UNIVERSITÀ DEGLI STUDI DI CAGLIARI

Facoltà di Ingegneria

Dipartimento di Ingegneria Elettrica ed Elettronica

Delft Center for Systems and Control

PWA AND MMPS FUNCTIONS: EQUIVALENCES AND TRANSFORMATIONS

Corso di Laurea Specialistica in Ingegneria Elettronica

Tesi di Laurea di/Master's Thesis by

ANDREA FRAU

Relatori/Advisors:
Prof. Alessandro Giua
Prof. Bart De Schutter
Dr. Ir. Ton J.J. van den Boom

Anno Accademico
2006-2007

Alla mia famiglia,
a chi mi vuole bene
e a chi non beve...

Ringraziamenti

Vorrei innanzitutto ringraziare la TU Delft, che mi ha ospitato durante il soggiorno Erasmus tra Marzo e Settembre del 2007 e ha permesso lo svolgimento di questa tesi di laurea. Un grazie va soprattutto ai miei supervisors esteri Bart De Schutter e Ton van den Boom, che mi hanno supportato durante la realizzazione del seguente lavoro. Vorrei ringraziare inoltre il Professor Alessandro Giua, per il suo aiuto e per avermi concesso l'opportunità di svolgere questa tesi all'estero.

Abstract

Hybrid systems are a mixture of interacting time-driven and event-driven dynamics. There are some classes of hybrid systems for which tractable control design techniques are available. Two of these subclasses are Piecewise Affine (PWA) and Max-Min-Plus-Scaling (MMPS) systems. The context of this project is the equivalence between continuous PWA systems and MMPS systems, in their discrete-time version. The goal of this thesis is to study connections between them, and to do this we will focus on the two kinds of functions that define these systems: continuous PWA functions and MMPS functions. In this project we will therefore investigate how these two types of functions are related, and we will develop algorithms to transform continuous PWA functions into MMPS functions and vice versa.

Sommario

I sistemi ibridi rappresentano un connubio tra sistemi ad avanzamento temporale e sistemi ad eventi discreti. Esistono alcune classi di sistemi ibridi per le quali delle tecniche efficaci di controllo sono disponibili. Due di queste classi sono i sistemi Piecewise Affine (PWA) e i sistemi Max-Min-Plus-Scaling (MMPS). Il contesto in cui si svolge questa tesi è l'equivalenza tra sistemi PWA continui e sistemi MMPS, nella loro versione a tempo discreto. L'obiettivo di questa tesi è lo studio dei legami tra queste due classi di sistemi, e a questo scopo, focalizzeremo la nostra attenzione sui due tipi di funzione che definiscono questi sistemi: funzioni PWA continue e funzioni MMPS. In questa tesi verranno quindi analizzate le relazioni tra questi due tipi di funzione, e verranno inoltre sviluppati degli algoritmi allo scopo di trasformare funzioni PWA continue in funzioni MMPS e viceversa.

Contents

Chapter 1

Introduction

In this chapter we present a description of hybrid systems and a short survey of some of their subclasses. We also see that these subclasses are equivalent under some assumptions. At the end of the chapter there is an overview of this thesis.

This introduction is based on [3, 6, 7, 11, 12].

1.1 Hybrid systems

An hybrid system is a mixture of interacting time-driven and event-driven dynamics. Some examples are manufacturing systems, traffic control systems, logistic systems, electrical networks, etc. In time-driven systems, the time is the independent variable: so, these systems can be described by using differential equations (if they are continuous-time system) or difference equations (if they are discrete-time systems). On the contrary, if we consider event-driven systems, the state changes only when an event occurs. The states of an event-driven system are often called modes. We can define an event as the start of an action (e.g. the semaphore light changing colour, the shifting of the gears in a car): it could be an input event, an external event, or also an event generated by time-driven dinamics (e.g. when the state reaches a particular region of state space). So, we can consider an hybrid system as a set of modes in which the system switches from a mode to another mode when a particular event occurs. Moreover, for each mode, the system is described by a set of differential or difference equations (in fact, for each mode, the system is a time-driven system). A graphical representation is shown in Figure 1.1.

Recently, these systems have received a lot of attention from control community, even if they have always been with us.

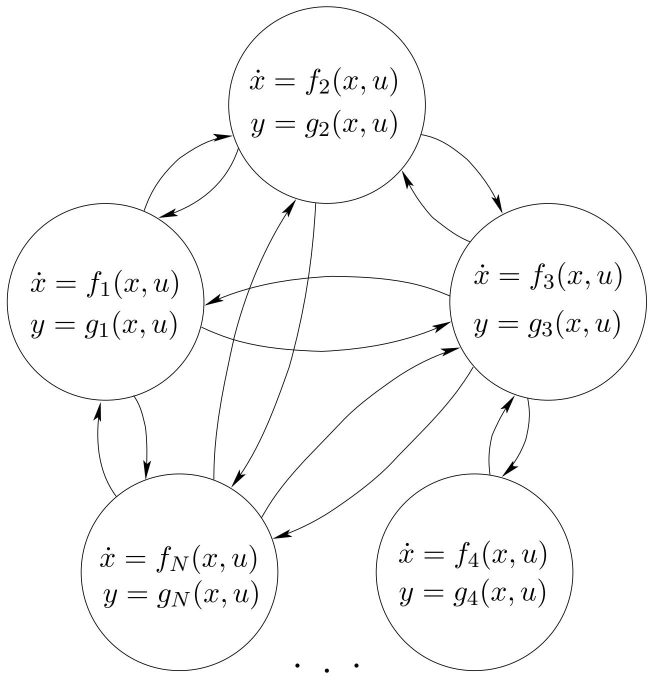

Figure 1.1: *Schematic representation of an hybrid system.*

1.2 Different subclasses of hybrid systems

Since there do not exist methods to analyze the general class of hybrid systems, several authors have focused on special subclasses of them. In this introduction, we will focus on discrete-time linear hybrid models only. If we consider this kind of systems, each subclass has some advantages over the others, but since these subclasses are equivalent (although some of the equivalences are obtained under additional assumptions on boundedness and well-posedness), we can transfer the advantages and the techniques utilized for each one, to any of the other equivalent ones.

1.2.1 Piecewise Affine (PWA) systems

PWA systems [16] are described by

$$\begin{aligned} x(k+1) &= A_i x(k) + B_i u(k) + f_i \\ y(k) &= C_i x(k) + D_i u(k) + g_i \end{aligned} \qquad \text{for} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \in \Omega_i \tag{1.1}$$

where the finite set of polyhedra $\{\Omega_i\}$ is a partition of input/state space, while the variables $x(k) \in \mathbb{R}^n$, $u(k) \in \mathbb{R}^m$ and $y(k) \in \mathbb{R}^p$ are respectively the states, the inputs and the outputs at time k.

A more exhaustive description of PWA systems will be given in Chapter 2.

The subclass of PWA systems is the most studied one, since they represent the "simplest" extension of linear system that can model hybrid phenomena. They can also easily approximate smooth nonlinear processes with arbitrary accuracy.

1.2.2 Max-Min-Plus-Scaling (MMPS) systems

MMPS functions [5] $f : \mathbb{R}^q \longrightarrow \mathbb{R}$ are defined by the recursive grammar

$$f(x) = x_i|\alpha|\max(f_k(x), f_l(x))|\min(f_k(x), f_l(x))|f_k(x) + f_l(x)|\beta f_k(x) \tag{1.2}$$

with $i \in \{1, \ldots, q\}$, $\alpha, \beta \in \mathbb{R}$ and where $f_k(x) : \mathbb{R}^q \longrightarrow \mathbb{R}$, $f_l(x) : \mathbb{R}^q \longrightarrow \mathbb{R}$ are again MMPS functions; the symbol | stands for "or".

A vector-valued function $f : \mathbb{R}^q \longrightarrow \mathbb{R}^r$ is MMPS if the above statement holds component-wise.

MMPS systems are described by

$$x(k+1) = \mathcal{M}_x(x(k), u(k)) \tag{1.3a}$$
$$y(k) = \mathcal{M}_y(x(k), u(k)) \tag{1.3b}$$

and in addition, if the system is constrained, also by

$$\mathcal{M}_c(x(k), u(k)) \leq c \tag{1.3c}$$

where $\mathcal{M}_x$, $\mathcal{M}_y$ and $\mathcal{M}_c$ are MMPS functions and the variables $x(k) \in \mathbb{R}^n$, $u(k) \in \mathbb{R}^m$ and $y(k) \in \mathbb{R}^p$ are respectively the states, the inputs and the outputs at time k.

The model (1.3) is a generalized framework that includes, e.g., max-plus linear systems, max-min-plus linear systems, max-plus polynomial systems, etc.

1.2.3 Mixed Logical Dynamical (MLD) Systems

MLD systems [2] are systems in which logic, dynamics and constraints are integrated. They are described by the following model

$$x(k+1) = Ax(k) + B_1u(k) + B_2\delta(k) + B_3z(k) \tag{1.4a}$$
$$y(k) = Cx(k) + D_1u(k) + D_2\delta(k) + D_3z(k) \tag{1.4b}$$
$$E_1x(k) + E_2u(k) + E_3\delta(k) + E_4z(k) \leq e_5 \tag{1.4c}$$

where $x(k) = \left[\, x_r^T(k)\, x_b^T(k) \,\right]^T$ with $x_r(k) \in \mathbb{R}^{n_r}$ and $x_b(k) \in \{0,1\}^{n_b}$ ($y(k)$ and $u(k)$ have a similar structure), and where $z(k) \in \mathbb{R}^{r_r}$ and $\delta(k) \in \{0,1\}^{r_b}$ are auxiliary variables.

In MLD systems, the logic is converted into binary variables in linear equalities, for obtaining a more mathematically tractable description.

MLD systems include linear hybrid systems, finite state machines, some classes of discrete event systems, etc.

1.2.4 Linear Complementarity (LC) systems

LC systems [17] are described by the equations

$$\begin{aligned} x(k+1) &= Ax(k) + B_1u(k) + B_2w(k) && (1.5a)\\ y(k) &= Cx(k) + D_1u(k) + D_2w(k) && (1.5b)\\ v(k) &= E_1x(k) + E_2u(k) + E_3w(k) + g_4 && (1.5c)\\ 0 \le v(k) &\perp w(k) \ge 0 && (1.5d) \end{aligned}$$

where $v(k)$, $w(k) \in \mathbb{R}^s$ and $v(k) \perp w(k)$ means that the two vectors are orthogonal, so $v^T(k)w(k) = 0$. $v(k)$ and $w(k)$ are called the complementarity variables.

LC systems are often used to describe constrained mechanical systems, electrical networks with ideal diodes or other dynamical systems with piecewise linear relations, variable structure systems, etc.

1.2.5 Extended Linear Complementarity (ELC) systems

ELC systems [4] are an extension of LC systems and they are given by the equations

$$\begin{aligned} x(k+1) &= Ax(k) + B_1u(k) + B_2d(k) && (1.6a)\\ y(k) &= Cx(k) + D_1u(k) + D_2d(k) && (1.6b)\\ & E_1x(k) + E_2u(k) + E_3d(k) \le g_4 && (1.6c)\\ & \sum_{i=1}^{p} \prod_{j\in\phi_i} (g_4 - E_1x(k) - E_2u(k) - E_3d(k))_j = 0 && (1.6d) \end{aligned}$$

in which $d(k) \in \mathbb{R}^r$ is an auxiliary variable. If we look at (1.6c) and at (1.6d), we can note that (1.6d) is equivalent to

$$\prod_{j\in\phi_i} (g_4 - E_1x(k) - E_2u(k) - E_3d(k))_j = 0 \qquad \text{for each}\, i \in 1,\dots,p \qquad (1.7)$$

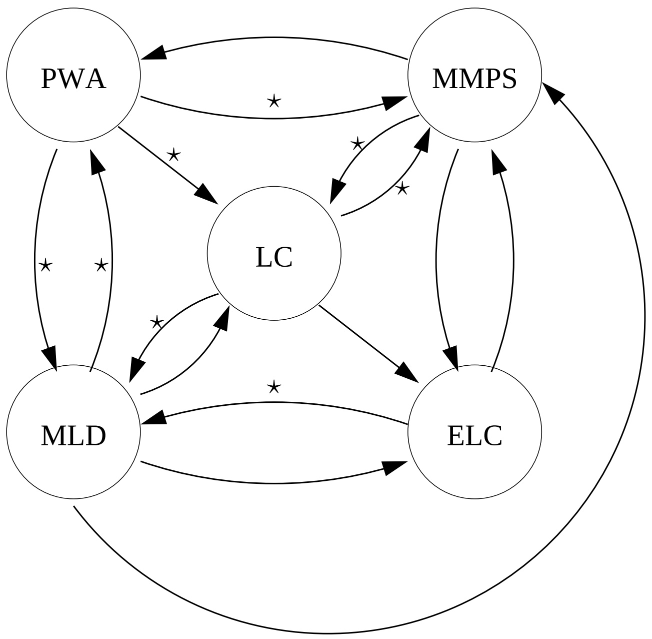

Figure 1.2: *Graphical representation of the links between some subclasses of hybrid systems. An arrow going from class A to class B means that A is a subset of B. Arrows with a star mean that additional assumptions are required.*

and therefore, we can see that (1.6c) can be considered as a system of linear inequalities where there are p groups of them, and for each group at least one inequality must hold with equality.

1.2.6 Equivalences

It can be proven that all subclasses of hybrid systems mentioned in this section are equivalent, even if some assumptions relate to well-posedness of the problem and to boundedness of inputs, states, outputs and auxiliary variables (or combinations of them) are required sometimes [11, 12]. These equivalences are shown in Figure 1.2. Fortunately, these assumptions are not restrictive, so we can choose the suitable model for a particular hybrid system and, if necessary, we can transforme it into the subclass for which the desired technique is known or more efficient.

1.3 Overview of the MSc Thesis

In this thesis we study in an thoroughly way two subclasses of discrete-time linear hybrid models: PWA and MMPS systems, in their discrete-time version. The goal of this thesis is to study connections between them, and to do it we focus on the two kinds of functions that define these systems: PWA and MMPS functions.

In Chapter 2 there is the background knowledge: basic definitions and properties for PWA and MMPS functions and an illustration of their equivalence (under some assumptions). Moreover, it will be shown how to represent PWA functions and MMPS functions in canonical form in MATLAB.

Chapter 3 deals with minimal realizations of PWA and MMPS functions.

In Chapter 4, we illustrate two strategies for the transformation from a continuous PWA function into an equivalent MMPS one: the Gorokhovik-Zorko strategy and the Ovchinnikov strategy. We also discuss about reduced realizations with Gorokhovik-Zorko strategy.

In Chapter 5 is presented a method for transforming a MMPS function in canonical form into the equivalent continuous PWA one.

Tests and results will be given in Chapter 6.

Chapter 7 contains the conclusions and considerations about this thesis and some suggestions for future research.

In this thesis, in order to prevent confusion, we have chosen to always explain new definitions (if not trivial) and to repeat the ones already given, when it is necessary for a better understanding.

Chapter 2

PWA and MMPS functions

In this chapter we thoroughly analyse two kinds of functions:

- piecewise affine (PWA) functions
- max-min-plus-scaling (MMPS) functions

By means of these functions we can define PWA and MMPS systems of course. We show properties of both functions, their equivalence, and a way for their representation in MATLAB.

2.1 Basic definitions and properties of PWA functions

As already said in Section 1.2.1, PWA systems are defined by partitioning the state/input space in a finite number of polyhedral regions and associating to each one an affine dynamic. Now, we give a precise definition for them.

Let us consider some definitions:

Definition 2.1. A polyhedron is a convex set $\mathcal{X} \subseteq \mathbb{R}^n$ given as an intersection of a finite number of half-spaces. Each half-space can be either closed (i.e $\{x \in \mathbb{R}^n : a^T x \leq b\}$ where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$) or open (i.e $\{x \in \mathbb{R}^n : a^T x < b\}$ where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$). A bounded polyhedron is also called polytope.

Definition 2.2. Given a polyhedron $\mathcal{X}$, then a polyhedral partition of $\mathcal{X}$ is a finite set of closed polyhedra $\{\mathcal{X}_i\}_{i \in \mathcal{I}}$ satisfying:

1. $\operatorname{int} \mathcal{X}_i \neq \emptyset$ for each $i \in \mathcal{I}$;

2. $\cup_{i \in \mathcal{I}} \mathcal{X}_i = \mathcal{X}$;

3. $\operatorname{int} \mathcal{X}_i \cap \operatorname{int} \mathcal{X}_j = \emptyset$ for each $i, j \in \mathcal{I}$, $i \neq j$.

in which $\operatorname{int} \mathcal{X}_i$ denotes the *interior* of $\mathcal{X}_i$.

Now, we are in the position to define a PWA function:

Definition 2.3. A function $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X} \subseteq \mathbb{R}^n$ is a polyhedron, is PWA if there exists a polyhedral partition $\{\mathcal{X}_i\}_{i \in \mathcal{I}}$ of $\mathcal{X}$ such that f is affine on each $\mathcal{X}_i$, i.e. $f(x) = \alpha_i^T x + \beta_i$ for all $x \in \mathcal{X}_i$ and $i \in \mathcal{I}$.

A vector-valued function $f : \mathcal{X} \to \mathbb{R}^k$ is PWA if the Definition 2.3 holds component-wise.

An example of a PWA function is the following:

Example 2.1.

$$f(x) = \begin{cases} f_1(x) = x + 1 & x \leq 3 \\ f_2(x) = 3 & 3 \leq x \leq 6 \\ f_3(x) = 0.5x + 2 & x \geq 6 \end{cases} \tag{2.1}$$

Remark 2.1. Strictly speaking, by definition 2.3 we are defining a relation and not a function. Indeed, if we look at the Example 2.1, we can see that the function $f(x)$ can have more than one output in the points on the boundaries between two regions (e.g., we can have both $f(3) = 4$ and $f(3) = 3$). Instead, we do not have this problem when we consider continuous PWA functions (see Section 2.1.1 for their definition), since we have a unique output for every $x \in \mathcal{X}$.

2.1.1 Continuous PWA functions

Let us consider a PWA function as given in Definition 2.3:

Definition 2.4. A PWA function $f : \mathcal{X} \longrightarrow \mathbb{R}^k$ is said to be continuous if it is continuous on any boundary between two regions.

A simple example of a continuous PWA function is the following (see Figure 2.1):

Example 2.2.

$$f(x) = \begin{cases} f_1(x) = x + 1 & x \leq 3 \\ f_2(x) = 4 & 3 \leq x \leq 6 \\ f_3(x) = 2x - 8 & x \geq 6 \end{cases} \tag{2.2}$$

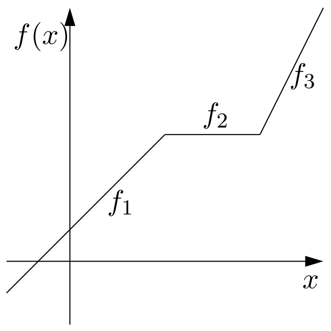

Figure 2.1: *Graphical representation of the function in Example 2.2.*

In Theorem 2.1 below, an alternative definition for continuous PWA functions is given. In order to give it, let us see some introductive definitions.

Definition 2.5. The epigraph of a function $f : \mathcal{X} \to \mathbb{R}$ is defined as:

$$\operatorname{epi} f := \{(x,t) \in \mathcal{X} \times \mathbb{R} : \, f(x) \leq t\} \tag{2.3}$$

Definition 2.6. The hypograph of a function $f : \mathcal{X} \to \mathbb{R}$ is defined as:

$$\operatorname{hyp} f := \{(x,t) \in \mathcal{X} \times \mathbb{R} : \, f(x) \geq t\} \tag{2.4}$$

Theorem 2.1. *[10] A function $f : \mathcal{X} \to \mathbb{R}^k$ is a continuous PWA function if both the epigraph and the hypograph of each scalar component are a finite union of closed polyhedra.*

2.2 Basic definitions and properties of MMPS functions

As already seen in Section 1.2.2, MMPS functions are defined by the operations maximization, minimization, addition and scalar multiplication. Let us recall the definition of an MMPS function:

Definition 2.7. MMPS functions $f : \mathbb{R}^q \longrightarrow \mathbb{R}$ are defined by the recursive grammar:

$$f(x) = x_i | \alpha | \max(f_k(x), f_l(x)) | \min(f_k(x), f_l(x)) | f_k(x) + f_l(x) | \beta f_k(x) \tag{2.5}$$

where $i \in \{1, \ldots, q\}$, $\alpha, \beta \in \mathbb{R}$ and where $f_k(x) : \mathbb{R}^q \longrightarrow \mathbb{R}$, $f_l(x) : \mathbb{R}^q \longrightarrow \mathbb{R}$ are again MMPS functions; the symbol | stands for "or". A vector valued function $f : \mathbb{R}^q \longrightarrow \mathbb{R}^r$ is MMPS if the above statement holds component-wise.

Example 2.3. Examples of MMPS functions are:

$$\begin{aligned} f(x) &= \min(2x_2 + 3\max(\min(3, 2x_1 + 3x_3), x_1 + \max(x_2, 0))) \\ g(x) &= 2x_1 + 3x_2 + 7 + 6\min(x_1, x_3) \\ h(x) &= \max(\min(1, -x_1 + 2), x_1 - 4) \qquad \text{(see Figure 2.3)} \end{aligned}$$

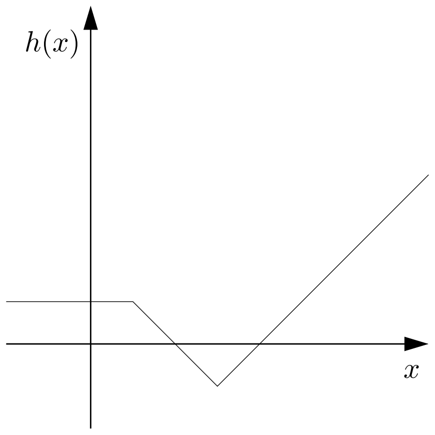

Figure 2.2: *Graphical representation of the function $h(x)$ in Example 2.3.*

Let us see now some properties of MMPS functions [6]:

1. The addition is distributive over both minimum and maximum:

$$\min(f_1, f_2) + f_3 = \min(f_1 + f_3, f_2 + f_3) \tag{2.6a}$$
$$\max(f_1, f_2) + f_3 = \max(f_1 + f_3, f_2 + f_3) \tag{2.6b}$$

2. Let $\beta \in \mathbb{R}^+$. Then we can write:

$$\beta \min(f_1, f_2) = \min(\beta f_1, \beta f_2) \tag{2.7a}$$
$$\beta \max(f_1, f_2) = \max(\beta f_1, \beta f_2) \tag{2.7b}$$

If the constant is negative we have to convert min into max and max into min (let us consider again $\beta \in \mathbb{R}^+$):

$$-\beta \min(f_1, f_2) = \max(-\beta f_1, -\beta f_2) \tag{2.8a}$$
$$-\beta \max(f_1, f_2) = \min(-\beta f_1, -\beta f_2) \tag{2.8b}$$

3. Nestings of min operations can always be simplified in the following way (the same holds for max operations):

$$\begin{aligned}&\min(\ldots(\min(\min(f_1, f_2), f_3), f_4), \ldots, f_k)\\ &\quad = \min(f_1, \ldots, f_k)\end{aligned} \tag{2.9}$$

4. If an expression is in the form $\min() + \ldots + \min()$, we can always reduce it to $\min()$ (the same holds for max operations):

$$\begin{aligned}&\min(f_{11}, \ldots, f_{k_1}) + \min(f_{21}, \ldots, f_{2k_2}) + \ldots + \min(f_{l1}, \ldots, f_{lk_l})\\ &\quad = \min(f_{11} + f_{21} + \ldots + f_{l1},\\ &\qquad f_{11} + f_{21} + \ldots + f_{l2}, \ldots, f_{1k_1} + f_{2k_2} + \ldots + f_{lk_l})\end{aligned} \tag{2.10}$$

and so, in a compact notation:

$$\sum_{i=1}^{l} \min_{j=1,\ldots,k_i}(f_{ij}) = \min_{\substack{(j_1,\ldots,j_l)\in \\ \{1,\ldots,k_1\}\times\ldots \\ \times\{1,\ldots,k_l\}}} (\sum_{i=1}^{l} f_{ij_i})$$

5. Minimization is distributive with respect to maximization and vice versa. Let us see a simple example:

$$\begin{aligned} &\min(\max(f_1,f_2),\max(f_3,f_4)) \\ &\qquad = \max(\min(f_1,f_3),\min(f_1,f_4),\min(f_2,f_3),\min(f_2,f_4)) \\ &\max(\min(f_1,f_2),\min(f_3,f_4)) \\ &\qquad = \min(\max(f_1,f_3),\max(f_1,f_4),\max(f_2,f_3),\max(f_2,f_4)) \end{aligned} \tag{2.11}$$

2.2.1 Canonical forms of MMPS functions

A canonical form is a structure where all expression can be written into. It is very important to write every MMPS expression in a standard form with as few nestings as possible, since in this way it becomes more easy, e.g., translating an MMPS function into the other subclasses of hybrid systems, reducing the computational complexity when we use them, etc.

