Abstract—The Multiparametric 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 Multiparametric Toolbox (MPT) is a software tool for Matlab [16] 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 [12] 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 exists toolboxes that offer operations involved purely in computational geometry, i.e. GEOMETRY toolbox [3], CGLAB [20], and Ellipsoidal Toolbox [11]. Other toolboxes offer application oriented computations that focus on implementation of specific control routines e.g. the Hybrid toolbox [1], MOBYDIC toolbox [18], RACT toolbox [25], and RoMulOC [19]. Many of these toolboxes rely on the YALMIP toolbox [14] 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 [24]. 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 platforms.

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

A. Set Description

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

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

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

**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.,

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

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.,

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

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 3.0 of MPT comes with a new numerical approach to solve optimization problems arising in parametric programming and computational geometry. The primal motivation for changing the underlying optimization solvers was to improve the robustness of the toolbox by implementing various techniques from perturbation theory. The new optimization engine is based on solving linear complementarity problems (LCP). The added benefit is that LCP formulations naturally encompass linear (LP) and quadratic programming (QP) problems, hence a single solver covers all these three scenarios. The next section shows the principles of formulating optimization problems as LCPs and discusses implementation details of the new LCP solver in MPT 3.0.

1) **Linear-Complementarity Problem Solver:** The linear-complementarity problem (LCP) represents the class of optimization problems given as

\[
\begin{align*}
\text{find } w, z \\
\text{s.t.: } w - Mz &= q \tag{3a} \\
w^T z &= 0 \tag{3b} \\
w, z &\geq 0 \tag{3c}
\end{align*}
\]

where the problem data is given by a sufficient real 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 (3a). LCP problems are well studied in the literature [17] and several efficient methods for solving such problems have been proposed. One of the most successful approaches to solve LCPs [3] is by employing the *lexicographic Lemke’s algorithm*. It is an active set method that ensures unique pivot step selection at each iteration, which prevents the algorithm from internal cycling. This advantage makes the method suitable for resolving degenerate cases that often arise in formulations of MPC problems.

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 [22] 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. The LCP solver is seamlessly integrated in MPT, but can also be installed separately via the Toolbox manager.

2) **Parametric LCP Solver:** MPT 3.0 also allows to solve parametric versions of LCP problems [4], defined
as
\[
\begin{align*}
\text{find } & \quad w, z \\ 
\text{s.t.:} & \quad w - Mz = q + Q\theta, \quad (4a) \\
& \quad w^Tz = 0, \quad (4b) \\
& \quad w, z \geq 0, \quad (4c) \\
& \quad \theta \in \Theta, \quad (4d)
\end{align*}
\]
which differs from (2) by the addition of the 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 (4b), where $\Theta \subset \mathbb{R}^d$ is a polytope. The problem data are furthermore given by a sufficient real matrix $M \in \mathbb{R}^{n \times n}$ and the vector $q \in \mathbb{R}^n$. The parametric LCP (PLCP) formulation (4a) naturally entails parametric linear programming (PLP) and parametric quadratic programming (PQP) as special cases. In particular, a PQP formulation of the form
\[
\begin{align*}
\min & \quad \frac{1}{2} x^THx + (C_\theta + c)^T x \\
\text{s.t.:} & \quad A_\theta x = b_\theta + E\theta \\
& \quad Ax \leq b + B\theta \\
& \quad \theta \in \Theta
\end{align*}
\]
can be converted into the PLCP setup (4). Moreover, if the cost function is linear, i.e., $H = 0$, formulation (5a) simplifies to a PLP problem. Conversion from (4) to (5) requires designing suitable affine transformations between variables $z$, $w$, $z$, and $\theta$. Since the process can be tedious for a human, MPT 3.0 provides automatic routines that convert PQP/PLP setups into the PLCP form.

