Multi-Parametric Toolbox 3.0

Martin Herceg, Michal Kvasnica, Colin N. Jones, and Manfred Morari

***Abstract*— The Multi-Parametric Toolbox is a collection of algorithms for modeling, control, analysis, and deployment of constrained optimal controllers developed under Matlab. It features a powerful geometric library that extends the application of the toolbox beyond optimal control to various problems arising in computational geometry. The new version 3.0 is a complete rewrite of the original toolbox with a more flexible structure that offers faster integration of new algorithms. The numerical side of the toolbox has been improved by adding interfaces to state of the art solvers and by incorporation of a new parametric solver that relies on solving linear-complementarity problems. The toolbox provides algorithms for design and implementation of real-time model predictive controllers that have been extensively tested.**

I. INTRODUCTION

The Multi-Parametric Toolbox (MPT) is a software tool for Matlab [19] that aims at solving parametric optimization problems that arise in constrained optimal control. In particular, as the name of the toolbox suggests, its primal objective is to provide computationally efficient means for design and application of explicit model predictive control (MPC). Since the initial release in 2004 [14] there has been a significant progress in the development of the toolbox and the scope of the toolbox has widened to deal also with problems arising in computational geometry.

On the market there exist toolboxes that offer operations involved purely in computational geometry, i.e. GEOMETRY toolbox [3], CGLAB [23], and Ellipsoidal Toolbox [13]. Other toolboxes beside geometrical tools offer also algorithms for computing and implementation of control routines e.g. the Hybrid toolbox [1], MOBY-DIC toolbox [21], RACT toolbox [29], PnPMPC toolbox [25], and RoMulOC [22]. MPT is also one of the tools that combines computational geometry with control routines. Many of these toolboxes including MPT rely on YALMIP [17] which provides a high level language for modeling and formulating optimization problems.

The content of MPT can be divided into four modules:

- modeling of dynamical systems,
- MPC-based control synthesis,
- closed-loop analysis,
- deployment of MPC controllers to hardware.

Each part represents one stage in design and implementation of explicit MPC. The modeling module of MPT allows to describe discrete-time systems with either linear or hybrid dynamics. The latter can be directly imported from the HYSDEL environment [28]. The control module allows to formulate and solve constrained optimal control problems for both linear and hybrid systems. For a detailed overview of employed mathematical formulations the reader is referred to [2]. The analysis module provides methods for investigation of closed-loop behavior and performance. Moreover, it also features methods to reduce complexity of explicit MPC feedbacks. The deployment part allows to export control routines to the ANSI-C language, which can be subsequently downloaded to a target hardware implementation platform.

Compared to the previous release, the 3.0 version of MPT significantly improves capabilities of all four aforementioned modules. The main advances can be summarized as follows:

- Completely new installation procedure using a software manager.
- New optimization engines based on linear-complementarity problem solvers.
- Extended support for computational geometry.
- New flexible user interface based on object-oriented programming.
- Modular structure for easier integration of new algorithms.
- Extended support for real-time control.
- Improved numerical reliability based on extensive testing.
- Detailed documentation including examples and demos.

This paper describes the new features of MPT in detail and highlights the key properties that may be of interest to a broader control community.

II. Notations

To clarify the presentation, this section introduces the common notations to be referred throughout the text. Specifically, the notation is required to define the solutions to multi-parametric problems and is used in description of the polyhedral sets to understand the features of the geometric library.

Martin Herceg and Manfred Morari are with the Automatic Control Laboratory, ETH Zurich, Switzerland; {herceg,morari}@control.ee.ethz.ch.

Michal Kvasnica is with the Institute of Information Engineering, Automation, and Mathematics, STU Bratislava, Slovakia; michal.kvasnica@stuba.sk.

Colin N. Jones is with the Automatic Control Laboratory, EPF Lausanne, Switzerland; colin.jones@epfl.ch.

A. Set Description

Definition 2.1 (Convex set): A set $\mathcal{S} \subseteq R^n$ is convex if the line segment connecting any pair of points of $\mathcal{S}$ lies entirely in $\mathcal{S}$, i.e. if for any $s_1, s_2 \in \mathcal{S}$ and any α with $0 \leq \alpha \leq 1$, we have $\alpha s_1 + (1-\alpha)s_2 \in \mathcal{S}$.

Definition 2.2 (Set collection): $\mathcal{S}$ is called a set collection (union) in $\mathbb{R}^n$ if it is a collection of a finite number of n-dimensional sets $\mathcal{S}_i$, i.e., $\mathcal{S} := \bigcup_{i=1}^{N_\mathcal{S}} \mathcal{S}_i$, where $N_\mathcal{S} < \infty$.

Definition 2.3 (Piecewise function): The function $f : \mathcal{S} \mapsto \mathbb{R}^{n_f}$ is called a piecewise function if its domain is defined over a collection of a finite number of n-dimensional sets $\mathcal{S}_i$, and each set is associated with a particular vector field f_i, i.e., $f(x) := f_i(x)$ if $x \in \mathcal{S}_i$, where $\mathcal{S} := \bigcup_{i=1}^{N_\mathcal{S}} \mathcal{S}_i$ and $N_\mathcal{S} < \infty$.

Definition 2.4 (Polyhedron): A polyhedron is a convex set given as the intersection of a finite number of hyperplanes and half-spaces or as a convex combination of a finite number of vertices and rays.

Definition 2.5 (Polytope): A polytope is a bounded polyhedron.

Definition 2.6 (H-representation): The polyhedron $\mathcal{P}$ is formed by the intersection of m inequalities and m_e equalities, i.e.

