Design and Verification of Low-Complexity Explicit MPC Controllers in MPT3 (Extended version)

Michal Kvasnica, Juraj Holaza, Bálint Takács, and Deepak Ingole

Abstract— **This paper reviews the Multi-Parametric Toolbox 3, a new version of the easy-to-use software tool for design, verification, and implementation of optimization-based controllers. Specifically, we introduce advanced building blocks which allow to synthesize and analyze explicit representations of model predictive controllers of low real-time implementation complexity. These building blocks include, but are not limited to, integration of functions over polytopes, computational geometry operations, as well as procedures to analyze invariance and closed-loop stability. We show how to combine these building blocks as to create sophisticated algorithms which lead to well-performing, yet simple controllers which adhere to prescribed requirements. This paper is an extended version of our ECC'15 submission of the same title.**

I. Introduction

Since the seminal work of [4], the concept of *explicit model predictive control* (MPC) has garnered significant popularity among theoreticians and practitioners of MPC alike. Applications of explicit MPC range from power electronics, through control of mechanical and automotive systems, up to control of autonomous vehicles, see [12] for an overview.

The popularity of the framework is due to the fact that it allows to implement MPC in real time using low computational resources. This is achieved by pre-computing, off-line, the analytic solution to a given optimal control problem. For a large class of MPC setups, it can be shown [6] that the analytic (also called the explicit) solution takes the form of a piecewise affine (PWA) feedback law which consists of a set of critical regions and associated local affine feedback laws. Obtaining the optimal control action for a particular value of the state measurements then reduces to a mere function evaluation, which is typically faster and simpler compared to solving the optimal control problem numerically.

Explicit MPC solutions provide three main advantages. First, evaluation of the PWA feedback law is division-free, i.e., only additions and multiplications are required to obtain the optimal control action. This significantly simplifies implementation of MPC controllers in safety-critical applications. Second, the implementation of the PWA evaluation procedure is very simple and does not exceed 20 lines of C-code. This reduces the cost of certification. Finally, having the analytical solution allows to rigorously analyze properties of the closed-loop system, such as closed-loop stability or recursive feasibility for a whole range of initial conditions.

The main limitation which restrict applicability of explicit MPC in real-world scenarios is the complexity of such solutions. We distinguish two types of complexity. The runtime complexity relates to the off-line computational time required to construct the solution. Although synthesis of explicit MPC feedback laws was (and probably still is) believed to be restricted to problems of small state dimensionality (typically with less than 8 states), recent results [10, 7] suggest that this limitation can be alleviated at least for short prediction horizons.

However, even if the explicit solution can be constructed off-line, it is often prohibitively large to be used for real time control, especially when control hardware with limited storage is considered. To decrease the memory footprint to a desired level, it is therefore often necessary to simplify a given explicit MPC controller. Although a plethora of methods for achieving such a task has been proposed in the past 15 years (see, e.g., [14, 15] and references therein), few authors have gone as far as making software implementations of their respective methods publicly available for others to make use of.

The objective of this paper is to plug this hole and share with the public the software implementation of routines which enable synthesis, analysis and reduction of complexity of explicit MPC feedback laws. Specifically, we show how the Multi-Parametric Toolbox[1] version 3 (MPT3) can be used to set up MPC problems, calculate explicit MPC feedback laws, decrease their complexity, and analyze closed-loop systems. The proposed paper supplements our previous contribution [11] in which the toolbox was introduced in general terms. In the current paper we provide a much more detailed look into the internals of MPT3 and introduce advanced modeling principles which allow even complex MPC setups to be created with ease. We also provide a peek behind the curtains and expose to the reader basic building blocks which can be combined to create sophisticated algorithms.

The paper is composed of three main parts. First, Section II provides a detailed description of modeling principles and shows how MPC optimal control problems are set up in MPT3. Then we introduce methods which serve to reduce complexity of explicit MPC solutions in Section III where

All authors are with the Slovak university of Technology in Bratislava, Slovakia, Slovak University of Technology in Bratislava, Slovakia, {michal.kvasnica, juraj.holaza, balint.takacs, deepak.ingole}@stuba.sk. The authors gratefully acknowledge the financial contribution of the Scientific Grant Agency of the Slovak Republic under grant 1/0403/15 and the contribution of the Slovak Research and Development Agency under the project APVV 0551-11. The research leading to these results has received funding from the People Programme (Marie Curie Actions) of the European Unions Seventh Framework Programme (FP7/2007-2013) under REA grant agreement no 607957 (TEMPO).

This report is an extended version of our ECC'15 paper of the same title.

[1] Freely available from http://control.ee.ethz.ch/~mpt

we also discuss MPT3's computational geometry features required to implement such algorithms. Finally, Section IV discusses how the toolbox can be used to verify closed-loop stability and invariance as to attest that the controller exhibits desired safety properties.