Let us consider now the following definitions:

Definition 2.8. [6] An MMPS function is in conjunctive form if it is written as

$$\min_{j\in 1,\ldots,l}(\max_{i\in I_j}(a_i^T x + b_i)) \tag{2.12}$$

or in disjunctive form if it is written as

$$\max_{j\in 1,\ldots,l}(\min_{i\in I_j}(a_i^T x + b_i)) \tag{2.13}$$

where $I_1,\ldots,I_l \subseteq \{1,\ldots,N\}$ are index sets and there are N possible components in the form $a_i^T x + b_i$.

Definition 2.9. A level-n expression (with $n \in \mathbb{N}, n \neq 0$) is an expression with $n-1$ nestings. The number n equals the maximum number of min and max operations encountered in each MMPS expression before arriving at an argument of the form $a_i^T x + b_i$.

Example 2.4. $\min(2x+1, 3x+4)+6$ is a level-1 expression whereas $\max(\min(3x+7,0),\min(2x-6,-3x)) + 6\min(4x,3)$ is a level-2 expression.

Now we can see some properties about conjunctive and disjunctive form [3, 6]:

1. The expression $\max(f_1,\ldots,f_k)+\min(g_1,\ldots,g_l)$ can be written into either conjunctive form or disjunctive form

$$\begin{aligned}\max(f_1,\ldots,f_k)&+\min(g_1,\ldots,g_l)\\&=\max(\min(f_1+g_1,\ldots,f_1+g_l),\ldots,\\&\qquad\min(f_k+g_1,\ldots,f_k+g_l))\\&=\min(\max(f_1+g_1,\ldots,f_k+g_1),\ldots,\\&\qquad\max(f_1+g_l,\ldots,f_k+g_l))\end{aligned}\tag{2.14}$$

   We have used the property (2.6) in this case. By property (2.14) we can see again that the max operator is distributive with respect to the min operator and vice versa.

2. We can always convert a conjunctive form into a disjunctive form and vice versa:

$$\begin{aligned}\min(\max(f_{11},\ldots,f_{1k_1}),&\ldots,\max(f_{l1},\ldots,f_{lk_l}))\\&=\max(\min(f_{11},f_{21},\ldots,f_{l1}),\\&\qquad\min(f_{11},f_{21},\ldots,f_{l2}),\ldots,\min(f_{1k_1},\ldots,f_{lk_l}))\end{aligned}\tag{2.15}$$

   and therefore, in a compact notation

$$\min_{i=1,\ldots,l}(\max_{j=1,\ldots,k_i}(f_{ij}))=\max_{\substack{(j_1,\ldots,j_l)\in\\ \{1,\ldots,k_1\}\times\ldots\\ \times\{1,\ldots,k_l\}}}(\min_{i=1,\ldots,l}(f_{ij_i}))$$

   We have used the distributive property of min and max operation (see (2.11)) to interchange the order of the two operations. We can obviously convert a disjunctive form in a conjunctive one in the same way.

3. The expression $\min(\max(),\ldots,\max(),\min(),\ldots,\min())$ can be easily written in a conjunctive form. Let's see the following example:

$$\begin{aligned}\min(\max(f_1,f_2),&\min(f_3,f_4))\\&=\min(\max(f_1,f_2),f_3,f_4)\\&=\min(\max(f_1,f_2),\max(f_3,f_3),\max(f_4,f_4))\end{aligned}\tag{2.16}$$

   We have used the property (2.9) in this case.

4. By using properties (2.9) and (2.11) we can show that the expression $\max(\min(\max(f_1,f_2),f_3),f_4)$ can be easily written in a conjunctive form. Let

us consider the following example:

$$\begin{aligned}
&\max(\min(\max(f_1, f_2), f_3), f_4)\\
&\quad = \max(\max(\min(f_1, f_3), \min(f_2, f_3)), f_4)\\
&\quad = \max(\min(f_1, f_3), \min(f_2, f_3), f_4)\\
&\qquad = \min(\max(f_1, f_2, f_4), \max(f_1, f_3, f_4), \max(f_3, f_2, f_4), \dots,\\
&\qquad\quad \max(f_3, f_3, f_4))\\
&\qquad = \min(\max(f_1, f_2, f_4), \max(f_1, f_3, f_4), \max(f_3, f_2, f_4), \dots,\\
&\qquad\quad \max(f_3, f_4))\\
&\qquad = \min(\max(f_1, f_2, f_4), \max(f_3, f_4))
\end{aligned} \tag{2.17}$$

The terms $\max(f_1, f_3, f_4)$ and $\max(f_3, f_2, f_4)$ are superfluous, because they are always bigger than $\max(f_3, f_4)$. Then we can remove them from the expression.

By using properties (2.9) and (2.15), we have seen some examples of reductions of level-2 and level-3 expressions into a conjunctive canonical form. It is possible to formally prove that every MMPS expression can be transformed into a conjunctive or disjunctive form.

Theorem 2.2. *[6] The conjunctive and disjunctive form are canonical forms for any MMPS expression.*

2.3 Equivalence between continuous PWA and MMPS functions

Let us see some important results on PWA and MMPS functions:

Theorem 2.3. *[10, 14] Let $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a continuous PWA function as defined in (2.4): then there exists index sets $I_1, \dots, I_l \subseteq \{1, \dots, N\}$ such that:*

$$f = \min_{j \in 1,\dots,l} \max_{i \in I_j} (a_i^T x + b_i)$$

and there also exists index sets $J_1, \dots, J_k \subseteq \{1, \dots, N\}$

$$f = \max_{j \in 1,\dots,k} \min_{i \in J_j} (a_i^T x + b_i)$$

From [10, 14] and from Theorem 2.2, it follows that:

Theorem 2.4. *Every MMPS function is also a continuous PWA function.*

Therefore, we can give the following proposition:

Proposition 2.5. *[10, 14] MMPS and continuous PWA functions are equivalent.*

By Proposition 2.5, it is clear that MMPS and continuous PWA systems are equivalent too.

2.4 Implementation in MATLAB

For our aims, we need a MATLAB representation of PWA functions and MMPS functions in canonical form. We only consider scalar functions for both cases. In this section we show the MATLAB representation of these functions that we have chosen for our aims.

2.4.1 Implementation of PWA functions

The representation chosen for PWA functions is quite similar to the one of MPT toolbox [13], but with some differences.

Let us consider a PWA function $f : \mathcal{X} \longrightarrow \mathbb{R}$ as defined in Definition 2.3:

$$f(x) = \alpha_i^T x + \beta_i \qquad \text{if } x \in \mathcal{X}_i$$

For each $i \in \{1, \dots, N\}$, we can say that $x \in \mathcal{X}_i$ if $H_i x \leq K_i$. Then we can represent the PWA function with the following structure and fields:

$$\begin{aligned} f.\alpha &= \{\alpha_1^T, \dots, \alpha_N^T\} \\ f.\beta &= \{\beta_1, \dots, \beta_N\} \\ f.H &= \{H_1, \dots, H_N\} \\ f.K &= \{K_1, \dots, K_N\} \end{aligned} \tag{2.18}$$

where N is the number of affine terms. The fields H and K are necessary for defining every polyhedron of the polyhedral partition $\{\mathcal{X}_i\}$.

Example 2.5. Let us consider the function given in Example 2.2

$$f(x) = \begin{cases} f_1(x) = x + 1 & x \leq 3 \\ f_2(x) = 4 & 3 \leq x \leq 6 \\ f_3(x) = 2x - 8 & x \geq 6 \end{cases}$$

For this function we have:

$$\begin{aligned} f.\alpha &= \{1, 0, 2\} \\ f.\beta &= \{1, 4, -8\} \\ f.H &= \{1, [1\,;\,-1]\,, -1\} \\ f.K &= \{3, [6\,;\,-3]\,, -6\} \end{aligned}$$

Sometimes it can happen that a single term is defined in more than one polyhedron. This consideration lead us to study the generic case, in which N is the number of different terms, and M is the number of polyhedra that define the polyhedral partition $\{\mathcal{X}_i\}$. Of course, we always have $M \geq N$, with $M = N$ only if the number of terms equals the number of polyhedral regions. We can choose between two different strategies for defining the PWA function:

1. We can use the previous notation, and so we have:

$$\begin{aligned} f.\alpha &= \{\alpha_1^T, \ldots, \alpha_M^T\} \\ f.\beta &= \{\beta_1, \ldots, \beta_M\} \\ f.H &= \{H_1, \ldots, H_M\} \\ f.K &= \{K_1, \ldots, K_M\} \end{aligned} \tag{2.19}$$

   We can see that all fields of this structure are cell arrays containing matrices.

   In this case, if $N < M$, there surely are some index sets $L_1, \ldots, L_j \subseteq \{1, \ldots, M\}$ with more than one index in them, for which $\alpha_k = \alpha_l$ and $\beta_k = \beta_l$ for every pair of indices $k, l \in L_m$, for $m = 1, \ldots, j$.

2. Otherwise, we can use a new notation:

$$\begin{aligned} f.\alpha &= \{\alpha_1^T, \ldots, \alpha_N^T\} \\ f.\beta &= \{\beta_1, \ldots, \beta_N\} \\ f.H &= \Big\{\{H_{11}, \ldots, H_{1t_1}\}, \ldots, \{H_{N1}, \ldots, H_{Nt_N}\}\Big\} \\ f.K &= \Big\{\{K_{11}, \ldots, K_{1t_1}\}, \ldots, \{K_{N1}, \ldots, K_{Nt_N}\}\Big\} \end{aligned} \tag{2.20}$$

   Here, all fields are cell arrays whose the elements are cell arrays containing matrices.

   Therefore, every dynamic i is defined over t_i different polyhedra. In this case, $f(x) = \alpha_i^T x + \beta_i$ if $H_{ij} x \leq K_{ij}$, for $j = 1, \ldots, t_i$.

For our purposes, it is better to use notation (2.20) in all cases (also when $N = M$ and therefore $t_i = 1$ for each $i \in \{1, \ldots, N\}$). In fact, as we will see in the continuation of the thesis, this representation is more suitable, especially for the conversion from a continuous PWA function into the equivalent MMPS one. Anyway, it is possible to use both notations, or also a mix between the two notations.

Example 2.6. Let us consider the following PWA function:

$$f(x) = \begin{cases} f_1(x) = 0 & -5 \le x \le -2 \; or \; 2 \le x \le 5 \\ f_2(x) = x + 2 & -2 \le x \le 0 \\ f_3(x) = -x + 2 & \;\; 0 \le x \le 2 \end{cases}$$

If we use the MATLAB notation (2.20), we have the following representation for f:

$$\begin{aligned} & f.\alpha = \{0, 1, -1\} \\ & f.\beta = \{0, 2, 2\} \\ & f.H = \{\{[1\,;\,-1]\,,[1\,;\,-1]\}, \{[1\,;\,-1]\}, \{[1\,;\,-1]\}\} \\ & f.K = \{\{[-2\,;\,5]\,,[5\,;\,-2]\}, \{[0\,;\,2]\}, \{[2\,;\,0]\}\} \end{aligned}$$

2.4.2 Implementation of MMPS functions in canonical form

We have already seen by Theorem 2.2 that every MMPS function can be represented by one of its canonical forms, the conjunctive or the disjunctive one. For our aims, we need to represent by MATLAB MMPS functions in canonical form only [1].

Let $f : \mathcal{X} \longrightarrow \mathbb{R}$ be an MMPS function in the form

$$\min_{j=1,\ldots,l} \max_{i \in I_j} (\alpha_i^T x + \beta_i)$$

where $\mathcal{X} \subset \mathbb{R}^n$. We can use for it the following MATLAB representation

$$\begin{aligned} & f.\alpha = \{\alpha_1^T, \ldots, \alpha_N^T\} \\ & f.\beta = \{\beta_1, \ldots, \beta_N\} \\ & f.terms = \{[I_{1_1} \; \ldots \; I_{1_{m_1}}], \ldots, [I_{l_1} \; \ldots \; I_{l_{m_l}}]\} \\ & f.form = \text{‘}conj\text{’} \\ & f.A = A \\ & f.b = b \end{aligned} \tag{2.21}$$

[1] Actually, it is not always a strictly canonical form, i.e., we use the form

$$f(x) = \max(f_2(x), \min(f_1(x), f_3(x)))$$

instead of

$$f(x) = \max(\min(f_2(x), f_2(x)), \min(f_1(x), f_3(x)))$$

where the fields α and β have the same meaning as in PWA representation, and $x \in \mathcal{X}$ if $Ax \leq b$. The terms A and b are not necessary since if they are absent, a domain is defined by default (see Section 4.1.1). If we look at the field *terms*, we have that $I_{j_1}, \ldots, I_{j_{m_j}}$ for $j = 1, \ldots, l$ are the indices in I_j. The string contained in the field *form* is 'conj' if the function is in conjunctive form, while is 'disj' if it is in the disjunctive one.

Example 2.7. Let us consider the following MMPS function

$$f(x) = \max(\min(f_1(x), f_2(x)), \min(f_1(x), f_3(x))) \qquad \forall x \in \mathcal{X}$$

in which

$$\begin{aligned} f_1(x) &= x_2 + 1 \\ f_2(x) &= x_1 \\ f_3(x) &= -x_1 + 4 \end{aligned}$$

and $\mathcal{X} := \{x \in \mathbb{R}^2 : -20 \leq x_1 \leq 20,\ -20 \leq x_1 \leq 20\ \}$. In MATLAB we have the following representation:

$$\begin{aligned} &f.\alpha = \{[0\ 1], [1\ 0], [-1\ 0]\} \\ &f.\beta = \{1, 0, 4\} \\ &f.terms = \{[1\ 2], [1\ 3]\} \\ &f.form = \text{`}disj\text{'} \\ &f.A = [1\ 0; -1\ 0; 0\ 1; 0\ \ -1] \\ &f.b = [20; 20; 20; 20] \end{aligned}$$

2.5 Summary

In this chapter we have at first described in detail PWA functions, for which we have especially focused on the continuous ones, and MMPS functions, with a special consideration for its canonical forms. Next, we have shown the equivalences between continuous PWA and MMPS functions. At the end of the chapter we have seen how we can describe PWA and MMPS functions in canonical form in MATLAB.

Chapter 3

Minimal realizations of PWA and MMPS functions

When we use PWA and MMPS functions, it is suitable to have a computational complexity as small as possible. In order to do it, we need to have both kinds of functions in their minimal representation. In this chapter we will describe these representations for the two kinds of functions.

3.1 Minimal representation of PWA functions and merging of polyhedra

Let us consider a PWA function $f : \mathcal{X} \longrightarrow \mathbb{R}$ as defined in Definition 2.3: so there exists a polyhedral partition $\{\mathcal{X}_i\}_{i\in\mathcal{I}}$ of $\mathcal{X}$ such that f is affine on each $\mathcal{X}_i$, i.e. $f(x) = \alpha_i^T x + \beta_i$ for all $x \in \mathcal{X}_i$ and $i \in \mathcal{I}$. As already said in Section 2.4.1, it could happen that an affine term is defined in more than one polyhedron: this is the reason why it is better to use for PWA functions a new definition different from Definition 2.3.

Definition 3.1. A function $f : \mathcal{X}' \to \mathbb{R}$, where $\mathcal{X}' \subseteq \mathbb{R}^n$ is a polyhedron, is PWA if there exists a polyhedral partition $\{\mathcal{X}'_{ij}\}_{i\in\{1,\ldots,M\},j\in\{1,\ldots,m_i\}}$ of $\mathcal{X}'$ such that $f(x) = \alpha_i^T x + \beta_i$ on each $\mathcal{X}'_{ij}$ for every $i \in \{1,\ldots,M\}$ and $j \in \{1,\ldots,m_i\}$, where m_i is the number of polyhedra in which the affine term $\alpha_i^T x + \beta_i$ is defined and M is the number of affine terms. All the affine terms $\alpha_i^T x + \beta_i$ are different.

We therefore have some sets of polyhedra, that we can denote as $\mathcal{X}'_1, \ldots, \mathcal{X}'_M$ where for every $i \in \{1,\ldots,M\}$ we have that $\mathcal{X}'_i = \{\mathcal{X}'_{i1}, \ldots, \mathcal{X}'_{im_i}\}$ and $f(x) = \alpha_i^T x + \beta_i$ on each polyhedra in $\mathcal{X}'_i$.

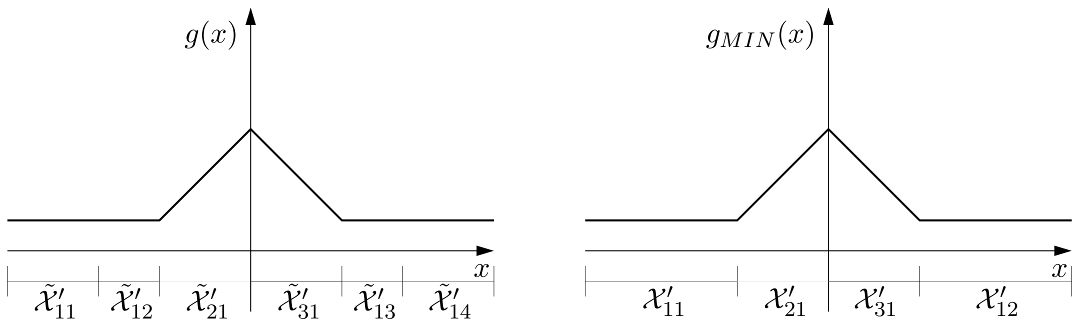

Figure 3.1: *The PWA function $g(x)$ in a non minimal representation (on the left) and in a minimal one $g_{MIN}(x)$ (on the right).*

Now we are in the position to define the minimal realization of a PWA function:

Definition 3.2. A minimal representation of a PWA function is a representation in which each set of polyhedra $\mathcal{X}'_i$ contains the minimal number of polyhedra.

So, when some or all the polyhedral regions in which the same affine term is defined form a convex union, they have to be merged in order to compact the representation.

Example 3.1. Let us consider the function

$$g(x) = \begin{cases} 1 & x \in \{\tilde{\mathcal{X}}'_{11}, \tilde{\mathcal{X}}'_{12}, \tilde{\mathcal{X}}'_{13}, \tilde{\mathcal{X}}'_{14}\} \\ x+4 & x \in \tilde{\mathcal{X}}'_{21} \\ -x+4 & x \in \tilde{\mathcal{X}}'_{31} \end{cases} \tag{3.1}$$

where $g : \mathcal{X}' \longrightarrow \mathbb{R}$ and $\{\tilde{\mathcal{X}}'_{ij}\}$ is a polyhedral partition of $\mathcal{X}'$. This function is shown in Figure 3.1 on the left, and, as we can see by the graph, is not in its minimal representation. This one is given by the function

$$g_{MIN}(x) = \begin{cases} 1 & x \in \{\mathcal{X}'_{11}, \mathcal{X}'_{12}\} \\ x+4 & x \in \mathcal{X}'_{21} \\ -x+4 & x \in \mathcal{X}'_{31} \end{cases} \tag{3.2}$$

where $g_{MIN} : \mathcal{X}' \longrightarrow \mathbb{R}$ and in which $\{\mathcal{X}'_{ij}\}$ is a new polyhedral partition of $\mathcal{X}'$. We can note that $\mathcal{X}'_{11} = \tilde{\mathcal{X}}'_{11} \cup \tilde{\mathcal{X}}'_{12}$ and $\mathcal{X}'_{12} = \tilde{\mathcal{X}}'_{13} \cup \tilde{\mathcal{X}}'_{14}$, as is also shown in Figure 3.1 on the right. We can also see that $\mathcal{X}'_{11}$ and $\mathcal{X}'_{12}$ cannot be merged because they do not form a convex union.

It frequently happens that there is not only one way for merging the polyhedra in which an affine component f_i is defined. Moreover, if the number of polyhedra that should be merged is large, the number of possible combinations in which they can be merged becomes huge. A method for obtaining the minimal representation of a PWA function, is given by the algorithm of optimal merging given in [9], that is a quite efficient algorithm.

3.2 Minimal realizations for canonical forms of MMPS functions

Let us consider an MMPS function in one of its canonical forms: we often have that these canonical forms are not in their minimal representation.

As we already know from Definition 2.8, a generic MMPS function f in its conjunctive form can be represented as

$$f(x) = \min_{j \in 1,\ldots,l} \max_{i \in I_j} (a_i^T x + b_i)$$

where $I_1, \ldots, I_l \subseteq \{1, \ldots, N\}$ are index sets and there are N possible components in the form $a_i^T x + b_i$.

Definition 3.3. The function f is in its minimal realization if there does not exist a function

$$f_1(x) = \min_{j \in 1,\ldots,l'} \max_{i \in I'_j} (a_i^T x + b_i)$$

with $l' < l$, such that $f(x) \equiv f_1(x)$, and next, if we cannot remove any entry in some of the index sets I_j without modifying the meaning of the function.

We therefore have two kinds of reduction:

1. The first one is the reduction of max terms. Let us consider two different indices $l_1, l_2 \in \{1, \ldots, l\}$: we can remove, e.g., the index l_1, if

$$\max_{i \in I_{l_1}} (a_i^T x + b_i) \geq \max_{i \in I_{l_2}} (a_i^T x + b_i)$$

2. Once the first reduction has been applied, we have l' max terms, with $l' \leq l$. Let us now consider an index $k \in \{1, \ldots, l'\}$: we can replace the index set I_k with another index set I'_k with a smaller cardinality if

$$\max_{i \in I_k} (a_i^T x + b_i) \geq \max_{i \in I'_k} (a_i^T x + b_i)$$

The two kinds of reductions shown above can also be done in other cases.

Definition 3.3 is also valid, with proper modifications, for MMPS functions in disjunctive form, of course.

Example 3.2. Let us consider the scalar MMPS function

$$\begin{aligned} g(x) &= \min_{j\in 1,\ldots,l} \max_{i\in I_j}(a_i^T x + b_i) \\ &= \min(\max(2x+1,0,4),\max(x+2,-0.5x+2),\max(x+8,-x+8)) \end{aligned}$$

We can note that $l = 3$. $g(x)$ is not in its minimal form: in fact we can equivalent write it as

$$\begin{aligned} g_1(x) &= \min_{j\in 1,\ldots,l'} \max_{i\in I'_j}(a_i^T x + b_i) \\ &= \min(\max(2x+1,0,4),\max(x+2,-0.5x+2)) \end{aligned}$$

where $l' = 2$ and therefore $l' < l$. So we have $g_1(x) \equiv g(x)$. $g_1(x)$ is not the minimal form of $g(x)$ yet. In fact we can equivalent write $g_1(x)$ as

$$g_2(x) = \min(\max(2x+1,4),\max(x+2,-0.5x+2))$$

We have removed an entry in I'_1, but we still have $g_2(x) \equiv g_1(x) \equiv g(x)$.

At the moment, there does not exist a general efficient algorithm for generating the minimal realization of an MMPS function in canonical form yet [6].

3.3 Summary

In this chapter we have described the minimal realizations for PWA and canonical forms of MMPS functions. These realizations are very important for having a smaller computational complexity in our tasks. We have also seen that there exists an efficient algorithm for obtaining the minimal realization of a PWA function, whereas there does not exist an algorithm for doing it in the case of MMPS functions in canonical form.

Chapter 4

From continuous PWA to MMPS functions

We know from Theorem 2.3 that every continuous PWA function can be transformed in an equivalent MMPS function, where the latter is in one of its canonical forms. In this chapter we show two strategies for doing it: the Gorokhovik-Zorko strategy [10] and the Ovchinnikov strategy [14]. In both cases we illustrate them with reference to the MATLAB codes written for implementing these methods. All the routines for the conversions can be launched by the code **pwa2mmps** and all the codes are given in Appendix A. We only show the conversion into the disjunctive form, since because of the duality between the canonical forms, we can easily obtain the conversion into the conjunctive form through a few modifications of the strategies introduced here.

4.1 The Gorokhovik-Zorko strategy

4.1.1 Strategy

In this section we show the Gorokhovik-Zorko strategy, that is based on paper [10].

At first, we need to be sure that the PWA function in input to the code has some characteristics:

- It must respect either the notation (2.19) or the notation (2.20) (it is also possible a mix between the two notations);
- It must be continuous and PWA.