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax \leq b,\ A_e x = b_e\} \tag{1}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $A_e \in \mathbb{R}^{m_e \times n}$, $b_e \in \mathbb{R}^{m_e}$ are the data representing the halfspaces and hyperplanes, respectively.

Definition 2.7 (V-representation): The polyhedron $\mathcal{P}$ is formed by a convex combination of n_v vertices and n_r rays, i.e.

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid x = \lambda^T V + \gamma^T R,\ \lambda, \gamma \geq 0,\ 1^T\lambda = 1\} \tag{2}$$

where $V \in \mathbb{R}^{n \times n_v}$, $R \in \mathbb{R}^{n \times n_r}$ represent vertices and rays, respectively.

III. Features of MPT 3.0

A. Installation and Updates

The new version of MPT is distributed in a modular structure that is operated by a *Toolbox Manager* available at `www.tbxmanager.com`. Toolbox Manager provides means for automatic installation, uninstallation and updates of Matlab toolboxes. The manager can be installed as per instructions on its web page.

MPT 3.0 is composed of several modules that are required to achieve the full functionality. The base package is referred to as *mpt* and the related documentation as *mptdoc* which can be installed by issuing

```
tbxmanager install mpt mptdoc
```

at the Matlab prompt. The other modules can be installed by pointing to the names of the submodules

```
tbxmanager install lcp hysdel cddmex clpmex
                glpkmex fourier sedumi yalmip
```

After installation of the submodules, the user can start using the software directly. If any module has been updated, the new versions can be obtained and installed with the help of

```
tbxmanager update
```

The Toolbox Manager thus provides a very simple approach to keep updated with any future releases of MPT, including its submodules.

B. Modular Structure

The MPT 3.0 comes with an extended structure that is based on submodules and object-oriented programming. The main motivation for this change was to achieve easier maintainability of the toolbox and to provide flexible structure for possible future enhancements. For instance, in the previous version of MPT there was a single object encompassing multiple algorithms. In the version 3.0, several new objects have been introduced that follow a hierarchy derived from object-oriented programming approach. Using this hierarchy it is possible to introduce new objects and methods to the existing framework with a minimal effort. The new class can be added by creating a new folder and by subclassing an existing object. The new object inherits properties and methods of the superclass and can be used to associate specific methods for tackling a particular problem. In the next sections some of the new objects will be presented and the main functionality will be explained.

C. Core Numerical Engines

Majority of the optimization problems involved in the computational geometry can be expressed as linear (LP) or quadratic problems (QP). To solve these problems effectively, MPT requires additional solvers that can be installed easily as submodules using the Toolbox Manager. Version 3.0 of MPT comes with new solvers that tackle both of these problems effectively. The new optimization engines are based on solvers for a linear complementarity problem (LCP) that represents a superclass for LP and QP. The advantage of representing and solving the optimization problems as LCPs is that a single solver covers all three scenarios and there is no need for multiple solvers that could potentially return different results. There are two new solvers implemented in MPT 3.0: LCP solver and parametric LCP solver. Both of these solvers will be reviewed next including their properties and implementation details.

1) LCP Solver: The linear-complementarity problem represents the class of optimization problems given as

$$\begin{aligned} & \text{find } w, z \\ \text{s.t.:} \quad & w - Mz = q && \text{(3a)} \\ & w^T z = 0 && \text{(3b)} \\ & w, z \geq 0 && \text{(3c)} \end{aligned}$$

where the problem data is given by a sufficient matrix $M \in \mathbb{R}^{n \times n}$ and vector $q \in \mathbb{R}^n$. The unknown variables are z and w that are coupled by the linear complementarity constraints (3b). LCP problems are well studied in the

literature [20] and several efficient methods for solving such problems have been proposed. One of the most successful approaches to solve LCP (3) is by employing the *lexicographic Lemke's algorithm* [20, Chap 2.]. This active set algorithm features a symbolic perturbation technique that ensures unique pivot step selection at each iteration, which prevents the method from internal cycling.

MPT 3.0 provides a C-code implementation of the lexicographic Lemke's algorithm, enriched by various techniques and methods to improve speed and numerical robustness of the method. In particular, the LU recursive factorization based on rank-one updates [26] has been incorporated to reduce computational time at each iteration. The LCP solver automatically performs scaling of the input data in case the problem is not well-conditioned. In addition, the LCP solver executes re-factorization of the basis if the lexicographic perturbation did not properly identify the unique pivot. The package is linked to BLAS and LAPACK numerical routines that provide state-of-the art algorithms for implementation of linear algebra. With all these features implemented, the LCP solver should provide a numerically reliable engine for resolving also difficult degenerate cases that may easily arise in formulations of MPC problems. The LCP solver is seamlessly integrated in MPT, but can also be installed separately via the Toolbox manager.

2) Parametric LCP Solver: The parametric LCP (PLCP) solver aims at solving the following class of problems

$$\begin{aligned} & \text{find } w, z \\ \text{s.t.:} \quad & w - Mz = q + Q\theta, && \text{(4a)} \\ & w^T z = 0, && \text{(4b)} \\ & w, z \geq 0, && \text{(4c)} \\ & \theta \in \Theta, && \text{(4d)} \end{aligned}$$

which differs from (3) by the addition of the parametric term $Q\theta$ in (4a) with $Q \in \mathbb{R}^{n\times d}$. Here, $\theta \in \mathbb{R}^d$ represents a free parameter, which is assumed to be bounded by (4d), where $\Theta \subset \mathbb{R}^d$ is a polytope. The problem data are furthermore given by a sufficient matrix $M \in \mathbb{R}^{n\times n}$ and the vector $q \in \mathbb{R}^n$.

