Multi-Parametric Toolbox (MPT)

M. Kvasnica[*,†], P. Grieder[*], M. Baotić[*] and F.J. Christophersen[*]

March 30, 2006

[*]Institut für Automatik, ETH - Swiss Federal Institute of Technology, CH-8092 Zürich
[†]Corresponding Author: E-mail: kvasnica@control.ee.ethz.ch, Tel. +41 01 632 4274

Contents

1

Introduction

Optimal control of constrained linear and piecewise affine (PWA) systems has garnered great interest in the research community due to the ease with which complex problems can be stated and solved. The aim of the *Multi-Parametric Toolbox* ($\mathbb{MPT}$) is to provide efficient computational means to obtain feedback controllers for these types of constrained optimal control problems in a MATLAB [27] programming environment. By multi-parametric programming, a linear or quadratic optimization problem is solved off-line. The associated solution takes the form of a PWA state feedback law. In particular, the state-space is partitioned into polyhedral sets and for each of those sets the optimal control law is given as one affine function of the state. In the on-line implementation of such controllers, computation of the controller action reduces to a simple set-membership test, which is one of the reasons why this method has attracted so much interest in the research community.

As shown in [8] for quadratic objectives, a feedback controller may be obtained for constrained linear systems by applying multi-parametric programming techniques. The linear objective was tackled in [4] by the same means. The multi-parametric algorithms for constrained finite time optimal control (CFTOC) of linear systems contained in the $\mathbb{MPT}$ are based on [1] and are similar to [28]. Both [1] and [28] give algorithms that are significantly more efficient than the original procedure proposed in [8].

It is current practice to approximate the constrained infinite time optimal control (CITOC) by receding horizon control (RHC) - a strategy where CFTOC problem is solved at each time step, and then only the initial value of the optimal input sequence is applied to the plant. The main problem of RHC is that it does not, in general, guarantee stability. In order to make receding horizon control stable, conditions (e.g., terminal set constraints) have to be added to the original problem which may result in degraded performance [25, 24]. The extensions to make RHC stable are part of the $\mathbb{MPT}$. It is furthermore possible to impose a minimax optimization objective which allows for the computation of robust controllers for linear systems subject to polytopic and additive uncertainties [6, 19]. As an alternative to computing suboptimal stabilizing controllers, the procedures to compute the infinite time optimal solution for constrained linear systems [13] are also provided.

Optimal control of piecewise affine systems has also received great interest in the research community since PWA systems represent a powerful tool for approximating non-linear systems and because of their equivalence to hybrid systems [17]. The algorithms for computing the feedback controllers for constrained PWA systems were presented for quadratic and linear objectives in [10] and [3] respectively, and are also included in this toolbox. Instead of computing the feedback controllers which minimize a finite time cost objective, it is also possible to obtain the infinite time optimal solution for PWA systems [2].

Even though the multi-parametric approaches rely on off-line computation of a feedback law, the computation can quickly become prohibitive for larger problems. This is not only due to the high complexity of the multi-parametric programs involved, but mainly because of the exponential number of transitions between regions which can occur when a controller is computed in a dynamic programming fashion [10, 20]. The $\mathbb{MPT}$ therefore also includes schemes to obtain controllers of low complexity for linear and PWA systems as presented in [15, 14, 16].

2

Installation

2.1 Installation

Remove any previous copy of $\mathbb{MPT}$ from your disk before installing any new version!

The $\mathbb{MPT}$ toolbox consists of the following directories

`mpt/`	toolbox main directory
`mpt/@mptctrl`	directory of the `mptctrl` object
`mpt/@polytope`	directory of the `polytope` object
`mpt/examples`	documentation
`mpt/examples`	sample dynamical systems
`mpt/extras`	auxiliary routines
`mpt/solvers`	different solvers

In order to use $\mathbb{MPT}$, set a Matlab path to the whole `mpt/` directory and to all it's subdirectories. If you are using Matlab for Windows, go to the "File - Set Path..." menu, choose "Add with Subfolders..." and pick up the $\mathbb{MPT}$ directory. Click on the "Save" button to store the updated path setting. Under Unix, you can either manually edit the file "startup.m", or to use the same procedure described above.

Once you install the toolbox, please consult Section 3 on how to set default values of certain parameters.

To explore functionality of $\mathbb{MPT}$, try one of the following:

```
help mpt
help mpt/polytope
help mpt_sysStruct
help mpt_probStruct
mpt_demo1
mpt_demo2
mpt_demo3
mpt_demo4
mpt_demo5
```

```
mpt_demo6
```

```
runExample
```

MPT toolbox comes with a set of pre-defined examples which the user can go through to get familiar with basic features of the toolbox.

If you wish to be informed about new releases of the toolbox, subscribe to our mailing list by sending an email to:

```
mpt-request@list.ee.ethz.ch
```

and put the word

```
subscribe
```

to the subject field. To unsubscribe, send an email to the same mail address and specify

```
unsubscribe
```

on the subject field.

If you have any questions or comments, or you observe buggy behavior of the toolbox, please send your reports to

```
mpt@control.ee.ethz.ch
```

2.2 Additional software requirements

LP and QP solvers

The MPT toolbox is a package primary designed to tackle multi-parametric programming problems. It relies on external Linear programming (LP) and Quadratic programming (QP) solvers. Since the LP and QP solvers shipped together with Matlab (linprog and quadprog) are rather slow, the toolbox provides a unified interface to other solvers.

One of the supported LP solvers is the free CDD package from Komei Fukuda (`http://www.cs.mcgill.ca/~fukuda/soft/cdd_home/cdd.html`)

The CDD is not only a fast and reliable LP solver, it can also solve many problems from computational geometry, e.g. computing convex hulls, extreme points of polytopes, calculating projections, etc.

A pre-compiled version of the Matlab interface to CDD is included in this release of the MPT toolbox. The interface is available for Windows, Solaris and Linux. Source code of the interface comes along with this distribution of MPT . For more details, visit `http://control.ee.ethz.ch/~hybrid/cdd.php`

Please consult Section 2.3 on how to make CDD a default LP solver for the MPT toolbox.

The NAG Foundation Toolbox for Matlab provides a fast and reliable functionality to tackle many different optimization problems. It's LP and QP solvers are fully supported by MPT .

An another alternative is the commercial CPLEX solver from ILOG. The authors provide an interface to call CPLEX directly from Matlab, you can download source codes and pre-compiled libraries for Windows, Solaris and Linux from
`http://control.ee.ethz.ch/~hybrid/cplexint.php`

Please note that you need to be in possession of a valid CPLEX license in order to use CPLEX solvers.

The free GLPK (GNU Linear Programming Kit) solver is also supported by MPT toolbox and a MEX interface is included in the distribution. You can download the latest version of GLPKMEX written by Nicolo Giorgetti from:
`http://www-dii.ing.unisi.it/~giorgetti/downloads.php`

Note that we have experienced several numerical inconsistencies when using GLPK.

Semi-definite optimization packages

Some routines of the MPT toolbox rely on Linear Matrix Inequalities (LMI) theory. Certain functions therefore require solving a semidefinite optimization problem. The YALMIP interface by Johan Lofberg `http://control.ee.ethz.ch/~joloef/`

is included in this release of MPT toolbox. Since the interface is a wrapper and calls external LMI solver, we strongly recommend to install one of the solvers supported by YALMIP. You can obtain a list of free LMI solvers here:
`http://control.ee.ethz.ch/~joloef/yalmip.php`

YALMIP supports a large variety of Semi-Definite Programming packages. One of them, namely the SeDuMi solver written by Jos Sturm, comes along with MPT . Source codes as well as binaries for Windows are included directly, you can compile the code for other operating systems by following the instructions in `mpt/solvers/SeDuMi105/Install.unix`. For more information consult `http://fewcal.kub.nl/sturm/software/sedumi.html`

Solvers for projections

MPT allows to compute orthogonal projections of polytopes. To meet this task, several methods for projections are available in the toolbox. Two such methods – ESP and Fourier-Motzkin Elimination are coded in C and need to be accessible as a mex library. These libraries are already provided in compiled form for Linux and Windows. For other architectures you will need to compile the corresponding library on your own. To do so follow instructions in `mpt/solvers/esp` and `mpt/solvers/fourier`, respectively.

2.3 Setting up default parameters

By default, it is not necessary to modify the default setting stored in `mpt_init.m`. However if you decide to do so, we strongly recommend to use the GUI setup function

```
mpt_setup
```

Any routine of the $\mathbb{MPT}$ toolbox can be called with user-specified values of certain global parameters. To make usage of $\mathbb{MPT}$ toolbox as user-friendly as possible, we provide the option to store default values of the parameters in variable `mptOptions`, which is kept in MATLAB's workspace as a global variable (i.e. it stays there unless one types `clear all`).

The variable is created when the toolbox get's initialized through a call to `mpt_init`.

Default LP solver: In order to set the default LP solver to be used, open the file `mpt_init.m` in your editor. Scroll down to the following line:

```
mptOptions.lpsolver = [];
```

Integer value on the right-hand side specifies the default LP solver. Allowed values are:

0	NAG Foundation LP solver
3	CDD Criss-Cross Method
2	CPLEX
4	GLPK
5	CDD Dual-Simplex Method
1	linprog

If the argument is empty, the fastest available solver will be enabled. Solvers presented in the table above are sorted in the order of preference.

Default QP solver: To change the default QP solver, locate and modify this line in `mpt_init.m`:

```
mptOptions.qpsolver = [];
```

Allowed values for the right-hand side argument are the following:

0	NAG Foundation QP solver
2	CPLEX
1	quadprog

Again, if there is no specification provided, the fastest alternative will be used.

Note: Quadratic Program solver is not necessarily required by $\mathbb{MPT}$. If you are not in possession of any QP solver, you still will be able to use large part of functionality involved in the toolbox. But the optimization problems will be limited to linear performance objectives.

Default solver for extreme points computation: Some of the functions in $\mathbb{MPT}$ toolbox require computing of extreme points of polytopes given by their H-representation and calculating convex hulls of given vertices respectively. Since efficient analytical methods are limited to low dimensions only, we provide the possibility to pass this computation to an external software package (CDD). However, if the user for any reason does not want to use third-party tools, the problem can still be tackled in an analytical way (with all the limitations mentioned earlier).

To change the default method for extreme points computation, locate the following line in `mpt_init.m`:

```
mptOptions.extreme_solver = [];
```

and change the right-hand side argument to one of these values:

3 CDD (faster computation, works also for higher dimensions)
0 Analytical computation (limited to dimensions up to 3)

Default tolerances: The Multi-Parametric Toolbox internally works with 2 types of tolerances: - absolute tolerance - relative tolerance

Default values for these two constants can be set by modifying the following lines of `mpt_init.m`:

```
mptOptions.rel_tol = 1e-6;

mptOptions.abs_tol = 1e-7;
```

Default values for Multi-parametric solvers: Solving a given QP/LP in a multi-parametric way involves making ”steps” across given boundaries. Length of this ”step” is given by the following variable:

```
mptOptions.step_size = 1e-4;
```

Due to numerical problems tiny regions are sometimes difficult to calculate, i.e. are not identified at all. This may create ”gaps” in the computed control law. For the exploration, these will be jumped over and the exploration in the state space will continue. See [1] for details.

Level of detecting those gaps is given by the following variable:

```
mptOptions.debug_level = 1;
```

The right-hand side argument can have three values:

1 No debug done

2 A tolerance is given to find gap in the region partition, small empty regions inside the region partition will be discarded. Note that this is generally not a problem, since the feedback law is continuous and can therefore be interpolated easily. Correction to the calculation of the outer hull is performed as well.

3 Zero tolerance to find gaps in the region partition, empty regions if they exist, will be detected, i.e. the user will be notified. Correction to the calculation of the outer hull is performed.

Default Infinity-box: $\mathbb{MPT}$ internally converts the $\mathcal{R}^n$ to a box with large bounds. The following parameter specifies size of this box:

```
mptOptions.infbox = 1e4;
```

Note that also polyhedra (unbounded polytopes) are converted to bounded polytopes by making an intersection with the "Infinity-box".

Default values for plotting: The overloaded `plot` function can be forced to open a new figure windows every time the user calls it. If you want to disable this feature, go to the following line in `mpt_init.m`:

```
mptOptions.newfigure = 0;
```

and change the constant to 0 (zero)

1 means "enabled", 0 stands for "disabled"

Default level of verbosity: Text output from functions can be limited or suppressed totally by changing the following option in `mpt_init.m`:

```
mptOptions.display = 1;
```

Allowed values are:

0	only important messages
1	displays also intermediate information
2	no output suppression

Level of details: Defines how many details about the solution should be stored in the resulting controller structure. This can have a significant impact on the size of the controller structure. If you want to evaluate open-loop solution for PWA systems, set this to 1. Otherwise leave the default value to save memory and disk space.

```
mptOptions.details = 0;
```

Once you modify the `mpt_init.m` file, type:

```
mpt_init
```

to initialize the toolbox.

3

Theory of Polytopes and Multi-Parametric Programming

3.1 Polytopes

Polytopic (or, more general, polyhedral) sets are an integral part of multi-parametric programming. For this reason we give some of the definitions and fundamental operations with polytopes. For more details we refer reader to [30, 12].

Definition 3.1.1 (polyhedron)**:** A convex set $\mathcal{Q} \subseteq \mathbb{R}^n$ given as an intersection of a finite number of closed half-spaces

$$\mathcal{Q} = \{x \in \mathbb{R}^n \mid Q^x x \leq Q^c\}, \tag{3.1}$$

is called *polyhedron*.

Definition 3.1.2 (polytope)**:** A bounded polyhedron $\mathcal{P} \subset \mathbb{R}^n$

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid P^x x \leq P^c\}, \tag{3.2}$$

is called *polytope*.

It is obvious from the above definitions that every polytope represents a convex, compact (i.e., bounded and closed) set. We say that a polytope $\mathcal{P} \subset \mathbb{R}^n$, $\mathcal{P} = \{x \in \mathbb{R}^n \mid P^x x \leq P^c\}$ is *full dimensional* if $\exists x \in \mathbb{R}^n : P^x x < P^c$. Furthermore, if $\|(P^x)_i\| = 1$, where $(P^x)_i$ denotes i-th row of a matrix P^x, we say that the polytope $\mathcal{P}$ is *normalized*. One of the fundamental properties of a polytope is that it can also be described by its vertices

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid x = \sum_{i=1}^{v_P} \alpha_i V_P^{(i)},\ 0 \leq \alpha_i \leq 1,\ \sum_{i=1}^{v_P} \alpha_i = 1\}, \tag{3.3}$$

where $V_P^{(i)}$ denotes the i-th vertex of $\mathcal{P}$, and v_P is the total number of vertices of $\mathcal{P}$.

We will henceforth refer to the half-space representation (3.2) and vertex representation (3.3) as $\mathcal{H}$ and $\mathcal{V}$ representation respectively.

Definition 3.1.3 (face)**:** Linear inequality $a'x \leq b$ is called *valid* for a polyhedron $\mathcal{P}$ if $a'x \leq b$ holds for all $x \in \mathcal{P}$. A subset of a polyhedron is called a *face* of $\mathcal{P}$ if it is represented as

$$\mathcal{F} = \mathcal{P} \cap \{x \in \mathbb{R}^n \mid a'x = b\}, \tag{3.4}$$

for some valid inequality $a'x \leq b$. The faces of polyhedron $\mathcal{P}$ of dimension 0, 1, $(n-2)$ and $(n-1)$ are called vertices, edges, ridges and facets, respectively.

We say that a polytope $\mathcal{P} \subset \mathbb{R}^n$, $\mathcal{P} = \{x \in \mathbb{R}^n \mid P^x x \leq P^c\}$ is in a *minimal representation* if a removal of any of the rows in $P^x x \leq P^c$ would change it (i.e., there are no redundant halfspaces). It is straightforward to see that a normalized, full dimensional polytope $\mathcal{P}$ has a *unique* minimal representation. This fact is very useful in practice. Normalized, full dimensional polytopes in a minimal representation allow us to avoid any ambiguity when comparing them and very often speed-up other polytope manipulations. We will now define some of the basic manipulations on polytopes.

3.2 Basic Polytope Manipulation

The Set-Difference of two polytopes $\mathcal{P}$ and $\mathcal{Q}$ is a union of polytopes $\mathcal{R} = \bigcup_i \mathcal{R}_i$

$$\mathcal{R} = \mathcal{P} \setminus \mathcal{Q} := \{x \in \mathbb{R}^n \mid x \in \mathcal{P}, x \notin \mathcal{Q}\}. \tag{3.5}$$

The Pontryagin-Difference of two polytopes $\mathcal{P}$ and $\mathcal{W}$ is a polytope

$$\mathcal{P} \ominus \mathcal{W} := \{x \in \mathbb{R}^n \mid x + w \in \mathcal{P},\ \forall w \in \mathcal{W}\}. \tag{3.6}$$

The Minkowski-Addition of two polytopes $\mathcal{P}$ and $\mathcal{W}$ is a polytope

$$\mathcal{P} \oplus \mathcal{W} := \{x + w \in \mathbb{R}^n \mid x \in \mathcal{P},\ w \in \mathcal{W}\}. \tag{3.7}$$

The convex hull of a union of polytopes $\mathcal{P}_i \subset \mathbb{R}^n$, $i = 1, \ldots, p$, is a polytope

$$\text{hull}\left(\bigcup_{i=1}^{p} \mathcal{P}_i\right) := \{x \in \mathbb{R}^n \mid x = \sum_{i=1}^{p} \alpha_i x_i,\ x_i \in \mathcal{P}_i,\ 0 \leq \alpha_i \leq 1,\ \sum_{i=1}^{p} \alpha_i = 1\}. \tag{3.8}$$

The envelope of two $\mathcal{H}$-polyhedra $\mathcal{P} = \{x \in \mathbb{R}^n \mid P^x x \leq P^c\}$ and $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Q^x x \leq Q^c\}$ is an $\mathcal{H}$-polyhedron

$$\text{env}(\mathcal{P}, \mathcal{Q}) = \{x \in \mathbb{R}^n \mid \bar{P}^x x \leq \bar{P}^c,\ \bar{Q}^x x \leq \bar{Q}^c\}, \tag{3.9}$$

where $\bar{P}^x x \leq \bar{P}^c$ is the subsystem of $P^x x \leq P^c$ obtained by removing all the inequalities not valid for the polyhedron $\mathcal{Q}$, and $\bar{Q}^x x \leq \bar{Q}^c$ are defined in the similar way with respect to $Q^x x \leq Q^c$ and $\mathcal{P}$ [7].

3.3 Multi-Parametric Programming

This section first covers some of the fundamentals of multi-parametric programming for linear systems before restating results for PWA systems. Consider a discrete-time linear time-invariant system

$$
\begin{aligned}
x(t+1) &= Ax(t) + Bu(t) && \text{(3.10a)}\\
y(t) &= Cx(t) + Du(t) && \text{(3.10b)}
\end{aligned}
$$

with $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$. Let $x(t)$ denote the state at time t and $x_{t+k|t}$ denote the predicted state at time $t+k$ given the state at time t. For brevity we denote $x_{k|0}$ as x_k. Let u_k be the computed input for time k, given $x(0)$. Assume now that the states and the inputs of the system in (3.10) are subject to the following constraints

$$x \in \mathbb{X} \subset \mathbb{R}^n, \qquad u \in \mathbb{U} \subset \mathbb{R}^m \tag{3.11}$$

where $\mathbb{X}$ and $\mathbb{U}$ are compact polyhedral sets containing the origin in their interior, and consider the constrained finite-time optimal control (CFTOC) problem

$$
\begin{aligned}
J_N^*(x(0)) &= \min_{u_0,\dots,u_{N-1}} ||Q_f x_N||_\ell + \sum_{k=0}^{N-1} ||Ru_k||_\ell + ||Qx_k||_\ell && \text{(3.12a)}\\
\text{subj. to} \quad & x_k \in \mathbb{X}, \qquad \forall k \in \{1,\dots,N\}, && \text{(3.12b)}\\
& x_N \in \mathcal{X}_{set}, && \text{(3.12c)}\\
& u_k \in \mathbb{U}, \qquad \forall k \in \{0,\dots,N-1\}, && \text{(3.12d)}\\
& x_0 = x(0),\ x_{k+1} = Ax_k + Bu_k,\ \forall k \in \{0,\dots,N-1\}, && \text{(3.12e)}\\
& \begin{cases} Q = Q' \succeq 0,\ \ Q_f = Q_f' \succeq 0,\ \ R = R' \succ 0, & \text{if} \quad \ell = 2,\\ \operatorname{rank}(Q) = n,\ \ \operatorname{rank}(R) = m, & \text{if} \quad \ell \in \{1,\infty\}. \end{cases} && \text{(3.12f)}
\end{aligned}
$$

where (3.12c) is a user defined set-constraint on the final state which may be chosen such that stability of the closed-loop system is guaranteed [24]. The cost (3.12a) may be linear (e.g., $\ell \in \{1,\infty\}$) [4] or quadratic (e.g., $\ell = 2$) [8] whereby the matrices Q, R and Q_f represent user-defined weights on the states and inputs.