II. MPC Design

MPC-based control synthesis in MPT3 is based on the following finite-time optimal control problem:

$$\min \ \ell_N(x_N) + \sum_{k=0}^{N-1} \ell_x(x_k) + \ell_y(y_k) + \ell_u(u_k) \tag{1a}$$

$$\text{s.t. } x_{k+1} = f(x_k, u_k), \ k = 0, \dots, N-1, \tag{1b}$$

$$y_k = g(x_k, u_k), \ k = 0, \dots, N-1, \tag{1c}$$

$$x_k \in \mathcal{X}, \ k = 0, \dots, N-1, \tag{1d}$$

$$y_k \in \mathcal{Y}, \ k = 0, \dots, N-1, \tag{1e}$$

$$u_k \in \mathcal{U}, \ k = 0, \dots, N-1, \tag{1f}$$

$$x_N \in \mathcal{T}, \tag{1g}$$

where x_k, y_k, u_k denote, respectively, predictions of system's states, outputs, and inputs over a finite prediction horizon N. The cost function employs a terminal state penalty $\ell_N(\cdot)$, and stage penalties $\ell_x(\cdot)$ for states, $\ell_y(\cdot)$ for outputs, and $\ell_u(\cdot)$ for inputs. Particular types of the penalty functions are discussed in Section II-C. The predictions are based on time-invariant state-update equation $f(\cdot,\cdot)$ and an output equation $g(\cdot,\cdot)$, as described in Section II-A. The toolbox also allows to specify piecewise affine (PWA) and mixed logical-dynamical (MLD) [3] dynamics. The reader is referred to [11] for details. Moreover, $\mathcal{X}$, $\mathcal{U}$, and $\mathcal{Y}$ are the constraint sets for states, inputs, and outputs, respectively. Finally, $\mathcal{T}$ is the terminal set. Available constraints are described in detail in Section II-B.

A. Linear Time-Invariant (LTI) Dynamics

Here we assume that the state-update and output equations are both linear in the form

$$x(t+\Delta) = Ax(t) + Bu(t), \tag{2a}$$

$$y(t) = Cx(t) + Du(t), \tag{2b}$$

where Δ is the sampling time. Such LTI systems are defined using the `LTISystem` constructor, whose general form is

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

where `A`, `B`, `C`, `D` are matrices of compatible dimensions, and `Ts` specifies the required sampling period. It is worth noting that the constructor also allows to omit the output equation, which is achieved by

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

Autonomous LTI systems can be defined as well by omitting the `'B'` parameter:

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

We note that if the sampling time is not provided `Ts=1` is assumed. Finally, MPT3 also allows to define state-update and output equations with constant terms, i.e., $x(t+\Delta) = Ax(t) + Bu(t) + f$ and $y(t) = Cx(t) + Du(t) + g$. As an example, consider an autonomous affine system $x(t+\Delta) = Ax(t) + f$. This system can be defined by

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

Once the LTI system is specified, its properties can be accessed using the "dot" syntax, e.g. `sys.A` returns the A matrix in (2a).

B. Constraints

MPT3 allows to specify various types of constraints for states, inputs, and outputs in (1), regardless of the type of state-update and output equations. The basic type of constraints is represented by lower/upper bounds on corresponding signals. This is achieved by setting the `sys.x.min` and `sys.x.max` properties (or `sys.u.min`, `sys.u.max` for inputs and `sys.y.min`, `sys.y.max` for outputs).

In addition, MPT3 provides an easy-to-use way to define additional constraints by a mechanism that we call "filters". Filters are user-defined properties of system's signals (states, inputs, outputs) that have to be enabled on a per-demand basis. To add a filter to a system's signal, the user calls

```
sys.signal.with('filter_name')
```

and removes it by

```
sys.signal.without('filter_name')
```

Here `sys.signal` is either `sys.x` for states, `sys.u` for inputs and `sys.y` for outputs.

MPT3 provides following filters to specify additional constraints:

1) Slew-rate constraints of the form

$$\Delta_{\min} \leq u_{k+1} - u_k \leq \Delta_{\max}, \tag{3}$$

are specified by

```
sys.u.with('deltaMin')
sys.u.with('deltaMax')
sys.u.deltaMin = dmin
sys.u.deltaMax = dmax
```

Slew-rate constraints can also be enabled for states (`sys.x`) and outputs (`sys.y`).

2) Soft min/max constraints

$$x_{\min} - s_x \leq x_k \leq x_{\max} + s_x \tag{4}$$

use slack variables $s_x \geq 0$ to allow for violation of the hard constraints:

```
sys.x.with('softMin')
```

The filter exposes two user-defined properties. First, `sys.x.softMin.maximalViolation` specifies the maximal allowed value of the constraint violation. The default value is 1000. To prevent the MPC controller from violating the constraints unless necessary, value of the slack variables need to

be penalized by including $q_s s_x$ into the cost function (1a). The value of the penalty q_s can be specified by `sys.x.softMin.penalty`, which defaults to $1 \cdot 10^4$ (cf. Section II-C for more information about penalties). Input and output constraints can be softened in a similar way.

