HAL
open science

Implementation of Lipsol in Scilab

Héctor E. Rubio Scola

To cite this version:

Héctor E. Rubio Scola. Implementation of Lipsol in Scilab. [Research Report] RT-0215, INRIA. 1997, pp.59. inria-00069956

HAL Id: inria-00069956

https://inria.hal.science/inria-00069956v1

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implementation of Lipsol in Scilab

Héctor E. Rubio Scola

No 0215

Décembre 1997

THÈME 4

Rapport technique

ISSN 0249-0803

Implementation of Lipsol in Scilab

Héctor E. Rubio Scola

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Meta2

Rapport technique n° 0215 — Décembre 1997 — 59 pages

Abstract: We show how the LIPSOL library (Linear programming Interior-Point SOLvers) can be used in Scilab. To do this, fast Scilab-Fortran interfaces for sparse Cholesky decomposition have been developed. All other calculations are made using Scilab functions, which take into account sparsity for storing and factorizing the matrices. We make use of structural matrices algorithms from Metanet (Scilab toolbox for graphs and networks computations), in particular the Dulmage Mendelsohn algorithm.

Key-words: Linear programming, Interior Point methods, Mehrotra predictor-corrector algorithm, bipartite graphs, maximum matching, Dulmage-Mendelsohn decomposition.

(Résumé : tsvp)

* CIUNR - FCEIA - FCEE - Universidad Nacional de Rosario, Argentina.

Unit´e de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
T´el´ephone : 01 39 63 55 11 - International : +33 1 39 63 55 11
T´el´ecopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Mise en œuvre de Lipsol dans Scilab

Résumé : On montre l'adaptation de la bibliothèque LIPSOL (Linear programming Interior-Point SOLvers) au système Scilab. Pour réaliser cet objectif, on a développé les interfaces Scilab-Fortran pour la décomposition de Cholesky des matrices creuses. Tous les autres calculs sont faits en utilisant les fonctions Scilab, et ils tiennent compte de la structure creuse des matrices. On utilise les matrices structurées de Metanet (toolbox de Scilab pour les calculs des graphes et des réseaux), en particulier, l'algorithme de Dulmage-Mendelsohn.

Mots-clé : Linear programming, Interior Point methods, Mehrotra's predictor-corrector algorithm, bipartite graphs, maximum matching, Dulmage-Mendelsohn decomposition.

Contents

1 Introduction **5**

2 Installing Scilab Lipsol **5**

3 LIPSOL Linear programming Interior Point SOLvers **5**
3.1 What is LIPSOL? 6
3.2 How to use lipsol ? 6
3.3 Main functions 7
3.4 Main steps of calculation 11

4 Sparse matrix representantion **12**
4.1 SPARSPAK (A Sparse Matrix Package). Symmetric matrix data structure 12
4.2 Metanet 15
4.3 SCILAB sparse matrices 16
4.4 Conversion functions 18
4.5 Loadmps - reads a file in MPS format 21

5 Algorithms **25**
5.1 Spchol function 25
5.2 Scilab Dmperm Function 28
5.3 Sprank Function. 32
5.4 Dulmage and Mendelsohn decomposition of a matrix 33
5.5 Fullsprank Function 41
5.6 Fullsprank vs. fullrank 45
5.7 Lipsol algorithm: basic steps 47

6 Appendix **49**
6.1 Steps of lipsol calculation 49
6.1.1 Preprocess description 49
6.1.2 Calculus description 53
6.2 Fortran-Scilab Interfaces 56
6.3 Elementary Functions 57
6.4 Fortran Subroutines 57

6.4.1 Spchol Function 57

7 Conclutions **58**

1 Introduction

The objective of this work is to adjust the LIPSOL software [10] to Scilab system. LIPSOL is a sparse linear programming solver based on interior point methodes. The Matlab implementation is done by a number of Matlab functions and large Fortran programs, mainly a sparse Cholesky factorization (4.1). The Scilab implementation which is described here has required significant work since the Scilab representation of sparse matrices is not the same as the Matlab one. Also, some specific tools such as the Dulmage Mendelsohn algorithm have been developed using the Metanet toolbox. To do that, fast Scilab-Fortran interfaces have been made to calculate Cholesky decomposition, as well as a Scilab function spchol (4.1). All the other calculations are made with Scilab functions. In particular, the Dulmage Mendelsohn (DM) algorithm was programmed using METANET to have the `sprank` (4.3), `fullsprank` (4.5) and `dmperm` functions (4.2).

2 Installing Scilab Lipsol

Obtain a copy of LIPSOL from the web at ftp.inria.fr: INRIA/ Projects/ Meta2/ Scilab/ contrib/ lipsol/ LIPSOL.tar.gz and decompress it with:

`gunzip LIPSOL.tar.gz` and `tar -xvf LIPSOL.tar`

in the subdirectory where you want to install the LIPSOL main directory. This will create the LIPSOL subdirectory structure (see "Contents" below).

Read and folow the README instructions

3 LIPSOL Linear programming Interior Point SOLvers

In this report, we adress the problem of solving large scale linear programs expressed in the following so-called Standard Form:

$$\begin{array}{ll} \text{minimize} & c \times x \\ \text{subject to} & A \times x = b \\ & x \geq 0 \end{array}$$

here $x \in \Re^n$ is the unknown, A is a matrix in $\Re^{m*n}, m < n$, and $b \in \Re^m$, $c \in \Re^n$, $c \times x$ is the objective function, and the equations $A \times x = b$ are the constraints. All these entities must have consistent dimensions, of course. Usually A has more columns than rows, and $A \times x = b$ is therefore quite likely to be under-determined, leaving great latitude in the choice of x which minimizes $c \times x$.

3.1 What is LIPSOL?

LIPSOL (Linear programming Interior-Point SOLvers) is a package that uses Matlab's sparse-matrix data structure and MEX (Fortran-Matlab interfaces files) utilities to achieve both program simplicity and computational efficiency. The Fortran subroutines are from SPARSPAK (Sparse Matrix Package). The internal storage structure of SPARSPAK is different from Scilab's, therefore the corresponding changes were made (`splget` function). The objective of using sparse matrix techniques for solving linear systems is to reduce costs by exploiting sparsity. It is possible to achieve drastic reductions in storage and arithmetic requirements when comparing the solutions of dense and tridiagonal systems.

3.2 How to use lipsol ?

The notations used througout are the following:

minimize $c \times x$
subject to $A \times x \leq b$
lbounds $\leq x \leq$ ubounds

here $x \in \Re^n$ is the unknown, A is a matrix in $\Re^{m*n}, m < n$, and $b \in \Re^m$, $c \in \Re^n$, $c \times x$ is the objective function, and the equations $A \times x = b$ are the constraints.

Lipsol can be used by two ways. Directly by calling the `lipsol` Scilab function or by means of the `splinpro` function. The difference is how to take to in account the constraints.

Let us describe now in more detail the functions `lipsol` and `splinpro`.The functions which have been implemented, including `lipsol` and `splinpro`, have

an `help` manual, and the following subsections are made of the `lipsol` and `splinpro` manuals.

3.3 Main functions

Lipsol function
CALLING SEQUENCE

```
[x,obj]  = lipsol(A,b,c,lbounds,ubounds,BIG [,fr] )
```

PARAMETERS

A	:	sparse real matrix (constraints)
b	:	sparse column vector (constraints)
c	:	sparse column vector (objective)
lbounds	:	sparse column vector of lower bounds
ubounds	:	sparse column vector of upper bounds
BIG	:	large real number
fr	:	real number, if the rank A < m, it must be fr = 1
x	:	sparse column vector solution
obj	:	real number,value of objetive function at x.

DESCRIPTION

Solver for the sparse linear program
min $c' \times x$
s.t. $A \times x = b$
lbounds $\leq x \leq$ ubounds, BIG is a large number such that ubounds(i) = BIG if no upper-bound for x(i) exists.

EXAMPLES

$$A = \begin{bmatrix} 1. & -1. & 1. & 0. & 3. & 1. & 0. & 0. \\ -1. & 0. & -3. & -4. & 5. & 6. & 0. & 0. \\ 2. & 5. & 3. & 0. & 1. & 0. & 0. & 0. \\ 0. & 1. & 0. & 1. & 2. & -1. & 1. & 0. \\ -1. & 0. & 2. & 1. & 1. & 0. & 0. & 1. \end{bmatrix}$$

$$c = \begin{bmatrix} 1. & 2. & 3. & 4. & 5. & 6. & 0. & 0. \end{bmatrix}'$$

$$b = \begin{bmatrix} 1. & 2. & 3. & -1. & 2.5 \end{bmatrix}'$$

$$lbounds = \begin{bmatrix} -1000. & -10000. & 0. & -1000. & -1000. & -1000. & 0. & 0. \end{bmatrix}'$$

$$ubounds = \begin{bmatrix} 10000. & 100. & 1.5 & 100. & 100. & 1000. & 1.D+09 & 1.D+09 \end{bmatrix}'$$

$$BIG = 1.000D+09$$

```
-->[x,obj] = lipsol(A,b,c,lbounds,ubounds,BIG),

 objp  =
  - 7706.4681

 x  =

(    8,    1) sparse matrix
(    1,    1)        275.2766
(    2,    1)      - 129.51064
(    3,    1)        1.158D-08
(    4,    1)      -1000.
(    5,    1)       100.
(    6,    1)      - 703.78723
(    7,    1)        224.7234
(    8,    1)        1177.7766
```

Function `splinpro` allows to handle inequality constraints. Its syntax is similar to the built-in `linpro` function, used for non-sparse systems.