Definition 3.3.1: We define the N-step feasible set $\mathcal{X}_f^N \subseteq \mathbb{R}^n$ as the set of initial states $x(0)$ for which the CFTOC problem (3.12) is feasible, i.e.

$$\mathcal{X}_f^N = \{x(0) \in \mathbb{R}^n \mid \exists (u_0,\dots,u_{N-1}) \in \mathbb{R}^{Nm},\ x_k \in \mathbb{X},\ u_{k-1} \in \mathbb{U},\ \forall k \in \{1,\dots,N\}\}. \tag{3.13}$$

For a given initial state $x(0)$, problem (3.12) can be solved as an LP or QP for linear or quadratic cost objectives respectively. However, this type of on-line optimization may be prohibitive for control of fast processes.

By substituting $x_k = A^k x(0) + \sum_{j=0}^{k-1} A^k B u_{k-1-j}$, problem (3.12) for the quadratic cost objective can be reformulated as

$$
\begin{aligned}
J_N^*(x(0)) &= x(0)'Yx(0) + \min_{U_N} \left\{ U_N' H U_N + x(0)' F U_N \right\}\\
s.t. \quad & GU_N \leq W + Ex(0)
\end{aligned} \tag{3.14}
$$

where the column vector $U_N \triangleq [u'_0, \dots, u'_{N-1}]' \in \mathbb{R}^s$ is the optimization vector, $s \triangleq mN$ and H, F, Y, G, W, E are easily obtained from Q, R, Q_f, (3.10) and (3.11) (see [8] for details). The same transformation can trivially be applied to linear cost objectives in (3.12a). Because problem (3.14) depends on $x(0)$, it can be also solved as a multi-parametric program [8]. Denoting with $U_N = [u'_0, \dots, u'_{N-1}]'$ the optimization vector and considering $x(0)$ as a parameter, problem (3.12) can then be solved for all parameters $x(0)$ to obtain a feedback solution with the following properties,

Theorem 3.3.2: [8, 9] Consider the CFTOC problem (3.12). Then, the set of feasible parameters $\mathcal{X}_f^N$ is convex, the optimizer $U_N^* : \mathcal{X}_f^N \to \mathbb{R}^{Nm}$ is continuous and piecewise affine (PWA), i.e.

$$U_N^*(x(0)) = F_r x(0) + G_r \quad \text{if} \quad x(0) \in \mathcal{P}_r = \{x \in \mathbb{R}^n | H_r x \le K_r\},\ r = 1, \dots, R \tag{3.15}$$

and the optimal cost $J_N^* : \mathcal{X}_f^N \to \mathbb{R}$ is continuous, convex and piecewise quadratic ($\ell = 2$) or piecewise linear ($\ell \in \{1, \infty\}$).

According to Theorem 3.3.2, the feasible state space $\mathcal{X}_f^N$ is partitioned into R polytopic regions, i.e., $\mathcal{X}_f^N = \{\mathcal{P}_r\}_{r=1}^R$. Though the initial approach was presented in [8], more efficient algorithms for the computation are given in [1, 28]. With sufficiently large horizons or appropriate terminal set constraints (3.12c) the closed-loop system is guaranteed to be stabilizing for receding horizon control [13, 24]. However, no robustness guarantees can be given. This issue is addressed in [19, 6] where the authors present minimax methods which are able to cope with additive disturbances

$$x(t+1) = Ax(t) + Bu(t) + w, \qquad w \in \mathcal{W}, \tag{3.16}$$

where $\mathcal{W}$ is a polytope with the origin in its interior. The minimax approach can be applied also when there is polytopic uncertainty in the system dynamics,

$$x(t+1) \quad = \quad A(\lambda)x(t) + B(\lambda)u(t), \tag{3.17}$$

with $\lambda \in \mathbb{R}^L$ and

$$\Omega := \text{conv}\{[A^{(1)}|B^{(1)}], [A^{(2)}|B^{(2)}], \dots, [A^{(L)}|B^{(L)}]\}, \tag{3.18a}$$

$$[A(\lambda)|B(\lambda)] \in \Omega, \tag{3.18b}$$

i.e., there exist L nonnegative coefficients $\lambda_l \in \mathbb{R}$ ($l = 1, \dots, L$) such that

$$\sum_{l=1}^{L} \lambda_l = 1\ , \qquad [A(\lambda)|B(\lambda)] = \sum_{l=1}^{L} \lambda_l [A^{(l)}|B^{(l)}]. \tag{3.19}$$

The set of admissible λ can be written as $\Lambda := \{x \in [0,1]^L \mid ||x||_1 = 1\}$. In order to guarantee robust stability of the closed loop system, the objective (3.12a) is modified such that the feedback law which minimizes the worst case is computed, hence the name *minimax* control.

The results in [8] were extended in [5, 10, 3] to compute the optimal explicit feedback controller for PWA systems of the form

$$\begin{aligned} x(k+1) \quad &= \quad A_i x(k) + B_i u(k) + f_i, && (3.20a)\\ & \quad\ L_i x(k) + E_i u(k) \le W_i, \quad i \in I && (3.20b)\\ &\text{if} \quad [x'(k) \ \ u'(k)]' \in \mathcal{D}_i && (3.20c) \end{aligned}$$

whereby the dynamics (3.20a) with the associated constraints (3.20b) are valid in the polyhedral set $\mathcal{D}_i$ defined in (3.20c). The set $I \subset \mathbb{N}$, $I = \{1, \ldots, d\}$ represents all possible dynamics, and d denotes the number of different dynamics. Henceforth, we will abbreviate (3.20a) and (3.20c) with $x(k+1) = f_{PWA}(x(k), u(k))$. Note that we do not require $x(k+1) = f_{PWA}(x(k), u(k))$ to be continuous. The optimization problem considered here is given by

$$
\begin{aligned}
J_N^*(x(0)) \quad &= \quad \min_{u_0,\ldots,u_{N-1}} ||Q_f x_N||_\ell + \sum_{k=0}^{N-1} ||Ru_k||_\ell + ||Qx_k||_\ell && \text{(3.21a)}\\
\text{subj. to} \quad & L_i x_k + E_i u_k \leq W_i, \text{ if } [x_k \;\; u_k]' \in \mathcal{D}_i,\ i \in I,\ \forall k \in \{0, \ldots, N-1\}, && \text{(3.21b)}\\
& x_N \in \mathcal{X}_{set}, && \text{(3.21c)}\\
& x_{k+1} \; = \; f_{PWA}(x_k, u_k),\ x_0 = x(0),\ \forall k \in \{0, \ldots, N-1\}, && \text{(3.21d)}\\
& \begin{cases} Q = Q' \succeq 0,\ \ Q_f = Q_f' \succeq 0,\ \ R = R' \succ 0, & \text{if } \ \ell = 2, \\ \operatorname{rank}(Q) = n,\ \ \operatorname{rank}(R) = m, & \text{if } \ \ell \in \{1, \infty\}. \end{cases} && \text{(3.21e)}
\end{aligned}
$$

Here (3.21c) is a user-specified set constraint on the terminal state which may be used to guarantee stability [23, 14, 9]. As an alternative, the infinite horizon solution to (3.21) guarantees stability as well [2]. In order to robustify controllers with respect to additive disturbances, a minimax approach is taken [20] which is identical to what was proposed for linear systems [19, 9].

All multi-parametric programming methods suffer from the curse of dimensionality. As the prediction horizon N increases, the number of partitions R ($\mathcal{X}_f^N = \{\mathcal{P}_r\}_{r=1}^R$) grows exponentially making the computation and application of the solution intractable. Therefore, there is a clear need to reduce the complexity of the solution. This was tackled in [16, 15, 14] where the authors present two methods for obtaining feedback solutions of low complexity for constrained linear and PWA systems. The first controller drives the state in minimum time into a convex set $\mathcal{X}_{set}$, where the cost-optimal feedback law is applied [15, 14]. This is achieved by iteratively solving one-step multi-parametric optimization problems. Instead of solving one problem of size N, the algorithm solves N problems of size 1, thus the decrease in both on- and off-line complexity. This scheme guarantees closed-loop stability. If a linear system is considered, an even simpler controller may be obtained by solving only one problem of size 1, with the additional constraint that $x_1 \in \mathcal{X}_f^N$ [15, 16]. In order to guarantee stability of this closed-loop system, an LMI analysis is performed which aims at identifying a Lyapunov function [18, 11].

4

MPT in 15 minutes

This short introduction is not meant to (and does not) replace the $\mathbb{MPT}$ manual. It serves to clarify the key points of Model Predictive Control and application thereof within the framework of the $\mathbb{MPT}$ toolbox. Specifically, the main problems which arise in practice are illustrated in a concise manner without going into the technical details.

4.1 First Steps

Before reading the rest of this introduction, have a close look at the provided demonstrations and go through them slowly. At the Matlab command prompt, type `mpt_demo1`, `mpt_demo2`, ..., `mpt_demo6`. After completing the demos, run some examples by typing `runExample` at the command prompt. More demos can be found in the `mpt/examples/ownmpc` and `mpt/examples/nonlin` directories of your $\mathbb{MPT}$ installation. Finally, for a good overview, type `help mpt` and `help polytope` to get the list and short descriptions of (almost) all available functions.

4.2 How to Obtain a Tractable State Feedback Controller

In this section the regulation problem will be treated. See the subsequent section for the special case of tracking.

Guidelines for Modelling a Dynamical System

The most important aspects in system modelling for $\mathbb{MPT}$ are given below:

1. Always make sure your dynamic matrices and states/inputs are well scaled. Ideally all variables exploit the full range between ± 10. See [26] for details.
2. Try to have as few different dynamics as possible when designing your PWA system model.
3. The fewer states and inputs your system model has, the easier all subsequent computations will be.
4. Use the largest possible sampling time when discretizing your system.

Control Schemes

In order to compute a controller, only one function call is needed

```
controller = mpt_control(sysStruct,probStruct)
```

For a detailed description of how to define your system `sysStruct` and problem `probStruct`, see Sections 5.7 and 6.9), respectively. We also suggest you examine the m-files in the 'Examples' directory of the $\mathbb{MPT}$ toolbox and take a closer look at the `runExample.m` file. Detailed examples for controller computations are also provided in the $\mathbb{MPT}$ manual (Section *Examples*).

Computing explicit state feedback controllers via multi-parametric programming may easily lead to controllers with prohibitive complexity (both in runtime and solution) and the following is intended to give a brief overview of the existing possibilities to obtain tractable controllers for the problems $\mathbb{MPT}$ users may face. Specifically, there are three aspects which are important in this respect: performance, stability and constraint satisfaction.

Infinite Time Optimal Control: [13, 2]
To use this method, set `probStruct.N=Inf`, `probStruct.subopt_lev=0`. This will yield the infinite time optimal controller, i.e., the best possible performance for the problem at hand. Asymptotic stability and constraint satisfaction are guaranteed and all states which are controllable (maximum controllable set) will be covered by the resulting controller. However, the complexity of the associated controller may be prohibitive. Note that the computation of this controller may take forever.

Finite Time Optimal Control [8, 3, 9, 24]
To use this method, set `probStruct.N` $\in \mathbb{N}^+ \triangleq \{1, 2, \ldots\}$ and `probStruct.subopt_lev=0`. This will yield the finite time optimal controller, i.e. performance will be N-step optimal but may not be infinite horizon optimal. The complexity of the resulting controller depends strongly on the prediction horizon (large $N \rightarrow$ complex controller). It is furthermore necessary to differentiate the following cases:

`probStruct.Tconstraint=0`: No terminal set constraint. The controller will be defined over a superset of the maximum controllable set, but no guarantees on stability or closed-loop constraint satisfaction can be given. As the prediction horizon N is increased the feasible set of states will converge to the maximum controllable set from 'the outside-in', i.e. the controlled set will shrink as N increases. To extract the set of states which satisfy the constraints for all time, call `mpt_invariantSet`. To analyze these states for stability, call `mpt_lyapunov`. Note all these functions may have prohibitive run times for large partitions.

`probStruct.Tconstraint=1`: Stabilizing terminal set is automatically computed. The resulting controller will guarantee stability and constraint satisfaction for all time, but will only cover a subset of the maximum controllable set of states. By increasing the prediction horizon, the controllable set of states will converge to the maximum controllable set from 'the inside-out', i.e. the controlled set will grow larger as N increases.

`probStruct.Tset=P`: User defined terminal set. Depending on the properties (e.g., invariance, size) of the target set P, any combination of the two cases previously described may occur.

Minimum Time Control [15, 14]
To use this method, set `probStruct.subopt_lev=1`. This will yield the minimal time controller with respect to a target set around the origin, i.e. the controller will drive the state into this set in minimal time. In general, the complexity of minimum time controllers is significantly lower than that of their $1/2/\infty$-norm cost optimal counterparts. The controller is guaranteed to cover all controllable states and asymptotic stability and constraint satisfaction are guaranteed. Note that if you choose to manually define your target set by setting `probStruct.Tset=P`, these properties may not hold.

Low Complexity Controller [15, 16]
To use this method, set `probStruct.subopt_lev=2`. This will yield a controller for a prediction horizon $N = 1$ with additional constraints which guarantee asymptotic stability and constraint satisfaction in closed-loop. The controller covers all controllable states. The complexity of this 1-step controller is generally significantly lower than all other control schemes in $\mathbb{MPT}$ which cover the maximal controllable set. However, the computation of the controller may take a long time.

Conclusion

The key influence on controller complexity are as follows

1. Prediction horizon N
2. Number of different dynamics of the PWA system
3. Dimension of state and input.
4. Type of control scheme.

Furthermore, 2-norm objectives generally yield controllers of lower complexity than their $1/\infty$-norm counterparts. Therefore, we suggest you try the control schemes in the following order to trade-off performance for complexity

1. Finite Horizon Optimal Control for small N (i.e., $N = 1, 2$); `probStruct.Tconstraint=0`
2. Low Complexity (1-step) Controller
3. Minimum Time Control
4. Finite Horizon Optimal Control for large N (i.e., $N = 9, 10$); `probStruct.Tconstraint=1`
5. Infinite Horizon Optimal Control

Note that for your specific system, the order of preference may be different, so it may yet be best if you try all.

4.3 Tracking

If you are solving a tracking problem, everything becomes more complicated. It is necessary to differentiate between the case of constant reference tracking (reference state is fixed a priori) and varying reference tracking (reference is arbitrarily time varying).

For constant reference tracking (`probStruct.xref` $\in \mathbb{R}^n$ or `probStruct.yref` $\in \mathbb{R}^p$), the problem setup reduces to a normal regulation problem where all of the observations from the previous section hold.

Time varying reference tracking (`probStruct.tracking=1`) is implemented for LTI as well as for PWA systems. For time varying reference states, it is necessary to augment the state space matrices. The process of augmenting the state update equations is performed automatically by $\mathbb{MPT}$, the following exposition is intended to give you some flavor of what is going on.

First the state vector x is extended with the reference state vector x_{ref}, i.e. the reference states are added to the dynamical model. The input which is necessary such that the state remains at the reference is not generally known. Therefore the dynamics are reformulated in Δu-form. In this framework the system input at time k is $\Delta u(k)$ whereby $u(k-1)$ is an additional state in the dynamical model, i.e. the system input can be obtained as $u(k) = u(k-1) + \Delta u(k)$. The state update equation is then given by

$$\begin{pmatrix} x(k+1) \\ u(k) \\ x_{ref}(k+1) \end{pmatrix} = \begin{pmatrix} A & B & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} x(k) \\ u(k-1) \\ x_{ref}(k) \end{pmatrix} + \begin{pmatrix} B \\ I \\ 0 \end{pmatrix} \Delta u(k)$$

Assume a 3rd order system with 2 inputs. In Δu-tracking formulation, the resulting dynamical model will have 8 states (3 system states x + 3 reference states x_{ref} + 2 input states $u(k-1)$) and 2 inputs ($\Delta u(k)$). If we solve the regulation problem for the augmented system (see previous sections) we obtain a controller which allows for time varying references. For control purposes, the reference state x_{ref} is imposed by the user, i.e. x_{ref} is set to a specific value. The regulation controller then automatically steers the state x to the reference state.

Note that time varying tracking problems are generally of high dimension, such that controller computation is expensive. If you can reduce your control objective to a regulation problem for a set of predefined reference points, we suggest you solve a sequence of fixed state tracking problems instead of the time varying tracking problem, in order to obtain computational tractability.

5

Modelling of Dynamical Systems

In this chapter we will show how to model dynamical systems in $\mathbb{MPT}$ framework. As already described before, each system is defined by means of a `sysStruct` structure which is described in more details in Section 5.7.

Behavior of a plant is in general driven by two major components: system dynamics and system constraints. Both these components has to be described in the system structure.

5.1 System dynamics

$\mathbb{MPT}$ can deal with two types of discrete-time dynamical systems:

1. Linear Time-Invariant (LTI) dynamics
2. Piecewise-Affine (PWA) dynamics

LTI dynamics

LTI dynamics can be captured by the following linear relations:

$$x(k+1) = Ax(k) + Bu(k) \tag{5.1}$$

$$y(k) = Cx(k) + Du(k) \tag{5.2}$$

where $x(k) \in \mathcal{R}^{n_x}$ is the state vector at time instance k, $x(k+1)$ denotes the state vector at time $k+1$, $u(k) \in \mathcal{R}^{n_u}$ and $y(k) \in \mathcal{R}^{n_y}$ are values of the control input and system output, respectively. A, B, C and D are matrices of appropriate dimensions, i.e. A is a $n_x \times n_x$ matrix, dimension of B is $n_x \times n_u$, C is a $n_y \times n_x$ and D a $n_y \times n_u$ matrix.

Dynamical matrices are stored in the following fields of the system structure:

```
sysStruct.A = A
sysStruct.B = B
sysStruct.C = C
sysStruct.D = D
```

Example 5.1.1: Assume a double integrator dynamics sampled at 1 second:

$$x(k+1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} u(k) \tag{5.3}$$

$$y(k) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u(k) \tag{5.4}$$

In $\mathbb{MPT}$, the above described system can be defined as follows:

```
sysStruct.A = [1 1; 0 1];
sysStruct.B = [1; 0.5];
sysStruct.C = [1 0; 0 1];
sysStruct.D = [0; 0]
```

PWA dynamics

Piecewise-Affine systems are systems whose dynamics are affine and can be different in different parts of the state-input state. In particular, they are defined by

$$x(k+1) = A_i x(k) + B_i u(k) + f_i \tag{5.5}$$

$$y(k) = C_i x(k) + D_i u(k) + g_i \tag{5.6}$$

$$\text{if } \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \in \mathcal{D}_i \tag{5.7}$$

The subindex i takes values $1 \dots N_{PWA}$, where N_{PWA} is total number of PWA dynamics defined over a polyhedral partition $\mathcal{D}$. Dimensions of matrices in (5.5)–(5.7) are summarized in Table 5.1.

Matrix	Dimension
`A`	$n_x \times n_x$
`B`	$n_x \times n_u$
`f`	$n_x \times 1$
`C`	$n_y \times n_x$
`D`	$n_y \times n_u$
`g`	$n_y \times 1$

Tab. 5.1: Dimensions of matrices of a PWA system.