For verifying it, we have created the MATLAB code **testPWA**, that verifies whether the properties above have been respected and returns an error message otherwise. Moreover, this code checks if some of the affine terms in the PWA function given as input are equal (e.g., as in the MATLAB notation (2.19)) and returns as output the same PWA function in the notation (2.20), and so all the affine terms will be different. The code also performs the merging of polyhedral regions when it is necessary, by means of an algorithm of greedy (non-optimal) merging given in [13], that is much faster of the algorithm of optimal merging given in [9].

We can now illustrate the Gorokhovik-Zorko strategy, that has been implemented by the MATLAB code **pwa2mmpsDisjGor**.

At first, of course, the subroutine **testPWA** above described has to be executed. Next, we can proceed with the veritable strategy.

Let us consider a PWA function $f : \mathcal{X}' \to \mathbb{R}$ as described in Definition 3.1, but with $\mathcal{X}' \subset \mathbb{R}^n$, where $\mathcal{X}'$ is a closed polyhedron. So there exists a polyhedral partition $\{\mathcal{X}'_{ij}\}_{i\in\{1,\dots,M\},j\in\{1,\dots,m_i\}}$ of $\mathcal{X}'$ such that $f(x) = \alpha_i^T x + \beta_i$ on each $\mathcal{X}'_{ij}$ for every $i \in \{1,\dots,M\}$ and $j = 1,\dots,m_i$, where m_i is the number of polyhedra in which the affine term $\alpha_i^T x + \beta_i$ is defined and M is the number of affine terms. This strategy allows us to convert this PWA function into the equivalent MMPS function that we can write, e.g., in its disjunctive form:

$$y = \max_{j=1,\dots,l} \min_{i\in I_j} f_i \tag{4.1}$$

where $f_i = \alpha_i^T x + \beta_i$ and $I_1,\dots,I_l \subseteq \{1,\dots,M\}$. But how can we find the index sets I_j, for $j = 1,\dots,l$? Let us consider the following proposition:

Proposition 4.1. *We can say that $I_j \subseteq \{1,\dots,M\}$ can be an index set for the MMPS function y in* (4.1)*, and so $I_j \in \{I_1,\dots,I_l\}$ if and only if*

$$\min_{i\in I_j} f_i \leq f \tag{4.2a}$$

or equivalently

$$\operatorname{hyp}(\min_{i\in I_j} f_i) \subseteq \operatorname{hyp} f \tag{4.2b}$$

We now give a proof of Proposition 4.1

Proof 4.1. We know from Theorem 2.3 that there surely exists an MMPS function $y = \max_{j=1,\dots,l} \min_{i\in I_j} f_i$ such that $y = f$. Let us now consider an index set $I_{j_1} \subseteq \{1,\dots,M\}$ such that $\min_{i\in I_{j_1}} f_i \leq f$. We can note that j_1 can belong to $\{1,\dots,l\}$ without modifying the meaning of the function. Then, j_1 can be included in $\{1,\dots,l\}$.

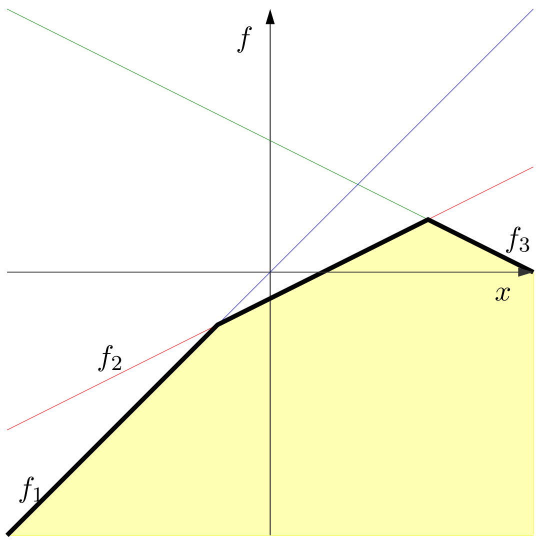

Figure 4.1: hyp(min(f_1, f_2, f_3))

Suppose now that there exists an index set $I_{j_2} \subseteq \{1, \ldots, M\}$ with $j_2 \in \{1, \ldots, l\}$, such that $\min_{i \in I_{j_2}} f_i \nleq f$, and so $\min_{i \in I_{j_2}} f_i > f$ for some $x \in \mathcal{X}'$. It is clear that this is not possible and so j_2 cannot be included in $\{1, \ldots, l\}$, because in this case we should have $y > f$ in the points of $\mathcal{X}'$ in which $\min_{i \in I_j} f_i > f$.

With MATLAB we can easily check if the condition (4.2b) is satisfied or not. Since every *min* term is a concave function [15], the code can compute the *hypograph* of each *min* term as the intersection of the *hypographs* of all its arguments, as shown in the example of Figure 4.1. Therefore the *hypograph* of the PWA function can be computed as the *union* of the polyhedra H_i, for $i = 1, \ldots, M$, where we define H_i as follows:

$$H_i = \text{hyp}(f_i) \cap ((\cup_{j=1,\ldots,m_i} \mathcal{X}'_{ij}) \times \mathbb{R}) \tag{4.3a}$$

and so

$$\text{hyp}\, f = \cup_{i=1,\ldots,M} H_i \tag{4.3b}$$

An example of the method given by (4.3) is shown in Figure 4.2.

We are now able to find all *min* terms that satisfy the condition (4.2). Unfortunately, finding all terms that satisfy the condition (4.2b) is very inefficient. For showing this, let us first introduce the following definition:

Definition 4.1. The power set of a set R is the set of all its subsets, and it is denoted as $\mathcal{P}(R)$.

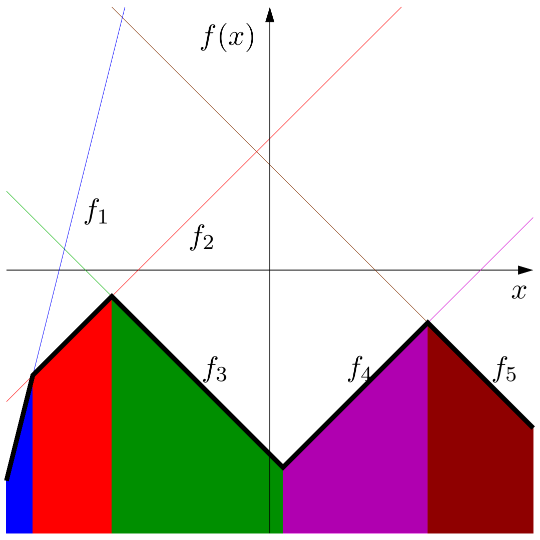

Figure 4.2: *Computation of* $\operatorname{hyp} f$ *as given in* (4.3)

Example 4.1. The power set of the set $R = \{1, 2, 3\}$ is given by

$$\mathcal{P}(R) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

The code must verify the condition (4.2b) for each entry of the power set of the set $I = \{1, \dots, M\}$ (except for the empty set of course). Let us try to better explain this with an algorithm:

Algorithm 4.1.

```
1. Let $I = \{1, \dots, M\}$ and $S$ a set defined as $S = \mathcal{P}(I) - \emptyset$;
2. for each set $I_j \in S$ do;
3. if $\operatorname{hyp}(\min_{i \in I_j} f_i) \not\subseteq \operatorname{hyp} f$, then remove $I_j$ from $S$;
4. endfor;
5. let $S' = S$; return $S'$.
```

Now, if we consider the set S' returned by Algorithm 4.1 we have that

$$y = \max_{I_j \in S'} \min_{i \in I_j} f_i \tag{4.4}$$

The code becomes very inefficient when $|I|$ is too high, where $|I|$ is the *cardinality* of the set I: in fact, for a generic set J with $|J| = N$, the *cardinality* of its power set is $|\mathcal{P}(J)| = 2^N$. So, the Algorithm 4.1 must verify the condition (4.2b) for $2^N - 1$ times. It is evident that if $|I|$ is quite high, the execution of the code could require a lot of time.

Moreover, the MMPS function y obtained as illustrated in Proposition 4.2 is not in the minimal form (see Definition 3.3), but on the contrary there are all possible *min* terms. In Section 4.1.2 we will illustrate the remaining part of the code **pwa2mmpsDisjGor**, in which a method has been implemented for obtaining a reduced form for the function y.

4.1.2 Reduction of the obtained MMPS function

As seen in Section 3.2, it is better to find either the minimal form of an MMPS function or at least a reduced form. Now we are going to see a way for obtaining a reduced realization of (4.4).

Let us consider two index sets $J_{k_1}, J_{k_2} \in S'$. We can surely remove the index J_{k_1} from S' if

$$\min_{i \in J_{k_1}} f_i \leq \min_{i \in J_{k_2}} f_i \tag{4.5a}$$

or equivalently

$$\text{hyp}(\min_{i \in J_{k_1}} f_i) \subseteq \text{hyp}(\min_{i \in J_{k_2}} f_i) \tag{4.5b}$$

Example 4.2. Let f be the function in Figure 4.3, where $\text{hyp}(\min(f_2, f_3)) \subseteq \text{hyp}(\min(f_1, f_2))$: therefore, $\min(f_2, f_3)$ is a redundant term and can be removed.

So, in order to discover and remove some redundant *min* terms for y, the code makes use of the following algorithm:

Algorithm 4.2.

1. Let S' be the collection of index sets returned by the Algorithm 4.1, ordered on the basis of an increasing number of entries;
2. **for** each pair $J_k, J_m \in S'$, $J_k \neq J_m$ **do**;
3. **if** $\text{hyp}(\min_{i \in J_k} f_i) \subseteq \text{hyp}(\min_{i \in J_m} f_i)$, **then** remove J_k from S';
4. **endfor**;
5. let $S'' = S'$; **return** S''.

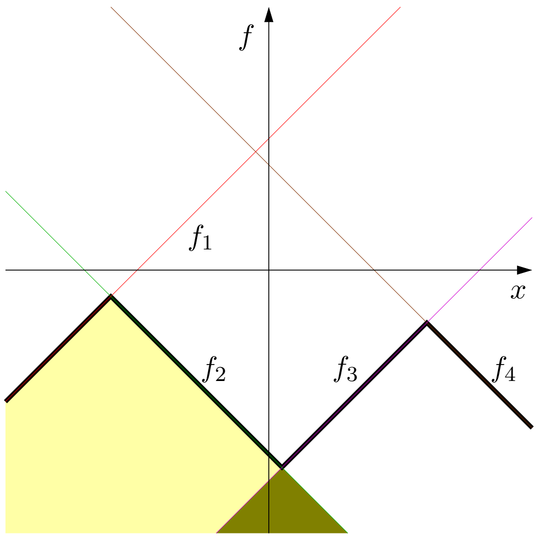

Figure 4.3: $\mathrm{hyp}(\min(f_2, f_3)) \subseteq \mathrm{hyp}(\min(f_1, f_2))$

Let us now illustrate the Algorithm 4.2: if we look at the collection of index sets S'', we can note that there does not exist a pair of index sets $J_{m_1}, J_{m_2} \in S''$ such that

$$\min_{i \in J_{m_1}} f_i \le \min_{i \in J_{m_2}} f_i$$

Moreover, let us consider the collection of index sets in input S'. Let us also denote as $K_1, \ldots, K_r$ all subcollections of index sets included in S' such that, for each one of these subcollections and for all index sets $J_{l_1}, \ldots, J_{l_{n_l}} \in K_l$, with $l = 1, \ldots, r$, we have:

$$\min_{i \in J_{l_1}} f_i = \min_{i \in J_{l_2}} f_i = \ldots = \min_{i \in J_{l_{n_l}}} f_i$$

Because of the order of the sets in the collection S', we can note that for each subcollection K_l, only the index set with the smallest number of elements can be kept (or one of the index sets if there are more than one of them). Therefore, we can now equivalently represent the function y in the reduced form

$$\tilde{y} = \max_{I_j \in S''} \min_{i \in I_j} f_i \tag{4.6}$$

However, by using the Algorithm 4.2, we can remove lots of redundant *min* terms, but not all. Let f be, e.g., the function in Figure 4.4: we can note that $\mathrm{hyp}(\min(f_5, f_{10})) \subseteq \mathrm{hyp}(f)$, but there does not exist an index set $I_p \in S''$, with $I_p \neq \{5, 10\}$, such that $\mathrm{hyp}(\min(f_5, f_{10})) \subseteq \mathrm{hyp}(\min_{i \in I_p}(f_i))$, so the term $\min(f_5, f_{10})$ cannot be removed through the Algorithm 4.2.

If we want to remove this kind of terms once the Algorithm 4.2 has already been applied, we can use the following algorithm:

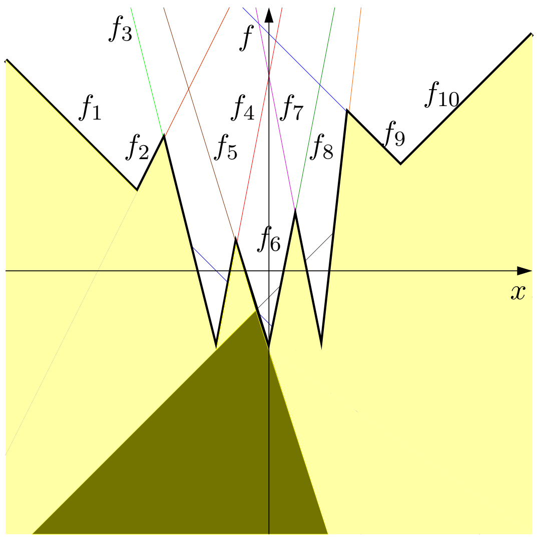

Figure 4.4: hyp$(\min(f_5, f_{10})) \subseteq$ hyp f, *but we cannot remove the term* $\min(f_5, f_{10})$ *from* $\tilde{y}$ *through the strategy of Algorithm 4.2*

Algorithm 4.3.

```
1. Let $S''$ be the collection of index sets returned by the Algorithm 4.2, ordered on the basis of a decreasing number of entries;
2. **for** each $I_k \in S''$ **do**
3. **if** hyp$(\min_{i\in I_{k'}} f_i) \subseteq$ hyp$(\max_{I_k\in S'', I_k\neq I_{k'}} \min_{i\in I_k} f_i)$, **then** remove $I_{k'}$ from $S''$;
4. **endfor**;
5. let $S''' = S''$; **return** $S'''$.
```

By means of Algorithm 4.3 above, the code checks for each index $I_k \in S''$ if

$$\min_{i\in I_{k'}} f_i \leq \max_{\substack{I_k\in S'' \\ I_k\neq I_{k'}}} \min_{i\in I_k} f_i \tag{4.7}$$

In this case the index set $I_{k'}$ is removed from S''.

Moreover, let us consider two index sets $S_1, S_2 \in S''$ such that

$$\max_{I_k\in S''-S_1} \min_{i\in I_k} f_i = f \tag{4.8a}$$

$$\max_{I_k\in S''-S_2} \min_{i\in I_k} f_i = f \tag{4.8b}$$

$$\max_{I_k\in S''-\{S_1,S_2\}} \min_{i\in I_k} f_i \leq f \tag{4.8c}$$

where $|S_1| > |S_2|$. In this case we can equivalently remove from S'' either the index set S_1 or S_2, but not both two. In order to obtain a MMPS function as reduced as possible, we have an order of entries in S'' as specified in the Step 1 of the Algorithm 4.3: because of this, the Algorithm 4.3 removes the index set S_1, that is the one with a bigger *cardinality*.

Unfortunately, we cannot say that the MMPS function

$$\tilde{\tilde{y}} = \max_{I_j \in S'''} \min_{i \in I_j} f_i \tag{4.9}$$

is in the minimal form yet. For doing this, we have to prove it, but we are not able yet.

4.1.3 Example

Let us consider the following continuous PWA function $f : \mathcal{X} \longrightarrow \mathbb{R}$:

$$f(x) = \begin{cases} f_1(x) = 3 & -20 \leq x \leq -7 \textit{ or } -3 \leq x \leq -2 \textit{ or} \\ & 2 \leq x \leq 3 \textit{ or } 7 \leq x \leq 20 \\ f_2(x) = 3x + 24 & -7 \leq x \leq -6 \\ f_3(x) = -x & -6 \leq x \leq -3 \\ f_4(x) = -3x - 3 & -2 \leq x \leq -1 \\ f_5(x) = 4x + 4 & -1 \leq x \leq 0 \\ f_6(x) = -4x + 4 & 0 \leq x \leq 1 \\ f_7(x) = 3x - 3 & 1 \leq x \leq 2 \\ f_8(x) = x & 3 \leq x \leq 6 \\ f_9(x) = -3x - 24 & 6 \leq x \leq 7 \end{cases} \tag{4.10}$$

As we can see from the definition of the function, we have $\mathcal{X} =: \{x \in \mathbb{R} : -20 \leq x \leq 20\}$.

We want to convert it into the equivalent MMPS function in disjunctive form, by means of the Algorithms 4.1, 4.2 and 4.3.

At first, by using the Algorithm 4.1 we obtain an equivalent MMPS function $y_1 : \mathcal{X} \longrightarrow \mathbb{R}$ with 432 min terms.

If we apply the Algorithm 4.2 to the function y_1 we obtain the following function y_2:

$$\begin{aligned} y_2(x) = \max(&\min(f_1, f_4), \min(f_1, f_7), \min(f_3, f_5), \\ &\min(f_3, f_8), \min(f_5, f_6), \min(f_6, f_8), \min(f_2, f_3, f_4), \\ &\min(f_7, f_8, f_9)) \qquad \forall x \in \mathcal{X} \end{aligned} \tag{4.11}$$

The function y_2 is not in the minimal form yet.

By applying the Algorithm 4.3 to the function y_2, we obtain the following function:

$$\begin{aligned} y(x) = \max(&\min(f_1, f_4), \min(f_1, f_7), \min(f_5, f_6), \min(f_2, f_3, f_4), \\ &\min(f_7, f_8, f_9)) \qquad \forall x \in \mathcal{X} \end{aligned} \tag{4.12}$$

This function y is in the minimal form.

This example is given in the file **PWA_example**.

4.2 The Ovchinnikov strategy

4.2.1 Strategy

Another strategy for the conversion of a continuous PWA function into the equivalent MMPS one is the Ovchinnikov strategy that is based on his paper [14]. In order to implement this strategy we have created the code **pwa2mmpsDisjOvc**.

In this section, for a better comprehension, we proceed in the same way: at first we illustrate the strategy as in [14], and next we give the algorithm used in the MATLAB code **pwa2mmpsDisjOvc**.

Let $f : \mathcal{X}' \longrightarrow \mathbb{R}$ be the continuous PWA function that we want to convert into the equivalent MMPS function. We can define f as in Definition 3.1, but with $\mathcal{X}' \subset \mathbb{R}^n$, where $\mathcal{X}'$ is a closed polyhedron. So there are M affine components of f, that we denote as f_i for $i = 1, \dots, M$. All the components f_i are different.

Let us consider now all the hyperplanes that are nonempty solution sets of the equations in the form $f_i = f_j$, for $i < j$ and have nonempty intersections with the *interior* of $\mathcal{X}'$. These hyperplanes form an hyperplane arrangement, that is, a finite set of hyperplanes in an n dimensional space: therefore, every hyperplane is $n-1$ dimensional. We can define this hyperplane arrangement as $\mathcal{H}$. The subdivision of $\mathcal{X}'$ by the hyperplanes in $\mathcal{H}$ generates a polyhedral partition in $\mathcal{X}'$: we denote the set of regions of this polyhedral partition as $\mathcal{T}$.

Before to continue, some definitions have to be introduced:

Definition 4.2. A facet is a $n-1$ dimensional face of a polyhedron in $\mathbb{R}^n$.

Definition 4.3. Two polyhedral regions are adjacent if they have a facet in common.

Let us choose now all the pairs of affine component f_p, f_q of f such that the following conditions are satisfied:

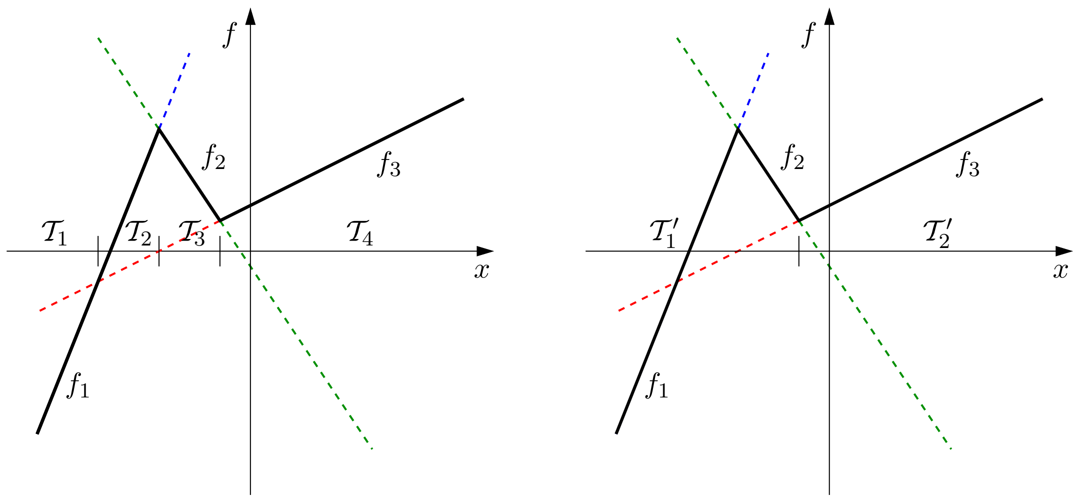

Figure 4.5: *A continuous PWA function f and its region sets $\mathcal{T}$ and $\mathcal{T}'$.*

1. There is a pair of adjacent regions $\mathcal{P}, \mathcal{Q} \in \mathcal{T}$ such that $f_p = f$ on $\mathcal{P}$ and $f_q = f$ on $\mathcal{Q}$.
2. $f = \max(f_p, f_q)$ on $\mathcal{P} \cup \mathcal{Q}$

We now consider the collection of the hyperplanes that are nonempty solution sets of the equations in the form $f_p = f_q$, for each pair f_p, f_q that satisfies the previous conditions, and we denote as $\mathcal{H}'$ the hyperplane arrangement given by them. We can easily see that $\mathcal{H}' \subseteq \mathcal{H}$. Therefore let us denote as $\mathcal{T}'$ the set of regions obtained through the subdivision of $\mathcal{X}'$ by the hyperplanes in $\mathcal{H}'$. The set of regions $\mathcal{T}'$ generates another polyhedral partition on $\mathcal{X}'$. We denote the regions of $\mathcal{T}'$ as $\mathcal{T}_1', \ldots, \mathcal{T}_t'$, and for each $j = 1, \ldots, t$ we define the index set S_j as $S_j = \{i \in \{1, \ldots, M\} : f_i(x) \geq f(x), \forall x \in \mathcal{T}_j'\}$.

We can then represent the function f by the following equivalent MMPS function y:

$$y = \max_{j=1,\ldots,t} \min_{i \in S_j} f_i \tag{4.13}$$

Unfortunately, the function (4.13) obtained by this strategy is not always in the minimal form.

Example 4.3. In Figure 4.5 a continuous PWA function f is shown, together with the region sets $\mathcal{T}$ and $\mathcal{T}'$ computed as previous described. For this function we have $S_1 = \{1, 2\}$ and $S_2 = \{1, 3\}$, so the equivalent MMPS function is $y = \max(\min(f_1, f_2), \min(f_1, f_3))$. In this case the function y obtained by the Ovchinnikov strategy is in the minimal form, but this is not true for every function.

But how can we implement this strategy in a MATLAB code? At first, as for the code **pwa2mmpsDisjGor** described in Section 4.1, the subroutine **testPWA** already seen there has to be executed, for the same reasons.

Next, we can find the region set $\mathcal{T}'$ by the following algorithm:

Algorithm 4.4.

```
1. Let $\mathcal{X}'$ be the domain of the function $f$;
2. for each pair of adjacent polyhedral regions $\mathcal{X}_{ik}, \mathcal{X}_{jl} \in \mathcal{X}'$, with $i < j$, do;
3. if $f_i \geq f_j$ on $\mathcal{X}_{ik}$ and $f_i \leq f_j$ on $\mathcal{X}_{jl}$, insert the hyperplane that splits the two regions in $\mathcal{H}'$;
4. endfor;
5. let $\mathcal{T}'$ the set of regions given by the intersection of $\mathcal{X}'$ with the regions of the hyperplane arrangement $\mathcal{H}'$; return $\mathcal{T}'$; stop;
```

Let us now explain Algorithm 4.4 in more detail. At first, we can see that for each pair of polyhedral regions $\mathcal{X}_{ik}, \mathcal{X}_{jl}$ that satisfies the condition given in Step 2, the hyperplane that splits these two regions is the set of solutions of the equation $f_i = f_j$. So it can be put in $\mathcal{H}$. Next, if also the condition in Step 3 is satisfied, we have $f = \max(f_i, f_j)$ on $\mathcal{X}_{ik} \cup \mathcal{X}_{jl}$, and so, since the components are affine, there surely exists at least a pair of adjacent regions $\mathcal{P}, \mathcal{Q} \in \mathcal{T}$, with $\mathcal{P} \subseteq \mathcal{X}_{ik}$ and $\mathcal{Q} \subseteq \mathcal{X}_{jl}$ such that $f = \max(f_i, f_j)$ on $\mathcal{P} \cup \mathcal{Q}$. The hyperplane that splits these two regions can therefore be put in $\mathcal{H}'$. By this method we can find all the possible hyperplanes that have to be inserted in $\mathcal{H}'$. We can then find all the regions $\mathcal{T}'_1, \ldots, \mathcal{T}'_t \in \mathcal{T}'$.