Splinpro function

CALLING SEQUENCE

```
[x,f]=splinpro(p,C,b)

[x,f]=splinpro(p,C,b,ci,cs )

[x,f]=splinpro(p,C,b,ci,cs,mi[,fr])
```

PARAMETERS

p	:	real column vector (objective)
C	:	real matrix (dimension (mi + md) x n)
		If the rank C(1:mi,n) < mi, it must be fr = 1
b	:	RHS vector (dimension (mi + md) x 1)
ci	:	(column) vector of lower-bounds (dimension n)
		If there are no lower bound constraints, put ci = [ ]
		If some components of x are bounded from below,
		set the other (unconstrained) values of
		ci to a very large negative number (e.g. ci(j) = -1/% eps.
cs	:	(column) vector of upper-bounds. (Same remarks as above).
mi	:	number of equality constraints (i.e. C(1:mi,:)*x = b(1:mi))
fr	:	real number, fr = 1 if the rank C(1:mi,n) < mi
x	:	optimal solution found.
f	:	optimal value of the cost function (i.e. f=p'*x).

DESCRIPTION

sparse linear programming solver
`[x,f]=splinpro(p,C,b)`
Minimize $p' * x$, under the constraints,
$C * x \leq b$
`[x,f]=splinpro(p,C,b,ci,cs)`
Minimize $p' * x$, under the constraints
$C * x \leq b \; ci \leq x \leq cs$

`[x,f]=splinpro(p,C,b,ci,cs,mi)`
Minimize $p' * x$, under the constraints
$C(j,:) * x = b(j), j = 1, ..., mi$
$C(j,:) * x \leq b(j), j = mi + 1, ..., mi + md$
$ci \leq x \leq cs$

EXAMPLE

$$C = \begin{bmatrix} 1. & -1. & 1. & 0. & 3. & 1. \\ -1. & 0. & -3. & -4. & 5. & 6. \\ 2. & 5. & 3. & 0. & 1. & 0. \\ 0. & 1. & 0. & 1. & 2. & -1. \\ -1. & 0. & 2. & 1. & 1. & 0. \end{bmatrix}$$

$$p = \begin{bmatrix} 1. & 2. & 3. & 4. & 5. & 6. & 0. & 0. \end{bmatrix}'$$

$$b = \begin{bmatrix} 1. & 2. & 3. & -1. & 2.5 \end{bmatrix}'$$

$$ci = \begin{bmatrix} -1000. & -10000. & 0. & -1000. & -1000. & -1000. \end{bmatrix}'$$

$$cs = \begin{bmatrix} 10000. & 100. & 1.5 & 100. & 100. & 1000. \end{bmatrix}'$$

$$mi = 3.$$

```
-->[x,f]=splinpro(p,C,b,ci,cs,mi);

 f  =
  - 7706.4681

 x  =

(    6,    1) sparse matrix
(    1,    1)        275.2766
(    2,    1)      - 129.51064
(    3,    1)        1.158D-08
(    4,    1)      -1000.
(    5,    1)       100.
(    6,    1)      - 703.78723
```

3.4 Main steps of calculation

Lipsol algorithm performs the following steps:
1.- Preprocess
2.- Calculus
3.- Postprocess

1.- Preprocess
The preprocess step is done using three Scilab functions: `preprocessing`, `scaling` and `checkdense`.

The function "`preprocessing`" makes a number of checks about the input data.

- removing fixed variables
- removing zero rows
- making A "full row rank", the independent linear rows in A are found using, the sparse lu factorization (Scilab functions `lufact` and `luget`).
- removing zero columns
- solving singleton rows shift nonzero lower bounds
- finding upper bound, comparing the upper bound with the BIG (large real number)

The function "`scalin`" of a matrix makes a linear transform of the variables. It is doing following the next steps:

- column scaling
- row scaling

The function makes this to reduces the difference between the maximum element of `abs`(A) and the minimum element of `abs`(A).

Function "`checkdense`" determines and locates dense columns of sparse matrix A.

2.- Calculation The basic lipsol algorithm is performed by the function "`lipalg`".

3.- Postprocess In the last step of lipsol calculation, the `postprocess` function recovers the original variables for the solution

4 Sparse matrix representantion

Here we describe the different storage schemes of the sparse matrices and their graph correlation. We give the structural algorithms of sparse matrices and at last the Lipsol algorithm.

The objective of using sparse matrix techiniques for solving linear systems is to reduce costs by exploiting sparsity. It is possible to achive drastic reductions in storage and arithnetic requirements, when the solutions of dense and tridiagonal systems are compared. There are various kinds of sparse storage schemes, which differ in the way zeros are exploited. The choice of a storage method naturally affects the storage requirement, and the use of ordering strategies (choice of permutation matrix P). Moreover, it has significant impact on the implementation of the factorization and solution, and hence on the complexity of the programs and the execution time.

4.1 SPARSPAK (A Sparse Matrix Package). Symmetric matrix data structure

We establish the relationship between graphs and symmetrix matrices. Let A be an N by N symmetric matrix. The ordered graph of A, denoted by G=(X,E) is the graph for which the N vertices of G are numbered from 1 to N, and $\{x(i), x(j)\}$ is in E if and only if $a_{ij} = a_{ji} \neq 0$, $i \neq j$. Here $x(i)$ denotes the node de X with label i.

Two nodes x and y in G=(X,E) are adjacent if $\{x, y\} \in$ E. For Y$\subset$ X the adjacent set of Y, denoted by Adj(Y), is

Adj(Y) = $\{x \in X - Y : \{x, y\} \in$ E for some $y \in$ Y $\}$

In words, Adj(Y) is simply the set of nodes in G which are not in Y but are adjacent to at least one one in Y.

Computer Representation of Graphs (Adjancy structure): Let G=(X,E) be a graph with N nodes. We denote an adjacency list for x in X is a list containing all the nodes in Adj(x). An adjacency structure for G is simply the set of adjacency lists for all x in X. Such a structure can be implemented in a one dimensional array ADJNCY along with an index array XADJ of length N+1 containing pointers to the beginning of each adjacency list in ADJNCY.

To have an extra entry in XADJ such that XADJ(N+1) points to the next available storage location in ADJNCY.

Furthermore the diagonal is storaged separate.

In other words, the column j of A has XADJ(j+1) - XADJ(J) elements; these elements are the rows ADJNCY(XADJ(j):(XADJ(j+1)-1)).

EXAMPLE

$$A = \begin{bmatrix} 0. & 0. & 1. & 0. & 1. \\ 0. & 0. & 1. & 0. & 0. \\ 1. & 1. & 0. & 1. & 0. \\ 0. & 0. & 1. & 0. & 1. \\ 1. & 0. & 0. & 1. & 0. \end{bmatrix}$$

$$\text{XADJ} = \begin{bmatrix} 1. & 3. & 4. & 7. & 9. & 11. \end{bmatrix}$$

$$\text{ADJNCY} = \begin{bmatrix} 3. & 5. & 3. & 1. & 2. & 4. & 3. & 5. & 1. & 4. \end{bmatrix}$$

See Adjacency structure shown in figure 1

Note: This structure (XADJ, ADJNCY) can be used to save the diagonal and the non-symmetric matrices too.

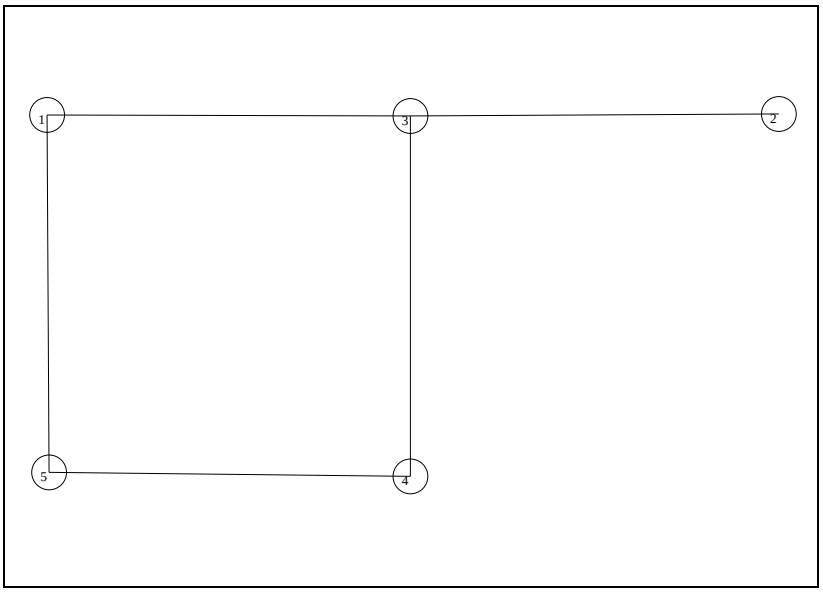

Figure 1: Adjacency structure

Compressed Scheme

The scheme we will use is oriented to the inner-product formulation of the factorization (inferior triangular matrix). If the structure is stored using the uncompressed scheme (XADJ, ADJNCY), the row subcripts (ADJNCY's components) may be repeted of that of the previous column. Naturally, the subscript vector ADJNCY can be compressed so that redundant information is not stored.

In others words, if

ADJNCY(XADJ(j):(XADJ(j+1)-1)=ADJNCY(XADJ(j-1)-1:(XADJ(j)-1)

the sequences ADJNCY(XADJ(j):(XADJ(j+1)-1) is suprimed.

It is done by removing the row subcripts for a column if they appear as a final subsequence of the previous column.

In exchange for the compression, we need to have an auxiliary index vector XLINDX which points to the start of row subcripts in LINDX for each column. See [5].