MPT 3.0 provides a numerically-robust way of solving parametric LCP problems [4]. Increased numerical robustness is achieved by employing the concept of lexicographic perturbations [10]. 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 parametric solution for the remaining space. Optimal solutions $z^\ast$ and $w^\ast$ are then generated as piecewise affine (PWA) functions of the parameters $\theta$:
\[
\begin{pmatrix} w^\ast \\ z^\ast \end{pmatrix} = F_\theta \theta + g_\theta, \quad \text{if } \theta \in P_\theta.
\]
Here, $F_\theta$, $g_\theta$ define the $i$-th local affine function which determines values of $w^\ast$ and $z^\ast$ for any $\theta$ that resides in the $i$-th polytope $P_\theta$. The PWA function (6) is represented in MPT 3.0 using the new interface of the geometric library and can be employed for further analysis and postprocessing.

3) Interfaces to External Solvers: Besides the new LCP solver, MPT 3.0 provides interfaces to external state-of-the-art solvers. Supported solvers include, but are not limited to, CDD [6], GLPK [15], CLP [9], QPOASES [5], QPSPLINE [13], SeDuMi [21], Gurobi [8], 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 the 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 [7]. 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) problems. The gateway is represented by the `Opt` class. An instance of this class encapsulates the problem data, classifies the problem, and delegates it to a suitable solver. The general syntax is as follows:
\[
\begin{align*}
\text{problem} &= \text{Opt('H', 'H', 'f', 'f', 'A', 'A', 'b', 'b')} \\
\text{solution} &= \text{problem.solve()}
\end{align*}
\]
which accepts the problem data as matrices and vectors of the form (4) or (5). 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/PLQP, the transformation to LCP/PLCP can be invoking the `qp2lcp` method as follows:
\[
\begin{align*}
\text{problem.qp2lcp()} \\
\text{solution} &= \text{problem.solve()}
\end{align*}
\]
In general, the optimization problem can be solved by calling the `solve` method, i.e.,
\[
\text{solution} = \text{problem.solve()}
\]
and the output is returned in a corresponding form for non-parametric and parametric solvers.

MPT 3.0 also allows to import optimization problems defined using the YALMIP environment [14]. Here, an instance of the `Opt` class can be created by
\[
\begin{align*}
\text{problem} &= \text{Opt(constraints, objective, theta, x)} \\
\text{solution} &= \text{problem.solve()}
\end{align*}
\]
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.

C. 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:

\[ P = \text{Polyhedron}('A', A, 'b', b, 'Ae', Ae, 'be', be) \]

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

\[ P = \text{Polyhedron}(A, b) \]

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

\[ P = \text{Polyhedron}('V', V, 'R', R) \]

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

\[ P = \text{Polyhedron}(V) \]

can be used.

It is worth noting that 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

\[ P.\text{minHRep()} \]

to compute the minimal H-representation, or by

\[ P.\text{minVRep()} \]

for obtaining a minimal V-representation of a polyhedron.

Once the polyhedron is constructed using the \texttt{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:

\[ \text{vertices} = P.V \]

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 \texttt{Polyhedron} class. As an illustrative example, consider two polyhedra \( P \) and \( Q \). Then their Minkowski sum is given by \( P + Q = \{ x + z \mid x \in P, z \in Q \} \). 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 subsetness check \( P \subseteq Q \). This can be achieved by using the overloaded \( \leq \) operator as follows

\[ \text{issubset} = P \leq Q \]

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 \texttt{Polyhedron} class. As an example, consider a 1D
polytope \( P = \{ x \in \mathbb{R} \mid -1 \leq x \leq 1 \} \), 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 by

\[
P = \text{Polyhedron}('lb', -1, 'ub', 1)
\]

followed by attaching the function:

\[
P.addFunction(@(x) \sin(x)+2, 'f1')
\]

Here, the string ‘f1’ denotes the name of the function. Multiple functions can be associated to each polyhedron objects. Then one can either plot the function by calling

\[
P.fplot('f1')
\]

or evaluate the function at a particular point \( z \) by

\[
\text{value} = P.feval(z, 'f1')
\]

