University of Campinas
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT OF TELEMATICS
Convex Analysis Laboratory

LMISol (VERSION 1.0)

User's Guide

Maurício C. de Oliveira

Daniela P. de Farias

José C. Geromel

— April 1997 —

Contents

Chapter 1

Tutorial

In this chapter we provide a broad number of examples that show many of the features available in **LMISol**. Most of the commands and functions are not described in detail, which can be found in the reference guide, and are introduced as problems require them. The **LMISol** code of this tutorial is also provided and at the command line interface one can try `exec tutorial`.

1.1 A linear programming problem

Semidefinite programs can be thought as a generalization of linear programs. The optimization code contained in **LMISol** is also LP oriented, that means, it provides specialized routines that are called whenever the interface detects that some linear matrix inequality reduces to a scalar linear inequality. Furthermore it provides a way to explicitly define element-by-element inequalities, rather than matrix inequalities. Let us see how to do that with the help of the following example.

$$
\begin{array}{rl}
\min & x_1 + x_2 + x_3 \\
\text{s.t.} & x_1 + 2x_2 + x_3 \leq 15 \\
 & 3x_1 + 2x_2 + x_3 \geq 10 \\
 & x_i \geq 0, i = 1, 2 \\
 & x_3 \geq 2 \\
 & x_1 + x_2 = 4
\end{array}
$$

In **LMISol**, one can define the LP problem above either defining scalar variables and writing scalar linear inequalities, or with the help of matrix variables. Let us define it in the second form. At first, we define some matrices. The syntax required by **LMISol** to define matrices is almost identical to the `MATLAB` syntax and we will see that many commands are quite similar. So

```
[1]> A = [1 2 1; -3 -2 -1; -eye(3)];
[2]> B = [15 -10 0 0 -2]';
[3]> C = [1 1 0];
[4]> D = 4;
[5]> f = [1 1 1]';
```

The next step is variable declaration, which is carried by the command `var`. We define vector x as a generic 3×1 matrix.

```
[6]> var x(3,1)
```

All done it is time to define the problem, its constraints and the objective function. Linear matrix inequalities are defined assigning to the word `lmi` a linear matrix function. Doing this we are adding to the current problem the linear matrix inequality constraint `lmi` > 0. Every `lmi` must be a linear square or column vector matrix function, fact that will be analized and checked at assignment time. So all matrices and variables referenced in the `lmi` must be already defined with proper dimensions. In **LMISol** the meaning of the relational operator '>' may differ with the dimension of the `lmi`. For a column vector `lmi` it is understood in the element-by-element sense and for a square matrix `lmi` in the symmetric matrix sense. Linear equality constraints are introduced with the help of the word `lme`. As `lmi`s, `lme`s are additively defined assigning a linear matrix function to the `lme` word. All `lme` can hold linear functions of any size and the assignment is understood as the element-by-element assignment operation `lme` $= 0$. The objective function is introduced assigning to the word `obj` a linear scalar function. That function will always be minimized. In our example we may have

```
[7]> lmi = B-A*x
lmi(1) =
 B - A * x
[8]> lme = C*x-D
lme(1) =
 C * x - D
[9]> obj = f'*x
obj =
 f' * x
```

Notice that none of the matrices (`A`, `B`, `C` or `D`) were replaced by their values in the assignments. In fact, those values are only considered for consistency purposes and will be replaced in the actual problem at solution time. Therefore, if we change the values of one of the matrices, for instance `A` or `B`, before solving the problem the constraints will change as well. For this reasons, matrices referenced in the problem should remain in the workspace at least until the solution is obtained.

By this time let us take a look at the matrices, variables and constraints we have in the workspace. This can be done with the help of the command `whos`.

```
[10]> whos
Current workspace:
- Parameters:
      rows     columns  inline  name
         5           3  no      A
         5           1  no      B
         1           3  no      C
         1           1  no      D
         0           0  no      ans
```

```
         1           1  no      display
         1           1  no      eps
         3           1  no      f
         1           1  no      fact
         1           1  no      max_iter
         1           1  no      noprog
         1           1  no      rank_eps
  Total: 12
- Variables:
      rows     columns  type  mask  name
         3           1  gen   no    x
  Total: 1
- Number of LMIs: none.
- Number of LMEs: none.
- Linear objective function: defined.
```

If we want brief information, we should try `who` instead.

```
[11]> who
Current workspace:
- Parameters:
          A        B        C        D      ans  display      eps        f
       fact max_iter   noprog rank_eps
  Total: 12
- Variables:
          x
  Total: 1
- Number of LMIs: none.
- Number of LMEs: none.
- Linear objective function: defined.
```

Finally, we can solve the problem and see the optimal solution:

```
[12]> solve;
Building...
- Evaluating algebraic expressions.
  Evaluating LMIs. Ok.
  Evaluating LMEs. Ok.
  Evaluating objective. Ok.
- Variables. Ok.
- LME basis.
  Processing LMEs. Ok.
  Processing structural LMEs. Ok.
  Processing LME basis. Ok.
- LMI basis.
  Gathering LMIs. Ok.
  Simplifying LMIs. Not implemented yet...
  Processing linear inequalities basis. Ok.
```

```
  No matrix inequalities defined by user.
  No block matrix inequalities defined by user.
- Objective basis.
  Processing objective. Ok.
- Last checks.
  Checking unreferenced variables. Ok.
  Checking symmetry. Ok.
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    0.967038
--------------------------
Optimizing...
---------------------------------------
 Iteration    Objective       Relative
   counter    value           accuracy
---------------------------------------
         1    2.000000e+00    --------
         2    2.000000e+00    0.000000
---------------------------------------
- Timing information.

    Start time: Sun Apr 27 16:23:57 1997
      End time: Sun Apr 27 16:23:57 1997
  Elapsed time: 0 seconds.
[13]> x
x =
     1.9316
     2.0684
     2.0000
```

Looking at the **LMISol** output we notice the problem was solved in three steps. Initially an internal problem representation is built. In this step some consistency checks are made. Then the solver starts looking for a feasible point. At each iteration the feasibility radius is printed (for further information, see the references). Finally we have the optimization. **LMISol** prints the objective value and relative accuracy at each iteration. In our example, the method stopped because the relative accuracy was smaller than the prespecified tolerance.

Also notice that the solution provided is not a polytope vertex, as a consequence of the interior point machinery we have used to solve the problem. This may happen whenever the problem has multiple solutions.

Now, to illustrate other features of **LMISol** we will solve this same problem in a slightly different but equivalent formulation. For instance, the constraint $Ax < B$ could be split in two simpler constraints: $A_1x < B_1$ and $x > B_2$. Let us solve the problem using those constraints. First of all, we define the new parameters

```
[14]> A1 = A(1:2,:), B1 = B(1:2,:)
A1 =
     1.0000     2.0000     1.0000
    -3.0000    -2.0000    -1.0000
B1 =
    15.0000
   -10.0000
[15]> B2 = -B(3:5,:)
B2 =
     0.0000
     0.0000
     2.0000
```

Then we clear the old LMI constraints, all at once, with

```
[16]> clear lmi
```

and define the new ones

```
[17]> lmi = B1-A1*x
lmi(1) =
 B1 - A1 * x
[18]> lmi = x-B2
lmi(2) =
 x - B2
```

To solve we just call `solve` once again.

```
[19]> solve;  x
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    0.967038
--------------------------
Optimizing...
---------------------------------------
 Iteration    Objective       Relative
   counter    value           accuracy
---------------------------------------
         1    2.000000e+00    --------
         2    2.000000e+00    0.000000
---------------------------------------
x =
     1.9316
```

```
    2.0684
    2.0000
```

Now suppose we want to solve the same problem for a different B_2. With **LMISol**, all we have to do is change B_2 and call `solve`!

```
[20]> B2 = [2 -1 4]';
[21]> solve; x
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    1.450517
         2    0.637711
--------------------------
Optimizing...
---------------------------------------
 Iteration    Objective        Relative
   counter    value            accuracy
---------------------------------------
         1    4.000000e+00     --------
         2    4.000000e+00     0.000000
---------------------------------------
x =
     3.1797
     0.8203
     4.0000
```

Any component of a problem can be changed, from parameters and variables to the constraints and the objective function. Problem consistency, however, is checked only at assignment and solution time. Changing a matrix which is referenced inside a `lmi`, for instance, does not force an immediate consistency check of the `lmi`. This will happen only at solution time.

1.2 A quadratic programming problem

The second example is the quadratic programming problem

$$\begin{array}{ll} \min & x_1 \\ \text{s.t.} & x_2 \geq x_1^2 - 1 \\ & x_1 \geq x_2^2 - 0.5 \end{array}$$

To be solved by **LMISol** the constraints of this problem must be rewritten as linear matrix inequalities, which can be done with the help from Schur's complement [1]. Doing this we have the equivalent set of constraints

$$\begin{bmatrix} x_2+1 & x_1 \\ x_1 & 1 \end{bmatrix} > 0$$

$$\begin{bmatrix} x_1+0.5 & x_2 \\ x_2 & 1 \end{bmatrix} > 0$$

To define this problem in **LMISol** we just do

```
[1]> var x1 x2
[2]> lmi = block(x2+1; x1 1)
lmi(1) =
 (1, 1): x2 + 1
 (2, 1): x1
 (2, 2): 1
[3]> lmi = block(x1+0.5; x2 1)
lmi(2) =
 (1, 1): x1 + 0.5
 (2, 1): x2
 (2, 2): 1
[4]> obj = x1
obj =
 x1
```

Notice that we have used the command `block` to define a `lmi` which can be partitioned in terms of linear functions of its variables, and it is only necessary to specify the lower triangular part of the `lmi`. Now we solve the problem

```
[5]> solve;
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    0.399275
--------------------------
Optimizing...
---------------------------------------
 Iteration    Objective       Relative
   counter    value           accuracy
---------------------------------------
         1   -4.979856e-01    --------
         2   -4.992297e-01    0.002498
         3   -4.993504e-01    0.000242
         4   -4.999996e-01    0.001300
         5   -4.999997e-01    0.000000
---------------------------------------
```

and inspect the optimal values of x_1 and x_2

```
[6]> x1
x1 =
    -0.5000
[7]> x2
x2 =
 -5.1140186 x 1e-4
```

1.3 Control design problems

Given the linear system

$$\begin{cases} \dot{x} = Ax + Bw \\ z = Cx \end{cases}$$

where A is stable, x is the state, w is the input and z is the exit, the RH_2 norm of the transfer function from w to z, $H_{zw}(s)$, is

$$\|H_{wz}(s)\|_2^2 = \frac{1}{\pi}\int_0^{\infty} \text{trace}(H_{wz}^*(j\omega)H_{wz}(j\omega))d\omega$$

where $*$ denotes the transpose conjugate operator. One way of calculating this norm, among many others, is through the following semidefinite program [2]

$$\begin{array}{rl} \min & \text{trace}(CPC') \\ \text{s.t.} & AP + PA' + BB' \leq 0 \\ & P = P' > 0 \end{array}$$

The constraints above define a nonempty set if and only if A is a stable matrix. In fact, it is easy to see that if the last problem is feasible for some matrix P then there exists a matrix $Q = Q' > 0$ such that the lyapunov equation

$$AP + PA' + Q = 0$$

is satisfied, thus A is stable. This fact can be explored in several ways in control design and many of the resulting problems reduce to semidefinite programs. Indeed this is our motivation for the development of **LMISol**. In the rest of this tutorial we solve some control design examples that we believe will show how **LMISol** can help the user interested in solving such problems.

