Introduction to Sparse Matrices In Scilab

Michael Baudin

November 2011

Résumé

The goal of this document is to present the management of sparse matrices in Scilab. We present the basic features of Scilab, which can create sparse matrices and can convert from and to dense matrices. We show how to solve sparse linear equations in Scilab, by using sparse LU decomposition and iterative methods. We present the sparse Cholesky decomposition. We present the functions from the UMFPACK and TAUCS modules. We briefly present the internal sparse API. We introduce to the Arnoldi package. We present the Matrix Market toolbox, which reads and writes sparse matrix files. We analyze how to solve the Poisson Partial Differential Equation with sparse matrices. We present the Imsls toolbox, a set of iterative solvers for sparse linear systems of equations.

Table des matières

1 Introduction

In numerical analysis, a sparse matrix is a matrix populated primarily with zeros[13]. Huge sparse matrices often appear in science or engineering when solving partial differential equations.

Scilab provides several features to manage sparse matrices and perform usual linear algebra operations on them. These operations includes all the basic linear algebra including addition, dot product, transpose and the matrix vector product. For these basic operations, we do not make any difference in a script managing a dense matrix and a script managing a sparse matrix.

1.1 Note on this document

This document is an open-source project. The LaTeX sources are available on the Scilab Forge :

`http://forge.scilab.org/index.php/p/docscisparse/`

The LaTeX sources are provided under the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License :

`http://creativecommons.org/licenses/by-sa/3.0`

The Scilab scripts are provided on the Forge, inside the project, under the `scripts` sub-directory. The scripts are available under the CeCiLL licence :

`http://www.cecill.info/licences/Licence_CeCILL_V2-en.txt`

1.2 Overview

In this section, we make an overview of the features provided by Scilab for sparse matrices.

The current set of tools available in Scilab for sparse matrices are the following :
- management of sparse matrices of arbitrary size,
- basic algebra on sparse matrices, including, sum, dot product, transpose, matrix-matrix product,
- sparse LU decomposition and resolution of linear equations from this decomposition based on the Sparse package written by Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli,

– sparse Cholesky decomposition and resolution of linear equations from this decomposition, based on a sparse Cholesky factorization package developed by Esmond Ng and Barry Peyton at ORNL and a multiple minimun-degree ordering package by Joseph Liu at University of Waterloo,
– iterative methods of linear systems of equations, including Preconditionned Conjugate Gradient (pcg), Generalized Minimum Residual method (gmres), Quasi Minimal Residual method with preconditionning (qmr),
– sparse LU decomposition and resolution of linear equations from this decomposition based on the UMFPACK library,
– sparse Cholesky decomposition and resolution of linear equations from this decomposition based on the TAUCS library,
– sparse eigenvalue computations, based on the Arpack library, using the Implicitly Restarted Arnoldi Method.

Scilab manages several file formats read and write sparse matrices.
– Scilab is able to read matrices in the Harwell-Boeing format, thanks to the ReadHBSparse function of the Umfpack module.
– The Matrix Market external module (available in ATOMS) provides functions to read and write matrices in the Matrix Market format.

2 Basic features

In this section, we present the basic sparse matrix features. In the first part, we review how to create a sparse matrix. Then we analyze how a sparse matrix is stored internally.

2.1 Creating sparse matrices

The figure 1 present several functions to create sparse matrices.

In the following session, we use the `sprand` function to create a 100×1000 sparse matrix with density 0.001. The density parameter makes so that the number of non-zero entries in the sparse matrix is approximately equal to $100 \cdot 1000 \cdot 0.001 = 100$. Then we use the `size` function to compute the size of the matrix. Finally, we compute the number of non-zero entries with the `nnz` function.

```
-->A=sprand(100,1000,0.001);
```

sp2adj	converts sparse matrix into adjacency form
speye	sparse identity matrix
spones	sparse matrix
sprand	sparse random matrix
spzeros	sparse zero matrix
full	sparse to full matrix conversion
sparse	sparse matrix definition
mtlb_sparse	convert sparse matrix
nnz	number of non zero entries in a matrix
spcompack	converts a compressed adjacency representation
spget	retrieves entries of sparse matrix

FIGURE 1 – Basic features for sparse matrices.

```
-->size(A)
 ans  =
    100.    1000.
-->nnz(A)
 ans  =
    100.
```

The `sparse` and `full` functions works as complementary functions. Indeed, the `sparse` function converts a full matrix into a sparse one, while the `full` function converts a sparse matrix into a full one.

In the following session, we create a 3×5 dense matrix. Then we use the `sparse` function to convert it into a sparse matrix. Scilab then displays all the non-zero entries of the matrix one at a time. Finally, we use the `full` function to convert the sparse matrix into a dense one.

```
-->A = [
-->1 2 0 0 0
-->3 4 5 0 0
-->0 6 7 8 0
-->]
 A  =
    1.    2.    0.    0.    0.
    3.    4.    5.    0.    0.
    0.    6.    7.    8.    0.
-->B = sparse(A)
 B  =
(    3,    5) sparse matrix
(     1,     1)         1.
(     1,     2)         2.
(     2,     1)         3.
(     2,     2)         4.
(     2,     3)         5.
(     3,     2)         6.
(     3,     3)         7.
(     3,     4)         8.
-->C = full(B)
 C  =
    1.    2.    0.    0.    0.
    3.    4.    5.    0.    0.
    0.    6.    7.    8.    0.
```

Sparse matrices can be real or complex. In the following session, we define a 3-by-5 complex matrix of doubles.

```
-->A = [
-->1 2 0 0 0
-->3 4 5 0 0
-->0 6 7 8 0
-->];
-->B = [
-->9  10  0  0 0
-->11 12 13  0 0
-->0  14 15 16 0
-->];
-->C=complex(A,B)
 C  =
  1. + 9.i    2. + 10.i   0           0           0
  3. + 11.i   4. + 12.i   5. + 13.i   0           0
  0           6. + 14.i   7. + 15.i   8. + 16.i   0
-->D=sparse(C)
 D  =
(     3,     5) sparse matrix
(     1,     1)       1. + 9.i
(     1,     2)       2. + 10.i
(     2,     1)       3. + 11.i
(     2,     2)       4. + 12.i
(     2,     3)       5. + 13.i
(     3,     2)       6. + 14.i
(     3,     3)       7. + 15.i
(     3,     4)       8. + 16.i
```

2.2 Storage format

There are various ways of storing the nonzero entries of a sparse matrix. In this section, we review how sparse matrices are stored internally, at the library level.

As we are going to see, the format used in Scilab is very similar (but not exactly identical) to the compressed sparse row format (CSR), in which the nonzero entries are stored row-by-row. While this detail does not change the way a user user sparse matrices at the interpreter level, it does have an impact on the way the libraries are connected to Scilab. Hence, this is an important feature of sparse matrices in Scilab and it is worthwhile to know precisely what is stored.

