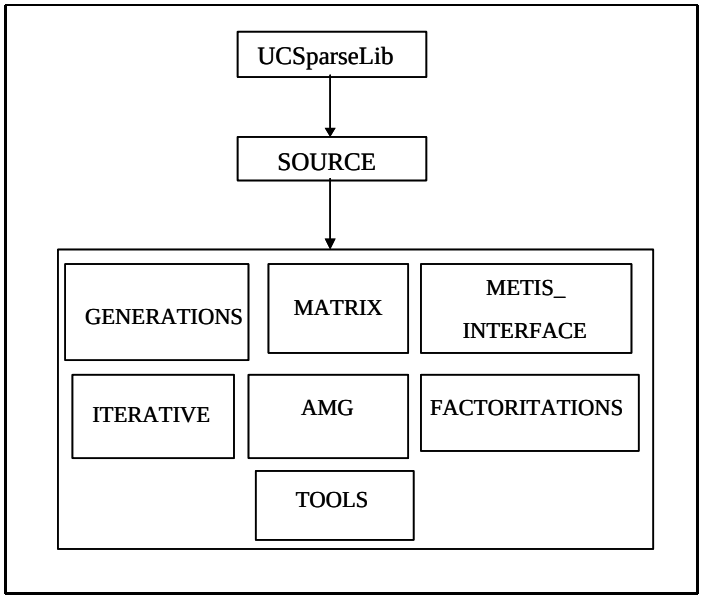

Figura 1: UCSparseLib v 1.0.

4 INTEVEP module

In this version, we have to develop an interface module to SEMIYA porous fluid flow simulator. This simulator has been developed in INTEVEP. This module uses a subroutine to change format storages due to SEMIYA has a diagonal storage. This subroutine is called `diag2TDMatrix`. Following, we present the prototype for it.

```
void diag2TDMatrix( double b1[],double b2 [], double b3[],
                double b4[],double b5[], double b6[],
                double b7[], int nx, int ny,int nz);
```

Where, `b1, ... ,b7` are the diagonal of a matrix, `nx` is `x` dimension, `ny` is `y` dimension, `nz` is `z` dimension.

Moreover, this module have two iterative preconditionated solvers: GMRES(m) and BiCGstab. We have to use two preconditioners: incomplete factorizations and SPAI. Following, we show the prototype for each subroutine.

```
void solver_gmres_( int iparam[], double dparam[], int nx, int ny,
                int nz, double b1[], double b2[], double b3[],
                double b4[],double b5[],double b6[], double b7[],
                double rhs[], double xx[] );
```

Where, `iparam` is a vector of integer parameters, `dparam` is a vector of double parameters, `nx` is `x` dimension, `ny` is `y` dimension, `nz` is `z` dimension, `b1,..,b7` are the diagonal of a matrix , `rhs` is a right hand side vector, `xx` is an output solution vector.

```
void solver_bicgstab_ ( int iparam[], double dparam[], int nx, int ny,
                  int nz, double b1[], double b2[],double double b3[],
                  double b4[], double b5[], double b6[], double b7[],
                  double b7[], double rhs[], double xx[]);
```

Where, `iparam` is a vector of integer parameters, `dparam` is a vector of double parameters, `nx` is `x` dimension, `ny` is `y` dimension, `nz` is `z` dimension, `b1,..,b7` are the diagonal of a matrix, `rhs` is a right hand side vector and `xx` is an output solution vector.

