Scilab and MATLAB Interfaces to MUMPS (version 4.6 or greater)

Aurélia Fèvre, Jean-Yves l'Excellent, Stéphane Pralet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scilab and MATLAB Interfaces to MUMPS (version 4.6 or greater)

Aurélia Fèvre — Jean-Yves L'Excellent — Stéphane Pralet (ENSEEIHT-IRIT)

N° 5816

January 2006

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5816--FR+ENG

INRIA
RHÔNE-ALPES

Scilab and MATLAB Interfaces to MUMPS (version 4.6 or greater)

Aurélia Fèvre , Jean-Yves L'Excellent , Stéphane Pralet (ENSEEIHT-IRIT)

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° 5816 — January 2006 — 24 pages

Abstract: This document describes the Scilab and MATLAB interfaces to MUMPS version 4.6. We describe the differences and similarities between usual Fortran/C MUMPS interfaces and its Scilab/MATLAB interfaces, the calling sequences and functionalities. Examples of use and experimental results are also provided.

Key-words: Direct solver, Sparse matrices, MUMPS, Scilab, MATLAB.

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Description des interfaces Scilab et MATLAB au logiciel MUMPS (version 4.6 et plus)

Résumé : Ce document décrit l'interface Scilab et l'interface MATLAB de la version 4.6 de MUMPS. Nous décrivons les séquences d'appel et les fonctionnalités de nos interfaces Scilab/MATLAB et nous évoquons ses différences et similarités avec les interfaces Fortran/C habituelles de MUMPS. Nous présentons aussi des exemples d'utilisation et quelques résultats expérimentaux.

Mots-clés : Solveur direct, Matrices creuses, MUMPS, Scilab, MATLAB.

1 Introduction

We consider the sparse direct solver MUMPS [3, 4, 5]. It computes the solution of linear systems of the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a real or complex square sparse matrix, that can be either unsymmetric, symmetric positive definite or general symmetric, and $\mathbf{b}$ is a dense or sparse vector or matrix. The MUMPS package uses a multifrontal technique to form the $\mathbf{LU}$ or the $\mathbf{LDL^T}$ factorization of the matrix $\mathbf{A}$, and to perform a forward and backward substitutions on the triangular factors.

Sparse direct methods are widely used to solve large systems of linear equations. They are very attractive because of their robustness (their successful completion is less dependant from the numerical properties of the matrix $\mathbf{A}$ than iterative methods) and because of the efficient implementations that have been developed during the last decades. Scilab (www.scilab.org) and MATLAB (www.mathworks.com) are two user-friendly scientific packages often used in the context of scientific and engineering applications. To give access to the main MUMPS functionalities within these two environments, we have developed both a Scilab interface and a MATLAB interface to MUMPS. In this report we describe the MUMPS functionalities available through our Scilab/MATLAB interfaces and detail their usage. The general scheme to use these two interfaces is similar to the more classical C and Fortran interfaces to MUMPS. Therefore, in the following, we refer the reader to the MUMPS user's guide[1] when a more detailed level of explanation is needed (on a particular control parameter, for example).

Section 2 describes the Scilab/MATLAB interfaces to MUMPS. In Section 2.1 we describe our implementation choices. In particular we stress the similarities and differences between our Scilab/MATLAB interfaces and the Fortran/C MUMPS interface. Section 2.2 describes the input and output parameters of the interfaces. Section 2.3 details the calling sequence. Performance results and comparisons with other solvers callable from Scilab/MATLAB are given in Section 3. Finally, in Section 4, we discuss possible future evolutions of these interfaces and give concluding remarks.

2 Description of the Scilab/MATLAB interfaces to MUMPS

This section gives an overview of the Scilab/MATLAB interfaces, their implementation and usage. In particular we describe the interface parameters and present examples of calling sequences. In the MUMPS distribution, we included our interfaces in the two subdirectories SCILAB and MATLAB (see Figure 2.1). Both directories contain three main callable functions `initmumps`, `dmumps` and `zmumps` that are used to initialize, and to call MUMPS with double and complex arithmetic, respectively. The `Scilab/examples` directory contains examples of scripts whereas MATLAB examples are available in the MATLAB directory.

[1] See http://graal.ens-lyon.fr/MUMPS/doc.html or http://www.enseeiht.fr/lima/apo/MUMPS/doc.html

```
MUMPS_4.6  --|- src
             |- include
             |- lib
             |
             :
             :
             |- SCILAB -|- initmumps.sci
             |          |- dmumps.sci
             |          |- zmumps.sci
             |          |- examples -|- double_example.sce
             |                      |- cmplx_example.sce
             |                      |- sparseRHS_example.sce
             |                      |- schur_example.sce
             |
             |- MATLAB -|- initmumps.m
                        |- dmumps.m
                        |- zmumps.m
                        |- printmumpsstat.m
                        |- simple_example.m
                        |- zsimple_example.m
                        |- sparserhs_example.m
                        |- multiplerhs_example.m
                        |- schur_example.m
```

Figure 2.1: User callable functions and examples of Scilab/MATLAB scripts.

2.1 Implementation choices

We have designed interfaces as similar as possible to the existing C and Fortran interfaces to `MUMPS`. For example, most parameters are grouped as components of a structure, and we offer the possibility to solve a sparse system of equations $\mathbf{Ax} = \mathbf{b}$ in three main steps (analysis, factorization, and solution), that can be called separately thanks to a parameter `JOB` (see below).

Complex and real arithmetics. The Scilab/MATLAB interfaces have been developed for real and complex arithmetics:

- the function `dmumps` must be called to solve real linear systems (corresponding to the double precision arithmetic in `MUMPS`);
- the function `zmumps` must be called to solve complex linear systems (corresponding to the complex double precision arithmetic in `MUMPS`).

In the following, we will use the notation `[dz]mumps` to refer to common features of `dmumps` and `zmumps`.