1.3.1 RH_2 state feedback control problem

The first problem we solve is the basic one. Given the system

$$\begin{cases} \dot{x} = Ax + B_1w + B_2u \\ z = C_1x + D_{12}u \end{cases}$$

where $D_{12}'D_{12} > 0$, $x \in \Re^n$, $w \in \Re^q$, $z \in \Re^p$ and $u \in \Re^m$, the RH_2 state feedback design problem can be stated as "find a gain K such that the input $u = Kx$ stabilizes the system above and minimizes the RH_2 norm of the transfer function $H_{zw}(s)$". The substitution of this input in the norm calculation problem given before provides

$$\begin{array}{ll} \min & \operatorname{trace}\left[(C_1 + D_{12}K)P(C_1 + D_{12}K)'\right] \\ \text{s.t.} & (A + B_2K)P + P(A + B_2K)' + B_1B_1' \leq 0 \\ & P = P' > 0 \end{array}$$

So, defining the variables $Y = Y' := P$, $L := KP$ and $W = W'$ and using Schur's complement it is possible to rewrite the problem above as the LMI problem

$$\begin{array}{ll} \min & \operatorname{trace}(W) \\ \text{s.t.} & AY + YA' + B_2L + L'B_2' + B_1B_1' \leq 0 \\ & \left[\begin{array}{cc} Y & YC_1' + L'D_{12}' \\ C_1Y + D_{12}L & W \end{array}\right] \geq 0 \end{array}$$

Let us consider a numerical example in **LMISol**. We define the matrices

```
[1]> A = [-2 1 1 1; 3 0 0 2; -1 0 -2 -3; -2 -1 2 -1];
[2]> B1 = [1 0 0; 0 0 0; 1 0 0; 0 0 0];
[3]> B2 = [0 0; 1 0; 0 0; 0 1];
[4]> C1 = [1 0 -1 0; 0 0 0 0; 0 0 0 0];
[5]> D12 = [0 0; 1 0; 0 1];
[6]> D21 = [0 1 0; 0 0 1];
[7]> n = size(A,1);
[8]> m = size(B2,2);
[9]> eps = 1e-4;
```

and the variables

```
[9]> var sym Y(n) L(m,n)
[10]> var sym W(size(C1*C1'))
```

Then we look at the workspace

```
[11]> whos
Current workspace:
- Parameters:
      rows     columns  inline  name
         4           4  no      A
         4           3  no      B1
         4           2  no      B2
         3           4  no      C1
         3           2  no      D12
         2           3  no      D21
         0           0  no      ans
         1           1  no      display
         1           1  no      eps
         1           1  no      fact
         1           1  no      m
```

```
        1          1  no      max_iter
        1          1  no      n
        1          1  no      noprog
        1          1  no      rank_eps
  Total: 15
- Variables:
      rows     columns  type  mask  name
        2          4  gen   no    L
        3          3  sym   no    W
        4          4  sym   no    Y
  Total: 3
- Number of LMIs: none.
- Number of LMEs: none.
- Linear objective function: none.
```

We notice that Y is a symmetric [1] 4×4 matrix, L is a generic 2×4 matrix and W is a symmetric 3×3 matrix. The structural constraints on Y and on W ($Y - Y' = 0, W - W' = 0$) are automatically associated to the variables, thus there is no need to explicitly declare them. We could have declared X and Z as generic matrices and imposed $X = X'$ and $Z = Z'$ as LME constraints (`lme = X-X'; lme = Z-Z'`), however this is less efficient and should be avoided. Structural constraints, typed and masked variables, should be used whenever possible. Another important point is that we have set `eps` to the value 10^{-4}, The scalar `eps` controls the precision of the solution. Now we define the LMI constraints

```
[12]> lmi = block(Y;
              C1*Y+D12*L  W)
lmi(1) =
 (1, 1): Y
 (2, 1): C1 * Y + D12 * L
 (2, 2): W
[13]> lmi = -(A * Y + Y * A' + B2 * L + L' * B2' + B1 * B1')
lmi(2) =
 - ( A * Y + Y * A' + B2 * L + L' * B2' + B1 * B1' )
```

It is important to notice the minus signal in the last LMI constraint. This is necessary since all LMI constraints must be put in the form $\texttt{lmi} > 0$. Finally, we define the objective function

```
[14]> obj = trace(W)
obj =
 trace( W )
```

and we are ready to solve the problem.

```
[15]> solve;
Building...
Searching feasible point...
```

[1]Variables can be of four types : generic (`gen`), symmetric (`sym`), scalar (`scl`) or diagonal (`diag`).

```
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    1.8615
         2    0.9048
--------------------------
Optimizing...
-------------------------------------
 Iteration    Objective     Relative
   counter    value         accuracy
-------------------------------------
         1    2.3679e+00    ------
         2    1.1735e+00    0.5044
         3    9.3062e-01    0.2070
         4    8.0194e-01    0.1383
         5    7.6735e-01    0.0431
         6    7.3963e-01    0.0361
         7    7.2842e-01    0.0152
         8    7.2173e-01    0.0092
         9    7.1848e-01    0.0045
        10    7.1673e-01    0.0024
        11    7.1583e-01    0.0013
        12    7.1537e-01    0.0006
        13    7.1513e-01    0.0003
        14    7.1501e-01    0.0002
        15    7.1496e-01    0.0001
-------------------------------------
```

After all, we look at the optimal values of `Y`, `L` and `W`

```
[16]> Y, L, W
Y =
     0.7695     0.8297     0.8367    -0.6280
     0.8297     1.1039     1.0009    -0.8777
     0.8367     1.0009     1.0265    -0.7964
    -0.6280    -0.8777    -0.7964     0.7166
L =
    -0.6189    -0.7346    -0.7174     0.5717
    -0.2306    -0.2374    -0.2460     0.1756
W =
     0.1227     0.0985     0.0154
     0.0985     0.5219     0.1823
     0.0154     0.1823     0.0703
```

and calculate the controller. We also check if the closed loop system is stable

```
[17]> K = L / Y
```

```
K =
    -0.2829    -0.3924    -0.2320    -0.1887
    -0.2428    -0.1885    -0.0875    -0.2959
[18]> Af = A + B2 * K;
[19]> eig(Af)
ans =
    -1.4546     2.9794
    -1.4546    -2.9794
    -2.5429     0.0000
    -0.2361     0.0000
```

Notice that as **LMISol** is not able to handle complex numbers the output of function `eig` is a two column vector which holds the real and imaginary parts of the eigenvalues in the first and second columns, respectively.

1.3.2 RH_2 decentralized state feedback control problem

Now we want to solve the problem of the last section with an additional decentralization constraint. In our example we would like to have an input vector with the following structure

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = Kx = K \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} K_1 x_1 \\ K_2 x_2 \end{bmatrix}$$

which is achieved with the following controller structure

$$K = \begin{bmatrix} K_1 & 0 \\ 0 & K_2 \end{bmatrix}$$

With our parametrization this can be ensured imposing the following structural constraints to the variables

$$Y = \begin{bmatrix} Y_1 & 0 \\ 0 & Y_2 \end{bmatrix}, \; L = \begin{bmatrix} L_1 & 0 \\ 0 & L_2 \end{bmatrix}$$

This kind of structural constraint can be easily handled by **LMISol**. For the same data of the last section we redeclare `Y` as a nonsymmetric matrix and assign to `Y` and `L` the required `masks`.

```
[20]> clear Y, var Y(n)
[21]> mask(Y) = [ [1 2; 2 3] zeros(2,2); zeros(2,2) [4 5; 5 6] ]
ans =
     1.0000     2.0000     0.0000     0.0000
     2.0000     3.0000     0.0000     0.0000
     0.0000     0.0000     4.0000     5.0000
     0.0000     0.0000     5.0000     6.0000
[22]> mask(L) = [ones(1,2) zeros(1,2); zeros(1,2) ones(1,2)]
ans =
     1.0000     1.0000     0.0000     0.0000
     0.0000     0.0000     1.0000     1.0000
```

The kind of mask assigned to `Y` is called an enumerated mask. It consists of a matrix with the same dimension of `Y` containing the indices of the scalar variables associated to each matrix entry. Repeated indices mean that different matrix entries are identical and null indices denote null scalar entries. Nonnull indices must be contiguous integer numbers starting from 1. The mask of `L` is a boolean mask. A boolean mask should contains only zeros and ones, where each 'one' entry corresponds to a different nonnull scalar variable. We should also notice that although `Y` is no longer declared symmetric its `mask` is symmetric!

Now all we have to do is call `solve`.

```
[23]> solve;
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    2.1480
         2    1.0842
         3    0.4951
--------------------------
Optimizing...
-------------------------------------
 Iteration    Objective     Relative
   counter    value         accuracy
-------------------------------------
         1    5.4252e+00    ------
         2    4.7172e+00    0.1305
         3    4.0450e+00    0.1425
         4    3.9251e+00    0.0297
         5    3.9251e+00    0.0297 *
         6    3.9119e+00    0.0033
         7    3.9119e+00    0.0033 *
         8    3.9115e+00    0.0001
         9    3.9115e+00    0.0001 *
        10    3.9115e+00    0.0000
-------------------------------------
```

Looking at the solution and calculating the controller we get the desired structure.

```
[24]> Y, L
Y =
     0.6613     0.0363     0.0000     0.0000
     0.0363     0.2490     0.0000     0.0000
     0.0000     0.0000     0.7706    -0.2710
     0.0000     0.0000    -0.2710     0.5368
L =
    -1.0755    -0.4829     0.0000     0.0000
```

```
     0.0000     0.0000    -0.0456     0.0138
[25]> K = L / Y
K =
    -1.5324    -1.7164     0.0000     0.0000
     0.0000     0.0000    -0.0610    -0.0051
```

Its is worth remembering that whereas the example of the last section may be handled by Ricatti equation solvers this kind of synthesis proposed here, and in the following sections, can only be achieved through semidefinite programs.

