Manual of "The Robust LMI Parser" – Version 3.0

Cristiano M. Agulhari*, Alexandre Felipe†, Ricardo C. L. F. Oliveira†, and Pedro L. D. Peres†

March 21, 2019

Abstract

The ROLMIP (Robust LMI Parser), built to work under MATLAB jointly with YALMIP, is a toolbox aimed to ease the programming of Linear Matrix Inequality (LMI) relaxations for parameter-dependent LMIs with parameters lying in unit simplexes (or inside known intervals). Through simple commands, the user is able to define matrix polynomials, as well as to described the desired parameter-dependent LMI conditions in a easy way, considerably reducing the programming time. The main commands and definitions are thoroughly explained in this manual.

1 Introduction

In the last decades, problems formulated in terms of Linear Matrix Inequality (LMI) conditions and solved by Semidefinite Programming (SPD) techniques became more and more common in several fields related to engineering and applied mathematics. Specifically in control theory, the growing usage of such relevant tools has led to important results on the analysis of systems stability, synthesis of robust controllers and filters for uncertain systems and synthesis of optimal control models, just to name a few problems [4].

Accompanying the growth of the usage of LMI optimization, a large number of solvers based on interior point methods were developed, as well as interfaces for parsing the LMIs, most of them free and easily accessible. Thanks to such remarkable advance in the computational tools to define, manipulate and solve LMIs, in many cases one can say that if a problem can be cast as a set of LMIs, then it can be considered as solved [4]. Unfortunately, this is not completely true for large scale systems, since LMI solvers are limited to a few thousands of variables and LMI rows, but progresses are being made.

Usually, the approach to solve LMIs is decomposed in two steps: first, an interface for parsing the conditions is used, for example the YALMIP [11] or the LMI Control Toolbox from MATLAB [7]; then, an LMI solver is invoked to find a solution (if any), for example SeDuMi [22], SDPT3 [23] or MOSEK [1]. Some auxiliary toolboxes may also be used in addition to the parser and solver, for example the SOSTOOLS [17], which is used to transform a sum of squares problem into an SDP formulation, GloptiPoly [9], used to handle optimization problems over polynomials, and the R-RoMuLOC [16, 5, 15, 24], a toolbox for the manipulation and resolution of conditions related to robust multi-objective control.

To motivate the use of ROLMIP consider, for instance, the problem of analyzing the stability of a discrete-time linear system given by

$$x(k+1) = Ax(k), \tag{1}$$

with $x(k) \in \mathbb{R}^n$ being the state vector of the system and $A \in \mathbb{R}^{n\times n}$ being the dynamic matrix. Following the Lyapunov stability theory, such system is stable if and only if there exists a symmetric matrix $P \in \mathbb{R}^{n\times n}$ such that the LMI

$$\begin{bmatrix} P & A'P \\ PA & P \end{bmatrix} > 0 \tag{2}$$

holds. Such LMI can be easily programmed and solved using, respectively, any LMI parser and solver available.

Consider now that system (1) is affected by uncertainties, i.e., the system matrix is parameter-dependent and is given by $A(\alpha)$. The robust stability analysis of such system can be performed by rewriting the LMI

*Federal University of Technology of Paraná – UTFPR, Campus Cornélio Procópio, PR. E-mail: agulhari@utfpr.edu.br

†School of Electrical and Computer Engineering, University of Campinas – UNICAMP, 13083-852, Campinas, SP, Brazil. E-mails: o.alexandre.felipe@gmail.com, {ricfow,peres}@dt.fee.unicamp.br

condition (2) as

$$\begin{bmatrix} P(\alpha) & A(\alpha)'P(\alpha) \\ P(\alpha)A(\alpha) & P(\alpha) \end{bmatrix} > 0 \tag{3}$$

but, in this case, (3) must hold for all admissible α. In order to transform the parameter-dependent LMI into a finite set of standard LMIs, some information about $A(\alpha)$ must be added, as well as some structure must be imposed to the unknown variable $P(\alpha)$. For instance, consider that $A(\alpha)$ and $P(\alpha)$ have a polytopic structure

$$A(\alpha) = \sum_{i=1}^{N} \alpha_i A_i \ , \ P(\alpha) = \sum_{i=1}^{N} \alpha_i P_i \ , \quad \alpha \in \Delta_N, \tag{4}$$

being A_i and P_i the vertices of the respective polytopes, N the number of vertices and Δ_N the set known as unit simplex, given by

$$\Delta_N = \Big\{ \alpha \in \mathbb{R}^N : \sum_{i=1}^{N} \alpha_i = 1 \ , \quad \alpha_i \geq 0 \ , \quad i = 1, \ldots, N \Big\}. \tag{5}$$

Applying the definition of $A(\alpha)$ and the chosen structure for $P(\alpha)$, given by (4), to the robust stability condition expressed in the parameter-dependent LMI (3), one gets the following homogeneous polynomial matrix inequality, of degree 2 on α,

$$\begin{bmatrix} P(\alpha) & A(\alpha)'P(\alpha) \\ P(\alpha)A(\alpha) & P(\alpha) \end{bmatrix} = \sum_{i=1}^{N} \alpha_i^2 \begin{bmatrix} P_i & A_i'P_i \\ P_iA_i & P_i \end{bmatrix} + \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \alpha_i \alpha_j \begin{bmatrix} P_i + P_j & A_i'P_j + A_j'P_i \\ P_iA_j + P_jA_i & P_i + P_j \end{bmatrix} > 0. \tag{6}$$

A sufficient (but not necessary) way to guarantee that (3) holds is to impose that all the matrix coefficients of the monomials are positive definite. This has been done, for instance, in [18] (discrete-time systems) and [19] (continuous-time systems). A less conservative set of conditions may be obtained by modeling the variable $P(\alpha)$ in (6) as a homogeneous polynomial with generic degree $g > 1$ and then imposing the positivity of all matrix coefficients [12, 3]. Programming these LMIs requires an *a priori* knowledge on the formation law of the monomials, which depends on the number N of uncertain parameters and on the degree of the polynomial variable $P(\alpha)$. In [13], a systematic way to deal with such cases has been developed, but in the context of robust LMIs presenting at most products between two parameter-dependent matrices. When the LMIs to be solved are more complex and have products involving three or more parameter-dependent matrices, a systematic procedure to generate the coefficients of the monomials can be very hard (at least tedious) to be developed.

Consider now that matrix $A(\cdot)$ depends on two parameters, α and β, being each parameter contained in an independent unit simplex, *i.e.*,

$$A(\alpha, \beta) = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \alpha_i \beta_j A_{ij}, \quad \alpha \in \Delta_{N_1}, \beta \in \Delta_{N_2}, \tag{7}$$

That is, the parameters α and β lie in the multi-simplex $\Omega = \Delta_{N_1} \times \Delta_{N_2}$ (*i.e.*, the Cartesian product of Δ_{N_1} and Δ_{N_2}). Defining

$$P(\alpha, \beta) = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \alpha_i \beta_j P_{ij}, \tag{8}$$

then the parameter-dependent LMI to verify the robust stability of the system becomes

$$\begin{bmatrix} P(\alpha, \beta) & A(\alpha, \beta)'P(\alpha, \beta) \\ P(\alpha, \beta)A(\alpha, \beta) & P(\alpha, \beta) \end{bmatrix} > 0, \tag{9}$$

which is a completely different case than the one in (6), with $A(\alpha)$ described in terms of one simplex as in (4), therefore with a different formation law for the monomials. This illustrates that each new case requires the manipulation of different polynomials and such task, as well as programming the resulting LMIs, can be tedious, time-demanding and also a source of programming errors.

The toolbox described in this manual, called Robust LMI Parser[1], intends to overcome these problems by automatically computing all the coefficients of the monomials of a given parameter-dependent LMI. Those

[1] Available at `http://rolmip.github.io`

coefficients define the set of sufficient LMI conditions to be programmed. A feasible solution assures that the parameter-dependent LMI conditions are verified for the entire domain of uncertainty. The toolbox is developed for MATLAB and works jointly with YALMIP, returning the entire set of LMIs through a few simple commands that describe the structure of the known matrices and variables involved in the parameter-dependent LMIs to be investigated.

2 Installation notes

ROLMIP was built on the top of YALMIP [11], a freely distributed general purpose optimization parser MATLAB toolbox. Therefore the first step before using ROLMIP is to download[2] and install YALMIP.

The ROLMIP installation is simple: just unzip the installation file (`robust_lmi_parser.zip`) and then add the decompressed folder on MATLAB's path. To check if ROLMIP is working properly, run the script `rolmiptest.m`.

In order to solve the semidefinite problems proposed in this manual, it is necessary to install a proper solver. Among the several solvers publicly available, we recommend the following:

- SeDuMi (`http://sedumi.ie.lehigh.edu/`)
- MOSEK (`http://www.mosek.com/`)
- SDPT3 (`http://github.com/SQLP/SDPT3`)

The examples and outputs shown in this manual were generated by using YALMIP R20180612 with SeDuMi 1.3, operating under MATLAB R2015a.

2.1 Notes on Version 3.0

The current version is considerably different from the previous ones, mainly due to the introduction of a new variable type, named `rolmipvar`. With this modification, the commands and definitions are more straightforward and easy to use and understand, and even the computational time necessary to parse the parameter-dependent LMIs has been optimized. We strongly suggest the use of `rolmipvar` instead of the old commands `poly_struct` and `parser_poly`. However, such old commands are still implemented and even somewhat improved in the current version, and the MATLAB routines that already use ROLMIP continue to work.

Concerning specifically the automatic creation of executable MATLAB files: the latter version provided a set of procedures (using the old commands `poly_struct` and `parser_poly`) whose objective was to generate automatically MATLAB .m files, saving computational time if a set of conditions need to be executed several times. Such features are still part of ROLMIP, but not yet adapted to work using the `rolmipvar` variables included in the present version. Please refer to the manual of the previous version for further details.

2.2 YALMIP compatibility issues

Since YALMIP does not recognize the classes of ROLMIP in its source code, some compatibility issues may occur. Usually, such problems may happen if `sdpvar` and `rolmipvar` variables are mixed in the same expression. Thus, in order to avoid conflicts, we strongly suggest that the script under development be solely based on `rolmipvar` variables. If using both structures is unavoidable, it is important to guarantee that `rolmipvar` variables appear before the `sdpvar` ones when writing the expressions, as emphasized in Remark 2 in Section 7.2.

[2] `https://yalmip.github.io/download/`

3 Preliminaries

ROLMIP currently considers the following assumptions:

- The polynomial matrices are dependent on parameters that either are defined over a unit (or multi) simplex domain, or have known bounds;
- The number of vertices of a given simplex is always the same, although different simplex domains may present different numbers of vertices.

Several examples illustrating the usage of the commands are presented throughout the manual. Single commands are displayed using a font `similar to the one used in MATLAB` and start with >>. Fully functional scripts, presented in some examples, are encapsulated by a black box, while the MATLAB outputs generated by such scripts are displayed inside a red box.

4 Structure and definition of a polynomial variable

The main structure of ROLMIP is the `rolmipvar` class, which stores the informations regarding the polynomial structure of the parameter-dependent matrices. Internally, a `rolmipvar` variable is a structure composed by the following fields:

- `vertices`: Contains the number of vertices of each simplex;
- `data`: This field is a vector of elements (stored using the type `cell array` from MATLAB) that describes all the monomials of the variable. Each position of the vector is composed by the fields `value`, which contains the value of coefficient of the monomial, and `exponent`, describing the exponent of the monomial;
- `label`: Informs the label of the variable, which is used mainly for the LaTeX text generation;
- `opcode`: Describes how a variable is constructed, either if it is a user-defined variable or if it is the result of a previous mathematical operation;
- `bounds`: Used only if the polynomial depends on time-varying parameters with known variation rates.

The fields of the `rolmipvar` structure are not directly accessible by the user, and some of them (namely `opcode` and `bounds`) are defined only for internal purposes. For instance, consider a polynomial matrix $M(\alpha,\beta)$ defined in a multi-simplex domain as

$$M(\alpha,\beta) = \alpha_1^2\beta_1 M_1 + \alpha_1\alpha_2\beta_1 M_2 + \alpha_2^2\beta_1 M_3 + \alpha_1^2\beta_2 M_4 + \alpha_1\alpha_2\beta_2 M_5 + \alpha_2^2\beta_2 M_6 + \alpha_1^2\beta_3 M_7 + \alpha_1\alpha_2\beta_3 M_8 + \alpha_2^2\beta_3 M_9, \quad (10)$$

with $\alpha \in \Lambda_2$ and $\beta \in \Lambda_3$. The internal structure of the related `rolmipvar` variable is illustrated on Table 1.