Data structures. Similarly to the C/Fortran interfaces, the Scilab/MATLAB interfaces consist in a callable subroutine `[dz]mumps` whose main argument is a structure (`mlist` in Scilab, `struct` in MATLAB). To fix the ideas, we will call this structure `id` in the rest of this document. Note that the names of the fields of the Scilab/MATLAB structure are the same as the names of the fields of the `MUMPS` C/Fortran structure (see structure of type `[DZ]MUMPS_STRUC` in `MUMPS` user's guide).

Runtime libraries. To interface `MUMPS`, which is mainly implemented in Fortran 90 but also offers a C interface, we have chosen to develop a Scilab/MATLAB interface in C. The main difficulty to link our interface with `MUMPS` is to provide the Fortran 90 runtime libraries corresponding to the compiled `MUMPS` package. Hence the user has to edit the makefile for the MATLAB interface (file `make.inc`) and/or the builder of the Scilab interface (file `builder.sce`). Note that some examples of paths to runtime libraries are given in the `MATLAB/INSTALL` directory for the MATLAB interface and in comments of the `SCILAB/builder.sce` file of the Scilab interface.

Matrix and vector parameters (matrix A, right-hand side b, solution x and Schur complement). A major difference between `MUMPS` interface and our new interfaces concerns the formats for the input matrix and right-hand side. Indeed, in the sequential version of `MUMPS` (case of an assembled matrix) the user supplies a matrix **A** by defining the following characteristics: the order of the matrix (`N`), the number of entries (`NZ`), the arrays containing the row and column indices for the matrix entries (`IRN` and `JCN`) and the array containing the numerical values (`A`). With the Scilab/MATLAB interfaces, in order to ease the use of `MUMPS`, the user should only provide the sparse matrix as a MATLAB or Scilab object. All the components above (`N`, `NZ`...) are automatically set within the interface. For the right-hand side, **b**, we simplified the interface similarly. If **b** is a full vector/matrix, the user has to provide a full Scilab/MATLAB vector/matrix in `id.RHS`, and if it is a sparse vector/matrix, the user has to provide a sparse Scilab/MATLAB vector/matrix in `id.RHS`. Then internal data, such as, for example, the number of right-hand sides or the parameter `ICNTL(20)` which specifies if the right-hand side is sparse, are automatically set within the interface.

Another difference is that we have chosen to create a variable `id.SOL` that contains the solution, instead of overwriting the right-hand side as this is done in the C/Fortran interfaces.

Also, the Schur complement matrix, if required, is allocated within the interface and returned as a Scilab/MATLAB dense matrix. Furthermore, the parameters `SIZE_SCHUR` and `ICNTL(19)` need not be set by the user; they are automatically set depending on the availability and size of the list of Schur variables (`id.VAR_SCHUR`).

Other indications.

- The Scilab/MATLAB interfaces only support `MUMPS` sequential version.
- Note that a problem of stack size may occur with Scilab. That is why before any use of this interface, we recommend to increase the stack size (type `help stacksize` within Scilab for more details).

- For efficiency issues we advise the user to use an optimized BLAS (ATLAS [12, 13], GOTO BLAS [9] or a vendor BLAS) whenever possible.

2.2 Parameters description

In this section we describe the control parameters of the interfaces and in particular the different components of the structure `id`.

Input Parameters

- **mat** : sparse matrix which has to be provided as the second argument of dmumps if `id.JOB` is strictly larger than 0.

- **id.SYM** : controls the matrix type (symmetric positive definite, symmetric indefinite or unsymmetric) and it has do be initialized by the user before the initialization phase of `MUMPS` (see `id.JOB`). Its value is set to 0 after the call of `initmumps`.

- **id.JOB** : defines the action that will be realized by `MUMPS`: initialize, analyze and/or factorize and/or solve and release `MUMPS` internal C/Fortran data. It has to be set by the user before any call to `MUMPS` (except after a call to `initmumps`, which sets its value to -1).

- **id.ICNTL and id.CNTL** : define control parameters that can be set after the initialization call (`id.JOB=-1`). See Section "Control parameters" of the `MUMPS` user's guide for more details. If the user does not modify an entry in `id.ICNTL` then `MUMPS` uses the default parameter. For example, if the user wants to use the AMD ordering, he/she should set `id.ICNTL(7)=0`. Note that the following parameters are inhibited because they are automatically set within the interface: `id.ICNTL(19)` which controls the Schur complement option and `id.ICNTL(20)` which controls the format of the right-hand side.

- **id.PERM_IN** : corresponds to the given ordering option (see Section "Input and output parameters" of the `MUMPS` user's guide for more details). Note that this permutation is only accessed if the parameter `id.ICNTL(7)` is set to 1.

- **id.COLSCA and id.ROWSCA** : are optional scaling arrays (see Section "Input and output parameters" of the `MUMPS` user's guide for more details)

- **id.RHS** : defines the right-hand side. The parameter `id.ICNTL(20)` related to its format (sparse or dense) is automatically set within the interface. Note that `id.RHS` is not modified (as in `MUMPS`), the solution is returned in `id.SOL`.

- **id.VAR_SCHUR** : corresponds to the list of variables that appear in the Schur complement matrix (see Section "Input and output parameters" of the `MUMPS` user's guide for more details).

Output Parameters

- **id.SCHUR** : if `id.VAR_SCHUR` is provided of size SIZE_SCHUR, then `id.SCHUR` corresponds to a dense array of size (SIZE_SCHUR,SIZE_SCHUR) that holds the Schur complement matrix (see Section "Input and output parameters" of the `MUMPS` user's guide for more details). The user does not have to initialize it.

- **id.INFO and id.RINFO** : information parameters (see Section "Information parameters" of the `MUMPS` user's guide).