3) Polyhedral set constraints $Hy_k \leq h$ are defined using the `setConstraint` filter:
```
sys.y.with('setConstraint')
sys.y.setConstraint=Polyhedron(H,h)
```
4) Terminal set constraints $\mathcal{T} = \{x_N \mid Hx_N \leq h\}$ in (1g) are enabled via the `terminalSet` filter:
```
sys.x.with('terminalSet')
sys.x.terminalSet=Polyhedron(H,h)
```
Unlike previous filters, the terminal set can only be enabled for state variables.
5) Blocking constraints $u_i = u_{i+1} = \cdots = u_j$ are commonly used in MPC to decrease the number of degree of freedom and thus to reduce the computational complexity. In the traditional setup, the first N_c control moves are free, while $u_{N_c} = u_{N_c+1} = \cdots = u_{N-1}$ are fixed. This behavior can be achieved by the `block` filter:
```
sys.u.with('block')
sys.u.block.from = Nc
sys.u.block.to = N
```

Filters enabled for a particular signal can be listed by the `sys.signal.listFilters()` method. Moreover, presence of a particular filter can be programatically checked by the `sys.signal.hasFilter('name')` method.

C. Cost Function

The cost function in (1a) is composed of the terminal penalty $\ell_N(\cdot)$ and the stage penalties $\ell_x(\cdot)$, $\ell_u(\cdot)$, $\ell_y(\cdot)$. The stage penalties are specified by setting the `sys.signal.penalty` attribute. Its value depends on the type of the function which penalizes the respective signal. MPT3 supports three types of penalty functions:

1) Quadratic penalties of the form $x_k^T Q x_k$ are specified by
```
sys.x.penalty = QuadFunction(Q)
```
where the penalty matrix `Q` must be positive semidefinite for states and outputs (via `sys.y.penalty`) and positive definite for inputs (`sys.u.penalty`).
2) One-norm penalties $\|Qx_k\|_1$ are defined by
```
sys.x.penalty = OneNormFunction(Q)
```
where $\|z\|_1 = \sum |z|_i$.
3) Infinity-norm penalties $\|Qx_k\|_\infty$, enabled by
```
sys.x.penalty = InfNormFunction(Q)
```
with $\|z\|_\infty = \max |z|_i$.

The terminal penalty $\ell_N(\cdot)$ is not added by default. To enable it, the user has to activate the corresponding filter manually:

```
sys.x.with('terminalPenalty')
sys.x.terminalPenalty = QuadFunction(QN)
```

where one can also use `OneNormFunction` and `InfNormFunction` as terminal penalties. Moreover, we note that MPT3 seamlessly handles cost functions composed of different cost functions, e.g. quadratic penalization of states but 1-norm penalization of inputs.

In addition to the basic terminal and stage penalties, MPT3 also allows to penalize the increments of system's signals. E.g. to obtain a smooth control profile, one can penalize the control increments $\Delta u_k = u_k - u_{k-1}$ by $\|Q_d \Delta u_k\|$ via

```
sys.u.with('deltaPenalty')
sys.u.deltaPenalty=OneNormFunction(Qd)
```

Penalization of increments of system's states and outputs is also supported.

Note that, by default, the MPC setup in MPT3 considers that the origin is the regulation objective. However, tracking of non-zero trajectories can also be enabled via the `reference` filter:

```
sys.y.with('reference')
sys.y.reference = value
```

Here, `value` can either be a vector of constant reference values, or the string `'free'` which denotes an a-priori unknown, possibly time-varying reference signal. Once tracking is enabled, the terminal and stage penalty functions penalize $\|Q(z_k - z_{\text{ref}})\|$ where z_k is a general placeholder for the states, inputs and outputs.

D. Controller Design

Once the prediction model along with all constraints is specified, the MPC controller is created by

```
ctrl = MPCController(sys, N)
```

where `sys` is the prediction model and `N` denotes the value of the prediction horizon in (1). The object `ctrl` represents an *implicit* MPC controller in which the values of the optimal control inputs, i.e., $u_0^\star, \ldots, u_{N-1}^\star$ are calculated by solving (1) for a particular initial condition x_0 using numerical optimization. Depending on the type of the prediction model, the cost function and the constraints, such a problem can either be a convex quadratic (QP) or linear (LP) problem for linear prediction models, or a mixed-integer LP or QP for PWA and MLD models.

The calculation of the optimal control inputs is performed by calling

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

which returns only the feedback control action, i.e., the first term of the sequence $\{u_0^\star, \ldots, u_{N-1}^\star\}$. The longer syntax

```
[u0, feas, open] = ctrl.evaluate(x0)
```

also returns the true/false feasibility flag `feas` and the structure `open` which denotes information of the open-loop optimal solution. Specifically, `open.cost` contains the value of (1a) at the optimum, `open.U` is the whole

open-loop optimal sequence of control inputs, and `open.X`, `open.Y` are the open-loop optimal predicted state and output trajectories, respectively.