In Scilab, a sparse matrix is defined in the `modules/core/includes/s-parse.h`. This is a C `struct` with type `scisparse` and the following fields :

– `int m` : the number of rows.
– `int n` : the number of columns.
– `int it` : is equal to 0 if the matrix is real, and 1 if the matrix is complex.
– `int nel` : number of nonzero elements */
– `int *mnel` : an array with `m` entries. The entry `mnel[i]` is the number of non nul elements of the i-th row.
– `int *icol` : an array with `nel` entries. The entry `icol[j]` is the column of the j-th nonzero element.
– `double *R` : an array with `nel` entries. The entry `R[j]` is the real part of the j-th nonzero element.
– `double *I` : an array with `nel` entries. The entry `I[j]` is the imaginary part of the j-th nonzero element.

Consider the sparse matrix in the following example.

```
-->A = [
-->1 2 0
-->3 4 5
-->0 6 7
-->];
-->B = sparse(A);
```

In this case, the fields are the following.

```
B.m: 3
B.n: 3
B.it: 0
B.nel: 7
B.mnel: [2, 3, 2]
```

```
B.icol: [1, 2, 1, 2, 3, 2, 3]
B.R: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
B.I: [undefined]
```

This current storage is different from the CSR format, in the sense that there is no array containing the row indices corresponding to the values and no array containing the cumulated indexes where each column starts.

2.3 Sparse arithmetic

Scilab provides arithmetic operators for sparse matrices.

Indeed, most operators are defined for sparse matrices. For example, the operators +, - and * can be used for sparse matrices, as shown in the following example.

```
Afull= [
  2  3  0  0  0;
  3  0  4  0  6;
  0 -1 -3  2  0;
  0  0  1  0  0;
  0  4  2  0  1
];
A = sparse(Afull);
B = 2*A
C = A+B
x = [1;2;3;4;5]
b = A * x
```

Another operator is the backslash operator \, which is presented in the next section.

3 Solving sparse linear equations

Scilab provide direct and iterative methods to solve linear systems of equations. The figure 2 presents these methods.

3.1 Sparse LU decomposition

The sparse LU decomposition available in Scilab is based on the Sparse package written by Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli [5]. This package is available in Netlib [4].

lufact	sparse lu factorization
lusolve	sparse linear system solver
luget	extraction of sparse LU factors
ludel	utility function used with lufact
chfact	sparse Cholesky factorization
chsolve	sparse Cholesky solver
spchol	sparse Cholesky factorization
gmres	Generalized Minimum RESidual method
pcg	precondioned conjugate gradient
qmr	quasi minimal resiqual method with preconditioning

FIGURE 2 – Direct and iterative functions to solve sparse linear equations.

Sparse is a flexible package of subroutines written in C used to quickly and accurately solve large sparse systems of linear equations. The package is able to handle arbitrary real and complex square matrix equations. Besides being able to solve linear systems, it is also able to quickly solve transposed systems, find determinants, and estimate errors due to ill-conditioning in the system of equations and instability in the computations. Sparse also provides a test program that is able read matrix equations from a file, solve them, and print useful information about the equation and its solution.

Sparse is generally as fast or faster than other popular sparse matrix packages when solving many matrices of similar structure. Sparse does not require or assume sym- metry and is able to perform numerical pivoting to avoid unnecessary error in the solution. It handles its own memory allocation, which allows the user to forgo the hassle of providing adequate memory. It also has a natural, flexi- ble, and efficient interface to the calling program.

Sparse was originally written for use in circuit simu- lators and is particularly apt at handling node- and modified-node admittance matrices. The systems of linear generated in a circuit simulator stem from solving large systems of nonlinear equations using Newton's method and integrating large stiff systems of ordinary differential equations. However, Sparse is also suitable for other uses, one in particular is solving the very large systems of linear equations resulting from the numerical solution of partial differential equations.

The `lufact`, `lusolve`, `luget` and `ludel` functions is a set of functions providing a sparse direct method to solve linear systems of equations. As their names suggest it, they use a sparse LU decomposition to compute the solution.

In the following script, we define a sparse matrix `spA` with entries $(1, 2, 3, 4)$ on the diagonal only. Then we use the `lufact` function to compute the LU decomposition of the matrix `spA`. The output of the `lufact` function are a handle to the factors `h` and the rank `rk` of the matrix. The variable `h` has no direct use for the user. In fact, it represents a memory location where Scilab has stored the actual factors. The purpose of the variable `h` is to be an input argument of the `lusolve`, `luget` and `ludel` functions. In the script, we pass `h` and the right hand-side `b` to the `lusolve` function, which produces the solution `x` of the equation $Ax = b$. Finally, we delete the LU factors with the `ludel` function.

```
non_zeros=[1,2,3,4];
rows_cols=[1,1;2,2;3,3;4,4];
spA = sparse(rows_cols,non_zeros)
[h,rk] = lufact(sp);
b = [1 1 1 1]';
x=lusolve(h,b)
ludel(h);
```

The previous script produces the following output.

```
-->non_zeros=[1,2,3,4];
-->rows_cols=[1,1;2,2;3,3;4,4];
-->spA = sparse(rows_cols,non_zeros)
 spA  =
(    4,    4) sparse matrix
(    1,    1)        1.
(    2,    2)        2.
(    3,    3)        3.
(    4,    4)        4.
-->[h,rk] = lufact(sp);
-->b = [1 1 1 1]';
-->x=lusolve(h,b)
 x  =
    1.
    0.5
    0.3333333
    0.25
-->ludel(h);
```

3.2 Sparse backslash

The backslash operator `\` can be used with sparse matrices. Depending on the size of the sparse matrix `A`, the statement `A\b` has two different meanings.

– If the matrix `A` is square, therefore the linear system of equations $Ax = b$ is solved. In this case, the sparse backslash uses a sparse LU decomposition to solve the problem.
– If the matrix `A` is non square, therefore the linear least squares problem $\min \|Ax - b\|_2$ is solved. In this case, the sparse backslash uses the normal equations and form the linear system of equations $A^T Ax = A^T b$. Then a sparse LU decomposition is used to solve the problem.

In the following example, we solve a sparse 5-by-5 system of linear equations.

```
Afull= [
  2  3  0  0  0;
  3  0  4  0  6;
  0 -1 -3  2  0;
  0  0  1  0  0;
  0  4  2  0  1
];
A = sparse(Afull);
b = sparse([8 ; 45; -3; 3; 19]);
x = A\b
```

In the following example, we solve a 7-by-5 overdetermined system of equations.

```
Afull= [
  2  3  0  0  0
  3  0  4  0  6
  0 -1 -3  2  0
  0  0  1  0  0
  0  4  2  0  1
  2 -5  0 -6  3
  2  0 -5 -6  3
];
A = sparse(Afull);
b = [8 ; 45; -3; 3; 19; -5; 8];
x = A\b
norm(A*x-b)
```

The previous script produces the following output.

```
-->x = A\b
 x  =
  - 0.3094732
    2.9663371
    0.7519368
    1.6123139
    7.0851981
-->norm(A*x-b)
 ans  =
    3.2147514
```

The sparse backslash operator \ is based on the `lufact` and `lusolve` functions. It is implemented with overloading. For example, the `%sp_l_sp` function provides the operation `A\b`, where both `A` and `b` are sparse.