In the following algorithm we show the complete INTEVEP software module. Into SEMIYA Makefile file we have to link with UCSparseLib.a and to copy the intevep.o in this file.

```
#ifdef SOURCE

/* ---------------------------------- Include Files */

# include <stdio.h>
```

```
# include "Macros.h"
# include "TDMatrix.h"

# include "TDMatrix_ITERATIVE.h"
# include "intevep.h"
/*---------------------------------- Global Variables */

  static int control = 1;
  static TDMatrix AA;

static void diag2TDMatrix(double b1[],double b2[],double b3[],
                          doubleb4[],double b5[],double b6[], double b7[],
                          int nx,int ny, int nz )

{ /*---------------------------------- Local Variables */

  int          ii, jj, kk, ff, ll, cont, nx_x_ny, kk_x_nx_x_ny, jj_x_nx,
               nnz, pos, nbnodes;
  double       *wa;
  TDSparseVec  row;

/* --------------------------------- */ BEGIN( diag2TDMatrix );

  nx_x_ny = nx * ny;
  nbnodes = nx_x_ny * nz;

  if (control)
  {
    TDMatrixCreate( &AA, MATRIX_CSR, nbnodes, nbnodes );
  }
  ff = 0;
  jj = 0;
  kk = 0;
  ii = -1;
  for (ll = 0; ll < nbnodes; ll++)
  {
    ii++;
    if (ii == nx)
    {
      ii = 0;
      jj++;
    }
    if (jj == ny)
    {
      jj = 0;
```

```
  kk++;
}
if (control)
{
  nnz = 1;
  cont = 0;
  if (ii > 0)    nnz++;   /** case x-1 (b3) **/
  if (ii < nx-1) nnz++;   /** case x+1 (b5) **/
  if (jj > 0 )   nnz++;   /** case y-1 (b2) **/
  if (jj < ny-1) nnz++;   /** case y+1 (b6) **/
  if (kk > 0)    nnz++;   /** case z-1 (b1) **/
  if (kk < nz-1) nnz++;   /** case z+1 (b7) **/
}
For_TDMatrix_Row( AA, ff, row, ACCESS_WRITE )
{
  cont = 0;
  if (control)
  {
    row.nz  = nnz;
    row.id  = ALLOC( row.nz, int, "Row->Id" );
    row.val = ALLOC( row.nz, double, "Row->Val" );
    if (kk > 0 ) /** case z-1 (b1) **/
    {
      pos = ll - nx_x_ny;
      row.id[cont]  = pos;
      row.val[cont++] = b1[ff - nx_x_ny];
    }
    if (jj > 0)    /** case y-1 (b2) **/
    {
      pos = ll - nx;
      row.id[cont]  = pos;
      row.val[cont++] = b2[ff - nx];
    }
    if (ii > 0 )    /** case x-1 (b3) **/
    {
      pos = ll - 1;
      row.id[cont]  = pos;
      row.val[cont++] = b3[ff - 1];
    }
    /** case xyz (b4) (diagonal) **/
    row.id[cont] = ll;
    row.diag = cont;
    row.val[cont++] = b4[ff];
    if  (ii < nx-1 )/** case x+1 (b5) **/
    {
```

```
    pos = ll + 1;
    row.id[cont]  =  pos;
    row.val[cont++] = b5[ff];
  }
  if (jj < ny-1 )  /** case y+1 (b6) **/
  {
    pos = ll + nx;
    row.id[cont]  = pos;
    row.val[cont++] = b6[ff];
  }
  if (kk < nz-1 )  /** case z+1 (b7) **/
  {
    pos = ll + nx_x_ny;
    row.id[cont]  = pos;
    row.val[cont++] = b7[ff];
  }
}
else
{
  if (kk > 0 ) /** case z-1 (b1) **/
  {
    row.val[cont++] = b1[ff - nx_x_ny];
  }
  if (jj > 0)    /** case y-1 (b2) **/
  {
    row.val[cont++] = b2[ff - nx];
  }
  if (ii > 0 )    /** case x-1 (b3) **/
  {
    row.val[cont++] = b3[ff - 1];
  }
  /** case xyz (b4) (diagonal) **/
  row.val[cont++] = b4[ff];

  if  (ii < nx-1 )/** case x+1 (b5) **/
  {
    row.val[cont++] = b5[ff];
  }
  if (jj < ny-1 )  /** case y+1 (b6) **/
  {
    row.val[cont++] = b6[ff];
  }
  if (kk < nz-1 )  /** case z+1 (b7) **/
  {
    row.val[cont++] = b7[ff];
```

```
        }
      }
    }
    ff++;
  }
  wa = TDMatrixInvd( AA );
  for (ii= 0; ii< nbnodes; ii++)
  {
     For_TDMatrix_Row( AA, ii, row, ACCESS_READ )
     {
        wa[ii] = 1.0 / row.val[row.diag];
     }
  }
  if (control) control = 0;
/* ---------------------------------- */ END( diag2TDMatrix ); }

void solver_gmres_( int iparam[], double dparam[],
                    int *nbx,int*nby, int *nbz, double b1[], double b2[],
                    double b3[], double b4[], double b5[], double b6[],
                    double b7[], double rhs[], double xx[] )

{ /* ---------------------------------- Local Variables */

  int       ii, nbnodes, totiter, ierr, nx, ny, nz;
  double   *plot;
  TDprecond PM;

/* ---------------------------------- */ BEGIN( solver_gmres_ );

  nx = *nbx; ny = *nby; nz = *nbz;
  nbnodes = nx * ny * nz;
  if (control)
    AA = (TDMatrix) NULL;
  diag2TDMatrix( b1, b2, b3, b4, b5, b6, b7, nx, ny, nz);
  if (iparam[4]) /* trace */
    plot = ALLOC( iparam[1]+1, double, "plot" );
  else
    plot = (double *)NULL;
  computePrecond( iparam, dparam, AA, &PM );
  GMRES( AA, xx, rhs, &PM, iparam, dparam, plot, &totiter, &ierr );
  if (iparam[4])
  {
    for (ii= 0; ii <= totiter; ii++)
      printf(" %d  %.15lE\n", ii, plot[ii] );
  }
```

```
  if (iparam[4])
  FREE( plot );
  FREE( PM.L );
  FREE( PM.U );

/* ---------------------------------- */ END( solver_gmres_ ); }

void solver_bicgstab_( int iparam[], double dparam[], int *nbx,
                       int *nby, int *nbz, double b1[], double b2[],
                       double b3[], double b4[], double b5[], double b6[],
                       double b7[], double rhs[], double xx[] )

{ /* ---------------------------------- Local Variables */

  int       ii, nbnodes, totiter, ierr, nx, ny, nz;
  double   *plot;
  TDprecond PM;

/* ---------------------------------- */ BEGIN( solver_bicgstab_ );

  nx = *nbx; ny = *nby; nz = *nbz;
  nbnodes = nx * ny * nz;
  if (control)
    AA = (TDMatrix) NULL;
  diag2TDMatrix( b1, b2, b3, b4, b5, b6, b7, nx, ny, nz );
  if (iparam[4]) /* trace */
    plot = ALLOC( iparam[1]+1, double, "plot" );
  else
    plot = (double *)NULL;
  computePrecond( iparam, dparam, AA, &PM );
  BiCGstab( AA, xx, rhs, &PM, iparam, dparam, plot, &totiter, &ierr );
  if (iparam[4])
  {
    for (ii= 0; ii <= totiter; ii++)
      printf(" %d  %.15lE\n", ii, plot[ii] );
  }
  if (iparam[4])
  FREE( plot );
  FREE( PM.L );
  FREE( PM.U );

/* ---------------------------------- */ END( solver_bicgstab_ ); }

void free_matrix_( void )
```

```
 { /*---------------------------------*/BEGIN( free_matrix_ );

  TDMatrixDelete( AA );

/* --------------------------------- */ END( free_matrix_ ); }

#endif /* SOURCE */
```