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, 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:

\[
U = \text{PolyUnion}([P1 P2 P3 P4])
\]

where \( P1, P2, P3, P4 \) are Polyhedron objects that form the union. Methods that operate on such unions are summarized in Table IV. If each element of the union has a function associated to it, the PolyUnion object represents a piecewise function over polyhedra, cf. Definition 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

\[
\text{value} = U.feval(z, 'f1')
\]

Here, ‘f1’ denotes name of the function that should be evaluated. If the union only has a single function associated to it, the second input argument can be omitted.

\[
\text{Table I}
\]

<table>
<thead>
<tr>
<th>Implemented methods that support geometric operations with single polyhedra.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chebyCenter, interiorPoint, contains, ==, &lt;=, &gt;=</td>
</tr>
<tr>
<td>distance</td>
</tr>
<tr>
<td>extreme</td>
</tr>
<tr>
<td>N, V</td>
</tr>
<tr>
<td>grid, meshGrid</td>
</tr>
<tr>
<td>intersect, &amp;</td>
</tr>
<tr>
<td>isAdjacent, isBounded, isEmptySet, isFullDim</td>
</tr>
<tr>
<td>minus, -</td>
</tr>
<tr>
<td>mldivide, \</td>
</tr>
<tr>
<td>plus, +</td>
</tr>
<tr>
<td>project, projection</td>
</tr>
<tr>
<td>shoot</td>
</tr>
<tr>
<td>support</td>
</tr>
<tr>
<td>slice</td>
</tr>
<tr>
<td>triangulate</td>
</tr>
<tr>
<td>volume</td>
</tr>
</tbody>
</table>

\[
\text{Table II}
\]

<table>
<thead>
<tr>
<th>Implemented methods that support geometric operations with unions of polyhedra.</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains, ==, &lt;=, &gt;=</td>
</tr>
<tr>
<td>convexHull</td>
</tr>
<tr>
<td>isBounded, isConnected, isConvex, isFullDim, isOverlapping</td>
</tr>
<tr>
<td>join, merge, reduce</td>
</tr>
<tr>
<td>outerApprox</td>
</tr>
<tr>
<td>plus, minus</td>
</tr>
</tbody>
</table>

2) General Convex Sets: The geometric library in MPT 3.0 is not restricted to polyhedra. It allows to define and process arbitrary convex set defined using YALMIP. Such sets are represented by the YSet class, whose constructor can be used as follows:

\[
x = sdpvar(2, 1)
disc = \text{YSet}(x, \ [ x(1)^2 + x(2)^2 <= 1 ])
\]

Here, the first line defines a YALMIP variable \( x \) as a real \( 2 \times 1 \) 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.

D. 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

\[ x(t + \Delta) = Ax(t) + Bu(t) + f, \]
\[ y(t) = Cx(t) + Du(t) + g, \]

where \( x(t) \) is the state vector at time instant \( t \), \( x(t + \Delta) \) is the successor state at time \( t + \Delta \) with \( \Delta \) 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:

```python
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

```python
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

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

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

\[ x(t + \Delta) = \begin{cases} A_1 x(t) + B_1 u(t) + f_1 & \text{if } x(t) \in P_1, \\ \vdots \\ A_L x(t) + B_L u(t) + f_L & \text{if } x(t) \in P_L. \end{cases} \]

Such systems are composed of \( L \) local affine models whose parameters (matrices \( A, B, f \)) change according to which polyhedron \( P_i \) 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:

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

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

```python
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 \( P_1 = \{ x \mid x \geq 0 \} \) for the first local model and \( P_2 = \{ x \mid x \leq 0 \} \) for the second. Such polyhedra are specified by

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

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

```python
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

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

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

\[
x(t + \Delta) = Ax(t) + B_1 u(t) + B_2 \delta(t) + B_3 z(t) + B_5,
\]
\[
y(t) = Cx(t) + D_1 u(t) + D_2 \delta(t) + D_3 z(t) + D_5,
\]
\[
E_2 \delta(t) + E_3 z(t) \leq E_4 x(t) + E_1 u(t) + E_5,
\]

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, \( \delta \) 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

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

Here, model.hys defines the name of the file which contains the system’s description.