1.3.3 Guaranteed cost RH_2 state feedback control problem

The problem that we are going to solve in this section is another example of control synthesis that can only be achieved with the help of mathematical programming methods. It is an extension of the RH_2 state feedback control problem and is related to the concept of quadratic stability [2]. The idea is that the solution of the problem

$$\begin{array}{ll} \min & \mathrm{trace}(W) \\ \mathrm{s.t.} & A_iY + YA_i' + B_{2i}L + L'B_{2i}' + B_1B_1' \leq 0 \ , \quad i = 1, \dots, N \\ & \left[\begin{array}{cc} Y & YC_1' + L'D_{12}' \\ C_1Y + D_{12}L & W \end{array} \right] \geq 0 \end{array}$$

simultaneously stabilizes all plants described by the pair of matrices (A_i, B_{2i}). Indeed, since we consider the same matrix Y for all plants, a feasible solution to the problem above stabilizes all convex combinations of the N plant vertices (A_i, B_{2i}). At the same time trace(W) is an upperbound for the RH_2 norm of $H_{zw}(s)$ in each plant and its minimization provides to the solution some degree of performance. An appropriate choice of the vertices may lead to design problems of theoretical and practical interest. As an example we have selected the reliable control design problem. In this problem we look for a controller which keeps system stability and performance even in the presence of some actuator failure. We consider the same data as before and clear all current `lmis` and `variables`, keeping all matrices in the workspace. We also define matrix `B2x`, which holds the actuator failure data under consideration, and the `lmi` associated to the cost calculation.

```
[26]> clear lmi var
[27]> var sym Y(n) L(m,n) sym W(size(C1*C1'))
[28]> B21 = [0 0; 1 0; 0 0; 0 1];  // Everything is ok
[29]> B22 = [0 0; 0 0; 0 0; 0 1];  // First actuator failure
[30]> B23 = [0 0; 1 0; 0 0; 0 0];  // Second actuator failure
[31]> B2x = [B21; B22; B23];
[32]> lmi = block(Y;
          C1*Y+D12*L  W);
```

Then we define the inequalities associated to matrices $B_{2i}, i = 1, 2, 3$ with the help of a `for` loop and the submatrix operator. The syntax of the `for` statement is the same implemented in `MATLAB`. **LMISol** consecutively assigns to the `for` variable the entries of the matrix after the assignment. That assignment is column oriented.

```
[33]> for i = 1:3
        lmi = -(sym(A * Y + B2x((i-1)*n+1:i*n,:) * L) + B1*B1');
end
```

In the last declaration the use of a construction like `sym(A*X*B)` means that this term should be considered as `A*X*B + B'*X'*A'`. The use of `sym` is computationally more efficient and helps to improve overall performance. Taking a look at the `lmi` we have defined

```
[34]> lmi
lmi(1) =
 (1, 1): Y
 (2, 1): C1 * Y + D12 * L
 (2, 2): W
lmi(2) =
 - ( sym( A * Y + B2x( ( 1 - 1 ) * n + 1 : 1 * n , : ) * L ) + B1 * B1' )
lmi(3) =
 - ( sym( A * Y + B2x( ( 2 - 1 ) * n + 1 : 2 * n , : ) * L ) + B1 * B1' )
lmi(4) =
 - ( sym( A * Y + B2x( ( 3 - 1 ) * n + 1 : 3 * n , : ) * L ) + B1 * B1' )
```

notice that the actual value of `i`, the `for` variable, and not its name, is in the `lmis`. This is necessary since `i` does not have any meaning outside the `for` loop. To make things clearer we add another inequality as an example

```
[35]> lmi = -(sym(A * Y + B2x((i-1)*n+1:i*n,:) * L) + B1*B1')
lmi(5) =
 - ( sym( A * Y + B2x( ( i - 1 ) * n + 1 : i * n , : ) * L ) + B1 * B1' )
```

In this case `i` is not a `for` variable and is not replaced by its value. That feature can be assigned to any scalar in **LMISol** through the command `inline`. For instance

```
[36]> inline n
[37]> lmi = -(sym(A * Y + B2x((i-1)*n+1:i*n,:) * L) + B1*B1')
lmi(6) =
 - ( sym( A * Y + B2x( ( i - 1 ) * 4 + 1 : i * 4 , : ) * L ) + B1 * B1' )
```

By default, `for` variables are always inlined. We can disable the inline property of any scalar calling `inline` followed by `off`. The syntax is the following

```
[38]> inline off n
```

Finally, we clean the workspace, define the objective and solve the problem.

```
[39]> clear lmi(6) lmi(5)
[40]> obj = trace(W)
obj =
 trace( W )
[41]> solve;
Building...
Searching feasible point...
```

```
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    2.2442
         2    1.8192
         3    0.8834
--------------------------
Optimizing...
-------------------------------------
 Iteration    Objective     Relative
   counter    value         accuracy
-------------------------------------
         1    1.2144e+01    ------
         2    7.9953e+00    0.3416
         3    7.1881e+00    0.1010
         4    6.7187e+00    0.0653
         5    6.6029e+00    0.0172
         6    6.6029e+00    0.0172 *
         7    6.5943e+00    0.0013
         8    6.5904e+00    0.0006
         9    6.5870e+00    0.0005
        10    6.5857e+00    0.0002
        11    6.5854e+00    0.0001
-------------------------------------
```

With the purpose of illustrating the result we look at the solution and check if the system is stable in all proposed situations.

```
[42]> Y, L, W
Y =
     0.8701      0.5638      1.1433     -0.6689
     0.5638      1.3129      0.8016     -1.4266
     1.1433      0.8016      2.9061     -1.9449
    -0.6689     -1.4266     -1.9449      2.5917
L =
    -1.0857     -0.7833     -1.4918      0.7538
    -1.6015     -1.6701     -2.6593      2.1518
W =
     1.4896      0.4061      1.0578
     0.4061      1.4990      2.1454
     1.0578      2.1454      3.5968
[43]> K = L / Y
K =
    -0.6042     -0.7335     -0.5087     -0.6505
    -0.6564     -0.9905     -0.6152     -0.3460
[44]> v = e = [];
```

```
[45]> for i = 1:3
        e = eig(A + B2x((i-1)*n+1:i*n,:) * K);
        v = [v; e(:,1)];
end
[46]> v'
ans =
 Columns 1 to 6
   -1.3524    -1.3524    -0.8580    -2.5167    -1.2648    -1.2648

 Columns 7 to 12
   -2.5615    -0.2549    -1.3750    -1.3750    -0.4573    -2.5262
```

1.3.4 Mixed RH_2/RH_∞ state feedback control problem

As the last state feedback example we present a mixed RH_2/RH_∞ control design. Given the linear system

$$\begin{cases} \dot{x} = Ax + B_1 w + B_2 u \\ z_0 = C_0 x + D_0 u \\ z_1 = C_1 x + D_{12} u \end{cases}$$

the following problem [2]

$$\begin{array}{ll} \min & \operatorname{trace}(W) \\ \text{s.t.} & \begin{bmatrix} AY + YA' + B_2 L + L'B_2' + B_1 B_1' & YC_1' + L'D_{12}' \\ C_1 Y + D_{12} L & -\gamma I \end{bmatrix} \leq 0 \\ & \begin{bmatrix} Y & YC_0' + L'D_0' \\ C_0 Y + D_0 L & W \end{bmatrix} \geq 0 \end{array}$$

provides a suboptimal solution to this very involved problem. The first LMI ensures that feasible solutions have $\|H_{z_0 w}(s)\|_\infty \leq \sqrt{\gamma}$ while we minimize an upperbound for the $\|H_{z_1 w}(s)\|_2^2$. To solve this problem in **LMISol** we follow the same steps as before: define the data, variables, LMIs, objective function and call `solve`.

```
[1]> A = [-2 1 1 1; 3 0 0 2; -1 0 -2 -3; -2 -1 2 -1];
[2]> B1 = [1 0 0; 0 0 0; 1 0 0; 0 0 0];
[3]> B2 = [0 0; 1 0; 0 0; 0 1];
[4]> C0 = [1 1 0 0; 0 0 0 0; 0 0 0 0];
[5]> C1 = [1 0 -1 0; 0 0 0 0; 0 0 0 0];
[6]> D0 = D12 = [0 0; 1 0; 0 1];
[7]> D21 = [0 1 0; 0 0 1];
[8]> gamma = 2;
[9]> n = size(A,1);
[10]> m = size(B2,2);
[11]> var sym Y(n) L(m,n) sym W(size(C0*C0'))
[12]> lmi = block(Y;
          C0*Y+D0*L  W);
```

```
[13]> lmi = block( -(sym(A*Y+B2*L)+B1*B1');
                  -(C1*Y+D12*L)   gamma*eye(size(C1*C1')) )
lmi(2) =
 (1, 1): - ( sym( A * Y + B2 * L ) + B1 * B1' )
 (2, 1): - ( C1 * Y + D12 * L )
 (2, 2): gamma * eye( size( C1 * C1' ) )
[14]> obj = trace(W)
obj =
 trace( W )
[15]> solve;
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    2.0757
         2    1.6616
         3    1.2891
         4    0.9404
--------------------------
Optimizing...
-------------------------------------
 Iteration    Objective     Relative
   counter    value         accuracy
-------------------------------------
         1    4.2920e+00    ------
         2    3.6712e+00    0.1446
         3    3.2933e+00    0.1029
         4    3.1805e+00    0.0342
         5    3.1640e+00    0.0052
         6    3.1567e+00    0.0023
         7    3.1520e+00    0.0015
         8    3.1516e+00    0.0001
         9    3.1516e+00    0.0001 *
        10    3.1516e+00    0.0000
-------------------------------------
```

Notice that it is possible to perform multiple assignments by instruction in **LMISol**.