- **id.SYM_PERM** : corresponds to a symmetric permutation of the variables (see discussion regarding `ICNTL(7)` in Section "Control parameters" of the `MUMPS` user's guide). This permutation is computed during the analysis and is followed by the numerical factorization except when numerical pivoting occurs.

- **id.UNS_PERM** : column permutation (if any) on exit from the analysis phase of `MUMPS` (see discussion regarding `ICNTL(6)` in Section "Control parameters" of the `MUMPS` user's guide).

- **id.SOL** : dense vector or matrix containing the solution after `MUMPS` solution phase.

Internal Parameters

- id.INST: (`MUMPS` reserved component) `MUMPS` internal parameter.

- id.TYPE: (`MUMPS` reserved component) defines the arithmetic (complex or double precision).

We refer the reader to the `MUMPS` user's guide and also to the examples included in the interfaces' package for more details on the use of these components.

2.3 Calling sequence

The usage of the interfaces reflects the `MUMPS` three-phase approach. The calling sequences of the main functions `initmumps`, `dmumps` and `zmumps` are:

```
id = initmumps;
id = dmumps(id [,mat] );
id = zmumps(id [,mat] );
```

and they are illustrated in Figure 2.2. Firstly a `MUMPS` structure has to be created. This is done via a call to `id=initmumps`, and a new structure `id` is built. Then the initialization part of the `MUMPS` is performed, thanks to a call of the form `id=[dz]mumps(id)`, where `id.JOB=-1` (default value after `initmumps`). The choice of the solver (`id.SYM=0,1, or 2`) should be done before this call, that also sets the control parameters to their default values. Note that each `MUMPS` instance is associated to a single matrix. Thus if the user wants to have multiple `MUMPS` factorizations available at the same time, he/she should define several Scilab/MATLAB instances `id1`, `id2`, ...

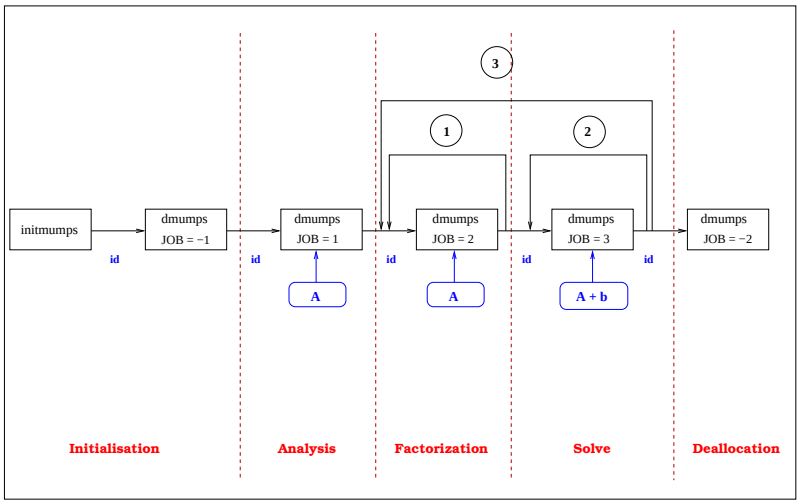

Figure 2.2: A typical calling sequence for real arithmetic with the three steps: analysis, factorization and solve.

The calling sequence of Figure 2.2 shows that the three phases (analysis, factorization, solve) can be called separately, if needed. This can be done thanks to the parameter `id.JOB`:

- if `id.JOB=1`, `MUMPS` will perform an analysis;
- if `id.JOB=2`, `MUMPS` will perform a factorization;
- if `id.JOB=3`, `MUMPS` will perform a solution phase;
- if `id.JOB=4`, `MUMPS` will perform an analysis followed by a factorization;
- if `id.JOB=5`, `MUMPS` will perform a factorization followed by a solution phase;
- if `id.JOB=6`, `MUMPS` will perform all three phases.

Moreover a factorization can be performed several times (see circled point 1 in Figure 2.2) with the same analysis (only if the pattern of the matrix did not change). It is also possible to perform several solution phases (see circled point 2 on Figure 2.2) with successive right-hand sides. In that case the interfaces use the previously computed factors. Note that two special cases are `id.JOB=-1` (for the initialization of an instance of the package), and `id.JOB=-2`, for the termination of an instance (deallocation of the factors and all internal data related to that instance).

At each step, the structure `id` is both an input and an output parameter, except for its initialization (call to `initmumps`) where the structure is first built. The matrix, $\mathbf{A}$, has to be supplied to the analysis, the factorization and the solution phases. The right-hand side, $\mathbf{b}$, has to be provided only to the solution phase (it is given through the parameter `id.RHS`). Figure 2.3 and 2.4 respectively illustrate the usage of `MUMPS` in MATLAB and of `MUMPS` Schur complement functionality in Scilab.

```
% initialisation of a matlab MUMPS structure
id = initmumps;
% here JOB = -1, the call to MUMPS will initialize C
% and fortran MUMPS structure
id = dmumps(id);
% load a sparse matrix
load lhr01;
mat = Problem.A;
% JOB = 6 means analysis+facto+solve
id.JOB = 6;
id = dmumps(id,mat);
% check the computed solution
berr = norm(mat*id.SOL - ones(size(mat,1),1),'inf') / ...
       (norm(mat,'inf') * norm(id.SOL,'inf') + 1);
if(berr > sqrt(eps))
        disp('WARNING : precision may not be OK');
else
        disp('SOLUTION OK');
end
norm(mat*id.SOL - ones(size(mat,1),1),'inf')
% destroy mumps instance
id.JOB = -2;
id = dmumps(id)
```