EXAMPLE

$$A = \begin{bmatrix} 1. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 1. & 1. & 0. & 0. & 0. \\ 0. & 0. & 1. & 1. & 1. & 0. & 0. \\ 0. & 0. & 1. & 1. & 0. & 1. & 0. \\ 0. & 0. & 1. & 1. & 0. & 1. & 1. \end{bmatrix}$$

$$XADJ = \begin{bmatrix} 1. & 2. & 3. & 8. & 12. & 13. & 15. & 16 \end{bmatrix}$$

$$ADJNCY = \begin{bmatrix} 1. & 2. & 3. & 4. & 5. & 6. & 7. & 4. & 5. & 6. & 7. & 5. & 6. & 7. & 7. \end{bmatrix}$$

$$XLINDX = \begin{bmatrix} 1. & 2. & 3. & 8. & 9. & 11. \end{bmatrix}$$

$$LINDX = \begin{bmatrix} 1. & 2. & 3. & 4. & 5. & 6. & 7. & 5 & 6 & 7 \end{bmatrix}$$

4.2 Metanet

Structural sparse matrix representantion by a graph

Let us define the graph associated with a matrix structure. This graph has one vertex per row and one per column, and an edge asociated with each element non zero of the matrix. This edge begins in the corresponding row and finishes in the corresponding column (see Exemple 5.4 of how Metanet is used to calculate this, in fig. 2).

EXAMPLE

$$A = \begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 6. & 0. & 0. & 0 & 0 \\ 3. & 0. & 5. & 0. & 0. & 0. & 0. & 5. & 0 & 0 \\ 0. & 0. & 0. & 3. & 0. & 0. & 0. & 0. & 0 & 0 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 0 \\ 0. & 7. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 0 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 3 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 2 & 0 \end{bmatrix}$$

See the graph G = (V,E) shown in figure 2

The we show the correpondence beteween the rows and the vertices

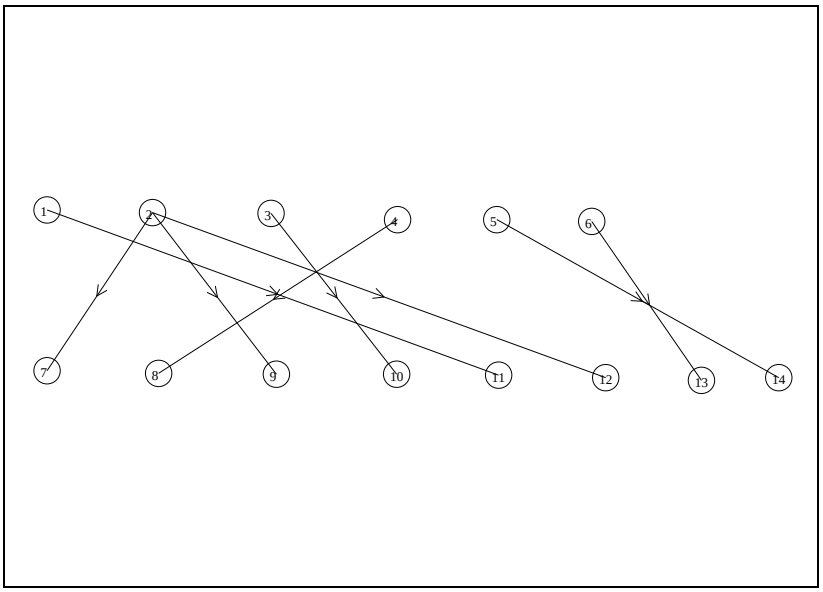

Figure 2: Graph G = (V,E) associated with matrix A

row	1	2	3	4	5	6	7
vertex	1	2	3	-	4	5	6

and the correpondence beteween the columns and the vertices

column	1	2	3	4	5	6	7	8	9	10
vertex	7	8	9	10	-	11	-	12	13	14

4.3 SCILAB sparse matrices

The Scilab database is organized in Fortran arrays; it is composed of 3 arrays, names of the variables (IDSTK), addresses of the starting location (LSTK) and definition of all variables (STK and ISTK occupying the same place in the memory).

Coding the sparse matrix type
ISTK(il) = 5
ISTK(il+1) contains the number of rows nr of the matrix
ISTK(il+2) contains the number of columms nc of the matrix
ISTK(il+3) = 0 if the matrix coefficients are real and = 1 if they are complex numbers.
ISTK(il+4) contains the number of non zero elements nel of the matrix

ISTK(il+5:il+5+nel+nc-1)=inda

If l1=sadr(il+5+nel+nc), then the internal stack STK is a follows:
STK(l1:l1+nel-1)=a

where

inda	:	a matrix control data
	:	For i=1:nr, inda(i) contains the number of non zero elements in row i
	:	For i=1:nel, inda(nr+i) contains the column indices of each non zero element
a	:	column vector of length nel, containing the non-zero elements

EXAMPLE

$$A = \begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 6. & 0. & 0. & 0 & 0 \\ 3. & 0. & 5. & 0. & 0. & 0. & 0. & 5. & 0 & 0 \\ 0. & 0. & 0. & 3. & 0. & 0. & 0. & 0. & 0 & 0 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 0 \\ 0. & 7. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 0 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0 & 3 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 2 & 0 \end{bmatrix}$$

$$\text{ISTK(il:il+5)} = \begin{bmatrix} 5. & nr & nc & nel \end{bmatrix}$$

$$\text{ISTK(il:il+5)} = \begin{bmatrix} 5. & 7. & 10. & 8 \end{bmatrix}$$

inda(1:nr) = ISTK(il+5:il+5+nr)

$$\text{ISTK(il+5:il+5+nr)} = \begin{bmatrix} 1. & 3. & 1. & 0. & 1. & 1. & 1. \end{bmatrix}$$

inda(nr+1:nel) = ISTK(il+5+nr:il+5+nr+nel) =

$$= \begin{bmatrix} 6. & 1. & 3. & 8. & 4. & 2. & 10. & 9. \end{bmatrix}$$

$$\text{a} = \text{STK(l1:l1+nel)} = \begin{bmatrix} 6. & 3. & 5. & 5. & 3. & 7. & 3. & 2. \end{bmatrix}$$

4.4 Conversion functions

splget - converts a sparse matrix Scilab into SPARSPAK representation

CALLING SEQUENCE

```
[xadj,adjncy,anz] = splget(p)
```

PARAMETERS

p	:	real or complex Scilab sparse matrix containing nz non-zero elements, dimension (m x n)
adjncy	:	integer vector of length nz containing the row indices for the corresponding elements in anz
xadj	:	integer vector of length (n+1) which contains column indices information.
anz	:	column vector of length nz, containing the non-zero elements of p

DESCRIPTION

Splget is used to convert the Scilab internal representation of sparse matrices into the representation of SPARSPAK.

xadj : integer vector of length (n+1) which contains column index information. For i=1:n, xadj(i) is the indices in adjncy and anz of the first nonzero entry in the ith column. Finally xadj(n+1)-1 is the indices of the last nonzero entry (nz).

EXAMPLES

A is the same the previous example

```
[xadj,adjncy,anz]=splget(A);
-->xadj'
!  1. 2. 3. 4. 5. 5. 6. 6. 7. 8. 9. !
-->adjncy'
!  2. 5. 2. 3. 1. 2. 7. 6. !
-->anz'
!  3. 7. 5. 3. 6. 5. 2. 3. !
```

```
-->[xadj,adjncy,anz]=splget(A');
-->xadj'
!  1. 2. 5. 6. 6. 7. 8. 9. !
-->adjncy'
!  6. 1. 3. 8. 4. 2. 10. 9. !
-->anz'
!  6. 3. 5. 5. 3. 7. 3.  2. !
```

Note: Relation betwen Scilab sparce matrices ans Sparspak representation. We remark that the vector xadj de A' is the same that inda(nr+1:nel) of A and the vector anz de A' is the same that a the A.

spcompak - converts a compressed representation of sparse matrix into SPARSPAK representation .

CALLING SEQUENCE

```
[adjncy] = spcompak(xadj,xlindx,lindx)
```

PARAMETERS

xadj	:	integer vector of length (n+1) which contains column index information.
xlindx	:	array of length neqns+1, containing pointers into the subscript vector
lindx	:	array of length maxsub, containing the compressed
adjncy	:	integer vector of length nz containing the row indices

DESCRIPTION

Spcompak is used to convert the compressed representation of sparse matrix into the representation of SPARSPAK.

splarse - converts a SPARSPAK representation into a Scilab sparse matrix

CALLING SEQUENCE

```
[p] = splarse(xadj,adjncy,anz)
```

PARAMETERS

See splget

DESCRIPTION

Splarse is used to convert the representation of SPARSPAK into the internal representation of sparse matrices.

4.5 Loadmps - reads a file in MPS format

MPS is an old format, used to describe linear programs. Fields start in column 1, 5, 15, 25, 40 and 50. Sections of an MPS file are marked by so-called header cards, which are distinguished by their starting in column 1. The names chosen for the individual entities (constraints or variables) are not important to the solver.