MPT 3.0 implements the algorithm of [12] to solve PLCP (4) that employs the lexicographic perturbations to improve robustness. In the initial phase of the algorithm, the PLCP (4) is solved as an LCP for a particular value of the parameter $\theta \in \Theta$. This results in a starting basis for exploration of the parametric space. In subsequent steps the algorithm proceeds by lexicographic pivot steps that identify the solution for the remaining space of parameters. The solution is given by an optimal pair of z^*, w^* that forms a piecewise affine (PWA) function of the parameters θ:

$$\begin{pmatrix} w^* \\ z^* \end{pmatrix} = F_i\theta + g_i \text{ if } \theta \in \mathcal{P}_i. \quad (5)$$

Here, F_i, g_i define the i-th local affine function which determines values of w^* and z^* for any θ that resides in the i-th polytope $\mathcal{P}_i$.

The PLCP formulation (4a) naturally entails parametric linear programming (PLP) and parametric quadratic programming (PQP) as special cases. In particular, consider a formulation of the form

$$\begin{aligned} \min_x \quad & \frac{1}{2}x^T Hx + (F\theta + f)^T x && \text{(6a)} \\ \text{s.t.:} \quad & A_e x = b_e + E\theta && \text{(6b)} \\ & Ax \leq b + B\theta && \text{(6c)} \\ & \theta \in \Theta && \text{(6d)} \end{aligned}$$

which consists of a parametrized objective function (6a) with problem data $H \in \mathbb{R}^{n\times n}$, $H \succeq 0$, $F \in \mathbb{R}^{n\times d}$, $f \in \mathbb{R}^n$, and parametrized polyhedron (6b)-(6c) where $E \in \mathbb{R}^{m_e\times d}$, $B \in \mathbb{R}^{m\times d}$. The formulation (6) can be converted into the PLCP setup (4) using suitable affine transformations between variables x, w, z, and θ. Since the process can be tedious for a human, MPT 3.0 provides automatic routines that convert PQP/PLP setups into the PLCP form.

There are few advantages of solving PLP/PQP (6) as PLCP (4). Firstly, a single method is used to tackle all three classes of problems that prevents from encoding inconsistencies that may be eventually caused by different algorithms and different tolerance settings. This was one of the problems in the previous version where there were multiple versions for multi-parametric LP/QP solvers. Secondly, according to [12], the PLCP approach is numerically robust and superior in efficiency to other methods. Furthermore, the PLCP approach can handle PLP/PQP problems where the parameters appear linearly in the cost function and in the right hand side of constraints and therefore is applicable to solve wider classes of practical problems.

3) Interfaces to External Solvers: Besides the new LCP solvers, MPT 3.0 provides interfaces to external state-of-the-art solvers. Supported solvers include, but are not limited to, CDD [7], GLPK [18], CLP [10], QPOASES [6], QPSPLINE [16], SeDuMi [24], GUROBI [9], and CPLEX [4]. With the exception of the latter two, all other solvers are provided under an open-source license and can easily be installed using the Toolbox manager.

It is worth noting that MPT 3.0 relies heavily on CDD solver for performing many tasks related to computational geometry. In particular, facet and vertex enumeration for convex polyhedra and polytopes, as well as elimination of redundant constraints, are delegated to CDD. For more information, the interested reader is referred to [8]. In addition, MPT 3.0 also requires a freely-available *Fourier* solver for computing projections of polyhedra and polytopes.

4) Interface for General Optimization Problems: MPT 3.0 provides a unified gateway for formulating LP/QP/LCP (and parametric versions thereof). The

gateway is represented by the `Opt` class. Using this class it is possible to formulate the optimization problem and to pass the data to any supported solver that has been installed. The general syntax is as follows:

```
problem = Opt('H',H,'f',f,'A',A,'b',b)
```

which accepts the problem data as matrices and vectors of the form (6) or (4). At the time of creation of the object, the type of the optimization problem is recognized and the appropriate solver from the list of available solvers is associated with the problem. If the problem is formulated as an LP/QP/PLP/PQP, the transformation to LCP/PLCP can be invoking the `qp2lcp` method as follows:

```
problem.qp2lcp()
```

In general, the optimization problem can be solved by calling the `solve` method, i.e.,

```
solution = problem.solve()
```

and the output is returned in a corresponding form for non-parametric and parametric solvers.

MPT 3.0 also allows to import parametric optimization problems defined using the YALMIP environment [17]. This is achieved by creating an instance of the `Opt` class as follows

```
problem = Opt(constraints, objective, theta, x)
```

Here, `constraints` and `objective` define, respectively, constraints and the cost function of a particular optimization problem, and `theta` and `x` denote YALMIP's variables used to define the problem. New versions of YALMIP also directly interact with MPT 3.0's PLCP solvers to directly solve parametric optimization setups via the `solvemp` command of YALMIP. For further details the user is referred to documentation of YALMIP.

D. Enhancements in the Geometric Library

The geometric library is a vital part of MPT since it provides basic building blocks for solving parametric optimization problems that arise in explicit MPC. The increasing interest in using features of the geometric library has motivated the development of new supported sets and related operations. In this section the enhancements in the geometric library are reviewed.