Figure 2.3: A simple example of using `MUMPS` in double precision arithmetic in MATLAB.

```
//------------------------ INITIALISATION --------------------//
// load the matrix
exec('ex2.sci');
mat=sparse(a);
n = size(mat,1);
themax = max(max(abs(mat)));
mat = mat+sparse([1:n;1:n]',3*themax*ones(1,n));
[id]=initmumps();
//Job=-1: the call to dmumps will initialise the internal C and Fortran structures
[id]=dmumps(id);
id.RHS=ones(n,1);
// Schur corresponds to the last 5 variables:
id.VAR_SCHUR = [n-4:n];
//------------------------ RESOLUTION ------------------------//
// We want to use the Schur complement to solve A x sol = rhs
// with sol = x   , rhs = rhs1 , A = A_{1,1} A_{1,2}
//            y           rhs2       A_{2,1} A_{2,2}
// and S = A_{2,2} - A_{2,1} x A_{1,1}^{-1} x A_{1,2} the Schur complement
// computed by MUMPS in the previous call and stored in id.SCHUR
// We have y = S^(-1) x (rhs2 - A_{2,1} x A_{1,1}^(-1) x rhs1)
// and x = A_{1,1}^(-1) x (rhs1 - A_{1,2} x y)
// job = 6: perform analysis, factorization plus computation of
// Schur complement and solve the system  A_{1,1} sol1 = rhs1
id.JOB=6;[id]=dmumps(id,mat);
sol1 = id.SOL(1:n-5);
// Set rhsy = rhs2 - A_{2,1} x A_{1,1}^(-1) x rhs1
rhsy = ones(5,1)-mat(n-4:n,1:n-5)*sol1;
//------------------------
// TO MODIFY :
// usually the resolution below is replaced by an iterative scheme
y = id.SCHUR \ rhsy;
//------------------------
// Set rhsx =  A_{1,2} * y
rhsx = mat(1:n-5,n-4:n)*y;
// Solve the linear system A_{1,1} rhsx = A_{1,2} * y
id.RHS(1:n-5) = rhsx;
id.JOB = 3; id = dmumps(id,mat);
rhsx = id.SOL(1:n-5);
x = sol1-rhsx;
// assemble solution
sol = [x;y];
// realease mumps instance associated to id
id.JOB=-2;[id]=dmumps(id);
```

Figure 2.4: Example of using the MUMPS Schur complement feature in Scilab.

3 Experiments

After presenting our experimental environment, we present in this section some results obtained with our Scilab/MATLAB interface to `MUMPS`. Firstly, we compare it to the default backslash operation and to UMFPACK v4.4 [7, 8] MATLAB and Scilab interfaces. The UMFPACK MATLAB interface is provided within UMFPACK package and the UMFPACK Scilab has been developed by Bruno Pinçon (see `http://www.iecn.u-nancy.fr/~pincon/scilab/scilab.html`). Secondly, we illustrate the impact of the ordering used during the analysis for some of the matrices of our test set. Thirdly, we give an illustration of the use of the parameter `id.SYM` for symmetric matrices. Finally, we show the interest of allowing successive solution phases without a refactorization of the matrix.

3.1 Experimental environment

We split our test set into two parts: the real/complex unsymmetric matrices of Table 3.1.a and the real/complex symmetric matrices of Table 3.1.b. For each matrix we report its group, its name, its order, its number of nonzeros and the symmetry (number of nonzero entries (i, j) for which (j, i) is also in the pattern of the matrix divided by the total number of nonzeros). The matrices come from the University of Florida Sparse Matrix Collection [6], where matrices are provided in both MATLAB and Harwell-Boeing formats. Note that for the Scilab interface of `MUMPS`, in order to read the matrices in Harwell-Boeing format, we have used a function developed by Bruno Pinçon, ReadHBSparse available at `http://www.iecn.u-nancy.fr/~pincon/scilab/scilab.html`.

Group	Name	Order	Nnz	Sym
Real matrices				
Hollinger	jan99jac120	41374	260202	0.18
Bomhof	circuit_4	80209	307604	0.87
Hollinger	mark3jac140	64089	399735	0.22
HB	psmigr_3	3140	543162	0.48
Hollinger	g7jac200sc	59310	837936	0.10
ATandT	twotone	120750	1224224	0.28
Vavasis	av41092	41092	1683902	0.00
Complex matrices				
Okunbor	aft02	8184	127762	1.0
Bai	qc2534	2534	463360	1.0

(3.1.a) Unsymmetric matrices. Nnz: number of nonzeros.

Group	Name	Order	Nnz
Real matrices			
BOEING	ct20stif	52329	1375396
GHS_indef	helm2d03	392257	1567096
GHS_psdef	oilpan	73752	1835470
Complex matrices			
HB	young2c	841	4089
Cote	mplate	5962	74076

(3.1.b) Symmetric matrices. Nnz: number of nonzeros.

Table 3.1: Test set.

The platform used to run the tests is Linux PC with a 3.4 GHz Intel Xeon processor and 4 GB of main memory. We used Scilab Version 3.1.1 and MATLAB version 7.0.4. `MUMPS` has been compiled with g95 version 4.0.1 with the -O option. UMFPACK has been compiled with gcc compiler version 3.4.4 and the -O3 and -FPIC options.

3.2 General comparison

In this section, our Scilab/MATLAB interfaces are compared to UMFPACK (v4.4) interface and to the default backslash operation of MATLAB and/or Scilab. The purpose of this comparison is not to compare `MUMPS` with other solvers but to prove the efficiency and the correctness of our choices. We run all solvers with default parameters. Results are given in Tables 3.2 and 3.4 for Scilab and in Tables 3.3 and 3.5 for MATLAB. We report for each method:

- the total CPU time (in seconds) corresponding to the time of the analysis phase, the numerical factorization and the solving phase plus the overhead cost of the interface (`timer` in Scilab and `cputime` in MATLAB are used to get this statistic);
- the number of operations performed during factorization, as reported by the package; note that for complex matrices, UMFPACK reports floating-point operations while `MUMPS` reports the number of operations on complex numbers;
- the number of nonzeros in the factors, as reported by the package.