CALLING SEQUENCE

```
[A,b,c,lbounds,ubounds,BIG,name,ifree] =
 loadmps('file-name' [,mmax,nmax,nnza[,freemax]])
```

PARAMETERS

file-name	:	character string
mmax	:	integer - maximum row numbers of A (default = 300)
nmax	:	integer - maximum columns numbers of A (default = 600)
nnza	:	integer - maximum entries numbers of A (default = 90000)
freemax	:	integer - maximum free variables numbers (default = 50)
A	:	sparse real marix (constraints)
b	:	full column vector (constraints)
c	:	full column vector (objective)
lbounds	:	full column vector of lower bounds
ubounds	:	full column vector of upper bounds
BIG	:	large real number
name	:	character string (problem name)
ifree	:	free variables

DESCRIPTION

[A, b, c, lbounds, ubounds, big, name, ifree] = loadmps('file-name')
Loads the variables saved in file 'file-name'//.mps
These variables: mmax,nmax,nnza,freemax take their values by default
[A, b, c, lbounds, ubounds, big, name, ifree] = loadmps(file-name, mmax, nmax, nnza)

Loads the variables saved in file 'file-name'//.mps

The variable freemax takes its value by default

[A, b, c, lbounds, ubounds, big, name, ifree] = loadmps(file-name, mmax, nmax, nnza, freemax)

Loads the variables saved in file 'file-name'//.mps

EXAMPLE

File in MPS format: test.mps

```
NAME          TESTPROB
ROWS
 N  COST
 L  LIM1
 G  LIM2
 E  MYEQN
COLUMNS
    XONE      COST                 1   LIM1                 1
    XONE      LIM2                 1
    YTWO      COST                 4   LIM1                 1
    YTWO      MYEQN               -1
    ZTHREE    COST                 9   LIM2                 1
    ZTHREE    MYEQN                1
RHS
    RHS1      LIM1                 5   LIM2                10
    RHS1      MYEQN                7
BOUNDS
 UP BND1      XONE                 4
 LO BND1      YTWO                -1
 UP BND1      YTWO                 1
ENDATA
```

Here is the same model written out in an equation-oriented format:

Optimize

COST: XONE + 4 YTWO + 9 ZTHREE

Subject to

LIM1: XONE + YTWO $\leq$ 5

LIM2: XONE + ZTHREE $\geq$ 10
MYEQN: - YTWO + ZTHREE = 7
Bounds
0 $\leq$ XONE $\leq$ 4
-1 $\leq$ YTWO $\leq$ 1
End

```
[A,b,c,lbounds,ubounds,BIG,name,ifree] = loadmps('tes')

-->A
!   1.    0.    1.    1.    0. !
!   0.  - 1.    1.    0.    1. !
!   0.    0.    0.  - 1.    1. !
-->b'
!   5.    10.    7. !
-->c'
!   0.    0.    1.    4.    9. !
-->ubounds'

!   1.000D+32    1.000D+32    0.    0.    1.000D+32 !
-->lbounds'
!   0.    0.    0.  - 1.    0. !
-->BIG
    1.000D+32

-->ifree
     []
```

The main things to know about MPS format are that it is column oriented (as opposed to entering the model as equations), and everything (variables, rows, etc.) gets a name.

There is nothing in MPS format that specifies the direction of optimi zation. And there really is no standard "default" direction; `lipsol` and `splinpro` codes will minimize if you don't specify otherwise.

The NAME card can have anything you want, starting in column 15. The ROWS section defines the names of all the constraints; entries in column 2 or 3 are E for equality rows, L for less-than (*leq*) rows, G for greater-than ($\geq$) rows, and N

for non-constraining rows (the first of which would be interpreted as the objective function). The order of the rows named in this sect ion is unimportant. The largest part of the file is in the COLUMNS section, which is the place where the entries of the A-matrix are put. All entries for a given column must be placed consecutively, although within a column the order of the entries (rows) is irrelevant. Rows not mentioned for a column are implied to have a coefficient of zero. The RHS section allows one or more right-hand-side vectors to be defined; most people don't bother having more than one. In the above example, the name of the RHS vector is RHS1, and has non-zero values in all 3 of the constraint rows of the problem. Rows not mentioned in an RHS vector would be assumed to have a right-hand-side of zero. The optional BOUNDS section lets you put lower and upper bounds on individual variables , instead of having to define extra rows in the matrix. All the bounds that have a given name in column 5 are taken together as a set. Variables not mentioned in a given BOUNDS set are taken to be non-negative (lower bound zero, no upper bound). A bound of type UP means an upper bound is applied to the variable. A bound of type LO means a lower bound is applied. A bound type of FX ("fixed") means that the variable has upper and lower bounds equal to a single value. A bound type of FR ("free") means the variable has neither lower nor upper bounds.

5 Algorithms

5.1 Spchol function

CALLING SEQUENCE

```
[R,P] = spchol(X)
```

PARAMETERS

X : symmetric positive definite real or complex sparse matrix
P : permutation matrix
R : cholesky factorization

DESCRIPTION

Spchol perfoms the sparse cholesky factorization. It is a primitive funtion, coded in Fortran. If X is symmetric positive definite, then `[R,P] = spchol(X)` produces a lower triangular matrix `R` such that `P * R *R' *P' = X`.

EXAMPLE

$$A = \begin{bmatrix} 3. & 0. & 0. & 2. & 0. & 0. & 2. & 0. & 2. & 0. & 0. \\ 0. & 5. & 4. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 4. & 5. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 2. & 0. & 0. & 3. & 0. & 0. & 2. & 0. & 2. & 0. & 0. \\ 0. & 0. & 0. & 0. & 5. & 0. & 0. & 0. & 0. & 0. & 4. \\ 0. & 0. & 0. & 0. & 0. & 4. & 0. & 3. & 0. & 3. & 0. \\ 2. & 0. & 0. & 2. & 0. & 0. & 3. & 0. & 2. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 3. & 0. & 4. & 0. & 3. & 0. \\ 2. & 0. & 0. & 2. & 0. & 0. & 2. & 0. & 3. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 3. & 0. & 3. & 0. & 4. & 0. \\ 0. & 0. & 0. & 0. & 4. & 0. & 0. & 0. & 0. & 0. & 5. \end{bmatrix}$$

```
-->[r,p]=spchol(a)
```

```
  p  =

(   11,   11) sparse matrix

(    1,    9)        1.
(    2,    4)        1.
(    3,    3)        1.
(    4,   10)        1.
(    5,    2)        1.
(    6,    6)        1.
(    7,   11)        1.
(    8,    7)        1.
(    9,    8)        1.
(   10,    5)        1.
(   11,    1)        1.

 r  =

(   11,   11) sparse matrix

(    1,    1)        2.236068
(    2,    1)        1.7888544
(    2,    2)        1.3416408
(    3,    3)        2.236068
(    4,    3)        1.7888544
(    4,    4)        1.3416408
(    5,    5)        2.
(    6,    5)        1.5
(    6,    6)        1.3228757
(    7,    5)        1.5
(    7,    6)        0.5669467
(    7,    7)        1.1952286
(    8,    8)        1.7320508
(    9,    8)        1.1547005
(    9,    9)        1.2909944
(   10,    8)        1.1547005
(   10,    9)        0.5163978
```

```
(   10,   10)        1.183216
(   11,    8)        1.1547005
(   11,    9)        0.5163978
(   11,   10)        0.3380617
(   11,   11)        1.1338934
```

Algorithm outline: Irrespective of what sparse storage scheme is used, there are four distinct phases that can

be identified in the entire computational process.

1. Ordering: Find a good ordering (permutation P) for the given matrix A, with respect to the chosen storage method (ordmmd.f).

2. Storage allocation: Determine the necessary information about the Cholesky factor R of P'AP to set up the storage scheme (symfct.f).

3. Factorization: Factor the permuted matrix P'AP into RR (inpnv,blkfc1)

4. Triangular solution: Solve R'y=P'b and Rz=y. Then set x=Pz (blkslv).

5.2 Scilab Dmperm Function

CALLING SEQUENCE

```
[P,Q,r,s,pp]=dmperm(A)
```

PARAMETERS

A	:	a sparse matrix of scalars
P	:	permutation vector
Q	:	permutation vector
r	:	index vector
s	:	index vector
pp	:	a maximum matching

DESCRIPTION

Dulmage-Mendelsohn decomposition of a matrix.

`[P,Q,r,s,pp]=dmperm(A)` finds permutations P and Q and index vector r and s so that A(p,q) is block upper triangular. The blocks have row and column indices defined by `(r(i):r(i+1)-1 , s(i):s(i+1)-1)`. If A has full column rank then A(pp,:) is square with nonzero diagonal.

EXAMPLE

$$A = \begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. \\ 1. & 0. & 1. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 1. & 0. & 0. & 0. \\ 1. & 0. & 1. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 1. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. \\ 0. & 1. & 0. & 1. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 1. & 0. \\ 0. & 0. & 0. & 0. & 1. & 0. & 1. & 0. & 1. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. & 0. & 1. & 0. & 1. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0. & 0. & 1. \\ 0. & 0. & 0. & 0. & 1. & 0. & 1. & 0. & 1. & 0. & 0. & 0. & 0. & 0. \\ 0. & 1. & 0. & 1. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 1. & 0. \end{bmatrix}$$

[P,Q,r,s,pp]=dmperm(A)

$$P = \begin{bmatrix} 3. & 2. & 12. & 5. & 11. & 9. & 6. & 10. & 8. & 4. & 1. & 7. \end{bmatrix}$$
$$Q = \begin{bmatrix} 6. & 11. & 8. & 3. & 1. & 13. & 12. & 4. & 2. & 9. & 7. & 5. & 14. & 10. \end{bmatrix}$$
$$r = \begin{bmatrix} 1. & 1. & 3. & 5. & 8. & 12. & 13. \end{bmatrix}$$
$$s = \begin{bmatrix} 1. & 2. & 6. & 10. & 13. & 15. & 15. \end{bmatrix}$$
$$pp = \begin{bmatrix} 3. & 12. & 6. & 11. & 9. & 8. & 2. & 5. & 10. \end{bmatrix}$$

$$A(P,Q) = \left[\begin{array}{c|cccc|cccc|ccc|cc}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]$$

A(:,1) corresponds to zero columns

A(1:4,2:9) to the under-constrained (G3), blocks : (1:2,2:5), (3:4,6:9)

A(5:7,10:12) to the well constrained part of the system (G1), it would be divided if correspond

A(8:11,13:14) to the over-constrained part (G2), it would be divided if correspond

A(12,:) row of zeros

Dmperms Function.