The sparse backslash in Scilab v5.3.2 has a major drawback : it does not manage sparse triangular matrices [8].

3.3 Iterative methods for sparse linear equations

The `pcg` function is a precondioned conjugate gradient algorithm for symmetric positive definite matrices. It can managed dense or sparse matrices, but is mainly designed for large sparse systems of linear equations. It is based on a Scilab port of the Matlab scripts provided in [1].

In the following example, we define a dense, well-conditionned 10×10 dense matrix A and a right hand size b made of ones. We use the `sparse` function to convert the `A` matrix into the sparse matrix `Asparse`. We finally use the `pcg` function on the sparse matrix `Asparse` in order to solve the equations $Ax = b$. This produces the solution `x`, a status flag `fail`, the relative residual norm `err`, the number of iterations `iter` and the vector of the residual relative norms `res`.

```
A=[ 94  0   0   0    0   28  0   0   32  0
    0   59  13  5    0   0   0   10  0   0
    0   13  72  34   2   0   0   0   0   65
    0   5   34  114  0   0   0   0   0   55
    0   0   2   0    70  0   28  32  12  0
    28  0   0   0    0   87  20  0   33  0
    0   0   0   0    28  20  71  39  0   0
    0   10  0   0    32  0   39  46  8   0
    32  0   0   0    12  33  0   8   82  11
    0   0   65  55   0   0   0   0   11  100];
b=ones(10,1);
Asparse=sparse(A);
[x, fail, err, iter, res]=..
    pcg(Asparse,b,maxIter=30,tol=1d-12)
```

The previous script produces the following output.

```
-->[x, fail, err, iter, res]=..
-->    pcg(Asparse,b,maxIter=30,tol=1d-12)
 res  =
    1.
    0.2302743
    0.1102172
    0.0223463
    0.0096446
    0.0052038
    0.0037525
    0.0006959
    0.0000207
    0.0000042
    1.697D-13
 iter  =
    10.
 err  =
    1.697D-13
 fail  =
    0.
 x  =
    0.0071751
    0.0134492
    0.0067610
    0.0050339
    0.0073735
    0.0065248
    0.0042064
    0.0093434
    0.0044640
    0.0023456
```

We see that 10 iterations were required to get a residual close to 10^{-13}.

4 Cholesky factorizations

In this section, we review the `chfact` and `spchol` functions, which both perform sparse Cholesky decomposition.

The figure 3 presents the functions which allow to compute the Cholesky decomposition of sparse matrices.

chfact	sparse Cholesky factorization
chsolve	sparse Cholesky solver
spchol	sparse cholesky factorization with permutations

Figure 3 – Cholesky factorizations for sparse matrices.

4.1 The `chfact` and `chsolve` functions

The `chfact` and `chsolve` functions can be combined in order to solve sparse linear systems of equations, if the matrix is symmetric positive definite. In the following example, we solve the equation $Ax = b$ where A is a sparse 5-by-5 symmetric definite positive matrix.

```
Afull= [
    2 -1  0  0  0;
    -1  2 -1  0  0;
    0 -1  2 -1  0;
    0  0 -1  2 -1;
    0  0  0 -1  2
];
A = sparse(Afull);
h = chfact(A);
b = [0 ; 0; 0; 0; 6];
chsolve(h,b)
```

In the previous script, the variable `h` is a complex data structure which contains the Cholesky decomposition. This decomposition makes use of a minimal degree algorithm, which reduces the fill-in in the Cholesky factors. Hence, the Cholesky decomposition makes implicitely use of a permutation matrix P, such that $P'AP$ can be more efficiently factored than A.

The previous script produces the following output.

```
-->chsolve(h,b)
 ans  =
    1.
    2.
    3.
    4.
    5.
```

4.2 The `spchol` function

The `[L,P]=spchol(X)` statement produces a sparse lower triangular matrix L and a sparse permutation matrix P such that $PLL^TP^T = X$. In the following script, we use the `spchol` function to compute the sparse Cholesky decomposition of a symmetric definite positive matrix.

```
-->Afull= [
-->     2 -1  0  0  0;
-->     -1  2 -1  0  0;
-->     0 -1  2 -1  0;
-->     0  0 -1  2 -1;
-->     0  0  0 -1  2
-->];
-->A = sparse(Afull);
[L,P]=spchol(A)
```

The previous script produces the following output.

```
-->[L,P]=spchol(A)
 P  =
(    5,    5) sparse matrix
(    1,    3)        1.
(    2,    4)        1.
(    3,    5)        1.
(    4,    2)        1.
(    5,    1)        1.
 L  =
(    5,    5) sparse matrix
(    1,    1)        1.4142136
(    2,    1)      - 0.7071068
(    2,    2)        1.2247449
(    3,    3)        1.4142136
(    4,    3)      - 0.7071068
(    4,    4)        1.2247449
(    5,    2)      - 0.8164966
(    5,    4)      - 0.8164966
(    5,    5)        0.8164966
```

The `spchol` function [10] uses two Fortran packages : a sparse Cholesky factorization package developed by Esmond Ng and Barry Peyton at ORNL and a multiple minimun-degree ordering package by Joseph Liu at University of Waterloo.

The `spchol` function can be used to solve linear systems of equations. Indeed, assume that A is a sparse n-by-n real symmetric matrix and that b

is a n-by-1 vector. We may want to solve the system of equations

$$Ax = b \tag{1}$$

for x. Assume that the matrix A can be decomposed into :

$$A = PLL^T P^T, \tag{2}$$

where L is an n-by-n lower triangular real matrix and P is an n-by-n permutation matrix. The product $P^T P$ is the identity matrix, which implies :

$$(AP)(P^T x) = b \tag{3}$$

We left multiply this equation by P^T and get :

$$(P^T AP)(P^T x) = P^T b \tag{4}$$

Hence

$$(P^T PLL^T P^T P)(P^T x) = P^T b \tag{5}$$

which simplifies into

$$(LL^T)(P^T x) = P^T b. \tag{6}$$

In order to solve this equation, we first solve the equation

$$(LL^T)y = P^T b \tag{7}$$

where

$$y = P^T x. \tag{8}$$

In the Scilab language, the solution is

```
y = L'\(L\P'*b)
```

Moreover, `x=P*y`, which implies

```
x = P*(L'\(L\(P'*b)))
```

In the following script, we use the `spchol` function to solve a symmetric sparse linear system of equations.

```
Afull= [
    2 -1  0  0  0;
    -1  2 -1  0  0;
    0 -1  2 -1  0;
    0  0 -1  2 -1;
    0  0  0 -1  2
];
A = sparse(Afull);
[L,P] = spchol(A);
n = size(A,"r");
e = (1:n)';
b = A * e;
x = P*(L'\(L\(P'*b)))
```

The previous script produces the following output.

```
-->x = P*(L'\(L\(P'*b)))
 x  =
    1.
    2.
    3.
    4.
    5.
```

4.3 Conclusion

The spchol and chfact actually use the same algorithms. The method is based on the minimum degree algorithm, which produces a permutation matrix P which reduces the amount of fill-in in the Cholesky factors. The chsolve function may be considered as unnecessary, since the sparse backslash operator is designed for the same purpose.