Table 1: Structure of the `rolmipvar` variable representing the polynomial defined in (10).

`label` = 'M'						
`vertices` = [2 3]						
`data(1)`		`data(2)`		...	`data(9)`	
`value` = M_1		`value` = M_2			`value` = M_9	
`exponent{1}` [2 0]	`exponent{2}` [1 0 0]	`exponent{1}` [1 1]	`exponent{2}` [1 0 0]	...	`exponent{1}` [0 2]	`exponent{2}` [0 0 1]

4.1 Accessing the monomials of a `rolmipvar` variable

There are two ways to access the informations of the polynomials:

- Using the index operator ( ) with the exponents of the desired monomial. For instance, to consult the coefficient related to the monomial $\alpha_1^2\beta_2$ of the polynomial $M(\alpha,\beta)$ defined in (10), enter the command

```
>> M([2 0],[0 1 0]);
```

- By using the command

```
>> Mi = coeffs(poly);
```

which returns a cell structure containing all the coefficients of the monomials from the `rolmipvar` variable `poly`. In this case, the output contains only the monomials, and not the respective exponents, which can be obtained through the command

```
>> expMi = exponents(poly);
```

The output variable `expMi` is structured as

```
expMi{index}
```

where `index` corresponds to the coefficient stored in `ndexMii`. The structure of `ndexexpMii` is similar to the `exponent` component shown in Table 1.

4.2 Defining polynomial variables composed by known matrices in a multi-simplex domain

Every polynomial variable, regardless if depending on (multi) simplex or bounded parameters, is defined of type `rolmipvar`. The variable definition may be performed using several different syntaxes[3] depending on how the polynomial is described by the user, as detailed below.

If the matrix polynomial has known coefficients (as opposed to `sdpvar`[4] variables), the polynomial variable is obtained by

```
poly = rolmipvar(M,label,vertices,degree)
```

The output `poly` is a `rolmipvar` variable that fully describes the polynomial. The input `M` contains data about the coefficients and the monomials of the polynomial to be defined. The input `label` is a string and consists of a name used to refer the variable, used mainly to generate the LaTeX code of a polynomial through the command `texify`, presented later. The inputs `vertices` and `degree` inform, respectively, the number of parameters (vertices) and the polynomial degrees of the parameters associated to each simplex of the polynomial. If a parameter-independent matrix is to be defined, then the parameters `vertices` and `degree` may be omitted or set to zero.

For instance, consider that matrix $A(\alpha)$ has the following structure

$$A(\alpha) = \alpha_1 A_1 + \alpha_2 A_2, \quad \alpha_1 + \alpha_2 = 1, \quad \alpha_1 \geq 0, \alpha_2 \geq 0$$

i.e., $A(\alpha)$ is characterized by a polytope of $N = 2$ vertices and is modeled as a homogeneous polynomial of degree $g = 1$ with parameters in the unit simplex. There are three ways of inputting the coefficients A_1 and A_2 through variable `M`:

1. Concatenating the matrices A_1 and A_2;

```
>> M = [A1 A2];
```

2. Using `M` as a cell array;

```
>> M{1} = A1; M{2} = A2;
```
(11)

3. Informing, in the cell array `M`, the exponent of the monomial to which each matrix is related.

```
>> M{1} = {[1 0],A1}; M{2} = {[0 1],A2};
```
(12)

Using this syntax, the exponent of the monomial $\alpha_1^{n_1}\alpha_2^{n_2}\ldots\alpha_N^{n_N}$ is encoded as $[n_1\ n_2\ \ldots\ n_N]$. If more simplexes are used, then each cell has extra elements; refer to Example 2.

Please note that, if the first two ways are used, then every monomial must be defined, and the order of elements is important. On the other hand, if the third way is used, then there is no specific order, and the undefined monomials are set to zero by default.

Constant matrices may be defined by setting `vertices` with the number of vertices for the problem under investigation and `degree = 0`. However, the structure will be more complex and may result on more expensive computations.

[3] The syntaxes are somewhat similar to the `poly_struct` command available on prior versions.

[4] Variables used in YALMIP toolbox.

Example 1

Consider the polynomial matrix $A(\alpha)$ with degree $g = 1$ that represents a polytopic linear system with $N = 3$ vertices given by

$$A(\alpha) = \alpha_1 A_1 + \alpha_2 A_2 + \alpha_3 A_3, \quad \alpha \in \Delta_3,$$

being

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad A_3 = \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}.$$

Such a polynomial can be defined using the following script[5]

```
%Example 1
A1 = eye(2);
A2 = 2*eye(2);
A3 = 3*eye(2);
A = [A1 A2 A3];
polyA = rolmipvar(A,'A',3,1);
```

The contents of the `rolmipvar` variable `polyA` are displayed as

```
Label: A
Vertices: [3]
Degrees:  [1]

a1*[1  0]  + a2*[2  0]  + a3*[3  0]
   [0  1]       [0  2]       [0  3]
```

To retrieve the matrix-valued coefficient A_2, for example, it suffices to type

```
>> A2 = polyA([0 1 0]);
```

and, since the returned variable is a cell array, the coefficient can be accessed by typing

```
>> A2{1}
```

Also, the execution of the command

```
>> Ai = coeffs(polyA);
```

outputs, to the variable `Ai`, a cell structure containing each of the monomials of `polyA`. For instance, typing `Ai1` results in

```
ans =

     1     0
     0     1
```

Note that the command `coeffs` only displays the coefficients of the monomials, but not the respective exponents. Such information can be retrieved by using the command

```
>> expAi = exponents(polyA);
```

For instance, entering `expAi11` yields

```
ans =

     1     0     0
```

indicating that the coefficient `Ai{1}` refers to the monomial α_1.

[5] Available in the file **example_1.m** within the folder **manual_examples**.

Example 2

Let $A(\alpha,\beta)$ be the polynomial matrix given by

$$A(\alpha,\beta) = \alpha_1^2\beta_1 A_1 + \alpha_1\alpha_2\beta_1 A_2 + \alpha_2^2\beta_1 A_3 + \alpha_1^2\beta_2 A_4 + \alpha_1\alpha_2\beta_2 A_5 + \alpha_2^2\beta_2 A_6 + \alpha_1^2\beta_3 A_7 + \alpha_1\alpha_2\beta_3 A_8 + \alpha_2^2\beta_3 A_9,$$

i.e., with 2 vertices and degree 2 on the first simplex, and 3 vertices and degree 1 on the second simplex, being

$$A_k = \begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix}, \quad k = 1,\ldots,9.$$

The polynomial $A(\alpha,\beta)$ can be defined by using the script[6]

```
%Example 2
A1 = eye(2);
A2 = 2*eye(2);
A3 = 3*eye(2);
A4 = 4*eye(2);
A5 = 5*eye(2);
A6 = 6*eye(2);
A7 = 7*eye(2);
A8 = 8*eye(2);
A9 = 9*eye(2);

A{1} = {[2 0],[1 0 0],A1};
A{2} = {[1 1],[1 0 0],A2};
A{3} = {[0 2],[1 0 0],A3};
A{4} = {[2 0],[0 1 0],A4};
A{5} = {[1 1],[0 1 0],A5};
A{6} = {[0 2],[0 1 0],A6};
A{7} = {[2 0],[0 0 1],A7};
A{8} = {[1 1],[0 0 1],A8};
A{9} = {[0 2],[0 0 1],A9};
polyA = rolmipvar(A,'A',[2 3],[2 1]);
```

The contents of the variable `polyA` are given by

```
Label: A
Vertices: [2  3]
Degrees:  [2  1]

a1^2*b1*[1  0]  + a1*a2*b1*[2  0]  + a2^2*b1*[3  0]
        [0  1]             [0  2]             [0  3]

+ a1^2*b2*[1  0  0] + a1*a2*b2*[5  0] + a2^2*b2*[6  0]
          [0  1  0]            [0  5]           [0  6]

+ a1^2*b3*[7  0]  + a1*a2*b3*[8  0]  + a2^2*b3*[9  0]
          [0  7]             [0  8]            [0  9]
```

To retrieve the matrix-valued coefficient A_4, for example, it suffices to type

```
>> A4 = polyA([2 0],[0 1 0]);
```

and the desired coefficient is available in `A4{1}`.

In Example 2, two simplexes are used: one with α and another with β. Note that, when defining the structure of `A{·}`, the first element of the cell array corresponds to α and the second corresponds to β. When using ROLMIP, it is important to keep the track on the assortment of the defined simplexes, otherwise errors can occur. For ease of notation, in this manual the simplex names are sorted, by default, in the order $\{\alpha,\beta,\gamma,\delta,\ldots\}$.

A monomial can be individually set after being defined. For instance, in Example 2, to set the monomial with the term $\alpha_1^2\beta_2$ equal to identity I, one might enter the command `polyA([2 0],[0 1 0]) = eye(2);`

[6] Available in the file **example_2.m** within the folder **manual_examples**.

Example 3

Parameter-independent matrices can be constructed by simply omitting the input parameters `vertices` and `degree`. For instance, a 3×3 identity matrix can be defined using

```
polyI = rolmipvar(eye(3),'I');
```

Example 4

The parameter-dependent scalar variables

$$a(\alpha) = -\alpha_1 + \alpha_2, \quad b(\beta) = -3\beta_1 + 2\beta_2$$

can be defined through the script

```
%Example 4
ma{1} = {[1 0],-1};
ma{2} = {[0 1],1};
a = rolmipvar(ma,'a',2,1);
mb{1} = {[0 0],[1 0],-3};
mb{2} = {[0 0],[0 1],2};
b = rolmipvar(mb,'b',[0 2],[0 1]);
```

The contents of `a` and `b` are shown in the following.

```
>> a
Label: a
Vertices: [2]
Degrees:  [1]

a1*[-1]  + a2*[1]

>> b
Label: b
Vertices: [0  2]
Degrees:  [0  1]

b1*[-3]  + b2*[2]
```

4.3 Defining polynomials composed by optimization `sdpvar` variables in a multi-simplex domain

To declare a polynomial matrix as an optimization variable, one may use the following syntax

```
poly = rolmipvar(rows,cols,label,param,vertices,degree).
```

The coefficients of the monomials are matrices with dimensions `rows` × `cols` whose entries are optimization variables and, since ROLMIP is built on the top of YALMIP, they are internally stored using the `sdpvar` type provided by YALMIP. The argument `param` is a string that indicates if the variable is symmetric (`'symmetric'`), rectangular (`'full'`), symmetric Toeplitz (`'toeplitz'`), symmetric Hankel (`'hankel'`) or Skew-symmetric (`'skew'`). If `param` is not informed, square matrices are declared as symmetric and non-square matrices are defined as full. Note that such argument is similar to the `sdpvar` instruction used to define the variables in the YALMIP parser.

To define a known scalar in terms of a ROLMIP structure, one may use the syntax

```
poly = rolmipvar(M,label,'scalar'),
```

which returns the structure related to the scalar `M` with label given by `label`. If the scalar is a `sdpvar` variable, the following syntax may be used

```
poly = rolmipvar(label,'scalar').
```

It is important to highlight that it is not necessary to define parameter-independent matrices and scalars as `rolmipvar` variables, unless the user intends to generate the LaTeX code of the polynomials, as discussed in details in Section 5.

Example 5

A symmetric polynomial variable $P(\alpha) \in \mathbb{R}^{3\times 3}$ of degree 2, with α in a simplex of dimension $N = 3$, can be defined by

```
>> polyP = rolmipvar(3,3,'P','symmetric',3,2);
```

Example 6

A full polynomial variable $P(\alpha, \beta) \in \mathbb{R}^{3\times 3}$ of degree 2 and $N = 3$ vertices on the first simplex, and degree 1 and $N = 4$ vertices on the second simplex, can be defined by

```
>> polyP = rolmipvar(3,3,'P','symmetric',[3 4],[2 1]);
```

4.4 Defining matrices polynomially dependent on bounded parameters

In several situations, the system matrices depend on uncertain parameters with known bounds, and a series of transformations must be applied in order to rewrite such matrices within a multi-simplex domain. The situations considered in the prior cases are all matrices already represented on a multi-simplex. However, it is possible to use ROLMIP to represent matrices polynomially dependent on bounded parameters without previously applying such transformations.