If the MPC controller was designed to track time-varying reference trajectories (cf. Section II-C), the value of the reference can be provided as follows:

```
u0 = ctrl.evaluate(x0,'x.reference',xr)
```

Similarly, to provide the value of u_{-1} to achieve slew-rate constraints per (3), one would call

```
u0 = ctrl.evaluate(x0,'u.previous',up)
```

It is worth noting that the controller objective is "live", i.e., it reacts immediately to changes of the underlying prediction model and/or its constraints. As an example, the user can update constraints and/or penalty matrices via

```
ctrl.model.x.min = new_value
ctrl.model.u.penalty = QuadFunction(R)
```

The *explicit* representation of the MPC controller (1) is constructed by

```
expc = ctrl.toExplicit()
```

where `ctrl` is the implicit MPC controller defined previously. Here MPT3 will employ parametric programming to calculate the parameters of the explicit PWA feedback law given by

$$U_N^\star = F_i x_0 + g_i \text{ if } x_0 \in \mathcal{P}_i, \tag{5}$$

where $\mathcal{P}_i = \{x \mid H_i x \leq h_i\}$, $i = 1, \ldots, R$ are the polyhedral critical regions of the state space, R is the total number of regions, and F_i, g_i are parameters of the local affine functions that represent the open-loop optimal control sequence. Once the explicit representation is calculated, the explicit controller can be operated using the same methods as shown above. For example, `u = exp.evaluate(x0)` will provide the value of the feedback control action by searching for which region $\mathcal{P}_i$ contains x_0 and then evaluating the corresponding affine representation of the feedback law.

The user can query properties of explicit MPC controllers as follows:

- The number of critical regions is available in `expc.nr`.
- The union of critical regions is stored in `expc.partition` and can be plotted via `expc.partition.plot()`.
- The feasible set can be obtained by `expc.partition.Domain`.
- The PWA feedback law (5) is in `expc.feedback` and can be plotted by `expc.feedback.fplot()`.
- The PWA/PWQ cost function can be accessed via `expc.cost` and plotted by `expc.cost.fplot()`.

MPT3 allows to export explicit MPC controller to ANSI-C code by

```
expc.optimizer.toC('primal', 'filename')
```

in which case a C-version of the sequential search procedure in (5) will be created. The exported code can then be linked with custom application and/or be used in Simulink.

Since constructing the explicit representation of an MPC controller is a time-consuming operation, we suggest the following workflow. First, the prediction model along with constraints and parameters of the cost functions are set up. Then an implicit MPC controller object is created by the `MPCController` command, which takes little time. This controller is then used to verify whether the MPC setup meets user's expectations e.g. by performing closed-loop simulations, adjusting the MPC setup if necessary. Only when the MPC setup is deemed appropriate, the user converts the implicit controller to an explicit one.

III. Low-Complexity Explicit MPC

The well-know technical limitations which impedes applications of explicit MPC in real time are its on-line computational and memory requirements, both of which are directly proportional to the number of critical regions in (5). Therefore if the explicit MPC controller is to be applied on hardware platforms which have limited computational and storage capabilities, it is important to reduce the number of critical regions to a desired value.

MPT3 provides various means to achieve this goal. Two classes of methods are available to reduce complexity. The first one decreases the number of regions while preserving optimality of the simplified feedback, while the second category reduces complexity by inducing suboptimality.

Throughout this section we will illustrate capabilities of individual methods on the following example. The system's dynamics is

$$x^+ = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}, \tag{6}$$

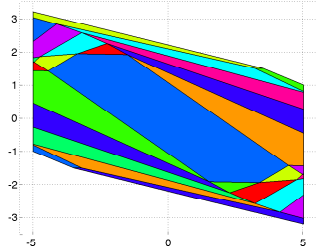
which represents a double integrator sampled at 1 second. The system is subject to constraints $\left[\begin{smallmatrix} -5 \\ -5 \end{smallmatrix}\right] \leq x \leq \left[\begin{smallmatrix} 5 \\ 5 \end{smallmatrix}\right]$ and $-1 \leq u \leq 1$. The cost function to be minimized in (1a) is given by $x_N^T Q_N x_n + \sum_{k=0}^{N} x_k^T Q_x x_k + u_k^T Q_u u_k$ with $Q_N = 5I$, $Q_x = I$, $Q_u = 1$, $N = 5$, and $\mathcal{T} = \{x_N \mid -1 \leq x_N \leq 1\}$. Such a problem is set up using

```
sys = LTISystem('A', [1 1; 0 1], ...
                'B', [1; 0.5])
sys.x.min = [-5; -5], sys.x.max = [5, 5]
sys.u.min = -1, sys.u.max = 1
sys.x.penalty = QuadFunction(eye(2))
sys.u.penalty = QuadFunction(1)
sys.x.with('terminalPenalty')
sys.x.terminalPenalty=QuadFunction(5*eye(2))
sys.x.with('terminalSet')
sys.x.terminalSet = ...
    Polyhedron('lb',[-1;-1], 'ub', [1;1])
N = 5
```

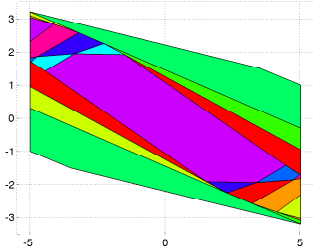
and the explicit MPC controller in (5) is constructed by

```
expc = MPCController(sys,N).toExplicit()
```

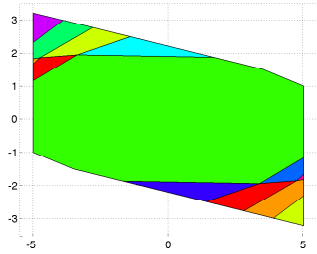
For the settings above, the controller consists of 29 critical regions in the two-dimensional state space, which are plotted in Fig. 1(a) by `expc.partition.plot()`.