5 Functions from the UMFPACK module

The UMFPACK module provide several functions related to sparse matrices. These functions are presented in the figure figure 4.

In the following script, we read a sparse matrix provided in the Umfpack module with the ReadHBSparse function. Then we plot the sparsity pattern with the PlotSparse function.

```
umfdir = fullfile(SCI,"modules","umfpack","examples");
filename = fullfile(umfdir,"arc130.rua");
```

PlotSparse	plot the pattern of non nul elements of a sparse matrix
ReadHBSparse	read a Harwell-Boeing sparse format file
cond2sp	computes an approximation of the 2-norm condition number of a s.p.d. sparse matrix
condestsp	estimate the condition number of a sparse matrix
rafiter	(obsolete) iterative refinement for a s.p.d. linear system
res_with_prec	computes the residual r = Ax-b with precision

FIGURE 4 – Functions from the umfpack module.

```
A = ReadHBSparse(filename);
PlotSparse(A,"y+");
```

The previous script produces the figure 5.

6 The UMFPACK package

The UMFPACK package provide several direct algorithms to compute LU decompositions of sparse matrices. The algorithms also solve sparse linear systems of equations, that is, they solve the equation $Ax = b$, where A is a sparse squares matrix and b is a sparse vector. These functions are presented in the figure 6.

The following example shows how to combine the `umf_lufact` function with the `umf_lusolve` function in order to solve the linear system of equations $Ax = b$.

```
Afull= [
  2  3  0  0  0;
  3  0  4  0  6;
  0 -1 -3  2  0;
  0  0  1  0  0;
  0  4  2  0  1
];
A = sparse(Afull)
b = [8 ; 45; -3; 3; 19];
h = umf_lufact(A);
x = umf_lusolve(h,b)
umf_ludel(h);
```

The previous script produces the following output.

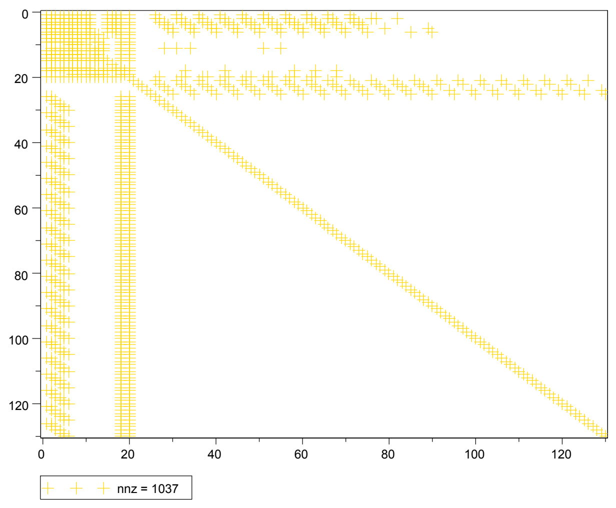

FIGURE 5 – Sparsity pattern of the arc130 matrix.

umf_license	display the umfpack license
umf_ludel	utility function used with umf_lufact
umf_lufact	lu factorisation of a sparse matrix
umf_luget	retrieve lu factors at the scilab level
umf_luinfo	get information on LU factors
umf_lusolve	solve a linear sparse system given the LU factors
umfpack	solve sparse linear system

FIGURE 6 – Functions from the umfpack module.

taucs_chdel	utility function used with taucs_chfact
taucs_chfact	cholesky factorisation of a sparse s.p.d. matrix
taucs_chget	retrieve the Cholesky factorization at the scilab level
taucs_chinfo	get information on Cholesky factors
taucs_chsolve	solve a linear sparse system given the Cholesky factors
taucs_license	display the taucs license

FIGURE 7 – Functions from the TAUCS module.

```
-->x = umf_lusolve(h,b)
 x  =
    1.
    2.
    3.
    4.
    5.
```

In the previous script, the variable **`h`** is a matrix handle, which contains informations related to the sparse matrix. This is why it is necessary to explicitely delete the matrix with the **`umf_ludel`** function. Indeed, if we do not delete the matrix handle, there is a loss of memory.

The Umfpack library was developped by Timothy Davis [2], from the University of Florida. His package is distributed under the GNU GPL license.

7 The TAUCS package

The TAUCS package provide several direct algorithms to compute Cholesky decompositions of sparse matrices. This package only manage symmetric positive definite matrices. The algorithms also solve sparse linear systems of equations, that is, they solve the equation $Ax = b$, where A is a sparse squares matrix and b is a sparse vector.

The TAUCS package is available in the UMFPACK module. The functions provided in the TAUCS package are presented in the figure 7.

In the following example, we factor a sparse matrix and solve the associated linear system of equations. Notice that the matrix **`A`** is symmetric positive definite.

```
Afull= [
    2 -1  0  0  0;
    -1   2 -1   0   0;
    0 -1   2 -1   0;
    0   0 -1   2 -1;
    0   0   0 -1   2
];
A = sparse(Afull);
b = [0 ; 0; 0; 0; 6];
h = taucs_chfact(A);
x = taucs_chsolve(h,b)
taucs_chdel(h);
```

In the previous script, the variable `h` is a matrix handle, which contains informations related to the sparse matrix. This is why it is necessary to explicitely delete the matrix with the `taucs_chdel` function. Indeed, if we do not delete the matrix handle, there is a loss of memory.

The TAUCS library was developed by Sivan Toledo from Tel-Aviv University [12].

8 The API

Scilab provides an API to manage sparse matrices. The figure 8 presents the functions to read or write sparse matrices in gateways. The figure 9 presents the functions to read or write sparse matrices in lists.

The following `read_sparse` function is a sample example of some of the functions which may be used in gateways which manage sparse matrices. This example is extracted from the help pages of Scilab.