Suppose, for instance, that one needs to define a polynomial matrix $A(\theta_1, \theta_2)$, given by

$$A(\theta_1, \theta_2) = A_0 + \theta_1 A_1 + \theta_2 A_2 + \theta_1^2 \theta_2 A_3,$$

with

$$\underline{a}_1 \leq \theta_1 \leq \overline{a}_1, \quad \underline{a}_2 \leq \theta_2 \leq \overline{a}_2.$$

ROLMIP allows the definition of such polynomial, provided that the bounds of the parameters are also informed. Internally the transformation to a multi-simplex domain is applied, but such a transformation is not explicit to the user, easing further polynomial operations. The syntax for such is

```
poly = rolmipvar(M,label,bounds).
```

The input `M` is a cell array that contains the informations on the matrix coefficients and the associated monomials, defined in a similar way as in (12). The input `label` informs the label of the variable, and `bounds` is an $m \times 2$ matrix that contains the bounds of the m parameters considered (first column with the lower limits and the second column with the upper limits).

Example 7

The polynomial matrix $A(\theta_1, \theta_2)$, given by

$$A(\theta_1, \theta_2) = A_0 + \theta_1 A_1 + \theta_2 A_2 + \theta_1^2 \theta_2 A_3,$$

with

$$-2 \leq \theta_1 \leq 3, \quad 4 \leq \theta_2 \leq 8,$$

and

$$A_k = \begin{bmatrix} k+1 & 0 \\ 0 & k+1 \end{bmatrix}, \quad k = 0, 1, 2, 3$$

can be defined using the following sequence of commands.

```
%Example 7
A0 = [1 0; 0 1];
A1 = [2 0; 0 2];
A2 = [3 0; 0 3];
A3 = [4 0; 0 4];

A{1} = {[0 0],A0};
A{2} = {[1 0],A1};
A{3} = {[0 1],A2};
A{4} = {[2 1],A3};
polyA = rolmipvar(A,'A',[-2 3; -4 8]);
```

The contents of the variable `polyA` are output in MATLAB as follows.

```
Label: A
Vertices: [2  2]
Degrees:  [2  1]

a1^2*b1*[-79   0]  + a1*a2*b1*[172    0]
        [0 -79]               [0  172]

+ a2^2*b1*[-149    0]  + a1^2*b2*[149    0]
          [0 -149]               [0  149]

+ a1*a2*b2*[-332    0]  + a2^2*b2*[319    0]
           [0 -332]               [0  319]
```

If the polynomial matrix is an optimization variable, one may use the syntax

`poly = rolmipvar(rows,cols,label,parametr,polmask,bounds).`

The matrix coefficients with dimensions `rows` × `cols` are internally declared, and returned as `sdpvar` variables to the output `poly`. The argument `parametr` is a string that indicates if the variable is symmetric (`'symmetric'`), rectangular (`'full'`), symmetric Toeplitz (`'toeplitz'`), symmetric Hankel (`'hankel'`) or Skew-symmetric (`'skew'`). If `parametr` is not informed, square matrices are declared as symmetric and non-square matrices are defined as full. Note that such command is similar to the `sdpvar` instruction used to define the variables in the YALMIP parser. The input `polmask` is a cell array containing the exponents of the parameters desired for the output polynomial, and `bounds` contains the bounds of the parameters.

Example 8

To define the 3×3 symmetric polynomial variable $P(\theta_1, \theta_2)$, whose desired structure is given by

$$P(\theta_1, \theta_2) = P_0 + \theta_1 P_1 + \theta_2^3 P_2 + \theta_1^2 \theta_2^4 P_3,$$

with

$$-2 \leq \theta_1 \leq 3, \quad 4 \leq \theta_2 \leq 8,$$

one may use the syntax

```
>> polyP = rolmipvar(3,3,'P','sym',{[0 0],[1 0],[0 3],[2 4]},[-2 3; -4 8]);
```

The output of typing `polyP` is shown as follows.

```
Linear matrix variable 3x3 (symmetric, real, 24 variables)
Label: P
Vertices: [2  2]
Degrees:  [2  4]
```

Internally, the polynomial with parameters lying in known intervals is converted to a multi-simplex representation, each parameter generating a different simplex with 2 vertices. In general terms, the parameter

θ_i is rewritten as

$$\theta_i = \alpha_1 \underline{a}_i + \alpha_2 \overline{a}_i,$$

and then the polynomial is homogenized.

5 Operating on polynomials

Once the `rolmipvar` variables are defined, the most common operations may be performed between polynomials or between constant matrices and polynomials. Every necessary homogenization is automatically performed by ROLMIP, as illustrated in the following example.

Example 9

Suppose that the `rolmipvar` variables `polyA` and `polyB` have already been defined and are respectively given by

$$A(\alpha) = \alpha_1 A_1 + \alpha_2 A_2, \quad B(\alpha) = \alpha_1 B_1 + \alpha_2 B_2.$$

The sum

```
>> resul1 = polyA + polyB;
```

yields a polynomial given by

$$A(\alpha) + B(\alpha) = \alpha_1(A_1 + B_1) + \alpha_2(A_2 + B_2)$$

The product

```
>> resul2 = polyA*polyB;
```

yields a polynomial given by

$$A(\alpha)B(\alpha) = \alpha_1^2(A_1B_1) + \alpha_1\alpha_2(A_1B_2 + A_2B_1) + \alpha_2^2(A_2 + B_2).$$

Finally, operations between polynomials and non-polynomial matrices are performed as expected. Let C be a matrix (with compatible dimensions) defined on MATLAB. The operation

```
>> resul3 = polyA + C;
```

produces

$$A(\alpha) + C = \alpha_1(A_1 + C) + \alpha_2(A_2 + C),$$

and the operation

```
>> resul4 = polyA*C;
```

results on

$$A(\alpha)C = \alpha_1(A_1C) + \alpha_2(A_2C).$$

The following script[7] is an example for the implementation of the given commands, along with the resulting outputs.

```
%Example 9
A1 = eye(2);
A2 = 2*eye(2);
A = [A1 A2];
polyA = rolmipvar(A,'A',2,1);

B1 = 3*eye(2);
B2 = 4*eye(2);
B = [B1 B2];
polyB = rolmipvar(B,'B',2,1);

resul1 = polyA + polyB

resul2 = polyA*polyB

C = 5*eye(2);
resul3 = polyA + C

resul4 = polyA*C
```

[7] Available in the file `example_9.m` within the folder `manual_examples`.

```
>> resul1
Label: A+B
Vertices: [2]
Degrees:  [1]

a1*[4  0]  + a2*[6  0]
   [0  4]       [0  6]

>> resul2
Label: A*B
Vertices: [2]
Degrees:  [2]

a1^2*[3  0]  + a1*a2*[10   0]  + a2^2*[8  0]
     [0  3]          [0  10]          [0  8]

>> resul3
Label: A+<>
Vertices: [2]
Degrees:  [1]

a1*[6  0]  + a2*[7  0]
   [0  6]       [0  7]

>> resul4
Label: A*<>
Vertices: [2]
Degrees:  [1]

a1*[5  0]  + a2*[10   0]
   [0  5]       [0  10]
```

Remark 1
The string <> appearing in the `Label` property is a standard string to refer to non-labeled variables, such as the variable `C`.

The concatenation of several polynomial `rolmipvar` variables in a matrix is done using the same notation of other matrix concatenations in MATLAB, as illustrated in the following example.

Example 10

Consider the following matrix-valued variables

$$A(\alpha) = \alpha_1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \alpha_2 \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad B(\beta) = \beta_1 \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix} + \beta_2 \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}.$$

The matrix

$$T(\alpha, \beta) = \begin{bmatrix} A(\alpha) & B(\beta) \\ B(\beta) & 0 \end{bmatrix}$$

can be constructed through the following script.

```
%Example 10
A1 = eye(2);
A2 = 2*eye(2);
B1 = 3*eye(2);
B2 = 4*eye(2);

A{1} = {[1 0],A1};
A{2} = {[0 1],A2};
B{1} = {[1 0],B1};
B{2} = {[0 1],B2};

polyA = rolmipvar(A,'A',2,1);
polyB = rolmipvar(B,'B',2,1);
polyB = fork(polyB,'B',1,2);

polyT = [polyA polyB;
    polyB zeros(2,2)];
```

```
>> polyT
Label: [[A,B];[B,<>]]
Vertices: [2  2]
Degrees:  [1  1]

a1*b1*[1  0  3  0]  + a2*b1*[2  0  3  0]
      [0  1  0  3]          [0  2  0  3]
      [3  0  0  0]          [3  0  0  0]
      [0  3  0  0]          [0  3  0  0]

+ a1*b2*[1  0  4  0]  + a2*b2*[2  0  4  0]
        [0  1  0  4]          [0  2  0  4]
        [4  0  0  0]          [4  0  0  0]
        [0  4  0  0]          [0  4  0  0]
```

In general, the MATLAB operations that are adapted to work over `rolmipvar` variables are:

- Transpose operation: the transpose is applied over the matrix-valued coefficients of all monomials;
- Command `blkdiag`: used if one needs to construct a `rolmipvar` block-diagonal matrix;
- Command `trace`: returns a `rolmipvar`, with each coefficient corresponding to the trace of the respective matrix monomial, since

$$\text{trace}\left(\sum_{i=1}^{N}\alpha_i A_i\right)=\sum_{i=1}^{N}\alpha_i \text{trace}(A_i).$$

Other implemented commands are described in the following.

Command FORK

This command replaces the parameters from one (multi) simplex domain to another.

The syntax

```
polyout = fork(polyin,newlabel)
```

changes the entire multi-simplex domain of the polynomial `polyin`, generating a new variable with label `newlabel`.

Example 11

Suppose that `polyin` represents the polynomial

$$A(\alpha)=\alpha_1 A_1+\alpha_2 A_2.$$

The application of the command `fork` results on the polynomial `polyout` that represents

$$A(\beta) = \beta_1 A_1 + \beta_2 A_2,$$

as shown in the following script[8]. Note that the `fork` command is useful to declare variables in different simplexes (for instance the scalar variables in Example 4) in a simpler way.

```
%Example 11
A1 = eye(2);
A2 = 2*eye(2);
A = [A1 A2];
polyA = rolmipvar(A,'A',2,1)

polyAfork = fork(polyA,'Afork')
```

```
>> polyA
Label: A
Vertices: [2]
Degrees:  [1]

a1*[1  0]  + a2*[2  0]
   [0  1]       [0  2]

>> polyAfork
Label: Afork
Vertices: [0  2]
Degrees:  [0  1]

b1*[1  0]  + b2*[2  0]
   [0  1]       [0  2]
```

Example 12

Suppose that `polyin` represents the polynomial

$$A(\alpha, \beta) = \alpha_1\beta_1 A_1 + \alpha_2\beta_1 A_2 + \alpha_1\beta_2 A_3 + \alpha_2\beta_2 A_4.$$

The application of the command `fork` results on the polynomial `polyout` that represents

$$A(\gamma, \delta) = \gamma_1\delta_1 A_1 + \gamma_2\delta_1 A_2 + \gamma_1\delta_2 A_3 + \gamma_2\delta_2 A_4,$$

as shown in the following[9].

```
%Example 12
A1 = eye(2);
A2 = 2*eye(2);
A3 = 3*eye(2);
A4 = 4*eye(2);

A{1} = {[1 0],[1 0],A1};
A{2} = {[0 1],[1 0],A2};
A{3} = {[1 0],[0 1],A3};
A{4} = {[0 1],[0 1],A4};
polyA = rolmipvar(A,'A',[2 2],[1 1])

polyAfork = fork(polyA,'Afork')
```

[8] Available in the file **example_11.m** within the folder **manual_examples**.
[9] Available in the file **example_12.m** within the folder **manual_examples**.

```
>> polyA
Label: A
Vertices: [2  2]
Degrees:  [1  1]

a1*b1*[1  0]  + a2*b1*[2  0]  + a1*b2*[3  0]  + a2*b2*[4  0]
      [0  1]          [0  2]          [0  3]          [0  4]

>> polyAfork
Label: Afork
Vertices: [0  0  2  2]
Degrees:  [0  0  1  1]

c1*d1*[1  0]  + c2*d1*[2  0]  + c1*d2*[3  0]  + c2*d2*[4  0]
      [0  1]          [0  2]          [0  3]          [0  4]
```

Note that the command `fork` only changes the parameters that define the monomials, but not the values of the coefficients of the monomials.