1) Polyhedral Library: Polyhedra and polytopes are represented in MPT 3.0 as instances of the `Polyhedron` class. The half-space representation of a polyhedron as in Definition 2.6 is created by calling the class constructor as follows:

```
P = Polyhedron('A',A,'b',b,'Ae',Ae,'be',be)
```

where `A`, `b` specify the inequalities $Ax \leq b$, and `Ae` with `be` define the equalities $A_e x = b_e$. If the polyhedron has no equality constraints, then a shorter version of the constructor can be used:

```
P = Polyhedron(A, b)
```

V-representation of a polyhedron as in (2) can be created by calling

```
P = Polyhedron('V',V,'R',R)
```

where `V` are the vertices (stored row-wise), and `R` specifies the rays. If no rays are present, the shorter syntax

```
P = Polyhedron(V)
```

can be used.

The main improvement in the geometric library comparing to previous version of MPT, is that the polyhedral sets can be constructed not just as bounded polytopes but also as general polyhedra according to Def. 2.6 and Def. 2.7. With this feature the geometric library in MPT 3.0 seamlessly supports unbounded and lower-dimensional polyhedra, as illustrated in Fig. 1. Another point worth mentioning is that, unlike the previous versions, MPT 3.0 *does not* automatically convert between H- and V-representations, neither does it eliminate redundant constraints. The H-to-V and V-to-H conversions, as well as elimination of redundant data, have to be manually requested by calling

```
P.minHRep()
```

to compute the minimal H-representation, or by

```
P.minVRep()
```

for obtaining a minimal V-representation of a polyhedron.

Once the polyhedron is constructed using the `Polyhedron` constructor, the user can directly access data of the (irredundant) H- and V-representations by accessing the `A`, `b`, `Ae`, `be`, `V`, and `R` fields. For instance:

```
vertices = P.V
```

will return vertices of the polyhedron, regardless of whether the object `P` was originally constructed as a V-polyhedron or from a half-space representation. Similarly, to access the H-representation, use

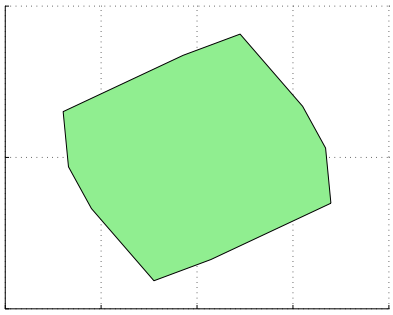
```
A = P.A, b = P.b, Ae = P.Ae, be = P.be
```

The new geometric library in MPT 3.0 implements various methods that operate on instances of the `Polyhedron` class. The new feature is that these methods apply also to lower-dimensional and unbounded sets. As an illustrative example, consider two polyhedra $\mathcal{P}$ and $\mathcal{Q}$. Then their Minkowski sum is given by $\mathcal{P} + \mathcal{Q} = \{x + z \mid x \in \mathcal{P},\ z \in \mathcal{Q}\}$. Such an operation can be straightforwardly performed in MPT 3.0 using the overloaded `+` (plus) operator:

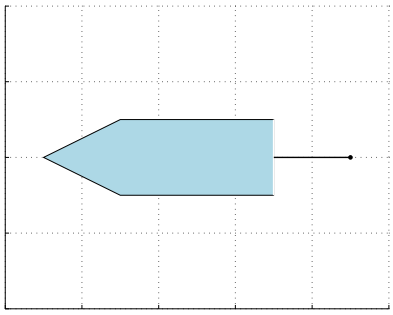
```
S = P + Q
```

As an another example, consider the subset test check $\mathcal{P} \subseteq \mathcal{Q}$. This can be achieved by using the overloaded `<=` operator as follows

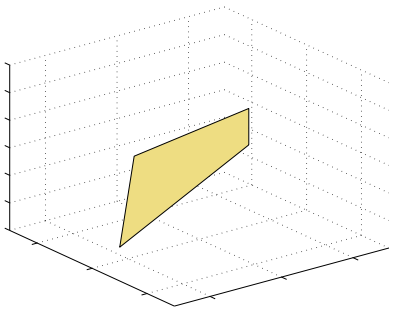
```
issubset = P <= Q
```



(a) Bounded polyhedron.



(b) Unbounded polyhedron.



(c) Lower-dimensional polytope.

Fig. 1. Extended polyhedral library supports bounded, unbounded and low-dimensional polyhedra.

Additional methods are summarized in Table I.

Another new feature of the geometric library is simplified creation of functions defined over polyhedra by allowing function handles to be associated to instances of the `Polyhedron` class. As an example, consider a 1D polytope $\mathcal{P} = \{x \in \mathbb{R} \mid -1 \leq x \leq 1\}$, over which we want to define the function $f(x) = \sin(x) + 2$. In MPT 3.0 this can be achieved first by creating a `Polyhedron` object and a `Function` object followed by attaching the function:

```
P = Polyhedron('lb', -1, 'ub', 1)
F = Function(@(x) sin(x)+2)
P.addFunction(F, 'f1')
```

Here, the string `'f1'` denotes the name of the function. Multiple functions can be associated to each polyhedron

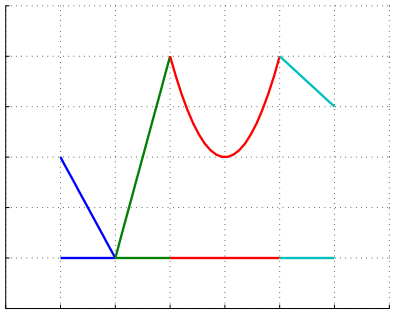


Fig. 2. Depiction of piecewise function consisting of three affine parts and one quadratic part.

objects. Then one can either plot the function by calling

```
P.fplot('f1')
```

or evaluate the function at a particular point z by

```
value = P.feval(z, 'f1')
```

If the polyhedron only has a single function associated to it, the second input argument can be omitted.