The `read_sparse` function takes a sparse matrix of doubles as input argument and prints its content in the console. The sparse matrix may be real or complex, which is taked into account in the gateway, based on the output of the `isVarComplex` function. If the matrix is complex, we call the `getComplexSparseMatrix`, which sets the data structures of the sparse matrix. The data structures associated with sparse matrices are presented in the section 2.2 :

– `iRows` : the number of rows,
– `iCols` : the number of columns,
– `iNbItem` : the number of nonzero entries,
– `piNbItemRow` : the number of nonzeros on each row,
– `piColPos` : the column index of each nonzero entry,
– `pdblReal` : the real part of the nonzero entry,

Read sparse doubles matrices in gateways	
getSparseMatrix	R. a sp. mat.
getComplexSparseMatrix	R. a complex sp. mat.
readNamedSparseMatrix	R. a named sp. mat.
readNamedComplexSparseMatrix	R. a named complex sp. mat.
Write sparse doubles matrices in gateways	
createSparseMatrix	W. a sp. mat.
createComplexSparseMatrix	W. a complex sp. mat.
createNamedSparseMatrix	W. a named sp. mat.
createNamedComplexSparseMatrix	W. a named complex sp. mat.
Read/Write sparse boolean matrices in gateways	
getBooleanSparseMatrix	R. a boolean sp. mat.
readNamedBooleanSparseMatrix	R. a named boolean sp. mat.
createBooleanSparseMatrix	W. a boolean sp. mat.
createNamedBooleanSparseMatrix	W. a named boolean sp. mat.

FIGURE 8 – The sparse API, to be used in gateways.

– `pdblImg` : the imaginary part of the nonzero entry.

The `getSparseMatrix` function has the same effect on real matrices, but only the `pdblReal` array is set.

```
int read_sparse(char *fname,unsigned long fname_len)
{
   SciErr sciErr;
   int i,j,k;
   int* piAddr          = NULL;
   int iRows          = 0;
   int iCols          = 0;
   int iNbItem          = 0;
   int* piNbItemRow    = NULL;
   int* piColPos       = NULL;
   double* pdblReal    = NULL;
   double* pdblImg       = NULL;
   CheckRhs(1,1);
   sciErr = getVarAddressFromPosition(pvApiCtx, 1, &piAddr);
   if(sciErr.iErr)
   {
      printError(&sciErr, 0);
      return 0;
   }
   if(isVarComplex(pvApiCtx, piAddr))
```

Read sparse matrices in a list
getSparseMatrixInList
R. a sp. mat. of doubles in a list.
getComplexSparseMatrixInList
R. a sp. mat. of complex doubles in a list.
readSparseMatrixInNamedList
R. a named sp. mat. of doubles in a list.
readComplexSparseMatrixInNamedList
R. a named sp. mat. of complex doubles in a list.

Write sparse matrices in lists
createSparseMatrixInList
W. a sp. mat. of doubles in a list.
createComplexSparseMatrixInList
W. a sp. mat. of complex doubles in a list.
createSparseMatrixInNamedList
W. a sp. mat. of doubles in a named list.
createComplexSparseMatrixInNamedList
W. a sp. mat. of complex doubles in a named list.

Read/Write sparse boolean matrices in lists
getBooleanSparseMatrixInList
R. a sp. mat. of booleans in a list.
readBooleanSparseMatrixInNamedList
R. a sp. mat. of booleans in a named list.
createBooleanSparseMatrixInList
W. a sp. mat. of booleans in a list.
createBooleanSparseMatrixInNamedList
W. a sp. mat. of booleans in a named list.

FIGURE 9 – The sparse API to read/write in lists.

```
    {
        sciErr = getComplexSparseMatrix(pvApiCtx,
            piAddr, &iRows, &iCols, &iNbItem, &piNbItemRow,
            &piColPos, &pdblReal, &pdblImg);
    }
    else
    {
        sciErr = getSparseMatrix(pvApiCtx, piAddr,
             &iRows, &iCols, &iNbItem,
             &piNbItemRow, &piColPos, &pdblReal);
    }
    if(sciErr.iErr)
    {
        printError(&sciErr, 0);
        return 0;
    }
    sciprint("Sparse %d item(s)\n", iNbItem);
    k = 0;
    for(i = 0 ; i < iRows ; i++)
    {
        for(j = 0 ; j < piNbItemRow[i] ; j++)
        {
            sciprint("(%d,%d) = %f",
                i+1, piColPos[k], pdblReal[k]);
            if(isVarComplex(pvApiCtx, piAddr))
            {
                sciprint(" %+fi", pdblImg[k]);
            }
            sciprint("\n");
            k++;
        }
    }
    LhsVar(1) = 0;
    return 0;
}
```

9 The ARnoldi PACKage

ARPACK is a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems. The functions available in Scilab are presented in the figure 10.

The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A. It is most appropriate for large sparse or structured matrices A where structured means that a matrix-vector product $w = Av$ requires order n rather than the usual order n^2 floating point operations. This software is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix A is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). These variants may be viewed as a synthesis of the Arnoldi/Lanczos process with the Implicitly Shifted QR technique that is suitable for large scale problems. For many standard problems, a matrix factorization is not required. Only the action of the matrix on a vector is needed.

ARPACK software is capable of solving large scale symmetric, nonsymmetric, and generalized eigenproblems from significant application areas. The software is designed to compute a few (k) eigenvalues with user specified features such as those of largest real part or largest magnitude. Storage requirements are on the order of nk locations. No auxiliary storage is required. A set of Schur basis vectors for the desired k-dimensional eigen-space is computed which is numerically orthogonal to working precision. Numerically accurate eigenvectors are available on request.

The following is a list of the main features of the library :

– Reverse Communication Interface.
– Single and Double Precision Real Arithmetic Versions for Symmetric, Non-symmetric,
– Standard or Generalized Problems.
– Single and Double Precision Complex Arithmetic Versions for Standard or Generalized Problems.
– Routines for Banded Matrices - Standard or Generalized Problems.
– Routines for The Singular Value Decomposition.
– Example driver routines that may be used as templates to implement numerous Shift-Invert strategies for all problem types, data types and precision.

10 The Matrix Market module

The Matrix Market (MM) exchange formats provide a simple mechanism to facilitate the exchange of matrix data. In particular, the objective has been to define a minimal base ASCII file format which can be very easily explained and parsed, but can easily adapted to applications with a more

dnaupd	compute approximations to a few eigenpairs of a real linear operator
dneupd	compute the converged approximations to eigenvalues of $Az = \lambda Bz$, approximations to a few eigenpairs of a real linear operator
dsaupd	compute approximations to a few eigenpairs of a real and symmetric linear operator
dsaupd	compute approximations to the converged approximations to eigenvalues of $Az = \lambda Bz$
znaupd	compute a few eigenpairs of a complex linear operator with respect to a semi-inner product defined by a hermitian positive semi-definite real matrix B.
zneupd	compute approximations to the converged approximations to eigenvalues of $Az = \lambda Bz$

FIGURE 10 – Functions from the Arnoldi Package.

rigid structure, or extended to related data objects. The MM exchange format for matrices is really a collection of affiliated formats which share design elements.

In the specification, two matrix formats are defined.

- Coordinate Format. A file format suitable for representing general sparse matrices. Only nonzero entries are provided, and the coordinates of each nonzero entry is given explicitly. This is illustrated in the example below.
- Array Format. A file format suitable for representing general dense matrices. All entries are provided in a pre-defined (column-oriented) order.