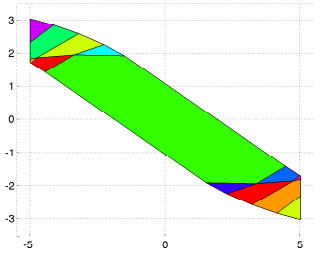
(a) 29 regions of the optimal controller.



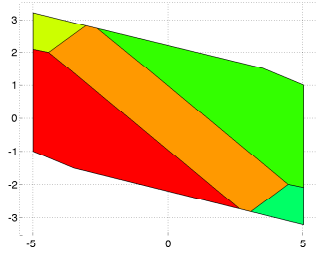
(b) 23 regions after ORM.



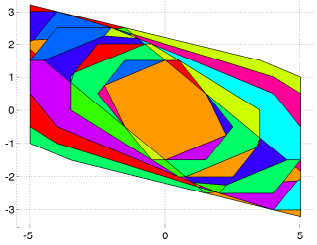
(c) 13 regions after clipping.



(d) 13 regions after separation (only the unsaturated regions are plotted).



(e) 5 regions after fitting.



(f) 20 regions in the minimum-time setup.

Fig. 1. Results of complexity reduction for the double integrator example in Section III.

A. Complexity Reduction without Inducing Suboptimality

MPT3 implementes following complexity reduction schemes which do not induce loss of optimality:

1) The optimal region merging method (ORM) of [8].
2) Clipping-based complexity reduction of [13].
3) Separation-based scheme of [14].

In all three methods a given explicit MPC controller is simplified by reducing the number of its critical regions.

The ORM method is based on merging together critical regions which share the same expression of the feedback law (i.e., identical expressions of F_i, g_i in (5)), and whose union is convex. This is achieved by using the `simplify()` gateway method as follows:

```
simple = expc.simplify('orm')
```

which returns a new explicit controller object `simple` which behaves in the same way as the input object `expc`, i.e., it can be analyzed and/or exported to C-code using the methods described in Section II-D. The ORM method is applicable to arbitrary explicit MPC feedback laws regardless of their properties, e.g., continuity of the feedback is not required. Thus this method allows to simplify explicit MPC controllers designed for PWA and MLD prediction models as well. For the double-integrator setup, the ORM method decreases the number of critical regions from 29 to 23, as can be seen in Fig. 1(b), which was generated by invoking `simple.partition.plot()`.

The clipping-based method exploits continuity of the explicit MPC feedback law (which is guaranteed if the prediction model is linear) to remove critical regions in which the control action is saturated at $u_{\min}$ or at $u_{\max}$. The space covered by such "saturated" regions is then replaced by extensions of the unsaturated critical regions, followed by applying a clipping filter. This complexity reduction scheme is available[2] via

```
simple = expc.simplify('clipping')
```

The application of the clipping-based method to the double integrator example led to 13 regions, which are shown in Fig. 1(c).

Finally, the separation-based scheme again exploits saturation properties of continuous explicit MPC feedbacks. Unlike clipping, however, saturated regions are removed altogether without replacement and only the unsaturated critical regions are kept. Then a separation function is devised to uniquely determine the value of the optimal control input for initial conditions which are not contained in the retained part of the explicit solution. The simplification is invoked by

```
simple = expc.simplify('separation')
```

which first analyzes saturation status of individual regions and then devises an appropriate separator. The 13 saturated regions for the double integrator example are depicted in Fig. 1(d).

In most practical cases, both the clipping method and the separation method yield the same result, i.e., they achieve the same level of complexity reduction. However, the clipping-based method is usually computationally less intense. In pathological cases the separation method yields higher reduction of complexity at the expense of more off-line computational effort.

We remind that all three aforementioned methods yield simplified explicit controllers which maintain the optimality of the original complex controller.

B. Complexity Reduction with Suboptimality

One way to reduce the number of critical regions of a given explicit MPC controller is to replace the optimal (but complex) feedback law $f_{\text{opt}}(x)$ by a simpler PWA function $f_{\text{aprx}}(x)$ while minimizing the suboptimality represented by

$$\int_{\mathcal{X}} \|f_{\text{opt}}(x) - f_{\text{aprx}}(x)\| \mathrm{d}x. \tag{7}$$

MPT3 solves this problem by implementing a two-step "fitting" procedure suggested in [16]. First, a simpler partition consisting of a fewer critical regions is designed by solving

[2]Note that functionality is implemented in a separate module, which can be installed by `tbxmanager install mpt3lowcom`

an MPC problem (1) for a shorter value of the prediction horizon. Subsequently, we optimize the parameters F_i, g_i of the simpler feedback in (5) as to minimize suboptimality represented by the integrated squared error in (7). Such a complexity reduction scheme is available via

```
simple = expc.simplify('fitting')
```

To assess the loss of optimality, one can compare the average and worst-case decrease of performance of a simplified controller versus an optimal one by

```
[a, w] = expc.comparePerformance(simple)
```

which returns the average loss of optimality in `a` and the worst-case drop in `w` in percentage units. Note that the method supports implicitly defined MPC controllers as well. For the double integrator example, the fitting-based scheme allows to decrease the complexity from 27 to just 5 regions, which are shown in Fig. 1(e). The simplified controller exhibits 2.7% average and 11.0% worst-case suboptimality versus the optimal controller.