MPT 3.0 also provides a new `PolyUnion` class that represents unions of polyhedra of identical dimensions. Purpose of this class is to capture geometric properties of such unions, such as boundedness, connectivity, overlaps, convexity, and full-dimensionality. Subsequent computation can then benefit from these stored properties to reduce computational time. Unions of polyhedra can be created as follows:

```
U = PolyUnion([P1 P2 P3 P4])
```

where `P1`, `P2`, `P3`, `P4` are `Polyhedron` objects that form the union. Methods that operate on such unions are summarized in Table II. If each element of the union has a function associated to it, the `PolyUnion` object represents a piecewise function over polyhedra, cf. Def. 2.3. A plot of such a function, obtained by calling

```
U.fplot()
```

is shown in Fig. 2. The function value at a point z can be obtained by

```
value = U.feval(z, 'f1')
```

Here, `'f1'` denotes name of the function that should be evaluated. The syntax for function evaluation is the same as for the `Polyhedron` object, i.e. if the union only has a single function associated to it, the second input argument can be omitted.

Using this feature of the geometric library it is possible assign any type functions to a set to form piecewise functions. For instance, Tab. III lists methods that operate over PWA functions. These methods are used in the analysis module of MPT to postprocess explicit solution that is obtained in the hybrid MPC design.

2) General Convex Sets: The geometric library in MPT 3.0 is not restricted to polyhedra. It allows to

TABLE I

Implemented methods that support geometric operations with single polyhedra.

`affineHull`, `affineMap`, `invAffineMap`, `*`	Computation of affine hulls, affine maps, and inverse affine maps.
`chebyCenter`, `interiorPoint`	Computations of Chebyshev centre and arbitrary interior point.
`contains`, `==`, `~=`, `<=`, `>=`	Set containment tests.
`distance`	Compute distance of a polyhedron from a point or set
`extreme`	Computations of vertices in a given direction
`H`, `V`	Facet and vertex enumeration involving redundancy elimination.
`grid`, `meshGrid`	Generate points inside the polyhedron
`intersect`, `&`	Intersections of polyhedra
`isAdjacent`, `isBounded`, `isEmptySet`, `isFullDim`	Checking properties of polyhedra.
`minus`, `-`	Pontryagin difference.
`mldivide`, `\`	Set difference.
`plus`, `+`	Minkowski summations.
`project`, `projection`	Projections.
`shoot`	Ray maximization in the given direction.
`support`	Compute support of the set in the given direction.
`slice`	Cuts.
`triangulate`	Triangular decomposition.
`volume`	Volume of a polyhedron.

TABLE II

Implemented methods that support geometric operations with unions of polyhedra.

`contains`, `==`, `~=`, `<=`, `>=`	Set containment tests.
`convexHull`	Computation of a convex hull.
`isBounded`, `isConnected`, `isConvex`, `isFullDim`, `isOverlapping`	Checking properties of the union.
`join`, `merge`, `reduce`	Methods for merging of polyhedra.
`outerApprox`	Outer approximation of the union.
`plus`, `minus`	Minkowski summation and Pontryagin difference for unions of polyhedra.

define and process arbitrary convex set defined using YALMIP. Such sets are represented by the `YSet` class, whose constructor can be used as follows:

```
x = sdpvar(2, 1)
disc = YSet(x, [ x(1)^2 + x(2)^2 <= 1 ])
```

Here, the first line defines a YALMIP variable `x` as a real 2×1 vector. The second line then defines a disc centered at the origin with radius of 1. Such sets can then be processed by applying one of the methods listed in Table IV.

Note that only convex sets can be imported from YALMIP to form objects of `YSet` class. This is not a limitation, rather an interface for accessing the methods that operate over convex sets in geometric library of MPT.

TABLE III

Implemented methods that support operations with PWA functions.

`addFunction`, `removeFunction`	Adding and removing of function handles to a set.
`min`, `max`	Computing minimum, maximum of overlapping PWA function.
`fplot`	Plotting of functions.

TABLE IV

Implemented methods that support operations with convex sets.

`isBounded`, `isEmptySet`	Check properties of the set.
`contains`, `==`, `~=`, `<=`, `>=`	Set containment tests.
`distance`	Compute distance between the set and a point.
`extreme`	Computations of vertices in a given direction
`grid`	Generate points inside the set
`outerApprox`	Outer approximation of the set.
`project`	Projection of a point onto a set.
`separate`	Separating hyperplane between a point and a set.
`shoot`	Ray maximization in a given direction.
`support`	Compute support of the set in a given direction.

E. MPC-Based Control Design

MPT 3.0 allows to formulate and solve model predictive control problems for discrete-time linear and hybrid prediction models. The control synthesis is split into two parts. First, the user specifies the prediction model either as a linear time invariant system, as a piecewise affine system, or as a Mixed Logical Dynamical (MLD) system. Subsequently, the model, along with constraints and specifications of the objective function, are passed to the control module which converts them into a suitable mathematical description of the optimal control problem.