2) Formulation of MPC Problems: The basic type of an optimal control problem assumed in MPT 3.0 takes the following form:

\[
\begin{align}
\min_{x} & \sum_{k=0}^{N-1} (\|Q_x x_k\|_p + \|Q_u u_k\|_p) \\
\text{s.t.} & \quad x_{k+1} = f(x_k, u_k), \\
& \quad u \leq u_k \leq u^u, \\
& \quad x \leq x_k \leq x^u,
\end{align}
\]

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, \(x, u^u, x^u\) are lower/upper limits on the states, and \(u, u^u\) represent limits of the control authority. If \(p \in \{1, \infty\}\) in (10a), then \(\| \cdot \|_{\{1, \infty\}}\) denotes the standard vector 1- or \(\infty\)-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, PW A 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:

```matlab
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:

```matlab
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:\n
```matlab
controller.model.x.penalty = Penalty(10, 2)
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:

```matlab
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 (11) for a particular value of the initial condition, and to inspect open-loop optimal profiles of states and inputs, request additional output arguments:

```matlab
[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 \(x_N \in T\), where \(x_N\) is the final predicted state and \(T\) represents a polyhedron. Such a constraint can be added by

```matlab
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:

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

As another example, consider adding polyhedral constraints of the form \(x_k \in X\). This can be achieved by the `setConstraint` filter as follows:

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

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

<table>
<thead>
<tr>
<th>binary</th>
<th>Indicates that a variable is to be treated as binary.</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>Move blocking.</td>
</tr>
<tr>
<td>initialSet</td>
<td>Set constraint on the initial condition.</td>
</tr>
<tr>
<td>reference</td>
<td>Specifies a non-zero reference for a particular variable.</td>
</tr>
<tr>
<td>softMin, softMax</td>
<td>Constraint softening.</td>
</tr>
<tr>
<td>terminalPenalty</td>
<td>Penalty on the final predicted value.</td>
</tr>
<tr>
<td>terminalSet</td>
<td>Set constraint on the final predicted value.</td>
</tr>
</tbody>
</table>

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

\[
u^*_0 = \kappa(x_0) = \begin{cases}
F_1 x_0 + g_1 & \text{if } x_0 \in \mathcal{P}_1, \\
\vdots & \\
F_L x_0 + g_L & \text{if } x_0 \in \mathcal{P}_L
\end{cases}
\]

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

```matlab
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
\[ u_0 = \text{explicit_solution.evaluate}(x_0) \]
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 \( P_i \) that constitute the domain of the feedback law in (11) by
\[ \text{explicit_solution.partition.plot()} \]
To plot the PWA function \( \kappa() \), call
\[ \text{explicit_solution.feedback.fplot()} \]

### E. 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
\[ \text{loop} = \text{ClosedLoop}(\text{controller}, \text{system}) \]
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
\[ \text{data} = \text{loop.simulate}(x_0, \text{steps}) \]
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.

### F. Deployment of Explicit MPC to Hardware
The deployment module of MPT 3.0 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.

MPT 3.0 supports the export of online controllers with the help of the LCP solver. Moreover, it can be employed in other tasks where online optimization is required.

The explicit controllers are exported with the algorithms for evaluation of PWA functions, including the methods for effective evaluation using binary search trees [23]. The code generation is invoked using the exportToC() method.

### IV. SUMMARY
MPT is a software tool that allows efficient formulation and solutions of optimization problems involved in multiparametric 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](http://control.ee.ethz.ch/~mpt).

## References


[9] I. Hall. CLP. [https://projects.coin-or.org/Clp](https://projects.coin-or.org/Clp)