CALLING SEQUENCE

```
[P,Q,r,s,pp]=dmperms(A)
```

PARAMETERS

A	:	a sparse matrix of scalars
P	:	permutation vector
Q	:	permutation vector
r	:	index vector
s	:	index vector
pp	:	a maximum matching

DESCRIPTION

Simplified Dulmage-Mendelsohn decomposition of a matrix.

dmperms finds permutations P and Q and index vector r and s so

that A(p,q) is block upper triangular. The blocks have row and columns indices defined by: `(r(i):r(i+1)-1 , s(i):s(i+1)-1)`

EXAMPLE

A is the same the previous example

[p,q,r,s,pp]=dmperms(A)

$$p = \begin{bmatrix} 12. & 5. & 3. & 2. & 11. & 9. & 6. & 10. & 8. & 4. & 1. & 7. \end{bmatrix}$$
$$q = \begin{bmatrix} 6. & 13. & 12. & 11. & 8. & 4. & 3. & 2. & 1. & 9. & 7. & 5. & 14. & 10. \end{bmatrix}$$
$$r = \begin{bmatrix} 1. & 1. & 5. & 8. & 12. & 13 \end{bmatrix}$$
$$s = \begin{bmatrix} 1. & 2. & 10. & 13. & 15. & 15 \end{bmatrix}$$
$$pp = \begin{bmatrix} 3. & 12. & 6. & 11. & 9. & 8. & 2. & 5. & 10. \end{bmatrix}$$

$$A(p,q) = \left[\begin{array}{c|ccccccccc|ccc|cc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]$$

A(:,1) column of zeros

A(1:4,2:9) to the under-constrained (G3), in this case it is not divided

A(5:7,10:12) to the well constrained part of the system (G1), it would be divided if corresponde

A(8:11,13:14) to the over-constrained part (G2), in this case it is not divide

A(12,:) row of zeros

5.3 Sprank Function.

sprank - Sparse matrix structural rank

CALLING SEQUENCE

```
sr = sprank(A)
```

PARAMETERS

A : sparse matrix
sr : is the structural rank of the matrix A
This is also size of a maximun matching in the bipartite graph of A

DESCRIPTION

If A is a matrix of scalars, then the structural rank of the matrix A satifies sprank(A) $\geq$ rank(A). This is also known as maximum transversal, maximum assignment, and size of a maximum matching in the bipartite graph of A. Calls `m6bmatch` of Metanet.

EXAMPLE

A is the same the previous examples
sprank(A)=9; and the rank is 4

5.4 Dulmage and Mendelsohn decomposition of a matrix

Let us consider the natural bipartite graph associated with systems of equations and give some structural properties of this graph, which can be used to simplify the resolution. This bipartite graph has one vertex per equation, one vertex per unknown, and an edge between an unknown x and an equation y iff x appears in equation y.

By convention, equation vertices are elements of y , unknowns are elements of x, and in all figures, equation vertices are drawn above unknown vertices. In the general case, this graph is sufficient enough to decompose the system of equations in three parts: well constrained, over-constrained and under-constrained subsystems. Some of these parts can be empty. This decomposition, which always exists and is unique, is due to Dulmage and Mendelsohn.

Let us now present the fundamental results of Dulmage & Mendelsohn about the decomposition of bipartite graphs, see Dulmage-Mendelsohn. The decomposition algorithm will be described in the next paragraph.

First we need to recall some definitions and to give some notations.

An over-constrained system has more equations than unknowns; its equations are either redundant, or contradictory and thus yield no solution. An under-constrained system has more unknowns than equations and has generally an infinite and not enumerable set of solutions. In a well constrained system, the number of equations is equal to the number of unknowns, the system contains no over-constrained subsystem, and it has a finite number of solutions. The associated graph is said to be well constrained (respectively under-, over-), when its system is well constrained respectively under-, over-).

Let G = (V, E) be a bipartite graph with edges E and with vertices V. Then V = Y ∪ X and Y ∩ X= {} is the set of equations, X is the set of unknowns. A matching M of G is any subset of E such that any two distinct edges in M not have a common vertex. M is a maximum matching iff it is maximal in cardinality. A vertex is saturated, or covered, by M iff it is a vertex of one edge in M. A matching saturating all vertices of G called perfect. We use all along the paper the classical notions of graph theory.

Let us just recall connected components and of strongly connected components. A graph G = (V, E) is said to be connected iff for any pair x and y of vertices, there exists a non directed path from y to x. The connect components of G are the maximal connected subgraphs of G. They partition V and E

A directed graph G = (V, E) is said to be strongly connected iff for any pair x and y of vertices, there exist a directed path from x to y and a directed path from y to x. The strongly connected components of G are the maximal strongly connected subgraphs of partition.

In a directed graph G, a vertex s is said to be a source iff G does not contain any arc vs and s said to be a sink iff G does not contain any arc sv.

Any bipartite graph can be canonically partitioned in three parts. Let G = (V, E) be a bipartite graph associated with a system of equations. Then G can be partitioned intro three bipartite graphs.

G1 Corresponds to the well constrained part of the system.

G2 to the over-constrained part.

G3 to the under-constrained

DM ALGORITHM Let G = (V, E) be a bipartite graph associated with a system of equations. We use the notations n = |V|, m = |E| We now give the algorithms to obtain the previously presented decompositions. Their proofs are direct consequences of the properties described in [4].

The subgraphs G1,G2,G3 of G can be obtained by the following algorithm:

1. Find a maximum matching M of A
2. Build the directed graph GP from G by replacing each edge xy in M by two arcs xy and yx, and by orienting all other edges from Y to X
3. G2 is the set of all descendants of sources of GP
4. Symmetrically, G3 is the set of all ancestors of sinks of GP
5. Finally, G1 = GP - G2 - G3
6. Find the connected components G1, G2, G3

Remark: The function `dmperms` find the connected components in G1, and `dmperm` find the connected components in G1, G2, G3

Decomposition into irreducible parts

a.- Well constrained systems

Suppose G is well constrained, i.e. G=G1, G2=G3={}. The following algorithm gives the unique decomposition of G into its irreducible components.

1. Find a maximum matching M of G (actually, M is a perfect matching).
2. Built the directed graph GP from G replacing each edge xy in M by two arcs xy and yx , and by orienting all other edges from Y to X.
3. Compute the strongly connected components of GP. Each of these strongly connected components is irreducible.

4. To compute the dependencies between these irreducible subgraphs, build the reduced graph R from GP by contracting each strongly connected component in a vertex.

Each arc of R, say from S1 to S2,means: solve subsystem S2 before S1. A compatible total order between subsystems can be obtained by any topological sorting of R.

b.- Well and over-constrained systems

The previous method can be applied to G1 U G2. However, the maximum matching

M is not perfect, and remember that the decomposition of G2 depends on the maximum matching M. The method corresponds to reject the non saturated vertices of G2. Thus we obtain a well constrained system which can be completely solved. At the end, we have to verify that the discarded equations are satisfied by the found solutions.

We use the function `m6bmatch` of Metanet to find a maximum maching in an undirected bipartite graph G = (V ,E).

USING METANET TO CALCULATE DM DECOMPOSITION

Let us consider the 7 x 6 matrix

$$A = \begin{bmatrix} 0. & 0. & 1. & 0. & 1. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. \\ 1. & 0. & 0. & 0. & 0. & 0. \\ 1. & 0. & 0. & 1. & 1. & 1. \\ 0. & 1. & 0. & 0. & 0. & 0. \\ 1. & 0. & 1. & 0. & 1. & 0. \\ 1. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$

```
-->sprank(A)
    5.
```

Let us form the graph associated with matrix A

Y = { 1, 2, 3, 4, 5, 6, 7 } is the set of row (equations)

X = { 8, 9, 10, 11, 12, 13 } is the set of columm (unknowns)

V = Y ∪ X = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 }

Consider the graph G = (V,E) shown in figure 3, with E edges between column j and row i iff a(i,j) $\neq$ 0.

In the beginning the algorithm will take the graph GP. Subgraphs G1, G2, G3 wich can be obtained as follows:

1.- Find a maximun matching M of G

M = { (1,12), (2,9), (4,13), (6,10), (7,8) }

2.- Build the directed graph GP from G by replacing each edge of M by two arcs (i,j) and (j,i). See figure 4.

3.- G2 is the graph of all descendants of sources of GP, { 3, 5, 7 } are the non saturated node of Y ($\leq$ 7). See figure 5.

4.- G3 is the set of all ancestors of sinks of GP, { 11 } are the non saturated node of X (> 7). See figure 6.