1) Modeling of Dynamical Systems: MPC synthesis for linear systems in MPT 3.0 assumes that the prediction model takes the form

$$x(t+\Delta) = Ax(t) + Bu(t) + f, \tag{7a}$$

$$y(t) = Cx(t) + Du(t) + g, \tag{7b}$$

where $x(t)$ is the state vector at time instant t, $x(t+\Delta)$ is the successor state at time $t+\Delta$ with Δ denoting the sampling time, $u(t)$ is the vector of control inputs, and $y(t)$ denotes the vector of outputs. Such systems are represented in MPT 3.0 as instances of the `LTISystem` class. A general way to create such systems is to call the constructor as follows:

```
sys = LTISystem('A', A, 'B', B, 'C', C,
        'D', D, 'f', f, 'g', g, 'Ts', Ts)
```

Note that all but the `A` parameters can be omitted (if the sampling time is not provided, MPT assumes $\Delta = 1$). Hence a quick way to specify an LTI system described

by the state-update equation $x(t+1) = Ax(t) + Bu(t)$ is to use

```
sys = LTISystem('A', A, 'B', B)
```

Such a syntax conveniently allows to specify autonomous systems as well. For example, the autonomous system $x(t+1) = Ax(t) + f$ can be created by

```
sys = LTISystem('A', A, 'f', f)
```

Another option is to specify the prediction model as a piecewise affine system of the form

$$x(t+\Delta) = \begin{cases} A_1 x(t) + B_1 u(t) + f_1 & \text{if } \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} \in \mathcal{P}_i, \\ \vdots & \\ A_L x(t) + B_L u(t) + f_L & \text{if } \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} \in \mathcal{P}_L. \end{cases} \tag{8}$$

Such systems are composed of L local affine models whose parameters (matrices A, B, f) change according to which polyhedron $\mathcal{P}_i$ contains the state-input vector. In MPT 3.0, such piecewise affine systems are defined by the `PWASystem` constructor, which takes as the input an array of local affine systems, each being an instance of the `LTISystem` class. As an example, consider the following PWA system:

$$x(t+1) = \begin{cases} 0.6x(t) + 1.2u(t) \text{ if } x(t) \geq 0, \\ -0.3x(t) + 0.9u(t) \text{ if } x(t) \leq 0. \end{cases} \tag{9}$$

First, the user has to specify the two local affine models by

```
local_1 = LTISystem('A', 0.6, 'B', 1.2)
local_2 = LTISystem('A', -0.3, 'B', 0.9)
```

Subsequently, the two local models are assigned to respective regions of validity. We have $\mathcal{P}_1 = \{x \mid x \geq 0\}$ for the first local model and $\mathcal{P}_2 = \{x \mid x \leq 0\}$ for the second. Such polyhedra are specified by

```
P1 = Polyhedron('lb', 0)
P2 = Polyhedron('ub', 0)
```

Finally, the polyhedra are attached to local models using the `setDomain()` method:

```
local_1.setDomain('x', P1)
local_2.setDomain('x', P2)
```

The `'x'` parameter specifies that the regions of validity should only be defined in the state space, as opposed to the general formulation which assumes state-input regions of validity. With the local models at hand, the overall description of the PWA model is obtained by

```
pwasys = PWASystem([local_1, local_2])
```

MPT 3.0 also allows to define mixed-logical dynamical systems of the form

$$\begin{aligned} x(t+\Delta) &= Ax(t) + B_1 u(t) + B_2 \delta(t) + B_3 z(t) + B_5, \\ y(t) &= Cx(t) + D_1 u(t) + D_2 \delta(t) + D_3 z(t) + D_5, \\ E_2 \delta(t) + E_3 z(t) &\leq E_4 x(t) + E_1 u(t) + E_5, \end{aligned}$$

where x is the state vector, whose components can either be real or binary, u is a mixed real-binary control vector, y a mixed real-binary vector of outputs, δ is a vector of auxiliary binary variables, and z denotes the vector of auxiliary real variables. The easiest way to define such systems is to use the HYSDEL modeling language. Once the model is specified using the HYSDEL syntax, it can be imported into MPT 3.0 by

```
sys = MLDSystem('model.hys');
```

Here, `model.hys` defines the name of the file which contains the system's description. Version 3.0 of MPT allows MLD models to be imported from HYSDEL; other features of HYSDEL are not yet incorporated.

2) Control Interface: The basic type of an optimal control problem assumed in MPT 3.0 is formulated the following form:

$$\min \sum_{k=0}^{N-1} (\|Q_x x_k\|_p + \|Q_u u_k\|_p) \tag{10a}$$

$$\text{s.t. } x_{k+1} = f(x_k, u_k), \tag{10b}$$

$$\underline{u} \leq u_k \leq \overline{u}, \tag{10c}$$

$$\underline{x} \leq x_k \leq \overline{x}, \tag{10d}$$

where x_k and u_k denote, respectively, prediction of states and inputs at the k-th step of the prediction horizon N, $f(\cdot,\cdot)$ is the prediction equation, $\underline{x}$, $\overline{x}$ are lower/upper limits on the states, and $\underline{u}$, $\overline{u}$ represent limits of the control authority. If $p \in \{1, \infty\}$ in (10a), then $\|\cdot\|_{\{1,\infty\|}$ denotes the standard vector 1- or ∞-norm. If $p = 2$, then $\|Q_x x_k\|_2 = x_k^T Q_x x_k$ is assumed.