Now, for each region $\mathcal{T}'_j \in \mathcal{T}'$, we can check for the affine components f_i such that $f_i \geq f$ on $\mathcal{T}'_j$, with $j = 1, \ldots, t$. In order to do it we can use the following algorithm:

Algorithm 4.5.

```
1. Let $\mathcal{T}'$ be the region set returned by Algorithm 4.4;
2. for each region $\mathcal{T}'_j \in \mathcal{T}'$, with $j = 1, \ldots, t$ and for each affine component $f_i$ of $f$, with $i = 1, \ldots, M$, do;
3. if $f_i \geq f$ on $\mathcal{T}'_j$, then insert the index $i$ in the index set $S_j$;
4. endfor;
5. return the index sets $S_j$ for $j = 1, \ldots, t$;
```

Now, we can finally write the PWA continuous function f as

$$\max_{j=1,\ldots,t} \min_{i\in S_j} f_i \tag{4.14}$$

that is as in (4.13).

4.2.2 Example

Let us consider the continuous PWA function $f : \mathcal{X} \longrightarrow \mathbb{R}$ given in (4.10):

$$f(x) = \begin{cases} f_1(x) = 3 & -20 \le x \le -7 \textit{ or } -3 \le x \le -2 \textit{ or} \\ & 2 \le x \le 3 \textit{ or } 7 \le x \le 20 \\ f_2(x) = 3x + 24 & -7 \le x \le -6 \\ f_3(x) = -x & -6 \le x \le -3 \\ f_4(x) = -3x - 3 & -2 \le x \le -1 \\ f_5(x) = 4x + 4 & -1 \le x \le 0 \\ f_6(x) = -4x + 4 & 0 \le x \le 1 \\ f_7(x) = 3x - 3 & 1 \le x \le 2 \\ f_8(x) = x & 3 \le x \le 6 \\ f_9(x) = -3x - 24 & 6 \le x \le 7 \end{cases} \tag{4.15}$$

where $\mathcal{X} := \{x \in \mathbb{R} : -20 \le x \le 20\}$.

By converting f into the equivalent MMPS function in disjunctive form, by means of the Algorithms 4.4, and 4.5, we obtain the following MMPS function:

$$\begin{aligned} g(x) = \max(&\min(f_1, f_3, f_4, f_6, f_9), \min(f_1, f_2, f_4, f_6, f_9), \min(f_2, f_5, f_6, f_9), \\ &\min(f_1, f_2, f_5, f_7, f_9), \min(f_1, f_2, f_5, f_7, f_8), \min(f_2, f_5, f_7, f_8, f_9), \\ &\min(f_2, f_3, f_4, f_6, f_9)) \qquad \forall x \in \mathcal{X} \end{aligned} \tag{4.16}$$

It is clear that the the function g is not in minimal form.

This example is given in the file **PWA_example**.

4.3 Summary

Two strategies for the conversion of a continuous PWA function into the equivalent MMPS one are discussed in this chapter: the Gorokhovik-Zorko strategy and the Ovchinnikov one.

We have also examined how to obtain a reduced realization of the MMPS function obtained by the Gorokhovik strategy.

Moreover, some algorithms for implementing each one of these methods by MATLAB are given.

Chapter 5

From MMPS to continuous PWA functions

We have shown in Chapter 4 two strategies for the conversion of a continuous PWA function into an equivalent MMPS one given in canonical form. In this chapter, we introduce the opposite strategy. In fact we know from Theorem 2.4 that every MMPS function can be converted into an equivalent continuous PWA one. In this thesis we only convert MMPS functions given in canonical form. We give here only the strategy for the conversion from a function in disjunctive form since the conversion from a function in conjunctive form is quite similar, with proper modifications. The strategy for obtaining the continuous PWA function from an MMPS function in disjunctive form has been implemented in the code **mmps2pwaDisj**, whereas the strategy for obtaining it from an MMPS function in conjunctive form has been implemented in the code **mmps2pwaConj**. Both routines can be launched by the code **mmps2pwa**. All these codes are given in Appendix A.

5.1 Strategy

Let us consider an MMPS function $y : \mathcal{X} \longrightarrow \mathbb{R}$, with $\mathcal{X} \subseteq \mathbb{R}^n$, given in its disjunctive canonical form

$$y = \max_{j=1,\dots,l} \min_{i \in I_j} f_i \tag{5.1}$$

where $f_i = \alpha_i^T x + \beta_i$ for $i = 1, \dots, P$ and $I_j \subseteq \{1, \dots, P\}$ for $j = 1, \dots, l$. This MMPS function can be either in the minimal form or in a non minimal one. We know from Theorem 2.4 that there surely exists an equivalent continuous PWA function f in which (5.1) can be transformed. We have therefore developed the MATLAB code **mmps2pwaDisj**, in which a strategy for doing this is implemented.

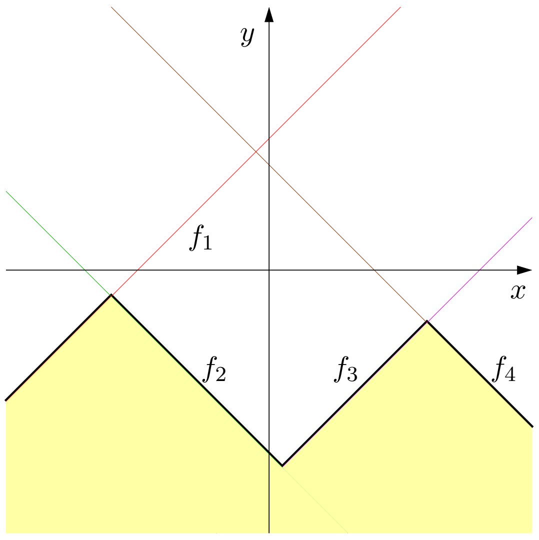

Figure 5.1: *Computation of* hyp y *as given in* (5.2)

At the beginning of the code, we have to be sure that the MMPS function in input respects the notation (2.21). The MATLAB code **testMMPS** verifies this, returning an error message if some of the fields do not respect the notation (2.21). Moreover, the domain is added by default if the fields A and b are not present.

Now we can start describing the strategy. First of all we have to compute the *hypograph* of each *min* term of y. We already know (see Section 4.1) that the *hypograph* of each *min* term can be computed as the intersection of the *hypographs* of its arguments. Next we can compute the *hypograph* of y as the *union* of the *hypographs* of all *min* terms. So we have

$$\mathrm{hyp}(y) = \cup_{j=1,\dots,l} \cap_{i \in I_j} \mathrm{hyp}(f_i) \tag{5.2}$$

as we can see in Figure 5.1.

Therefore, for each affine component f_i for $i = 1, \dots, P$ we have to compute:

1. The set of polyhedra given by the *set difference* between the *hypograph* of y and the *hypograph* of f_i. We can denote it as F_i and so we have

$$F_i = \mathrm{hyp}(y) \setminus \mathrm{hyp}(f_i) = \{x \in \mathcal{X} \times \mathbb{R} : x \in \mathrm{hyp}(y),\, x \notin \mathrm{hyp}(f_i)\} \tag{5.3}$$

2. The set of polyhedra given by the *set difference* between the *hypograph* of f_i and the *hypograph* of y. We can denote it as E_i and so we have

$$E_i = \mathrm{hyp}(f_i) \setminus \mathrm{hyp}(y) = \{x \in \mathcal{X} \times \mathbb{R} : x \in \mathrm{hyp}(f_i),\, x \notin \mathrm{hyp}(y)\} \tag{5.4}$$

3. The polyhedral set given by the *projection* of the set $F_i \cup E_i$ on $\mathcal{X}$. We can denote it as γ_i.

4. The *set difference* between $\mathcal{X}$ and γ_i, that we can denote as $\mathcal{D}_i$. We have

$$\mathcal{D}_i = \mathcal{X} \setminus \gamma_i = \{x \in \mathbb{R}^n : x \in \mathcal{X},\, x \notin \gamma_i\} \tag{5.5}$$

Until now we have obtained some polyhedral sets $\mathcal{D}_i$, for $i = 1, \dots, P$. Every set $\mathcal{D}_i$ is defined as $\mathcal{D}_i = \cup_{j=1,\dots,d_i} \mathcal{D}_{ij}$, since there are d_i polyhedra in $\mathcal{D}_i$. Let us now remove from every set $\mathcal{D}_i$ all polyhedra $\mathcal{D}_{ij}$ such that $\operatorname{int} \mathcal{D}_{ij} = \emptyset$. We therefore have

$$\mathcal{X}_i := \{\mathcal{D}_{ij} \in \mathcal{D}_i : \operatorname{int} \mathcal{D}_{ij} \neq \emptyset\} \tag{5.6}$$

So, we have obtained some polyhedral sets $\mathcal{X}_i$, for $i = 1, \dots, P$, such that $\mathcal{X}_i = \cup_{j=1,\dots,x_i} \mathcal{X}_{ij}$, where $x_i \leq d_i$ is the number of polyhedra in $\mathcal{X}_i$. We are now in the position to give the following proposition:

Proposition 5.1. *Every polyhedral set $\mathcal{X}_i$, for $i = 1, \dots, P$, is the set of polyhedral regions in which every affine term f_i is defined in the continuous PWA function f equivalent to* (5.1)*.*

Proof 5.1. Let us consider the polyhedral set F_i: we can note that its projection in $\mathcal{X}$ is not empty in the regions of $\mathcal{X}$ in which $\operatorname{hyp} y \supset \operatorname{hyp} f_i$ and therefore $y > f_i$. If we indeed consider the set E_i, its projection in $\mathcal{X}$ is not empty in the regions of $\mathcal{X}$ in which $\operatorname{hyp} y \subset \operatorname{hyp} f_i$ and therefore $y < f_i$. Then, for these two sets of polyhedral regions of $\mathcal{X}$, whose union has been defined as γ_i, we have either $y > f_i$ or $y < f_i$, and so $y \neq f_i$. Therefore f_i is not the affine component defined on the points of the polyhedral set γ_i and for obtaining the polyhedral regions in which f_i is defined we can start removing the set γ_i from $\mathcal{X}$, and we can do this computing $\mathcal{D}_i = \mathcal{X} \setminus \gamma_i$. For all the points in $\mathcal{D}_i$, we have $\operatorname{hyp} y = \operatorname{hyp} f_i$ and therefore $y = f_i$. So the points in $\mathcal{D}_i$ are the candidates for belonging to the polyhedral regions in which f_i is defined. Let us now consider all polyhedra $\mathcal{D}_{ij} \in \mathcal{D}_i$ such that $\operatorname{int} \mathcal{D}_{ij} = \emptyset$. These polyhedra cannot belong to the polyhedral set $\mathcal{X}_i$ in which f_i is defined because their *interior* is empty and therefore they cannot be included in a polyhedral partition (see Definition 2.2) of $\mathcal{X}$. However, since the function f must be continuous and PWA, for every polyhedron $\mathcal{D}_{ij}$ such that $\operatorname{int} \mathcal{D}_{ij} = \emptyset$, there surely exists a polyhedron $\mathcal{D}_{i'j'} \in \mathcal{D}'_i$, with $i' \neq i$, such that $\operatorname{int} \mathcal{D}_{i'j'} \neq \emptyset$ and $\mathcal{D}_{ij} \subset \mathcal{D}_{i'j'}$, and so there also exists an affine component f'_i defined on the points of $\mathcal{D}_{ij}$. This ends the proof. □

We can obtain a reduced realization of f doing the merging on the regions of $\mathcal{X}_i$.

If $\mathcal{X}_i = \emptyset$, and so $x_i = 0$, it means that there does not exist any polyhedron $\mathcal{X}_{ij} \in \mathcal{X}_i$, and so the affine component f_i is not necessary for defining the function f. This can happen only if the MMPS function y is not in the minimal form. Indeed, no unnecessary components can appear in an MMPS function in the minimal form.

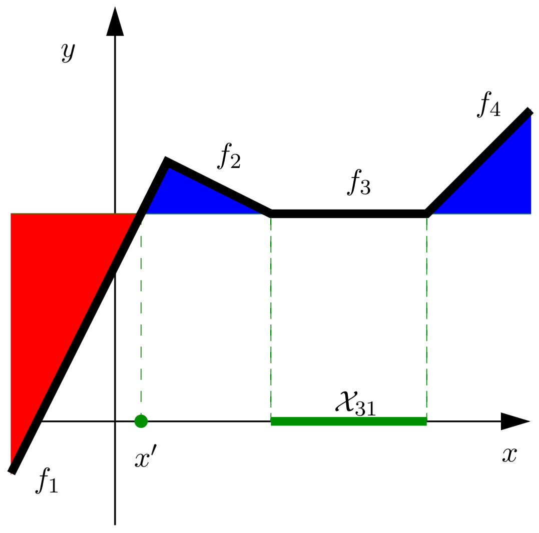

Figure 5.2: *An example of the application of the strategy.*

Example 5.1. In Figure 5.2 an example of the application of this strategy is shown. Only the computation of $\mathcal{X}_3$ is illustrated in the figure. We can see that in the point $x' \in \mathcal{D}_3$ we have $f_3(x') = y(x')$. However, the point x' is not a full dimensional polyhedron in $\mathcal{D}_3$, and so it is not included in $\mathcal{X}_3$. It is also easy to note that $x' \in \mathcal{X}_1$ since it is a subset of a full dimensional polyhedron in $\mathcal{D}_1$.

The strategy for obtaining the sets of polyhedral regions $\mathcal{X}_i$ in which every affine component f_i for $i = 1, \ldots, P$ is defined can be implemented in MATLAB by means of the following algorithm:

Algorithm 5.1.

```
1. Let $\mathcal{X}$ be the domain of $y$;
2. **for** each $i = 1, \ldots, P$ **do**
3. **let** $F_i = \text{hyp}(y) \setminus \text{hyp}(f_i)$;
4. **let** $E_i = \text{hyp}(f_i) \setminus \text{hyp}(y)$;
5. **let** $\gamma_i$ the polyhedral set given by the *projection* of $F_i \cup E_i$ on $\mathcal{X}$;
6. **let** $\mathcal{X}_i = \mathcal{X} \setminus \gamma_i$;
7. merge the polyhedra in $\mathcal{X}_i$;
8. **endfor**;
```

9. **return** $\mathcal{X}_i$ for each $i = 1, \dots, P$.

In the MPT toolbox [13], the polyhedra that are not fully dimensional (and so with an empty *interior*) are considered as empty. Then, by Step 6 of Algorithm 5.1, we directly obtain the polyhedral sets $\mathcal{X}_i$.

5.1.1 Example

Let us consider the MMPS function $y : \mathcal{X} \longrightarrow \mathbb{R}$, as given in (4.12): so we have:

$$y(x) = \max(\min(f_1, f_4), \min(f_1, f_7), \min(f_5, f_6), \min(f_2, f_3, f_4), \min(f_7, f_8, f_9)) \tag{5.7}$$

where $\mathcal{X} =: \{x \in \mathbb{R} : -20 \leq x \leq 20\}$, $f_1(x) = 3$, $f_2(x) = 3x + 24$, $f_3(x) = -x$, $f_4(x) = -3x - 3$, $f_5(x) = 4x + 4$, $f_6(x) = -4x + 4$, $f_7(x) = 3x - 3$, $f_8(x) = x$ and $f_9(x) = -3x - 24$.

If we convert this function into an equivalent continuous PWA, we exactly obtain the function f given in (4.10) and therefore we have:

$$f(x) = \begin{cases} f_1(x) = 3 & -20 \leq x \leq -7 \textit{ or } -3 \leq x \leq -2 \textit{ or} \\ & 2 \leq x \leq 3 \textit{ or } 7 \leq x \leq 20 \\ f_2(x) = 3x + 24 & -7 \leq x \leq -6 \\ f_3(x) = -x & -6 \leq x \leq -3 \\ f_4(x) = -3x - 3 & -2 \leq x \leq -1 \\ f_5(x) = 4x + 4 & -1 \leq x \leq 0 \\ f_6(x) = -4x + 4 & 0 \leq x \leq 1 \\ f_7(x) = 3x - 3 & 1 \leq x \leq 2 \\ f_8(x) = x & 3 \leq x \leq 6 \\ f_9(x) = -3x - 24 & 6 \leq x \leq 7 \end{cases} \tag{5.8}$$

This example is given in the file **MMPS_example**.

5.2 Summary

In this chapter, we have described a strategy for the conversion between a MMPS function into an equivalent continuous PWA one.

The algorithm for the implementation of this strategy by MATLAB is also given.

Chapter 6

Tests and results

Now, we can finally test the codes illustrated in the previous chapters, in order to judge the obtained results and so the strategies and the performances of the codes.

For testing this codes, we have first generated random continuous PWA and MMPS functions given in canonical form. In every test, the generated functions is converted twice, and in different ways. At the end, the complete equivalence between the generated function and the functions obtained by the conversions is tested.

Therefore, we analyse the results of these tests.

6.1 Tests description

In order to analyse the performances of the codes, we have implemented three different tests. For implementing them, we must be able to generate random functions (continuous PWA and MMPS) and to check whether these functions are equivalent. In the following subsections we describe the strategies used for doing this. In the last subsection we give the algorithms used for the implementation of the tests. All related MATLAB codes are given in Appendix A.

6.1.1 Creation of a continuous PWA function as the solution of a Multi-Parametric Linear Programming Problem

For the realization of two of the tests, we must be able to generate some continuous PWA functions. This is not an easy work, since the continuity on any boundary of the regions must be respected, and therefore, especially for dimensions higher than 1, the task is prohibitive. So, we have to find an alternative way for generating continuous PWA functions of any dimension easily. Let us start considering the following problem:

$$\min_{z} \quad V = h^T z \tag{6.1a}$$

$$\text{s. t.} \quad Gz \leq S + Fx \tag{6.1b}$$

where $x \in \mathbb{R}^n$ is the vector of parameters, $z \in \mathbb{R}^m$ is the optimization vector, $V \in \mathbb{R}$ is the objective function, $h \in \mathbb{R}^m$, $G \in \mathbb{R}^{f \times m}$, $S \in \mathbb{R}^f$, and $F \in \mathbb{R}^{f \times n}$.

Problem (6.1) is known as multi-parametric linear programming. Suppose that we want to compute the solution of (6.1) in a convex polyhedron K such that

$$K := \{x \in \mathbb{R}^n : Ax \leq b\}$$

We denote as $K^\star \subseteq K$ the set of parameters such that problem (6.1) has a feasible solution and this solution is unique. For any $\bar{x} \in K^\star$, let $z^\star(\bar{x})$ denote the optimal value of the optimizer z for $x = \bar{x}$, and therefore let $z^\star : K^\star \longrightarrow \mathbb{R}^m$ denote the function expressing the dependance on x of the optimal value of the optimizer.

We are now in the position to give the following theorem:

Theorem 6.1. *[8] If there exists a solution of* (6.1) *and this solution is unique, then the related set of feasible parameters $K^\star$ is convex and the related optimizer $z^\star(x)$ is continuous and PWA.*

Then, by starting from problem (6.1) and thanks to Theorem 6.1, we have found a method for generating continuous PWA functions.

Now, we must implement this strategy in a MATLAB code in order to generate the continuous PWA functions that will be used for the simulations. We have done this in the MATLAB code **continuousPWAgenerator**.

In this code we have made some choices. Here, we explain the most relevant ones only. For n, a value between 1 and 3 is chosen for avoiding large computational times. This is the same reason why we have used for f a value between 1 and 6 and for m a value between 1 and 5. In fact, when m grows also the number of regions in $K^\star$ increases [1], and therefore the complexity too. After the choice of the other parameters (f, h, G, S, F, A and b), the function **mpt_mplp** [13] is launched. This code solves the multi-parametric linear programming for the chosen parameters h, G, S, F, A, b. We know from Theorem 6.1 that the optimizer function returned as output by the code **mpt_mplp** must be continuous and PWA, and therefore we can use it in the simulations, after having converted it in the notation given in (2.20). We have also decided not to use the returned functions for which the number of different affine terms is not included between 3 and 10, for avoiding the use of either too simple functions or too complex ones in the tests.

6.1.2 Creation of a random MMPS function in canonical form

For one of the tests, we need to create some MMPS function in canonical form. We have done this in the MATLAB code **MMPSgenerator**.

In this case, the implementation of the strategy is much easier than the implementation of the strategy for the creation of continuous PWA functions. Suppose that we want to generate an MMPS function $y : \mathbb{R}^n \longrightarrow \mathbb{R}$ in the form

$$\min_{j=1,\dots,l} \max_{i \in I_j} (\alpha_i^T x + \beta_i)$$

where $\alpha_i \in \mathbb{R}^n$, $\beta_i \in \mathbb{R}$ and $I_j \subseteq \{1, \dots, P\}$ for $j = 1, \dots, l$. For doing this, we only have to generate P different affine terms and l max terms, with the notation given in (2.21). For avoiding to have a large complexity, we have chosen $1 \leq n \leq 3$, $3 \leq P \leq 10$ and $1 \leq l \leq 10$. We have also decided not to specify the domain of the function here, since it will be added in the code **testMMPS** (see Appendix A) by default, as we know from Section 5.1. The MMPS function created is not always in the minimal form and moreover, most of the affine components could be useless for the definition of the function, since they might be defined in no regions.

6.1.3 Checking for equivalence between two continuous PWA functions

In two tests we need to know whether or not two continuous PWA functions are equivalent. This is done in the MATLAB code **isequalPWA**.

We now describe the strategy used for this aim. Assume that we have two continuous PWA functions $f : \mathcal{X} \longrightarrow \mathbb{R}$ and $g : \mathcal{X} \longrightarrow \mathbb{R}$, where $\mathcal{X} \subset \mathbb{R}^n$ is a polyhedron. The procedure used to prove that $f \equiv g$ is the following:

- Let $\{\mathcal{X}_{f,ij}\}_{i \in \{1,\dots,N_f\}, j \in \{1,\dots,d_{f_i}\}}$ and $\{\mathcal{X}_{g,kl}\}_{k \in \{1,\dots,N_g\}, l \in \{1,\dots,d_{g_k}\}}$ the polyhedral partitions that define f and g respectively.
- Let H be a hyperbox in $\mathbb{R}^n$ such that all vertices of the polyhedra in $\{X_{f,ij}\}$ and $\{\mathcal{X}_{g,kl}\}$ are contained in the *interior* of H.
- Determine a new polyhedral partition $\{\mathcal{X}_m\}_{m \in \{1,\dots,N\}}$ obtained by considering all possible full-dimensional intersections between the polyhedra H, $X_{f,ij}$ and $\mathcal{X}_{g,kl}$ for $i \in \{1, \dots, N_f\}$, $j \in \{1, \dots, d_{f_i}\}$, $k \in \{1, \dots, N_g\}$ and $l \in \{1, \dots, d_{g_k}\}$.
- Consider now the set of all the vertices of the polyhedra in $\{\mathcal{X}_m\}$: we denote this set as V.

- If $f(v) = g(v)$ for all vertices $v \in V$, then $f \equiv g$.

Indeed, since every polyhedron $\mathcal{X}_m$ is full dimensional, it is defined by at least $n+1$ distinct vertices $v_{m,1}, \ldots, v_{m,n+1}$, and therefore, if $f(v_{m,i}) = g(v_{m,i})$ for $i = 1, \ldots, n+1$, then this means that $f(x) \equiv g(x)$ for all $x \in \mathcal{X}_m$ since f and g are affine on $\mathcal{X}_m$ and since an affine function is uniquely defined by specifying the function in distinct $n+1$ points. Moreover, if this holds for all $m \in \{1, \ldots, N\}$ we get $f(x) = g(x)$ for all $x \in \mathcal{X}$.