The Matrix Market file format can be used to manage dense or sparse matrices. MM coordinate format is suitable for representing sparse matrices. Only nonzero entries need be encoded, and the coordinates of each are given explicitly.

http://atoms.scilab.org/toolboxes/MatrixMarket

In order to install the Matrix Market module, we use the atoms system, as in the following script.

mminfo	Extracts size and storage information
mmread	Reads a Matrix Market file
mmwrite	Writes a sparse or dense matrix

FIGURE 11 – Functions from the Matrix Market module.

```
atomsInstall("MatrixMarket")
```

The table 11 presents the functions provided by the Matrix Market module.

In the following script, we create a sparse matrix with the `sparse` function and save it into a file with the `mmwrite` function.

```
A=sparse([1,1;1,3;2,2;3,1;3,3;3,4;4,3;4,4],..
  [9;27+%i;16;27-%i;145;88;88;121],[4,4]) ;
filename = TMPDIR+"/A.mtx";
mmwrite(filename,A);
```

In the following session, we use the `mminfo` function to extract informations from this file.

```
-->mminfo(filename);
=====================================
Information about MatrixMarket file :
C:\Users\myname\AppData\Local\Temp\SCI_TMP_8216_/A.mtx
%%% Generated by Scilab 11-Jun-2010
storage: coordinate
entry type: complex
symmetry: hermitian
=====================================
```

In the following session, we use another calling sequence of the `mminfo` function to extract data from the file. This allows to get the number of rows `rows`, the number of columns `cols`, the number of entries and other informations as well.

```
-->[rows,cols,entries,rep,field,symm,comm] = mminfo(filename)
 comm  =
 %%% Generated by Scilab 11-Jun-2010
 symm  =
 hermitian
 field  =
 complex
 rep  =
 coordinate
 entries  =
    6.
 cols  =
    4.
 rows  =
    4.
```

In the following session, we use the `mmread` function to read the content of the file and create the matrix `A`. We use the `nnz` function to compute the number of nonzero entries in the matrix.

```
-->A=mmread(filename)
 A  =
(    4,    4) sparse matrix
(    1,    1)       9.
(    1,    3)       27. + 1.i
(    2,    2)       16.
(    3,    1)       27. - 1.i
(    3,    3)       145.
(    3,    4)       88.
(    4,    3)       88.
(    4,    4)       121.
 -->nnz(A)
 ans  =
    8.
```

11 Solving Poisson PDE with Sparse Matrices

In this section, we present the resolution of the Poisson Partial Differential Equation in Scilab with sparse matrices. We show that Scilab 5 can solve in a few seconds sparse linear systems of equations with as many as 250 000 unknowns because Scilab only store nonzero entries. The computations are based on the Scibench module, a toolbox which provides a collection of benchmarks for Scilab. This section was first published at [7].

11.1 Introduction

Sharma and Gobbert analyzed the performance of Scilab for the resolution of sparse linear systems of equations associated with the Poisson equation [11]. In this document, we try to reproduce their experiments.

We consider the Poisson problem with homogeneous Dirichlet boundary conditions and are interested in the numerical solution based on finite differences.

We consider the 2 dimensional Partial Differential Equation :

$$\begin{aligned} -\Delta u &= f \quad \text{in the domain,} \\ u &= 0 \quad \text{on the frontier.} \end{aligned}$$

where the two dimensionnal Laplace operator is

$$\Delta u = \frac{\partial^2 u}{dx^2} + \frac{\partial^2 u}{dy^2}$$

We consider the domain $0 \leq x \leq 1$, $0 \leq y \leq 1$.

The function f is defined by

$$f(x,y) = -2\pi^2 cos(2\pi x) sin^2(\pi y) - 2\pi^2 sin^2(\pi x) cos(2\pi y)$$

The solution is

$$u(x,y) = sin^2(\pi x) sin^2(\pi y)$$

We use a second order finite difference approximation of the Laplace operator based on a grid of N-by-N points.

Sparse matrices can be managed in Scilab since 1989 [3]. In Scilab 5.0 (i.e. in 2008), the UMFPACK module was added.

The Scibench external module :

http://atoms.scilab.org/toolboxes/scibench

provides a collection of benchmark scripts to measure the performance of Scilab. For example, it contains benchmarks for the dense matrix-matrix product, the dense backslash operator, the Cholesky decomposition or 2D Lattice Boltzmann simulations.

In order to install this module, we use the statement :

```
atomsInstall("scibench")
```

and restart Scilab.

The version 0.6 includes benchmarks for sparse matrices, based on the Poisson equation. The performance presented in this document are measured with Scilab 5.3.2 on Windows XP with a 4GB computer using Intel Xeon E5410 4*2.33 GHz processors.

The script that we are going to use is `poisson.sce`, which is provided in the demos directory of the module.

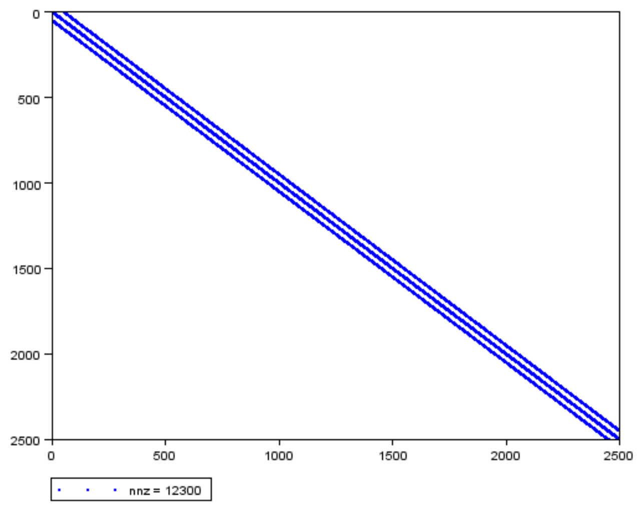

FIGURE 12 – Sparsity pattern of the Poisson matrix.

11.2 The sparse matrix

The `scibench_poissonA(n)` statement creates a sparse matrix equation associated with n cells in the X coordinate and n cells in the Y coordinate. Before calling this session, we call the stacksize function in order to let Scilab allocate as much memory as possible. Then we call the `PlotSparse` function to plot the sparsity pattern of the matrix.

```
stacksize("max");
A = scibench_poissonA(50);
PlotSparse(A)
```

The previous script produces the plot in the figure 12.

We emphasize that the previous matrix is a n^2-by-n^2 sparse matrix. Even for moderate values of n, this creates huge matrices. Fortunately, only nonzero entries are stored, so that Scilab has no problem to store it, provided that the nonzero entries can be stored in the memory.

```
-->size(A)
 ans  =
    2500.    2500.
```

The Kronecker operator is used, so that the computation of the matrix is vectorized. To edit the code, we just run the following code.

```
-->edit scibench_poissonA
```

At the core of the algorithm, we find the statement

```
A = I .*. T + T .*. I
```

where T is a sparse matrix and I is the sparse n-by-n identity matrix. Hence, creating the whole matrix is done with only one statement.