The second option to reduce complexity is to employ a simpler cost function in (1a). Specifically, as suggested in [9] we can reduce complexity if instead of (1a) we solve a minimum-time problem where the objective is to minimize the arrival time to a certain terminal set. Although the synthesis of such controllers is quite involved, both theoretically as well as computationally, it is available to users via

```
simple = EMinTimeController(sys)
```

Applying this method to the double integrator setup results in an explicit MPC controller composed of 20 critical regions depicted in Fig. 1(f). Its average suboptimality is 4.4% and the worst-case drop of performance is 18.0% versus the original complex optimal controller.

C. Behind the Scenes

The purpose of this section is to enlighten the lesser known functions of MPT3 with respect to computational geometry. Specifically, we discuss basic building blocks which, when assembled together, implement the complexity reduction techniques described above.

The basic functionality required by the ORM method is the recognition of convexity of an union of polyhedra [2]. Specifically, given polytopes $\mathcal{P}_i = \{x \mid H_i x \leq h_i\}$, $i = 1, \ldots, m$, the task is to determine whether their union $\mathcal{P}_u = \cup_i \mathcal{P}_i$ is convex or not. In MPT3 this can be achieved using the `isConvex` method:

```
P1 = Polyhedron(H1, h1)
P2 = Polyhedron(H2, h2)
convex = isConvex(PolyUnion([P1, P2]))
```

More technically, the answer is obtained by realizing that $\mathcal{P}_u$ is convex if and only if $\mathrm{convh}(\mathcal{P}_u) \setminus \mathcal{P}_u = \emptyset$ where $\mathrm{convh}(\cdot)$ is the convex hull operator and "$\setminus$" represents the set difference. The convex hull $\mathcal{P}_h$ of polytopes $\mathcal{P}_i$ can be obtained by first enumerating the vertices of the polytopes, say, $\mathcal{V}(\mathcal{P}_i)$, followed by removing redundant vertices and converting the resulting set back to the half-space representation. Thus the funcitonality of the `isConvex` method can be replicated by

```
V = [P1.V; P2.V]      % vertex enumeration
Ph = Polyhedron(V)    % convex hull
diff = Ph \ [P1, P2] % set difference
convex = diff.isEmptySet() % comparison
```

The clipping- and saturation-based complexity reduction schemes rely on identification of critical regions of the PWA feedback law in which the control action is saturated. MPT3 automates such an identification procedure by means of the `findSaturated` method:

```
expc.optimizer.findSaturated('primal')
```

which returns a structure which contains information about the saturated and unsaturated regions. Note that the method analyzes the investigated PWA function and automatically determines the saturation limits without user's involvement. In addition, MPT3 provides methods for devising a linear separation of nonconvex sets given as unions of polytopes. In particular, let $\mathcal{S}_1 = \cup_i \mathcal{P}_i$, $\mathcal{S}_s = \cup_j \mathcal{Q}_j$ with $\mathcal{P}_i$, $\mathcal{Q}_j$ polytopes. The objective is to find the separator $p(x) := \alpha x + \beta$ such that $p(x) > 0$ for all $x \in \mathcal{S}_1$ and $p(x) < 0$ for all $x \in \mathcal{S}_2$. The search for the separator's parameters α and β can be done via the `separate` method:

```
S1 = PolyUnion([P1, ..., Pn])
S2 = PolyUnion([Q1, ..., Qm])
p = SeparationController.separate(S1,S2)
```

In the fitting-based approach of Section III-B the main technical difficulty is represented by integration of functions over polytopes in generic n-dimensional Euclidean space. Specifically, let $f(x)$ be a homogeneous polynomial of degree d in n variables $x \in \mathbb{R}^n$. Then the integral of f over a unit simplex Δ is given by [1]:

$$\int_{\Delta} f(y)\mathrm{d}y = \gamma \sum_{1 \leq i_1 \leq \cdots \leq i_d \leq n+1} \sum_{\epsilon \in \{\pm 1\}^d} \epsilon_1 \cdots \epsilon_d f\left(\textstyle\sum_{k=1}^{d} \epsilon_k s_{i_k}\right) \tag{8}$$

where

$$\gamma = \frac{\mathrm{vol}(\Delta)}{2^d d! \binom{d+n}{d}}, \tag{9}$$

and $\mathrm{vol}(\Delta)$ is the volume of the simplex. Thus to integrate $f(x)$ over a polytope $\mathcal{P}$, following steps are needed:

1) Homogenization of $f(x)$.
2) Tessellation of $\mathcal{P}$ into simplices $\Delta_1, \ldots, \Delta_m$. This is achieved by `D = P.triangulate()`.
3) Computation of volumes of each Δ_i by `vol = D(i).volume()`.
4) Enumeration of vertices of individual simplicies via `Vi = D(i).V`.
5) Evaluation of the integral from (8).