We can now give the Algorithm used in MATLAB for implementing this strategy

Algorithm 6.1.

```
1. Let $\{\mathcal{X}_{f,ij}\}$ and $\{\mathcal{X}_{g,kl}\}$ the polyhedral partitions in which $f$ and $g$ are defined,
   and $R$ an empty set;
2. for each pair of polyhedra $P_f \in \{\mathcal{X}_{f,ij}\}$ and $P_g \in \{\mathcal{X}_{g,kl}\}$ do;
3. if $M = \mathcal{P}_f \cap P_g \cap H$ is full-dimensional, insert $M$ in $R$;
4. endfor;
5. let $V$ the set of all vertices of the polyhedra in $R$;
6. if for each vertex $v \in V$ we have $f(v) = g(v)$, then return true, else return
   false.
```

6.1.4 Checking for equivalence between two MMPS functions in canonical form

In all tests we check whether two MMPS functions given in canonical form are equivalent. For doing this we have decided to compare the *hypographs* of the two functions and, if they are equal, we can say that the functions are equivalent.

We have implemented this strategy in the MATLAB code **isequalMMPS**.

6.1.5 Checking for equivalence between a continuous PWA function and an MMPS function in canonical form

The comparison between a continuous PWA function and an MMPS one in canonical form is required in two tests. In the codes **isequalPWA2MMPSovc** and **isequalPWA2MMPSgor** we have implemented in two different ways the strategy for checking whether these two functions are equal.

By the code **isequalPWA2MMPSovc**, first the MMPS function in the input is converted into the equivalent continuous PWA one and it is checked whether the latter is equal to the continuous PWA in the input. Next, the continuous PWA function is translated into the equivalent MMPS one (by the Ovchinnikov strategy and in the same canonical form of the MMPS function in input), and then compared with the MMPS function in the input. If in both cases the result of the test is positive, then we can say that the two functions in input are equivalent.

The code **isequalPWA2MMPSgor** is almost equal, except that the conversion from the continuous PWA function into the equivalent MMPS one is made by the Gorokhovik-Zorko strategy.

6.1.6 Tests

The first test has been implemented in the code **MMPSsimulations** by the following algorithm:

Algorithm 6.2. First test

1. Let y be a random MMPS function;
2. Convert y into the equivalent continuous PWA function. Denote this function as p;
3. Convert p into the equivalent MMPS one in the canonical form of y by the Gorokhovik-Zorko strategy. Denote this function as m_g;
4. Convert p into the equivalent MMPS function in the canonical form of y by the Ovchinnikov strategy. Denote this function as m_o;
5. **if** $y \equiv m_g$, $y \equiv m_o$ and $p \equiv m_o$, **then return** true, **else return** false.

The equivalence between p and m_o is checked by the code **isequalPWA2MMPSgor**. We can note that we indirectly check for the equivalence between m_g and m_o too (in fact, in one step of the code **isequalPWA2MMPSgor**, p is converted into the equivalent MMPS one by the Gorokhovik-Zorko strategy. The obtained function is m_g, and is compared with m_o).

This test is shown in Figure 6.1.

The second test has been implemented in the code **PWAsimulationsGor** by the following algorithm:

Algorithm 6.3. Second test

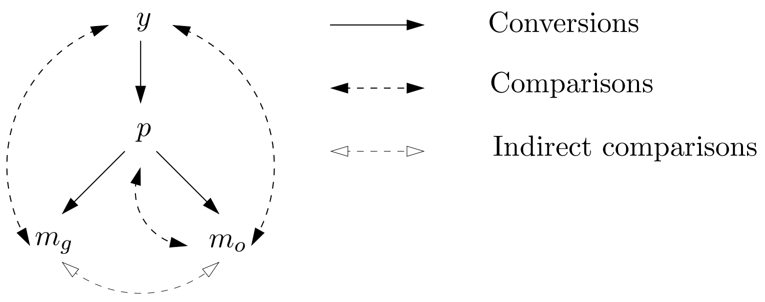

Figure 6.1: *Graphical representation of the first test*

1. Let f be a random continuous PWA function;
2. Convert f into the equivalent MMPS function in conjunctive form by the Gorokhovik-Zorko strategy. Denote this function as m_c;
3. Convert f into the equivalent MMPS function in disjunctive form by the Gorokhovik-Zorko strategy. Denote this function as m_d;
4. Convert m_c into the equivalent continuous PWA function. Denote this function as p_c;
5. Convert m_d into the equivalent continuous PWA function. Denote this function as p_d;
6. **if** $f \equiv p_c$, $p_d \equiv m_c$ and $m_c \equiv m_d$, **then return** true, **else return** false.

The equivalence $p_d \equiv m_c$ is checked by the code **isequalPWA2MMPSgor**.

The third test has been implemented in the code **PWAsimulationsOvc** by the following algorithm:

Algorithm 6.4. Third test

1. Let f be a random continuous PWA function;
2. Convert f into the equivalent MMPS function in conjunctive form by the Ovchinnikov strategy. Denote this function as m_c;
3. Convert f into the equivalent MMPS function in disjunctive form by the Ovchinnikov strategy. Denote this function as m_d;
4. Convert m_c into the equivalent continuous PWA function. Denote this function as p_c;
5. Convert m_d into the equivalent continuous PWA function. Denote this function as p_d;

$$\begin{tikzcd}
& f \arrow[dl] \arrow[dr] \arrow[ddl, dashed, <->, bend right=60] & \\
m_c \arrow[rr, dashed, <->] \arrow[d] \arrow[drr, dashed, <->] & & m_d \arrow[d] \\
p_c \arrow[rr, dashed, <->, "\triangleleft\ \triangleright" description] & & p_d
\end{tikzcd}$$

$\longrightarrow$ Conversions

$\longleftarrow\!\!-\,-\!\!\longrightarrow$ Comparisons

$\triangleleft\,-\,-\,-\,\triangleright$ Indirect comparisons

Figure 6.2: *Graphical representation of the second and third tests*

6. **if** $f \equiv p_c$, $p_d \equiv m_c$ and $m_c \equiv m_d$, **then return** true, **else return** false.

The equivalence $p_d \equiv m_c$ is checked by the code **isequalPWA2MMPSovc**.

The second and the third tests are shown in Figure 6.2.

As regards these tests, the equivalence between p_c and p_d is also checked, since of the characteristics of the codes **isequalPWA2MMPSgor** and **isequalPWA2MMPSovc** (see Section 6.1.5).

6.2 Results

Let us now analyse the results of the tests, in order to give the main conclusions. For each test, we have realised a table in which the main results are shown as follows: at first, we give the number of functions created (continuous PWA or MMPS depending on the test), the number of functions tested (i.e the functions created for which the whole test has been done), the number of functions discarded (i.e. the functions created but not tested because the code has not worked and has returned an error), the simulation time and the average simulation time for every function tested. In the following part of the table we show the results for each comparison made in the test. Finally, in the last two parts, we give the global results of the test, and also the global results on the basis of the dimension of the function and of the number of different affine terms in the function. We recall that a test is succesful only if all the comparisons in it are succesful.

We can note by the first test (see Table 6.1) that the results are not always good, since we have the 45.12% of failures. However, in this test, we do not have any failure checking the equivalence between y and m_g. We can therefore say that something goes wrong when we use the Ovchinnikov strategy for the conversion of a continuous PWA function into an equivalent MMPS one. This can be easily seen looking at the second (see Table 6.2) and at the third test (see Table 6.3). In the second test, where we only use the Gorokhovik-Zorko strategy for the conversion of the continuous

PWA function generated, we only have the 0.94% of failures, whereas in the third test, where we only use the Ovchinnikov strategy, we have the 52.46% of failures.

If we analyse the results of the third test, we can see that the number of failures grows with the dimension and the number of affine terms of f. Indeed, for the 1-dimensional case we do not have any failure whereas for the 3-dimensional case we only have the 4.4% of successful tests. Moreover, when f has at least 8 affine terms, we have the 100% of failures. We can say that this could be due to numerical problems. In fact, the computational complexity of the tests (and therefore the number of operations that must be executed) grows with the dimension of f and with the number of its affine terms. But why do we have this difference of performances between the two strategies? Let us try to study this. The codes for the Gorokhovik-Zorko conversion makes mainly use of the functions belonging to MPT toolbox [13]. Instead, in the codes for the Ovchinnikov conversion, there are many operations such as additions, multiplications, equality and inequality tests. Therefore, in the second test, since the MPT toolbox has been developing for some years, it is very likely that the numerical problems have already taken into account, in order to decrease their influence. On the contrary, in the third test, the influence of numerical problems has not been adequately studied yet.

Numerical problems could be also the main reason of the high number of discarded functions in the third test.

Failures of tests could also be due to a programming error somewhere in the code, since a strict debugging has not been done.

However, if we look at the simulations times, we can see that in the second test the average simulation time for every function tested is bigger than the average time required by the third test. So, if the reasons of the failures of tests will be reduced, the Ovchinnikov strategy could become a valid alternative of the Gorokhovik-Zorko one.

6.2.1 Numerical issues

We have just seen that numerical problems should be the main reason of unsuccessful tests and discarded functions. As we can see from the codes given in Appendix A and especially from the codes for the conversion of a PWA function into the equivalent MMPS one through the Ovchinnikov strategy, there are a lot of equality and inequality comparisons. For the implementation of these comparisons, a tolerance has been considered. In this work, the choice of the optimal value of this tolerance for the different comparisons has not been studied in detail: therefore, in order to obtain better results, this issue must be studied in a thoroughly way.

First Test				
This test is illustrated in Algorithm 6.2 and in Figure 6.1.				
Functions created: 244				
Functions tested: 215				
Functions discarded: 29				
Simulation time: 8:21 hours				
Average simulation time for every function tested: 123.20 s				
Comparison	Successful comparisons	Unsuccessful comparisons	Successful comparisons (%)	Unsuccessful comparisons (%)
$y \equiv m_o$	97	118	45.12%	54.88%
$y \equiv m_g$	215	0	100%	0%
$p \equiv m_o$	97	118	45.12%	54.88%
Dimension of the function	Globally successful tests	Globally unsuccessful tests	Globally successful tests (%)	Globally unsuccessful tests (%)
1	48	0	100%	0%
2	35	45	43.75%	56.25%
3	14	73	16.09%	83.91%
Total	97	118	45.12%	54.88%
Different affine terms in the function	Globally successful tests	Globally unsuccessful tests	Globally successful tests (%)	Globally unsuccessful tests (%)
3	16	1	94.12%	5.88%
4	29	2	93.55%	6.45%
5	12	15	44.44%	55.56%
6	9	14	39.13%	60.87%
7	8	17	32%	68%
8	11	27	28.95%	71.05%
9	8	26	23.53%	76.47%
10	4	16	20%	80%
Total	97	118	45.12%	54.88%

Table 6.1: *Results of the first test.*

Second Test
This test is illustrated in Algorithm 6.3 and in Figure 6.2.
Functions created: 752
Functions tested: 744
Functions discarded: 8
Simulation time: 27:50 hours
Average simulation time for every function tested: 134.68 s

Comparison	Successful comparisons	Unsuccessful comparisons	Successful comparisons (%)	Unsuccessful comparisons (%)
$f \equiv p_c$	740	4	99.46%	0.54%
$p_d \equiv m_c$	741	3	99.60%	0.4%
$m_d \equiv m_c$	744	0	100%	0%

Dimension of the function	Globally successful tests	Globally unsuccessful tests	Globally successful tests (%)	Globally unsuccessful tests (%)
1	55	0	100%	0%
2	316	0	100%	0%
3	366	7	98.12%	1.88%
Total	737	7	99.06%	0.94%

Different affine terms in the function	Globally successful tests	Globally unsuccessful tests	Globally successful tests (%)	Globally unsuccessful tests (%)
3	283	1	99.65%	0.35%
4	152	0	100%	0%
5	99	1	99%	1%
6	78	0	100%	0%
7	44	1	97.78%	2.22%
8	34	0	100%	0%
9	22	1	95.65%	4.35%
10	25	3	89.29%	10.71%
Total	737	7	99.06%	0.94%

Table 6.2: *Results of the second test.*

Third Test				
This test is illustrated in Algorithm 6.4 and in Figure 6.2.				
Functions created: 482 Functions tested: 345 Functions discarded: 137 Simulation time: 7:40 hours Average simulation time for every function tested: 80 s				
Comparison	Successful comparisons	Unsuccessful comparisons	Successful comparisons (%)	Unsuccessful comparisons (%)
$f \equiv p_c$	194	151	56.23%	43.77%
$p_d \equiv m_c$	165	180	47.83%	52.17%
$m_d \equiv m_c$	184	161	53.33%	46.67%
Dimension of the function	Globally successful tests	Globally unsuccessful tests	Globally successful tests (%)	Globally unsuccessful tests (%)
1	56	0	100%	0%
2	104	94	52.53%	47.47%
3	4	87	4.4%	95.6%
Total	164	181	47.54%	52.46%
Different affine terms in the function	Globally successful tests	Globally unsuccessful tests	Globally successful tests (%)	Globally unsuccessful tests (%)
3	112	61	64.74%	35.26%
4	36	49	42.35%	57.65%
5	11	30	26.83%	73.17%
6	4	18	18.18%	81.82%
7	1	9	10%	90%
8	0	9	0%	100%
9	0	2	0%	100%
10	0	3	0%	100%
Total	164	181	47.54%	52.46%

Table 6.3: *Results of the third test.*

Chapter 7

Conclusions and future research

In this chapter we give the most important conclusions of this thesis and some recommendations for future research.

7.1 Conclusions

At the beginning of this thesis, we have given a definition for hybrid systems and we have defined them as a mixture of interacting time-driven and event-driven dynamics. Next we have described some subclasses of equivalent discrete-time linear hybrid models.

Therefore we have focused on two classes of functions: PWA functions and MMPS functions. By means of these functions we can define PWA and MMPS systems, which are two subclasses of the discrete-time linear hybrid models. We have also dealt with minimal realizations of both kinds of functions: we have seen that for PWA functions there exists a quite efficient algorithm for obtaining it, whereas this is not true for MMPS functions.

It has already been shown by other authors that the classes of continuous PWA functions and MMPS functions are equivalent. Therefore, we have implemented by MATLAB two methods for the conversion from a continuous PWA function into the equivalent MMPS: the Gorokhovik-Zorko strategy and the Ovchinnikov strategy. Next, we have developed and implemented by MATLAB a strategy for the opposite conversion, that is, from an MMPS function (given in canonical form) into the equivalent continuous PWA function.

After the Gorokhovik-Zorko method we have also been able to obtain a reduced realization of the MMPS function returned as output.

At the end of the thesis, we have tested these codes, by making a comparison between the two different strategies implemented for the conversion from a continuous PWA function into the equivalent MMPS one. For doing it we have also used our code for the opposite conversion and other codes for the generation of partially random functions (of both kinds) and for checking if these functions are equivalent.

We can surely say that the results obtained by the Gorokhovik-Zorko strategy are better than the ones obtained by the Ovchinnikov strategy. Almost the 100% of the tests has given a positive result. The few failures of tests could be due to numerical problems. Unfortunately this code is not so efficient, since for higher dimensions and especially for large numbers of affine terms, the code becomes very slow. Indeed, we have seen that the computational complexity grows exponentially with the number of affine terms. On the contrary the conversion obtained by the implementation of the Ovchinnikov strategy is faster, but because of numerical problems the results obtained are not so good. Only the 47% of the tests are positive, and this percentage drastically decreases for higher dimensions and number of affine terms. Moreover, the codes for the conversion do not work for the 28% of the functions that have to be tested.

We can therefore say that the goals of this thesis have been partially reached, even if we still need to improve on the codes, for having better results and efficiency.

7.2 Future research

We have implemented some algorithms for the conversion of a continuous PWA function into the equivalent MMPS one through two different methods: the Gorokhovik-Zorko strategy and the Ovchinnikov strategy.

The algorithm by means of which we have implemented the Gorokhovik-Zorko strategy has given good results, but its efficiency should be increased, since the code becomes too slow when the dimension and the number of affine components of the continuous PWA function are too high.

As regards the implementation of the Ovchinnikov strategy, the related algorithm is quite fast, but the results are not so good as we wanted. This is because the influence of numerical problems is very high, and so the code should be improved in order to reduce this influence, therefore increasing the performances of the code.

After the conversion of a continuous PWA function through the Gorokhovik strategy we are also able to obtain a reduced realization of the equivalent MMPS function obtained: the realization of the latter could be minimal, since a counter-example of this has not been found. Then, it should be proven if the realization obtained is indeed minimal or not.

To make this more general, we can say that some efficient strategies for obtaining a minimal realization of an MMPS function should be developed, by starting both from an arbitrary MMPS function and from an equivalent continuous PWA function.

A strategy for the conversion of an MMPS function in canonical form into the equivalent continuous PWA function has been developed. This strategy should partially be extended, in order to convert an arbitrary MMPS function.

All this problems can be summarized into a more general one: make a MATLAB toolbox to convert any equivalent class of discrete-time linear hybrid models into each other in the most efficient way, in order to use, for each kind of problem, the best technique for solving this.

Appendix A

Matlab codes

In this appendix we give the Matlab codes used for our aims. Most of the functions present in these codes belong to the MPT toolbox [13].

A.1 mmps2pwa

```
function pwa = mmps2pwa(mmps)

%   pwa = mmps2pwa(mmps)
%   Convert a MMPS function into the equivalent continuous PWA function
%   mmps ->  MMPS function that must be converted
%   pwa -> continuous PWA function obtained as output

%   This function calls:
%       mmps2pwaConj
%       mmps2pwaDisj

%   This function is called by:
%       PWAsimulationsGor
%       PWAsimulationsOvc
%       MMPSsimulations

%   A. Frau
%   2/7/07

if isfield(mmps,'form')
    form = mmps.form;
    if strcmp(form,'conj')
        pwa = mmps2pwaConj(mmps);
    elseif strcmp(form,'disj')
```

```
        pwa = mmps2pwaDisj(mmps);
    else
        error('Not valid string for the field "form"');
    end
else
    pwa = mmps2pwaDisj(mmps);   % default conversion
end
```

A.2 mmps2pwaConj

```
function pwa = mmps2pwaConj(mmps)

%   pwa = mmps2pwaConj(mmps)
%   Convert a MMPS function given in conjunctive form into
%       the equivalent continuous PWA function
%   mmps ->  MMPS function that must be converted
%   pwa ->  continuous PWA function obtained as output

%   This function calls:
%       testMMPS

%   This function is called by:
%       mmps2pwa

%   A. Frau
%   1/6/07

%   Last modification   6/7/07

%   Test

mmpsTest = testMMPS(mmps);

alfa = mmpsTest.alfa; beta = mmpsTest.beta;
maxTerms = mmpsTest.terms;
N_affine_terms_old = mmpsTest.N_affine_terms;
dimension = mmpsTest.dimension; domain = mmpsTest.domain;

%   Computation of the epigraph of the MMPS function

[H_domain,K_domain] = double(domain);
H_termEpigraph = {}; K_termEpigraph = {};   termEpigraph = polytope;
for i = 1:N_affine_terms_old
    H_termEpigraph{i} = [H_domain zeros(size(H_domain,1),1); alfa{i} -1];
    K_termEpigraph{i} = [K_domain; -beta{i}];
    %   epigraph of each affine term
    termEpigraph(i) = polytope(H_termEpigraph{i},K_termEpigraph{i});
end

maxTermArrayEpigraphs = {};  maxTermEpigraph = {};   mmpsEpigraph = [];
for i = 1:length(maxTerms)
    maxTermArrayEpigraphs{i} = [];
    %   Let us construct a polyarray for each max term
    for j = 1:length(maxTerms{i})
        %   set of epigraphs of the affine components of every max term
        maxTermArrayEpigraphs{i} = horzcat(maxTermArrayEpigraphs{i},...
            termEpigraph(maxTerms{i}(j)));
    end
    %   epigraph of each max term
    maxTermEpigraph{i} = and(maxTermArrayEpigraphs{i});
    %   epigraph of the MMPS function
    mmpsEpigraph = horzcat(mmpsEpigraph,maxTermEpigraph{i});
end

%   Computation of the regions in which every affine component is defined

P_proj1 = {};    P_proj2 = {};   Regions = {};
H = {};    K = {};
for i = 1:N_affine_terms_old
    %   projection of the set-difference between the epigraph of
    %   the affine term i and the epigraph of the MMPS function
    P_proj1{i} = union(projection((mldivide(termEpigraph(i),...
        mmpsEpigraph)),1:1:dimension));
    %   projection of the set-difference between the epigraph
    %   of the MMPS function and the epigraph of the affine term i
    P_proj2{i} = union(projection((mldivide(mmpsEpigraph,...
        termEpigraph(i))),1:1:dimension));
    %   computation of the set of regions in which the affine term i is
    %   defined
    Regions{i} = merge(mldivide(domain,[P_proj1{i},P_proj2{i}]));
    [H{i},K{i}] = double(Regions{i});
end

%   Elimination of the useless functions, that is, of the functions such
%   that their set of regions is empty

alfa_new = {};  beta_new = {};  H_new = {};  K_new = {};
count = 0;
for i = 1:length(alfa)
    if ~isempty(H{i})
        count = count + 1;
        alfa_new{count} = alfa{i};
        beta_new{count} = beta{i};
        H_new{count} = H{i};
        K_new{count} = K{i};
    end
end

%   Output

pwa.alfa = alfa_new;
pwa.beta = beta_new;
pwa.H = H_new;
pwa.K = K_new;
pwa.dimension = dimension;      %   additional field
pwa.N_affine_terms = count;     %   additional field
```

A.3 mmps2pwaDisj

```
function pwa = mmps2pwaDisj(mmps)

%   pwa = mmps2pwaDisj(mmps)
%   Convert a MMPS function given in disjunctive form into
%   the equivalent continuous PWA function
%   mmps ->  MMPS function that must be converted
%   pwa -> continuous PWA function obtained as output

%   This function calls:
%       testMMPS

%   This function is called by:
%       mmps2pwa

%   A. Frau
%   1/6/07

%   Last modification   6/7/07

%   Test

mmpsTest = testMMPS(mmps);

alfa = mmpsTest.alfa; beta = mmpsTest.beta;
minTerms = mmpsTest.terms;
N_affine_terms_old = mmpsTest.N_affine_terms;
dimension = mmpsTest.dimension; domain = mmpsTest.domain;
```

```
%   Computation of the hypograph of the MMPS function

[H_domain,K_domain] = double(domain);
H_termHypograph = {}; K_termHypograph = {};   termHypograph = polytope;
for i = 1:N_affine_terms_old
    H_termHypograph{i} = [H_domain zeros(size(H_domain,1),1); -alfa{i} 1];
    K_termHypograph{i} = [K_domain; beta{i}];
    %   hypograph of each affine term
    termHypograph(i) = polytope(H_termHypograph{i},K_termHypograph{i});
end

minTermArrayHypographs = {};  minTermHypograph = {};   mmpsHypograph = [];
for i = 1:length(minTerms)
    minTermArrayHypographs{i} = [];
    %   Let us construct a polyarray for each min term
    for j = 1:length(minTerms{i})
        %   set of hypographs of the affine components of every min term
        minTermArrayHypographs{i} = horzcat(minTermArrayHypographs{i},...
            termHypograph(minTerms{i}(j)));
    end
    %   hypograph of each min term
    minTermHypograph{i} = and(minTermArrayHypographs{i});
    %   hypograph of the MMPS function
    mmpsHypograph = horzcat(mmpsHypograph,minTermHypograph{i});
end

%   Computation of the regions in which every affine component is defined

P_proj1 = {};    P_proj2 = {};   Regions = {};
H = {};    K = {};
for i = 1:N_affine_terms_old
    %   projection of the set-difference between the hypograph of the
    %   affine term i and the hypograph of the MMPS function
    P_proj1{i} = union(projection((mldivide(termHypograph(i),...
        mmpsHypograph)),1:1:dimension));
    %   projection of the set-difference between the hypograph of the
    %   MMPS function and the hypograph of the affine term i
    P_proj2{i} = union(projection((mldivide(mmpsHypograph,...
        termHypograph(i))),1:1:dimension));
    %   computation of the set of regions in which the affine term i is
    %   defined
    Regions{i} = merge(mldivide(domain,[P_proj1{i},P_proj2{i}]));
    [H{i},K{i}] = double(Regions{i});
end

%   Elimination of the useless functions, that is, of the functions such
%   that their set of regions is empty
```

```
alfa_new = {};  beta_new = {};  H_new = {};  K_new = {};
count = 0;
for i = 1:length(alfa)
    if ~isempty(H{i})
        count = count + 1;
        alfa_new{count} = alfa{i};
        beta_new{count} = beta{i};
        H_new{count} = H{i};
        K_new{count} = K{i};
    end
end

%   Output

pwa.alfa = alfa_new;
pwa.beta = beta_new;
pwa.H = H_new;
pwa.K = K_new;
pwa.dimension = dimension;      %   additional field
pwa.N_affine_terms = count;     %   additional field
```