Matrices in equations (5.5) and (5.6) are stored in the following fields of the system structure:

Equation (5.7) defines a polyhedral partition of the state-input space over which the different dynamics are active. Different segments of the polyhedral partition $\mathcal{D}$ are defined using so-called *guard lines*, i.e. constraints on state and input variables. In general, the guard lines are described by the following constraints:

$$G_i^x x(k) + G_i^u u(k) \leq G_i^c \tag{5.8}$$

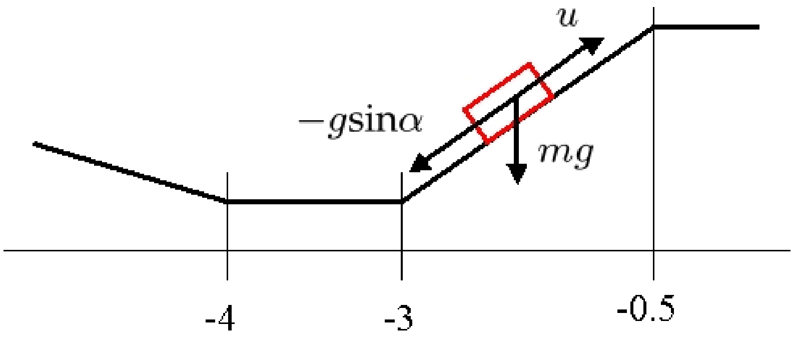


Fig. 5.1: Car moving on a PWA hill.

which means that dynamics i represented by the tuple $[A_i, B_i, f_i, C_i, D_i, g_i]$ will be active in the part of state-input space which satisfies constraints (5.8). If at future time the state $x(k+T)$ or input $u(k+T)$ moves to a different sector of the polyhedral partition, say $G_j^x x(k+T) + G_j^u u(k+T) \leq G_j^c$, the dynamics will be driven by the tuple $[A_j, B_j, f_j, C_j, D_j, g_j]$, and so on.

In $\mathbb{MPT}$, PWA systems are represented by the following fields of the system structure:

```
sysStruct.A = {A1, A2, ..., An}
sysStruct.B = {B1, B2, ..., Bn}
sysStruct.f = {f1, f2, ..., fn}
sysStruct.C = {C1, C2, ..., Dn}
sysStruct.D = {D1, D2, ..., Cn}
sysStruct.g = {g1, g2, ..., gn}
sysStruct.A = {A1, A2, ..., An}
sysStruct.guardX = {Gx1, Gx2, ..., Gxn}
sysStruct.guardU = {Gu1, Gu2, ..., Gun}
sysStruct.guardC = {Gc1, Gc2, ..., Gcn}
```

In PWA case, each field of the structure has to be a cell array of matrices of appropriate dimensions. Each index $i \in 1, 2, \ldots, n$ corresponds to one PWA dynamics, i.e. to one tuple $[A_i, B_i, f_i, C_i, D_i, g_i]$ and one set of constraints $G_i^x x(k) + G_i^u u(k) \leq G_i^c$

Unlike the LTI case, you can omit `sysStruct.f` and `sysStruct.g` if they are zero. All other matrices have to be defined in the structure.

We will illustrate modelling of PWA systems on the following example:

Example 5.1.2: Assume a frictionless car moving horizontally on a hill with different slopes, as illustrated in Figure 5.1.

Dynamics of the car is driven by Newton's laws of motion:

$$\frac{\mathrm{d}p}{\mathrm{d}t} = v \tag{5.9}$$

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = u - mg\sin\alpha \tag{5.10}$$

where p denotes horizontal position and v stands for velocity of the object. If we now define $x = y = [p\ v]^T$, assume the mass $m = 1$ and discretize the system with sampling time of 0.1 seconds, we obtain the following affine system:

$$x(k+1) = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0.005 \\ 0.1 \end{bmatrix} u(k) + \begin{bmatrix} c \\ -g\sin\alpha \end{bmatrix} \tag{5.11}$$

$$y(k) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.12}$$

It can be seen that speed of the car depends only on the force applied to the car (manipulated variable u) and slope of the road α. Slope is different in different sectors of the road. In particular we have:

$$\begin{array}{lccl} \text{Sector 1:} & p \geq -0.5 & \Rightarrow \alpha = 0^o \\ \text{Sector 2:} & -3 \leq p \leq -0.5 & \Rightarrow \alpha = 10^o \\ \text{Sector 3:} & -4 \leq p \leq -3 & \Rightarrow \alpha = 0^o \\ \text{Sector 4:} & p \leq -4 & \Rightarrow \alpha = -5^o \end{array} \tag{5.13}$$

Substituting slopes α from (5.13) to (5.11), we obtain 4 tuples $[A_i, B_i, f_i, C_i, D_i, g_i]$ for $i \in 1, \ldots, 4$. Furthermore we need to define parts of the state-input space where each dynamics is active. We do that using the guard-lines $G_i^x x(k) + G_i^u u(k) \leq G_i^c$. With this formulation we can describes each sector as follows:

$$\begin{array}{ll} \text{Sector 1:} & \begin{bmatrix} -1 & 0 \end{bmatrix} x(k) \leq 0.5 \\ \text{Sector 2:} & \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} x(k) \leq \begin{bmatrix} -0.5 \\ 3 \end{bmatrix} \\ \text{Sector 3:} & \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} x(k) \leq \begin{bmatrix} -3 \\ 4 \end{bmatrix} \\ \text{Sector 4:} & \begin{bmatrix} 1 & 0 \end{bmatrix} x(k) \leq -4 \end{array} \tag{5.14}$$

Note that the state vector x consists of two components (position and velocity) and our sectors do not depend on value of the manipulated variable u, hence G^u is zero in our case and can be omitted from the definition. Once different dynamics and the corresponding guard-lines are defined, they must be linked together to tell $\mathbb{MPT}$ which dynamics is active in which sector. To do so, one needs to fill out the system structure in a prescribed way, i.e. by putting dynamics i and guard-lines i at the same position in the corresponding cell array. If you, for instance, put guard-lines defining sector 1 at first position in the cell array `sysStruct.guardX`, you link this sector with the proper dynamics by putting A_1, B_1, f_1, C_1, D_1 also on the first position in the corresponding fields. The whole system structure will then look as follows:

- Sector 1 - Guard-lines and dynamics:

```
sysStruct.guardX{1} = [-1 0]
sysStruct.guardC{1} = 0.5
sysStruct.A{1} = [1 0.1; 0 1]
```

```
sysStruct.B{1} = [0.005; 0.1]
sysStruct.f{1} = [c; -g*sin(alpha1)]
sysStruct.C{1} = [1 0;0 1]
sysStruct.D{1} = [0; 0]
```

- Sector 2 - Guard-lines and dynamics:

```
sysStruct.guardX{2} = [1 0; -1 0]
sysStruct.guardC{2} = [-0.5; 3]
sysStruct.A{2} = [1 0.1; 0 1]
sysStruct.B{2} = [0.005; 0.1]
sysStruct.f{2} = [c; -g*sin(alpha2)]
sysStruct.C{2} = [1 0;0 1]
sysStruct.D{2} = [0; 0]
```

- Sector 3 - Guard-lines and dynamics:

```
sysStruct.guardX{3} = [1 0; -1 0]
sysStruct.guardC{3} = [-3; 4]
sysStruct.A{3} = [1 0.1; 0 1]
sysStruct.B{3} = [0.005; 0.1]
sysStruct.f{3} = [c; -g*sin(alpha3)]
sysStruct.C{3} = [1 0;0 1]
sysStruct.D{3} = [0; 0]
```

- Sector 4 - Guard-lines and dynamics:

```
sysStruct.guardX{4} = [1 0]
sysStruct.guardC{4} = -4
sysStruct.A{4} = [1 0.1; 0 1]
sysStruct.B{4} = [0.005; 0.1]
sysStruct.f{4} = [c; -g*sin(alpha4)]
sysStruct.C{4} = [1 0;0 1]
sysStruct.D{4} = [0; 0]
```

Note that since g_i is always zero in (5.12), you can omit it from the system definition (the same holds for G_i^u if it is always zero).

We now consider a slight extension of Example 5.1.2 and show how to define a PWA system which also depends on values of the manipulated variable(s) u.

Example 5.1.3: Assume the Car on a PWA hill system as depicted in Figure 5.1. In addition to the original setup we assume different behavior of the car when applying positive and negative control action. In particular we assume that translation of the force u on the car is limited by half when u is negative. We can then consider two cases:

1. $u \geq 0$:

$$\frac{\mathrm{d}p}{\mathrm{d}t} = v \tag{5.15}$$

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = u - mg\sin\alpha \tag{5.16}$$

2. $u \leq 0$:

$$\frac{\mathrm{d}p}{\mathrm{d}t} = v \tag{5.17}$$

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{1}{2}u - mg\sin\alpha \tag{5.18}$$

With $m = 1$ and $x = [p\ v]^T$, discretization with sampling time of 0.1 seconds leads the following state-space representation:

1. $u \geq 0$:

$$x(k+1) = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0.005 \\ 0.1 \end{bmatrix} u(k) + \begin{bmatrix} c \\ -g\sin\alpha \end{bmatrix} \tag{5.19}$$

$$= Ax(k) + B_1u(k) + f_i \tag{5.20}$$

$$y(k) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.21}$$

$$= Cx(k) + Du(k) + g \tag{5.22}$$

2. $u \leq 0$:

$$x(k+1) = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0.0025 \\ 0.05 \end{bmatrix} u(k) + \begin{bmatrix} c \\ -g\sin\alpha \end{bmatrix} \tag{5.23}$$

$$= Ax(k) + B_2u(k) + f_i \tag{5.24}$$

$$y(k) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u(k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.25}$$

$$= Cx(k) + Du(k) + g \tag{5.26}$$

Value of the slope α again depends on horizontal position of the car according to sector conditions (5.13). Model of such a system now consists of 8 PWA dynamics (4 for positive u, 4 for negative u) which are defined over 8 sectors of the state-input space. Note that matrices A, C, D and g in (5.19)–(5.25) are constant and do not depend on the slope α nor on value of the control input u. With f_i we abbreviate matrices we obtain by substituting α from (5.13) into the equations of motion. B_1 and B_2 take different values depending on the orientation of the manipulated variable u. We can now define 8 segments of the state-input space and link dynamics to these sectors. We define these sectors using guard lines on states and inputs as follows:

- Sectors for $u \geq 0$

$$\begin{array}{lcl} \text{Sector 1:} & p \geq -0.5 & \Rightarrow \alpha = 0^o \\ \text{Sector 2:} & -3 \leq p \leq -0.5 & \Rightarrow \alpha = 10^o \\ \text{Sector 3:} & -4 \leq p \leq -3 & \Rightarrow \alpha = 0^o \\ \text{Sector 4:} & p \leq -4 & \Rightarrow \alpha = -5^o \end{array} \tag{5.27}$$

- Sectors for $u \leq 0$

$$\begin{array}{lcl} \text{Sector 5:} & p \geq -0.5 & \Rightarrow \alpha = 0^o \\ \text{Sector 6:} & -3 \leq p \leq -0.5 & \Rightarrow \alpha = 10^o \\ \text{Sector 7:} & -4 \leq p \leq -3 & \Rightarrow \alpha = 0^o \\ \text{Sector 8:} & p \leq -4 & \Rightarrow \alpha = -5^o \end{array} \tag{5.28}$$

which we can translate into guard-line setup $G_i^x x(k) + G_i^u u(k) \leq G_i^c$ as follows:

- Sectors for $u \geq 0$

$$
\begin{aligned}
\text{Sector 1:} \quad & \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ -1 \end{bmatrix} u(k) \leq \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \\
\text{Sector 2:} \quad & \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} u(k) \leq \begin{bmatrix} -0.5 \\ 3 \\ 0 \end{bmatrix} \\
\text{Sector 3:} \quad & \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} u(k) \leq \begin{bmatrix} -3 \\ 4 \\ 0 \end{bmatrix} \\
\text{Sector 4:} \quad & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ -1 \end{bmatrix} u(k) \leq \begin{bmatrix} -4 \\ 0 \end{bmatrix}
\end{aligned}
\tag{5.29}
$$

- Sectors for $u \leq 0$

$$
\begin{aligned}
\text{Sector 5:} \quad & \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(k) \leq \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \\
\text{Sector 6:} \quad & \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u(k) \leq \begin{bmatrix} -0.5 \\ 3 \\ 0 \end{bmatrix} \\
\text{Sector 7:} \quad & \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u(k) \leq \begin{bmatrix} -3 \\ 4 \\ 0 \end{bmatrix} \\
\text{Sector 8:} \quad & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(k) \leq \begin{bmatrix} -4 \\ 0 \end{bmatrix}
\end{aligned}
\tag{5.30}
$$

Now we can define the system by filling out the system structure:

- Sector 1 - Guard-lines and dynamics:

```
sysStruct.guardX{1} = [-1 0; 0 0]
sysStruct.guardU{1} = [0; -1]
sysStruct.guardC{1} = [0.5; 0]
sysStruct.A{1} = A
sysStruct.B{1} = B_1
sysStruct.f{1} = f_1
sysStruct.C{1} = C
sysStruct.D{1} = D
```

- Sector 2 - Guard-lines and dynamics:

```
sysStruct.guardX{2} = [1 0; -1 0; 0 0]
sysStruct.guardU{2} = [0; 0; -1]
sysStruct.guardC{2} = [-0.5; 3; 0]
sysStruct.A{2} = A
sysStruct.B{2} = B_1
sysStruct.f{2} = f_2
sysStruct.C{2} = C
```

```
sysStruct.D{2} = D
⋮
```

- Sector 8 - Guard-lines and dynamics:

```
sysStruct.guardX{8} = [1 0; 0 0]
sysStruct.guardU{8} = [0; 1]
sysStruct.guardC{8} = [-4; 0]
sysStruct.A{8} = A
sysStruct.B{8} = B_2
sysStruct.f{8} = f_4
sysStruct.C{8} = C
sysStruct.D{8} = D
```

5.2 Import of models from various sources

$\mathbb{MPT}$ can design control laws for discrete-time constrained linear, switched linear and hybrid systems. Hybrid systems can be described in Piecewise-Affine (PWA) or Mixed Logical Dynamical (MLD) representations and an efficient algorithm is provided to switch from one representation to the other form and vice-versa. To increase user's comfort, models of dynamical systems can imported from various sources:

- Models of hybrid systems designed in the HYSDEL [29] language,
- MLD structures generated by `mpt_pwa2mld`
- Nonlinear models defined by `mpt_nonlinfcn` template
- State-space and transfer function objects of the Control toolbox,
- System identification toolbox objects,
- MPC toolbox objects.

In order to import a dynamical system, one has to call

`model=mpt_sys(object, Ts)`

where `object` can be either a string (in which case the model is imported from a corresponding HYSDEL source file), or it can be a variable of one of the above mentioned object types. The second input parameter `Ts` denotes sampling time and can be omitted, in which case `Ts = 1` is assumed.

Example 5.2.1: The following code will first define a continuous-time state-space object which is then imported to $\mathbb{MPT}$:

```
% sampling time
Ts = 1;

% continuous-time model as state-space object
```

```
di = ss([1 1; 0 1], [1; 0.5], [1 0; 0 1], [0; 0]);

% import the model and discretize it
sysStruct = mpt_sys(di, Ts);
```

Note: If the state-space object is already in discrete-time domain, it is not necessary to provide the sampling time parameter `Ts` to `mpt_sys`. After importing a model using `mpt_sys` it is still necessary to define system constraints as described previously.

5.3 Modelling using HYSDEL

Models of hybrid systems can be imported from HYSDEL source (see HYSDEL documentation for more details on HYSDEL modelling), e.g.

```
sysStruct = mpt_sys('hysdelfile.hys', Ts);
```

Note: Hybrid systems modeled in HYSDEL are already defined in the discrete-time domain, the additional sampling time parameter `Ts` is only used to set the sampling interval for simulations. If `Ts` is not provided, it is set to 1.

Model of a hybrid system defined in `hysdelfile.hys` is first transformed into an Mixed Logical Dynamical (MLD) form using the HYSDEL compiler and then an equivalent PWA representation is created using $\mathbb{MPT}$. It is possible to avoid the PWA transformation by calling

```
sysStruct = mpt_sys('hysdelfile.hys', Ts, 'mld');
```

in which case only an MLD representation is created. Note, however, that systems only in MLD form can be controlled only with the on-line MPC schemes.

After calling `mpt_sys` it is still necessary to define system constraints as described in the next section.

5.4 System constraints

$\mathbb{MPT}$ allows to define following types of constraints:

- Min/Max constraints on system outputs
- Min/Max constraints on system states
- Min/Max constraints on manipulated variables
- Min/Max constraints on slew rate of manipulated variables
- Polytopic constraints on states

Constraints on system outputs

Output equation is in general driven by the following relation for PWA systems

$$y(k) = C_i x(k) + D_i u(k) + g_i \tag{5.31}$$

and by

$$y(k) = Cx(k) + Du(k) \tag{5.32}$$

for LTI systems. It is therefore clear that by choice of $C = I$ one can use these constraints to restrict system states as well. Min/Max output constraints have to be given in the following fields of the system structure:

```
sysStruct.ymax = outmax
sysStruct.ymin = outmin
```

where `outmax` and `outmin` are $n_y \times 1$ vectors.

Constraints on system states

Constraints on system states are optional and can be defined by

```
sysStruct.xmax = xmax
sysStruct.xmin = xmin
```

where `xmax` and `xmin` are $n_x \times 1$ vectors.

Constraints on manipulated variables

Goal of each control technique is to design a controller which chooses a proper value of the manipulated variable in order to achieve the given goal (usually to guarantee stability, but other aspects like optimality may also be considered at this point). In most real plants values of manipulated variables are restricted and these constraints have to be taken into account in controller design procedure. These limitations are usually saturation constraints and can be captured by min / max bounds. In $\mathbb{MPT}$, constraints on control input are given in:

```
sysStruct.umax = inpmax
sysStruct.umin = inpmin
```

where `inpmax` and `inpmin` are $n_u \times 1$ vectors.

Constraints on slew rate of manipulated variables

Another important type of constraints are rate constraints. These limitations restrict the variation of two consecutive control inputs ($\delta u = u(k) - u(k-1)$) to be within of prescribed bounds. One can use slew rate constraints when a "smooth" control action is required, e.g. when controlling a gas pedal in a car to prevent the car from jumping due to sudden changes of the controller action. Min/max bounds on slew rate can be given in:

```
sysStruct.dumax = slewmax
sysStruct.dumin = slewmin
```

where `slewmax` and `slewmin` are $n_u \times 1$ vectors.
Note: This is an optional argument and does not have to be defined. If it is not given, bounds are assumed to be $\pm\infty$.

Polytopic constraints on states

MPT also supports one additional constraint, the so-called `Pbnd` constraint. If you define `sysStruct.Pbnd` as a polytope object of the dimension of your state vector, this entry will be used as a polytopic constraint on the initial condition, i.e.

$$x_0 \in \texttt{sysStruct.Pbnd}$$

This is especially important for explicit controllers, since `sysStruct.Pbnd` there limits the state-space which will be explored. If `sysStruct.Pbnd` is not specified, it will be set as a ”large” box of size defined by `mptOptions.infbox` (see `help mpt_init` for details). **Note:** `sysStruct.Pbnd` does NOT impose any constraints on predicted states!

If you want to enforce polytopic constraints on predicted states, inputs and outputs, you need to add them manually using the ”Design your own MPC” function described in Section 6.4.

5.5 Systems with discrete valued inputs

$\mathbb{MPT}$ allows to define system with discrete-valued control inputs. This is especially important in framework of hybrid systems where control inputs are often required to belong to certain set of values. We distinguish between two cases:

1. All inputs are discrete
2. Some inputs are discrete, the rest are continuous

Purely discrete inputs

Typical application of discrete-valued inputs are various on/off switches, gears, selectors, etc. All these can be modelled in $\mathbb{MPT}$ and taken into account in controller design. Defining discrete inputs is fairly easy, all you need to do is to fill out

```
sysStruct.Uset = Uset
```

where `Uset` is a cell array which defines all possible values for every control input. If your system has, for instance, 2 control inputs and the first one is just an on/off switch (i.e. $u_1 = \{0, 1\}$) and the second one can take values from set $\{-5, 0, 5\}$, you define it as follows:

```
sysStruct.Uset{1} = [0, 1]
sysStruct.Uset{2} = [-5, 0, 5]
```

where the first line corresponds to u_1 and the second to u_2. If your system has only one manipulated variable, the cell operator can be omitted, i.e. one could write:

```
sysStruct.Uset = [0, 1]
```

The set of inputs doesn't have to be ordered.