1.3.5 RH_2 output feedback control problem

In this section we move our attention to a more involved problem. Given the system

$$\begin{cases} \dot{x} = Ax + B_1 w + B_2 u \\ z = C_1 x + D_{12} u \\ y = C_2 x + D_{21} w \end{cases}$$

we want to design a strictly proper output feedback controller in the form

$$\begin{cases} \dot{x_c} = A_c x_c + B_c y \\ u = C_c x_c \end{cases}$$

which stabilizes the system and minimize the RH_2 norm of $H_{zw}(s)$. The solution of this problem is given by the semidefinite program

$$\begin{array}{ll} \min & \operatorname{trace}(W) \\ \text{s.t.} & \begin{bmatrix} Y & I & B_1 \\ I & X & XB_1 + FD_{21} \\ B_1' & B_1'X + D_{21}'F' & W \end{bmatrix} \geq 0 \\ & \begin{bmatrix} AY + YA' + B_2L + L'B_2' & YC_1' + L'D_{12}' \\ C_1Y + D_{12}L & -I \end{bmatrix} \leq 0 \\ & A'X + XA + C_2'F' + FC_2 + C_1'C_1 \leq 0 \end{array}$$

where X, F, Y, L and W are the optimization variables. The controller matrices are recovered from the following equations

$$UV' = I - XY$$

$$Z := A + \begin{bmatrix} Y & L' \end{bmatrix} \begin{bmatrix} A & B_2 \\ C_2 & 0 \end{bmatrix}' \begin{bmatrix} X \\ F' \end{bmatrix}$$

$$M := -Z - (YC_1' + L'D_{12}')C_1$$

$$A_c := U^{-1}M'(V')^{-1}\ ,\quad B_c := U^{-1}F\ ,\quad C_c := L(V')^{-1}$$

Solving this problem in **LMISol** and calculating this controller is as simple as in the other examples. We define the data

```
[1]> A = [-2 1 1 1; 3 0 0 2; -1 0 -2 -3; -2 -1 2 -1];
[2]> B1 = [1 0 0; 0 0 0; 1 0 0; 0 0 0];
[3]> B2 = [0 0; 1 0; 0 0; 0 1];
[4]> C1 = [1 0 -1 0; 0 0 0 0; 0 0 0 0];
[5]> C2 = [1 0 0 0; 0 0 1 0];
[6]> D12 = [0 0; 1 0; 0 1];
[7]> D21 = [0 1 0; 0 0 1];
[8]> n = size(A,1);
[9]> m = size(B2,2);
[10]> p = size(C2,1);
[11]> eps = 1e-4;
[12]> display = 5;
```

variables, constraints and objective

```
[13]> var sym X(n) F(n,p) sym Y(n) L(m,n) sym W(size(B1'*B1))
[14]> lmi = block( Y;
             eye(n) X;
             B1'    B1'*X+D21'*F' W )
lmi(1) =
 (1, 1): Y
 (2, 1): eye( n )
 (2, 2): X
 (3, 1): B1'
 (3, 2): B1' * X + D21' * F'
 (3, 3): W
[15]> lmi = block( -sym(A*Y+B2*L);
             -(C1*Y+D12*L)   eye(size(C1*C1')) )
lmi(2) =
 (1, 1): - sym( A * Y + B2 * L )
 (2, 1): - ( C1 * Y + D12 * L )
 (2, 2): eye( size( C1 * C1' ) )
[16]> lmi = -( sym(X*A+F*C2)+C1'*C1 )
lmi(3) =
 - ( sym( X * A + F * C2 ) + C1' * C1 )
[17]> obj = trace(W);
```

and call `solve`.

```
[18]> solve
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    2.8903
         4    0.5812
--------------------------
Optimizing...
-------------------------------------
 Iteration    Objective     Relative
   counter    value         accuracy
-------------------------------------
         1    4.4930e+00    ------
         6    1.5985e+00    0.0517
        11    1.3777e+00    0.0178
        16    1.3042e+00    0.0078
        21    1.2711e+00    0.0037
        26    1.2562e+00    0.0017
        31    1.2488e+00    0.0009
        36    1.2445e+00    0.0006
```

```
        41    1.2419e+00    0.0004
        46    1.2401e+00    0.0002
        51    1.2388e+00    0.0002
        56    1.2379e+00    0.0001
        61    1.2372e+00    0.0001
        63    1.2369e+00    0.0001
------------------------------------
```

Notice that the output of this problem was done in steps of 5 iterations, fact that is controlled by the scalar `display`, which we have set to 5 in the data initialization. Setting `display = 0` disables output printing.

At last we look at the solution, calculate the controller and verify closed loop stability.

```
[19]> X, F, Y, L, W
X =
     0.4556     0.3309     0.0140     0.2646
     0.3309     0.6270     0.3747     0.3871
     0.0140     0.3747     0.4216     0.2172
     0.2646     0.3871     0.2172     0.4911
F =
    -0.0641    -0.0644
    -0.1849    -0.1873
    -0.1086    -0.1106
    -0.1396    -0.1410
Y =
    12.3196   -12.4708    10.9338    -3.4673
   -12.4708    18.5696   -14.5846     0.7429
    10.9338   -14.5846    15.1294    -2.5012
    -3.4673     0.7429    -2.5012     5.6433
L =
    -6.9138     6.5670    -6.4557     2.7048
    -2.5235     2.7043    -2.3068     0.0609
W =
     0.9053    -0.1726    -0.1750
    -0.1726     0.1634     0.1657
    -0.1750     0.1657     0.1682
[20]> V = Y;
[21]> U = inv(Y) - X;
[22]> Z = A + [Y L']*[A B2; C2 zeros(size(C2*B2))]'*[X; F'];
[23]> M = -Z-(Y*C1'+L'*D12')*C1;
[24]> Cc = L / V'
Cc =
    -0.5035    -0.0936    -0.1327     0.1235
    -0.3056    -0.0939    -0.0532    -0.1882
[25]> Bc = U \ F
Bc =
     0.0007    -0.2555
```

```
     1.1929     1.4374
    -0.0206    -0.1732
    -0.3939    -0.4643
[26]> Ac = U \ M' / V
Ac =
  1e4 *
 -1.2308284  1.6266492  0.5885818  1.7471127
  1.1555920 -1.5272108 -0.5526693 -1.6400938
 -1.1868073  1.5685033  0.5672583  1.6843518
 -0.0425539  0.0558824  0.0205074  0.0600307
[27]> Af = [A B2*Cc; Bc*C2 Ac];
[28]> e = eig(Af);
[29]> e(:,1)'
 ans =
  Columns 1 to 6
   1e4 *
 -2.1303697 -0.0001373 -0.0001373 -0.0001177 -0.0001177 -0.0002800

  Columns 7 to 8
   1e4 *
 -0.0000235 -0.0000669
```

1.3.6 Guaranteed cost RH_2 output feedback control problem

This problem is the output feedback counterpart of the problem considered in section 1.3.3. To this problem, however, till today there are no necessary and sufficient conditions that lead to convex optimization problems. The solution we present here is described in [3] and will make use of the programming features of **LMISol**. The problem we have to solve is

$$\begin{aligned}
&\min\ \operatorname{trace}(W)\\
&\text{s.t.}\ \begin{bmatrix} Y & I & B_1 \\ I & X & XB_1 + FD_{21} \\ B_1' & B_1'X + D_{21}'F' & W \end{bmatrix} \geq 0\\
&\begin{bmatrix} A_iY + YA_i' + B_{2i}L + L'B_{2i}' & * & * \\ Z_i' - M' & A_i'X + XA_i + C_{2i}'F' + FC_{2i} & * \\ C_1Y + D_{12}L & C_1 & -I \end{bmatrix} \leq 0,\\
&i = 1, ..., N
\end{aligned}$$

Where X, F, Y, L, M and W are the optimization variables and Z_i is a nonlinear matrix function in the form

$$Z_i := A_i + \begin{bmatrix} Y & L' \end{bmatrix} \begin{bmatrix} A_i & B_{2i} \\ C_{2i} & 0 \end{bmatrix}' \begin{bmatrix} X \\ F' \end{bmatrix}$$

Given a feasible solution to the last problem the controller matrices are recovered from equations

$$UV' = I - XY$$

$$A_c := U^{-1}M'(V')^{-1}\ ,\quad B_c := U^{-1}F\ ,\quad C_c := L(V')^{-1}$$

What we are going to do is iteratively search a solution for that nonlinear matrix inequality problem decomposing the problem in two semidefinite programs. That procedure is called in [3] the Cross Decomposition Algorithm. Starting from a feasible initial solution Y_0, L_0 we do the following.
while $(f_2 - f_1)/f_2 > \varepsilon$ *do*

1. *Set* $A_{ki} = A_i + B_{2i}L_kY_k^{-1}$, $C_k = C_1 + D_{12}L_kY_k^{-1}$ *and solve the problem*

$$f_1 = \min\ \mathrm{trace}(W)$$
$$\text{s.t.}\ \begin{bmatrix} P & P & PB_1 \\ P & X & XB_1 + FD_{21} \\ B_1'P & B_1'X + D_{21}'F' & W \end{bmatrix} \geq 0$$
$$\begin{bmatrix} A_{ki}'P + PA_{ki} & * & * \\ A_i'P + XA_{ki} + FC_{2i} + S' & A_i'X + XA_i + C_{2i}'F' + FC_{2i} & * \\ C_k & C_1 & -I \end{bmatrix} \leq 0,$$
$$i = 1, ..., N$$

2. *Set* $X_k = X, F_k = F$.

3. *Set* $A_{ki} = A_i + X_k^{-1}F_kC_{2i}$, $B_k = B_1 + X_k^{-1}F_kD_{21}$ *and solve the problem*

$$f_2 = \min\ \mathrm{trace}(W)$$
$$\text{s.t.}\ \begin{bmatrix} Y & Q & B_1 \\ Q & Q & B_k \\ B_1' & B_k' & W \end{bmatrix} \geq 0$$
$$\begin{bmatrix} A_iY + YA_i' + B_{2i}L + L'B_{2i}' & * & * \\ QA_i' + A_{ki}Y + B_{2i}L + S' & A_{ki}Q + QA_{ki}' & * \\ C_1Y + D_{12}L & C_1Q & -I \end{bmatrix} \leq 0,$$
$$i = 1, ..., N$$

4. *Set* $Y_k = Y, L_k = L$.