A.4 pwa2mmps

```
function mmps = pwa2mmps(pwa,str,str2)

%   mmps = pwa2mmps(pwa,str,str2)
%   Convert a PWA function into the equivalent MMPS function
%   pwa ->  PWA function that must be converted
%   str = 'conj' -> Conjunctive form
%   str = 'disj' -> Disjunctive form
%   str2 = 'Gor' -> Gorokhovik strategy
%   str2 = 'Ovc' -> Ovchinnikov strategy
%   mmps -> MMPS function obtained as output

%   This function calls:
%       pwa2mmpsConjGor
%       pwa2mmpsDisjGor
%       pwa2mmpsConjOvc
%       pwa2mmpsDisjOvc

%   This function is called by:
%       PWAsimulationsOvc
%       PWAsimulationsGor
%       MMPSsimulations

%   A. Frau
%   29/6/07

%   Last modification 13/7/07

if nargin == 1
    str = 'disj';
    str2 = 'Gor';
end

if nargin == 2
    str2 = 'Gor';
end

if ~isa(str,'char')
    error('Second input must be a string');
end

if ~isa(str2,'char')
    error('Third input must be a string');
end

if strcmp(str2,'Gor')
    if strcmp(str,'conj')
        mmps = pwa2mmpsConjGor(pwa);
    elseif strcmp(str,'disj')
        mmps = pwa2mmpsDisjGor(pwa);
    else
        error('Not a valid argument for the second input');
    end
elseif strcmp(str2,'Ovc')
    if strcmp(str,'conj')
        mmps = pwa2mmpsConjOvc(pwa);
    elseif strcmp(str,'disj')
        mmps = pwa2mmpsDisjOvc(pwa);
    else
        error('Not a valid argument for the second input');
    end
else
    error('Not a valid argument for the third input');
end
```

A.5 pwa2mmpsConjGor

```
function mmps = pwa2mmpsConjGor(pwa)

%   mmps = pwa2mmpsConjGor(pwa)
%   Changes a PWA function in the equivalent MMPS function
%   in the conjunctive form through the Gorokhovik-Zorko strategy
%   pwa ->  PWA function that must be converted
%   mmps -> MMPS function obtained as output

%   This function calls:
%       testPWA

%   This function is called by:
%       pwa2mmps

%   A. Frau
%   29/6/07

%   Last modification   6/7/07

%   Test

pwaTest = testPWA(pwa);
alfa = pwaTest.alfa;  beta = pwaTest.beta;
H = pwaTest.H;  K = pwaTest.K;
dimension = pwaTest.dimension;  P = pwaTest.P;
domain = pwaTest.domain;
N_affine_terms = pwaTest.N_affine_terms;

%   Computation of the epigraph of the PWA function

H_nPlus1 = H;   K_nPlus1 = K;
P_nPlus1 = {};  pwaEpigraph = polytope;
b = zeros(2,dimension+1);   b(:,dimension+1) = [1 -1]';
for i = 1:length(H)
    for j = 1:length(H{i})
        H_nPlus1{i}{j} = [H_nPlus1{i}{j} zeros(size(H_nPlus1{i}{j},1),1);...
            alfa{i} -1; b];
        K_nPlus1{i}{j} = [K_nPlus1{i}{j}; -beta{i}; 1e4; 1e4];
        P_nPlus1{i}(j) = reduce(polytope(H_nPlus1{i}{j},K_nPlus1{i}{j}));
    end
    pwaEpigraph = [pwaEpigraph,P_nPlus1{i}];
end
% it is not necessary, but useful for improving efficiency
pwaEpigraph = merge(pwaEpigraph);
```

```
%   Computation of the epigraph of each affine term

[H_domain,K_domain] = double(domain);
H_termEpigraph = {};   K_termEpigraph = {};
termEpigraph = polytope;
for i = 1:length(P)
    H_termEpigraph{i} = [H_domain zeros(size(H_domain,1),1);...
        alfa{i} -1; b];
    K_termEpigraph{i} = [K_domain; -beta{i}; 1e4; 1e4];
    termEpigraph(i) = polytope(H_termEpigraph{i},K_termEpigraph{i});
end

%   Computation of all max terms

all_max_terms = {};    count_all_max_terms = 0;
P_all_max_terms = {};    k = 0;  cont = 0;
P_all_max_terms_intersection = polytope;
while(1)
    if cont == 0
        k = k + 1;
        if k > N_affine_terms
            break   %   exit from while loop
        end
        combination = 1:k;
        cont = 1;
    else
        [combination,cont] = next_comb(combination,N_affine_terms);
        if cont == 0
            continue;   %   next while loop
        end
    end
    P_all_max_terms_array = polytope;
    for i = 1:k
        P_all_max_terms_array = [P_all_max_terms_array termEpigraph(combination)];
    end
    P_all_max_terms_intersection = and(P_all_max_terms_array);
    P_all_max_terms_intersection_ok = le(P_all_max_terms_intersection,pwaEpigraph);
    %   Is this a max term?
    if P_all_max_terms_intersection_ok == 1
        count_all_max_terms = count_all_max_terms + 1;
        all_max_terms{count_all_max_terms} = combination;
        P_all_max_terms{count_all_max_terms} = P_all_max_terms_intersection;
    end
end

%   Reduction
%   Computation of temporary max terms

index_all_max_terms = ones(count_all_max_terms,1);
for i = 1:count_all_max_terms
    for j = 1:count_all_max_terms
        if i~=j
            if index_all_max_terms(i) == 1 && index_all_max_terms(j) == 1
                if ge(P_all_max_terms{i},P_all_max_terms{j}) == 1
                    index_all_max_terms(j) = 0;
                end
            end
        end
    end
end

max_terms_temp = {};
P_max_terms_temp = polytope;
count_max_terms_temp = 0;

for i = 1:count_all_max_terms
    if index_all_max_terms(i) == 1
        count_max_terms_temp = count_max_terms_temp + 1;
        P_max_terms_temp(count_max_terms_temp) = P_all_max_terms{i};
        max_terms_temp{count_max_terms_temp} = all_max_terms{i};
    end
end

%   Reduction
%   Computation of final max terms

%   let us start from terms with a bigger number of terms
P_max_terms_temp = fliplr(P_max_terms_temp);
max_terms_temp = fliplr(max_terms_temp);

index_max_terms_temp = ones(count_max_terms_temp,1);

P_union = {};
for i = 1:count_max_terms_temp
    P_union{i} = polytope;
    for j = 1:count_max_terms_temp
        if i ~= j
            if index_max_terms_temp(j) == 1
                P_union{i} = [P_union{i} P_max_terms_temp(j)];
            end
        end
    end
    if le(P_max_terms_temp(i),P_union{i})
```

```
        index_max_terms_temp(i) = 0;
    end
end

max_terms = {};
count_max_terms = 0;
for i = 1:length(index_max_terms_temp)
    if index_max_terms_temp(i) == 1
        count_max_terms = count_max_terms + 1;
        max_terms{count_max_terms} = max_terms_temp{i};
    end
end

%   Output

mmps.alfa = alfa;   %   affine terms
mmps.beta = beta;   %   affine terms
mmps.terms = max_terms;     %   max terms
mmps.form = 'conj';     %   form
mmps.domain = domain;   %   domain
mmps.dimension = dimension;     %   dimension
mmps.terms_temp = max_terms_temp;   %   temporary max terms -> not useful for tests
mmps.all_terms = all_max_terms; %   all max terms -> not useful for tests
```

A.6 pwa2mmpsDisjGor

```
function mmps = pwa2mmpsDisjGor(pwa)

%   mmps = pwa2mmpsDisjGor(pwa)
%   Converts a PWA function in the equivalent MMPS function
%   in the disjunctive form through the Gorokhovik-Zorko strategy
%   pwa ->  PWA function that must be converted
%   mmps -> MMPS function obtained as output

%   This function calls:
%       testPWA

%   This function is called by:
%       pwa2mmps

%   A. Frau
%   29/6/07

%   Last modification   6/7/07
%   Test

pwaTest = testPWA(pwa);
alfa = pwaTest.alfa;  beta = pwaTest.beta;
H = pwaTest.H;  K = pwaTest.K;
dimension = pwaTest.dimension;  P = pwaTest.P;
domain = pwaTest.domain;
N_affine_terms = pwaTest.N_affine_terms;

%   Computation of the hypograph of the PWA function

H_nPlus1 = H;   K_nPlus1 = K;
P_nPlus1 = {};  pwaHypograph = polytope;
b = zeros(2,dimension+1);   b(:,dimension+1) = [1 -1]';
for i = 1:length(H)
    for j = 1:length(H{i})
        H_nPlus1{i}{j} = [H_nPlus1{i}{j} zeros(size(H_nPlus1{i}{j},1),1);...
            -alfa{i} 1; b];
        K_nPlus1{i}{j} = [K_nPlus1{i}{j}; beta{i}; 1e4; 1e4];
        P_nPlus1{i}(j) = reduce(polytope(H_nPlus1{i}{j},K_nPlus1{i}{j}));
    end
    pwaHypograph = [pwaHypograph,P_nPlus1{i}];
end
% it is not necessary, but useful for improving efficiency
pwaHypograph = merge(pwaHypograph);

%   Computation of the hypograph of each affine term

[H_domain,K_domain] = double(domain);
H_termHypograph = {};   K_termHypograph = {};
termHypograph = polytope;
for i = 1:length(P)
    H_termHypograph{i} = [H_domain zeros(size(H_domain,1),1); -alfa{i} 1; b];
    K_termHypograph{i} = [K_domain; beta{i}; 1e4; 1e4];
    termHypograph(i) = polytope(H_termHypograph{i},K_termHypograph{i});
end

%   Computation of all min terms

all_min_terms = {};    count_all_min_terms = 0;
P_all_min_terms = {};    k = 0;  cont = 0;
P_all_min_terms_intersection = polytope;
while(1)
    if cont == 0
        k = k + 1;
        if k > N_affine_terms
```

```
            break   %   exit from while loop
        end
        combination = 1:k;
        cont = 1;
    else
        [combination,cont] = next_comb(combination,N_affine_terms);
        if cont == 0
            continue;   %   next while loop
        end
    end
    P_all_min_terms_array = polytope;
    for i = 1:k
        P_all_min_terms_array = [P_all_min_terms_array termHypograph(combination)];
    end
    P_all_min_terms_intersection = and(P_all_min_terms_array);
    P_all_min_terms_intersection_ok = le(P_all_min_terms_intersection,pwaHypograph);
    %   Is this a min term?
    if P_all_min_terms_intersection_ok == 1
        count_all_min_terms = count_all_min_terms + 1;
        all_min_terms{count_all_min_terms} = combination;
        P_all_min_terms{count_all_min_terms} = P_all_min_terms_intersection;
    end
end

%   Reduction
%   Computation of temporary min terms

index_all_min_terms = ones(count_all_min_terms,1);
for i = 1:count_all_min_terms
    for j = 1:count_all_min_terms
        if i~=j
            if index_all_min_terms(i) == 1 && index_all_min_terms(j) == 1
                if ge(P_all_min_terms{i},P_all_min_terms{j}) == 1
                    index_all_min_terms(j) = 0;
                end
            end
        end
    end
end

min_terms_temp = {};
P_min_terms_temp = polytope;
count_min_terms_temp = 0;

for i = 1:count_all_min_terms
    if index_all_min_terms(i) == 1
        count_min_terms_temp = count_min_terms_temp + 1;
        P_min_terms_temp(count_min_terms_temp) = P_all_min_terms{i};
        min_terms_temp{count_min_terms_temp} = all_min_terms{i};
    end
end

%   Reduction
%   Computation of final min terms

%   let us start from terms with a bigger number of terms
P_min_terms_temp = fliplr(P_min_terms_temp);
min_terms_temp = fliplr(min_terms_temp);

index_min_terms_temp = ones(count_min_terms_temp,1);

P_union = {};
for i = 1:count_min_terms_temp
    P_union{i} = polytope;
    for j = 1:count_min_terms_temp
        if i ~= j
            if index_min_terms_temp(j) == 1
              %   union of hypographs of all min terms except the i
                P_union{i} = [P_union{i} P_min_terms_temp(j)];
            end
        end
    end
    if le(P_min_terms_temp(i),P_union{i})
        index_min_terms_temp(i) = 0;
    end
end

min_terms = {};
count_min_terms = 0;
for i = 1:length(index_min_terms_temp)
    if index_min_terms_temp(i) == 1
        count_min_terms = count_min_terms + 1;
        min_terms{count_min_terms} = min_terms_temp{i};
    end
end

%   Output

mmps.alfa = alfa;   %   affine terms
mmps.beta = beta;   %   affine terms
mmps.terms = min_terms;     %   min terms
mmps.form = 'disj';     %   form
mmps.domain = domain;   %   domain
mmps.dimension = dimension;     %   dimension
mmps.terms_temp = min_terms_temp;   %   temporary min_terms -> not useful for tests
mmps.all_terms = all_min_terms; %   all min terms -> not useful for tests
```

A.7 pwa2mmpsConjOvc

```
function mmps = pwa2mmpsConjOvc(pwa)

%   mmps = pwa2mmpsConjOvc(pwa)
%   Converts a PWA function in the equivalent MMPS function in
%   the conjunctive form through the Ovchinnikov strategy
%   pwa ->  PWA function that must be converted
%   mmps -> MMPS function obtained as output

%   This function calls:
%       testPWA
%       evalPWA

%   This function is called by
%       pwa2mmps

%   A. Frau
%   20/7/07

%   Last modification   25/5/07
%   Last modification   29/5/07
%   Last modification   30/5/07
%   Last modification   12/7/07

%   Test

pwaTest = testPWA(pwa);

alfa = pwaTest.alfa; beta = pwaTest.beta;
H = pwaTest.H; K = pwaTest.K;
N_affine_terms = pwaTest.N_affine_terms; dimension = pwaTest.dimension;
P = pwaTest.P; domain = pwaTest.domain;

%   Computation of vertices

V = {}; R = {}; adjV = {};
for i = 1:length(P)
    for j = 1:length(P{i})
        %   the vertices are already stored in polytope structure,
        %   so the computation is very fast
        [V{i}{j},R{i}{j},P{i}(j),adjV{i}{j}] = extreme(P{i}(j));
    end
end

%   Computation of adjacent regions in which
%   pwa = min(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))

adj_vertices = {};  adj_regions = [];   min_regions = [];
for i = 1:length(P)
    for j = 1:length(P)
        if i<j  % because otherwise is redundant (i > j) or wrong (i = j)
            for k = 1:length(P{i})
                for l = 1:length(P{j})
                    adj_vertices{i,j,k,l} = [];
                    for m = 1:size(V{i}{k},1)
                        for n = 1:size(V{j}{l},1)
                            if all(abs(V{i}{k}(m,:)-V{j}{l}(n,:)) <= 1e-10)
                                %   vertices in common between
                                %       P{i}(k) and P{j}(l)
                                adj_vertices{i,j,k,l} = ...
                                    [adj_vertices{i,j,k,l}; V{i}{k}(m,:)];
                            end
                        end
                    end
                    if isempty(adj_vertices{i,j,k,l})
                        %   no adjacent vertices between P{i}(k) and
                        %   P{j}(l)
                        adj_regions(i,j,k,l) = 0;
                        min_regions(i,j,k,l) = 0;
                    elseif size(adj_vertices{i,j,k,l},1) == dimension
                        %   P{i}(k) and P{j}(l) are adjacent
                        adj_regions(i,j,k,l) = 1;
                        %   comparison in all vertex of P{i}{k} and in all
                        %   vertex of P{j}{l}
                        if  all(alfa{i}*V{i}{k}'+beta{i}-...
                                (alfa{j}*V{i}{k}'+beta{j}) <= 1e-9) &&...
                            all(alfa{j}*V{j}{l}'+beta{j}-...
                            (alfa{i}*V{j}{l}'+beta{i}) <= 1e-9)
                            %   pwa = min(pwa_i,pwa_j)
                            %       on union(P{i}(k),P{j}(l))
                            min_regions(i,j,k,l) = 1;
                        else
                            %   pwa != min(pwa_i,pwa_j)
                            %       on union(P{i}(k),P{j}(l))
                            min_regions(i,j,k,l) = 0;
                        end
                    else
```

```
                        adj_regions(i,j,k,l) = 0;
                        min_regions(i,j,k,l) = 0;
                    end
                end
            end
        end
    end
end

min_array = [];
for i = 1:length(P)
    for j = 1:length(P)
        if i<j
            for k = 1:length(P{i})
                for l = 1:length(P{j})
                    if min_regions(i,j,k,l) == 1
                        % [i k j l] belong to min_array if pwa =
                        % min(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))
                        min_array = [min_array;i k j l];
                    end
                end
            end
        end
    end
end

%   Some outputs

mmps.alfa = alfa;
mmps.beta = beta;
mmps.domain = domain;
mmps.form = 'conj';

if isempty(min_array)
    %   The function is convex if there does not exist a pair of
    %   regions such that pwa = min(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))
    mmps.terms = {1:N_affine_terms};
    return
end

%   Computation of the hyperplanes that split the pairs of regions above
%   computed, and therefore computation of the constraint in common
%   between each pair of adjacent regions in
%   which pwa = min(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))

min_constrH = [];   min_constrK = [];
index = 1;  count_min_constr = 0;
```

```
for i = 1:size(min_array,1)
    for j = 1:size(H{min_array(i,1)}{min_array(i,2)},1)
        for k = 1:size(H{min_array(i,3)}{min_array(i,4)},1)
            if all(abs(H{min_array(i,1)}{min_array(i,2)}(j,:) + ...
                    H{min_array(i,3)}{min_array(i,4)}(k,:)) <= 1e-7) &&...
               all(abs(K{min_array(i,1)}{min_array(i,2)}(j) + ...
                    K{min_array(i,3)}{min_array(i,4)}(k)) <= 1e-7)
                %   let us verify if the constraint is already in min_constr
                for m = 1:count_min_constr
                    if (all(abs(min_constrH(m,:) - ...
                        H{min_array(i,1)}{min_array(i,2)}(j,:)) <= 1e-7) &&...
                        all(abs(min_constrK(m,:) - ...
                        K{min_array(i,1)}{min_array(i,2)}(j,:)) <= 1e-7)) ||...
                       (all(abs(min_constrH(m,:) + ...
                        H{min_array(i,1)}{min_array(i,2)}(j,:)) <= 1e-7) &&...
                        all(abs(min_constrK(m,:) + ...
                        K{min_array(i,1)}{min_array(i,2)}(j,:)) <= 1e-7))
                        index = 0;
                        break
                    end
                end
                if (index == 1)
                    min_constrH = ...
                        [min_constrH;H{min_array(i,1)}{min_array(i,2)}(j,:);];
                    min_constrK = ...
                        [min_constrK;K{min_array(i,1)}{min_array(i,2)}(j,:);];
                    count_min_constr = count_min_constr + 1;
                end
                index = 1;
            end
        end
    end
end

%   Computation of all regions in which we compute if pwa_i <= pwa

%   all possible combinations of 0 and 1
comb = makebits(size(min_constrH,1));

RegionsTempH = {};   RegionsTempK = {};
RegionsTemp = polytope;    RegionsTemp2 = polytope;
count = 0;
for i = 1:size(comb,1)
    RegionsTempH{i} = [];   RegionsTempK{i} = [];
    for j = 1:size(comb,2)
        if comb(i,j) == 1
            RegionsTempH{i} = [RegionsTempH{i};min_constrH(j,:);];
            RegionsTempK{i} = [RegionsTempK{i};min_constrK(j,:);];
        else
            RegionsTempH{i} = [RegionsTempH{i};-min_constrH(j,:);];
            RegionsTempK{i} = [RegionsTempK{i};-min_constrK(j,:);];
        end
    end
    RegionsTemp = polytope(RegionsTempH{i},RegionsTempK{i});
    if isfulldim(RegionsTemp)
        RegionsTempInters = intersect(RegionsTemp,domain);
        if isfulldim(RegionsTempInters);
            count = count + 1;
            RegionsTemp2(count) = RegionsTempInters;
        end
    end
end

%   The union of the temporary regions obtained must be equal to the domain

if union(RegionsTemp2) ~= domain
    error(['The union of the temporary regions',...
        ' obtained must be equal to the domain']);
end

%   Elimination of identical regions, they must appear only once

identical = ones(1,length(RegionsTemp2));
for i = 1:length(RegionsTemp2)
    for j = i+1:length(RegionsTemp2)
        if RegionsTemp2(i) == RegionsTemp2(j)
            identical(j) = 0;
        end
    end
end

Regions = polytope;   count_Regions = 0;
for i = 1:length(RegionsTemp2)
    if identical(i) == 1
        count_Regions = count_Regions + 1;
        Regions(count_Regions) = RegionsTemp2(i);
    end
end

%   The union of the regions obtained must be equal to the domain

if union(Regions) ~= domain
    error(['The union of the regions',...
        ' obtained must be equal to the domain']);
```

```
end

%   For each region in which the test must be done, we find all vertices of
%   the regions of pwa that lies in that region, and an affine component
%   defined in this vertex

VV_Regions = {};
for l = 1:length(Regions)
    VV_Regions{l} = [];
    for i = 1:length(V)
        for j = 1:length(V{i})
            for k = 1:size(V{i}{j},1)
                [isin,inwhich] = isinside(Regions(l),V{i}{j}(k,:)');
                if isin == 1
                    VV_Regions{l} = [VV_Regions{l};V{i}{j}(k,:)];
                end
            end
        end
    end
    VV_Regions{l} = unique(VV_Regions{l},'rows');
end

%   We verify in every region which affine component pwa_i is <= than pwa
%   in all points of the region

termsArray = []; terms = {};
for i = 1:length(Regions)
    terms{i} = [];
    for j = 1:N_affine_terms
        termsArray(i,j) = 1;
        for k = 1:size(VV_Regions{i},1)
            if alfa{j}*VV_Regions{i}(k,:)'+beta{j} - ...
                    evalPWA(pwaTest,VV_Regions{i}(k,:)') <= 1e-9;
                continue;
            else
                termsArray(i,j) = 0;
                break
            end
        end
        if termsArray(i,j) == 1
            terms{i} = [terms{i} j];
        end
    end
end

mmps.terms = terms;
```