Example 5.5.1: Consider a double integrator sampled at 1 second:

$$x(k+1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} u(k) \tag{5.33}$$

$$y(k) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(k) \tag{5.34}$$

and assume that the the manipulated variable can take only values from the set $\{-1, 01\}$. The corresponding MPT model would look like this:

```
sysStruct.A = [1 1; 0 1]
sysStruct.B = [1; 0.5]
sysStruct.C = [1 0]
sysStruct.D = 0
sysStruct.Uset = [-1 0 1]
sysStruct.ymin = -10
sysStruct.ymax = 10
sysStruct.umax = 1
sysStruct.umin = -1
```

Notice that constraints on control inputs `umax, umin` have to be provided even when manipulated variable is declared to be discrete.

Example 5.5.2: We consider system defined in Example 5.5.1. In addition we assume that when u is 1, dynamics of the system is driven by equation (5.33), otherwise the state-update equation takes the following xform:

$$x(k+1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 2 \\ 1 \end{bmatrix} u(k), \tag{5.35}$$

i.e. the system matrix B is amplified by a factor of 2.

Such a behavior can be efficiently captured by a PWA model, which will consist of two modes separated by a guard-line defined on the manipulated variable. The first dynamics (defined by (5.33)) will be active whenever $u \leq 0$ and dynamics 2 will be enforced once $u \geq 1$. The corresponding MPT model is then:

```
sysStruct.A = { [1 1; 0 1], [1 1; 0 1] }
sysStruct.B = { [1; 0.5], [2; 1] }
```

```
sysStruct.C = { [1 0], [1 0] }
sysStruct.D = { 0, 0}
sysStruct.guardX = { [0 0], [0 0] }
sysStruct.guardU = { 1, -1 }
sysStruct.guardC = { 0, -1 }
sysStruct.Uset = [-1 0 1]
```

with same constraints on outputs and manipulated variables as in example 5.5.1.

Mixed inputs

Mixed discrete and continuous inputs can be modelled by appropriate choice of `sysStruct.Uset`. For each continuous input it is necessary to set the corresponding entry to `[-Inf Inf]`, indicating to MPT that this particular input variable should be treated as a continuous input. For a system with two manipulated variables, where the first one takes values from a set $\{-2.5, 0, 3.5\}$ and the second one is continuous, one would set:

```
sysStruct.Uset{1} = [-2.5, 0, 3.5]
sysStruct.Uset{2} = [-Inf Inf]
```

5.6 Text labels

State, input and output variables can be assigned a text label which overrides the default axis labels in trajectory and partition plotting (x_i, u_i and y_i, respectively). To assign a text label, set the following fields of the system structure, e.g. as follows:

```
sysStruct.xlabels = {'position', 'speed'};
sysStruct.ulabels = 'force';
sysStruct.ylabels = {'position', 'speed'};
```

which corresponds to the `Double_Integrator` example. Each field is an array of strings corresponding to a given variable. If the user does not define any (or some) labels, they will be replaced by default strings (x_i, u_i and y_i). The strings are used once polyhedral partition of the explicit controller, or closed-loop (open-loop) trajectories are visualized.

5.7 System Structure `sysStruct`

System structure `sysStruct` is a structure which describes the system to be controlled. MPT can deal with two types of systems:

1. Discrete-time linear time-invariant (LTI) systems
2. Discrete-time Piecewise Affine (PWA) Systems

Both system types can be subject to constraints imposed on control inputs and system outputs. In addition, constraints on slew rate of the control inputs can also be given.

LTI systems

In general, a constrained linear time-invariant system is defined by the following relations:

$$
\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) \\
y(k) &= Cx(k) + Du(k) \\
\text{subt. to} & \\
y_{min} &\leq y(k) \leq y_{max} \\
u_{min} &\leq u(k) \leq u_{max}
\end{aligned}
$$

Such an LTI system is defined by the following mandatory fields:

```
sysStruct.A = A;
sysStruct.B = B;
sysStruct.C = C;
sysStruct.D = D;
sysStruct.ymax = ymax;
sysStruct.ymin = ymin;
sysStruct.umax = umax;
sysStruct.umin = umin;
```

Constraints on slew rate of the control input $u(k)$ can also be imposed by:

```
sysStruct.dumax = dumax;
sysStruct.dumin = dumin;
```

which enforces $\Delta u_{min} <= u(k) - u(k-1) <= \Delta u_{max}$.

Note: If no constraints are present on certain inputs/states, set the associated values to `Inf`.

LTI system which is subject to parametric uncertainty and/or additive disturbances is driven by the following set of relations:

$$
\begin{aligned}
x(k+1) &= A_{unc}x(k) + B_{unc}u(k) + w(k) \\
y(k) &= Cx(k) + Du(k)
\end{aligned}
$$

where $w(k)$ is an unknown, but bounded additive disturbance, i.e.

$$
w(n) \in W \qquad \forall n \in (0...Inf)
$$

To specify an additive disturbance, set

```
sysStruct.noise = W
```

where `W` is a polytope object bounding the disturbance. $\mathbb{MPT}$ also supports lower-dimensional noise polytopes. If you want to define noise only on a subset of system states, you can now do so by defining sysStruct.noise as a set of vertices representing the noise. Say you want to impose a `+/- 0.1` noise on `x1`, but no noise should be used for `x2`. You can do that by:

```
sysStruct.noise = [-0.1 0.1; 0 0];
```

Just keep in mind that the noise polytope must have vertices stored *c*olumn-wise.

A polytopic uncertainty can be specified by a cell array of matrices `Aunc` and `Bunc` as follows:

```
sysStruct.Aunc = {A1, ..., An};
sysStruct.Bunc = {B1, ..., Bn};
```

PWA Systems

PWA systems are models for describing hybrid systems. Dynamical behavior of such systems is captured by relations of the following form:

$$\begin{aligned} x(k+1) &= A_i x(k) + B_i u(k) + f_i \\ y(k) &= C_i x(k) + D_i u(k) + g_i \\ \text{subj. to} & \\ y_{min} &\leq y(k) \leq y_{max} \\ u_{min} &\leq u(k) \leq u_{max} \\ \Delta u_{min} &\leq u(k) - u(k-1) \leq \Delta u_{max} \end{aligned}$$

Each dynamics i is active in a polyhedral partition bounded by the so-called guard-lines:

$$guardX_i x(k) + guardU_i u(k) <= guardC_i$$

which means dynamics i will be applied if the above inequality is satisfied.

Fields of `sysStruct` describing a PWA system are listed below:

```
sysStruct.A = {A1, ..., An}
sysStruct.B = {B1, ..., Bn}
sysStruct.C = {C1, ..., Cn}
```

```
sysStruct.D = {D1, ..., Dn}
sysStruct.f = {f1, ..., fn}
sysStruct.g = {g1, ..., gn}
sysStruct.guardX = {guardX1, ..., guardXn}
sysStruct.guardU = {guardU1, ..., guardUn}
sysStruct.guardC = {guardC1, ..., guardCn}
```

Note that all fields have to be cell arrays of matrices of compatible dimensions, n stands for total number of different dynamics. If `sysStruct.guardU` is not provided, it is assumed to be zero.

System constraints are defined by:

```
sysStruct.ymax  = ymax;
sysStruct.ymin  = ymax;
sysStruct.umax  = umax;
sysStruct.umin  = umin;
sysStruct.dumax = dumax;
sysStruct.dumin = dumin;
```

Constraints on slew rate are optional and can be omitted.

$\mathbb{MPT}$ is able to deal also with PWA systems which are affected by bounded additive disturbances:

$$x(k+1) = A_i x(k) + B_i u(k) + f_i + w(k)$$

where the disturbance $w(k)$ is assumed to be bounded for all time instances by some polytope W. To indicate that your system is subject to such a disturbance, set

```
sysStruct.noise = W;
```

where `W` is a polytope object of appropriate dimension.

Mandatory and optional fields of the system structure are summarized in Tables 5.7 and 5.7, respectively.

`A, B, C, D, f, g`	State-space dynamic martices in (3.10) and (3.20a). Set elements to empty if they do not apply.
`umin, umax`	Bounds on inputs `umin` $\leq$ u(t) $\leq$ `umax`.
`ymin, ymax`	Constraints on the outputs `ymin` $\leq$ y(t) $\leq$ `ymax`.
`guardX, guardU, guardC`	Polytope cell array defining where the dynamics are active (for PWA systems). $\mathcal{D}_i = \{(x,u) \mid guardXi\ x + guardUi\ u \leq gu/ardCi\}$.

Tab. 5.2: Mandatory fields of the system structure `sysStruct`.

`Uset`	Declares discrete-valued inputs
`dumin, dumax`	Bounds on `dumin` $\leq$ u(t)-u(t-1) $\leq$ `dumax`.
`noise`	A `polytope` bounding the additive disturbance, i.e. `noise`=$\mathcal{W}$ in (3.16).
`Aunct, Bunct`	Cell arrays containing the vertices of the polytopic uncertainty (3.18).
`Pbnd`	Polytope limiting the feasible state-space of intersest.

Tab. 5.3: Optional fields of the system structure `sysStruct`.

6

Control Design

6.1 Controller computation

For constrained linear and hybrid systems, MPT can design optimal and sub-optimal control laws either in implicit form, where an optimization problem of finite size is solved on-line at every time step and is used in a Receding Horizon Control (RHC) manner or, alternatively, solve an optimal control problem in a multi-parametric fashion. If the latter approach is used, an explicit representation of the control law is obtained.

The solution to an optimal control problem can be obtained by a simple call of `mpt_control`. The general syntax to obtain an explicit representation of the control law is:

```
ctrl = mpt_control(sysStruct, probStruct)
```

On-line MPC controllers can be generated by

```
ctrl = mpt_control(sysStruct, probStruct, 'online')
```

Based on the system definition described by `sysStruct` (cf. Section 5.7) and problem description provided in `probStruct` (cf. Section 6.9), the main control routine automatically calls one of the functions reported in Table 6.1 to calculate the explicit solution to a given problem. `mpt_control` first verifies if all mandatory fields in `sysStruct` and `probStruct` structures are filled out, if not, the procedure will break with an appropriate error message. Note that the validation process sets the optional fields to default values if there are not present in the two respective structures. Again, an appropriate message is displayed.

Once the control law is calculated, the solution (here `ctrl`) is returned as an instance of the `mptctrl` object. Internal fields of this object, described in Section 6.2, can be accessed directly using the sub-referencing operator. For instance

```
Pn = ctrl.Pn;
```

will return the polyhedral partition of the explicit controller defined in the variable `ctrl`.

Control laws can further be analyzed and/or implemented by functions reported in Chapters 8 and 9.

System	N	Suboptimality	Problem	Function	Reference
LTI	fixed	0	CFTOC	`mpt_optControl`	[1, 9]
LTI	`Inf`	0	CITOC	`mpt_optInfControl`	[13]
LTI	`Inf`	1	CMTOC	`mpt_iterative`	[15, 16]
LTI	`Inf`	2	LowComp	`mpt_oneStepCtrl`	[15, 16]
PWA	fixed	0	CFTOC	`mpt_optControlPWA`	[10, 20, 9]
PWA	`Inf`	0	CITOC	`mpt_optInfControlPWA`	[2]
PWA	`Inf`	1	CMTOC	`mpt_iterativePWA`	[14]
PWA	`Inf`	2	LowComp	`mpt_iterativePWA`	[14]

Tab. 6.1: List of control strategies applied to different system and problem definitions.

MPT provides a variety of control routines which are being called from `mpt_control`. Solutions to the following problems can be obtained depending on the properties of the system model and the optimization problem. One of the following control problems can be solved:

A. Constrained Finite Time Optimal Control (CFTOC) Problem.
B. Constrained Infinite Time Optimal Control Problem (CITOC).
C. Constrained Minimum Time Optimal Control (CMTOC) Problem.
D. Low complexity setup.

The problem which will be solved depends on parameters of the system and problem structure, namely on the type of the system (LTI or PWA), prediction horizon (fixed or infinity) and the level of sub-optimality (optimal solution, minimum-time solution, low complexity). Different combinations of these three parameters lead to a different optimization procedure, as reported in Table 6.1.

See the documentation of the individual functions for more details. For a good overview of receding horizon control we refer the reader to [24, 22].

6.2 Fields of the `mptctrl` object

The controller object includes all results obtained as a solution of a given optimal control problem. In general, it describes the obtained control law and can be used both for analysis of the solution, as well as for an implementation of the control law.

Fields of the object are summarized in Table 6.2. Every field can be accessed using the standard `.` (dot) sub-referencing operator, e.g.

```
Pn = ctrl.Pn;
Fi = ctrl.Fi;
runtime = ctrl.details.runtime;
```

`Pn`	The polyhedral partition over which the control law is defined is returned in this field. It is, in general, a polytope array.
`Fi, Gi`	The PWA control law for a given state $x(k)$ is given by `u = Fi{r} x(k) + Gi{r}`. `Fi` and `Gi` are cell arrays.
`Ai, Bi, Ci`	Value function is returned in these three cell arrays and for a given state $x(k)$ can be evaluated as `J = x(k)' Ai{r} x(k) + Bi{r} x(k) + Ci{r}` where the prime denotes the transpose and r is the index of the active region, i.e. the region of `Pn` containing the given state $x(k)$.
`Pfinal`	In this field, the maximum (achieved) feasible set is returned. In general, it corresponds to the union of all polytopes in `Pn`.
`dynamics`	A vector which denotes which dynamics is active in which region of `Pn`. (Only important for PWA systems.)
`details`	More details about the solution, e.g. total run time.
`overlaps`	Boolean variable denoting whether regions of the controller partition overlap.
`sysStruct`	System description in the `sysStruct` format.
`probStruct`	Problem description in the `probStruct` format.

Tab. 6.2: Fields of MPT controller objects.

6.3 Functions defined for `mptctrl` objects

Once the explicit control law is obtained, the corresponding controller object is returned to the user. The following functions can then be applied:

`analyze`

Analyzes a given explicit controller and suggests which actions to take in order to improve the controller.

```
>> analyze(ctrl)

The controller can be simplified:
   ctrl = mpt_simplify(ctrl)  will reduce the number of regions.

The closed-loop system may not be invariant:
   ctrl = mpt_invariantSet(ctrl)  will identify the invariant subset.
```

`isexplicit`

Returns true if the controller is an explicit controller.

```
% compute an explicit controller
>> expc = mpt_control(sysStruct, probStruct);
```

```
% compute an on-line controller
>> onlc = mpt_control(sysStruct, probStruct, 'online');

>> isexplicit(expc)

ans =

     1

>> isexplicit(onlc)

ans =

     0
```

isinvariant

Returns true if a given explicit controller is invariant.

```
>> isinvariant(ctrl)

ans =

     1
```

isstabilizable

Returns true if a given controller guarantees closed-loop stability.

```
>> isstabilizable(ctrl)

ans =

     1
```

length

Returns the number of regions over which the explicit control law is defined.

```
>> length(ctrl)

ans =

    25
```

plot

Plots regions of the explicit controller.

```
>> plot(ctrl)
```

runtime

Returns runtime needed to compute a given explicit controller.

```
>> runtime(ctrl)

ans =

    0.5910
```

sim

Compute trajectories of the closed-loop system.

```
>> [X, U, Y] = sim(ctrl, x0, number_of_steps)
```

The `sim` command computes trajectories of the closed-loop system subject to a given controller. For a more detailed description, please see `help mptctrl/sim`.

simplot

Plot trajectories of the closed-loop system.

```
>> simplot(ctrl, x0, number_of_steps)
```

The `simplot` plots closed-loop trajectories of a given system subject to given control law. For a more detailed description, please see `help mptctrl/simplot`.

Evaluation of controllers as functions

In order to obtain a control action associated to a given initial state x_0, it is possible to evaluate the controller object as follows:

```
>> u = ctrl(x0)

ans =

    -0.7801
```

6.4 Design of custom MPC problems

This is the coolest feature in the whole history of MPT ! And the credits go Johan Löfberg and his YALMIP [21]. The function `mpt_ownmpc` allows you to add (almost) arbitrary constraints to an MPC setup and to define a custom objective functions.

First we explain the general usage of the new function. Design of the custom MPC controllers is divided into three parts:

1. *Design phase*. In this part, general constraints and a corresponding cost function are designed
2. *Modification phase*. In this part, the user is allowed to add custom constraints and/or to modify the cost function
3. *Computation phase*. In this part, either an explicit or an on-line controller which respects user constraints is computed.

Design phase

Aim of this step is to obtain constraints which define a given MPC setup, along with an associated cost function, and variables which represent system states, inputs and outputs at various prediction steps. In order to obtain said elements for the case of explicit MPC controllers, call:

```
>> [CON, OBJ, VARS] = mpt_ownmpc(sysStruct, probStruct)
```

or, for on-line MPC controllers, call:

```
>> [CON, OBJ, VARS] = mpt_ownmpc(sysStruct, probStruct, 'online')
```

Here the variable `CON` represents a set of constraints, `OBJ` denotes the optimization objective and `VARS` is a structure with the fields `VARS.x` (predicted states), `VARS.u` (predicted inputs) and `VARS.y` (predicted outputs). Each element is given as a cell array, where each element corresponds to one step of the prediction (i.e. `VARS.x1` denotes the initial state `x0`, `VARS.x2` is the first predicted state `x1`, etc.) If a particular variable is a vector, it can be indexed directly to refer to a particular element, e.g. `VARS.x3(1)` refers to the first element of the 2nd predicted state (i.e. `x2`).

Modification phase

Now you can start modifying the MPC setup by adding your own constraints and/or by modifying the objective. See examples below for more information about this topic.

Note: You should always add constraints on system states (`sysStruct.xmin`, `sysStruct.xmax`), inputs (`sysStruct.umin`, `sysStruct.umax`) and outputs (`sysStruct.ymin`, `sysStruct.ymax`) if you either design a controller for PWA/MLD systems, or you intend to add logic constraints later. Not adding the constraints will cause your problem to be very badly scaled, which can lead to very bad solutions.

Computation phase

Once you have modified the constraints and/or the objective according to your needs, you can compute an explicit controller by

```
>> ctrl = mpt_ownmpc(sysStruct, probStruct, CON, OBJ, VARS)
```

or an on-line MPC controller by

```
>> ctrl = mpt_ownmpc(sysStruct, probStruct, CON, OBJ, VARS, 'online')
```

Example 6.4.1 (Polytopic constraints 1)**:** Say we would like to introduce polytopic constraints of the form $Hx_k \leq K$ on each predicted state, including the initial state x_0. To do that, we simply add these constraints to our set CON:

```
for k = 1:length(VARS.x)
  CON = CON + set(H * VARS.x{k} <= K);
end
```

You can now proceed with the computation phase, which will give you a controller which respects given constraints.