We observe that our approach is competitive compared to both UMFPACK and the Scilab/MATLAB backslash operators. With default options, `MUMPS` is sometimes faster and sometimes slower than UMFPACK. Notice that the backslash operator of Scilab is far behind the other options in terms of performance. It has to be noted that the backslash operator of MATLAB sometimes uses UMFPACK (although not systematically) while Scilab uses older codes.

There are several differences between the statistics for the Scilab and the MATLAB interfaces. The difference in terms of CPU time between MATLAB and Scilab is due to the fact that both of these packages use different BLAS libraries. However, the ratio of total CPU time between `MUMPS` and UMFPACK is the same for the MATLAB and Scilab interfaces. For example, on the matrix Okunbor/qc2534.cua, this ratio is equal to 0.98 for Scilab and to 0.97 for MATLAB; on the matrix BOEING/ct20stif.rsa, the difference between the two ratios is 0.0026. We note that the default BLAS library used in the case of Scilab is a precompiled version of ATLAS (Linux binary package) that may not be fully optimized for the platform, whereas, by default, MATLAB uses the BLAS from the MKL library. While we found it difficult to modify the BLAS version used by Scilab, we did that in MATLAB easily by redefining the environment variable `BLAS_VERSION`. If we force MATLAB to use the same BLAS as Scilab, we obtain the results of Tables A.1 and A.2 for unsymmetric and symmetric matrices respectively. For `MUMPS` and UMFPACK, those results are now similar in terms of performance to the results from Scilab (see Tables 3.2 and 3.4). It confirms that performance differences between MATLAB and Scilab are due to the default choice for the BLAS library.

Concerning the number of operations performed during the factorization and the number of entries in the factors, the variation between MATLAB and Scilab can be explained by the default ordering. In fact, most of these results have been obtained with `MeTiS` and the differences are due to the use of random numbers within that package.

	MUMPS	UMFPACK	Scilab
Hollinger jan99jac120			
total CPU time	3.20	1.83	473.01
ops ($\times 10^6$)	2424.73	508.06	-
nz LU ($\times 1000$)	6005.6	2101.0	-
Bomhof circuit_4			
total CPU time	0.99	2.34	32.05
ops ($\times 10^6$)	10.84	7.94	-
nz LU ($\times 1000$)	510.4	436.6	-
Hollinger mark3jac140			
total CPU time	8.32	65.93	*
ops ($\times 10^6$)	8905.51	74731.19	-
nz LU ($\times 1000$)	19170.7	41594.4	-
HB psmigr_3			
total CPU time	6.72	7.03	806.39
ops ($\times 10^6$)	8925.92	8578.87	-
nz LU ($\times 1000$)	6416.1	5823.7	-
Hollinger g7jac200sc			
total CPU time	27.46	41.58	*
ops ($\times 10^6$)	30669.36	45142.68	-
nz LU ($\times 1000$)	34326.8	36186.6	-
Atandt twotone			
total CPU time	26.28	4.61	*
ops ($\times 10^6$)	36543.22	3709.94	-
nz LU ($\times 1000$)	28548.8	6769.7	-
Vavasis av41092			
total CPU time	15.12	40.51	*
ops ($\times 10^6$)	7506.37	29180.37	-
nz LU ($\times 1000$)	15157.0	34407.5	-
Bai aft02			
total CPU time	0.35	0.32	156.99
ops ($\times 10^6$)	40.91	146.80 (a)	-
nz LU ($\times 1000$)	688.7	592.9	-
Okundor qc2534			
total CPU time	0.91	0.93	73.45
ops ($\times 10^6$)	219.70	638.49 (a)	-
nz LU ($\times 1000$)	1034.7	888.4	-

Table 3.2: Scilab results for unsymmetric matrices. CPU time in seconds. ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands); *: failure; - : not available; (a): number of floating-point operations and not number of operations on complex numbers.

	MUMPS	UMFPACK	MATLAB
Hollinger jan99jac120			
total CPU time	2.78	1.50	2.24
ops ($\times 10^6$)	2445.75	505.03	-
nz LU ($\times 1000$)	6034.4	2093.3	-
Bomhof circuit_4			
total CPU time	0.99	2.36	4.23
ops ($\times 10^6$)	10.80	7.94	-
nz LU ($\times 1000$)	496.4	436.6	-
Hollinger mark3jac140			
total CPU time	6.11	34.68	39.20
ops ($\times 10^6$)	8420.09	74337.81	-
nz LU ($\times 1000$)	18826.1	41419.1	-
HB psmigr_3			
total CPU time	4.31	4.88	5.86
ops ($\times 10^6$)	8925.92	8578.87	-
nz LU ($\times 1000$)	6416.1	5823.7	-
Hollinger g7jac200sc			
total CPU time	20.78	23.49	27.91
ops ($\times 10^6$)	35416.76	45008.21	-
nz LU ($\times 1000$)	36566.5	36120.1	-
Atandt twotone			
total CPU time	15.62	3.60	5.20
ops ($\times 10^6$)	37598.09	3634.11	-
nz LU ($\times 1000$)	28831.3	6664.5	-
Vavasis av41092			
total CPU time	12.66	23.59	28.32
ops ($\times 10^6$)	7488.51	28651.36	-
nz LU ($\times 1000$)	15073.6	34098.7	-
Bai aft02			
total CPU time	0.32	0.33	0.49
ops ($\times 10^6$)	42.58	146.80 (a)	-
nz LU ($\times 1000$)	694.9	592.9	-
Okundor qc2534			
total CPU time	0.74	0.78	0.33
ops ($\times 10^6$)	219.70	638.49 (a)	-
nz LU ($\times 1000$)	1034.7	888.4	-

Table 3.3: MATLAB results for unsymmetric matrices. CPU time in seconds; ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands); - : not available; (a): number of floating-point operations and not number of operations on complex numbers.

