Chapter 1

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.

1.1 Matrices and Variables

1.1.1 `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.

1.1.2 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`.

1.1.3 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
```

1.1.4 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
```

1.1.5 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 =
     []
```

1.1.6 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.

1.1.7 `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
```

1.1.8 `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
```

1.1.9 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
```

1.1.10 `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
```

1.1.11 `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
```

1.1.12 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
```

1.1.13 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
```

1.1.14 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
```

1.1.15 `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 $-1.1469 - 2.7449j$.

1.1.16 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
```

1.1.17 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
```

1.1.18 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
```

1.2 Problem Definition

1.2.1 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`.

1.2.2 `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.

1.2.3 `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.

1.2.4 `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

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

1.2.5 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.

1.2.6 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.

1.2.7 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`.

1.2.8 `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.

1.2.9 `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
```

1.3 Workspace Control

1.3.1 who

Syntax : who

Description : who displays informations about the defined matrices, variables, the number of lmis and lmes 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.

1.3.2 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.
```

1.3.3 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.
```

1.3.4 import

Syntax : `import` *filename* [*obj* ···]

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
```

1.3.5 export

Syntax : `export` *filename* [*obj* ···]

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`.

1.3.6 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
```

1.3.7 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
```

1.3.8 `end, quit`

Syntax :

- `end`
- `quit`

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

1.3.9 `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
```

1.3.10 `exec`

Syntax : `exec` *filename*

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

1.3.11 `home`

Syntax : `home`

Description : `home` clears the screen.

1.3.12 `dir`

Syntax : `dir`

Description : `dir` shows current directory entries.

1.3.13 `ls`

Syntax : `ls`

Description : `ls` shows current directory entries.

1.3.14 `pwd`

Syntax : `pwd`

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

1.3.15 `cd`

Syntax : `cd` *directory name*

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

1.4 Flow Control

1.4.1 `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`).

1.4.2 `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
```

1.4.3 `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
```

1.4.4 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
```

1.4.5 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.