Example 6.4.2 (Polytopic constraints 2)**:** We now extend the previous example and add the specification that polytopic constraints should only be applied on the 1st, 3rd and 4th predicted state, i.e. on x_1, x_3 and x_4. It is important to notice that the variables contained in the VARS structure are organized in cell arrays, where the first element of VARS.x corresponds to x_0, i.e. to the initial condition. Therefore to meet or specifications, we would write following code:

```
for k = [1 3 4],
  % VARS.x{1} corresponds to x(0)
  % VARS.x{2} corresponds to x(1)
  % VARS.x{3} corresponds to x(2)
  % VARS.x{4} corresponds to x(3)
  % VARS.x{5} corresponds to x(4)
  % VARS.x{6} corresponds to x(5)
  CON = CON + set(H * VARS.x{k+1} <= K);
end
```

Example 6.4.3 (Move blocking)**:** Say we want to use more complicated move blocking with following properties: $u_0 = u_1$, $(u_1 - u_2) = (u_2 - u_3)$, and $u_3 = Kx_2$. These requirements can be implemented by

```
% VARS.u{1} corresponds to u(0)
% VARS.u{2} corresponds to u(1), and so on

% u_0 == u_1
>> CON = CON + set(VARS.u{1} == VARS.u{2});
```

```
% (u_1-u_2) == (u_2-u_3)
>> CON = CON + set((VARS.u{2}-VARS.u{3}) == (VARS.u{3}-VARS.u{4}));

% u_3 == K*x_2
>> CON = CON + set(VARS.u{4} == K * VARS.x{3});
```

Example 6.4.4 (Mixed constraints)**:** As illustrated in the move blocking example above, one can easily create constraints which involve variables at various stages of the prediction. In addition, it is also possible to add constraints which involve different types of variables. For instance, we may want to add a constraint that the sum of control inputs and system outputs at each step must be between certain bounds. This specification can be expressed by:

```
for k = 1:length(VARS.u)
  CON = CON + set(lowerbound < VARS.y{k} + VARS.u{k} < upperbound);
end
```

Example 6.4.5 (Equality constraints)**:** Say we want to add a constraint that the sum of all predicted control actions along the prediction horizon should be equal to zero. This can easily be done by

```
>> CON = CON + set((VARS.u{1}+VARS.u{2}+VARS.u{3}+...+VARS.u{end}) == 0);
```

or, in a vector notation:

```
>> CON = CON + set(sum([VARS.u{:}]) == 0);
```

Example 6.4.6 (Constraints involving norms)**:** We can extend the previous example and add a specification that the sum of absolute values of all predicted control actions should be less than some given bound. To achieve this goal, we can make use of the 1-norm function, which exactly represents the sum of absolute values of each element:

```
>> CON = CON + set(norm([V.u{:}], 1) <= bound);
```

The same can of course be expressed in a more natural form:

```
>> CON = CON + set(sum(abs([V.u{:}])) <= bound);
```

Example 6.4.7 (Contraction constraints)**:** The norm-type constraints can be used to define ”contraction” constraints, i.e. constraints which force state x_{k+1} to be closer to the origin (in the 1/Inf-norm sense) than the state x_k has been:

```
for k = 1:length(VARS.x)-1
  CON = CON + set(norm(VARS.x{k+1}, 1) <= norm(VARS.x{k}, 1));
end
```

Note that these types of constraints are not convex and resulting problems will be difficult to solve (time-wise).

Example 6.4.8 (Obstacle avoidance)**:** It is a typical requirement of control synthesis to guarantee that the system states will avoid some set of ”unsafe” states (typically an obstacle or a set of dangerous operating conditions). You can now solve these kind of problems with MPT if you add suitable constraints. If you, for instance, want to avoid a polytopic set of states, proceed as follows:

```
% first define set of unsafe states
>> Punsafe = polytope(H, K);

% now define the complement of the "usafe" set versus some large box,
% to obtain the set of states which are "safe":
>> Pbox = unitbox(dimension(Punsafe), 100);
>> Psafe =  Pbox \ Punsafe;

% now add constraints that each predicted state must be inside
% of the "safe" set of states
for k = 1:length(VARS.x)
  CON = CON + set(ismember(VARS.x{k}, Psafe));
end
```

Here `set(ismember(VARS.xk, Psafe))` will impose a constraint which tells MPT that it must guarantee that the state x_k belongs to at least one polytope of the polytope array `Psafe`, and hence avoiding the ”unsafe” set `Punsafe`. Notice that this type of constraints requires binary variables to be introduced, making the optimization problem difficult to solve.

Example 6.4.9 (Logic constraints)**:** Logic constraints in the form of IF-THEN conditions can be added as well. For example, we may want to require that if the first predicted input u_0 is smaller or equal to zero, then the next input u_1 has to be bigger than 0.5:

```
% if u(0) <= 0 then u(1) must be >= 0.5
>> CON = CON + set(implies(VARS.u{1} <= 0, VARS.u{2} >= 0.5));
```

Notice that this constraint only acts in one direction, i.e. if $u_0 \leq 0$ then $u_1 \geq 0.5$, but it does not say what should be the value of u_1 if $u_0 > 0$.

To add an ”if and only if” constraint, use the `iff()` operator:

```
% if u(0) <= 0 then u(1) >= 0.5, and
% if u(0) > 0 then u(1) < 0.5
>> CON = CON + set(iff(VARS.u{1} <= 0, VARS.u{2} >= 0.5));
```

which will guarantee that if $u_0 > 0$, then the value of u_1 will be smaller than 0.5.

Example 6.4.10 (Custom optimization objective)**:** In the last example we show how to define your own objective functions. Depending on the value of `probStruct.norm`, the objective can either be quadratic, or linear. By default, it is defined according to standard MPC theory (see `help mpt_probStruct` for details).

To write a custom cost function, simply sum up the terms you want to penalize. For instance, the standard quadratic cost function can be defined by hand as follows:

```
OBJ = 0;
for k = 1:length(VARS.u),
  % cost for each step is given by x'*Q*x + u'*R*u
  OBJ = OBJ + VARS.x{k}' * Q * VARS.x{k};
  OBJ = OBJ + VARS.u{k}' * R * VARS.u{k};
end

% cost for the last predicted state x_N'*P_N*x_N
OBJ = OBJ + VARS.x{end}' * P_N * VARS.x{end};
```

For 1/Inf-norm cost functions, you can use the overloaded `norm()` operator, e.g.

```
OBJ = 0;
for k = 1:length(VARS.u),
  % cost for each step is given by ||Q*x|| + ||R*u||
  OBJ = OBJ + norm(Q * VARS.x{k}, Inf);
  OBJ = OBJ + norm(R * VARS.u{k}, Inf;
end

% cost for the last predicted state ||P_N*x_N||
OBJ = OBJ + norm(P_N * VARS.x{end}, Inf);
```

If you, for example, want to penalize deviations of predicted outputs and inputs from a given time-varying trajectories, you can do so by defining a cost function as follows:

```
yref = [4 3 2 1];
uref = [0 0.5 0.1 -0.2]
OBJ = 0;
for k = 1:length(yref)
  OBJ = OBJ + (VARS.y{k} - yref(k))' * Qy * (VARS.y{k} - yref(k));
  OBJ = OBJ + (VARS.u{k} - uref(k))' * R * (VARS.u{k} - uref(k));
end
```

Example 6.4.11 (Defining new variables)**:** Remember the avoidance example? There we have used constraints to tell the controller that it should avoid a given set of unsafe states. Let's now modify that example a bit. Instead of adding constraints, we will introduce a binary variable which will take a true value if the system states are inside of a given location. Subsequently we will add a high penalty on that variable, which will tell the MPC controller that it should avoid the set if possible.

```
% first define the set which we want to avoid
>> Pavoid = polytope(H, K);

% define a new scalar binary variable
>> d = binvar(1, 1);
```

```
% now add a constraint which forces "d" to be true if x(0) is
% inside of Pavoid
>> CON = CON + set(implies(ismember(VARS.x{1}, Pavoid), d))

% penalize "d" heavily
>> OBJ = OBJ + 1000*d
```

Example 6.4.12 (Removing constraints)**:** When `mpt_ownmpc` constructs the constraints and objectives, it adds constraints on system states, inputs and outputs, providing they are defined in respective fields of `probStruct`. Though one may want to remove certain constraints, for instance the target set constraints imposed on the last predicted state. To do so, first notice that each constraint has an associated string tag:

```
>> Double_Integrator
>> sysStruct.xmax = sysStruct.ymax; sysStruct.xmin = sysStruct.ymin;
>> probStruct.N = 2;
>> [CON, OBJ, VARS] = mpt_ownmpc(sysStruct, probStruct);
>> CON
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   ID|      Constraint|                         Type|                 Tag|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   #1|   Numeric value|           Element-wise 2x1|      umin < u_1 < umax|
|   #2|   Numeric value|           Element-wise 4x1|      xmin < x_1 < xmax|
|   #3|   Numeric value|           Element-wise 4x1|      xmin < x_2 < xmax|
|   #4|   Numeric value|           Element-wise 4x1|      ymin < y_1 < ymax|
|   #5|   Numeric value|    Equality constraint 2x1|   x_2 == A*x_1 + B*u_1|
|   #6|   Numeric value|    Equality constraint 2x1|   y_1 == C*x_1 + D*u_1|
|   #7|   Numeric value|           Element-wise 6x1|            x_2 in Tset|
|   #8|   Numeric value|           Element-wise 4x1|            x_0 in Pbnd|
|   #9|   Numeric value|           Element-wise 2x1|      umin < u_0 < umax|
|  #10|   Numeric value|           Element-wise 4x1|      xmin < x_0 < xmax|
|  #11|   Numeric value|           Element-wise 4x1|      ymin < y_0 < ymax|
|  #12|   Numeric value|    Equality constraint 2x1|   x_1 == A*x_0 + B*u_0|
|  #13|   Numeric value|    Equality constraint 2x1|   y_0 == C*x_0 + D*u_0|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Then to remove certain constraints all you need to do is to substract the constraint you want to get rid of, identified by its tag. For instance

```
>> CON = CON - CON('x_2 in Tset');
>> CON = CON - CON('xmin < x_2 < xmax');
>> CON
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   ID|      Constraint|                         Type|                 Tag|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   #1|   Numeric value|           Element-wise 2x1|      umin < u_1 < umax|
```

```
|   #2|    Numeric value|           Element-wise 4x1|       xmin < x_1 < xmax|
|   #3|    Numeric value|           Element-wise 4x1|       ymin < y_1 < ymax|
|   #4|    Numeric value|   Equality constraint 2x1|   x_2 == A*x_1 + B*u_1|
|   #5|    Numeric value|   Equality constraint 2x1|   y_1 == C*x_1 + D*u_1|
|   #6|    Numeric value|           Element-wise 4x1|             x_0 in Pbnd|
|   #7|    Numeric value|           Element-wise 2x1|       umin < u_0 < umax|
|   #8|    Numeric value|           Element-wise 4x1|       xmin < x_0 < xmax|
|   #9|    Numeric value|           Element-wise 4x1|       ymin < y_0 < ymax|
|  #10|    Numeric value|   Equality constraint 2x1|   x_1 == A*x_0 + B*u_0|
|  #11|    Numeric value|   Equality constraint 2x1|   y_0 == C*x_0 + D*u_0|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

will remove any state constraints imposed on the last predicted state x_2. Alternativelly, it is also possible to identify constraints by their index (ID number in the first column of the above table). For example to remove the constraint on u_0 (constraint number 7 in the list above), one can do

```
>> CON = CON - CON(7)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   ID|       Constraint|                       Type|                     Tag|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   #1|    Numeric value|           Element-wise 2x1|       umin < u_1 < umax|
|   #2|    Numeric value|           Element-wise 4x1|       xmin < x_1 < xmax|
|   #3|    Numeric value|           Element-wise 4x1|       ymin < y_1 < ymax|
|   #4|    Numeric value|   Equality constraint 2x1|   x_2 == A*x_1 + B*u_1|
|   #5|    Numeric value|   Equality constraint 2x1|   y_1 == C*x_1 + D*u_1|
|   #6|    Numeric value|           Element-wise 4x1|             x_0 in Pbnd|
|   #7|    Numeric value|           Element-wise 4x1|       xmin < x_0 < xmax|
|   #8|    Numeric value|           Element-wise 4x1|       ymin < y_0 < ymax|
|   #9|    Numeric value|   Equality constraint 2x1|   x_1 == A*x_0 + B*u_0|
|  #10|    Numeric value|   Equality constraint 2x1|   y_0 == C*x_0 + D*u_0|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Notice that while the string tags associated to each constraint remain absolute, the relative position of constraints given by the ID number may change.

6.5 Soft constraints

Since MPT 2.6 it is possible to denote certain constraints as soft. This means that the respective constraint can be violated, but such a violation is penalized. To soften certain constraints, it is necessary to define the penalty on violation of such constraints:

- `probStruct.Sx` - if given as a "nx" x "nx" matrix, all state constraints will be treated as soft constraints, and violation will be penalized by the value of this field.
- `probStruct.Su` - if given as a "nu" x "nu" matrix, all input constraints will be treated as soft constraints, and violation will be penalized by the value of this field.

- `probStruct.Sy` - if given as a ”ny” x ”ny” matrix, all output constraints will be treated as soft constraints, and violation will be penalized by the value of this field.

In addition, one can also specify the maximum value by which a given constraint can be exceeded:

- `probStruct.sxmax` - must be given as a ”nx” x 1 vector, where each element defines the maximum admissible violation of each state constraints.
- `probStruct.sumax` - must be given as a ”nu” x 1 vector, where each element defines the maximum admissible violation of each input constraints.
- `probStruct.symax` - must be given as a ”ny” x 1 vector, where each element defines the maximum admissible violation of each output constraints.

The aforementioned fields also allow to tell that only a subset of state, input, or output constraint should be treated as soft constraints, while the rest of them remain hard. Say, for instance, that we have a system with 2 states and we want to soften only the second state constraint. Then we would write:

```
>> probStruct.Sx = diag([1 1000])
>> probStruct.sxmax = [0; 10]
```

Here `probStruct.sxmax(1)=0` tells $\mathbb{MPT}$ that the first constraint should be treated as a hard constraint, while we are allowed to exceed the second constraints at most by the value of 10, while every such violation will be penalized by the value of 1000.

Please note that soft constraints are not available for minimum-time (`probStruct.subopt_lev=1`) and low-complexity (`probStruct.subopt_lev=2`) strategies.

6.6 Control of time-varying systems

Time-varying system dynamics or systems with time-varying constraints can now be used for synthesis of optimal controllers. There are couple of limitations, though:

- Number of states, inputs and outputs must remain identical for each system.
- You cannot use time-varying systems in time-optimal (`probStruct.subopt_lev=1`) and low-complexity (`probStruct.subopt_lev=2`) strategies.

To tell $\mathbb{MPT}$ that it should consider a time-varying system, define one system structure for each step of the prediction, e.g.

```
>> Double_Integrator
>> S1 = sysStruct;
>> S2 = sysStruct; S2.C = 0.9*S1.C;
>> S3 = sysStruct; S3.C = 0.8*S1.C;
```

Here we have three different models which differ in the `C` element. Now we can define the time-varying model as a cell array of system structures by

```
>> model = {S1, S2, S3};
>> probStruct.N = 3;
```

Notice that order of systems in the `model` variable determines that the system `S1` will be used to make predictions of states `x(1)`, while the predicted value of `x(2)` will be determined based on model `S2`, and so on. Once the model is defined, you can now compute either the explicit, or an on-line MPC controller using the standard syntax:

```
>> explicitcontroller = mpt_control(model, probStruct)
>> onlinecontroller   = mpt_control(model, probStruct, 'online')
```

Systems with time-varying constraints can be defined in similar fashion, e.g.

```
>> Double_Integrator
>> S1 = sysStruct; S1.ymax = [5; 5]; S1.ymin = [-5; -5];
>> S2 = sysStruct; S2.ymax = [4; 4]; S2.ymin = [-4; -4];
>> S3 = sysStruct; S3.ymax = [3; 3]; S3.ymin = [-3; -3];
>> S4 = sysStruct; S4.ymax = [2; 2]; S4.ymin = [-2; -2];
>> probStruct.N = 4;
>> ctrl = mpt_control({S1, S2, S3, S4}, probStruct);
```

You can go as far as combining different classes of dynamical systems at various stages of the predictions, for instance you can arbitrary combine linear, Piecewise-Affine (PWA) and Mixed Logical Dynamical (MLD) systems. For instance you can use a detailed PWA model for the first prediction, while having a simple LTI model for the rest:

```
>> pwa_DI; pwa = sysStruct;                    % PWA model with 4 dynamics
>> Double_Integrator; lti = sysStruct;         % simple LTI model
>> probStruct.N = 5;
>> model = {pwa, pwa, lti, lti, lti};
>> ctrl = mpt_control(model, probStruct);
```

6.7 On-line MPC for nonlinear systems

With MPT 2.6 you can now solve on-line MPC problems based on nonlinear or piecewise nonlinear systems. In order to define models of such systems, one has to create a special function based on the `mpt_nonlinfcn.m` template (see for instance the `duffing_oscillator.m` or `pw_nonlin.m` examples contained in your MPT distribution). Once the describing function is defined, you can use `mpt_sys` to convert it into format suitable for further computation:

```
>> sysStruct = mpt_sys(@function_name)
```

where `function_name` is the name of the function you have just created. Now you can construct an on-line MPC controller using the standard syntax:

```
>> ctrl = mpt_control(sysStruct, probStruct, 'online');
```

or

```
>> [C, O, V] = mpt_ownmpc(sysStruct, probStruct, 'online');
% modify constraints and objective as needed
>> ctrl = mpt_ownmpc((sysStruct, probStruct, C, O, V, 'online');
```

After that you can use the controller either in Simulink, or in Matlab-based simulations invoked either by

```
>> u = ctrl(x0);
```

or by

```
>> [X, U, Y] = sim(ctrl, x0, number_of_simulation_steps)
>> simplot(ctrl, x0, number_of_simulation_steps)
```

Note: nonlinear problems are *very* difficult to solve, don't be surprised! Check the help of `mpt_getInput` for description of parameters which can affect quality and speed of the nonlinear solvers. Also note that currently only polynomial type of nonlinearities is supported, i.e. no `1/x` terms or `log/exp` functions are allowed. Moreover, don't even try to use nonlinear models for things like reachability or stability analysis, it wouldn't work.

6.8 Move blocking

Move blocking is a popular technique used to decrease complexity of MPC problems. In this strategy the number of free control moves is usually kept low, while some of the control moves are assumed to be fixed. To enable move blocking in $\mathbb{MPT}$, define the control horizon in

```
>> probStruct.Nc = Nc;
```

where `Nc` specifies the number of free control moves, and this value should be less than the prediction horizon `probStruct.N`. Control moves u_0 up to u_{N_c-1} will be then treated as free control moves, while $u_{N_c}, \ldots, u_{N-1}$ will be kept identical to u_{N_c-1}, i.e.