The syntax

```
polyout = fork(polyin,newlabel,targetin)
```

only change the simplexes given in input vector `targetin`.

Example 13

Suppose that `polyin` represents the polynomial

$$A(\alpha,\beta,\gamma) = \alpha_1\beta_1\gamma_1 A_1 + \alpha_2\beta_1\gamma_2 A_2 + \alpha_1\beta_2\gamma_3 A_3 + \alpha_2\beta_2\gamma_1 A_4.$$

The command

```
>> polyout = fork(polyin,'B',[2]);
```

provides the polynomial `polyout` that represents

$$B(\alpha,\delta,\gamma) = \alpha_1\delta_1\gamma_1 A_1 + \alpha_2\delta_1\gamma_2 A_2 + \alpha_1\delta_2\gamma_3 A_3 + \alpha_2\delta_2\gamma_1 A_4.$$

```
%Example 13
A1 = eye(2);
A2 = 2*eye(2);
A3 = 3*eye(2);
A4 = 4*eye(2);

A{1} = {[1 0],[1 0],[1 0 0],A1};
A{2} = {[0 1],[1 0],[0 1 0],A2};
A{3} = {[1 0],[0 1],[0 0 1],A3};
A{4} = {[0 1],[0 1],[1 0 0],A4};
polyA = rolmipvar(A,'A',[2 2 3],[1 1 1])

polyB = fork(polyA,'B',2)
```

```
>> polyA
Label: A
Vertices: [2  2  3]
Degrees:  [1  1  1]

a1*b1*c1*[1  0]  + a2*b1*c1*[0  0]  + a1*b2*c1*[0  0]
         [0  1]             [0  0]             [0  0]

+ a2*b2*c1*[4  0]  + a1*b1*c2*[0  0]  + a2*b1*c2*[2  0]
           [0  4]             [0  0]             [0  2]

+ a1*b2*c2*[0  0]  + a2*b2*c2*[0  0]  + a1*b1*c3*[0  0]
           [0  0]             [0  0]             [0  0]

+ a2*b1*c3*[0  0]  + a1*b2*c3*[3  0]  + a2*b2*c3*[0  0]
           [0  0]             [0  3]             [0  0]

>> polyB
Label: B
Vertices: [2  0  3  2]
Degrees:  [1  0  1  1]

a1*c1*d1*[1  0]  + a2*c1*d1*[0  0]  + a1*c2*d1*[0  0]
         [0  1]             [0  0]             [0  0]

+ a2*c2*d1*[2  0]  + a1*c3*d1*[0  0]  + a2*c3*d1*[0  0]
           [0  2]             [0  0]             [0  0]

+ a1*c1*d2*[0  0]  + a2*c1*d2*[4  0]  + a1*c2*d2*[0  0]
           [0  0]             [0  4]             [0  0]

+ a2*c2*d2*[0  0]  + a1*c3*d2*[3  0]  + a2*c3*d2*[0  0]
           [0  0]             [0  3]             [0  0]
```

Finally, the syntax

```
polyout = fork(polyin,newlabel,targetin,targetout)
```

changes the simplexes on input vector `targetin`, respectively, to the simplexes informed on `targetout`.

Example 14

Suppose that `polyin` represents the polynomial

$$A(\alpha,\beta,\gamma) = \alpha_1\beta_1\gamma_1 A_1 + \alpha_2\beta_1\gamma_2 A_2 + \alpha_1\beta_2\gamma_3 A_3 + \alpha_2\beta_2\gamma_1 A_4.$$

The command

```
>> polyout = fork(polyin,'B',[1 2],[2 4]);
```

results on the polynomial `polyout` that represents

$$A(\beta,\delta,\gamma) = \beta_1\delta_1\gamma_1 A_1 + \beta_2\delta_1\gamma_2 A_2 + \beta_1\delta_2\gamma_3 A_3 + \beta_2\delta_2\gamma_1 A_4.$$

```
%Example 14
A1 = eye(2);
A2 = 2*eye(2);
A3 = 3*eye(2);
A4 = 4*eye(2);

A{1} = {[1 0],[1 0],[1 0 0],A1};
A{2} = {[0 1],[1 0],[0 1 0],A2};
A{3} = {[1 0],[0 1],[0 0 1],A3};
A{4} = {[0 1],[0 1],[1 0 0],A4};
polyA = rolmipvar(A,'A',[2 2 3],[1 1 1])

polyB = fork(polyA,'B',[1 2],[2 4])
```

```
>> polyA
Label: A
Vertices: [2  2  3]
Degrees:  [1  1  1]

a1*b1*c1*[1  0]  + a2*b1*c1*[0  0]  + a1*b2*c1*[0  0]
         [0  1]             [0  0]             [0  0]

+ a2*b2*c1*[4  0]  + a1*b1*c2*[0  0]  + a2*b1*c2*[2  0]
           [0  4]             [0  0]             [0  2]

+ a1*b2*c2*[0  0]  + a2*b2*c2*[0  0]  + a1*b1*c3*[0  0]
           [0  0]             [0  0]             [0  0]

+ a2*b1*c3*[0  0]  + a1*b2*c3*[3  0]  + a2*b2*c3*[0  0]
           [0  0]             [0  3]             [0  0]

>> polyB
Label: B
Vertices: [0  2  3  2]
Degrees:  [0  1  1  1]

b1*c1*d1*[1  0]  + b2*c1*d1*[0  0]  + b1*c2*d1*[0  0]
         [0  1]             [0  0]             [0  0]

+ b2*c2*d1*[2  0]  + b1*c3*d1*[0  0]  + b2*c3*d1*[0  0]
           [0  2]             [0  0]             [0  0]

+ b1*c1*d2*[0  0]  + b2*c1*d2*[4  0]  + b1*c2*d2*[0  0]
           [0  0]             [0  4]             [0  0]

+ b2*c2*d2*[0  0]  + b1*c3*d2*[3  0]  + b2*c3*d2*[0  0]
           [0  0]             [0  3]             [0  0]
```

Command ADDSIMPLEX

The command `addsimplex`, whose syntax is

```
>> polyout = addsimplex(polyin,vertices);
```

adds a new simplex to the informed polynomial `polyin`, and the inserted simplex has the number of vertices indicated in input `vertices`.

Example 15

Suppose that `polyin` represents the polynomial

$$A(\alpha, \beta, \gamma).$$

The command

```
>> polyout = addsimplex(polyin,3);
```

results on the polynomial `polyout` that represents

$$A(\alpha, \beta, \gamma, \delta),$$

being δ a simplex with 3 vertices.

```
%Example 15
A1 = eye(2);
A2 = 2*eye(2);
A3 = 3*eye(2);
A4 = 4*eye(2);

A{1} = {[1 0],[1 0],[1 0 0],A1};
A{2} = {[0 1],[1 0],[0 1 0],A2};
A{3} = {[1 0],[0 1],[0 0 1],A3};
A{4} = {[0 1],[0 1],[1 0 0],A4};
polyA = rolmipvar(A,'A',[2 2 3],[1 1 1])

polyout = addsimplex(polyA,3)
```

```
>> polyA
Label: A
Vertices: [2  2  3]
Degrees:  [1  1  1]

a1*b1*c1*[1  0]  + a2*b1*c1*[0  0]  + a1*b2*c1*[0  0]
         [0  1]             [0  0]             [0  0]

+ a2*b2*c1*[4  0]  + a1*b1*c2*[0  0]  + a2*b1*c2*[2  0]
           [0  4]             [0  0]             [0  2]

+ a1*b2*c2*[0  0]  + a2*b2*c2*[0  0]  + a1*b1*c3*[0  0]
           [0  0]             [0  0]             [0  0]

+ a2*b1*c3*[0  0]  + a1*b2*c3*[3  0]  + a2*b2*c3*[0  0]
           [0  0]             [0  3]             [0  0]

>> polyout
Label: A
Vertices: [2  2  3  3]
Degrees:  [1  1  1  0]

a1*b1*c1*[1  0]  + a2*b1*c1*[0  0]  + a1*b2*c1*[0  0]
         [0  1]             [0  0]             [0  0]

+ a2*b2*c1*[4  0]  + a1*b1*c2*[0  0]  + a2*b1*c2*[2  0]
           [0  4]             [0  0]             [0  2]

+ a1*b2*c2*[0  0]  + a2*b2*c2*[0  0]  + a1*b1*c3*[0  0]
           [0  0]             [0  0]             [0  0]

+ a2*b1*c3*[0  0]  + a1*b2*c3*[3  0]  + a2*b2*c3*[0  0]
           [0  0]             [0  3]             [0  0]
```

Command EVALPAR

The command `evalpar` is used to compute the value of the matrix polynomial given the values of the parameters. The syntax is

```
>> var = evalpar(poly,paramval)
```

The input `poly` contains the `rolmipvar` variable, and `paramval` contains the values of the parameters. If the polynomial is defined directly on the (multi) simplex, then `paramval` is a cell array (even if the polynomial is defined over only one simplex), and `paramval` must be a vector if `poly` is a matrix polynomially dependent on bounded parameters.

Example 16

Suppose that `poly` represents the polynomial

$$A(\alpha) = \alpha_1^2 \begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} + \alpha_1\alpha_2 \begin{bmatrix} -3 & 4 \\ 2 & 3 \end{bmatrix} + \alpha_2^2 \begin{bmatrix} 0 & 1 \\ -2 & 4 \end{bmatrix}.$$

The command

```
>> var = evalpar(poly,{[0.3 0.7]})
```

performs the calculation

$$(0.3)^2 \begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} + (0.3)(0.7) \begin{bmatrix} -3 & 4 \\ 2 & 3 \end{bmatrix} + (0.7)^2 \begin{bmatrix} 0 & 1 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} -0.54 & 1.33 \\ -0.38 & 2.5 \end{bmatrix}$$

```
%Example 16
A{1} = {[2 0],[1 0; 2 -1]};
A{2} = {[1 1],[-3 4; 2 3]};
A{3} = {[0 2],[0 1; -2 4]};
poly = rolmipvar(A,'A',2,2);
var = evalpar(poly,{[0.3 0.7]})
```

```
>> var
var =

   -0.5400    1.3300
   -0.3800    2.5000
```

Example 17

Suppose that `poly` represents the polynomial

$$A(\alpha,\beta) = \alpha_1\beta_1^2 \begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} + \alpha_1\beta_1\beta_2 \begin{bmatrix} -3 & 4 \\ 2 & 3 \end{bmatrix} + \alpha_2\beta_1\beta_2 \begin{bmatrix} 0 & 1 \\ -2 & 4 \end{bmatrix} + \alpha_2\beta_2^2 \begin{bmatrix} 4 & 7 \\ 1 & -2 \end{bmatrix}.$$

The command

```
>> var = evalpar(poly,{[0.3 0.7],[0.1 0.9]})
```

performs the calculation

$$(0.3)(0.1)^2 \begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix} + (0.3)(0.1)(0.9) \begin{bmatrix} -3 & 4 \\ 2 & 3 \end{bmatrix} + (0.7)(0.1)(0.9) \begin{bmatrix} 0 & 1 \\ -2 & 4 \end{bmatrix} + (0.7)(0.9)^2 \begin{bmatrix} 4 & 7 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} 2.190 & 4.140 \\ 0.501 & -0.804 \end{bmatrix}$$

```
%Example 17
A{1} = {[1 0],[2 0],[1 0; 2 -1]};
A{2} = {[1 0],[1 1],[-3 4; 2 3]};
A{3} = {[0 1],[1 1],[0 1; -2 4]};
A{4} = {[0 1],[0 2],[4 7; 1 -2]};
poly = rolmipvar(A,'A',[2 2],[1 2]);
var = evalpar(poly,{[0.3 0.7],[0.1 0.9]})
```

```
>> var
var =

    2.1900    4.1400
    0.5010   -0.8040
```