5. $k = k + 1$.

end

The implementation of that procedure in **LMISol** from the LMI problems is almost immediate . Let us discuss that with a numerical example. We also consider here the reliable control problem, the one we have presented in the guaranteed cost state feedback control section. For the same matrices of the last example we define the following extreme matrices.

```
[30]> B21 = [0 0; 1 0; 0 0; 0 1];  // All actuators are ok
[31]> B22 = [0 0; 0 0; 0 0; 0 1];  // First actuator failure
[32]> B23 = [0 0; 1 0; 0 0; 0 0];  // Second actuator failure
[32]>
```

```
[33]> C21 = [1 0 0 0; 0 0 1 0];    // All sensors are ok
[34]> C22 = [0 0 0 0; 0 0 1 0];    // First sensor failure
[35]> C23 = [1 0 0 0; 0 0 0 0];    // Second sensor failure

[36]> Ax  = [A;   A;   A;   A;   A];
[37]> B2x = [B21; B22; B23; B21; B21];
[38]> C2x = [C21; C21; C21; C22; C23];
[38]> //      ^    ^    ^    ^    ^
[38]> //      |    |    |    |    |
[38]> //     Ok    |    |    |   Sensor 2 fails
[38]> //           |    |   Sensor 1 fails
[38]> //           |   Actuator 2 fails
[38]> //         Actuator 1 fails
```

As commented in the code we have five possible situations: all sensors and actuators are working well, one of the sensors fail but the actuators remain working, one of the actuators fail but the sensors remain working. The controller we are trying to design must be robust enough face to all considered failures. It is worth noticing that the controller design of the last section is not a reliable one and fails to keep closed loop stability in the presence of some of the proposed failures. To illustrate this we calculate the eigenvalues of the closed loop dynamic considering the pair (`B22`, `C21`).

```
[39]> Af = [A B22*Cc; Bc*C21 Ac];
[40]> e = eig(Af);
[41]> e(:,1)'
ans =
 Columns 1 to 6
  1e4 *
 -2.1303697  0.0000159 -0.0002624 -0.0001289 -0.0001353 -0.0001353

 Columns 7 to 8
  1e4 *
 -0.0001171 -0.0001171
```

Now let us solve the problem. In this example we find a feasible initial solution with the help of an auxiliary problem whose solution satisfies a necessary but not sufficient condition for the feasibility of the overall problem. That problem is simply our last example simultaneously applied to all extreme matrices. In this case it does provide a feasible initial solution. We will not discuss the implementation of this problem here since the extension of the last example to all extreme matrices is a mere exercise of the techniques we have been discussing. The source code for this function is also provided with this package and may be easily found along with the other demo files. In **LMISol** we just call this script with the help of the `exec` command, `solve` and set the initial solution.

```
[42]> echo off
[43]> exec ph2init
[44]> echo on
[45]> solve;
```

```
Building...
Searching feasible point...
--------------------------
 Iteration    Feasibility
   counter    radius
--------------------------
         1    5.2496
         6    1.6044
        11    1.3112
        13    0.9492
--------------------------
Optimizing...
-------------------------------------
 Iteration    Objective     Relative
   counter    value         accuracy
-------------------------------------
         1    2.6365e+01    ------
         6    1.6987e+01    0.0409
        11    1.5830e+01    0.0039
        16    1.5695e+01    0.0009
        21    1.5654e+01    0.0003
        26    1.5642e+01    0.0001
-------------------------------------
[46]> Yk = Y; Lk = L;
```

The command `exec` starts the execution of some text file containing **LMISol** commands and will be extensively used in the next steps. With the initial solution, the algorithm itself may be easily implemented. With the help of an infinite `while` loop and an `if` statement all we have to do is

```
[47]> prec = 1e-2;
[48]> while 1
        exec ph2XF
        solve;

        Xk = X; Fk = F;
        f1 = eval(obj)

        exec ph2YL
        solve;

        Yk = Y; Lk = L;
        f2 = eval(obj)

        if ( (f1 - f2) / f2 < prec )
                break;
        end
```

```
end
```

The syntaxes of `while` and `if` are the same implemented in `MATLAB` and should be checked in the reference guide. Notice the use of the `break` clause, which causes the interruption of the infinite `while` loop. Also notice that we have encapsulated the declaration of the auxiliary problems in script files. Script files are just text files containing **LMISol** commands which should be executed by the `exec` command or as a batch file. They can have any desired extension, that should be informed to the `exec`, unless we use the default extension `.lmi`. Sometimes we have to protect the file name inside `" "` to avoid conflict with the parser.

Let us take a more detailed look at the file `ph2XF.lmi`

```
/*

     Guaranteed cost H2 output feedback control problem
     Cross Decomposition algorithm - step XF

*/
clear lmi lme var

var sym X(n) F(n,p) sym P(n) S(n,n) sym W(size(B1'*B1))

N = size(Ax,1) / size(Ax,2);
n = size(Ax,2);
m = size(B2x,2);
p = size(C2x,1) / N;

Gg = Lk / Yk;

Akx = zeros(size(Ax));
for i = 1:N
  Akx((i-1)*n+1:i*n,:) = Ax((i-1)*n+1:i*n,:) + B2x((i-1)*n+1:i*n,:) * Gg;
end
Ck = C1 + D12 * Gg;

lmi = block( P;
             P      X;
             B1'*P  B1'*X+D21'*F' W );

for i = 1:N
  lmi = block( -sym(P*Akx((i-1)*n+1:i*n,:));
               - ( Ax((i-1)*n+1:i*n,:)'*P+...
                   X*Akx((i-1)*n+1:i*n,:)+...
                   F*C2x((i-1)*p+1:i*p,:)+S' ),...
                 -sym(X*Ax((i-1)*n+1:i*n,:)+F*C2x((i-1)*p+1:i*p,:));
               -Ck, -C1, eye(size(C1*C1')) );
end
```

```
obj = trace(W);
```

In this file we just use the usual **LMISol** commands to define the desired problems. All matrices have global scope and the user should be aware of that to avoid having problems with name definitions. Script files may have line comments introduced by the `%` symbol (`MATLAB` syntax) or by the `//` sequence (C++ syntax). Multiline comments should be always enclosed within the usual sequence of symbols `/* */`. Notice that we have used '`...`' in order to split some command lines. This feature is also implemented as in `MATLAB`.

In our example we have obtained the following sequence of objective values before satisfying the stopping criterion.

```
f1 =
    26.0405
f2 =
    22.5009
f1 =
    22.1988
f2 =
    22.1736
```

As can be verified after the exit of the last problem the controller equations we have given before reduce to the following calculations.

```
[101]> Cc = L /  (Q - Y);
[102]> Bc = Xk \ Fk;
[103]> Ac = S' / (Q - Y);
```

So, we check the stability of the closed loop system after applying the calculated controller

```
[104]> v = e = [];
[105]> for i = 1:5
        Afi = [A B2x((i-1)*n+1:i*n,:)*Cc; Bc*C2x((i-1)*p+1:i*p,:) Ac];
        e = eig(Afi);
        v = [v; e(:,1)];
end
[106]> v'
ans =
 Columns 1 to 6
    -1.6340    -1.6340    -1.2031    -1.2031    -2.9677    -0.4502

 Columns 7 to 12
    -0.4502    -2.4840    -1.5525    -1.5525    -1.2339    -1.2339

 Columns 13 to 18
    -1.1838    -2.7267    -2.3804    -0.1627    -1.2863    -1.2863

 Columns 19 to 24
```

```
   -1.2641    -1.2641    -0.1055    -2.6473    -2.0864    -2.0864

 Columns 25 to 30
   -1.7135    -1.7135    -1.1964    -1.1964    -2.7088    -0.2066

 Columns 31 to 36
   -1.2841    -2.0072    -0.0818    -2.6912    -2.6912    -1.6283

 Columns 37 to 40
   -1.2974    -1.2974    -1.1696    -1.1696
```

Chapter 2

Reference Guide

In this chapter we provide detailed information about **LMISol** syntax. We start describing the commands related to the manipulation of objects, i.e. matrices, variables and problem definitions, in **LMISol**. Then we describe the commands related to the workspace and some control operations.

Throughout this reference guide we have described the command syntax using a standard notation. In this notation [*syntax*] denotes that *syntax* is an optional construction and *syntax_a*|*syntax_b* denotes that both *syntax_a* and *syntax_b* are valid. Reserved words are in `typewriter` while parameters are in *italic*. '...' denotes that the construction supports multiple arguments separated by a space.

2.1 Matrices and Variables

2.1.1 `var`

Syntax : `var` [*type*] *variable_name*[*size*] $\cdots$

Description : `var` is the word that declares a variable. All problem variables must be declared, and their size can not be changed after declaration. To redeclare a variable use `clear` first. **LMISol** variable types are

- `scl` : scalar (default type)
- `gen` : generic matrix
- `sym` : symmetric matrix
- `diag` : diagonal matrix

The *size* of a variable must be specified in the form `(`*scalar_a*[,*scalar_b*]`)` or `(`*matrix*`)`, where *matrix* must have dimension 1×2 or 2×1 (its first element is the number of rows and the second the number of columns). If only one scalar argument is given, the object will be considered square with dimension *scalar_a*$\times$*scalar_a*. Otherwise, it will have dimension *scalar_a* $\times$ *scalar_b*. If no *size* is given, the variable is assumed to be a scalar. If no *type* is given, it is assumed to be either a scalar or a generic matrix, depending on its dimension.

Examples :

```
[1]> var scl A gen B C
[2]> var gen D(2,1) gen E(2) F(2,1)
[3]> var sym G(2) diag H(2,2)
[4]> var I([2 5]) J([2 5]') L(size(J))
```

In this example `A`, `B` and `C` are scalars, `D` and `F` are generic 2×1 matrices and `E` is a generic square 2×2 matrix. `G` and `H` are symmetric and diagonal matrices, respectively, both of them square and 2×2. `I`, `J` and `L` are generic 2×5 matrices.

Remarks : It is important to notice that general purpose matrices, called in **LMISol** parameters, do not need to be declared. They begin to exist just after an assignment operation. For instance

```
[1]> A = zeros(2,2)
A =
     0.0000     0.0000
     0.0000     0.0000
```

We can also change the value of A without caring about its dimension

```
[2]> A = 2
A =
     2.0000
```

As noted above, if a variable must have its size changed, first exclude it from the workspace (see `clear`) and use `var` to redeclare it.

2.1.2 mask

Syntax : `mask(`*variable_name*`)` = *matrix*

Description : `mask` enforces variables to have some specific structure, described by the enumerated or boolean mask *matrix*. Enumerated masks have as their entries contiguous integer numbers starting from 1 that refer to scalar optimization variables. Two or more mask entries containing an equal index assign to the corresponding variable entries the same scalar optimization variables. In boolean masks each entry equal to 1 refer to a different scalar optimization variable. In both cases null mask entries mean that the corresponding variable entries are set to zero.

Examples :

```
[1]> var gen A(3,3);
[2]> mask(A) = [1 0 0; 0 1 0; 0 0 1];
```

The off diagonal entries of `A` are set to zero.

```
[1]> var gen A(3,3);
[2]> mask(A) = [1 0 0; 0 2 0; 0 0 2];
```

The off diagonal entries are set to zero and $A_{2,2} = A_{3,3}$.

Remarks : the integers in an enumerated mask matrix must be contiguous, that means `mask(A) = [1 0 0; 0 3 0; 0 0 3]` is not a valid mask. To release a variable from its mask use the command `clear`.