MPT3 automates this procedure for integration of linear and quadratic functions via the `integrate` method, which

automatically splits such functions into homogeneous components. As an example, consider[3] $\mathcal{P} = \{x \mid \left[\begin{smallmatrix}-1\\-2\end{smallmatrix}\right] \leq x \leq \left[\begin{smallmatrix}3\\4\end{smallmatrix}\right]\}$ and $f(x) = x^T \left[\begin{smallmatrix}1 & 0.1\\0.1 & 2\end{smallmatrix}\right] x + \left[\begin{smallmatrix}1 & 0\end{smallmatrix}\right] x - 3.5$. The integral of f over $\mathcal{P}$ is obtained by

```
P = Polyhedron('lb', [-1; -2], ...
               'ub', [3; 4])
f = QuadFunction([1 0.1; 0.1 2], ...
                 [1 0], -3.5)
P.addFunction(f, 'fx')
int = P.integrate('fx')
```

which results in $\int_{\mathcal{P}} f(x)\mathrm{d}x = 192.8$.

IV. Verification and Analysis

The purpose of the verification and analysis modules in MPT3 is to verify whether a closed-loop system, composed of a discrete-time dynamical system and an MPC controller, exhibits certain safety properties. In particular, the toolbox provides methods to verify closed-loop stability by devising Lyapunov certificates, and to assess invariance by means of reachability analysis.

The analysis in MPT3 is based on investigation of the closed-loop system

$$x^+ = \widetilde{f}(x, \kappa(x)), \tag{10}$$

where $\widetilde{f}(\cdot,\cdot)$ is the state-update function of the system (which can be different from the prediction model $f(\cdot,\cdot)$ employed in (1b)), and $\kappa(x)$ is an MPC feedback law with $\kappa(x) = [I, 0, \ldots, 0]U_N^\star$ where $U_N^\star = [u_0^\star, \ldots, u_{N-1}^\star]$ is the open-loop optimal sequence obtained either by solving (1) numerically (for implicit MPC controllers) or by evaluating (5) for explicit MPC feedbacks. The closed-loop dynamics of the form (10) is specified in MPT3 using the `ClosedLoop` class. As an example, we will aim at verifying whether the MPC controller designed in Section III for the nominal prediction model (6) provides desired closed-loop properties even when connected to a different system with

$$x^+ = \begin{bmatrix}1 & 1\\0 & 0.9\end{bmatrix} x + \begin{bmatrix}1\\0.5\end{bmatrix}. \tag{11}$$

The closed-loop dynamics is then created by

```
sys2 = LTISystem('A', [1 1; 0 0.9], ...
                 'B', [1; 0.5])
loop = ClosedLoop(expc, sys2)
```

Such a closed-loop system can then be analyzed e.g. by performing simulations. To streamline the process, MPT3 introduces the `simulate` method which performs such a closed-loop simulation over a desired number of time steps:

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

where `x0` is the initial state of the simulation, `Nsim` is the number of simulation steps, and the `data` output contains the closed-loop profiles of states, inputs, and outputs respectively in `data.X`, `data.U`, and `data.Y`. Moreover, `data.cost` contains the value of the cost function (1a) for each simulation step. Note that the controller (provided as the first input to the `ClosedLoop` constructor) can be both an implicit as well as an explicit controller (cf. Section II-D).

A more rigorous way to verify closed-loop stability for all feasible initial conditions is to devise a Lyapunov function V which satisfies $V(0) = 0$, $V(x) > 0$ for all $x \neq 0$, and $V(x^+) \leq \gamma V(x)$ for some $\gamma \in [0, 1)$ and all x for which the MPC problem (1) is feasible. MPT3 implements the methods of [5] to synthesize piecewise quadratic (PWQ) and piecewise affine (PWA) Lyapunov functions provided the system's dynamics in (10) is linear or piecewise affine, and the explicit representation of the MPC feedback law is available. In such a case the closed-loop system (10) becomes a PWA system of the form

$$x^+ = \widetilde{A}_i x + \widetilde{c}_i \text{ if } x \in \mathcal{P}_i. \tag{12}$$

Such a PWA representation of the closed-loop system can be obtained by

```
pwa = loop.toSystem()
```

which automatically links critical regions of the controller with regions of the controlled PWA system (if the system is linear, the regions of (12) coincide with critical regions of the controller). The Lyapunov function for the autonomous PWA system of the form (12) is then computed by

```
lyap = pwa.lyapunov(type)
```

where `type='pwq'` or `type='pwa'` depending on the class of the desired Lyapunov function. For the double integrator example from Section III, a PWQ Lyapunov function was found, which certifies that the MPC controlled designed for (6) stabilizes (11). Such an approach can be also used to verify closed-loop stability when a suboptimal controller, e.g. one constructed per Section III-B, is employed. Specifically, by running

```
pwa = ClosedLoop(simple, sys).toSystem()
lyap = pwa.lyapunov('pwq')
```

we were able to verify that both the fitting controller as well as the minimum-time strategy described previously provide closed-loop stability.