A.8 pwa2mmpsDisjOvc

```
function mmps = pwa2mmpsDisjOvc(pwa)

%   mmps = pwa2mmpsDisjOvc(pwa)
%   Converts a PWA function in the equivalent MMPS function
%   in the disjunctive form through the Ovchinnikov strategy
%   pwa ->  PWA function that must be converted
%   mmps -> MMPS function obtained as output

%   This function calls:
%       testPWA
%       evalPWA

%   This function is called by:
%       pwa2mmps

%   A. Frau
%   20/7/07

%   Last modification   25/5/07
%   Last modification   29/5/07
%   Last modification   30/5/07
%   Last modification   12/7/07

%   Test

pwaTest = testPWA(pwa);

alfa = pwaTest.alfa; beta = pwaTest.beta;
H = pwaTest.H; K = pwaTest.K;
N_affine_terms = pwaTest.N_affine_terms;
dimension = pwaTest.dimension;
P = pwaTest.P; domain = pwaTest.domain;

%   Computation of vertices

V = {}; R = {}; adjV = {};
for i = 1:length(P)
    for j = 1:length(P{i})
        %   the vertices are already stored in polytope structure,
        %   so the computation is very fast
        [V{i}{j},R{i}{j},P{i}(j),adjV{i}{j}] = extreme(P{i}(j));
    end
end
```

```
%   Computation of adjacent regions in which
%   pwa = max(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))

adj_vertices = {};  adj_regions = [];   max_regions = [];
for i = 1:length(P)
    for j = 1:length(P)
        if i<j  % because otherwise is redundant or wrong (i = j)
            for k = 1:length(P{i})
                for l = 1:length(P{j})
                    adj_vertices{i,j,k,l} = [];
                    for m = 1:size(V{i}{k},1)
                        for n = 1:size(V{j}{l},1)
                            if all(abs(V{i}{k}(m,:)-V{j}{l}(n,:)) <= 1e-10)
                                %   vertices in common between P{i}(k)
                                %       and P{j}(l)
                                adj_vertices{i,j,k,l} = ...
                                    [adj_vertices{i,j,k,l}; V{i}{k}(m,:)];
                            end
                        end
                    end
                    if isempty(adj_vertices{i,j,k,l})
                        %   no adjacent vertices between P{i}(k) and
                        %   P{j}(l)
                        adj_regions(i,j,k,l) = 0;
                        max_regions(i,j,k,l) = 0;
                    elseif size(adj_vertices{i,j,k,l},1) == dimension
                        %   P{i}(k) and P{j}(l) are adjacent
                        adj_regions(i,j,k,l) = 1;
                        %   comparison in all vertex of P{i}{k} and in all
                        %   vertex of P{j}{l}
                        if  all(alfa{i}*V{i}{k}'+beta{i}-...
                                (alfa{j}*V{i}{k}'+beta{j}) >= -1e-9) &...
                            all(alfa{j}*V{j}{l}'+beta{j}-...
                                (alfa{i}*V{j}{l}'+beta{i}) >= -1e-9)
                            %   pwa = max(pwa_i,pwa_j)
                            %       on union(P{i}(k),P{j}(l))
                            max_regions(i,j,k,l) = 1;
                        else
                            %   pwa != max(pwa_i,pwa_j)
                            %       on union(P{i}(k),P{j}(l))
                            max_regions(i,j,k,l) = 0;
                        end
                    else
                        adj_regions(i,j,k,l) = 0;
                        max_regions(i,j,k,l) = 0;
                    end
                end
            end
        end
    end
end

max_array = [];
for i = 1:length(P)
    for j = 1:length(P)
        if i<j
            for k = 1:length(P{i})
                for l = 1:length(P{j})
                    if max_regions(i,j,k,l) == 1
                        % [i k j l] belong to max_array if pwa =
                        % max(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))
                        max_array = [max_array;i k j l];
                    end
                end
            end
        end
    end
end

%   Some outputs

mmps.alfa = alfa;
mmps.beta = beta;
mmps.domain = domain;
mmps.form = 'disj';

if isempty(max_array)
    %   The function is concave if there does not exist a pair of regions
    %   such that pwa = max(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))
    mmps.terms = {1:N_affine_terms};
    return
end

%   Computation of the hyperplanes that split the pairs of regions above
%   computed, and therefore computation of the constraint in common
%   between each pair of adjacent regions in
%   which pwa = max(pwa_i,pwa_j) on union(P{i}(k),P{j}(l))

max_constrH = [];   max_constrK = [];
index = 1;  count_max_constr = 0;
for i = 1:size(max_array,1)
    for j = 1:size(H{max_array(i,1)}{max_array(i,2)},1)
        for k = 1:size(H{max_array(i,3)}{max_array(i,4)},1)
            if all(abs(H{max_array(i,1)}{max_array(i,2)}(j,:) +...
                    H{max_array(i,3)}{max_array(i,4)}(k,:)) <= 1e-7) &&...
               all(abs(K{max_array(i,1)}{max_array(i,2)}(j) +...
                    K{max_array(i,3)}{max_array(i,4)}(k)) <= 1e-7)
                %   let us verify if the constraint is already in max_constr
                for m = 1:count_max_constr
                    if (all(abs(max_constrH(m,:) -...
                        H{max_array(i,1)}{max_array(i,2)}(j,:)) <= 1e-7) &&...
                        all(abs(max_constrK(m,:) -...
                        K{max_array(i,1)}{max_array(i,2)}(j,:)) <= 1e-7)) ||...
                       (all(abs(max_constrH(m,:) +...
                        H{max_array(i,1)}{max_array(i,2)}(j,:)) <= 1e-7) &&...
                        all(abs(max_constrK(m,:) +...
                        K{max_array(i,1)}{max_array(i,2)}(j,:)) <= 1e-7))
                        index = 0;
                        break
                    end
                end
                if (index == 1)
                    max_constrH = ...
                        [max_constrH;H{max_array(i,1)}{max_array(i,2)}(j,:);];
                    max_constrK = ...
                        [max_constrK;K{max_array(i,1)}{max_array(i,2)}(j,:);];
                    count_max_constr = count_max_constr + 1;
                end
                index = 1;
            end
        end
    end
end

%   Computation of all regions in which we compute if pwa_i >= pwa

%   all possible combinations of 0 and 1
comb = makebits(size(max_constrH,1));

RegionsTempH = {};   RegionsTempK = {};
RegionsTemp = polytope;    RegionsTemp2 = polytope;
count = 0;
for i = 1:size(comb,1)
    RegionsTempH{i} = [];   RegionsTempK{i} = [];
    for j = 1:size(comb,2)
        if comb(i,j) == 1
            RegionsTempH{i} = [RegionsTempH{i};max_constrH(j,:);];
            RegionsTempK{i} = [RegionsTempK{i};max_constrK(j,:);];
        else
            RegionsTempH{i} = [RegionsTempH{i};-max_constrH(j,:);];
            RegionsTempK{i} = [RegionsTempK{i};-max_constrK(j,:);];
        end
    end
    RegionsTemp = polytope(RegionsTempH{i},RegionsTempK{i});
    if isfulldim(RegionsTemp)
        RegionsTempInters = intersect(RegionsTemp,domain);
        if isfulldim(RegionsTempInters);
            count = count + 1;
            RegionsTemp2(count) = RegionsTempInters;
        end
    end
end

%   The union of the temporary regions obtained must be equal to the domain

if union(RegionsTemp2) ~= domain
    error(['The union of the temporary regions',...
        ' obtained must be equal to the domain']);
end

%   Elimination of identical regions, they must appear only once

identical = ones(1,length(RegionsTemp2));
for i = 1:length(RegionsTemp2)
    for j = i+1:length(RegionsTemp2)
        if RegionsTemp2(i) == RegionsTemp2(j)
            identical(j) = 0;
        end
    end
end

Regions = polytope;   count_Regions = 0;
for i = 1:length(RegionsTemp2)
    if identical(i) == 1
        count_Regions = count_Regions + 1;
        Regions(count_Regions) = RegionsTemp2(i);
    end
end

%   The union of the regions obtained must be equal to
%   the domain

if union(Regions) ~= domain
    error(['The union of the regions',...
        ' obtained must be equal to the domain']);
end

%   For each region in which the test must be done, we find all vertices of
%   the regions of pwa that lies in that region, and an affine component
%   defined in this vertex

VV_Regions = {};
for l = 1:length(Regions)
    VV_Regions{l} = [];   %   VV_Regions{l,2} = [];
    for i = 1:length(V)
        for j = 1:length(V{i})
            for k = 1:size(V{i}{j},1)
                [isin,inwhich] = isinside(Regions(l),V{i}{j}(k,:)');
                if isin == 1
                    VV_Regions{l} = [VV_Regions{l};V{i}{j}(k,:)];
                end
            end
        end
    end
    VV_Regions{l} = unique(VV_Regions{l},'rows');
end

%   We verify in every region which affine component pwa_i is >= than pwa
%   in all points of the region

termsArray = []; terms = {};
for i = 1:length(Regions)
    terms{i} = [];
    for j = 1:N_affine_terms
        termsArray(i,j) = 1;
        for k = 1:size(VV_Regions{i},1)
            if alfa{j}*VV_Regions{i}(k,:)'+beta{j} - ...
                    evalPWA(pwaTest,VV_Regions{i}(k,:)') >= -1e-9;
                continue;
            else
                termsArray(i,j) = 0;
                break
            end
        end
        if termsArray(i,j) == 1
            terms{i} = [terms{i} j];
        end
    end
end

mmps.terms = terms;
```

A.9 testMMPS

```
function mmpsTest = testMMPS(mmps)

%   mmpsTest = testMMPS(mmps);
%   The code tests if the input is a valid MMPS function
%   mmps -> MMPS function that must be tested
%   mmpsTest -> MMPS function after the tests, with some additional fields

%   This function is called by
%       mmps2pwaConj
%       mmps2pwaDisj

%   A. Frau
%   30/6/07

%   Last modification   2/7/07

%   If the test has already been done, don't do it again

if isfield(mmps,'test')
    mmpsTest = mmps;
    return
end

alfa = mmps.alfa; beta = mmps.beta; terms = mmps.terms;

%   The fields must have the right dimension

if length(alfa) ~= length(beta)
    error('alfa and beta must have the same length');
end

N_affine_terms = length(alfa);

%   Every component of alfa and beta must have 1 row
%   Every component of beta must have 1 column

for i = 1:N_affine_terms
    if size(alfa{i},1) ~= 1
        error(['alfa{',int2str(i),'} must have 1 row']);
    end
    if size(beta{i},1) ~= 1
        error(['beta{',int2str(i),'} must have 1 row']);
    end
    if size(beta{i},2) ~= 1
```

```
        error(['beta{',int2str(i),'} must have 1 column']);
    end
end

%   Any component of alfa,beta must have the same length

for i = 1:N_affine_terms-1
    if length(alfa{i}) ~= length(alfa{i+1})
        error('Any component of alfa must have the same length');
    end
end

dimension = length(alfa{1});

%   If A and b are fields, so they must have the right dimensions

if isfield(mmps,'A') && isfield(mmps,'b')
    if size(mmps.A,2) ~= dimension
        error(['The number of columns of A must be ',int2str(dimension)]);
        return
    end
    if size(mmps.b,2) ~= dimension
        error('The number of columns of b must be 1');
        return
    end
end

%   In the (max or min) terms of a MMPS function cannot appear a
%   non-existent affine term

max_affine_term = 0;
for i = 1:length(terms)
    if max(terms{i}) > max_affine_term
        max_affine_term = max(terms{i});
    end
end

if max_affine_term > N_affine_terms
    error('In a term of a MMPS function cannot appear a non-existent affine term');
end

%   Output

if isfield(mmps,'form')
    mmpsTest.form = mmps.form;
else
    %   if there is not the field 'form', then is disjunctive by
    %   default
    mmpsTest.form = 'disj';
end

if isfield(mmps,'domain')
    mmpsTest.domain = mmps.domain;
elseif isfield(mmps,'A') && isfield(mmps,'b')
    mmpsTest.domain = polytope(mmps.A,mmps.b);
else
    mmpsTest.domain = unitbox(dimension,1e4);   %   domain used by default
end

mmpsTest.alfa = alfa;
mmpsTest.beta = beta;
mmpsTest.terms = terms;
mmpsTest.N_affine_terms = N_affine_terms;
mmpsTest.dimension = dimension;
mmpsTest.test = 'ok';
```

A.10 testPWA

```
function pwaTest = testPWA(pwa)

%   pwaTest = testPWA(pwa);
%   The code tests if the input is a valid and continuous PWA function. The
%   same function is returned as output
%   pwa -> PWA function that must be tested
%   pwaTest -> PWA function after the tests, with some additional fields

%   This function is called by:
%       pwa2mmpsConjOvc
%       pwa2mmpsDisjOvc
%       pwa2mmpsConjGor
%       pwa2mmpsDisjGor

%   A. Frau
%   25/6/07

%   Last modification 28/6/07
%   Last modification 2/7/07

%   If the test has already been done, don't do it again

if isfield(pwa,'test')
```

```
    pwaTest = pwa;
    return
end

alfa = pwa.alfa;  beta = pwa.beta;  H = pwa.H;    K = pwa.K;

%   The fields must have the right dimension

if length(alfa) ~= length(beta) || length(alfa) ~= length(H) ||...
        length(alfa) ~= length(K)
    error('alfa, beta, H and K must have the same length');
    return
end

%   number of affine components of the PWA function in input

N_affine_terms_old = length(alfa);

%   Every component of alfa and beta must have 1 row

for i = 1:N_affine_terms_old
    if size(alfa{i},1) ~= 1
        error(['alfa{',int2str(i),'} must have 1 row']);
    end
    if size(beta{i},1) ~= 1
        error(['beta{',int2str(i),'} must have 1 row']);
    end
    if size(beta{i},2) ~= 1
        error(['beta{',int2str(i),'} must have 1 column']);
    end
end

%   All components of alfa must have the same length

for i = 1:N_affine_terms_old-1
    if length(alfa{i}) ~= length(alfa{i+1})
        error('Any component of alfa must have the same length');
    end
end

dimension = length(alfa{1});    %   dimension of the function

%   If H{i} and K{i} are cells, they must have the same length

H_cell = {};   K_cell = {};
for i = 1:length(H)
    if iscell(H{i}) == 0
```

```
        H_cell{i} = {H{i}};   % all the entries of H become cells
        K_cell{i} = {K{i}};   % all the entries of K become cells
    else
        H_cell{i} = H{i};
        K_cell{i} = K{i};
    end
end

for i = 1:length(H_cell)
    if length(H_cell{i}) ~= length(K_cell{i})
        error(['H{',int2str(i),'} and K{',int2str(i),...
            '} must have the same length']);
        return
    end
end

%   Every component of H must have the number of columns equal to the
%       dimension
%   Every component of K must have the number of columns equal to 1
%   The number of rows of H{i}{j} must be equal to the one of K{i}{j}

for i = 1:length(H_cell)
    for j = 1:length(H_cell{i})
        if size(H_cell{i}{j},2) ~= dimension
            error(['The number of columns of H{',int2str(i),...
                '}{',int2str(j),'} must be ',int2str(dimension)]);
            return
        end
        if size(K_cell{i}{j},2) ~= 1
            error(['The number of columns of K{',int2str(i),'}{'...
                ,int2str(j),'} must be equal to 1']);
            return
        end
        if size(H_cell{i}{j},1) ~= size(K_cell{i}{j},1)
            error(['H{',int2str(i),'}{',int2str(j),'} and K{',...
                int2str(i),'}{',int2str(j),...
                '} must have the same number of rows']);
            return
        end
    end
end

%   Redefinition the PWA function, for avoiding identical affine terms

index1 = zeros(N_affine_terms_old,1);
index2 = zeros(N_affine_terms_old,1);
count = 0;
```

```
for i = 1:N_affine_terms_old
    if index2(i) == 0   % if not comparised yet
        count = count + 1;
        for j = i+1:N_affine_terms_old
            if index2(j) == 0   % if not comparised yet
                if all(abs(alfa{i}-alfa{j}) <= 1e-10) &...
                        abs(beta{i}-beta{j}) <= 1e-10
                    index1(j) = count;
                    index2(j) = 1;
                end
            end
        end
        index1(i) = count;
        index2(i) = 1;  % not necessary
    end
end

alfa_new = {};  beta_new = {};
f = {};
for i = 1:max(index1)
    f{i} = find(index1 == i);
    alfa_new{i} = alfa{f{i}(1)};
    beta_new{i} = beta{f{i}(1)};
end

count2 = {};
guardH_new = {};    guardK_new = {};
for i = 1:max(index1)
    guardH_new{i} = {};
    guardK_new{i} = {};
    count2{i} = 0;
    for j = 1:N_affine_terms_old
        if index1(j) == i
            for k = 1:length(H_cell{j})
                count2{i} = count2{i} + 1;
                guardH_new{i}{count2{i}} = H_cell{j}{k} ;
                guardK_new{i}{count2{i}} = K_cell{j}{k} ;
            end
        end
    end
end

%   The set of regions that defines the PWA functions must be a
%   polyhedral partition of the domain

P = {};
```

```
for i = 1:length(guardH_new)
    for j = 1:length(guardH_new{i})
       P{i}(j) = reduce(polytope(guardH_new{i}{j},guardK_new{i}{j}));
    end
    P{i} = merge(P{i});
end

for i = 1:length(P)
    for j = 1:length(P{i})
        for k = 1:length(P)
            for l = 1:length(P{k})
                if i ~= k || j ~= l
                    %   if the intersection is not empty
                    if and(P{i}(j),P{k}(l)) ~= polytope
                        disp(['and(P{',int2str(i),'}(',int2str(j),...
                            '),P{',int2str(k),'}(',int2str(l),...
                            ')) is full dimensional']);
                        error(['The set of regions is not a',...
                            ' polyhedral partition of the domain']);
                    end
                end
            end
        end
    end
end

%   The domain must be a convex set of regions

P_domain = [];
for i = 1:length(P)
    for j = 1:length(P{i})
        P_domain = [P_domain P{i}(j)];
    end
end

if ~isconvex(P_domain)
    error('The domain is not a convex set of regions');
end

P_domain = union(P_domain); V = {};
H_def = {};    K_def = {};
for i = 1:length(P)
    for j = 1:length(P{i})
        [H_def{i}{j},K_def{i}{j}] = double(P{i}(j));
        %   R and adjV are not necessary
        [V{i}{j},R{i}{j},P{i}(j),adjV{i}{j}] = extreme(P{i}(j));
    end
```

```
end

%   Let us compute, for each couple of components, the vertices
%       in common between the two sets of regions

common_vertices = cell(length(alfa_new),length(alfa_new));
for i = 1:length(P)
    for j = 1:length(P)
        if i<j  % because otherwise is redundant
            for k = 1:length(P{i})
                for l = 1:length(P{j})
                    for m = 1:size(V{i}{k},1)
                        for n = 1:size(V{j}{l},1)
                            if all(abs(V{i}{k}(m,:)-V{j}{l}(n,:))...
                                    <= 1e-10)
                                common_vertices{i,j} = ...
                                    [common_vertices{i,j}; V{i}{k}(m,:)];
                            end
                        end
                    end
                end
            end
        end
    end
end

%   Verification of the continuity of the function

for i = 1:length(V)
    for j = 1:length(V)
        % if there are some vertices in common
        if ~isempty(common_vertices{i,j})
            if any(abs(alfa_new{i}*common_vertices{i,j}'+beta_new{i} - ...
                    alfa_new{j}*common_vertices{i,j}'-beta_new{j})...
                    >= 1e-9) == 1
                error('The function is not continuous');
            end
        end
    end
end

%   Output

pwaTest.alfa = alfa_new;
pwaTest.beta = beta_new;
pwaTest.H = H_def;
pwaTest.K = K_def;
```

```
%   if "test" is a field of pwa, the test won't be done anymore
pwaTest.test = 'ok';

%   Fields used for computations in functions that call this one

pwaTest.P = P;
pwaTest.domain = P_domain;
pwaTest.dimension = dimension;
pwaTest.N_affine_terms = length(alfa_new);
```

A.11 evalPWA

```
function value = evalPWA(pwa,x)

%   value = evalPWA(pwa,x)
%   The function computes the value of the PWA function in a selected point
%   value -> value of the PWA function "pwa" in "x"
%   x must be a column matrix

%   This function is called by
%       isequalPWA
%       pwa2mmpsConjOvc
%       pwa2mmpsDisjOvc

%   A.Frau
%   21/6/07

%   Test

pwaTest = testPWA(pwa);

if size(x,1) ~= pwaTest.dimension || size(x,2) ~= 1
    error('The dimension of x is incorrect');
end

alfa = pwaTest.alfa;
beta = pwaTest.beta;
P = pwaTest.P;

%   Computation of the value of the function in x

value = []; h = 0;
for i = 1:length(P)
    for j = 1:length(P{i})
```

```
        if isinside(P{i}(j),x) == 1
            value = alfa{i}*x + beta{i};
            h = 1;
            break
        end
    end
    if h == 1
        break
    end
end

if isempty(value) == 1
    error('x is not inside any polytope!!!');
end
```

A.12 isequalPWA

```
function value = isequalPWA(pwa1,pwa2)

%   value = isequalPWA(pwa1,pwa2)
%   Returns true(1) if the PWA functions are equivalent,
%       false(0) otherwise
%   pwa1, pwa2 ->   PWA functions that must be compared

%   This function calls
%       evalPWA

%   This function is called by
%       PWAsimulationsGor
%       PWAsimulationsOvc

%   A. Frau
%   24/6/07

%   Last modification   3/7/07

%   Test

pwaTest1 = testPWA(pwa1);  % the tests are already here
pwaTest2 = testPWA(pwa2);  % the tests are already here

%   The dimension and the domain of the functions must be the same

if pwaTest1.dimension ~= pwaTest2.dimension
```

```
    error('The dimensions of the PWA functions are different');
end

if pwaTest1.domain ~= pwaTest2.domain
    error('The domains must be equal');
end

P1 = pwaTest1.P;    %   sets of regions of the function pwa1
P2 = pwaTest2.P;    %   sets of regions of the function pwa2

%   Computation of all vertices of the regions of pwa1

VV1 = [];
for i = 1:length(P1)
    for j = 1:length(P1{i})
        %   the computation of vertices is fast because they are
        %       already stored in the polytope structure
        V1{i}{j} = extreme(P1{i}(j));
        VV1 = [VV1;V1{i}{j}];
    end
end

%   Computation of all vertices of the regions of pwa2

VV2 = [];
for i = 1:length(P2)
    for j = 1:length(P2{i})
        %   the computation of vertices is fast because they are
        %       already stored in the polytope structure
        V2{i}{j} = extreme(P2{i}(j));
        VV2 = [VV2;V2{i}{j}];
    end
end

%   Computation of the hyperbox. All vertices must lie in
%   the interior of this hyperbox

VVall = [VV1; VV2];
boundaries = max(VVall,[],1) + 1;
dimension = pwaTest1.dimension;
hbox_H = [eye(dimension);-eye(dimension)];
hbox_K = [boundaries';boundaries'];
hbox = polytope(hbox_H,hbox_K);     %   Hyperbox

count = 0;  P_int = {}; P_cell = {};
for i = 1:length(P1)
    for j = 1:length(P1{i})
```

```
        for k = 1:length(P2)
            for l = 1:length(P2{k})
                P_int{i,j,k,l} = and(P1{i}(j),P2{k}(l));
                % otherwise next intersection cannot be computed
                if ~isempty(P_int{i,j,k,l}) &&...
                        isfulldim(P_int{i,j,k,l})
                    P_int{i,j,k,l} = and(P_int{i,j,k,l},hbox);
                    if ~isempty(P_int{i,j,k,l}) &&...
                            isfulldim(P_int{i,j,k,l})
                        count = count + 1;
                        P_cell{count} = P_int{i,j,k,l};
                    end
                end
            end
        end
    end
end

VV = [];    V = {};
for i = 1:length(P_cell)
    V{i} = extreme(P_cell{i});
    VV = [VV;V{i}];
end
%   For having only different vertices
VV = unique (VV,'rows');

value1 = [];    value2 = [];
for i = 1:size(VV,1)
    value1(i) = evalPWA(pwa1,VV(i,:)');
    value2(i) = evalPWA(pwa2,VV(i,:)');
end

%   Output

value = all(abs(value1-value2) <= 1e-9);
```

A.13 isequalMMPS

```
function value = isequalMMPS(mmps1,mmps2)

%   value = isequalMMPS(mmps1,mmps2)
%   Returns true(1) if the MMPS functions are equivalent,
%       false(0) otherwise
%   mmps1, mmps2 ->   MMPS functions that must be compared
%   This function is called by:
%       PWAsimulationsGor
%       PWAsimulationsOvc
%       MMPSsimulations

%   A. Frau
%   5/7/07

mmpsTest1 = testMMPS(mmps1);
mmpsTest2 = testMMPS(mmps2);

%   The dimension and the domain of the functions must be the same

if mmpsTest1.dimension ~= mmpsTest2.dimension
    error('The functions must have the same dimension');
end

if mmpsTest1.domain ~= mmpsTest2.domain
    error('The functions must have the same domain');
end

domain = mmpsTest1.domain;
dimension = mmpsTest1.dimension;

alfa1 = mmpsTest1.alfa; beta1 = mmpsTest1.beta;
terms1 = mmpsTest1.terms; form1 = mmpsTest1.form;
N_affine_terms1 = mmpsTest1.N_affine_terms;

alfa2 = mmpsTest2.alfa; beta2 = mmpsTest2.beta;
terms2 = mmpsTest2.terms;   form2 = mmpsTest2.form;
N_affine_terms2 = mmpsTest2.N_affine_terms;

[H_domain,K_domain] = double(domain);

b = zeros(2,dimension+1);
b(:,dimension+1) = [1 -1]';
H_domain_nPlus1 = [H_domain zeros(size(H_domain,1),1); b];
K_domain_nPlus1 = [K_domain; 1e4; 1e4];
domain_nPlus1 = polytope(H_domain_nPlus1,K_domain_nPlus1);

%   Computation of the epigraph or hypograph of each affine
%   component of the function mmps1

H_affineEpiHyp1 = {}; K_affineEpiHyp1 = {};
affineEpiHyp1 = polytope;
for i = 1:N_affine_terms1
```

```
    if strcmp(form1,'disj')
        H_affineEpiHyp1{i} = [H_domain zeros(size(H_domain,1),1);...
            -alfa1{i} 1; b];
        K_affineEpiHyp1{i} = [K_domain; beta1{i}; 1e4; 1e4];
        %   Hypograph of each affine component
        affineEpiHyp1(i) = polytope(H_affineEpiHyp1{i},...
            K_affineEpiHyp1{i});
    else
        H_affineEpiHyp1{i} = [H_domain zeros(size(H_domain,1),1);...
            alfa1{i} -1; b];
        K_affineEpiHyp1{i} = [K_domain; -beta1{i}; 1e4; 1e4];
        %   Epigraph of each affine component
        affineEpiHyp1(i) = polytope(H_affineEpiHyp1{i},...
            K_affineEpiHyp1{i});
    end
end

TermArrayEpiHyp1 = {};  TermEpiHyp1 = {};   EpiHyp1 = polytope;
for i = 1:length(terms1)
    TermArrayEpiHyp1{i} = [];
    for j = 1:length(terms1{i})
        TermArrayEpiHyp1{i} = horzcat(TermArrayEpiHyp1{i},...
            affineEpiHyp1(terms1{i}(j)));
    end
    TermEpiHyp1{i} = and(TermArrayEpiHyp1{i});
    %   Computation of the epigraph or hypograph of the function mmps1
    EpiHyp1 = [EpiHyp1,TermEpiHyp1{i}];
end

if strcmp(form1,'conj')
    Hypograph1 = mldivide(domain_nPlus1,EpiHyp1);
else
    Hypograph1 = EpiHyp1;
end

%   Computation of the epigraph or hypograph of each affine
%   component of the function mmps2

H_affineEpiHyp2 = {}; K_affineEpiHyp2 = {};
affineEpiHyp2 = polytope;
for i = 1:N_affine_terms2
    if strcmp(form2,'disj')
        H_affineEpiHyp2{i} = [H_domain zeros(size(H_domain,1),1);...
            -alfa2{i} 1; b];
        K_affineEpiHyp2{i} = [K_domain; beta2{i}; 1e4; 1e4];
        %   Hypograph of each affine component
        affineEpiHyp2(i) = polytope(H_affineEpiHyp2{i},...
            K_affineEpiHyp2{i});
    else
        H_affineEpiHyp2{i} = [H_domain zeros(size(H_domain,1),1);...
            alfa2{i} -1; b];
        K_affineEpiHyp2{i} = [K_domain; -beta2{i}; 1e4; 1e4];
        %   Epigraph of each affine component
        affineEpiHyp2(i) = polytope(H_affineEpiHyp2{i},...
            K_affineEpiHyp2{i});
    end
end

TermArrayEpiHyp2 = {};  TermEpiHyp2 = {};   EpiHyp2 = polytope;
for i = 1:length(terms2)
    TermArrayEpiHyp2{i} = [];
    for j = 1:length(terms2{i})
        TermArrayEpiHyp2{i} = horzcat(TermArrayEpiHyp2{i},...
            affineEpiHyp2(terms2{i}(j)));
    end
    TermEpiHyp2{i} = and(TermArrayEpiHyp2{i});
    %   Computation of the epigraph or hypograph of the function mmps2
    EpiHyp2 = [EpiHyp2,TermEpiHyp2{i}];
end

if strcmp(form2,'conj')
    Hypograph2 = mldivide(domain_nPlus1,EpiHyp2);
else
    Hypograph2 = EpiHyp2;
end

Hypographs_diff = [Hypograph1\Hypograph2 Hypograph2\Hypograph1];

%  the 2 tests are identical if there are no numerical problems
if (Hypographs_diff == polytope) && Hypograph1 == Hypograph2
    value = 1;
else
    value = 0;
end
```