Example 18

Suppose that `poly` represents the polynomial

$$A(\theta) = \begin{bmatrix} 0 & 1 \\ 2 & 4 \end{bmatrix} + \theta_1 \begin{bmatrix} 2 & 3 \\ -2 & 5 \end{bmatrix} + \theta_1^2\theta_2 \begin{bmatrix} 5 & -1 \\ 2 & 0 \end{bmatrix} + \theta_2^2 \begin{bmatrix} 3 & 3 \\ 0 & 8 \end{bmatrix}, \quad -8 \leq \theta_1 \leq 8, \quad -8 \leq \theta_2 \leq 8.$$

The command

```
>> var = evalpar(poly,[2 3])
```

performs the calculation

$$A(\theta) = \begin{bmatrix} 0 & 1 \\ 2 & 4 \end{bmatrix} + 2 \begin{bmatrix} 2 & 3 \\ -2 & 5 \end{bmatrix} + (2^2)3 \begin{bmatrix} 5 & -1 \\ 2 & 0 \end{bmatrix} + 3^2 \begin{bmatrix} 3 & 3 \\ 0 & 8 \end{bmatrix} = \begin{bmatrix} 91 & 22 \\ 22 & 86 \end{bmatrix}.$$

```
%Example 18
A{1} = {[0 0],[0 1; 2 4]};
A{2} = {[1 0],[2 3; -2 5]};
A{3} = {[2 1],[5 -1; 2 0]};
A{4} = {[0 2],[3 3; 0 8]};
poly = rolmipvar(A,'A',[-8 8; -8 8]);
var = evalpar(poly,[2 3])
```

```
>> var
var =

    91    22
    22    86
```

Command DOUBLE

In YALMIP parser [11], the command `double` is applied on `sdpvar` variables, returning the double precision values of the variables (if such values are already assigned). If the command `double` is applied on a `rolmipvar` variable, it returns the respective polynomial with monomials given on the double precision values.

Command DIFF

Usually, when dealing with linear parameter-varying (LPV) systems, where the parameters may be time-varying, the parameter-dependent inequalities depend on the time-derivative of some parameter-dependent matrix variables. For example, the continuous-time LPV system

$$\dot{x}(t) = A(\alpha(t))x(t), \quad \alpha(t) \in \Delta_N,$$

is asymptotically stable if there is a symmetric positive definite matrix $P(\alpha(t))$ satisfying

$$A(\alpha(t))'P(\alpha(t)) + P(\alpha(t))A(\alpha(t)) + \frac{d}{dt}P(\alpha(t)) < 0.$$

For ease of notation, consider that

$$\frac{d}{dt}P(\alpha(t)) = \dot{P}(\alpha(t)).$$

Suppose that the parameter-dependent variable $P(\alpha(t))$ depends affinely on $\alpha(t)$, that is

$$P(\alpha(t)) = \sum_{i=1}^{N} \alpha_i(t) P_i.$$

Then one has

$$\dot{P}(\alpha(t)) = \sum_{i=1}^{N} \dot{\alpha}_i(t) P_i.$$

If the variation rates of the parameters $\alpha(t)$ are bounded, and such bounds are previously known, then the space where the parameters $\alpha_i(t)$ and $\dot{\alpha}_i(t)$ can assume values can be modeled as a polytope described by the set

$$\Upsilon = \left\{ \varphi \in \mathbb{R}^N \ : \ \varphi = \sum_{\ell=1}^{R} \beta_\ell h^\ell, \quad \sum_{i=1}^{N} h_i^\ell = 0, \quad \forall \ell = 1, \ldots, R, \quad \beta \in \Lambda_R \right\},$$

where the vectors h^ℓ are the vertices of the polytope. Using this representation, $\dot{P}(\alpha(t))$ can be represented in a polytopic way over a different simplex, such as

$$\dot{P}(\alpha(t)) = \sum_{i=1}^{N} \sum_{\ell=1}^{M} \beta_\ell h_i^\ell P_i, \qquad \beta \in \Delta_M,$$

For more details on this representation, please refer to [6, 8].

The presented transformation can be automatically performed using the command `diff` of ROLMIP, whose standard syntax is

```
>> dotpoly = diff(poly,labelout,dotbounds)
```

The polynomial to be derived is given by `poly`, the label of the derived polynomial is given by `labelout`[10], and `dotbounds` contains the bounds of the variation rates of the parameters. By default, the Multi-Parametric Toolbox (MPT) [10] is used, if installed, to generate the vectors h^ℓ, through the application of vertex enumeration algorithms [2], which are computationally faster. Otherwise, the transformation is computed using a recursive technique.

Note that, without knowing such bounds, it is not possible to generate a polytope where the vector $\dot{\alpha}(t)$ lies and, as a consequence, the variable $P(\alpha(t))$ must be $\alpha(t)$ independent, that is, a polynomial of degree zero on $\alpha(t)$. If the input polynomial depends on only one simplex, then `dotbounds` can be a vector; otherwise, it must be a cell array, each cell containing the information about each simplex.

Example 19

Suppose that one needs to calculate the derivative of the polynomial `poly`, defined over one simplex of three vertices, being the variation rates of each parameter bounded by

$$-1 \leq \dot{\alpha}_1(t) \leq 1, \quad -1 \leq \dot{\alpha}_2(t) \leq 1, \quad -1 \leq \dot{\alpha}_3(t) \leq 1.$$

The vectors that describe the polytopic region where the parameters $\dot{\alpha}_i(t)$ lie are given by

$$\begin{bmatrix} h^1 & \cdots & h^6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & -1 & 0 & 1 \\ 0 & 1 & 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

and a graphical illustration is depicted in Figure 1.

The derivative of the polynomial is obtained by the command

```
>> dotpoly = diff(poly,labelout,[-1 1; -1 1; -1 1]);
```

[10] If the input `labelout` is not informed, then the label of the output polynomial is composed by the string `dot_` concatenated with the label of the input polynomial.

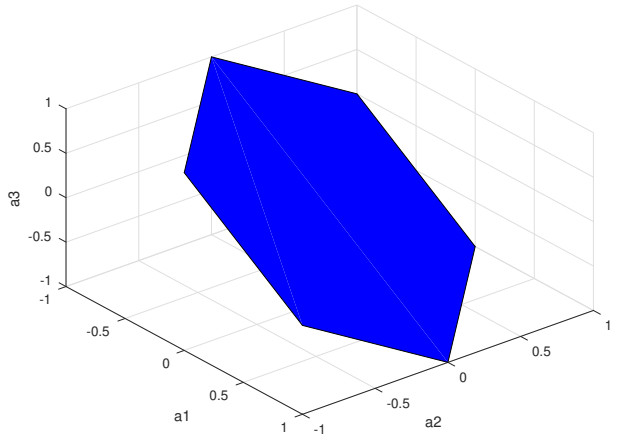

Figure 1: Feasible region associated to the constraints $-1 \leq \dot{\alpha}_i \leq 1$ and $\dot{\alpha}_1 + \dot{\alpha}_2 + \dot{\alpha}_3 = 0$.

```
%Example 19
A{1} = {[1 0 0],eye(2)};
A{2} = {[0 1 0],2*eye(2)};
A{3} = {[0 1 0],3*eye(2)};
poly = rolmipvar(A,'A',3,1);
dotpoly = diff(poly,'dpoly',[-1 1; -1 1; -1 1])
```

```
Label: dpoly
Vertices: [3  6]
Degrees:  [1  1]

a1*b1*[-1  0]  + a2*b1*[-1  0]  + a3*b1*[-1  0]  + a1*b2*[1   0]  + a2*b2*[1   0]  +
      [ 0 -1]          [ 0  1]          [ 0 -1]          [0   1]          [0   1]
a3*b2*[1   0]  + a1*b3*[-1  0]  + a2*b3*[-1  0]  + a3*b3*[-1  0]  + a1*b4*[-2  0]  +
      [0   1]          [0  -1]          [0  -1]          [0  -1]          [0  -2]
a2*b4*[-2  0]  + a3*b4*[-2  0]  + a1*b5*[1   0]  + a2*b5*[1   0]  + a3*b5*[1   0]  +
      [0  -2]          [0   1]          [0   1]          [0   1]          [0  -2]
a1*b6*[2   0]  + a2*b6*[2   0]  + a3*b6*[2   0]
      [0   2]          [0   2]          [0   2]
>>
```

Example 20

Suppose that one needs to calculate the derivative of the polynomial `poly`, defined over two simplexes: the first (α) of three vertices, and the second (β) of two vertices. The variation rates of each parameter are bounded by

$$-1 \leq \dot{\alpha}_1(t) \leq 2, \quad -3 \leq \dot{\alpha}_2(t) \leq 4, \quad -8 \leq \dot{\alpha}_3(t) \leq 6, \quad -4 \leq \dot{\beta}_1(t) \leq 1, \quad -6 \leq \dot{\beta}_2(t) \leq 9.$$

The derivative of the polynomial is obtained by the commands

```
>> dotbounds{1} = [-1 2; -3 4; -8 6];
>> dotbounds{2} = [-4 1; -6 9];
>> dotpoly = diff(poly,labelout,dotbounds);
```

```
%Example 20
A{1} = {[1 0 0],[1 0],eye(2)};
A{2} = {[1 0 0],[0 1],2*eye(2)};
A{3} = {[0 1 0],[1 0],3*eye(2)};
A{4} = {[0 1 0],[0 1],4*eye(2)};
A{5} = {[0 0 1],[1 0],5*eye(2)};
A{6} = {[0 0 1],[0 1],6*eye(2)};
poly = rolmipvar(A,'A',[3 2],[1 1]);
dotbounds{1} = [-1 2; -3 4; -8 6];
dotbounds{2} = [-4 1; -6 9];
dotpoly = diff(poly,'dpoly',dotbounds)
```

```
>> dotpoly
a1*b1*c1*d1*[14   0]  + a2*b1*c1*d1*[14   0]  + a3*b1*c1*d1*[14   0]
            [0  14]                 [0  14]                 [0  14]

+ a1*b2*c1*d1*[14   0]  + a2*b2*c1*d1*[14   0]  + a3*b2*c1*d1*[14   0]
              [0  14]                 [0  14]                 [0  14]

+ a1*b1*c2*d1*[0  0]  + a2*b1*c2*d1*[0  0]  + a3*b1*c2*d1*[0  0]
              [0  0]                [0  0]                [0  0]

+ a1*b2*c2*d1*[0  0]  + a2*b2*c2*d1*[0  0]  + a3*b2*c2*d1*[0  0]
              [0  0]                [0  0]                [0  0]

+ a1*b1*c3*d1*[2  0]  + a2*b1*c3*d1*[2  0]  + a3*b1*c3*d1*[2  0]
              [0  2]                [0  2]                [0  2]

+ a1*b2*c3*d1*[2  0]  + a2*b2*c3*d1*[2  0]  + a3*b2*c3*d1*[2  0]
              [0  2]                [0  2]                [0  2]

+ a1*b1*c4*d1*[-12   0]  + a2*b1*c4*d1*[-12   0]  + a3*b1*c4*d1*[-12   0]
              [0 -12]                 [0 -12]                 [0 -12]

+ a1*b2*c4*d1*[-12   0]  + a2*b2*c4*d1*[-12   0]  + a3*b2*c4*d1*[-12   0]
              [0 -12]                 [0 -12]                 [0 -12]

+ a1*b1*c1*d2*[9  0]  + a2*b1*c1*d2*[9  0]  + a3*b1*c1*d2*[9  0]
              [0  9]                [0  9]                [0  9]

+ a1*b2*c1*d2*[9  0]  + a2*b2*c1*d2*[9  0]  + a3*b2*c1*d2*[9  0]
              [0  9]                [0  9]                [0  9]

+ a1*b1*c2*d2*[-5  0]  + a2*b1*c2*d2*[-5  0]  + a3*b1*c2*d2*[-5  0]
              [0 -5]                 [0 -5]                 [0 -5]

+ a1*b2*c2*d2*[-5  0]  + a2*b2*c2*d2*[-5  0]  + a3*b2*c2*d2*[-5  0]
              [0 -5]                 [0 -5]                 [0 -5]

+ a1*b1*c3*d2*[-3  0]  + a2*b1*c3*d2*[-3  0]  + a3*b1*c3*d2*[-3  0]
              [0 -3]                 [0 -3]                 [0 -3]

+ a1*b2*c3*d2*[-3  0]  + a2*b2*c3*d2*[-3  0]  + a3*b2*c3*d2*[-3  0]
              [0 -3]                 [0 -3]                 [0 -3]

+ a1*b1*c4*d2*[-17   0]  + a2*b1*c4*d2*[-17   0]  + a3*b1*c4*d2*[-17   0]
              [0 -17]                 [0 -17]                 [0 -17]

+ a1*b2*c4*d2*[-17   0]  + a2*b2*c4*d2*[-17   0]  + a3*b2*c4*d2*[-17   0]
              [0 -17]                 [0 -17]                 [0 -17]
```

If the polynomial is defined from a matrix polynomially dependent on bounded parameters, then the derivative can be obtained using the syntax

```
>> dotpoly = diff(poly,labelout,bounds,dotbounds)
```

The input `bounds` informs the bounds of each parameter, and the bounds of their variation rates are given by the vector `dotbounds`.