MPT 3.0 allows to use LTI, PWA or MLD systems as prediction models in (10b). As an example, consider the LTI prediction model $x(t+1) = 0.6x(t) + u(t)$, along with constraints $-5 \leq x \leq 5$ and $-1 \leq u \leq 1$. To specify the MPC problem, the user can proceed as follows:

```
model = LTISystem('A', 0.6, 'B', 1)
controller = MPCController(model, N)
```

where `N` represents the prediction horizon. Now we can fine-tune the optimal control problem setup by modifying its properties. First, specify constraints:

```
controller.model.x.min = -5
controller.model.x.max = 5
controller.model.u.min = -1
controller.model.u.max = 1
```

Then we indicate that the objective function (10a) should use quadratic terms with $Q_x = 10$ and $Q_u = 0.1$:

```
controller.model.x.penalty = Penalty(10, 2)
```

TABLE V
FILTERS FOR MODIFICATION OF MPC PROBLEMS.

`binary`	Indicates that a variable is to be treated as binary.
`block`	Move blocking.
`initialSet`	Set constraint on the initial condition.
`reference`	Specifies a non-zero reference for a particular variable.
`softMin`, `softMax`	Constraint softening.
`terminalPenalty`	Penalty on the final predicted value.
`terminalSet`	Set constraint on the final predicted value.
`setConstraint`	Specifying polyhedral constraints.

```
controller.model.u.penalty = Penalty(0.1, 2)
```

The first argument is the value of the penalty matrix, while the second indicates which value of p in (10a) should be used. With the controller object in hand, we can then solve the optimal control problem (10) for a particular value of the initial condition as follows:

```
u0 = controller.evaluate(x0)
```

This method will solve (10) numerically and return the first element of the predicted optimal control sequence. To obtain information about feasibility of (10) for a particular value of the initial condition, and to inspect open-loop optimal profiles of states and inputs, request additional output arguments:

```
[u0, feas, OL] = controller.evaluate(x0)
```

The basic optimal control problem formulation (10) can be extended and customized easily. To do so, MPT 3.0 provides a mechanism referred to as *filters*. Each filter adds a new property to (10) and allows the user to fine-tune it. As an example, consider adding a terminal set constraint $x_N \in \mathcal{T}$, where x_N is the final predicted state and $\mathcal{T}$ represents a polyhedron. Such a constraint can be added by

```
controller.model.x.with('terminalSet')
controller.model.x.terminalSet = T
```

Here, the first line enables a new property, while the second line specifies the terminal set itself. To remove a property, use the `without()` method:

```
controller.model.x.without('terminalSet')
```

As an another example, consider adding polyhedral constraints of the form $x_k \in \mathcal{X}$. This can be achieved by the `setConstraint` filter as follows:

```
controller.model.x.with('setConstraint')
controller.model.x.setConstraint = X
```

List of filters provided in MPT 3.0 is shown in Table V.

It should be emphasized that, unlike previous versions, MPT 3.0 by default assumes that the `MPCController` object represents an on-line optimization controller. This means that the value of the optimal control input is determined by numerically solving (10) for a given value of the initial condition. An explicit representation of the feedback law, i.e., the function

$$u_0^* = \kappa(x_0) = \begin{cases} F_1 x_0 + g_1 & \text{if } x_0 \in \mathcal{P}_1, \\ \vdots & \\ F_L x_0 + g_L & \text{if } x_0 \in \mathcal{P}_L \end{cases} \tag{11}$$

is only computed on-demand by calling the `toExplicit()` method:

```
explicit_solution = controller.toExplicit()
```

where `explicit_solution` is an instance of the `EMPCController` class. Such an object can be further processed in the same way as discussed above. For instance, the value of optimal control input for a particular value of the initial condition can be obtained by

```
u0 = explicit_solution.evaluate(x0)
```

The difference being that no on-line optimization is required this time. Instead, the optimal control input is determined directly from (11) using sequential search. Explicit MPC controllers can also be visually inspected. One option is to plot regions $\mathcal{P}_i$ that constitute the domain of the feedback law in (11) by

```
explicit_solution.partition.plot()
```

To plot the PWA function $\kappa(\cdot)$, call

```
explicit_solution.feedback.fplot()
```

F. Analysis of MPC Controllers

The analysis module of MPT 3.0 contains methods for investigation of closed-loop properties, such as robustness or liveness. For this purpose, MPT provides the `ClosedLoop` class, which represents closed-loop systems consisting of an MPC controller and a dynamical system (either in LTI, PWA or MLD forms as discussed above). An instance of such a class is created by

```
loop = ClosedLoop(controller, system)
```

Note that the system employed in the closed loop can be different from the prediction model based on which the controller was generated.

To perform a closed-loop simulation over a given number of steps, starting from the initial condition x_0, call

```
data = loop.simulate(x0, steps)
```

The output is a structure that contains simulated closed-loop profiles of states, inputs and outputs. The same syntax applies to on-line as well as to explicit MPC controllers.

G. Deployment of Explicit MPC to Hardware

The deployment module of MPT is comprised of a code generation tool that exports control routines to the low level C programming language. This autogenerated code can be consequently compiled on the target hardware and the control routines executed in real-time.

Due to presence of the LCP solver, MPT 3.0 can support tasks where online optimization is required. The applicability of LCP solver for real-time optimization is rather limited to small processes because LCP solver has been designed as general-purpose solver. For high-speed online solutions one should still resort to specially tailored solvers such as FIORDOS [11] and FORCES [5].

The explicit controllers are exported with the algorithms for evaluation of PWA functions, including the methods for effective evaluation using binary search trees [27]. The code generation is invoked using the `exportToC()` method.

To better understand the process of deploying MPC controllers, one can have a look at the demos that focus on implementation of controllers in Real-Time Workshop of Matlab. To invoke the demos, type

```
mpt_demo_deployment_explicitMPC
mpt_demo_deployment_onlineMPC
```

at the Matlab prompt to see the functionality of the deployment module.