2.1.3 Arithmetical Operators

Description : the **LMISol** arithmetical operators, in precedence order, are the following

`=`	assignment
`+, -`	plus and minus
`*, /, \`	multiplication and division
`^`	scalar power
`+,-`	unary plus and minus

Examples :

```
[1]> L = [1 2; 3 4]
L =
     1.0000     2.0000
     3.0000     4.0000
[2]> M = N = 1
M =
     1.0000
[3]> N
N =
     1.0000
```

Notice that the assignment operator returns the value of the right hand operand which makes possible to perform multiple assignments in the same instruction. Matrices can be added to / subtracted from other matrices or scalars.

```
[14]> D
D =
     1.0000     2.0000
     2.0000     3.0000
[15]> D+2
ans =
     3.0000     4.0000
     4.0000     5.0000
[16]> A
A =
     1.0000     0.0000
     0.0000     1.0000
[17]> D+A
ans =
     2.0000     2.0000
     2.0000     4.0000
```

Matrices can be multiplied / divided by other matrices or by scalars. `X = B / A` gives the minimum norm (in the least squares sense) solution to $XA = B$. `X = A \ B` gives the minimum norm solution to $AX = B$.

```
[18]> D*2
ans =
     2.0000     4.0000
     4.0000     6.0000
[19]> D/2
ans =
     0.5000     1.0000
     1.0000     1.5000
[20]> A/D
ans =
    -3.0000     2.0000
     2.0000    -1.0000
[21]> D\A
ans =
    -3.0000     2.0000
     2.0000    -1.0000
[22]> A = [1 2; 3 5; 9 13]
A =
     1.0000     2.0000
     3.0000     5.0000
     9.0000    13.0000
[23]> B = [5 3 -2]'
B =
     5.0000
     3.0000
    -2.0000
[14]> A\B
ans =
   -10.6129
     7.1774
```

Scalar power can not be applied to matrices.

```
[22]> B = 2;
[23]> B^1.5
ans =
     2.8284

[24]> +D
D =
     1.0000     2.0000
     2.0000     3.0000
[25]> -D
ans =
    -1.0000    -2.0000
    -2.0000    -3.0000
```

2.1.4 Logical Operators

Description : **LMISol** provides the following logical operators, in precedence order

`>=`	greater than or equal to
`>`	greater than
`<=`	lower than or equal to
`<`	lower than
`|`	or
`&`	and
`~=`	different from
`==`	equal to
`~`	not (unary)

The operands must be the same size or one of them must be a scalar.

Examples :

```
[16]> A = [1 2; 0 1];
[17]> B = [3 1; 2 -1];
[18]> A>=B
ans =
     0.0000     1.0000
     0.0000     1.0000
[19]> A>=1
ans =
     1.0000     1.0000
     0.0000     1.0000
[20]> A>B
ans =
     0.0000     1.0000
     0.0000     1.0000
[21]> A>1
ans =
     0.0000     1.0000
     0.0000     0.0000
[22]> A<=B
ans =
     1.0000     0.0000
     1.0000     0.0000
[23]> A<=1
ans =
     1.0000     0.0000
     1.0000     1.0000
[24]> A<B
ans =
     1.0000     0.0000
     1.0000     0.0000
```

```
[25]> A<1
ans =
     0.0000     0.0000
     1.0000     0.0000
[26]> A|B
ans =
     1.0000     1.0000
     1.0000     1.0000
[27]> A|1
ans =
     1.0000     1.0000
     1.0000     1.0000
[28]> A&B
ans =
     1.0000     1.0000
     0.0000     1.0000
[29]> A&1
ans =
     1.0000     1.0000
     0.0000     1.0000
[30]> A==B
ans =
     0.0000     0.0000
     0.0000     0.0000
[31]> A==1
ans =
     1.0000     0.0000
     0.0000     1.0000
[32]> A~=B
ans =
     1.0000     1.0000
     1.0000     1.0000
[33]> A~=1
ans =
     0.0000     1.0000
     1.0000     0.0000
[34]> ~A
ans =
     0.0000     0.0000
     1.0000     0.0000
```

2.1.5 Colon Operator

Syntax : *scalar_a*[: *scalar_b*] : *scalar_c*

Description : the colon operator ':' generates a vector containing numbers from *scalar_a* to *scalar_c* with *scalar_b* increments. The default value for *scalar_b* is 1.

Examples :

```
[20]> A = 1:5
A =
     1.0000     2.0000     3.0000     4.0000     5.0000
[21]> B = 6:-2:1
B =
     6.0000     4.0000     2.0000
[22]> C = 1:0.1:1.5
C =
     1.0000     1.1000     1.2000     1.3000     1.4000     1.5000
[23]> D = 1:-1
D =
     []
```

2.1.6 Submatrix Operator

Syntax : *matrix*(*par*[,*par*])

Description : this is the MATLAB submatrix operator. It selects a part of the matrix *matrix* in an assignment or evaluation.

Examples :

```
[26]> M = [1 2 3; 4 5 6; 7 8 9]
M =
     1.0000     2.0000     3.0000
     4.0000     5.0000     6.0000
     7.0000     8.0000     9.0000
[27]> M(2,1)
ans =
     4.0000
```

We have selected a single entry of M. There are many other features of the submatrix operator. Interesting effects can be obtained combining vector indices and the colon operator

```
[28]> M(2:3,1:2)
ans =
     4.0000     5.0000
     7.0000     8.0000
[29]> M(:,1:2)
ans =
     1.0000     2.0000
     4.0000     5.0000
     7.0000     8.0000
[30]> M([1 3],:)
ans =
     1.0000     2.0000     3.0000
```

```
     7.0000     8.0000     9.0000
[31]> M(1:2:3,[3 2 1])
ans =
     3.0000     2.0000     1.0000
     9.0000     8.0000     7.0000
[32]> M(:,:)
M =
     1.0000     2.0000     3.0000
     4.0000     5.0000     6.0000
     7.0000     8.0000     9.0000
```

The colon operator works as a wildcard for columns, rows or even the entire matrix. In this last case we refer to a matrix as if it were a vector (column oriented).

```
[33]> M(4)
ans =
     2.0000
[34]> M([1 3 7])
ans =
     1.0000     7.0000     3.0000
[35]> M(:)
ans =
     1.0000
     4.0000
     7.0000
     2.0000
     5.0000
     8.0000
     3.0000
     6.0000
     9.0000
```

Submatrix operators can be used on both sides of an assignment. The single column operator applied to a matrix in the left hand side of an assignment leads to interesting effects.

```
[11]> M([1 3 7 4])=[0 0; 0 0]
M =
     0.0000     0.0000     0.0000
     4.0000     5.0000     6.0000
     0.0000     8.0000     9.0000
[12]> A
A =
     1.0000     2.0000
     3.0000     5.0000
     9.0000    13.0000
[13]> B = zeros(2,3);
[14]> B(:) = A
B =
```

```
     1.0000     9.0000     5.0000
     3.0000     2.0000    13.0000
```

In this example, the dimension of the left hand operand was preserved in the assignment even though the left hand operand does not have the same dimension as the right one.

2.1.7 `size`

Syntax :

- *matrix* = `size(`*matrix*`)`
- *scalar* = `size(`*matrix*, `1` | `2 )`

Description : `size` returns the numbers of rows and / or columns of *matrix*.

Examples :

```
[31]> N = [5 3]
N =
     5.0000     3.0000
[32]> size(N)
ans =
     1.0000     2.0000
[33]> size(N,1)
ans =
     1.0000
[34]> size(N,2)
ans =
     2.0000
```

2.1.8 `sqrt`

Syntax : *matrix_b* = `sqrt (`*matrix_a*`)`

Description : the entries of *matrix_b* are the square roots of the corresponding entries in *matrix_a*.

Examples :

```
[35]> D
D =
     1.0000     2.0000
     2.0000     3.0000
[36]> sqrt(D)
ans =
     1.0000     1.4142
     1.4142     1.7321
```

2.1.9 Transpose Operator

Syntax : *matrix_b* = *matrix_a*'

Description : *matrix_b* is the transpose of *matrix_a*.

Examples :

```
[18]> N
N =
     1.0000     5.0000
     4.0000     3.0000
     2.0000     6.0000
[19]> N'
ans =
     1.0000     4.0000     2.0000
     5.0000     3.0000     6.0000
```

2.1.10 `trace`

Syntax : *scalar* = `trace(`*matrix*`)`

Description : *scalar* is the sum of the elements on the diagonal of the square matrix *matrix*.

Examples :

```
[37]> trace(D)
ans =
     4.0000
```

2.1.11 `diag`

Syntax :

- *vector* = `diag(`*matrix*`)`
- *matrix* = `diag(`*vector*`)`

Description : in the first case, *vector* contains the diagonal of the matrix *matrix*. In the second one, *matrix* is a square matrix whose diagonal entries are the elements of *vector* and all other entries hold zeros.

Examples :

```
[5]> A
A =
     1.0000     3.0000
     2.0000     4.0000
[6]> B = diag(A)
B =
     1.0000
     4.0000
```

```
[7]> C = diag(B)
C =
     1.0000     0.0000
     0.0000     4.0000
```

2.1.12 inv

Syntax : *matrix_b* = `inv`(*matrix_a*)

Description : *matrix_b* is the inverse of *matrix_a*.

Examples :

```
[38]> inv(D)
ans =
    -3.0000     2.0000
     2.0000    -1.0000
```

2.1.13 null

Syntax : *matrix_b* = `null`(*matrix_a*)

Description : *matrix_b* holds a unitary norm base of the null space of *matrix_a*.

Examples :

```
[29]> A = [1 2 3; 1 2 3];
[30]> null(A)
ans =
     0.9562     0.1195
    -0.0439    -0.8440
    -0.2895     0.5228
[31]> A*ans
ans =
  1e-16 *
 -2.2204460 -2.2204460
 -2.2204460 -2.2204460
```

2.1.14 eig

Syntax : *matrix_b* = `eig`(*matrix_a*)

Description : `eig` computes the eigenvalues of *matrix_a*. When all eigenvalues are real, *matrix_b* has only one column whose elements are the eigenvalues of *matrix_a*. If any eigenvalue is complex, the first and second columns of *matrix_b* contain the real and imaginary parts of the eigenvalues, respectively.

Examples :

```
[33]> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1.0000     2.0000     3.0000
```

```
     4.0000     5.0000     6.0000
     7.0000     8.0000     9.0000
[34]> eig(A)
ans =
    16.1168
    -1.1168
     0.0000
[35]> B = [0 -2 3; 2 0 -1; 3 4 2]
B =
     0.0000    -2.0000     3.0000
     2.0000     0.0000    -1.0000
     3.0000     4.0000     2.0000
[36]> eig(B)
ans =
     4.2939     0.0000
    -1.1469     2.7449
    -1.1469    -2.7449
```

2.1.15 `eigvec`

Syntax : *matrix_b* = `eigvec`(*matrix_a*)

Description : `eigvec` computes the unitary norm eigenvectors of the square matrix *matrix_a*. Each column of *matrix_b* holds one of the eigenvectors. When there are complex eigenvalues, the columns of *matrix_b* corresponding to the complex conjugate eigenvalue rows returned by `eig` are the real and imaginary parts of the eigenvectors, respectively.