A.14 isequalPWA2MMPSovc

```
function value = isequalPWA2MMPSovc(pwa,mmps)

%   value = isequalPWA2MMPSovc(pwa,mmps)    Ok
%   Returns true(1) if the functions are equivalent,
%       false(0) otherwise
%   In this test the PWA function is converted into the
%       equivalent MMPS one by the Ovchinnikov strategy.
%   pwa, mmps ->   PWA function and MMPS one that must be compared

%   This function calls:
%       testMMPS
%       testPWA
%       pwa2mmps
%       mmps2pwa
%       isequalPWA
%       isequalMMPS

%   This function is called by:
%       PWAsimulationsGor

%   A. Frau
%   19/7/07

%   Test

mmpsTest = testMMPS(mmps);
pwaTest = testPWA(pwa);

%   The dimension and the domain of the functions must be the same

if mmpsTest.dimension ~= pwaTest.dimension
    error('The functions must have the same dimension');
end

if mmpsTest.domain ~= pwaTest.domain
    error('The functions must have the same domain');
end

%   Comparison between the MMPS function in input and the MMPS one
%   obtained by the conversion of the PWA function given in input

mmpsT = pwa2mmps(pwaTest,mmpsTest.form,'Ovc');
valueMMPS = isequalMMPS(mmpsTest,mmpsT);

%   Comparison between the PWA function in input and the PWA one
%   obtained by the conversion of the MMPS function given in input

pwaT = mmps2pwa(mmpsTest);
valuePWA = isequalPWA(pwaTest,pwaT);

%   Output
```

```
value = valueMMPS & valuePWA;
```

A.15 isequalPWA2MMPSgor

```
function value = isequalPWA2MMPSgor(pwa,mmps)

%   value = isequalPWA2MMPSgor(pwa,mmps)
%   Returns true(1) if the functions are equivalent,
%       false(0) otherwise
%   In this test the PWA function is converted into the equivalent
%       MMPS one by the Gorokhovik-Zorko strategy.
%   pwa, mmps ->   PWA function and MMPS one that must be compared

%   This function calls:
%       testMMPS
%       testPWA
%       pwa2mmps
%       mmps2pwa
%       isequalPWA
%       isequalMMPS

%   This function is called by:
%       PWAsimulationsGor
%       MMPSsimulations

%   A. Frau
%   19/7/07

%   Test

mmpsTest = testMMPS(mmps);
pwaTest = testPWA(pwa);

%   The dimension and the domain of the functions must be the same

if mmpsTest.dimension ~= pwaTest.dimension
    error('The functions must have the same dimension');
end

if mmpsTest.domain ~= pwaTest.domain
    error('The functions must have the same domain');
end
```

```
%   Comparison between the MMPS function in input and the MMPS one
%   obtained by the conversion of the PWA function given in input

mmpsT = pwa2mmps(pwaTest,mmpsTest.form,'Gor');
valueMMPS = isequalMMPS(mmpsTest,mmpsT);

%   Comparison between the PWA function in input and the PWA one
%   obtained by the conversion of the MMPS function given in input

pwaT = mmps2pwa(mmpsTest);
valuePWA = isequalPWA(pwaTest,pwaT);

%   Output

value = valueMMPS & valuePWA;
```

A.16 continuousPWAgenerator

```
function pwaOut = continuousPWAgenerator(num)

%   pwaOut = continuousPWAgenerator(num)
%   Generator of partially random continuous PWA functions

%   A. Frau
%   9/7/07

%   This function calls:
%       testPWA

%   This function is called by:
%       PWAsimulationsGor
%       PWAsimulationsOvc

if nargin == 0
    num = 1;
end

count = 0;
pwa = {};   pwaOut = {};

while(count <= num)
    try
        m = abs(rmat(1,1,3)) + 1;  %   inputs   1 < m < 4
        n = abs(rmat(1,1,2)) + 1;  %   states   1 < n < 3
        %   number of constraints in the MPLP   2 < n < 8
        f = abs(rmat(1,1,6)) + 2;
        %   The meaning of the fields is explained in the MPT toolbox
        matrices.H = rmat(1,m,5,0,0.8);
        matrices.F = zeros(1,n);
        matrices.G = abs(rmat(f,m,3,0,0.5));
        matrices.W = abs(rmat(f,1,3,0,0.5)) + 1;
        matrices.E = rmat(f,n,5,0,0.5);
        matrices.bndA = [eye(n); -eye(n)];
        matrices.bndb = 1e4*ones(2*n,1);
        Pn = polytope;  Fi = {};    Gi = {};
        Options.debug_level = 0;
        Options.verbose = 1;
        Options.max_iter = 5;   % default value is 20
        [Pn,Fi,Gi,activeConstraints,Phard] = mpt_mplp(matrices,Options);
        %   the domain must be convex, otherwise it is discarded
        if ~isconvex(Phard)
            continue
        end
        for i = 1:size(Fi{1},1)
            for j = 1:length(Pn)
                pwa{count+i}.alfa{j} = Fi{j}(i,:);
                pwa{count+i}.beta{j} = Gi{j}(i);
                [pwa{count+i}.H{j},pwa{count+i}.K{j}] = double(Pn(j));
            end
            pwaOut{count+i} = testPWA(pwa{count+i});
        end
        count = count + size(Fi{1},1);
        if count >= num
            break
        end
    catch
        continue
    end
end
```

A.17 MMPSgenerator

```
function mmps = MMPSgenerator(num)

%   mmps = MMPSgenerator(num);
%   Generator of partially random MMPS functions

%   This function is called by:
```

```
%       MMPSsimulations

%   A. Frau
%   19/6/07

if nargin == 0
    num = 1;
end

mmps = {};

for l = 1:num
    dimension = 1 + abs(rmat(1,1,2));   %   1 <= dimension <= 3
    N_affine_terms = 3 + abs(rmat(1,1,7));  % 3 < N_affine_terms < 10

    %   Generation of different affine components
    alfa = {}; beta = {};
    index = 0;
    while (index == 0)
        for i = 1:N_affine_terms
            alfa{i} = rmat(1,dimension,10);
            beta{i} = rmat(1,1,20);
        end
        index = 1;
        for i = 1:N_affine_terms
            for j = i+1:N_affine_terms
                if all(alfa{i} == alfa{j}) && all(beta{i} == beta{j})
                    index = 0;
                    break
                end
            end
            if index == 0
                break
            end
        end
    end

    N_terms = 10; % max number of terms
    cont = 0;   count = 0;  % initialization
    terms = {}; k = 0;

    while(1)
        if cont == 0
            k = k + 1;
            if k > N_affine_terms
                break
            end
            combination = 1:k;
            cont = 1;
        else
            [combination,cont] = next_comb(combination,N_affine_terms);
            if cont == 0
                continue
            end
        end
        if abs(rmat(1,1,1,0,0.70)) == 1  %   20% of possibility to be 1
            count = count + 1;
            terms{count} = combination;
        end
        if count >= N_terms
            break
        end
    end

    if count == 0
        terms{1} = 1:N_affine_terms;   %   there must be at least 1 term
    end

    mmps{l}.alfa = alfa;
    mmps{l}.beta = beta;
    mmps{l}.terms = terms;
    if abs(rmat(1,1,1,0,0.25)) == 1     %   50% of possibility to be 1
        mmps{l}.form = 'conj';
    else
        mmps{l}.form = 'disj';
    end
    mmps{l}.dimension = dimension;
    mmps{l}.N_affine_terms = N_affine_terms;
end
```

A.18 rmat

```
function matrix=rmat(rows,cols,bound,el,pel)

% Syntax:  matrix=rmat(rows,cols,bound,el,pel) or
%          matrix=rmat([rows,cols],bound,el,pel)
%
% Purpose: Generates a random integer matrix.
%          matrix=rmat(rows,cols,bound) returns a rows by cols matrix,
%          the elements of which are random integers in the interval
%          [-bound,bound].
%          matrix=rmat(rows,cols,bound,el,pel) returns a rows by cols
%          matrix, the elements of which are random integers in the
%          interval [-bound,bound], but some elements may be equal to
%          el with a probability pel.
%          matrix=rmat(size(a),bound,el,pel) returns a matrix that has the
%          same size as a.
%
% Inputs:  rows   integer, optional (default value: 3)
%          cols   integer, optional (default value: rows)
%          bound  integer, optional (default value: 5)
%          el     real
%          pel    real, optional (default value: 0.2)
%
% Outputs: matrix  rows by cols matrix

% Created: Nov 15, 1991       by Bart De Schutter
% Last revised: Oct 24, 2003  by Bart De Schutter

nargin_local=nargin;
if ( nargin_local < 1 ),
   rows=3;
   cols=3;
   bound=5;
else
   len_arg_1=length(rows);
   if ( len_arg_1 > 2 )
      error(...
      'Use rmat(size(a),bound,el,pel) instead of rbmat(a,bound,el,pel).');
   elseif ( len_arg_1 == 1 )
      if ( nargin_local < 2 )
         cols=rows;
      end;
      if ( nargin_local < 3 ),
         bound=5;
      end;
   else
      nargin_local=nargin_local+1;
      if ( nargin_local == 5 )
         pel=el;
      end;
      if ( nargin_local >= 4 ),
         el=bound;
      end;
      if ( nargin_local >= 3 )
         bound=cols;
      else
         bound=5;
      end;
      cols=rows(2);
      rows=rows(1);
   end;
end;
if ( rows*cols == 0 )
   matrix=zeros(rows,cols);
   return;
end;
bound=floor(bound);
matrix=floor((2*bound+1)*rand(rows,cols))-bound;
if ( nargin_local > 3 ),
   if ( nargin_local == 4 ),
      pel=0.2;
   end;
   index=find(rand(rows,cols)<pel);
   matrix(index)=el*ones(size(index));
end;
```

A.19 makebits

```
function A = makebits(n)

%   A = makebits(n)
%   Produce all combinations with n elements of 0s and 1s

%   This function is called by:
%       pwa2mmpsConjOvc
%       pwa2mmpsDisjOvc

r = [ 0:2^n-1 ]; A = [];
for i = 1 : n
A = [A;r];
end

c = [];
for i = 0 : n-1
c = [ 2^i c ];
end
c = c';
B = [];
for i = 1 : 2^n
B = [B c];
end
```

```
A = sign(bitand(A,B))';
```

A.20 next_comb

```
function [comb,cont]=next_comb(comb,n)

% Syntax:  [comb,cont]=next_comb(comb,n)
%
% Purpose: Calculates the next combination of the natural numbers from 1
%          to n, given the current combination comb. To get all possible
%          combinations, the initial combination should be [1:k] with
%          k the number of elements in the combination.
%          Returns cont=0 if there are no more combinations.
%
% Inputs:  comb  k element integer vector
%          n     integer
%
% Outputs: comb  k element integer vector
%          cont  boolean
%
% See also: all_combs

% Created: Oct 20, 1993       by Bart De Schutter
% Last revised

%   This function is called by
%       pwa2mmpsConjOvc
%       pwa2mmpsDisjOvc
%       MMPSgenerator

k=length(comb); max_el=[n+1-k:n]; index=find(comb==max_el); cont=1;
if ( length(index)==0 ),
   comb(k)=comb(k)+1;
else
   pos=min(index)-1;
   if ( pos==0 ),
      cont=0;
   else
      el=comb(pos)+1;
      comb([pos:k])=[el:k-pos+el];
   end;
end;
```

A.21 PWAsimulationsOvc

```
%   PWAsimulationsOvc
%   This script file continuously generates continuous PWA
%   function and checks if the transformations between the
%   PWA into the equivalent MMPS and vice versa are correct

%   A. Frau
%   7/9/07

m = [];
no_test = zeros(1,3);
count = 0;  tested = 0;    discarded = 0;
ok = 0; no = 0;
ok_dimension = zeros(1,3);      no_dimension = zeros(1,3);
ok_affine_terms = zeros(1,10);    no_affine_terms = zeros(1,10);

while(1)
    try
    %   one group of continuous PWA functions with the same
    %   polyhedral partition;
    pwa_set = continuousPWAgenerator;
        for i = 1:length(pwa_set)
            %   number of affine terms between 3 and 10
            if pwa_set{i}.N_affine_terms <= 2 ||...
                    pwa_set{i}.N_affine_terms >= 11
                continue
            end
            mmpsc = pwa2mmps(pwa_set{i},'conj','Ovc');
            mmpsd = pwa2mmps(pwa_set{i},'disj','Ovc');
            pwac = mmps2pwa(mmpsc);
            pwad = mmps2pwa(mmpsd);
            count = count + 1;
            m(count,1) = isequalPWA(pwa_set{i},pwac);
            m(count,2) = isequalPWA2MMPSovc(pwad,mmpsc);
            m(count,3) = isequalMMPS(mmpsc,mmpsd);
            tested = tested + 1;
            for j = 1:3
                if m(count,j) == 0
                    no_test(j) = no_test(j) + 1;
                end
            end
            if all(m(count,:))
                ok = ok + 1;
                ok_dimension(pwa_set{i}.dimension) =...
                    ok_dimension(pwa_set{i}.dimension) + 1;
                ok_affine_terms(pwa_set{i}.N_affine_terms) =...
                    ok_affine_terms(pwa_set{i}.N_affine_terms) + 1;
            else
                no = no + 1;
                no_dimension(pwa_set{i}.dimension) =...
                    no_dimension(pwa_set{i}.dimension) + 1;
                no_affine_terms(pwa_set{i}.N_affine_terms) =...
                    no_affine_terms(pwa_set{i}.N_affine_terms) + 1;
            end
            disp('.');
        end
    catch
        discarded = discarded + 1;
        disp(lasterr);
    end
end
```

A.22 PWAsimulationsGor

```
%   This script file continuously generates continuous PWA
%   function and checks if the transformations between the
%   PWA into the equivalent MMPS and vice versa are correct

%   A. Frau
%   7/9/07

m = [];
no_test = zeros(1,3);
count = 0;  tested = 0; discarded = 0;
ok = 0; no = 0;
ok_dimension = zeros(1,3);      no_dimension = zeros(1,3);
ok_affine_terms = zeros(1,10);    no_affine_terms = zeros(1,10);

while(1)
    try
    %   one group of continuous PWA functions with the same
    %   polyhedral partition
    pwa_set = continuousPWAgenerator;
        for i = 1:length(pwa_set)
            if pwa_set{i}.N_affine_terms <= 2 || pwa_set{i}.N_affine_terms >= 11
                continue
            end
            mmpsc = pwa2mmps(pwa_set{i},'conj','Gor');
            mmpsd = pwa2mmps(pwa_set{i},'disj','Gor');
            pwac = mmps2pwa(mmpsc);
            pwad = mmps2pwa(mmpsd);
            count = count + 1;
            m(count,1) = isequalPWA(pwa_set{i},pwac);
            m(count,2) = isequalPWA2MMPSgor(pwad,mmpsc);
            m(count,3) = isequalMMPS(mmpsc,mmpsd);
            tested = tested + 1;
            for j = 1:3
                if m(count,j) == 0
                    no_test(j) = no_test(j) + 1;
                end
            end
            if all(m(count,:))
                ok = ok + 1;
                ok_dimension(pwa_set{i}.dimension) = ...
                    ok_dimension(pwa_set{i}.dimension) + 1;
                ok_affine_terms(pwa_set{i}.N_affine_terms) = ...
                    ok_affine_terms(pwa_set{i}.N_affine_terms) + 1;
            else
                no = no + 1;
                no_dimension(pwa_set{i}.dimension) = ...
                    no_dimension(pwa_set{i}.dimension) + 1;
                no_affine_terms(pwa_set{i}.N_affine_terms) = ...
                    no_affine_terms(pwa_set{i}.N_affine_terms) + 1;
            end
            disp('.');
        end
    catch
        discarded = discarded + 1;
        disp(lasterr);
    end
end
```

A.23 MMPSsimulations

```
%   This script file continuously generates MMPS function and check
%   if the transformations between the MMPS into the equivalent
%   continuous PWA and vice versa are correct

%   A. Frau
%   13/9/07

m = [];
no_test = zeros(1,3);
```

```
count = 0;  tested = 0; discarded = 0;
ok = 0; no = 0;
ok_conj = 0;    ok_disj = 0;
no_conj = 0;    no_disj = 0;
ok_dimension = zeros(1,3);      no_dimension = zeros(1,3);
ok_affine_terms = zeros(1,10);    no_affine_terms = zeros(1,10);

while(1)
    try
        mmpsOut = MMPSgenerator;
        mmps = testMMPS(mmpsOut{1});
        pwa = mmps2pwa(mmps);
        mmpso = pwa2mmps(pwa,mmps.form,'Ovc');
        mmpsg = pwa2mmps(pwa,mmps.form,'Gor');
        count = count + 1;
        m(count,1) = isequalMMPS(mmps,mmpso);
        m(count,2) = isequalMMPS(mmps,mmpsg);
        m(count,3) = isequalPWA2MMPSgor(pwa,mmpso);
        tested = tested + 1;
        for j = 1:3
            if m(count,j) == 0
                no_test(j) = no_test(j) + 1;
            end
        end
        if all(m(count,:))
            ok = ok + 1;
            ok_dimension(mmps.dimension) = ...
                ok_dimension(mmps.dimension) + 1;
            ok_affine_terms(mmps.N_affine_terms) = ...
                ok_affine_terms(mmps.N_affine_terms) + 1;
            if strcmp(mmps.form,'disj')
                ok_disj = ok_disj + 1;
            else
                ok_conj = ok_conj + 1;
            end
        else
            no = no + 1;
            no_dimension(mmps.dimension) = ...
                no_dimension(mmps.dimension) + 1;
            no_affine_terms(mmps.N_affine_terms) = ...
                no_affine_terms(mmps.N_affine_terms) + 1;
            if strcmp(mmps.form,'disj')
                no_disj = no_disj + 1;
            else
                no_conj = no_conj + 1;
            end
        end
        disp('.');
    catch
        discarded = discarded + 1;
        disp(lasterr);
    end
end
```

A.24 PWA_example

```
%   PWA_example

PWA_ex.alfa = {0,3,-1,-3,4,-4,3,1,-3};
PWA_ex.beta = {3,24,0,-3,4,4,-3,0,24};
PWA_ex.H = {{[-1;1],[-1;1],[-1;1],[-1;1]},[-1;1],[-1;1],...
    [-1;1],[-1;1],[-1;1],[-1;1],[-1;1],[-1;1]};
PWA_ex.K = {{[20;-7],[3;-2],[-2;3],[-7;20]},[7;-6],[6;-3],...
    [2;-1],[1;0],[0;1],[-1;2],[-3;6],[-6;7]};

%   Gorokhovik_Zorko strategy
MMPS_ex_GOR = pwa2mmps(PWA_ex,'disj','Gor');

%   Ovchinnikov strategy
MMPS_ex_OVC = pwa2mmps(PWA_ex,'disj','Ovc');
```

A.25 MMPS_example

```
%   MMPS_example

MMPS_ex.alfa = {0,3,-1,-3,4,-4,3,1,-3};
MMPS_ex.beta = {3,24,0,-3,4,4,-3,0,24};
MMPS_ex.terms = {[7 8 9],[2 3 4],[5 6],[1 7],[1 4]};
MMPS_ex.form = 'disj';
MMPS_ex.A = [-1;1];
MMPS_ex.b = [20;20];

PWA_ex_Out = mmps2pwa(MMPS_ex);
```

Bibliography

[1] A. Bemporad, F. Borrelli, and M. Morari. The Explicit Solution of Constrained LP-Based Receding Horizon Control. In *IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[2] A. Bemporad and M. Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica*, 35(3):407–427, March 1999.

[3] G.J. Benschop. Model predictive control for a class of hybrid systems. Master's thesis, Delft University of Technology, 2001.

[4] B. De Schutter and B. De Moor. The extended linear complementarity problem. *Mathematical Programming*, 71(3):289–325, December 1995.

[5] B. De Schutter and T.J.J. van den Boom. On model predictive control for max-min-plus-scaling discrete event systems. Technical Report bds:00-04, Control Systems Engineering, Fac. of Information Technology and Systems, Delft University of Technology, Delft, The Netherlands, June 2000.

[6] B. De Schutter and T.J.J. van den Boom. Model predictive control for max-min-plus-scaling systems. In *Proceedings of the 2001 American Control Conference*, pages 319–324, Arlington, Virginia, June 2001.

[7] B. De Schutter and T.J.J. van den Boom. MPC for continuous piecewise-affine systems. *Systems & Control Letters*, 52(3–4):179–192, July 2004.

[8] T. Gal. *Postoptimal Analyses, Parametric Programming, and Related Topics.* Walter de Gruyter, Berlin, 1995.

[9] T. Geyer, F.D. Torrisi, and M. Morari. Optimal complexity reduction of piecewise affine models based on hyperplane arrangements. In *Proceeding of the 2004 American Control Conference*, pages 1190–1195, Boston, USA, July 2004.

[10] V.V. Gorokhovik and O.I. Zorko. Piecewise affine functions and polyhedral sets. *Optimization*, 31:209–221, 1994.

[11] W.P.M.H. Heemels, B. De Schutter, and A. Bemporad. Equivalence of hybrid dynamical models. *Automatica*, 37(7):1085–1091, July 2001.

[12] W.P.M.H. Heemels, B. De Schutter, and A. Bemporad. On the equivalence of classes of hybrid dynamical models. In *Proceedings of the 40th IEEE Conference on Decision and Control*, pages 364–369, Orlando, Florida, December 2001.

[13] M. Kvasnica, P. Grieder, M. Baotić, and F.J. Christophersen. *Multi-Parametric Toolbox (MPT)*, March 2006.

[14] S. Ovchinnikov. Max-min representation of piecewise linear functions. *Beiträge zur Algebra und Geometrie/Contributions to Algebra and Geometry*, 43(1):297–302, 2002.

[15] R.T. Rockafellar. *Convex Analysis.* Princeton University Press, Princeton, 1972.

[16] E.D. Sontag. Nonlinear regulation: the piecewise linear approach. *IEEE Transactions on Automatic Control*, 26(2):346–357, April 1981.

[17] A.J. van der Schaft and J.M. Schumacher. Complementarity modelling of hybrid systems. *IEEE Transactions on Automatic Control*, 43:483–490, 1998.