H. Extensive Testing

The numerical reliability of the toolbox has been tested on a large set of problems including randomly generated cases, MPC problems designed from a library of linear models [15], and numerous benchmark examples. At the time of alpha release the test set contained 1439 problems from which 206 were for interfaced solvers, 995 for the polyhedral library, and 238 for remaining functions in the control interface. These number are not final because the test problems are continuously added in the development process. During the testing period it has been shown that MPT 3.0 provides superior performance to the previous version and numerous problematic cases have been tackled by introducing new algorithms.

IV. SUMMARY

MPT is a software tool that allows efficient formulation and solutions of optimization problems involved in multi-parametric programming and computational geometry. The toolbox features various enhancements compared to the previous version that make it more robust and numerically reliable. The new user interface provides easier access to implemented methods and is flexible enough for possible future extensions of the toolbox. The toolbox is freely available from

`http://control.ee.ethz.ch/~mpt`

website.

References

[1] A. Bemporad. The Hybrid Toolbox. `http://cse.lab.imtlucca.it/~bemporad/hybrid/toolbox`.

[2] F. Borrelli, A. Bemporad, and M. Morari. *Predictive control for linear and hybrid systems*. preprint, 2012. `http://www.mpc.berkeley.edu/mpc-course-material`.

[3] J. Burkardt. GEOMETRY toolbox. `http://people.sc.fsu.edu/~jburkardt/m_src/geometry/geometry.html`.

[4] IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[5] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control*, pages 668 – 674, Maui, HI, USA, December 2012. `forces.ethz.ch`.

[6] H.J. Ferrau. QPOASES. `http://www.kuleuven.be/optec/software/qpOASES`.

[7] K. Fukuda. CDD. `http://www.inf.ethz.ch/personal/fukudak/cdd_home/index.html`.

[8] K. Fukuda. Frequently asked questions in polyhedral computation, 2004. `ftp://ftp.ifor.math.ethz.ch/pub/fukuda/reports/polyfaq040618.pdf`.

[9] GUROBI OPTIMIZER. `http://www.gurobi.com`.

[10] J. Hall. CLP. `https://projects.coin-or.org/Clp`.

[11] C.N. Jones, A. Domahidi, M. Morari, S. Richter, F. Ullmann, and M.N. Zeilinger. Fast Predictive Control: Real-Time Computation and Certification. In *IFAC Conference on Nonlinear Model Predictive Control*, pages 94–98, Noordwijkerhout, the Netherlands, August 2012. `fiordos.ethz.ch`.

[12] C.N. Jones and M. Morari. Multiparametric Linear Complementarity Problems. In *IEEE Conference on Decision and Control*, December 2006.

[13] A.A. Kurzhanskiy and P. Varaiya. Ellipsoidal toolbox. Technical Report UCB/EECS-2006-46, EECS Department, University of California, Berkeley, May 2006. `http://code.google.com/p/ellipsoids`.

[14] M. Kvasnica, P. Grieder, M. Baotic, and M. Morari. Multi-Parametric Toolbox (MPT). In *HSCC (Hybrid Systems: Computation and Control)*, pages 448–462, March 2004. `http://control.ee.ethz.ch/~mpt`.

[15] F. Leibfritz. COMPleib: COnstraint Matrix-optimization Problem library - a collection of test examples for nonlinear semidefinite programs, control system design and related problems. Technical report, University of Trier, Department of Mathematics, 2004. `www.compleib.de`.

[16] W. Li and J.J. de Nijs. An implementation of the QPspline method for solving convex quadratic programming problems with simple bound constraints. *Journal of Mathematical Sciences*, 116(4):3387–3410, 2003.

[17] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. `http://users.isy.liu.se/johanl/yalmip`.

[18] A.O. Makhorin. GLPK – GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`.

[19] The Mathworks Inc. `www.mathworks.com`.

[20] K.G. Murty. *Linear complementarity, linear and nonlinear programming*. University of Michigan, Ann Arbor, Michigan, USA, 1997. `http://ioe.engin.umich.edu/people/fac/books/murty/linear_complementarity_webbook/`.

[21] A. Oliveri. MOBY-DIC toolbox. `http://www.ncas.dibe.unige.it/software/MOBY-DIC_Toolbox/`.

[22] D. Peaucelle. RoMulOC – robust multi-objective control toolbox. `http://projects.laas.fr/OLOCEP/romuloc/`.

[23] S. Pion and N. Meskini. CGLAB. `http://cglab.gforge.inria.fr/`.

[24] I. Polik. SeDuMi. `http://sedumi.ie.lehigh.edu/`.

[25] S. Riverso, A. Battocchio, and G. Ferrari-Trecate. PnPMPC toolbox, 2013. `http://sisdin.unipv.it/pnpmpc/pnpmpc.php`.

[26] M.A. Saunders. LUMOD – updating a dense square factorization L*C = U. `http://www.stanford.edu/group/SOL/software/lumod.html`.

[27] P. Tøndel, T.A. Johansen, and A. Bemporad. Evaluation of Piecewise Affine Control via Binary Search Tree. *Automatica*, 39(5):945–950, May 2003.

[28] F.D. Torrisi and A. Bemporad. HYSDEL – A tool for generating computational hybrid models for analysis and synthesis problems. *IEEE Transactions on Control Systems Technology*, 12:235–249, March 2004. `http://control.ee.ethz.ch/~hybrid/hysdel`.

[29] A. Tremba. RACT – randomized algorithms control toolbox. `http://ract.sourceforge.net`.