5.- Finally, G1 = GP - G2 - G3 (nodes). See figure 7

The connected components of G1, G2, G3 are computed

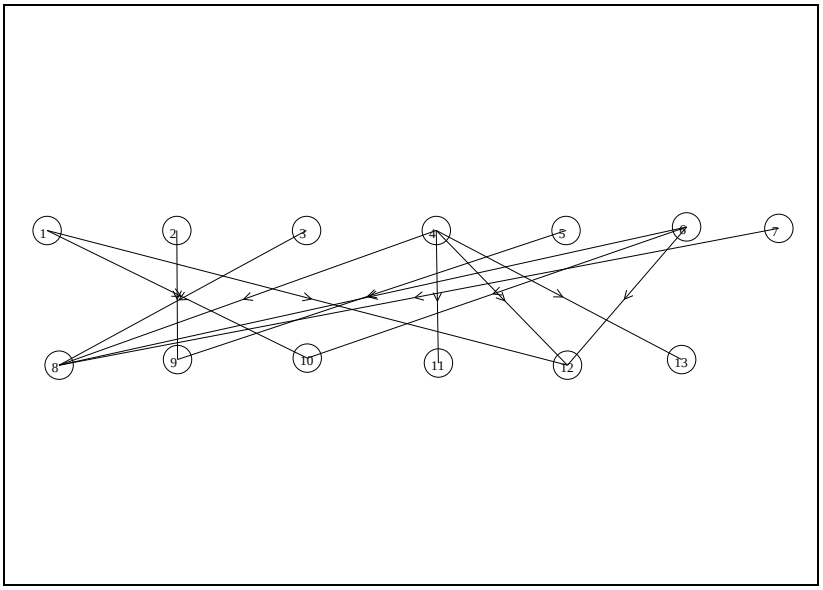

Figure 3: Graph associated with matrix A

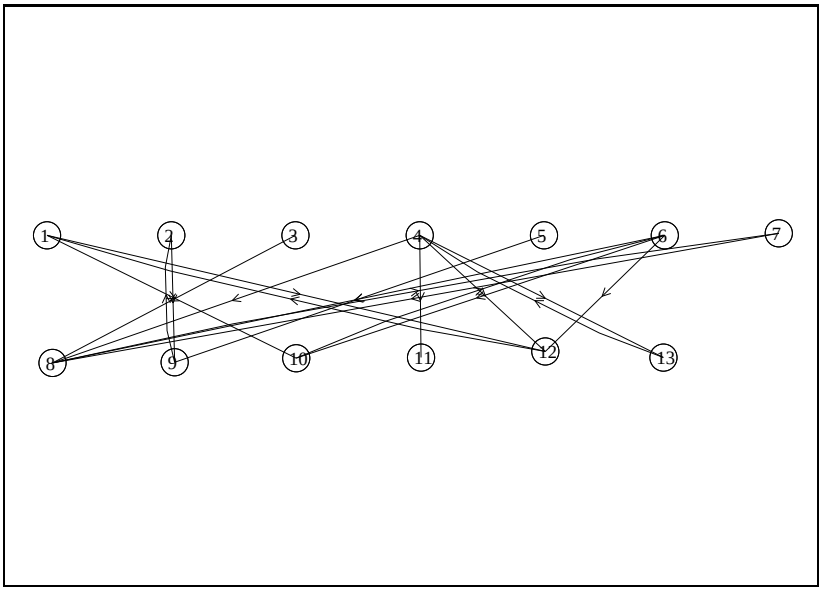

Figure 4: Graph GP

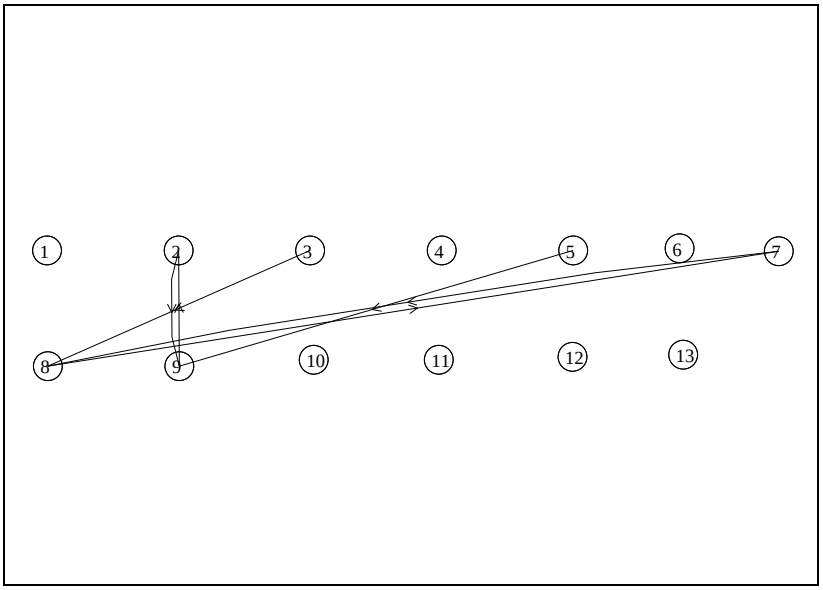

Figure 5: Graph G2

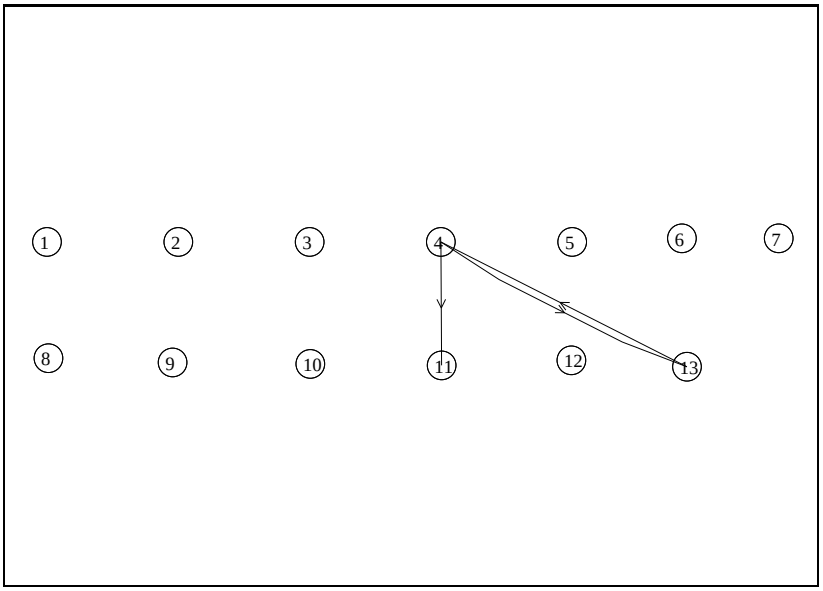

Figure 6: Graph G3

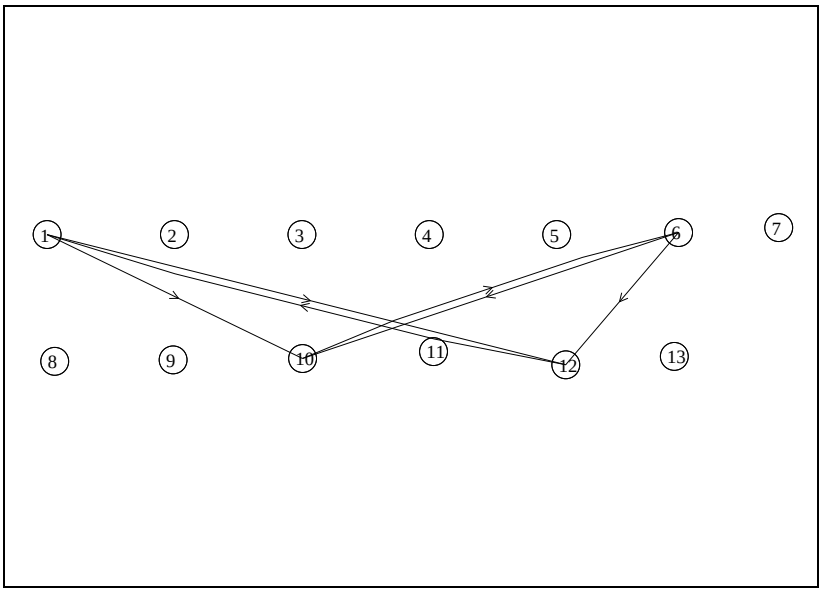

Figure 7: Graph G1

```
[p,q,r,s,pdi]=dmperm(A);

-->full(A(p,q))
 ans  =

!   1.    1.    1.    0.    0.    1. !
!   0.    0.    1.    1.    0.    1. !
!   0.    0.    1.    1.    0.    0. !
!   0.    0.    0.    0.    1.    0. !
!   0.    0.    0.    0.    1.    0. !
!   0.    0.    0.    0.    0.    1. !
!   0.    0.    0.    0.    0.    1. !

-->p
 p  =

!   4.    6.    1.    5.    2.    7.    3. !
```

```
-->q
 q  =

!   6.    4.    5.    3.    2.    1. !

-->r
 r  =

!   1.    2.    4.    6.    8. !

-->s
 s  =

!   1.    3.    5.    6.    7. !

-->pdi
 pdi  =

!   7.    2.    6.    1.    4. !
```

This decomposition does not allow to find a matrix A wuth structural "full rank", because if we take the 5 rows (sprank(A)=5), this submatrix does not have structural "full rank". Therefore, we design this new index permutation where we obtain the 5 first rows with structural "full rank". Furthermore [pdi;qdi] give us the maximun matching, and we can obtain a square matrix a(pdi,qdi) with a structural "full rank".

5.5 Fullsprank Function

CALLING SEQUENCE

```
[P,Q,R,S,PDI,QDI]=fullsprank(A)
```

PARAMETERS

A	:	a matrix of scalars
P	:	permutation vector
Q	:	permutation vector
R	:	index vector
S	:	index vector
PDI	:	a maximum matching
QDI	:	a maximum matching

DESCRIPTION

Dulmage-Mendelsohn decomposition of a matrix, with structural "full rank" `[P,Q,R,S]=fullsprank(A)` finds permutations p and q and index vector r and s so that `A(p(1:sprank(A),q)` is block upper triangular and structural "full rank".

The blocks have indices `(r(i):r(i+1)-1 , s(i):s(i+1)-1)`.

Furthermore [pdi;qdi] give us the maximun matching, and we can obtain a square matrix a(pdi,qdi) with a structural "full rank".