Example 21

Suppose that one needs to calculate the derivative of the polynomial `poly`, depending on parameters θ_1, θ_2 and θ_3. The bounds and variation rates are given by

$$-2 \leq \theta_1(t) \leq 4, \quad 3 \leq \theta_2(t) \leq 6, \quad 1 \leq \theta_3(t) \leq 8,$$

$$-1 \leq \dot{\theta}_1(t) \leq 5, \quad -3 \leq \dot{\theta}_2(t) \leq 2, \quad -7 \leq \dot{\theta}_3(t) \leq 3.$$

The derivative of such polynomial can be calculated using the command

```
>> dotpoly = diff(poly,labelout,[-2 4; 3 6; 1 8],[-1 5; -3 2; -7 3]);
```

```
%Example 21
A{1} = {[0 0 0],eye(2)};
A{2} = {[1 0 0],2*eye(2)};
A{3} = {[1 1 0],3*eye(2)};
A{4} = {[0 2 0],4*eye(2)};
A{5} = {[1 1 1],5*eye(2)};
A{6} = {[0 0 3],6*eye(2)};
poly = rolmipvar(A,'A',[-2 4; 3 6; 1 8]);
dotpoly = diff(poly,'dpoly',[-2 4; 3 6; 1 8],[-1 5; -3 2; -7 3])
```

Command PARTIAL

The command `partial` computes the partial derivatives of a polynomial variable with respect to the simplex parameters. The syntax of `partial` is

```
>> dpoly = partial(poly,labelout,simpnum)
```

where `poly` is the input polynomial, `labelout` is the label of the resultant polynomial, `simpnum` is the index of the simplex domain in which respect the polynomial is differentiated, and `dpoly` is a cell structure containing the computed gradient vector.

For instance, suppose that $P(\alpha, \beta)$ is a polynomial with α in a simplex with three vertices, and β in a simplex with two vertices. The command

```
>> dpoly = partial(poly,'dPda',1);
```

computes the gradient vector

$$\frac{\partial P(\alpha,\beta)}{\partial \alpha} = \left[\frac{\partial P(\alpha,\beta)}{\partial \alpha_1} \quad \frac{\partial P(\alpha,\beta)}{\partial \alpha_2} \quad \frac{\partial P(\alpha,\beta)}{\partial \alpha_3}\right].$$

Similarly, the command

```
>> dpoly = partial(poly,'dPdb',2);
```

yields

$$\frac{\partial P(\alpha,\beta)}{\partial \beta} = \left[\frac{\partial P(\alpha,\beta)}{\partial \beta_1} \quad \frac{\partial P(\alpha,\beta)}{\partial \beta_2}\right].$$

If the variable `simpnum` is not informed, then the partial derivative is calculated over the first simplex.

Example 22

Consider the polynomial $A(\alpha)$ given by

$$A(\alpha) = \alpha_1^2 A_1 + \alpha_1\alpha_2 A_2 + \alpha_2^2 A_3,$$

defined as the variable `poly`. The partial derivative $\dfrac{\partial A(\alpha)}{\partial \alpha}$ can be computed through the command `>> dpoly = partial(poly,'dAda');`

```
>> dotbounds{1} = [-1 2; -3 4; -8 6];
>> dotbounds{2} = [-4 1; -6 9];
>> dotpoly = diff(poly,labelout,dotbounds);
```

```
%Example 22
A{1} = {[2 0],eye(2)};
A{2} = {[1 1],3*eye(2)};
A{3} = {[0 2],5*eye(2)};

poly = rolmipvar(A,'A',2,2);
dpoly = partial(poly,'dAda')
```

```
>> dpoly
dpoly =

    [2 rolmipvar]    [2 rolmipvar]

>> dpoly{1}
Label: dAda
Vertices: [2]
Degrees:  [1]

a1*[2  0]  + a2*[3  0]
   [0  2]       [0  3]

>> dpoly{2}
Label: dAda
Vertices: [2]
Degrees:  [1]

a1*[3  0]  + a2*[10   0]
   [0  3]       [0  10]
```

Command DISCSHIFT

The stability analysis for discrete-time LPV systems can be performed using procedures similar to the ones used in the continuous-time case. For instance, the discrete-time LPV system

$$x(k+1) = A(\alpha(k))x(k), \quad \alpha(k) \in \Delta_N$$

is asymptotically stable if the exists a symmetric positive definite matrix $P(\alpha(k))$ satisfying

$$A(\alpha(k))' P(\alpha(k+1)) A(\alpha(k)) - P(\alpha(k)) < 0.$$

Therefore, it is necessary to define not only $P(\alpha(k))$, but also the respective time-shifted matrix $P(\alpha(k+1))$.

Suppose that $P(\alpha(k))$ is defined as

$$P(\alpha(k)) = \sum_{i=1}^{N} \alpha_i(k) P_i.$$

Consequently,

$$P(\alpha(k+1)) = \sum_{i=1}^{N} \alpha_i(k+1) P_i.$$

If the variation rates of the parameters $\alpha(k)$ are bounded, being such bounds previously known and given by

$$\underline{d}_i \leq \alpha_i(k+1) - \alpha_i(k) \leq \overline{d}_i, \quad 0 \in [\underline{d}_i, \overline{d}_i], \quad i = 1, \ldots, N,$$

then one may define a polytope to represent all the possible values of $\alpha(k)$ and $\alpha(k+1)$, resulting in

$$\begin{bmatrix} P(\alpha(k)) \\ P(\alpha(k+1)) \end{bmatrix} = \sum_{i=1}^{N} \sum_{\ell=1}^{M} \beta_\ell h_i^\ell \begin{bmatrix} P_i \\ P_i \end{bmatrix}, \quad \beta \in \Delta_M$$

where h^ℓ are the vertices of the polytope. The procedure is generalized to deal with an arbitrary number of time-instants η ahead, that is, for $P(\alpha(k+\eta))$.

Time-shifted matrices can be automatically computed using the command `discshift`, whose syntax is

```
>> polyout = discshift(polyin,eta,bounds,targetin,targetout);
```

The input polynomial is described by `polyin`, `eta` is an integer corresponding to the maximum number of shifts needed, the bounds $[\underline{d}_i, \overline{d}_i]$ are given by `bounds`, and the inputs `targetin` and `targetout` correspond, respectively, to the simplex that is shifted and to the index of the new simplex domain. The Multi-Parametric Toolbox (MPT) [10] is necessary to calculate the vectors h^ℓ and, if such toolbox is not installed, an error will occur. The output `polyout` is a cell structure with `eta`+1 positions containing all the time-shifted matrices up to `eta`, represented in the new simplex domain. If `bounds` is not informed then the parameters are supposed to vary arbitrarily ($\overline{d}_i = -\underline{d}_i = 1$, $i = 1, \ldots, N$), and if `targetout` is not given then the next simplex domain available is used to represent the output polynomials. If the variable is defined only over one simplex, then `targetin` does not need to be informed.

Example 23

Consider the matrix polynomial $A(\alpha(k))$ given by

$$A(\alpha(k)) = \alpha_1(k) \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \alpha_2(k) \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix},$$

with

$$|\alpha_1(k+1) - \alpha_1(k)| \leq 0.4, \quad |\alpha_2(k+1) - \alpha_2(k)| \leq 0.6.$$

The polynomials $A(\alpha(k))$, $A(\alpha(k+1))$ and $A(\alpha(k+2))$, represented in the same simplex domain $\beta \in \Delta_M$, can be obtained by the command

```
>> polyout = discshift(polyin,2,[-0.4 0.4; -0.6 0.6]);
```

The output `polyout{1}`, `polyout{2}` and `polyout{3}` contain, respectively, the resultant polynomials $A(\alpha(k))$, $A(\alpha(k+1))$ and $A(\alpha(k+2))$. The utilization of the `discshift` command is detailed in the following.

```
%Example 23
A{1} = eye(2);
A{2} = 3*eye(2);

polyA = rolmipvar(A,'A',2,1);
polyout = discshift(polyA,2,[-0.4 0.4; -0.6 0.6]);
```

```
>> polyout
polyout =

    [2 rolmipvar]    [2 rolmipvar]    [2 rolmipvar]

>> polyout{1}
Label: A
Vertices: [2  14]
Degrees:  [0  1]

b1*[3  0]  + b2*[2.2    0]  + b3*[2.6     0]  + b4*[1.8    0]
   [0  3]       [0    2.2]       [0     2.6]       [0    1.8]

+ b5*[1.8    0]  + b6*[1  0] + b7*[1  0]  + b8*[1  0]
     [0    1.8]       [0  1]      [0  1]       [0  1]

+ b9*[1  0]  + b10*[3  0]  + b11*[1.4     0]  + b12*[2.2     0]
     [0  1]        [0  3]        [0     1.4]        [0     2.2]

+ b13*[3    0]  + b14*[3  0]
      [0    3]        [0  3]

>> polyout{2}
Label: A(k+1)
Vertices: [2  14]
Degrees:  [0  1]

b1*[3  0]  + b2*[3  0]  + b3*[1.8    0]  + b4*[1  0] + b5*[1  0]
   [0  3]       [0  3]       [0    1.8]       [0  1]      [0  1]

+ b6*[1  0]  + b7*[1.8    0] + b8*[1  0]  + b9*[1.8    0] + b10*[2.2    0]
     [0  1]       [0    1.8]      [0  1]       [0    1.8]       [0    2.2]

+ b11*[2.2    0]  + b12*[3  0] + b13*[3  0]  + b14*[2.2     0]
      [0  2.2]          [0  3]       [0  3]        [0     2.2]

>> polyout{3}
Label: A(k+2)
Vertices: [2  14]
Degrees:  [0  1]

b1*[2.2    0]  + b2*[2.2    0]  + b3*[1  0] + b4*[1.8   0] + b5*[1  0]
   [0    2.2]       [0    2.2]       [0  1]      [0   1.8]      [0  1]

+ b6*[1.8     0] + b7*[2.6    0]  + b8*[1  0] + b9*[1  0] + b10*[1.4    0]
     [0     1.8]      [0    2.6]       [0  1]      [0  1]       [0    1.4]

+ b11*[3  0]  + b12*[3  0]  + b13*[3  0]  + b14*[3  0]
      [0  3]        [0  3]        [0  3]        [0  3]
```

Example 24

Consider now the matrix

$$A(\alpha(k),\beta(k)) = \alpha_1(k)\beta_1(k)\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \alpha_1(k)\beta_2(k)\begin{bmatrix}3 & 0\\ 0 & 3\end{bmatrix} + \alpha_2(k)\beta_1(k)\begin{bmatrix}5 & 0\\ 0 & 5\end{bmatrix} + \alpha_2(k)\beta_2(k)\begin{bmatrix}7 & 0\\ 0 & 7\end{bmatrix}.$$