Examples :

```
[36]> A
A =
     1.0000     2.0000     3.0000
     4.0000     5.0000     6.0000
     7.0000     8.0000     9.0000
[37]> eigvec(A)
ans =
     0.2320     0.7858     0.4082
     0.5253     0.0868    -0.8165
     0.8187    -0.6123     0.4082
[38]> B
B =
     0.0000    -2.0000     3.0000
     2.0000     0.0000    -1.0000
     3.0000     4.0000     2.0000
[39]> eig(B)
ans =
     4.2939     0.0000
    -1.1469     2.7449
```

```
    -1.1469    -2.7449
[40]> eigvec(B)
ans =
     0.5516     0.6355     0.0000
     0.0632    -0.2992    -0.4992
     0.8317    -0.4424     0.248
```

In this example, the first column of `ans` is the eigenvector which corresponds to the real eigenvalue 4.2939; the second and third columns are the real and imaginary parts of the eigenvector corresponding to the complex eigenvector $-1.1469 + 2.7449j$. The complex conjugate of this eigenvector is the eigenvector corresponding to $-1.1469 - 2.7449j$.

2.1.16 `chol`

Syntax : *matrix_b* = `chol(`*matrix_a*`)`

Description : `chol` *matrix_b* is the upper triangular matrix resulting from the Cholesky factorization of the positive definite matrix *matrix_a*.

Examples :

```
[40]> A
A =
    15.0000    32.0000    50.0000
    32.0000    78.0000   122.0000
    50.0000   122.0000   195.0000
[41]> chol(A)
ans =
     3.8730     8.2624    12.9099
     0.0000     3.1198     4.9148
     0.0000     0.0000     2.0440
```

2.1.17 `find`

Syntax : *vector* = `find(`*matrix*`)`

Description : *vector* holds the indices of the nonnull elements of *matrix*. The indices are column oriented. Interesting results can be obtained combining `find` and logical operators.

Examples :

```
[59]> A
A =
     1.0000     2.0000
     0.0000     1.0000
[60]> find(A)
ans =
     1.0000     3.0000     4.0000
[61]> B
B =
```

```
     3.0000     1.0000
     0.0000    -1.0000
[62]> find(A==B)
ans =
     2.0000
[63]> find(A>B)
ans =
     3.0000     4.0000
[64]> find(A~=1)
ans =
     3.0000     2.0000
```

2.1.18 `eye`, `ones` and `zeros`

Syntax : *matrix* = `function`(*size*)

Description : functions `eye`, `ones` and `zeros` return, respectively, matrix *matrix* equal to identity, filled with ones or filled with zeros. *size* accepts the same syntax employed for variable declaration (see `var`).

Examples :

```
[39]> zeros(2)
ans =
     0.0000     0.0000
     0.0000     0.0000
[40]> eye(2,3)
ans =
     1.0000     0.0000     0.0000
     0.0000     1.0000     0.0000
[41]> ones(2,3)
ans =
     1.0000     1.0000     1.0000
     1.0000     1.0000     1.0000
[42]> zeros([2 3])
ans =
     0.0000     0.0000     0.0000
     0.0000     0.0000     0.0000
```

2.2 Problem Definition

2.2.1 `lmi`

Syntax : `lmi`[= *linear_expression*] | [(*scalar*)]

Description : this command additively declares an LMI constraint or prints a stored `lmi`. In its first form, it adds to the current problem the constraint *linear_expression* > 0. In the second form it displays the algebraic expression of the *scalar*th LMI constraint. When no argument is given, all stored `lmis` are shown.

Examples :

```
[1]> A = [1 2; 3 4];
[2]> B = [-1 2; 2 3];
[3]> var X(2,2);
[4]> lmi = -(A*X+X*A'+B);
[5]> lmi(1)
lmi(1) =
 - ( A * X + X * A' + B )
```

We have added to the current problem the constraint $AX + XA' + B < 0$.

Remarks : this command is additive. That means, each time an assignment is made a new LMI constraint is added to the problem. For instance, adding $X > 0$ to the current problem

```
[6]> lmi = X
lmi(2) =
 X
```

the printed `lmi` index tells us that the current problem now has two `lmi` constraints. See also `block`.

2.2.2 `lme`

Syntax : `lme`[= *linear_expression*] | [(*scalar*)]

Description : this command additively declares an LME constraint or prints a stored `lme`. In its first form, it adds to the current problem the constraint *linear_expression* $= 0$. In the second form it displays the algebraic expression of the *scalar*th LME constraint. When no argument is given, all stored `lmes` are shown.

Examples :

```
[3]> var X(3,3)
[4]> lme = X-X';
[5]> lme(1)
lme(1) =
 X-X'
```

That is an inefficient way of declaring `X` is a symmetric matrix. In **LMISol** we should not add this constraint explicitly to the problem if we declare `X` of type `sym`. That declaration automatically includes this constraint into the problem. As a matter of fact, every structural constraints on variables, such as those due to the type of a variable or to the use of a mask, are automatically included in the problem. For performance reasons it is better to declare structural constraints than to add them explicitly as LMEs.

Remarks : `lme`, as well as `lmi`, is additive.

2.2.3 obj

Syntax : `obj`[= *linear_expression*]

Description : this command declares and prints the objective function.

Examples :

```
[5] var X(3,3);
[6]> obj = trace(X);
[7]> obj
obj =
 trace( X )
```

Remarks : this command is not additive and each time we define a new objective function, it replaces the current one.

2.2.4 block

Syntax : `lmi` = `block(`A_{11}; A_{21}, A_{22}; ... ; A_{n1}, ..., A_{nn}`)`

Description : this command is used to add `lmi` constraints which may be partitioned as linear functions of its variables. The declaration above is related to a general `block`'ed `lmi`

$$\begin{bmatrix} A_{11} & A'_{21} & \cdots & A'_{n1} \\ A_{21} & A_{22} & & \vdots \\ \vdots & & \ddots & \\ A_{n1} & \cdots & & A_{nn} \end{bmatrix} > 0$$

where A_{ij} is a *linear_expression*

Examples :

```
[13]> A = [1 2; 3 4];
[14]> B = [3 2; 4 5];
[15]> var X(2,2)
[16]> lmi = block(A*X+X'*A'; X*B, A*X*B+B'*X'*A')
lmi(1) =
 (1, 1): A * X + X' * A'
 (2, 1): X * B
 (2, 2): A * X * B + B' * X' * A'
```

This constraint is equivalent to

$$\begin{bmatrix} AX + X'A' & B'X' \\ XB & AXB + B'X'A' \end{bmatrix} > 0$$

2.2.5 sym

Syntax : `sym(`*linear_expression*`)`

Description : this command declares that `sym(`*linear_expression*`)` should be considered as *linear_expression* + *linear_expression*'. Its use improves performance.

Examples :

```
[19]> lmi = sym(A*X)+B
lmi(1) =
 sym( A * X ) + B
```

That is equivalent to

```
[20]> lmi = A*X+X*A'+B
lmi(2) =
 A * X + X * A' + B
```

Remarks : `sym` can be used even inside a `block` command.

2.2.6 build

Syntax : *scalar* = `build`

Description : this command starts building the internal representation for the current semidefinite program. It makes checks to ensure the symmetry of the `lmis`, the correct dimensions and the absence of inconsistency in the `lmes`. It returns a *scalar* that can be used as a flag and whose possible values are

- `0` : the internal representation was successfully built.
- `1` : there were problems during the construction of the internal representation.

Remarks : `solve` and `feas` automatically call `build` whenever necessary.

2.2.7 feas

Syntax : *scalar* = `feas`

Description : this command starts searching a feasible solution to the current semidefinite program. It returns a *scalar* that can be used as a flag and whose possible values are

- `0` : a feasible point has been found.
- `1` : maximum number of iterations reached. Problem could be unfeasible or more iterations are required.
- `2` : maximum number of iterations without changes in the feasibility radius reached.
- `3` : numerical degeneracy. Perhaps the problem is ill conditioned.
- `4` : numerical unfeasibility detected. The algorithm can not go further.

Remarks : See remarks in `solve`.

2.2.8 `solve`

Syntax : *scalar* = `solve`

Description : this command starts searching the optimal solution to the current semidefinite program. It returns a *scalar* that can be used as a flag and whose possible values are

- `0` : optimization successfully finished.
- `1` : maximum number of iterations reached. Solution may not be optimal.
- `2` : maximum number of iterations without changes in the objective function reached.
- `3` : numerical degeneracy. Perhaps the problem is ill conditioned.

Remarks : there are some default scalars that control the behavior of the solver. They are

- `max_iter` : maximum number of iterations. If the number of iterations reaches this limit the algorithm stops, even if it has not found a feasible or optimal solution yet. Its default value is 100.
- `eps` : relative precision of the solver. If the relative precision of the solution becomes lower than `eps` the algorithm stops. Its default value is 10^{-6}.
- `rank_eps` : internal numerical precision. This value is used in the linear algebra operations that handle the `lme` constraints and in other rank dependent operations. Its default value is 10^{-10}.
- `display` : the number of iterations the solver must skip before displaying more progress information. Setting `display = 0` means that no information is printed at all. Its default value is 1.
- `noprog` : maximum number of iterations the objective function can remain unchanged without interrupting the solver. Its default value is 10.
- `fact` : factorization method. If `fact` holds 1 the QR algorithm is selected and if `fact` holds 2 the Cholesky factorization is selected. Its default value is 1.

2.2.9 `eval`

Syntax : `eval(lmi(`*scalar*`)` | `lme(`*scalar*`)` | `obj)`

Description : using current matrices and variables `eval` evaluates the required `lme`, `lmi` or `obj` and return its value in *matrix*.

Examples :

```
[19]> A = [1 2; 3 4];
[20]> B = [3 1; 2 1];
[21]> C = [1 0; 2 1];
[22]> var sym X(2,2);
```

```
[23]> X = [1 2; 2 1];
[24]> lmi = sym(A*X);
[25]> lme = B*X-C;
[26]> obj = trace(X);
[27]> eval(lmi(1))
ans =
    10.0000    15.0000
    15.0000    20.0000
[28]> eval(lme(1))
ans =
     4.0000     7.0000
     2.0000     4.0000
[29]> eval(obj)
ans =
     2.0000
```

2.3 Workspace Control

2.3.1 who

Syntax : `who`

Description : `who` displays informations about the defined matrices, variables, the number of `lmi`s and `lme`s currently defined and reports the existence of an `obj` function.

Examples :

```
[1]> A = [1 2; -1 3]; B = [2; 1]; C = [1 0];
[2]> var X(2,2);
[3]> lmi = A*X+X*A'+B*B';
[4]> lme = X-X';
[5]> obj = trace(C*X*C');
[6]> who
Current workspace:
- Parameters:
          A        B        C      ans  display      eps     fact max_iter
     noprog rank_eps
  Total: 10
- Variables:
          X
  Total: 1
- Number of LMIs: 1
- Number of LMEs: 1
- Linear objective function: defined.
```

Besides the matrices we have defined, `who` reports the existence of other six matrices which are described in `solve`. Matrix `ans` has the same meaning as in `MATLAB`. Whenever the value returned by any **LMISol** operation is not assigned to any specific matrix it is stored in `ans`.