```
u_(Nc-1) == u_Nc == u_(Nc+1) == ... == u_(N-1)
```

6.9 Problem Structure `probStruct`

Problem structure `probStruct` is a structure which states an optimization problem to be solved by $\mathbb{MPT}$.

One and Infinity Norm Problems

The optimal control problem with a linear performance index is given by:

$$
\begin{aligned}
\min_{u(0),\ldots,u(N-1)} \quad & ||P_N x(N)||_p + \sum_{k=0}^{N-1} ||Ru(k)||_p + ||Qx(k)||_p \\
\text{subj. to} \quad & \\
& x(k+1) = f_{dyn}(x(k), u(k), w(k)) \\
& u_{min} \leq u(k) \leq u_{max} \\
& \Delta u_{min} \leq u(k) - u(k-1) \leq \Delta u_{max} \\
& y_{min} \leq g_{dyn}(x(k), u(k)) \leq y_{max} \\
& x(N) \in T_{set}
\end{aligned}
$$

where:

u	vector of manipulated variables over which the optimization is performed
N	prediction horizon
p	linear norm, can be 1 or `Inf` for 1- and Infinity-norm, respectively
Q	weighting matrix on the states
R	weighting matrix on the manipulated variables
P_N	weight imposed on the terminal state
u_{min}, u_{max}	constraints on the manipulated variable(s)
Δu_{min}, du_{max}	constraints on slew rate of the manipulated variable(s)
y_{min}, y_{max}	constraints on the system outputs
T_{set}	terminal set

the function $f_{dyn}(x(k), u(k), w(k))$ is the state-update function and is different for LTI and for PWA systems (see Section 5.7 for more details).

Quadratic Cost Problems

In case of a performance index based on quadratic forms, the optimal control problem takes the following form:

$$
\begin{aligned}
\min_{u(0),\ldots,u(N-1)} \quad & x(N)^T P_N x(N) + \sum_{k=0}^{N-1} u(k)^T R u(k) + x(k)^T Q x(k) \\
\text{subj. to} \quad & \\
& x(k+1) = f_{dyn}(x(k), u(k), w(k)) \\
& u_{min} \leq u(k) \leq u_{max} \\
& \Delta u_{min} \leq u(k) - u(k-1) \leq \Delta u_{max} \\
& y_{min} \leq g_{dyn}(x(k), u(k)) \leq y_{max} \\
& x(N) \in T_{set}
\end{aligned}
$$

If the problem is formulated for a fixed prediction horizon N, we refer to it as to Constrained Finite Time Optimal Control (CFTOC) problem. On the other hand, if N is infinity, the Constrained Infinite Time Optimal Control (CITOC) problem is formulated. Objective of the optimization is to choose the manipulated variables such that the performance index is minimized.

Mandatory fields of `probStruct`

In order to specify which problem the user wants to solve, mandatory fields of the problem structure `probStruct` are listed in Table 6.3.

`probStruct.N`	prediction horizon
`probStruct.Q`	weights on the states
`probStruct.R`	weights on the inputs
`probStruct.norm`	either 1 or `Inf` for linear cost, or 2 for quadratic cost objective
`probStruct.subopt_lev`	level of optimality

Tab. 6.3: Mandatory fields of the problem structure `probStruct`.

Level of Optimality

MPT can solve different control strategies:

1. The cost-optimal solution leads to a control law which minimizes a given performance index. This strategy is enforced by

   ```
   probStruct.subopt_lev = 0
   ```

   The cost optimal solution for PWA systems is currently supported only for linear performance index, i.e. `probStruct.norm = 1` or `probStruct.norm = Inf`.
2. Another possibility is to use the time-optimal solution, i.e. the control law will push a given state to an invariant set around the origin as fast as possible. This strategy usually leads to simpler control laws, i.e. less controller regions are generated. This approach is enforced by

   ```
   probStruct.subopt_lev = 1
   ```

3. The last option is to use a low-complexity control scheme. This approach aims at constructing a one-step solution and subsequent a PWQ or PWA Lyapunov function computation is performed to verify stability properties. The approach generally results in a small number of regions and asymptotic stability as well as closed-loop constraint satisfaction is guaranteed. If you want to use this kind of solution, use:

   ```
   probStruct.subopt_lev = 2
   ```

Optional fields of `probStruct`

Optional fields are summarized in Table 6.4.

`probStruct.Qy`	used for output regulation. If provided the additional term $\|Q(y - y_{ref})\|_p$ is introduced in the cost function and the controller will regulate the output(s) to the given references (usually zero, or provided by `probStruct.yref`.
`probStruct.tracking`	0/1/2 flag **0** – no tracking, resulting controller is a state regulator which drives all system states (or outputs, if probStruct.Qy is given) towards origin **1** – tracking with Δu-formulation. The controller will drive the system states (or outputs, if `probStruct.Qy` is given) to a given reference. The optimization is performed over the difference of manipulated variables ($u(k) - u(k-1)$), which involves an extension of the state vector by nu additional states where nu is the number of system inputs. **2** – tracking without Δu-formulation. The same as `probStruct.tracking=1` with the exception that the optimization is performed over $u(k)$, i.e. no Δu-formulation is used and no state vector extension is needed. Note, however, that offset-free tracking cannot be guaranteed with this setting. Default setting is `probStruct.tracking = 0`.
`probStruct.y0bounds`	boolean variable (**1** means **yes**, **0** stands for **no**) denoting whether or not to impose constraints also on the initial system output (default is 0)
`probStruct.yref`	instead of driving a state to zero, it is possible to reformulate the control problem and rather force the output to zero. To ensure this task, define `probStruct.Qy` which penalizes the difference of the actual output and the given reference.
`probStruct.P_N`	weight on the terminal state. If not specified, it is assumed to be zero for quadratic cost objectives, or $P_N = Q$ for linear cost.
`probStruct.Nc`	control horizon. Specifies the number of free control moves in the optimization problem.
`probStruct.Tset`	a polytope object describing the terminal set. If not provided and `probStruct.norm = 2`, the invariant LQR set around the origin will be computed automatically to guarantee stability properties.
`probStruct.Tconstraint`	an integer (0, 1, 2) denoting which stability constraint to apply. 0 – no terminal constraint, 1 – use LQR terminal set, 2 – use user-provided terminal set constraint. Note that if `probStruct.Tset` is given, `Tconstraint` will be set to 2 automatically.
`probStruct.feedback`	boolean variable, if set to 1, the problem is augmented such that $U = Kx + c$ where K is a state-feedback gain (typically an LQR controller) and the optimization aims to identify the proper offset c. (default is 0)
`probStruct.FBgain`	if the former option is activated, a specific state-feedback gain matric K can be provided (otherwise an LQR controller will be computed automatically)

Tab. 6.4: Optional field of the `probStruct` structure.

7

Analysis and Post-Processing

The toolbox offers broad functionality for analysis of hybrid systems and verification of safety and liveliness properties of explicit control laws. In addition, stability of closed-loop systems can be verified using different types of Lyapunov functions.

7.1 Reachability Computation

$\mathbb{MPT}$ can compute forward N-steps reachable sets for linear and hybrid systems assuming the system input either belongs to some bounded set of inputs, or when the input is driven by some given explicit control law.

To compute the set of states which are reachable from a given set of initial conditions `X0` in `N` steps assuming system input $u(k) \in \mathcal{U}_0$, one has to call:

```
R = mpt_reachSets(sysStruct, X0, U0, N);
```

where `sysStruct` is the system structure, `X0` is a polytope which defines the set of initial conditions ($x(0) \in \mathcal{X}_0$), `U0` is a polytope which defines the set of admissible inputs and `N` is an integer which specifies for how many steps should the reachable set be computed. The resulting reachable sets `R` are returned as a polytope array. We illustrate the computation on the following example:

Example 7.1.1: First we define the dynamical system for which we want to compute reachable sets

```
% define matrices of the state-space object
A = [-1 -4; 4 -1]; B = [1; 1]; C = [1 0]; D = 0;
syst = ss(A, B, C, D);
Ts = 0.02;

% create a system structure by discretizing the continous-time model
sysStruct = mpt_sys(syst, Ts);

% define system constraints
sysStruct.ymax = 10; sysStruct.ymin = -10;
sysStruct.umax = 1; sysStruct.umin = -1;
```

Now we can define a set of initial conditions X0 and a set of admissible inputs U0 as polytope objects.

```
% set of initial states
X0 = polytope([0.9 0.1; 0.9 -0.1; 1.1 0.1; 1.1 -0.1]);

% set of admissible inputs
U0 = unitbox(1,0.1);  % inputs should be such that |u| <= 0.1
```

Finally we can compute the reachable sets.

```
N = 50;
R = mpt_reachSets(sysStruct, X0, U0, N);

% plot the results
plot(X0, 'r', R, 'g');
```

The reachable sets (green) as well as the set of initial conditions (red) are depicted in Figure 7.1.

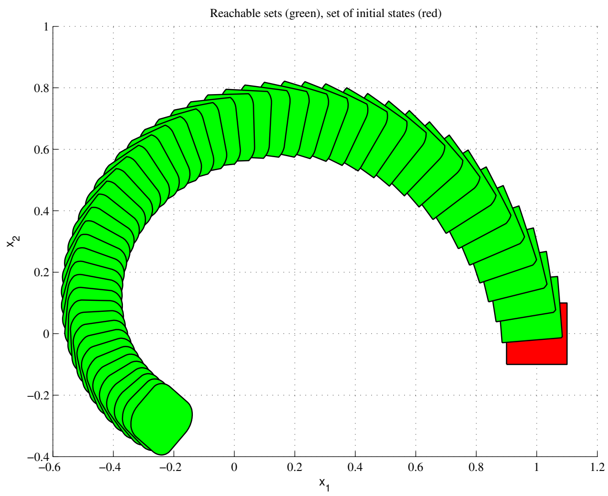


Fig. 7.1: Reachable sets for Example 7.1.1.

To compute reachable sets for linear or hybrid systems whose inputs are driven by an explicit control law, the following syntax can be used:

```
R = mpt_reachSets(ctrl, X0, N);
```

where ctrl is the controller object as generated by mpt_control, X0 is a polytope which defines a set of initial conditions ($x(0) \in \mathcal{X}_0$), and N is an integer which specifies for how many steps should the reachable set be computed. The resulting reachable sets R are again returned as polytope array.

Example 7.1.2: In this example we illustrate the reachability computation on the *double integrator* example

```
% load system and problem parameters
Double_Integrator

% compute explicit controller
ctrl = mpt_control(sysStruct, probStruct);

% define the set of initial conditions
X0 = unitbox(2,1) + [3;0];

% compute the 5-Steps reachable set
N = 5;
R = mpt_reachSets(ctrl, X0, N);

% plot results
plot(ctrl.Pn, 'y', X0, 'r', R, 'g');
```

The reachable sets (green) as well as the set of initial conditions (red) are depicted on top of the controller regions (yellow) in Figure 7.2.

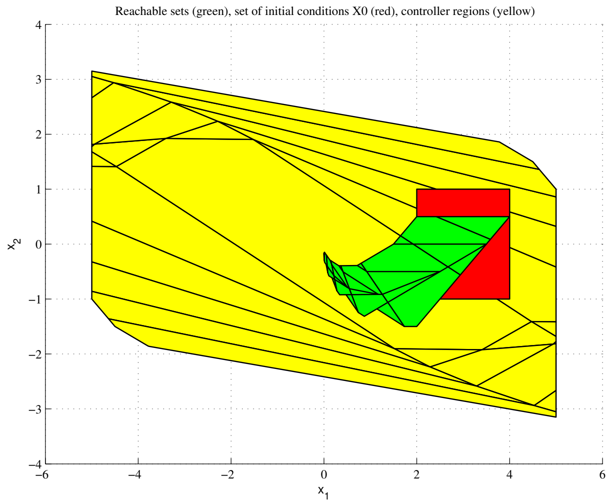


Fig. 7.2: Reachable sets for Example 7.1.2.

7.2 Verification

Reachability computation can be directly extended to answer the following question: *Do states of a dynamical system (whose inputs either belong to some set of admissible inputs, or whose inputs are driven by an explicit control law) enter some set of "unsafe" states in a given number of steps?*

Example 7.2.1: In this example we show how to answer the verification question for the first case, i.e. system inputs belong to some set of admissible inputs ($u(k) \in \mathcal{U}_0$). Although we use a linear system here, exactly the same procedure applies to hybrid systems in PWA representation as well.

```
% define matrices of the state-space object
A = [-1 -4; 4 -1]; B = [1; 1]; C = [1 0]; D = 0;
syst = ss(A, B, C, D);
Ts = 0.02;

% create a system structure by discretizing the continous-time model
sysStruct = mpt_sys(syst, Ts);

% define system constraints
sysStruct.ymax = 10; sysStruct.ymin = -10;
sysStruct.umax = 1; sysStruct.umin = -1;

% define the set of initial condintions as a polytope object
X0 = polytope([0.9 0.1; 0.9 -0.1; 1.1 0.1; 1.1 -0.1]);

% set of admissible inputs as a polytope object
U0 = unitbox(1,0.1);  % inputs should be such that |u| <= 0.1

% set of final states (the ``unsafe'' states)
Xf = unitbox(2,0.1) + [-0.2; -0.2];

% number of steps
N = 50;

% perform verification
[canreach, Nf] = mpt_verify(sysStruct, X0, Xf, N, U0);
```

If system states can reach the set `Xf`, `canreach` will be *true*, otherwise the function will return *false*. In case `Xf` can be reached, the optional second output argument `Nf` will return the number of steps in which `Xf` can be reached from `X0`.

Example 7.2.2: It is also possible to answer the verification question if the system inputs are driven by an explicit control law:

```
% load dynamical system
Double_Integrator

% compute explicit controller
expc = mpt_control(sysStruct, probStruct);

% define set of initial condintions as a polytope object
```

```
X0 = unitbox(2,1) + [3;0];

% set of final states (the ``unsafe'' states)
Xf = unitbox(2,0.1) + [-0.2; -0.2];

% number of steps
N = 10;

% perform verification
[canreach, Nf] = mpt_verify(expc, X0, Xf1, N);
```

7.3 Invariant set computation

For controllers for which no feasibility guarantee can be given a priori, the function `mpt_invariantSet` can compute an invariant subset of a controller, such that constraints satisfaction is guaranteed for all time.

```
ctrl_inv = mpt_invariantSet(ctrl)
```

7.4 Lyapunov type stability analysis

In terms of stability analysis, $\mathbb{MPT}$ offers functions which aim at identifying quadratic, sum-of-squares, piecewise quadratic, piecewise affine or piecewise polynomial Lyapunov functions. If such a function is found, it can be used to show stability of the closed-loop systems even in cases where no such guarantee can be given a priori. To compute a Lyapunov function, one has to call

```
ctrl_lyap = mpt_lyapunov(ctrl, lyaptype}
```

where `ctrl` is an explicit controller and `lyaptype` is a string parameter which defines type of a Lyapunov function to compute. Allowed values of the second parameter are summarized in Table 7.1. Parameters of the Lyapunov function, if one exists, will be stored in

```
lyapfunction = ctrl_lyap.details.lyapunov
```

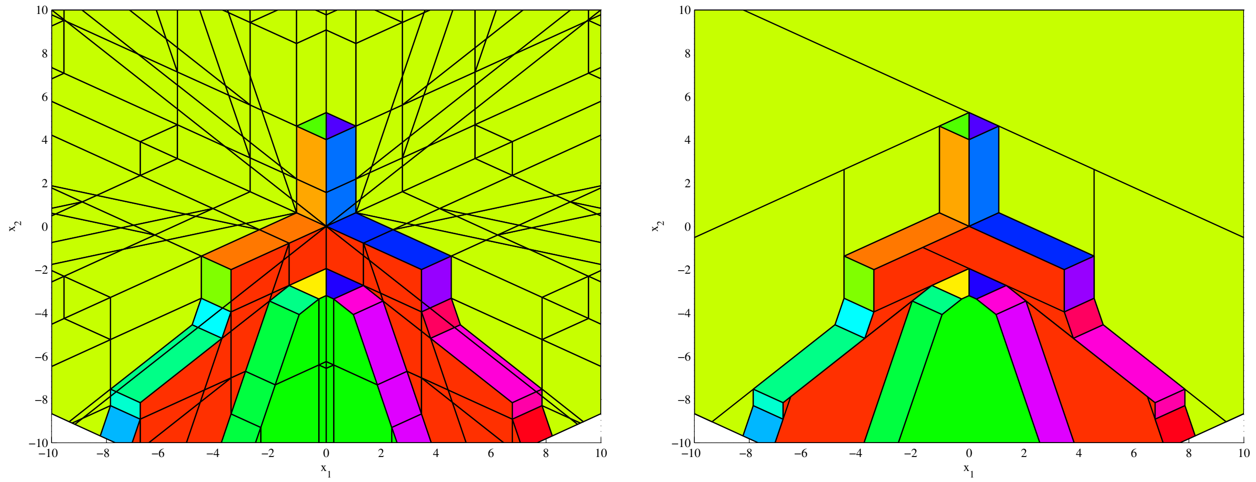
7.5 Complexity Reduction

$\mathbb{MPT}$ also addresses the issue of complexity reduction of resulting explicit control laws. As mentioned in previous sections, in order to apply an explicit controller to a real plant, a proper control law has to be identified. This involves checking which region of an explicit controller

lyaptype	Type of Lyapunov function
'quad'	Common quadratic Lyapunov function
'sos'	Common sum-of-squares Lyapunov function
'pwa'	Piecewise affine Lyapunov function
'pwq'	Piecewise quadratic Lyapunov function
'pwp'	Piecewise polynomial Lyapunov function

Tab. 7.1: Allowed values of the `functiontype` parameter in `mpt_lyapunov`.

contains a given measured state. Although such effort is usually small, it can become prohibitive for very complex controllers with several thousands or even more regions. MPT therefore allows to reduce this complexity by simplifying the controller partitions over which the control law is defined. This simplification is performed by merging regions which contain the same expression of the control law. By doing so, the number of regions is greatly reduced, while maintaining the same performance as the original controller. Results of the merging procedure for a sample explicit controller of a hybrid system is depicted in Figure 7.3.



(a) Regions of an explicit controller before simplification (252 regions).

(b) Regions of an explicit controller after simplification (39 regions).

Fig. 7.3: Region merging results.

To simplify the representation of a given explicit controller by merging regions which contain the same control law, one has to call:

```
ctrl_simple = mpt_simplify(ctrl)
```

If the function is called as indicated above, a heuristical merging will be used. It is also possible to use optimal merging based on boolean minimization:

```
ctrl_simple = mpt_simplify(ctrl, 'optimal')
```

Note, however, that the optimal merging can be prohibitive for dimensions above 2.

8

Implementation of Control Law

8.1 Algorithm

The control law obtained as a result of `mpt_control` is stored in a respective controller object `mptctrl` (see Section 6.2 for more details). The explicit controller takes the form a of Piecewise Affine control law where the actual control action is given by

$$U(k) = Kx(k) + F_i^r x(k) + G_i^r \tag{8.1}$$

where the superindex r denotes the active region, i.e. the region which contains the given state $x(k)$. If the solution was obtained with feedback pre-stabilization enabled (`probStruct.feedback=1`) K is the feedback gain (either user-provided or computed and stored in `ctrl.probStruct.FBgain`). K will be zero if pre-stabilization was not requested.

In the controller structure, matrices F_i and G_i are stored as cell arrays, i.e.

```
ctrl.Fi = { Fi{1} Fi{2} ...  Fi{n} }
ctrl.Gi = { Gi{1} Gi{2} ...  Gi{n} }
```

Regions of the state-space where each affine control (8.1) is active are stored as a polytope array in the following field:

```
ctrl.Pn = [ Pn(1) Pn(2) ... Pn(n)]
```

Moreover, expression of the value function is stored in

```
ctrl.Ai = { Ai{1} Ai{2} ... Ai{n} }
ctrl.Bi = { Bi{1} Bi{2} ... Bi{n} }
ctrl.Ci = { Ci{1} Ci{2} ... Ci{n} }
```

Cost associated to a given state $x(k)$ can therefore easily be obtained by simply evaluating the cost expression, which is defined by

$$J = x(k)^T A_i^r x(k) + B_i^r x(k) + C_i^r \tag{8.2}$$

The procedure to obtain the control action for a given state $x(k)$ therefore reduces to a simple membership-test. First, index of the active region r has to be identified. Since the polyhedral partition is a polytope object, the function `isinside` will return indices of regions which contain the given state $x(k)$. Since certain types of optimization problems naturally generate overlapped regions, the active region corresponds to the region in which the cost expression (8.2) is minimal. Once the active region is identified, the control action is calculated according to (8.1) and can be applied to the system.

If the optimal control problem was solved for a fixed prediction horizon N, evaluation (8.1) gives a vector of control moves which minimize the given performance criterion, i.e.

$$U \triangleq [u(0)^T u(1)^T ... u(N)^T]^T \tag{8.3}$$

When applying the obtained control law in the closed-loop, only the first input $u(0)$ is extracted from the sequence U and is applied to the system. This policy is refereed to as the Receding Horizon Policy.

The algorithm to identify the active control law is summarized below:

Algorithm 8.1.1: getInput($x_0, P_n, F_i, G_i, A_i, B_i, C_i$)

Input: Polyhedral partition P_n, PWA control law F_i, G_i, matrices of the cost expression A_i, B_i, C_i.

Output: Optimal control action U associated to a given state, Index of the active region r

1. Identify regions of P_n which contain the point x_0. Denote array of of the associated regions by R.
2. IF $R = \emptyset$, return ERROR - No associated control law found.
3. Set $\mathcal{J} = \emptyset$
4. FOR each element of R DO
   a) $J = x_0^T A_i^r x_0 + B_i^r x_0 + C_i$
   b) Add the ordered pair $\{J, r\}$ to $\mathcal{J}$
5. END FOR
6. Identify the minimal cost from the set of ordered pairs $\mathcal{J}$.
7. Extract from $\mathcal{J}$ the region r associated to the minimal cost
8. Compute the optimal input sequence $U = Kx_0 + F_i^r x_0 + G_i$ (K will be zero unless feedback pre-stabilization enabled.
9. Return U, r

8.2 Implementation

The Algorithm 8.1.1 is implemented by the function `mpt_getInput`. Syntax of the function is the following

```
[U, feasible, region, cost] = mpt_getInput(ctrl, x0)
```

`U`	Feedback control law obtained by (8.1)
`feasible`	Boolean variable (0/1) denoting whether there is at least one region which contains the point `x0` in it's interior
`region`	Index of the active region in `ctrl.Pn`
`cost`	Cost associated to the given state `x0`
`ctrl`	Controller structure
`x0`	State vector
`Options`	Additional optional arguments

Tab. 8.1: Input and output arguments of `mpt_getInput`.