EXAMPLE

```
-->[p,q,r,s,pdi,qdi]=fullsprank(a);

-->a(p,q)
 ans  =

!   1.    1.    1.    0.    0.    1. !
!   0.    0.    1.    1.    0.    1. !
!   0.    0.    1.    1.    0.    0. !
!   0.    0.    0.    0.    1.    0. !
!   0.    0.    0.    0.    0.    1. !
!   0.    0.    0.    0.    1.    0. !
!   0.    0.    0.    0.    0.    1. !
```

```
-->a(p(1:sprank(a)),q)
 ans  =

!   1.    1.    1.    0.    0.    1. !
!   0.    0.    1.    1.    0.    1. !
!   0.    0.    1.    1.    0.    0. !
!   0.    0.    0.    0.    1.    0. !
!   0.    0.    0.    0.    0.    1. !

-->sprank(ans)
 ans  =

    5.

// the matrix is structurally ``full rank''

-->p
 p  =

!   4.    6.    1.    2.    7.    5.    3. !

-->q
 q  =

!   4.    6.    5.    3.    2.    1. !

-->r
 r  =

!   1.    2.    4.    8. !

-->s
 s  =

!   1.    3.    5.    7. !
```

```
-->pdi
 pdi  =

!   7.    2.    6.    1.    4. !

-->qdi
 qdi  =

!   1.    2.    3.    5.    6. !

-->full(a(pdi,qdi))
 ans  =

!   1.    0.    0.    0.    0. !
!   0.    1.    0.    0.    0. !
!   1.    0.    1.    1.    0. !
!   0.    0.    1.    1.    0. !
!   1.    0.    0.    1.    1. !

-->sprank(ans))
 ans  =

    5.
```

Algorithm We obtain the subgraphs G1, G2, G3 of G by the same way that the Dulmage and Mendelsohn decomposition, but we change step 6 in the following way.

Step 6 :

Find the connected components G1.

Construction of the vectors P and Q

P = [P3, P1, P2]

Q = [Q3, Q1, Q2]

P, (Q) corresponding nodes to rows (columns) ordered in decreasing way of each one of the connected components.

Step 7 :

(P1, Q1) are the nodes of G1

P1 = [P11, ..., P1N]

Q1 = [Q11, ..., Q1N]

Step 8 :

(P2, Q2) are the nodes of G2

P2 = [P21, P22]

Q2 = [Q21, Q22]

(P21, Q21) are the nodes of graph G2 in the maximum matching M of G, P21 orderly in decreasing way.

(P22, Q22) are the remained nodes of graph G2 orderly in decresing way.

Step 9 :

(P3, Q3) are the nodes of G3.

P3 are the nodes of graph G3 in the maximum matching M of G, orderly in decresing way.

Q3 = [Q31, Q32]

Q32 are the nodes of graph G3 in the maximum matching M of G corresponding to P3, Q31 are the remained nodes of graph G3 orderly in decresing way.

5.6 Fullsprank vs. fullrank

Here we show two simple examples of constraints $A \times x = b$, where $x \in \Re^n$ is the unknown, A is a matrix in $\Re^{m*n}$, $m < n$, and $b \in \Re^m$, so we see the need of:

1.- Put the good problem (a problem with solution)

2.- Guarantee the full rank structure of the matrix that enter to the calculation program

to obtain the correct solution.

EXAMPLE 1

$$A1 = \begin{bmatrix} 5. & 2. & 0. & 0. \\ 2. & 3. & 0. & 0. \\ 4. & 5. & 0. & 0. \\ 0. & 0. & 6. & 7. \\ 0. & 0. & 8. & 9. \\ 0. & 0. & 1. & 1. \end{bmatrix}$$

$$b1 = \begin{bmatrix} 7. & 5. & 9. & 13. & 17. & 2. \end{bmatrix}$$

sprank(A1) = 4

[p,q,r,s,pdi]=dmperms(A1);

$$p = \begin{bmatrix} 4. & 5. & 1. & 2. & 3. & 6. \end{bmatrix}$$

$$A1(p(1:4),:) = \begin{bmatrix} 0. & 0. & 6. & 7. \\ 0. & 0. & 8. & 9. \\ 5. & 2. & 0. & 0. \\ 2. & 3. & 0. & 0. \end{bmatrix}$$

We can see how the original system is reduced to a 4 x 4 system. The 2 and 6 rows are suppress, so we can obtain the same result independent of the values of b(3) and b(6). Therefore we lost the information of the 2 and 6 rows.

EXAMPLE 2

$$A2 = \begin{bmatrix} 1. & 1. \\ 2. & 2. \\ 5. & 3. \end{bmatrix}$$

sprank(A2) = 2

[p,q,r,s,pdi]=dmperms(A2);

$$p = \begin{bmatrix} 1. & 2. & 3. \end{bmatrix}$$

$$A(p(1:2),:) = \begin{bmatrix} 1. & 1. \\ 2. & 2. \end{bmatrix}$$

We can see how the original system is reduced to a 2 x 2 system. This new system has maximum sprank, but not maximum rank, therefore we lost the information of the 3 row.

To obtain full rank matrices (lineary independent rows), we can calculate the LU factorization with the following Scilab function:

```
function [i]=fullrank(A)
if type(A)<>5 then A=sparse(A);end;
if m<>n then m=maxi([m,n]);A(m,m)=0;end;
[ptr,i]=lufact(A);[P,L,U,Q]=luget(ptr);
x0=lusolve(ptr,b);
ludel(ptr);
[ij,m,v]=spget(P');
i=ij(1:i,2);
```

$$fullrank(A1) = \begin{bmatrix} 1. & 2. & 4. & 6 \end{bmatrix}$$

In this case the result is the same that the preceding. To avoid these problems, we must to eliminate the lineary dependent rows and we must well pose the vector b. If the vector b is right, we will obtain the right solution. But if b is wrong we could have solution even though the system has not. Therefore we have to verify.

$$fullrank(A2) = \begin{bmatrix} 1. & 3. \end{bmatrix}$$

In this case the result is good because no information is lost. The eliminated row is a linear combination of the others and vector b is well posed.

Note: It is necessary to have rows linearly independent to obtain positive matrices to apply the Cholesky factorization.

5.7 Lipsol algorithm: basic steps

In this section, we outline the algorithm used in LIPSOL. For simplicity, we consider the following standard-form of linear program:

$min\{c^T x : Ax = b, x \geq 0\}$,

where $A \in \Re^{m*n}, m < n$.

Its dual linear program is

$max\{b^T y : A^T y + s = c, s \geq 0\}$.

The optimality conditions for this linear program are

$$F(z) = \begin{array}{l} Ax - b \\ A^T y + s - c \\ xs \end{array} = 0, (x, s) \geq 0,$$

where the variable $z = (x, y, s)$ contains both primal and dual variables and xs denotes the element-wise multiplication of vectors x and s.

The algorithm is a primal-dual algorithm, meaning that both the primal and the dual programs are solved simultaneously. It can be considered as a Newton-like method applied to the linear-quadratic system $F(z) = 0$ in the above optimality conditions, while keeping the iterates $\left(x^k, s^k\right)$ positive (thus the interior-point method). The algorithm can be schematically described as follows.

Let us consider just one iteration, suppressing the iteration count k for simplicity, at iterate $z = (x, y, s)$where $(x, s) > 0$. We first compute the so-called prediction direction

$\Delta z_1 = -\left[F(z)'\right]^{-1} F(z)$,

which is nothing but the Newton's direction; then the so-called corrector direction

$\Delta z_2 = -\left[F(z)\right]^{-1} \left(F(z + \Delta z_1) - \mu\, \hat{e}\right)$

where $\mu > 0$ is called the centering parameter which has to be chosen carefully, and $\hat{e}$ is a zero-one vector with ones corresponding to the nonlinear equations xs in $F(z)$. We combine the two directions with a step-length parameter α and update z to obtain the new iterate

$z^+ = z + \alpha\left(\Delta z_1 + \Delta_{Z2}\right)$

where the step-length parameter α is chosen so that $z^+ = (x^+, y^+, s^+)$ satisfies $(x^+, s^+) > 0$. The algorithm the repeats theses steps at the new iterate until convergence.

The above algorithmic scheme is a variant of the predictor-corrector algorithm proposed by Mehrotra. Conceptually, it is a relatively simple algorithm though an actual implementation can be much more complicated, especially for large sparse problems. Mehrotra's approcah uses three directions at each iteration: the predictor, the corrector and the centering directions. The first two directions were motivated by considering a certain trajectory linking the current iterate to a solution. The predictor is the first derivative of the trajectory and the corrector the second.

For more details on Mehrotra's predictor-corrector algorithm, its implementations and properties, see [7], [8] and [10].