Since the shift operation is performed over one simplex domain, the polynomial $A(\alpha(k+1),\beta(k+1))$ can be computed by applying the `discshift` command first over one simplex, and then using the command again on the resulting polynomial, as detailed in the following commands.

```
%Example 24
A{1} = {[1 0],[1 0],eye(2)};
A{2} = {[1 0],[0 1],3*eye(2)};
A{3} = {[0 1],[1 0],5*eye(2)};
A{4} = {[0 1],[0 1],7*eye(2)};
polyA = rolmipvar(A,'A',[2 2],[1 1]);

%Generates A(\alpha(k+1),\beta(k))
polyaux = discshift(polyA,1,[-0.4 0.4; -0.6 0.6],1,3);

%Applies discshift on A(\alpha(k+1),\beta(k)) to generate
%A(\alpha(k+1),\beta(k+1))
polyout = discshift(polyaux{1},1,[-0.4 0.4; -0.6 0.6],2,4);
polyout{1}
```

```
>> polyout{1}
Label: A
Vertices: [2  2  6  6]
Degrees:  [0  0  1  1]

c1*d1*[7   0]  + c2*d1*[3  0]  + c3*d1*[3  0]  + c4*d1*[4.6     0]
      [0   7]          [0  3]          [0  3]          [0     4.6]

+ c5*d1*[5.4   0]  + c6*d1*[7   0]  + c1*d2*[5  0]  + c2*d2*[1  0]
        [0   5.4]          [0   7]          [0  5]          [0  1]

+ c3*d2*[1  0]  + c4*d2*[2.6     0]  + c5*d2*[3.4     0] + c6*d2*[5  0]
        [0  1]          [0     2.6]          [0     3.4]         [0  5]

+ c1*d3*[5  0]  + c2*d3*[1  0]  + c3*d3*[1  0]  + c4*d3*[2.6    0]
        [0  5]          [0  1]          [0  1]          [0    2.6]

+ c5*d3*[3.4    0]  + c6*d3*[5  0]  + c1*d4*[5.8    0] + c2*d4*[1.8     0]
        [0    3.4]          [0  5]          [0    5.8]         [0     1.8]

+ c3*d4*[1.8   0]  + c4*d4*[3.4   0] + c5*d4*[4.2    0] + c6*d4*[5.8    0]
        [0   1.8]          [0   3.4]         [0    4.2]         [0    5.8]

+ c1*d5*[6.2   0] + c2*d5*[2.2   0]  + c3*d5*[2.2   0]  + c4*d5*[3.8    0]
        [0   6.2]         [0   2.2]          [0   2.2]          [0    3.8]

+ c5*d5*[4.6     0]  + c6*d5*[6.2    0]  + c1*d6*[7  0]  + c2*d6*[3  0]
        [0     4.6]          [0    6.2]          [0  7]          [0  3]

+ c3*d6*[3  0]  + c4*d6*[4.6    0]  + c5*d6*[5.4    0]  + c6*d6*[7  0]
        [0  3]          [0    4.6]          [0    5.4]          [0  7]
```

Command TEXIFY

The command `texify`, with syntax given by

```
>> out = texify(poly,option,var1,...,varN,simplexnames);
```

returns the LaTeX code of the input polynomial `poly`. The argument `option` is a string that informs if the desired output format presents an explicit dependence of the polynomial on the simplexes (`'explicit'`), only the polynomial variables without showing their dependence on the simplexes (`'implicit'`), or the polynomial structure showing each monomial (`'polynomial'`). The arguments `var1,...,varN` are the variables that are used in the expression to be output; it is necessary to inform separately each variable in such a way that the procedure can assess the information needed. Finally, the input `simplexnames` is a cell array of strings containing the desired names for each simplex, already in LaTeX format. If only one string is informed instead of a cell array of strings, then it is supposed that the user intends to represent all the

simplexes using one single (multi-simplex) variable. If `simplexnames` is not informed, then standard names are used. The name of the polynomial variables is equal to the `label` informed in their definition.

Example 25

Consider the polynomial $A(\alpha, \beta)$, being both simplexes of degree 1 and 2 vertices and defined by the variable `Ai`, and the polynomial $P(\alpha)$, of degree 1 and defined by the variable `Pi`. The command

```
>> out = texify(Ai'*Pi + Pi*Ai,'explicit',Ai,Pi,{'\alpha','\beta'});
```

produces the output

```
out =

A(\alpha,\beta)'P(\alpha)+P(\alpha)A(\alpha,\beta)
```

On the other hand, the command

```
>> out = texify(Ai'*Pi + Pi*Ai,'explicit',Ai,Pi,'\alpha')
```

returns

```
out =

A(\alpha)'P(\alpha)+P(\alpha)A(\alpha)
```

The command

```
>> out = texify(Ai'*Pi + Pi*Ai,'implicit',Ai,Pi)
```

yields

```
out =

A'P+PA
```

Finally, the command

```
>> out = texify(Ai'*Pi + Pi*Ai,'polynomial',Ai,Pi,{'\alpha','\beta'})
```

returns the LaTeX code

```
out =

\alpha_{1}^{2}\beta_{1}(A_{1}'P_{1}+P_{1}A_{1})
 + \alpha_{1}\alpha_{2}\beta_{1}(A_{1}'P_{2}+A_{2}'P_{1}+P_{1}A_{2}+P_{2}A_{1})
 + \alpha_{2}^{2}\beta_{1}(A_{2}'P_{2}+P_{2}A_{2})
 + \alpha_{1}^{2}\beta_{2}(A_{3}'P_{1}+P_{1}A_{3})
 + \alpha_{1}\alpha_{2}\beta_{2}(A_{3}'P_{2}+A_{4}'P_{1}+P_{1}A_{4}+P_{2}A_{3})
 + \alpha_{2}^{2}\beta_{2}(A_{4}'P_{2}+P_{2}A_{4})
```

6 Composing matrices and LMIs

The construction of matrices of polynomials and LMIs using ROLMIP is done in an intuitive way, just as in YALMIP[11]. For example, suppose that the LMIs associated with the parameter-dependent LMI conditiion given in (3) are to be implemented, and suppose that the matrices $A(\alpha)$ and $P(\alpha)$ are defined by the `rolmipvar` variables `Ai` and `Pi`, respectively. The resulting set of LMIs is defined using

```
>> LMIs = [[Pi Ai'*Pi; Pi*Ai Pi] >= 0];
```

Any necessary degree homogenization is automatically performed by ROLMIP. Note that the last command consists of the definition of a semidefinite inequality, instead of the strictly positive condition shown in (3). Such change is applied to every LMI in the examples in order to avoid warning messages from YALMIP, which does not support strict inequalities and change them to non-strict ones. In this case, the user need to perform an *a posteriori* verification for the feasibility of the strict inequalities.

6.1 Pólya's relaxation

An interesting relaxation usually performed when dealing with LMI conditions over homogeneous polynomials is based on the Pólya's theorem for the case of positive polynomials with matrix-valued coefficients [20, 21, 14]. For a better understanding of Pólya's relaxation, suppose for example condition (3), and suppose that there exists a matrix $P(\alpha) = P(\alpha)^T > 0$ of degree g satisfying the condition. A simple way to verify that $P(\alpha)$ satisfies the condition is to check if each monomial of the homogeneous polynomial is positive definite. This is sufficient, but not necessary; in some cases, the polynomial is positive definite but not all the coefficients of the monomials are positive. In this case, according to Pólya's theorem, there exists an integer $d > 0$ such that the monomials of

$$\left(\sum_{j=1}^{M}\sum_{i=1}^{N}\alpha_i^{(j)}\right)^d \quad \begin{bmatrix} P(\alpha) & A(\alpha)'P(\alpha) \\ P(\alpha)A(\alpha) & P(\alpha) \end{bmatrix} \tag{13}$$

are definite positive, being $\alpha^{(j)}$ the j-th simplex on a multi-simplex domain.

The Pólya's relaxation technique is currently implemented on ROLMIP within the command `polya`, with syntax

```
>> polyout = polya(polyin,d);
```

being `d` the degree d of the relaxation.

Example 26

The implementation of the LMIs resultant from the matrix polynomial (13) being positive definite can be performed through the following sequence of commands.

```
%Example 26
A{1} = {[1 0],eye(2)};
A{2} = {[0 1],2*eye(2)};

polyA = rolmipvar(A,'A',2,1);
polyP = rolmipvar(2,2,'P','sym',2,1);
cond = [polyP polyA'*polyP; polyP*polyA polyP];
LMIs = [cond >= 0]
d = 3;
condpolya = polya(cond,d);
LMIspolya = [condpolya >= 0]
```

[11] In previous versions of ROLMIP, such construction is performed using the command `construct_lmi`; this command is still implemented for backwards compatibility, but it will be discontinued.

```
>> LMIs
+++++++++++++++++++++++++++++++
|   ID|              Constraint|
+++++++++++++++++++++++++++++++
|    #1|   Matrix inequality 4x4|
|    #2|   Matrix inequality 4x4|
|    #3|   Matrix inequality 4x4|
+++++++++++++++++++++++++++++++


>> LMIspolya
+++++++++++++++++++++++++++++++
|   ID|              Constraint|
+++++++++++++++++++++++++++++++
|    #1|   Matrix inequality 4x4|
|    #2|   Matrix inequality 4x4|
|    #3|   Matrix inequality 4x4|
|    #4|   Matrix inequality 4x4|
|    #5|   Matrix inequality 4x4|
|    #6|   Matrix inequality 4x4|
+++++++++++++++++++++++++++++++
```

Note the increased number of LMIs after the application of Pólya's relaxation.

In the following section, more involved examples based on uncertain linear systems problems, are presented for a better understanding of ROLMIP.

7 Solving Uncertain Linear Systems Problems with ROLMIP

7.1 Robust Stability Analysis

Consider the problem of robust stability analysis of the discrete-time linear system given in (1) with the following uncertain dynamic matrix

$$A(\alpha) = \alpha_1 \begin{bmatrix} 0.1 & 0.9 \\ 0 & 0.1 \end{bmatrix} + \alpha_2 \begin{bmatrix} 0.5 & 0 \\ 1 & 0.5 \end{bmatrix}$$

The first step is the the declaration of $A(\alpha)$ as a `rolmipvar`, which can be performed as (there are other possibilities)

```
>> A{1} = [0.1 0.9;0 0.1];
>> A{2} = [0.5 0;1 0.5];
>> A = rolmipvar(A,'A(\alpha)',2,1)
```

and the last command prints in screen:

```
Label: A(\alpha)
Vertices: [2]
Degrees:  [1]

a1*[0.1         0.9]  + a2*[0.5           0]
   [0         0.1]         [1         0.5]
```

The next step is to choose the structure for the Lyapunov matrix. For instance, consider an affine (degree one) polynomial dependence on α, as in (4). In this case the declaration of $P(\alpha)$ is done through the command

```
>> P = rolmipvar(2,2,'P(\alpha)',2,1)
```

echoing in screen

```
Linear matrix variable 2x2 (symmetric, real, 3 variables)
Label: P(\alpha)
Vertices: [2]
Degrees:  [1]
```

As shown in last section, the programming of the robust stability condition is very easy (the echo in screen, produced by YALMIP, is also shown):

```
>> LMIs = [[P A'*P; P*A P] >= 0]
```

```
+++++++++++++++++++++++++++++++
|   ID|              Constraint|
+++++++++++++++++++++++++++++++
|   #1|   Matrix inequality 4x4|
|   #2|   Matrix inequality 4x4|
|   #3|   Matrix inequality 4x4|
+++++++++++++++++++++++++++++++
```

Note that the number of LMIs (three) agrees with the number of monomials of the expression given in (6) when $N = 2$. Actually, each monomial gives rise to one LMI. After solving the set of LMIs (using SeDuMi) with the command:

```
>> optimize(LMIs,[],sdpsettings('verbose',0,'solver','sedumi'));
```

and checking the status of the constraints after the optimization, one has

```
>> checkset(LMIs)

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   ID|          Constraint|   Primal residual|   Dual residual|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   #1|   Matrix inequality|           0.17957|      1.9148e-14|
|   #2|   Matrix inequality|           0.12168|       3.058e-14|
|   #3|   Matrix inequality|          0.056108|      2.2844e-14|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

As the primal residuals are positive, the LMIs are strictly feasible, assuring the robust stability of $A(\alpha)$. Moreover, the Lyapunov matrix that certifies the robust stability can be inspected using the command:

```
>> double(P)
Label: P(\alpha)
Vertices: [2]
Degrees:  [1]

a1*[0.56664   0.0039574]    + a2*[1.0987    0.062489]
   [0.0039574     0.87634]       [0.062489     0.45827]
```

It is worth to emphasize that, depending on the solver used or on the MATLAB version, different values for $P(\alpha)$ can be obtained. A script with the implementation of the present section can be found into the folder `manual_examples`, with the name `example_Section_7_1.m`.

7.2 Guaranteed $\mathcal{H}_\infty$ Cost Computation

Consider the continuous-time uncertain time-invariant system given by

$$\left\{ \begin{array}{rl} \dot{x}(t) = & A(\alpha)x(t) + B(\alpha)w(t) \\ y(t) = & C(\alpha)x(t) + D(\alpha)w(t) \end{array} \right. \tag{14}$$

with $A(\alpha) \in \mathbb{R}^{n\times n}$, $B(\alpha) \in \mathbb{R}^{n\times r}$, $C(\alpha) \in \mathbb{R}^{q\times n}$ and $D(\alpha) \in \mathbb{R}^{q\times r}$. The transfer function from the disturbance input w to the output y, for a fixed α, is given by

$$H(s,\alpha) = C(\alpha)\big(s\mathrm{I} - A(\alpha)\big)^{-1}B(\alpha) + D(\alpha)$$

The bounded real lemma assures the Hurwitz stability of $A(\alpha)$ (*i.e.*, all eigenvalues have negative real part) for all $\alpha \in \Delta_N$ and a bound γ to the $\mathcal{H}_\infty$ norm of the transfer function from w to y. It can be formulated as follows [4].

Lemma 1
Matrix $A(\alpha)$ is Hurwitz and $\|H(s,\alpha)\|_\infty < \gamma$ for all $\alpha \in \Delta_N$ if there exists a symmetric positive definite matrix $P(\alpha) = P(\alpha)' > 0$ such that

$$\begin{bmatrix} A(\alpha)'P(\alpha) + P(\alpha)A(\alpha) + C(\alpha)'C(\alpha) & \star \\ B(\alpha)'P(\alpha) + D(\alpha)'C(\alpha) & D(\alpha)'D(\alpha) - \gamma^2\mathbf{I} \end{bmatrix} < 0, \quad \forall \alpha \in \Delta_N \tag{15}$$

In this example, consider a fourth-order mass-spring system, also investigated in [14], whose matrices are given by

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{2}{m_1} & \frac{1}{m_1} & -\frac{c_0}{m_1} & 0 \\ \frac{1}{m_2} & -\frac{1}{m_2} & 0 & -\frac{c_0}{m_2} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_1} \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}, \quad D = 0,$$

with

$$0.5 \le m_1 \le 1.5, \quad 0.75 \le m_2 \le 1.25, \quad 1 \le c_0 \le 2.$$

Setting $\theta_1 = 1/m_1$, $\theta_2 = 1/m_2$ and $\theta_3 = c_0$, one has

$$2/3 \le \theta_1 \le 2, \quad 0.8 \le \theta_2 \le 4/3, \quad 1 \le \theta_3 \le 3,$$

and the matrices can be rewritten as

$$A(\theta) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \theta_1 \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \theta_1\theta_3 \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \theta_2 \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} + \theta_2\theta_3 \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix},$$

$$B(\theta) = \theta_1 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad C(\theta) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}, \quad D(\theta) = 0.$$

The implementation of LMI relaxations that search for a feasible solution while minimizing $\mu = \gamma^2$ is given in the sequence[12].

Algorithm 1

```
clear;
degP = 1;
A{1} = {[0 0 0],[0 0 1 0; 0 0 0 1; 0 0 0 0; 0 0 0 0]};
A{2} = {[1 0 0],[0 0 0 0; 0 0 0 0; -2 1 0 0; 0 0 0 0]};
A{3} = {[1 0 1],[0 0 0 0; 0 0 0 0; 0 0 -1 0; 0 0 0 0]};
A{4} = {[0 1 0],[0 0 0 0; 0 0 0 0; 0 0 0 0; 1 -1 0 0]};
A{5} = {[0 1 1],[0 0 0 0; 0 0 0 0; 0 0 0 0; 0 0 0 -1]};
Ai = rolmipvar(A,'A',[2/3 2; 0.8 4/3; 1 3]);
B{1} = {[1 0 0],[0;0;1;0]};
Bi = rolmipvar(B,'B',[2/3 2]);
Ci = rolmipvar([0 1 0 0],'C',0,0);
Di = rolmipvar(0,'D',0,0);
Pi = rolmipvar(4,4,'P','sym',[2 2 2],[degP degP degP]);
mu = sdpvar(1,1);
T11 = Ai'*Pi + Pi*Ai + Ci'*Ci;
T21 = Bi'*Pi + Di'*Ci;
T22 = Di'*Di - mu*eye(1);
T = [T11 T21'; T21 T22];
LMIs = [Pi >= 0, T <= 0];
optimize(LMIs,mu);
gama = sqrt(double(mu));
```

[12]This code is available as `Example_5_1.m` in the installation file.

Remark 2
Note that, when defining variable `T22` in row 17, a `rolmipvar` (`Di`) is mixed with a `sdpvar` (`mu`). However, as `Di` appears first, the subtraction is performed by the `rolmipvar` class, that is prepared to deal with `sdpvar` variables. On the other hand, the equivalent declaration: `-mu*eye(1)+Di'*Di` would produce an error, since the `sdpvar` comes first in the expression.

As illustrated, the implementation of LMI conditions using the Robust LMI Parser is simple and straightforward. Moreover, the different sets of LMIs for larger degrees of the homogeneous polynomial variable $P(\alpha)$ can be readily obtained by simply changing `degP`. Those LMIs, that would demand complex and tedious manipulations without using the parser, provide clear improvements in the computation of the $\mathcal{H}_\infty$ guaranteed cost. Table 2 shows the values of $\gamma^* = \min \gamma$ subject to (15) obtained when the degree of the variable $P(\alpha)$ increases. With $g = 2$, γ^* reaches the worst case value of the $\mathcal{H}_\infty$ norm (computed through brute force).

Table 2: Values of $\gamma^* = \min \gamma$ obtained when varying the degree of the polynomial variable $P(\alpha)$.

degP	γ^*
0	2.8429
1	1.0540
2	1.0108
3	1.0108
4	1.0108
5	1.0108

In order to analyze the results obtained when applying the Pólya's relaxation, as described in Section 6.1, on Algorithm 7.2 for $d = 3$, it suffices to insert the following block of commands between lines 18 and 19:

```
d = 3;
t = polya(T,d);
```

The results obtained for the application of the latter addition on the code on the minimization of γ, subject to (15) with Pólya's relaxation, are shown on Table 3.

Table 3: Values of $\gamma^* = \min \gamma$ obtained when varying the degree of the polynomial variable $P(\alpha)$ for $d = 3$.

degP	γ^*
0	2.8429
1	1.0308
2	1.0108
3	1.0108
4	1.0108
5	1.0108

8 Conclusion

A computational package, named Robust LMI Parser, is presented in this manual. The main objective of the toolbox is to facilitate the task of programming LMIs that are sufficient conditions for robust LMIs, i.e., for parameter-dependent LMI conditions whose entries are algebraic manipulations of homogeneous polynomials of generic degree with parameters in the unit simplex or in known intervals. The toolbox is under constant evolution and some new features are to be implemented. Suggestions of improvements and bug reports are welcome and can be sent to the email `agulhari@utfpr.edu.br`.

References

[1] E. D. Andersen and K. D. Andersen. The MOSEK interior point optimizer for linear programming: An implementation of the homogeneous algorithm. In H. Frenk, K. Roos, T. Terlaky, and S. Zhang, editors, *High Performance Optimization*, volume 33 of *Applied Optimization*, pages 197–232. Springer US, 2000. `http://www.mosek.com`.

[2] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete & Computational Geometry*, 8(3):295–313, September 1992.

[3] P.-A. Bliman, R. C. L. F. Oliveira, V. F. Montagner, and P. L. D. Peres. Existence of homogeneous polynomial solutions for parameter-dependent linear matrix inequalities with parameters in the simplex. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 1486–1491, San Diego, CA, USA, December 2006.

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

[5] M. Chamanbaz, F. Dabbene, D. Peaucelle, and R. Tempo. R-RoMulOC: A unified tool for randomized and robust multiobjective control. In *Proceedings of the 8th IFAC Symposium on Robust Control Design (ROCOND 2015)*, volume 48, pages 144–149, Bratislava, July 2015.

[6] G. Chesi, A. Garulli, A. Tesi, and A. Vicino. Robust stability of time-varying polytopic systems via parameter-dependent homogeneous Lyapunov functions. *Automatica*, 43(2):309–316, February 2007.

[7] P. Gahinet, A. Nemirovskii, A. J. Laub, and M. Chilali. *LMI Control Toolbox User's Guide*. The Math Works, Natick, MA, 1995.

[8] J. C. Geromel and P. Colaneri. Robust stability of time varying polytopic systems. *Systems & Control Letters*, 55(1):81–85, January 2006.

[9] D. Henrion and J. B. Lasserre. GloptiPoly: Global optimization over polynomials with Matlab and SeDuMi. *ACM Transactions on Mathematical Software*, 29(2):165–194, June 2003.

[10] M. Herceg, M. Kvasnica, C. N. Jones, and M. Morari. Multi-Parametric Toolbox 3.0. In *Proceedings of the 2013 European Control Conference*, pages 502–510, Zurich, Switzerland, July 2013.

[11] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the 2004 IEEE International Symposium on Computer Aided Control Systems Design*, pages 284–289, Taipei, Taiwan, September 2004. `https://yalmip.github.io/`.

[12] R. C. L. F. Oliveira and P. L. D. Peres. LMI conditions for robust stability analysis based on polynomially parameter-dependent Lyapunov functions. *Systems & Control Letters*, 55(1):52–61, January 2006.

[13] R. C. L. F. Oliveira and P. L. D. Peres. Parameter-dependent LMIs in robust analysis: Characterization of homogeneous polynomially parameter-dependent solutions via LMI relaxations. *IEEE Transactions on Automatic Control*, 52(7):1334–1340, July 2007.

[14] R. C. L. F. Oliveira and P. L. D. Peres. A convex optimization procedure to compute $\mathcal{H}_2$ and $\mathcal{H}_\infty$ norms for uncertain linear systems in polytopic domains. *Optimal Control Applications and Methods*, 29(4):295–312, July/August 2008.

[15] D. Peaucelle and D. Arzelier. Robust multi-objective control toolbox. In *Proceedings of the 2006 IEEE International Symposium on Computer Aided Control Systems Design*, pages 1152–1157, Munich, Germany, 2006.

[16] D. Peaucelle, A. Tremba, D. Arzelier, A. Bortott, G.C. Calafiore, G. Chevarria, F. Dabbene, E. Gryazina, B.T. Polyak, P.S. Shcherbakov, M. Sevin, Ph. Spiesser, and R. Tempo. R-romuloc : Randomized and robust multi-objective control toolbox, January 2014.

[17] S. Prajna, A. Papachristodoulou, P. Seiler, and P. A. Parrilo. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2004. `http://www.cds.caltech.edu/sostools`.

[18] D. C. W. Ramos and P. L. D. Peres. A less conservative LMI condition for the robust stability of discrete-time uncertain systems. *Systems & Control Letters*, 43(5):371–378, August 2001.

[19] D. C. W. Ramos and P. L. D. Peres. An LMI condition for the robust stability of uncertain continuous-time linear systems. *IEEE Transactions on Automatic Control*, 47(4):675–678, April 2002.

[20] C. W. Scherer. Higher-order relaxations for robust LMI problems with verifications for exactness. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, pages 4652–4657, Maui, HI, USA, December 2003.

[21] C. W. Scherer. Relaxations for robust linear matrix inequality problems with verifications for exactness. *SIAM Journal on Matrix Analysis and Applications*, 27(2):365–395, June 2005.

[22] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11(1–4):625–653, 1999. `http://sedumi.ie.lehigh.edu/`.

[23] K. C. Toh, M. J. Todd, and R. Tütüncü. SDPT3 — A Matlab software package for semidefinite programming, Version 1.3. *Optimization Methods and Software*, 11(1):545–581, 1999. `https://github.com/SQLP/SDPT3`.

[24] A. Tremba, G. C. Calafiore, F. Dabbene, E. Gryazina, B. T. Polyak, P. S. Shcherbakov, and R. Tempo. RACT: Randomized Algorithms Control Toolbox for MATLAB. In *Proceedings 17th IFAC World Congress, Seoul*, volume 41, pages 390–395, 2008.