```
[U, feasible, region, cost] = mpt_getInput(ctrl, x0, Options)
```

where the input and arguments are described in Table 8.1

The function returns the optimizer U associated to a region in which the cost expression (8.2) is minimal. If there is no region associated to a given state x_0, the variable `feasible` will be set to **0** (zero).

Unless specified otherwise, the function `mpt_getInput` returns only the first element of the sequence U (8.3), i.e. `U = u(0)`, which can be directly applied to the system to obtain the successor state $x(k+1)$. If the user wants to return the full sequence U, `Options.openloop` has to be set to **1**.

The above described function (`mpt_getInput`) processes the controller structure as an input argument. If, for any reason, the solution to a given multi-parametric program was obtained by a direct call to `mpt_mpLP` or `mpt_mpQP`, the function

```
[U, feasible, region]=mpt_getOptimizer(Pn, Fi, Gi, x0, Options)
```

can be used to extract the sequence of arguments which minimize the given performance criterion. Note that unlike Algorithm 8.1.1, `mpt_getOptimizer` does not take into account overlaps. This is due to the fact that overlapping regions are not (usually) not generated by mpLP and mpQP algorithms which are implemented in $\mathbb{MPT}$.

The function `sim` calculates the open-loop or closed-loop state evolution from a given initial state x_0. In each time step, the optimal control action is calculated according to Algorithm 8.1.1 by calling `mpt_getInput`. Subsequently, the obtained control move is applied to the system to obtain the successor state $x(k+1)$. The evolution is terminated once the state trajectory reaches the origin. Because of numerical issues, a small box centered at origin is constructed an the evolution is stopped as soon as all states enter this small box. Size of the box can be specified by the user. For tracking problems, the evolution is terminated when all states reach their respective reference signals. Validation of input and output constraints is performed automatically and the user is provided with a textual output if the bounds are exceeded.

General syntax is the following:

```
[X,U,Y]=sim(ctrl,x0)
[X,U,Y]=sim(ctrl,x0,N)
[X,U,Y]=sim(ctrl,x0,N,Options)
[X,U,Y,cost,feasible]=sim(ctrl,x0,N)
```

X	Matrix containing evolution of the system states, i.e. $X = [x(0)x(1)...x(n+1)]^T$
U	Matrix containing the control actions applied at each time step, i.e. $U = [u(0)u(1)...u(n)]^T$
Y	Matrix containing the evolution of system outputs, i.e. $Y = [y(0)y(1)...y(n)]^T$
cost	Overall cost obtained as a sum of (8.2).
feasible	Boolean variable (0/1) denoting whether there is at least one region which contains the point x0 in it's interior
ctrl	Controller object
x0	Initial conditions
N	Number of steps for which the evolution should be computed.
Options	Additional optional arguments

Tab. 8.2: Input and output arguments of the sim function.

```
[X,U,Y,cost,feasible]=sim(ctrl,x0,N,Options)
```

where the input and output arguments are summarized in Table 8.2. **Note:** If the third argument is an empty matrix (N = []), the evolution will be automatically stopped when system states (or system outputs) reach a given reference point with a pre-defined tolerance.

The trajectories can be visualized using the simplot function:

```
simplot(ctrl)
simplot(ctrl, x0)
simplot(ctrl, x0, N)
```

If x0 is not provided and the controller partition is in $\mathbb{R}^2$, you will be able to specify the initial state just by clicking on the controller partition.

Using different dynamical system in sim and simplot

It is possible to specify your own dynamical system to use for simulations. In such case control actions obtained by a given controller can be applied to a different system than that which was used for computing the controller:

```
sim(ctrl, system, x0, N, Options)
simplot(ctrl, system, x0, N, Options)
```

Note that the N and Options arguments are optional. You can specify your own dynamics in two ways:

1. By setting the system parameter to a system structure, i.e.

   ```
   sim(ctrl, sysStruct, x0, N, Options)
   ```

2. By setting the system parameter to a handle of a function which will provide updates of system states in a discrete-time fashion:

   ```
   sim(ctrl, @sim_function, x0, N, Options)
   ```

   Take a look at help di_sim_fun on how to write simulation functions compatible with this function.

8.3 Simulink library

MPT Simulink library can be accessed by starting

```
>> mpt_sim
```

on Matlab command prompt. By this time the library offers 3 blocks:

The `MPT Controller` block supplies (sub)optimal control action as a function of measured state. Auxiliary state/output references can be provided for tracking controllers (`probStruct.tracking = 1|2`. If the controller is an explicit one, it is possible to directly compile a Simulink model which includes one or more of the `MPT Controller` blocks using the Real Time Workshop.

The `Dynamical System` block serves for simulations of constrained linear and hybrid systems described by means of MPT `sysStruct` structures. The user must specify initial values of the state vector in a dialog box.

The `In polytope` block returns *true* if a input point lies inside of a given polytope, *false* otherwise. If the polytope variable denotes a polytope array, the output of this block will be the index of a region which contains a given point. If no such region exists, 0 (zero) will be returned.

8.4 Export of controllers to C-code

It is possible to export explicit controllers to standalone code using

```
  mpt_exportc(ctrl)
  mpt_exportc(ctrl, filename)
```

If the function is called with only one input argument, a file called `mpt_getInput.h` will be created in the working directory. It is possible to change the filename by providing second input argument to `mpt_exportc`. The header file is then compiled along with `mpt_getInput.c` and your target application. For more information, see the demo in `mpt/examples/ccode/mpt_example.c`:

```
% generate an explicit controller using 'mpt_control'
>> Double_Integrator
>> controller = mpt_control(sysStruct, probStruct);

% export the explicit controller to C-code
>> mpt_exportc(controller);

% compile the example
>> !gcc mpt_example.c -o mpt_example
```

8.5 Export of search trees to C-code

If a binary search tree was calculated for a given controller by calling `mpt_searchTree`, it is possible to export such tree into a standalone C-file by calling

```
>> mpt_exportST(ctrl, filename)
```

where the `filename` argument specifies the name of the file which should be created. The controller `ctrl` used in this example must have the search tree stored inside. If it does not, use the `mpt_searchTree` function to calculate it first:

```
>> ctrl = mpt_searchTree(ctrl);
```

9

Visualization

MPT provides various functionality for visualization of polytopes, polyhedral partitions, control laws, value functions, general PWA and PWQ functions defined over polyhedral partitions. Part of the functions operate directly on the resulting controller object `ctrl` obtained by `mpt_control`, while the other functions accept more general input arguments. Please consult help files of individual functions for more details.

9.1 Plotting of polyhedral partitions

Explicit solution to a optimal control problem results in a PWA control law which is defined over regions of polyhedral partition. If the solution was obtained by a call to `mpt_control`, it is returned back in the form of the controller object `ctrl`, which encompasses the polyhedral partition over which the control law is defined (see Section 6.2 for more details). The polyhedral partition `ctrl.Pn` is a polytope object and can therefore be plotted using the overloaded `plot` function. However, MPT provides also more sophisticated plotting method, where, depending on the type of the solution, regions are depicted in approapriate colors which helps to understand behavior of the controller. This kind of plots is obtained by a call to

```
plot(ctrl)
```

i.e. the `plot` function is overloaded to accept `mptctrl` objects directly.

If `ctrl` contains a solution to Constrained Infinite Time Optimal Control Problem, or to Constrained Time Optimal Control Problem, the regions are depicted in a red-green shading. Generally speaking, red regions are close to the origin, while the more green color the region contains, the more steps will be needed to reach the desired origin.

9.2 Visualization of closed-loop and open-loop trajectories

Once the explicit solution to a given optimal control problem is obtained, the resulting control law can be applied to the original dynamical system. MPT provides several functions for a user-friendly way of visualizing the state trajectories which are subject to control. As mentioned already in Chapter 8, the PWA feedback law which corresponds to a given state $x(k)$ has to be isolated and evaluated in order to obtain the successor state $x(k+1)$. Moreover, when applying the RHC strategy, this procedure has to be repeated at each time instance. The function `sim` described in Section 8.2 can be used to perform this repeated evaluation, and subsequently returns evolution of state, input and output trajectories assuming the initial state $x(0)$ was provided. To visualize the computed trajectories, following command can be used:

```
simplot(ctrl)
```

which allows to pick up the initial state $x(0)$ by a mouse click, providing the controller object represents an explicit controller and dimension of the associated polyhedral partition is equal to 2. Subsequently, state trajectory is calculated on plotted on top of the polyhedral partition over which the control law is defined. If the solution was obtained for a tracking problem, the user is first prompted to choose the reference point, again by a mouse click. Afterwards, the initial state $x(0)$ has to be selected. Finally, evolution of states is plotted again versus the polyhedral partition.

If the same command is used with additional input arguments, e.g.

```
simplot(ctrl, x0, horizon)
```

then the computed trajectories are visualized with respect to time. The system is not limited in dimension or number of manipulated variables. Unlike the point-and-click interface, the initial point $x(0)$ has to be provided by the user. In addition, the maximal number of steps can be specified in `horizon`. If this variable is missing, or set to an empty matrix, the evolution will continue until origin (or the reference point for tracking problems) is reached. Additional optional argument `Options` can be provided to specify additional requirements. Similarly as described by Section 8.2, also the `simplot` function allows the user to use different system dynamics when calculating the system evolution. Check the help description of `mptctrl/simplot` for more details.

9.3 Visualization of general PWA and PWQ functions

A Piecewise affine function is defined by

$$f(x) = L^r x + C^r \qquad \text{if } x \in P_n^r \tag{9.1}$$

where the superindex r indicates that the expression for the function is different in every region r of a polyhedral partition P_n.

Piecewise Quadratic functions can be described as follows

$$f(x) = x^T M^r x + L^r x + C^r \qquad \text{if } x \in P_n^r \tag{9.2}$$

Again, expression for the cost varies in different regions of the polyhedral set P_n.

$\mathbb{MPT}$ allows you to visualize both aforementioned types of functions.

The command

```
mpt_plotPWA(Pn, L, C)
```

plots the PWA function (9.1) defined over the polyhedral partition `Pn`. Typical application of this function is to visualize the control law and value function obtained as a solution to a given optimal control problem. For the first case (visualization of control action), one would type:

```
mpt_plotPWA(ctrl.Pn, ctrl.Fi, ctrl.Gi)
```

since the control law is affine over each polytope of `ctrl.Pn`.

Note: The function supports 2-D partitions only.

To visualize the value function, one simply calls

```
mpt_plotPWA(ctrl.Pn, ctrl.Bi, ctrl.Ci)
```

to get the desired result. The same limitation applies also in this case.

Piecewise quadratic functions defined by (9.2) can be plotted by function

```
mpt_plotPWQ(Pn, Q, L, C, meshgridpoints)
```

Inputs are the polytope array `Pn`, cell arays `Q`, `L` and `C`. When plotting a PWQ function, the space covered by `Pn` has to be divided into a mesh grid. The fourth input argument (`meshgridpoints`) states into how many points should each axis of the space of interest be divided. Default value for this parameter is 30. Note that dimension of `Pn` has to be at most 2.

MPT provides a "shortcut" function to plot value of the control action with respect to the polyhedral partition directly, without the need to pass each input (`Pn, L, C`) separately:

```
mpt_plotU(ctrl)
```

If the function is called with a valid controller object, value of the control action in each region will be depicted. If the polyhedral partition `Pn` contains overlapping regions, the user will be prompted to use the appropriate reduction scheme (`mpt_removeOverlaps`) first to get a proper result. See `help mpt_plotU` for more details.

Similarly, values of the cost function associated to a given explicit controller can be plotted by

```
mpt_plotJ(ctrl)
```

Also in this case the partition is assumed to contain no overlaps. See `help mpt_plotJ` for more details and list of available options.

10

Examples

In order to obtain a feedback controller, it is necessary to specify both a system as well as the problem. We demonstrate the procedure on a simple second-order double integrator, with bounded input $|u| \leq 1$ and output $||y(k)||_\infty \leq 5$:

Example 10.0.1:

```
>> sysStruct.A=[1 1; 0 1];            % x(k+1)=Ax(k)+Bu(k)
>> sysStruct.B=[0 1];                 % x(k+1)=Ax(k)+Bu(k)
>> sysStruct.C=[1 0; 0 1];            % y(k)=Cx(k)+Du(k)
>> sysStruct.D=[0;0];                 % y(k)=Cx(k)+Du(k)

>> sysStruct.umin=-1;                 % Input constraints umin<=u(k)
>> sysStruct.umax=1;                  % Input constraints u(k)<=umax
>> sysStruct.ymin=[-5 -5]';           % Output constraints ymin<=y(k)
>> sysStruct.ymax=[5 5]';             % Output constraints y(k)<=ymax
>> sysStruct.xmin = [-5; -5];         % State constraints x(k)>=xmin
>> sysStruct.xmax = [5; 5];           % State constraints x(k)<=xmax
```

For this system we will now formulate the problem with quadratic cost objective in (3.12) and a prediction horizon of $N = 5$:

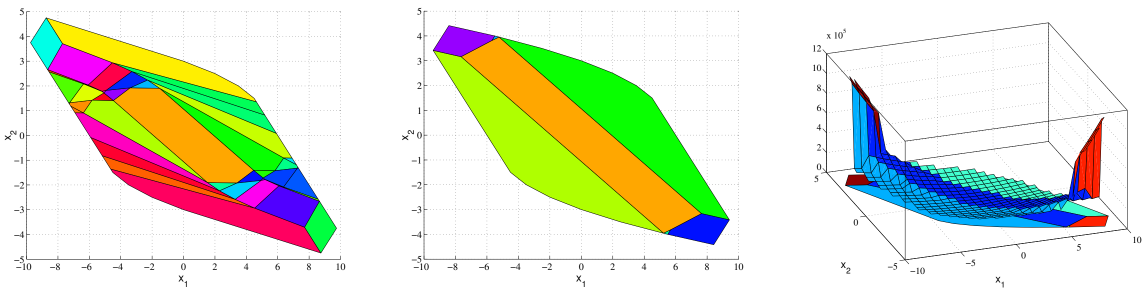
```
>> probStruct.norm=2;              %Quadratic Objective
>> probStruct.Q=eye(2);            %Objective: min_U J=sum x'Qx + u'Ru...
>> probStruct.R=1;                 %Objective: min_U J=sum x'Qx + u'Ru...
>> probStruct.N=5;                 %...over the prediction horizon 5
>> probStruct.subopt_lev=0;        %Compute optimal solution, not low complexity.
```

If we now call

```
>> ctrl=mpt_control(sysStruct,probStruct);   %Compute feedback controller
>> plot(ctrl)
```

the controller for the given problem is returned and plotted (see Figure 10.1(a)), i.e., if the state $x \in PA(i)$, then the optimal input for prediction horizon $N = 5$ is given by $u = F_i ix + G_i i$. If we wish to compute a low complexity solution, we can run the following:

```
>> probStruct.subopt_lev=2;        % Compute low complexity solution.
>> probStruct.N = 1;               % Use short prediction horizon
>> ctrl = mpt_control(sysStruct,probStruct);
>> plot(ctrl)                      % Plot the controller partition
```



(a) The $N = 5$ step optimal feedback solution.

(b) The iterative low complexity solution for the double integrator.

(c) Lyapunov function for the low complexity solution.

Fig. 10.1: Results obtained for Example 10.0.1.

```
>> Q = ctrl.details.lyapunov.Q;
>> L = ctrl.details.lyapunov.L;
>> C = ctrl.details.lyapunov.C;
>> mpt_plotPWQ(ctrl.finalPn,Q,L,C); % Plot the Lyapunov Function
```

The resulting partition and Lyapunov function is depicted in Figures 10.1(b) and 10.1(c) respectively. In the following we will solve the PWA problem introduced in [23] by defining two different dynamics which are defined in the left- and right half-plane of the state space respectively.

Example 10.0.2:

```
>> H=[-1 1; -3 -1; 0.2 1; -1 0; 1 0; 0 -1];  %Polytopic state constraints Hx(k)<=K
>> K=[  15;    25;      9;     6;    8;    10];  %Polytopic state constraints Hx(k)<=K

>> sysStruct.C{1} = [1 0];                 %System Dynamics 1: y(k)=Cx(k)+Du(k)+g
>> sysStruct.D{1} = 0;                     %System Dynamics 1: y(k)=Cx(k)+Du(k)+g
>> sysStruct.g{1} = [0];                   %System Dynamics 1: y(k)=Cx(k)+Du(k)+g
>> sysStruct.A{1} = [0.5 0.2; 0 1];        %System Dynamics 1: x(k+1)=Ax(k)+Bu(k)+f
>> sysStruct.B{1} = [0; 1];                %System Dynamics 1: x(k+1)=Ax(k)+Bu(k)+f
>> sysStruct.f{1} = [0.5; 0];              %System Dynamics 1: x(k+1)=Ax(k)+Bu(k)+f
>> sysStruct.guardX{1} = [1 0; H];         %Dynamics 1 defined in guardX*x <= guardC
>> sysStruct.guardC{1} = [  1; K];         %Dynamics 1 defined in guardX*x <= guardC

>> sysStruct.C{2} = [1 0];                 %System Dynamics 2: y(k)=Cx(k)+Du(k)+g
>> sysStruct.D{2} = 0;                     %System Dynamics 2: y(k)=Cx(k)+Du(k)+g
>> sysStruct.g{2} = [0];                   %System Dynamics 2: y(k)=Cx(k)+Du(k)+g
>> sysStruct.A{2} = [0.5 0.2; 0 1];        %System Dynamics 2: x(k+1)=Ax(k)+Bu(k)+f
>> sysStruct.B{2} = [0; 1];                %System Dynamics 2: x(k+1)=Ax(k)+Bu(k)+f
>> sysStruct.f{2} = [0.5; 0];              %System Dynamics 2: x(k+1)=Ax(k)+Bu(k)+f
>> sysStruct.guardX{2} = [-1 0; H];        %Dynamics 2 defined in guardX*x <= guardC
>> sysStruct.guardC{2} = [  -1; K];        %Dynamics 2 defined in guardX*x <= guardC

>> sysStruct.ymin = -10;                   %Output constraints for dynamic 1 and 2
>> sysStruct.ymax = 10;                    %Output constraints for dynamic 1 and 2
>> sysStruct.umin = -1;                    %Input constraints for dynamic 1 and 2
>> sysStruct.umax = 1;                     %Input constraints for dynamic 1 and 2
```

we can now compute the low complexity feedback controller by defining the problem

```
>> probStruct.norm=2;                %Quadratic Objective
>> probStruct.Q=eye(2);              %Objective: min_U J=sum x'Qx + u'Ru...
>> probStruct.R=0.1;                 %Objective: min_U J=sum x'Qx + u'Ru...
>> probStruct.subopt_lev=1;          %Compute low complexity controller.
```

and calling the control function,

```
>> ctrl=mpt_control(sysStruct,probStruct);
>> plot(ctrl)
```

The result is depicted in Figure 10.2.

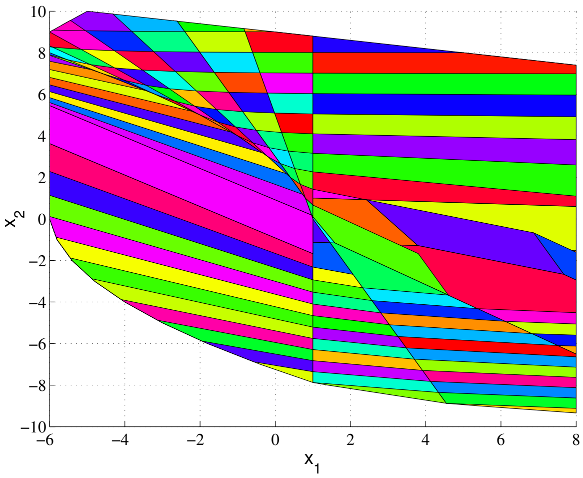


Fig. 10.2: Controller partition obtained for Example 10.0.2.

For more examples we recommend to look at the demos which can be found in respective subdirectories of the `mpt/examples` directory of your MPT installation.

11

Polytope Library

As already mentioned in Section 3.1, a polytope is a convex bounded set which can be represented either as an intersection of a finite number of half-spaces ($\mathcal{H}$-representation) or as a convex hull of vertices ($\mathcal{V}$-representation). Both ways of defining a polytope are allowed in $\mathbb{MPT}$ and you can switch from one representation to the other one. However, by default all polytopes are generated in $\mathcal{H}$-representation only to avoid unnecessary computation.

11.1 Creating a polytope

A polytope in $\mathbb{MPT}$ is created by a call to the polytope constructor as follows:

```
P = polytope(H,K)
```

creates a polytope by providing it's $\mathcal{H}$-representation, i.e. the matrices `H` and `K` which form the polytope $\mathcal{P} = \{x \in \mathbb{R}^n \mid Hx \leq K\}$. If input matrices define some redundant constraints, these will be automatically removed to form a minimal representation of the polytope. In addition, center and diameter of the largest ball which can be inscribed into the polytope are computed as well and the $\mathcal{H}$-representation is normalized to avoid numerical problems. The constructor then returns a polytope object.

Polytope can also be defined by it's vertices as follows:

```
P = polytope(V)
```

where `V` is a matrix which contains vertices of the polytope in the following format:

$$\mathtt{V} = \begin{bmatrix} v_{1,1} & \cdots & v_{1,n} \\ \vdots & \vdots & \vdots \\ v_{k,1} & \cdots & v_{k,n} \end{bmatrix} \tag{11.1}$$

where k is the total number of vertices and n is the dimension. Hence vertices are stored row-wise. Before the polytope object is created, $\mathcal{V}$-representation is first converted to half-space description by eliminating all points from `V` which are not extreme points. Convex hull of the remaining points is then computed to obtain the corresponding $\mathcal{H}$-representation. Extreme points will be stored in the polytope object and can be returned upon request without additional computational effort.

11.2 Accessing data stored in a polytope object

Each `polytope` object is internally represented as a structure, but because of the Object-Oriented approach, this information cannot be directly obtained by using structure deferencing through the . (dot) operator. Special functions have to be called in order to retrieve individual fields.

`H, K`	$\mathcal{H}$-representation of the polytope
`xCheb, RCheb`	Center and radius of Chebyshev's ball (largest ball inscribed in the polytope
`normal`	Flag whether the $\mathcal{H}$-representation is normalized (1/0)
`minrep`	Flag whether the $\mathcal{H}$-representation is reduced (1/0)
`Vertices`	Extreme points ($\mathcal{V}$-representation) of the polytope (may be empty)

Tab. 11.1: Data stored in the`polytope` object.

Fields of the polytope structure are summarized it Table 11.2.

In order to access the $\mathcal{H}$-representation (matrices `H` and `K`), one has to use the command `double` as follows:

```
[H,K] = double(P)
```

to store matrices `H` and `K` individually, or alternatively:

```
HK = double(P)
```

which returns a matrix $HK = [H\ K]$.

Center and radius of Chebyshev's ball can be obtined by:

```
[xCheb, RCheb] = chebyball(P)
```

If polytope is in normalized representation, call to

```
flag = isnormal(P)
```

will return 1, 0 otherwise.

The command

```
flag = isminrep(P)
```

return 1 if polytope `P` is in minimal representation (i.e. the $\mathcal{H}$-representation contains no redundant hyperplanes), 0 otherwise.

The polytope is bounded if

```
flag = isbounded(P)
```

returns 1 as the output.

Dimension of a polytope can be obtained by

```
d = dimension(P)
```

and

```
nc = nconstr(P)
```

will return number of constraints (i.e. number of half-spaces) defining the given polytope `P`.

Vertex representation of a polytope can be obtained by:

```
V = extreme(P)
```

which returns vertices stored row-vise in the matrix V. As enumeration of extreme vertices is an expensive operation, the computed vertices can be stored in the polytope object. To do it, we always recommend to call the function as follows:

```
[V,R,P] = extreme(P)
```

which returns extreme points V, extreme rays R and the update polytope object with vertices stored inside (P).

To check if a given point x lies in a polytope P, use the following call:

```
flag = isinside(P,x)
```

The function returns 1 if $x \in P$, 0 otherwise. If P is a polyarray (see Section 11.3 for more details about polyarrays), the function call can be extended to provide additional information:

```
[flag, inwhich, closest] = isinside(P,x)
```

which returns a 1/0 flag which denotes if the given point x belongs to any polytope of a polyarray P. If the given point lies in more than one polytope, inwhich contains indexes of the regions which contain x. If there is no such region, index of a region which is closest to the given point x is returned in closest.

Functions mentioned in this chapter are summarized in Table 11.2.

`P=polytope(H,K)`	Constructor for creating the polytope $P = \{x \in \mathbb{R}^n \mid Hx \leq K\}$.
`P=polytope(V)`	Constructor for creating the polytope out of extreme points
`double(P)`	Access internal data of the polytope, e.g. `[H,K]=double(P)`.
`display(P)`	Displays details about the polytope P.
`nx=dimension(P)`	Returns dimension of a given polytope P
`nc=nconstr(P)`	For a polytope $P = \{x \in \mathbb{R}^n \mid Hx \leq K\}$ returns number of constraints of the H matrix (i.e. number of rows).
`[ , ]`	Horizontal concatenation of polytopes into an array, e.g. `PA=[P1,P2,P3]`.
`( )`	Subscripting operator for polytope arrays, e.g. `PA(i)` returns the i-th polytope in `PA`.
`length(PA)`	Returns number of elements in a polytope array P_A.
`end`	In indexing functions returns the final element of an array.
`[c,r]=chebyball(P)`	Returns center c and radius r of the Chebychev ball inside P.
`V=extreme(P)`	Computes extreme points (vertices) of a polytope P.
`bool=isfulldim(P)`	Checks if polytope P is full dimensional.
`bool=isinside(P,x)`	Checks if $x \in P$. Works also for polytope arrays.

Tab. 11.2: Functions defined for class polytope.

11.3 Polytope arrays

polytope objects can be concatenated into arrays. Currently, only one-dimensional arrays are supported by $\mathbb{MPT}$ and it does not matter if the elements are stored row-wise or column-wise. Polytope array (or polyarray), is created using standard Matlab concatenation operators [,], e.g. A = [B C D].

It does not matter whether the concatenated elements are single polytopes or polyarrays. To illustrate this, assume we've defined polytopes `P1, P2, P3, P4, P5` and polyarrays `A = [P1 P2]` and `B = [P3 P4 P5]`. Then the following polyarrays `M` and `N` are equivalent:

```
M = [A B]
N = [P1 P2 P3 P4 P5]
```

Individual elements of a polyarray can be obtained using the standard referencing `(i)` operator, i.e.

```
P = M(2)
```

will return the second element of the polyarray `M` which is equal to `P2` in this case. More complicated expressions can be used for referencing:

```
Q = M([1,3:5])
```

will return a polyarray `Q` which contains first, third, fourth and fifth element of polyarray `M`.

If you want to remove some element from a polyarray, use the referencing command as follows:

```
M(2) = []
```

which will remove the second element from the polyarray `M`. Again, multiple indices can be specified, e.g.

```
M([1 3]) = []
```

will erase first and third element of the given polyarray.

Important: If some element of a polyarray is deleted, the remaining elements are shifted towards the start of the polyarray! This means that, assuming `N = [P1 P2 P3 P4 P5]`, after

```
N([1 3]) = []
```

the polyarray `N = [P2 P4 P5]` and the length of the array is 3. No empty positions in a polyarray are allowed! Similarly, empty polytopes are not being added to a polyarray.

A `polyarray` is still a polytope object, hence all functions which work on polytopes support also polyarrays. This is an important feature mainly in the geometric functions.

Length of a given `polyarray` is obtained by

```
l = length(N)
```

A polyarray can be flipped by the following command:

```
Nf = fliplr(N)
```

i.e. if `N = [P1 P2 P3 P4 P5]` then `Nf = [P5 P4 P3 P2 P1]`.

`P == Q`	Check if two polytopes are equal ($P = Q$).
`P ~= Q`	Check if two polytopes are not-equal ($P \neq Q$).
`P >= Q`	Check if $P \supseteq Q$.
`P <= Q`	Check if $P \subseteq Q$.
`P > Q`	Check if $P \supset Q$.
`P < Q`	Check if $P \subset Q$.
`P & Q`	Intersection of two polytopes, $P \cap Q$.
`P | Q`	Union of two polytopes, $P \cup Q$. If the union is convex, the polytope $P \cup Q$ is returned, otherwise the polyarray $[P\ Q]$ is returned.
`P + Q`	Minkowski sum, $P \oplus Q$.
`P - Q`	Pontryagin difference, $P \ominus Q$.
`P \ Q`	Set difference operator.
`B=bounding_box(P)`	Computes minimal hyper-rectangle containing a polytope P.
`E=envelope(P,Q)`	Computes envelope E of two polytopes P and Q.
`P=range(Q,A,f)`	Affine transformation of a polytope. $P = \{Ax + f \in \mathbb{R}^n \mid x \in Q\}$
`P=domain(Q,A,f)`	Compute polytope that is mapped to Q. $P = \{x \in \mathbb{R}^n \mid Ax + f \in Q\}$
`R=projection(P,dim)`	Orthogonal projection of `P` onto coordinates given in `dim`

Tab. 11.3: Computational geometry functions

11.4 Geometric operations on polytopes

The polytope library of MPT can efficiently perform many geometric manipulations on polytopes and polyarrays (non-convex unions of polytopes). Theoretical description of some basic operations has been already described in Section 3.1. List of computational geometry functions is provided in Table 11.3.

Except of `bounding_box`, all other functions are implemented to take polytopes and/or polyarrays as input arguments. We recommend to consult help files for respective functions for more details about extended function calls and other details.

The following examples show how to use some of the functionality described in Table 11.3:

Example 11.4.1:

```
>> P=polytope([eye(2);-eye(2)],[1 1 1 1]');      %Create Polytope P
>> [r,c]=chebyball(P)                            %Chebychev ball inside P
   r=[0 0]'
   c=1
>> W=polytope([eye(2);-eye(2)],0.1*[1 1 1 1]'); %Create Polytope W
>> DIF=P-W;                                      %Pontryagin difference P-W
>> ADD=P+W;                                      %Minkowski addition P+W
>> plot(ADD, P, DIF, W);                         %Plot polytope array
```

The resulting plot is depicted in Figure 11.1. When a `polytope` object is created, the constructor automatically normalizes its representation and removes all redundant constraints. Note that all elements of the polytope class are private and can only be accessed as described in the tables. Furthermore, all information on a polytope is stored in the internal polytope structure. In this way unnecessary repetitions of the computations during polytopic manipulations in the future can be avoided.

Example 11.4.2:

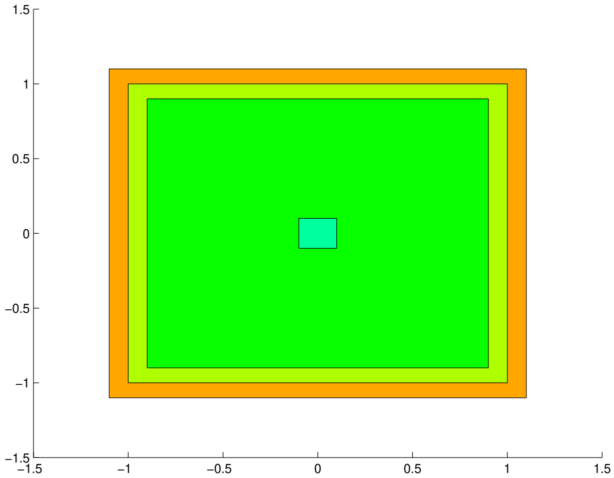
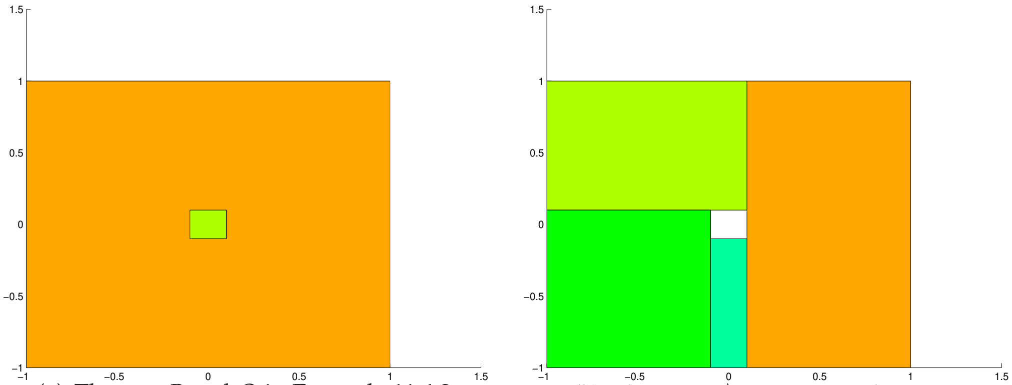


Fig. 11.1: The result of the `plot` call in Example 11.4.1

```
>> P=unitbox(2, 1);     %Create Polytope P as a box in 2D with sides of size 1
>> Q=unitbox(2, 0.1);   %Create Polytope Q as a box in 2D with sides of size 0.1
>> D=P\Q;               %Compute set difference between P and Q
>> length(D)            %D is a polytope array with 4 elements
     ans=4
>> U=D|Q;               %Compute union of D and Q
>> length(U)            %Union is again a polytope
     ans=1
>> U==P                 %Check if two polytopes are equal
     ans=1
```



(a) The sets P and Q in Example 11.4.2.

(b) The sets $P \setminus Q$ in Example 11.4.2.

The polytopes P and Q are depicted in Figure 11.4. The following will illustrate the `hull` and `extreme` functions.

Example 11.4.3:

```
>> P=polytope([eye(2);-eye(2)],[0 1 1 1]');      %Create Polytope P
>> Q=polytope([eye(2);-eye(2)],[1 1 0 1]');      %Create Polytope Q
>> VP=extreme(P);                              %Compute extreme vertices of P
```

```
>> VQ=extreme(Q);                          %Compute extreme vertices of P
>> D1=hull([P Q]);                         %Create convex Hull of P and Q
>> D2=hull([VP;VQ]);                       %Create convex Hull of vertices VP and VQ
>> D1==D2                                  %Check if hulls are equal
    ans=1
```

The `hull` function is overloaded such that it takes both elements of the `polytope` class as well as matrices of points as input arguments.

12

Acknowledgment

We would like to thank all contributors and those who report bugs. Specifically (in alphabetical order): Miroslav Baric, Alberto Bemporad, Francesco Borrelli, Frank J. Christophersen, Tobias Geyer, Eric Kerrigan, Adam Lagerberg, Arne Linder, Marco Lüthi, Saša V. Raković, Fabio Torrisi and Kari Unneland. A special thanks goes to Komei Fukuda (cdd), Johan Löfberg (Yalmip) and Colin Jones (ESP) for allowing us to include their respective packages in the distribution. Thanks to their help we are able to say that MPT truly is an 'unpack-and-use' toolbox.

Bibliography

[1] Baotić, M.: *An Efficient Algorithm for Multi-Parametric Quadratic Programming*. Technical Report AUT02-04, Automatic Control Laboratory, ETH Zurich, Switzerland, February 2002.

[2] Baotić, M., F. J. Christophersen and M. Morari: *Infinite Time Optimal Control of Hybrid Systems with a Linear Performance Index*. In *Proc. of the Conf. on Decision and Control, Maui, Hawaii, USA*, December 2003.

[3] Baotić, M., F.J. Christophersen and M. Morari: *A new Algorithm for Constrained Finite Time Optimal Control of Hybrid Systems with a Linear Performance Index*. In *European Control Conference*, Cambridge, UK, September 2003.

[4] Bemporad, A., F. Borrelli and M. Morari: *Explicit Solution of LP-Based Model Predictive Control*. In *Proc. 39th IEEE Conf. on Decision and Control*, Sydney, Australia, December 2000.

[5] Bemporad, A., F. Borrelli and M. Morari: *Optimal Controllers for Hybrid Systems: Stability and Piecewise Linear Explicit Form*. In *Proc. 39th IEEE Conf. on Decision and Control*, Sydney, Australia, December 2000.

[6] Bemporad, A., F. Borrelli and M. Morari: *Min-max Control of Constrained Uncertain Discrete-Time Linear Systems*. IEEE Trans. Automatic Control, 48(9):1600–1606, 2003.

[7] Bemporad, A., K. Fukuda and F.D. Torrisi: *Convexity Recognition of the Union of Polyhedra*. Computational Geometry, 18:141–154, April 2001.

[8] Bemporad, A., M. Morari, V. Dua and E.N. Pistikopoulos: *The Explicit Linear Quadratic Regulator for Constrained Systems*. Automatica, 38(1):3–20, January 2002.

[9] Borrelli, F.: *Constrained Optimal Control Of Linear And Hybrid Systems*, volume 290 of *Lecture Notes in Control and Information Sciences*. Springer, 2003.

[10] Borrelli, F., M. Baotić, A. Bemporad and M. Morari: *An Efficient Algorithm for Computing the State Feedback Optimal Control Law for Discrete Time Hybrid Systems*. In *Proc. 2003 American Control Conference*, Denver, Colorado, USA, June 2003.

[11] Ferrari-Trecate, G., F. A. Cuzzola, D. Mignone and M. Morari: *Analysis of discrete-time piecewise affine and hybrid systems*. Automatica, 38:2139–2146, 2002.

[12] Fukuda, K.: *Polyhedral computation FAQ*, 2000. On line document. Both html and ps versions available from `http://www.ifor.math.ethz.ch/staff/fukuda`.

[13] Grieder, P., F. Borrelli, F.D. Torrisi and M. Morari: *Computation of the Constrained Infinite Time Linear Quadratic Regulator*. In *Proc. 2003 American Control Conference*, Denver, Colorado, USA, June 2003.

[14] Grieder, P., M. Kvasnica, M. Baotić and M. Morari: *Low Complexity Control of Piecewise Affine Systems with Stability Guarantee*. In *American Control Conference, Boston, USA*, June 2004.

[15] Grieder, P. and M. Morari: *Complexity Reduction of Receding Horizon Control*. In *Proc. 42th IEEE Conf. on Decision and Control*, Maui, Hawaii, USA, December 2003.

[16] Grieder, P., P. Parillo and M. Morari: *Robust Receding Horizon Control - Analysis & Synthesis*. In *Proc. 42th IEEE Conf. on Decision and Control*, Maui, Hawaii, USA, December 2003.

[17] Heemels, W.P.M.H., B. De Schutter and A. Bemporad: *Equivalence of Hybrid Dynamical Models*. Automatica, 37(7):1085–1091, July 2001.

[18] Johannson, M. and A. Rantzer: *Computation of piece-wise quadratic Lyapunov functions for hybrid systems*. IEEE Trans. Automatic Control, 43(4):555–559, 1998.

[19] Kerrigan, E. C. and J. M. Maciejowski: *Robustly stable feedback min-max model predictive control*. In *Proc. 2003 American Control Conference*, Denver, Colorado, USA, June 2003.

[20] Kerrigan, E. C. and D. Q. Mayne: *Optimal control of constrained, piecewise affine systems with bounded disturbances*. In *Proc. 41st IEEE Conference on Decision and Control*, Las Vegas, Nevada, USA, December 2002.

[21] Löfberg, J.: *YALMIP : A Toolbox for Modeling and Optimization in MATLAB*. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. Available from `http://control.ee.ethz.ch/~joloef/yalmip.php`.

[22] Maciejowski, J.M.: *Predictive Control with Constraints*. Prentice Hall, 2002.

[23] Mayne, D. Q. and S. Raković: *Model predicitive control of constrained piecewise affine discrete-time systems*. Int. J. of Robust and Nonlinear Control, 13(3):261–279, April 2003.

[24] Mayne, D. Q., J.B. Rawlings, C.V. Rao and P.O.M. Scokaert: *Constrained model predictive control: Stability and Optimality*. Automatica, 36(6):789–814, June 2000.

[25] Rawlings, J.B. and K.R. Muske: *The stability of constrained receding-horizon control*. IEEE Trans. Automatic Control, 38:1512–1516, 1993.

[26] Skogestad, S. and I. Postlethwaite: *Multivariable Feedback Control*. John Wiley & Sons, 1996.

[27] The MathWorks, Inc.: *MATLAB Users Manual*. Natick, MA, US, 2003. `http://www.mathworks.com`.

[28] Tøndel, P., T.A. Johansen and A. Bemporad: *An Algorithm for Multi-Parametric Quadratic Programming and Explicit MPC Solutions*. In *Proc. 40th IEEE Conf. on Decision and Control*, Orlando, Florida, December 2001.

[29] Torrisi, F.D. and A. Bemporad: *HYSDEL — A Tool for Generating Computational Hybrid Models*. Technical Report AUT02-03, ETH Zurich, 2002. Submitted for publication on IEEE Trans. on Control Systems Technology.

[30] Ziegler, G. M.: *Lectures on Polytopes*. Springer, 1994.