5 Example

The figure 2 shows a code example that reads a matrix of a file previously generated by GENERATION module, then cholesky factorisation is applied to solve the linear system.

6 Future work

The next versions of the library will be developed SPAI preconditioners and the others iterative methods.

7 Conclusions

In this work we have presented a main features of a library to solve dense and sparse linear systems. One of the goals of the package is providing basic tools to facility the manipulations and computing on matrices dense and sparse. The library is a free software.

```
void main() {
 .
 .
 .
  /* Read input data */
  file = fopen( "int.dat", "r" );
  if (file == (FILE*) NULL)
    ERROR( "Data file erroneo" );
  /* Create TDMatrix object NULL*/
  AA = (TDMatrix) NULL;
  /* Read a matrix in CSR format from a file and return it */
  /* TDMatrix instance  */
  TDMatrixRead_Gen( file, &AA );
  fclose( file );
  /* obtain the rows number */
  nbrows = TDMatrixNbRows( AA );
  /* obtain the no nulls elements number */
  nnz = (int) TDMatrixNbNZ( AA );
  /* Allocate aproximate solution vector, solution vector and */
  /* right hand side */
  rhs = ALLOC( nbrows, double, "rhs" );
  sol = ALLOC( nbrows, double, "sol" );
  xx = ALLOC ( nbrows, double, "xx");
  /* Generated an artificial solution vector and rhs */
  TDMatrix_GenSolRhs( AA, rhs, sol );
  LL = (TDMatrix)NULL;
  DD = (TDMatrix)NULL;
  UU = (TDMatrix)NULL;
  /* Allocate the permute structure */
  TpermAlloc ( nbrows, nbrows, perm);
  TpermReset ( perm);
  /* Cholesky Factorizations */
  Factorisation( FAC_CHOL, 0.0, AA, &LL, &DD, &UU, perm );
  /* Solve triangular system  */
  TDMatrix_LLtsol( nbrows, LL, rhs, xx, perm );
  /* Test the aproximated solution */
  TDMatrix_TestSol ( AA, rhs, xx, sol);
  .
  .
  .
}
```

Figura 2: Cholesky Factorization and solving triangular System using UCSparseLib v1.0.