	MUMPS	UMFPACK	Scilab
BOEING ct20stif			
total CPU time	7.36	11.50	*
ops ($\times 10^6$)	9416.34	14254.96	-
nz LU ($\times 1000$)	20159.7	21410.2	-
GHS_indef helm2d03			
total CPU time	13.97	27.10	*
ops ($\times 10^6$)	8675.65	27986.95	-
nz LU ($\times 1000$)	39861.4	56474.4	-
GHS_psdef oilpan			
total CPU time	5.33	4.17	1753.64
ops ($\times 10^6$)	2760.47	3661.89	-
nz LU ($\times 1000$)	11631.1	11635.7	-
HB young2c			
total CPU time	0.02	0.01	0.52
ops ($\times 10^6$)	0.38	1.24 (a)	-
nz LU ($\times 1000$)	19.3	17.6	-
Cote mplate			
total CPU time	4.77	21.53	919.47
ops ($\times 10^6$)	1568.60	33717.53 (a)	-
nz LU ($\times 1000$)	2966.8	7961.9	-

Table 3.4: Scilab results for symmetric matrices. CPU time in seconds; ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands); *: failure; - : not available; (a): number of floating-point operations and not number of operations on complex numbers.

	MUMPS	UMFPACK	MATLAB
BOEING ct20stif			
total CPU time	5.14	8.18	65.50
ops ($\times 10^6$)	9546.08	14254.96	-
nz LU ($\times 1000$)	20384.2	21410.2	-
GHS_indef helm2d03			
total CPU time	12.29	18.91	33.55
ops ($\times 10^6$)	8675.65	27986.95	-
nz LU ($\times 1000$)	39861.4	56474.4	-
GHS_psdef oilpan			
total CPU time	4.74	3.34	13.76
ops ($\times 10^6$)	2755.79	3661.89	-
nz LU ($\times 1000$)	11609.8	11635.7	-
HB young2c			
total CPU time	0.01	0.02	0.02
ops ($\times 10^6$)	0.38	1.24 (a)	-
nz LU ($\times 1000$)	19.3	17.6	-
Cote mplate			
total CPU time	3.40	14.81	15.70
ops ($\times 10^6$)	1501.80	33717.52 (a)	-
nz LU ($\times 1000$)	2959.8	7961.9	-

Table 3.5: MATLAB results for symmetric matrices. CPU time in seconds; ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands); - : not available; (a): number of floating-point operations and not number of operations on complex numbers.

3.3 Illustration of sensitivity to orderings

This section discusses the possible choices of ordering heuristic. We select six matrices among our test set of Tables 3.1.a and 3.1.b and use the Scilab interface. Results are given in Table 3.6. We report for each ordering the total CPU time and the number of operations performed during the factorization. We present performance with the following orderings:

- the Approximate Minimum Degree (`AMD`)[2];
- the Approximate Minimum Fill (`AMF`);
- the `PORD` package [11];
- the `MeTiS` package[2] [10];
- `QAMD`, an Approximate Minimum Degree with automatic quasi-dense row detection [1].

	AMD	AMF	PORD	MeTiS	QAMD
Hollinger jan99jac120					
total CPU time	2.35	2.41	4.70	3.21	4.42
ops ($\times 10^6$)	2011.25	1464.47	2877.57	2424.73	5488.06
Hollinger g7jac200sc					
total CPU time	35.62	25.01	33.89	27.51	35.80
ops ($\times 10^6$)	53194.63	27322.45	39468.22	30669.36	53194.63
Atandt twotone					
total CPU time	22.72	22.89	29.06	26.24	25.91
ops ($\times 10^6$)	33366.50	30743.70	41141.29	36543.22	41743.53
GHS _indef helm2d03					
total CPU time	21.64	16.80	16.33	14.64	21.75
ops ($\times 10^6$)	28876.25	19775.68	8741.06	8675.65	28876.25
GHS _psdef oilpan					
total CPU time	3.51	3.28	5.64	5.71	3.51
ops ($\times 10^6$)	3762.24	3289.24	2898.63	2760.47	3762.24
Okundor qc2534					
total CPU time	0.88	0.91	1.37	1.38	0.92
ops ($\times 10^6$)	219.84	219.70	434.27	427.80	219.84

Table 3.6: Scilab results with `MUMPS` and different orderings. CPU time in seconds; ops: number of operations for factorization (millions).

We can see that the choice of the ordering is useful for users who want to tune `MUMPS` (memory and/or factorization time and/or solution time...) for their particular application.

For example, for the matrix ATandT/twotone.rua, the default ordering chosen by `MUMPS` is `MeTiS`. We see that, when the Approximate Minimum Degree (`AMD`) is used, the CPU time and number of operations decrease by 13.48% and 8.69%, respectively. This difference between `MeTiS` and `AMD` represents more than 3.10^9 operations.

[2] See http://www-users.cs.umn.edu/~karypis/metis/

3.4 Illustration for symmetric matrices

This section gives an illustration of the initialization of the parameter `id.SYM` (see Section 2.2). This parameter controls the matrix properties and has to be initialized before calling `[dz]mumps` with `id.JOB=-1`. It has three possible values:

- `id.SYM = 0` means that **A** is treated as unsymmetric (default value after the call to `initmumps`);
- `id.SYM = 1` means that **A** is treated as symmetric positive definite;
- `id.SYM = 2` means that **A** is treated as general symmetric.

	sym=0	sym=2
BOEING ct20stif		
total CPU time	7.36	5.64
ops ($\times 10^6$)	9416.34	4758.96
nz LU ($\times 1000$)	20159.7	10135.6
GHS_indef helm2d03		
total CPU time	13.97	13.08
ops ($\times 10^6$)	8675.65	4403.60
nz LU ($\times 1000$)	39861.4	20128.7
GHS_psdef oilpan		
total CPU time	5.33	5.03
ops ($\times 10^6$)	2760.47	1417.65
nz LU ($\times 1000$)	11631.1	5976.7
HB young2c.csa		
total CPU time	0.02	0.01
ops ($\times 10^6$)	0.38	0.22
nz LU ($\times 1000$)	19.3	10.1
Cote mplate.csa		
total CPU time	4.77	2.90
ops ($\times 10^6$)	1568.60	915.31
nz LU ($\times 1000$)	2966.8	1779.2

Table 3.7: Scilab results with `MUMPS` for symmetric matrices, with parameter `id.SYM` initialized to 0 and 2. CPU time in seconds; ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands).