11.3 The Poisson Solver

The `scibench_poisson` function solves the 2D Poisson equation. Its calling sequence is

```
scibench_poisson ( N , plotgraph , verbose , solver )
```

where `N` is the number of cells, `plotgraph` is a boolean to plot the graphics, `verbose` is a boolean to print messages and solver is a function which solves the linear equation `A*x=b`.

The solver argument is designed so that we can customize the linear equation solver which we want to use. For example, if we want to use the sparse backslash operator, all we have to do is to create the `mysolverBackslash` function as below.

```
function u=mysolverBackslash(N, b)
    A = scibench_poissonA(N);
    u = A\b;
endfunction
```

The following script solves the Poisson equation with N = 50.

```
scf();
scibench_poisson(50, %t , %t , mysolverBackslash );
```

The previous script produces the following output, where `h` is the dimensionnal spacee step and enorminf is the value of the infinite norm of the error.

```
->scibench_poisson(50, %t , %t , mysolverBackslash );
N =    50
h =  1.9607843137254902e-002
h^2 =  3.8446751249519417e-004
enorminf =  1.2634082165868810e-003
C = enorminf / h^2 =  3.2861247713424775e+000
wall clock time =       0.15 seconds
```

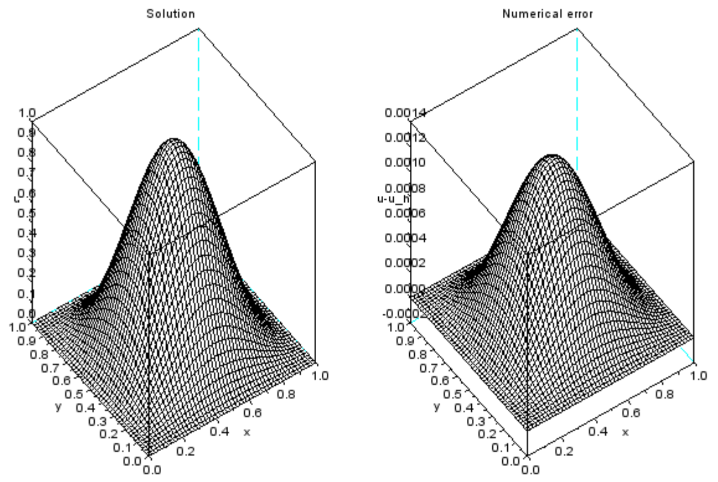

FIGURE 13 – Approximate solution and numerical error of the Poisson equation.

The previous script also produces the plot in the figure 13.

It is then easy to use the pcg function built in Scilab, which uses a preconditionned conjugate gradient algorithm. We use the `scibench_poissonAu` function, which computes the `A*u` product without actually storing the matrix A.

```
function u=mysolverPCG(N, b)
    tol = 0.000001;
    maxit = 9999;
    u = zeros(N^2,1);
    [u,flag,iter,res] = ..
            pcg(scibench_poissonAu,b,tol,maxit,[],[],u);
endfunction
scf();
[timesec,enorminf,h] = ..
    scibench_poisson(50, %f , %t , mysolverPCG );
```

11.4 The benchmark

Based on the scibench module, it is easy to compare various sparse linear equation solvers in Scilab. We compared the following functions :

- sparse backslash (which internally use a sparse LU decomposition),
- `pcg` function,
- `UMPFACK` module,
- the `TAUCS` module.

Both the UMFPACK and TAUCS modules were created by Bruno Pincon [9].

In order to use the UMFPACK module, we created the following `mysolverUMF` function which solves the A*u=b equation.

```
function u = mysolverUMF(N,b)
    A = scibench_poissonA(N);
    humf = umf_lufact(A);
    u = umf_lusolve(humf,b)
    umf_ludel(humf)
endfunction
```

We also created the following wrapper for the TAUCS module.

```
function u = mysolverTAUCS(N,b)
    A = scibench_poissonA(N);
    hchol = taucs_chfact(A);
    u = taucs_chsolve(hchol,b)
    taucs_chdel(hchol)
endfunction
```

We were unable to use the TAUCS solver, which makes Scilab unstable in this case. This problem was reported in the following bug report :

`http://bugzilla.scilab.org/show_bug.cgi?id=8824`

It is straightforward to created increasingly large matrices and to measure the time required to solve the Poisson equation. The script is provided within the demos of the scibench module. The plot in the figure 14 compares the various solvers.

It is obvious that, in this case, the UMFPACK module is much faster.

The fact that we use the pcg function without actually preconditionning the matrix is an obvious limitation of this benchmark. This is why we work on updating the sparse ILU preconditionners in the following Forge project :

`http://forge.scilab.org/index.php/p/spilu/`

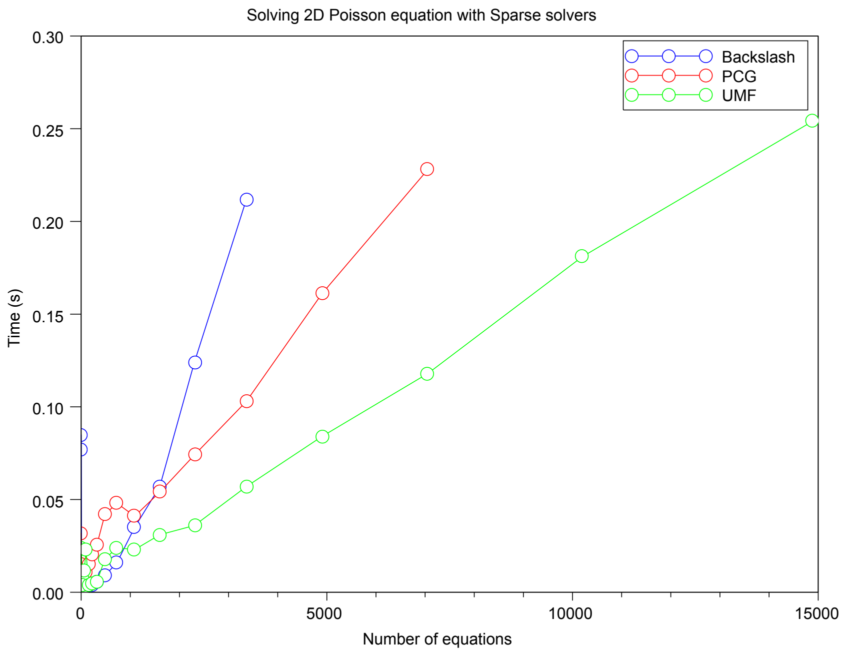

FIGURE 14 – Comparison of solvers for the sparse linear system of equations of the Poisson equation.