2.3.2 `whos`

Syntax : `whos`

Description : the same as who, with more detailed information.

Examples :

```
[1]> A = [1 2; -1 3]; B = [2; 1]; C = [1 0];
[2]> var X(2,2);
[3]> lmi = A*X+X*A'+B*B';
[4]> lme = X-X';
[5]> obj = trace(C*X*C');
[6]> whos
Current workspace:
- Parameters:
      rows     columns  inline  name
         2           2  no      A
         2           1  no      B
         1           2  no      C
         0           0  no      ans
         1           1  no      display
         1           1  no      eps
         1           1  no      fact
         1           1  no      max_iter
         1           1  no      noprog
         1           1  no      rank_eps
  Total: 10
- Variables:
      rows     columns  type  mask  name
         2           2  gen   no    X
  Total: 1
- Number of LMIs: 1
- Number of LMEs: 1
- Linear objective function: defined.
```

2.3.3 `clear`

Syntax : `clear` [*parameter_name* | *variable_name* | `lmi`[(*scalar*)] | `lme`[(*scalar*)] | `obj` | `var` | `mask(`*variable*`)`] $\cdots$

Description : `clear` excludes **LMISol** objects from the workspace. If no parameter is given, all objects are excluded.

Examples :

```
[11]> A = [1 2; -1 3]; B = [2; 1]; C = [1 0];
[12]> var X(2,2);
[13]> lmi = A*X+X*A'+B*B';
[14]> obj = trace(C*X*C');
```

```
[15]> clear A
[16]> who
Current workspace:
- Parameters:
          B        C      ans  display      eps     fact max_iter   noprog
   rank_eps
  Total: 9
- Variables:
          X
  Total: 1
- Number of LMIs: 1
- Number of LMEs: none.
- Linear objective function: defined.
[17]> clear X
[18]> who
Current workspace:
- Parameters:
          B        C      ans  display      eps     fact max_iter   noprog
   rank_eps
  Total: 9
- Variables:
    There are no user defined variables.
- Number of LMIs: 1
- Number of LMEs: none.
- Linear objective function: defined.
[19]> clear lmi(1)
[20]> who
Current workspace:
- Parameters:
          B        C      ans  display      eps     fact max_iter   noprog
   rank_eps
  Total: 9
- Variables:
    There are no user defined variables.
- Number of LMIs: none.
- Number of LMEs: none.
- Linear objective function: defined.
[21]> clear obj
[22]> who
Current workspace:
- Parameters:
          B        C      ans  display      eps     fact max_iter   noprog
   rank_eps
  Total: 9
- Variables:
    There are no user defined variables.
```

```
- Number of LMIs: none.
- Number of LMEs: none.
- Linear objective function: none.
[23]> clear
[24]> who
Current workspace:
- Parameters:
        ans  display      eps     fact max_iter   noprog rank_eps
  Total: 7
- Variables:
    There are no user defined variables.
- Number of LMIs: none.
- Number of LMEs: none.
- Linear objective function: none.
```

2.3.4 import

Syntax : `import` *filename* [*obj* $\cdots$]

Description : `import` reads object *obj* from the binary file *filename* into the current workspace. *filename* is a file in the `MATLAB` format. When no parameter is given, all matrices stored in *filename* are brought to the workspace. The default extension for `import` files is `.mat`.

Examples :

```
[1]> import "dados.mat" A B C
```

Remarks : Notice that the file name `dados.mat` is protected by quotations. This is because it has a dot among its characters. Whenever *filename* contains dots or slashes, it should be written in quotations. The given example is equivalent to

```
[2]> import dados A B C
```

2.3.5 export

Syntax : `export` *filename* [*obj* $\cdots$]

Description : `export` writes objects *obj* from the workspace into the binary file *filename*. *filename* is a file in the `MATLAB` format. When no parameter is given, all parameters in the workspace are written into *filename*. The default extension for `export` files is `.mat`.

Examples :

```
[5]> export "dados.mat" A B C
```

Remarks : see the remarks in `import`.

2.3.6 format

Syntax : format compact | [verbose]

Description : `format` sets the current output format. The default format is `verbose`.

Examples :

```
[19]> A = 1

A =

     1.0000

[20]> format compact
[21]> A = 1
A =
     1.0000
[22]> format
[23]> A = 1

A =

     1.0000

[24]> format compact
[25]> A = 1
A =
     1.0000
[26]> format verbose
[27]> A = 1

A =

     1.0000
```

2.3.7 echo

Syntax : echo [on] | off

Description : `echo` turns the echoing of the input lines on or off. It is useful when running batch and script files.

Examples :

```
[3]> A = 1
A =
     1.0000
[4]> echo
[5]> A = 1
```

```
A = 1
A =
     1.0000
[6]> echo off
echo off
[7]> A = 1
A =
     1.0000
[8]> echo
[9]> A = 1
A = 1
A =
     1.0000
```

2.3.8 `end, quit`

Syntax :

- `end`
- `quit`

Description : finish the current **LMISol** session.

2.3.9 `inline`

Syntax : `inline` [`on`] | `off` *scalar*

Description : `inline` turns a scalar inline on or off. When a scalar is inlined its value, and not its name, is considered in *linear_expression* assignments. Every `for` variable is by default inlined inside its `for` loop.

Examples :

```
[8]> n = 3;
[9]> var x(3,1)
[10]> inline n
[11]> lmi = x-n
lmi(1) =
 x - 3
[12]> inline off n
[13]> lmi = x-n
lmi(2) =
 x - n
```

2.3.10 `exec`

Syntax : `exec` *filename*

Description : `exec` runs the script file *filename*. The default extension for script files is `.lmi`.

2.3.11 `home`

Syntax : `home`

Description : `home` clears the screen.

2.3.12 `dir`

Syntax : `dir`

Description : `dir` shows current directory entries.

2.3.13 `ls`

Syntax : `ls`

Description : `ls` shows current directory entries.

2.3.14 `pwd`

Syntax : `pwd`

Description : `pwd` shows the name of the current directory.

2.3.15 `cd`

Syntax : `cd` *directory name*

Description : `cd` changes the current directory to *directory name*.

2.4 Flow Control

2.4.1 `for`

Syntax :

`for` *variable_name* = *matrix*
  [*statement*
   ⋮]
`end`

Description : *statement* is repeated as many times as the number of elements of *matrix*. At each iteration, one of the entries of *matrix* is assigned to the scalar *variable_name*. The assignment is column oriented.

Examples :

```
[8]> A = [1 2 3; 4 5 6];
[9]> for i=A
 A(i)
end
ans =
     1.0000
```

```
ans =
     5.0000
ans =
     4.0000
ans =
     3.0000
ans =
     2.0000
ans =
     6.0000
[10]> j = [];
[11]> for i=1:3
j = [j i^3];
end
[12]> j
j =
     1.0000     8.0000    27.0000
[13]> for i = 1:2
a = 5
for j = [0.5 -0.33]
a = i*j
end
a = 10
end
a =
     5.0000
a =
     0.5000
a =
    -0.3300
a =
    10.0000
a =
     5.0000
a =
     1.0000
a =
    -0.6600
a =
    10.0000
```

Remarks : `for` commands can be nested. All `for` variables are inlined (see also `inline`).

2.4.2 `while`

Syntax :

```
while condition
   [statement
     ⋮ ]
end
```

Description : the standard conditional `while` loop. While not all entries of the resultant matrix of *condition* are equal to zero, *statement* is run.

Examples :

```
[2]> a = 3;
[3]> while a>1
a = a-1
end
a =
     2.0000
a =
     1.0000
[4]> b = 3;
[5]> while a<3
while b>1
b = b-1
end
a = a+1
b = a
end
b =
     2.0000
b =
     1.0000
a =
     2.0000
b =
     2.0000
b =
     1.0000
a =
     3.0000
b =
     3.0000
[6]> a = [1 2 3];
[7]> i = 0;
[8]> while a>i
i = i+1
end
i =
     1.0000
```

```
i =
     2.0000
i =
     3.0000
[9]> a = [1 1 1];
[10]> i = 1;
[11]> while a
a(i) = 0;
i = i+1
end
a =
     0.0000     1.0000     1.0000
i =
     2.0000
a =
     0.0000     0.0000     1.0000
i =
     3.0000
a =
     0.0000     0.0000     0.0000
i =
     4.0000
```

2.4.3 `if`

Syntax :

`if` *condition*
  [*statement_a*
   ⋮]
[`else`
  [*statement_b*]
   ⋮]
`end`

Description : if at least one entry of matrix *condition* is different from zero *statement_a* is run, otherwise *statement_b* is run.

Examples :

```
[13]> a = 1; b = 0;
[14]> d = 0;
[15]> if a
d = 1
d =
     1.0000
end
[16]> if b
```

```
d = 2
else d = 3
d =
     3.0000
end
[17]> if a
if b
d = 4
else
d = 5
d =
     5.0000
end
else
d = 6
end
```

2.4.4 break

Syntax : `break`

Description : `break` exits the innermost loop.

Examples :

```
[3]> i = 0;
[4]> while i<4
for j=1:2
if i>2 break
end
j
end
i = i+1
end
j =
     1.0000
j =
     2.0000
i =
     1.0000
j =
     1.0000
j =
     2.0000
i =
     2.0000
j =
     1.0000
```

```
j =
     2.0000
i =
     3.0000
i =
     4.0000
```

2.4.5 pause

Syntax : `pause`

Description : `pause` stops the execution until the user strikes any key.

Examples :

```
[12]> for i=1:4
j = i+1
pause
end
j =
     2.0000
Press any key to continue...
j =
     3.0000
Press any key to continue...
j =
     4.0000
Press any key to continue...
j =
     5.0000
Press any key to continue...
```

Bibliography

[1] S. P. Boyd, L. El Ghaoui, E. Feron and V. Balakrishnan, *Linear Matrix Inequalities in System and Control Theory*, SIAM, Philadelphia, PA, 1994.

[2] P. Colaneri, J. C. Geromel and A. Locatelli. *Control Theory and Design : an $RH_2 - RH_\infty$ viewpoint.* to be published by Academic Press, 1997.

[3] J. C. Geromel, J. M. Bernussou and M. C. de Oliveira, "H_2 Norm Optimization with Constrained Dynamic Output Feedback Controllers : decentralized and reliable control", 1997, submitted.

[4] A. Nemirovskii, P. Gahinet, "The Projective Method for Solvig Linear Matrix Inequalities", *Proceedings of the American Control Conference*, Maryland, june. 1994.

[5] M. C. de Oliveira, "Algoritmos de Pontos Interiores e Desigualdades Matriciais Lineares", M.Sc. Thesis, FEEC-UNICAMP, 1996 (in portuguese).

[6] M. C. de Oliveira and J. C. Geromel "Interior-point Methods and LMI : a benchmark", 1996, submitted.