Table 3.7 compares the results obtained with the Scilab interface on the symmetric matrices with `id.SYM = 0` and `id.SYM = 2`. We see that, by taking into account the symmetry of the matrix (and initializing `id.SYM` accordingly), the number of operations and the number of non zeros in the factors are nearly twice smaller. The total CPU time also tends to decrease.

Remark. The matrix GHS_psdef/oilpan is in fact positive definite. Therefore, instead of `id.SYM`=2, we can set `id.SYM=1` to specify that no pivoting is necessary. In that case, we obtain a total CPU time of 4.46 seconds, with 1.418×10^9 operations and 5.977×10^6 entries in the factors.

3.5 Illustration of multiple solution steps

In this section we present the results obtained with the MATLAB interface of `MUMPS`. We report for each matrix the total CPU time and the CPU time for the solve step (see Tables 3.8.aand 3.8.b). Note that in this table, the CPU time for solve reported for UMFPACK is the one returned in `Info(86)` by the call `[x,Info]=umfpack(A,"\",b)` For `MUMPS`, we used the function `cputime` of MATLAB.

	`MUMPS`	UMFPACK
Hollinger jan99jac120		
total CPU time	2.78	1.50
CPU time for solve	0.08	0.07
Bomhof circuit_4		
total CPU time	0.99	2.36
CPU time for solve	0.09	0.06
Hollinger mark3jac140		
total CPU time	6.11	34.68
CPU time for solve	0.11	0.68
HB psmigr_3		
total CPU time	4.31	4.88
CPU time for solve	0.03	0.11
Hollinger g7jac200sc		
total CPU time	20.78	23.49
CPU time for solve	0.17	0.61
Atandt twotone		
total CPU time	15.62	3.60
CPU time for solve	0.20	0.25
Vavasis av41092		
total CPU time	12.66	23.59
CPU time for solve	0.11	0.59
Bai aft02		
total CPU time	0.32	0.33
CPU time for solve	0.02	0.09
Okundor qc2534		
total CPU time	0.74	0.78
CPU time for solve	0.02	0.19

(3.8.a) Unsymmetric matrices.

	`MUMPS`	UMFPACK
Okundor qc2534		
total CPU time	5.14	8.18
CPU time for solve	0.12	0.53
Okundor qc2534		
total CPU time	12.29	18.91
CPU time for solve	0.48	1.28
Okundor qc2534		
total CPU time	4.74	3.34
CPU time for solve	0.11	0.34
Okundor qc2534		
total CPU time	0.01	0.02
CPU time for solve	0.01	0.01
Okundor qc2534		
total CPU time	3.40	14.81
CPU time for solve	0.03	0.30

(3.8.b) Symmetric matrices.

Table 3.8: MATLAB results for all matrices. CPU times in seconds.

Once the factors have been computed, there is a clear advantage not refactoring the matrix when a succession of solution has to be performed. This demonstrates the advantage of offering this possibility within the interface over the standard backslash operator. An example of application is the inverse power method (when one is looking for the smallest eigenvalue of a problem), where a succession of solution steps with the same matrix has to be performed. Another example could be the optimization of a quadratic function with the Newton method, where a succession of solution steps involving the Hessian matrix must be performed.

4 Concluding remarks

We have presented an interface to MUMPS for both Scilab and MATLAB environments and have illustrated its interest on a range of sparse matrices. More functionalities are offered than the standard backslash (\) operator from these environments that can be very useful depending on the users applications. Also, we have shown that performance can be critical even in these convivial environments, and that using state-of-the-art solvers is useful. It is also critical to use an efficient BLAS library.

Concerning the Scilab interface, a further step of integration would consist in overloading the backslash operator and use an efficient solver as the default, although this may require distributing Fortran runtime libraries in the binary version. Of course less functionalities are available, but this is simpler for the user. One could detect automatically some properties of the matrix (as is done in MATLAB) to choose the appropriate solver or solver options.

However, one of the main drawback of only using the backslash operator is that a matrix **A** must be refactored each time a solution to a system of equations involving **A** is requested. For example if the operation `x1=A\x0` is performed, followed by `x2=A\x1`, ..., then each of these operations requires to execute the three steps of the direct solver (analysis, factorization, solve) each time whereas the interface described above offers the flexibility to do this more efficiently. It would be nice to also have this possibility with the backslash operator.

This could be done by having information `IsFactored` telling if a matrix has already been factored and a pointer `SolverPtr` associated to data structures allowing to reuse the previously computed factors (and other solver information) from a given matrix. In the Scilab case, those parameters could be added to the sparse matrix structure below.

```
typedef struct scisparse {
  integer m,n,it,nel ; /* nel : number of non nul elements */
  integer *mnel,*icol; /* mnel[i]: number of non nul elements of row i, size m
                        * icol[j]: column of the j-th non nul element, size nel
                        */
  double *R,*I ; /* R[j]: real value of the j-th non nul element, size nel
                  * I[j]: imag value of the j-th non nul element, size nel
                  */
} SciSparse ;
```

The first time `A\b` is performed on a matrix **A**, `IsFactored` must be set to true (or nonzero) and the pointer must be associated to the solver data structures. Then in subsequent calls, a check on `IsFactored` allows to reuse the previous factors. Of course, `IsFactored` must be reset to false each time the matrix is modified, and the computed factors must also be freed (termination routine from the solve). Note also that when Scilab runs out of memory, one could decide to free the factors corresponding to matrices on which the backslash operator has not been used for a long time (using, for example a *Least Recently Used policy*). We plan to discuss these issues with the developers of Scilab in the future.