Solver	N	Matrix Size	Time (s)
Backslash	50	2500-by-2500	0.15
PCG	50	2500-by-2500	0.09
Backslash	100	10000-by-10000	4.32
PCG	100	10000-by-10000	0.32
Backslash	200	40000-by-40000	88.89
PCG	200	40000-by-40000	2.03
UMFPACK	200	40000-by-40000	0.81
PCG	300	90000-by-90000	7.27
UMFPACK	300	90000-by-90000	2.35
PCG	500	250000-by-250000	44.77

FIGURE 15 – The Poisson benchmark for various size of matrices.

The current work is based on the former Scilin project.

This benchmark shows that we can solve really large systems of equations. The table in the figure 15 displays the performance measures.

For matrices larger than approximately 400, the UMFPACK functions fails to solve the equation because they require more memory than Scilab can provide to it.

```
-->  scibench_poisson(400, %t , %t , mysolverUMF )
 !--error 999
umf_lufact: An error occurred:
  symbolic factorization: not enough memory
at line       3 of function mysolverUMF called by :
at line      95 of function scibench_poisson called by :
 scibench_poisson(400, %t , %t , mysolverUMF )
```

We emphasize that the actual size of the matrix does not matter. What matters is the number of nonzero terms in the matrix. For example, with N=500, the matrix is 250000-by-250000 but has only 1 248 000 nonzero entries.

```
-->A = scibench_poissonA ( 500 );
-->size(A)
 ans  =
    250000.    250000.
-->nnz(A)
 ans  =
    1248000.
```

11.5 Conclusion

Scilab 5 can manage sparse matrices and solve partial differential equations such as the Poisson equation for example. Indeed, Scilab provides several sparse linear equation solvers, including a sparse backslash operator, iterative methods (e.g. the pre-conditionned conjugate gradient algorithm) and other direct solvers such as the UMFPACK or TAUCS modules. With these tools we can solve huge systems of linear equations, because Scilab only stores the nonzero entries of these massive matrices. The solvers may fail if the memory required to store the matrix is beyond the capacity of Scilab, but this happens only for huge matrices. In this case, the future Scilab 6 may help to overcome this limitation, by removing the use of the stack which is used by Scilab 5 (see [6] for more details on this topic). Another limitation of Scilab is that there is currently no preconditionner for sparse linear systems of equations. This limitation should be removed once the *spilu* module is ready for a release.

12 The Imsls toolbox

The Imsls toolbox provides iterative methods for sparse linear systems of equations.

More precisely, it provides functions to find x such that $Ax = b$, where A is a n-by-n matrix of doubles and b is a n-by-1 matrix of doubles. Although these functions can manage full matrices, it is mainly designed for sparse matrices. One of the interesting point here is that the matrix-vector products or the preconditionning steps `M\x` can be performed either with full or sparse matrices, or with callback functions. This flexibility makes the module convenient to use in situations when the sparse matrices are not stored in memory, since only the matrix-vector product (or the preconditionning step `M\x`) is required. Moreover, we provide Matlab-compatible pcg, gmres and qmr solvers :

– the order of the arguments are the same as in Matlab,
– the default values of the Matlab functions are the same as in Matlab,
– the headers of the callback functions are the same as in Matlab.

This contrasts with Scilab's internal functions, where the two last points are completely unsatisfied. Finally, we provide a complete test suite for these functions, which are using robust argument checking.

imsls_bicg	BIConjugate Gradient method
imsls_bicgstab	BIConjugate Gradient STABilized method
imsls_pcg	Conjugate Gradient method
imsls_cgs	Conjugate Gradient Squared method
imsls_cheby	CHEBYshev method
imsls_gmres	Generalized Minimal RESidual method
imsls_jacobi	JACOBI method
imsls_qmr	Quasi Minimal Residual method
imsls_sor	Successive Over-Relaxation method

FIGURE 16 – Main functions in the Imsls toolbox.

mtlb_bicg	BiConjugate Gradient method
mtlb_bicgstab	BiConjugate Gradient Stabilized method
mtlb_cgs	Conjugate Gradient Squared method
mtlb_gmres	Generalized Minimal residual method
mtlb_pcg	Conjugate Gradient method
mtlb_qmr	Quasi Minimal Residual method

FIGURE 17 – Compatibility functions in the Imsls toolbox.

The main functions are presented in the figure 16.

The Imsls toolbox also provides Matlab compatibility functions, which are presented in the figure 17.

The `imsls_gmres` function finds x such that $Ax = b$. In the following script, we create a 16-by-16 matrix of doubles. Although this matrix is symmetric, the `imsls_gmres` function can manage nonsymmetric matrices. Then we define the expected result `xe`, and compute the right hand side `b`. Finally, we call the `imsls_gmres`, which returns `x`.

```
A=imsls_makefish(4);
xe=(1:16)';
b=A*xe;
x = imsls_gmres(A,b)
```

The previous script produces the following output.

```
-->x = imsls_gmres(A,b)
 x  =
    1.
```

The Imsls toolbox is available on Atoms :

http://atoms.scilab.org/toolboxes/imsls

To install this module, we can use the statement

```
atomsInstall("imsls")
```

and restart Scilab.

Références

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[2] Timothy A. Davis. Umfpack. http://www.cise.ufl.edu/research/sparse/umfpack/, 2009.

[3] François Delebecque, Carlos Klimann, and Serge Steer. Basile, un système de CAO pour l'automatique, version 3.0 : guide d'utilisation. 1989.

[4] Kenneth Kundert and Alberto Sangiovanni-Vincentelli. Sparse 1.3. http://www.netlib.org/sparse/.

[5] Kenneth S. Kundert. Sparse matrix techniques. In Albert E. Ruehli, editor, *Circuit Analysis, Simulation and Design*, volume 3 of *pt. 1*. North-Holland, 1986.

[6] Consortium Scilab DIGITEO Michael Baudin. Programming in Scilab. http://forge.scilab.org/index.php/p/docprogscilab/downloads/, 2011.

[7] The Scilab Consortium Michael Baudin. Solving poisson PDE with sparse matrices. http://wiki.scilab.org, 2011.

[8] The Scilab Consortium Michael Baudin. The sparse backslash does not manage triangular matrices. http://bugzilla.scilab.org/show_bug.cgi?id=10265, 2011.

[9] Bruno Pincon and Karim Ramdani. Scilab tools for PDE : Application to time-reversal. http://www.iecn.u-nancy.fr/~pincon/scilab/expose_taiwan.pdf, 2004.

[10] Héctor E. Rubio Scola. Implementation of lipsol in scilab. 1997. http://hal.inria.fr/inria-00069956/en.

[11] Neeraj Sharma and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, Freemat, and Scilab for research and teaching. http://userpages.umbc.edu/~gobbert/papers/SharmaGobbertTR2010.pdf, 2010.

[12] with Doron Chen Sivan Toledo and Vladimir Rotkin. Taucs, a library of sparse solvers. http://www.tau.ac.il/~stoledo/taucs, 2009.

[13] Wikipedia. Sparse matrix — wikipedia, the free encyclopedia, 2010. [Online ; accessed 11-June-2010].