6 Appendix

6.1 Steps of lipsol calculation

1.- Preprocess
2.- lipalg
3.- postprocess

6.1.1 Preprocess description

It uses three Scilab functions:
1.1.- preprocess
1.2.- scalin
1.3.- checkdense
following the next tree:
1.1.- preprocess
1.1.1.- tra
1.2.- scalin
1.2.1.- reciprocal
1.2.2.- median
1.3.- checkdense

Preprocess Function Description. - Preprocessing input data

CALLING SEQUENCE

```
[A, b, c, lbounds, ubounds, Ubounds_exist, nub, list1] =
   preprocess (A, b, c, lbounds, ubounds, BIG [,fr])
```

PARAMETERS

A	:	real matrix (constraints)
b	:	column vector (constraints)
c	:	column vector (objective)
lbounds	:	column vector lower bounds
ubounds	:	column vector upper bounds
BIG	:	large number
fr	:	real number If the rank C < m, it must be fr = 1
Ubounds_exist	:	logical variable, %T if there are ubounds
nub	:	number of upper
list1	:	list

list1 = [lbounds, c_orig, Lbounds_non0, Fixed_exist, ifix, infx, xfix, Zrcols_exist, izrcol, inzcol, xzrcol, Sgtons_exist, isolved, insolved, xsolved]

lbounds	:	column vector lower bounds
c_orig	:	original sparse(c)
Lbounds_non0	:	logical variable, %T if there are lbounds
Fixed_exist	:	logical variable, %T if there are fixed variables
ifix	:	fixed variable indices
infx	:	no fixed variable indices
xfix	:	lbounds(ifix)
Zrcols_exist	:	logical variable, %T if there are zero column
izrcol	:	zero column indices
inzcol	:	no zero column indices
xzrcol	:	zero columm indices
Sgtons_exist	:	logical variable, %T if there are solve singleton rows
isolved	:	solve singleton rows indices
insolved	:	no solve singleton rows indices
xsolved	:	solve singleton rows

DESCRIPTION

The preprocessing input data is done through the following steps:
- delete fixed variables
- delete zero rows
- make A "full rank", sparse lu factorization (lufact, luget)
- delete zero columns

- solve singleton rows shift nonzero lower bounds
- find upper bound

Scaling function

CALLING SEQUENCE

```
[A, b, c, ubounds, col_scaled, data_changed, colscl ] =
scalin(A, b, c, ubounds, Ubounds_exist)
```

PARAMETERS

A	:	real matrix (constraints)
b	:	column vector (constraints)
c	:	column vector (objective)
ubounds	:	column vector upper bounds
Ubounds_exist	:	logical variable
col_scaled	:	logical variable, %T if there are changes
data_changed	:	logical variable %T if there are chabges
colscl	:	positive real vector (linear transform coef)

DESCRIPTION

The scaling of a matrix is doing following the next steps:
- vector scaling
- column scaling
- row scaling

Checkedense function

CALLING SEQUENCE

```
[Dense_cols_exist, idense, ispars] = checkdense(A)
```

PARAMETERS

A	:	sparse matrix
Dense_cols_exist	:	logical variable
idense	:	Dense column indices
ispars	:	Sparse column indices

DESCRIPTION

Determine and locate dense columns of sparse matrix A.

Lipalg function

CALLING SEQUENCE

```
[x,resi] = lipalg(A, b, c, ubounds, Ubounds_exist, nub,
Dense_cols_exist, idense, ispars)
```

PARAMETERS

x	:	solution
resi	:	real matrix with residuals
A	:	real matrix (constraints)
b	:	column vector (constraints)
c	:	column vector (objective)
lbounds	:	column vector lower bounds
Ubounds_exist	:	logical variable
nub	:	number of upper bound
Dense_cols_exist	:	logical variable
idense	:	Dense column indices
ispars	:	Sparse column indices

DESCRIPTION

It makes the calculation of optimization

Postprocess function

CALLING SEQUENCE

```
[x, objp] = postprocess( x, list1, col_scaled, colscl)
```

PARAMETERS

x	:	solution (input)
col_scaled	:	logical variable,%T if it is changed in scaling
colscl	:	positive real vector (linear transform coef)
list1	:	list
x	:	solution in the original variables (output)
objp	:	objective

DESCRIPTION

Recovers the original variables for the solution.

6.1.2 Calculus description

Performs the calculation following the next function tree (Scilab standard function is not written). The *.f are the Fortran subroutines.

2.- lipalg
2.1.- symbfct
2.1.1.- splget
2.1.2.- ordmmd
2.1.2.1.- ordmmd.f .
1.3.- symfct
2.1.3.1.- sfinit.f
2.1.3.2.- symfct.f
2.1.3.3.- bfinit.f
2.2.- initpoint
2.2.1.- getpu
2.2.2.- densol
2.2.2.1.- sherman
2.2.2.2.- pcg
2.2.2.2.1.- spnorm
2.2.2.2.2.- linsys

2.2.2.2.2.1.- splget
2.2.2.2.2.2.- inpnv
2.2.2.2.2.2.1.- inpnv.f
2.2.2.2.2.3.- blkfc1
2.2.2.2.2.3.1.- blkfc1.f
2.2.2.2.2.4.- blkslv
2.2.2.2.2.4.1.- blkslv.f
2.2.2.3.- spnorm
2.2.3.- linsys
2.2.3.1.- splget
2.2.3.2.- invpnv
2.2.3.2.1.- inpnv.f
2.2.3.3.- blkfc1
2.2.3.3.1.- blkfc1.f
2.2.3.4.- blkslv
2.2.3.4.1.- blkslv.f
2.3.- complementy
2.4.- feasility
2.4.1.- spones
2.4.2.- spnorm
2.4.3.- isnan
2.5.- errornobj
2.6.- stopping
2.6.1.- detectinf
2.6.2.- maxim
2.7.- getmisc
2.7.1.- reciprocal
2.8.- getpu
2.9.- direction
2.9.1.- densol
2.9.1.1.- sherman
2.9.1.2.- pcg
2.9.1.2.1.- spnorm
2.9.1.2.2.- linsys
2.9.1.2.2.1.- splget
2.9.1.2.2.2.- inpnv
2.9.1.2.2.2.1.- inpnv.f

2.9.1.2.2.3.- blkfc1
2.9.1.2.2.3.1.- blkfc1.f
2.9.1.2.2.4.- blkslv
2.9.1.2.2.4.1.- blkslv.f
2.9.2.- linsys
2.9.2.1.- splget
2.9.2.2.- invpnv
2.9.2.2.1.- inpnv.f
2.9.2.3.- blkfc1
2.9.2.3.1.- blkfc1.f
2.9.2.4.- blkslv
2.9.2.4.1.- blkslv.f
2.9.3.- spones
2.10.- ratiotest
2.10.1.- nozeros
2.11.- centering
2.12.- update
2.12.1.- complementy
2.13.- feasibility
2.13.1.- spones
2.13.2.- spnorm
2.13.3.- isnan

centering	-	Computing centering parameter mu
complementy	-	Evaluate the complementarity vectors and gap
densol	-	Solve linear system with dense columns
detectinf	-	test detention and message
direction	-	Computing search directions
errornobj	-	Calculate the total relative error and objective values
feasibility	-	Evaluate feasibility residual vectors and their norms
getmisc	-	Compute three quantities
getpu	-	Computes Matrices P and U
initpoint	-	Specifying the starting point
pcg	-	preconditioned conjugate gradient method
ratiotest	-	Ratio test
sherman	-	solve (P + U*U')*x = b
stopping	-	Test tolerance
update	-	Update the iterates

6.2 Fortran-Scilab Interfaces

blkfc1	-	subroutine which calls the block general sparse cholesky routine
blkslv	-	block triangular solutions
inpnv	-	input numerical values into sparse data structures put numerical
ordmmd	-	Liu's multiple minimum degree routine
symfct	-	symbolic factorizacion

Function using these interfaces

linsys	-	Solves a positive definite system of linear equations
symbfct	-	Symbolic factorization for sparse Cholesky factorization

For more information, the on line help is available.

6.3 Elementary Functions

descen	-	descendants of nodes of graphs
diff	-	Differences vector or matrix
isnan	-	detect Nan
maxim	-	maximun elements of a matrix for column
median	-	median value
minim	-	minimum elements of a matrix for column
nonzeros	-	is full column vector of the nozeros
rem	-	Remainder after division
repe	-	Extract repeated
repe1	-	Extract repeated leaving once each repeated
reciprocal	-	Inverts the nonzero entries of a matrix elementwise
spnorm	-	sparse vector norms
spones	-	replace nonzero elements wits ones
tra	-	transform a logical matrix sparse or full matrix into a binary matrix

For more information, the on line help is available.

6.4 Fortran Subroutines

Spchol use 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.

6.4.1 Spchol Function

Spchol uses the following fortran subroutines tree:

1.- spcho1
1.1.- ordmmd
1.2.- sfinit
2.- symfct
3.- bfinit
4.- inpnv
5.- spcho2
5.1.- blkfc1

Spho1	-	calculate a-diag and adjust storage
Ordmmd	-	multiple minimum external degree
Sfinit	-	set up for symbolic factorization
Symfct	-	symbolic factorization
Bfinit	-	initialization for block factorization
Inpnv	-	input numerical values into sparse data structures
Spcho2	-	Call to blkfc1 and adjust the storage to structure Scilab
Blkfc1	-	calls the block general sparse cholesky routine
blkslv	-	block triangular solutions

7 Conclutions

The software Lipsol was adapted to the Scilab system. Functions were made for the sparse matrix storage and for the Cholesky factorization computation, and also for structural matrices using Metanet.

References

[1] M. Minoux, G. Bartnik, Graphes algorithmes logiciels, Dunod,1986

[2] R. Sedgewick Algorithms in C, Addison-Wesley Publishing Company, 1990

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to algorithms. McGraw-Hill Book Company, 1990

[4] S. Ait-Aoudia, R. Jegou, D. Michelucci, Reduction of constraint Systems. In Compugraphic, pages 83-92, Alvor, Portugal, 1993.

[5] A. George, Computer Solution of Large Sparse Positive Definite Systems. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1981.

[6] A.L. Dulmage, N.S. Mendelsohn, Two algorithms for Bipartite Graphs, SIAM J., 11,183-194,1963

[7] I.J. Lustig, R.E. Marten, and D.F. Shanno, On implementing Mehrotra's predictor corrector interior point method for linear programming, SIAM J. Optimation 2, 435-449, 1992.

[8] S. Mehrotha, On the implementing of a primal-dual interior point method, SIAM J. Optimization 2. 575-601, 1992.

[9] Y. Zhang, and D. Zhang, On polynomiality of the Mehrotha type predictor corrector interior point algorithms, Mathematical Programming 68, 303-318, 1995.

[10] Y. Zhang, User's Guide to LIPSOL, Department of Mathematics and Statistics. University of Maryland Baltimore County. USA, 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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