Similarly we could also add a `IsAnalysed` flag which would be true if a valid symbolic factorization is available for the considered matrix. Thus the first time `A\b` is performed on a matrix **A**, `IsAnalysed` would be set to true (or nonzero). Then in subsequent calls, if `IsFactored` is false and `IsAnalysed` is true, it allows to reuse the previous symbolic factorization, i.e., call `MUMPS` with `id.JOB=5`. `IsAnalysed` would be reset to false each time the pattern of the matrix is modified.

To conclude, note that the MATLAB and Scilab interfaces we have described in this document correspond to the needs of some users of the `MUMPS` package. They are now distributed within `MUMPS` (see http://graal.ens-lyon.fr/MUMPS or http://www.enseeiht.fr/apo/MUMPS for more information on availability). We have also found these interfaces useful in experimenting new developed features, in building scripts to check the evolution of the sequential performance from one version to the other, and in validating some of the functionalities of the package.

A Auxiliary results

	MUMPS	UMFPACK	MATLAB
Hollinger jan99jac120			
total CPU time	3.34	1.89	2.61
CPU time for solve	0.10	0.07	-
ops ($\times 10^6$)	2445.75	503.25	-
nz LU ($\times 1000$)	6034.4	2089.8	-
Bomhof circuit_4			
total CPU time	1.03	2.44	4.24
CPU time for solve	0.15	0.07	-
ops ($\times 10^6$)	10.80	7.94	-
nz LU ($\times 1000$)	496.4	436.6	-
Hollinger mark3jac140			
total CPU time	8.31	67.65	72.18
CPU time for solve	0.16	0.66	-
ops ($\times 10^6$)	8420.09	74105.82	-
nz LU ($\times 1000$)	18826.1	41267.7	-
HB psmigr_3			
total CPU time	6.85	7.53	8.57
CPU time for solve	0.05	0.11	-
ops ($\times 10^6$)	8925.92	8578.87	-
nz LU ($\times 1000$)	6416.1	5823.7	-
Hollinger g7jac200sc			
total CPU time	30.76	43.01	47.42
CPU time for solve	0.23	0.60	-
ops ($\times 10^6$)	35416.76	44856.40	-
nz LU ($\times 1000$)	36566.5	36052.3	-
Atandt twotone			
total CPU time	26.97	4.97	6.57
CPU time for solve	0.28	0.24	-
ops ($\times 10^6$)	37598.09	3657.24	-
nz LU ($\times 1000$)	28831.3	6694.8	-
Vavasis av41092			
total CPU time	14.90	41.41	46.12
CPU time for solve	0.13	0.61	-
ops ($\times 10^6$)	7488.51	28476.60	-
nz LU ($\times 1000$)	15073.6	34050.3	-
Bai aft02			
total CPU time	0.37	0.35	0.51
CPU time for solve	0.03	0.06	-
ops ($\times 10^6$)	42.58	146.80 (a)	-
nz LU ($\times 1000$)	694.9	592.9	-
Okundor qc2534			
total CPU time	0.98	1.01	0.65
CPU time for solve	0.03	0.19	-
ops ($\times 10^6$)	219.70	638.49 (a)	-
nz LU ($\times 1000$)	1034.7	888.4	-

Table A.1: MATLAB results for unsymmetric matrices with the same BLAS as Scilab. CPU times in seconds; ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands); - : not available; (a): number of floating-point operations and not number of operations on complex numbers.

	MUMPS	UMFPACK	MATLAB
BOEING ct20stif			
total CPU time	7.61	12.51	65.62
CPU time for solve	0.15	0.52	-
ops ($\times 10^6$)	9546.08	14254.96	-
nz LU ($\times 1000$)	20384.2	21410.2	-
GHS_indef helm2d03			
total CPU time	14.10	27.53	42.19
CPU time for solve	0.59	1.28	-
ops ($\times 10^6$)	8675.65	27986.95	-
nz LU ($\times 1000$)	39861.4	56474.4	-
GHS_psdef oilpan			
total CPU time	5.41	4.45	13.77
CPU time for solve	0.14	0.35	-
ops ($\times 10^6$)	2755.79	3661.89	-
nz LU ($\times 1000$)	11609.8	11635.7	-
HB young2c			
total CPU time	0.01	0.02	0.02
CPU time for solve	0.01	0.01	-
ops ($\times 10^6$)	0.38	1.24 (a)	-
nz LU ($\times 1000$)	19.3	17.6	-
Cote mplate			
total CPU time	5.23	24.66	25.91
CPU time for solve	0.05	0.29	-
ops ($\times 10^6$)	1501.80	33717.51 (a)	-
nz LU ($\times 1000$)	2959.8	7962.0	-

Table A.2: MATLAB results for symmetric matrices with the same BLAS as Scilab. CPU times in seconds; ops: number of operations for factorization (millions); nz LU: number of entries in the factors (thousands); - : not available; (a): number of floating-point operations and not number of operations on complex numbers.

References

[1] P. R. Amestoy. Recent progress in parallel multifrontal solvers for unsymmetric sparse matrices. In *Proceedings of the 15th World Congress on Scientific Computation, Modelling and Applied Mathematics, IMACS 97, Berlin*, 1997.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1996.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[5] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 2005. To appear.

[6] T. A. Davis. University of Florida sparse matrix collection, 2002. http://www.cise.ufl.edu/research/sparse/matrices.

[7] T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

[8] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, 2004.

[9] K. Goto and R. Geijn. On reducing TLB misses in matrix multiplication, 2002. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note 9.

[10] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, September 1998.

[11] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.

[12] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. http://www.cs.utsa.edu/~whaley/papers/spercw04.ps.

[13] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399