MPT3 also provides functionality to verify invariance of a given closed-loop system and to perform reachability analysis. Invariance of an autonomous PWA system (12) (which includes autonomous LTI systems as a special case) can be assessed by

```
answer = pwa.isInvariant()
```

which returns true if for each x_0 the dynamics of (12) is such that $x_k \in \cup_i \mathcal{P}_i$ for all $k \geq 0$; and false otherwise. If the answer is negative, MPT3 can calculate the invariant subset of (12) via

```
pwa_inv = pwa.invariantSet()
```

which generates new regions $\mathcal{P}_i$ in (12) for which the invariance property holds ad infinitum.

[3]The toolbox naturally supports integration over generic polytopes, not just over hyperboxes.

Computation of invariant sets in MPT3 is in fact more general since it covers construction of maximal control invariant sets as well. Specifically, for a dynamical system $x_{k+1} = f(x_k, u_k)$ subject to constraints $x \in \mathcal{X}$ and $u \in \mathcal{U}$ the maximal control invariant set $\mathcal{C}_\infty \subseteq \mathcal{X}$ is

$$\mathcal{C}_\infty = \{x_0 \mid \exists u_k \in \mathcal{U} \text{ s.t. } f(x_k, u_k) \in \mathcal{X} \ \forall k \geq 0\}. \quad (13)$$

If the system's dynamics is linear of piecewise affine, the maximal control invariant set is constructed by

```
Cinf = sys.invariantSet()
```

What appears simple to the user is in fact an involved algorithm which is internally implemented by performing backwards reachability analysis. Specifically, let $\mathrm{Pre}(\mathcal{S}) = \{x \mid \exists u \in \mathcal{U} \text{ s.t. } f(x,u) \in \mathcal{S}\}$ be the one-step controllable set to the set $\mathcal{S}$. The existence operator can be eliminated by projecting the polyhedron $\mathcal{R} = \{(x,u) \mid u \in \mathcal{U},\ f(x,u) \in \mathcal{S}\}$ onto the x-space. Then $\mathcal{C}_\infty$ is obtained by running the recursion $\mathcal{S}_{k+1} = \mathrm{Pre}(\mathcal{S}_k) \cap \mathcal{S}_k$, initialized by $\mathcal{S}_0 = \mathcal{X}$ until convergence, detected when $\mathcal{S}_{k+1} = \mathcal{S}_k$. In MPT3, each individual pre-set can be obtained by

```
S(k+1) = sys.reachableSet('X', S(k), ...
                'direction', 'backward')
```

which automates the process of projecting $\mathcal{R}$ onto the x-space and intersecting the new set $\mathcal{S}_{k+1}$ with $\mathcal{S}_k$.

One case where maximal invariant sets are used frequently is when the so-called LQR set $\mathcal{O}_{\text{LQR}}$ needs to be constructed as the set where the LQR controller does not activate any constraints. Specifically, $\mathcal{O}_{\text{LQR}}$ is the maximal positive invariant set for the autonomous system $x_{k+1} = (A + BK_{\text{LQR}})x_k$ with $x_k \in \mathcal{X}$, $u_k \in \mathcal{U}$ for all $k \geq 0$. For a given linear system $x_{k+1} = Ax_k + Bu_k$ modeled per Section II-A, the set $\mathcal{O}_{\text{LQR}}$ is constructed by

```
OLQR = sys.LQRSet()
```

What MPT3 does internally is to first compute the LQR gain K_{LQR}, then form the closed-loop system composed of the linear system and the LQR controller, and then call the `invariantSet` method for such a closed-loop system.

MPT3 also computes forward reachable sets for autonomous systems of the form $x^+ = f(x)$ and for systems with control inputs given by $x^+ = f(x,u)$ where the state-update equation is either linear or piecewise affine. The set of states reachable by the system in one time step starting from some initial set $\mathcal{S}$ is given by $\mathcal{F}(\mathcal{S}) = \{f(x) \mid x \in \mathcal{S}\}$ for the autonomous dynamics, or $\mathcal{F}(\mathcal{S}) = \{f(x,u) \mid x \in \mathcal{S}, u \in \mathcal{U}\}$ for non-autonomous systems. In MPT3, this is achieved by

```
F = sys.reachableSet('X', S, ...
                'direction', 'forward')
```

Depending on the level of granularity the user wants to operate with, basis reachability tasks can be performed by applying direct and inverse affine transformations of polyhedra. Specifically, len $\mathcal{S}$ be a polyhedron in $\mathcal{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $c \in \mathbb{R}^m$. Then $A \circ \mathcal{S} + c = \{Ax + c \mid x \in \mathcal{S}\}$ denotes the (direct) affine transformation of the set $\mathcal{S}$ by the affine map (A, c), and $\mathcal{S} \circ A + c = \{x \mid (Ax + c) \in \mathcal{S}\}$ is the inverse affine transform. If A and c would represent matrices of an autonomous affine system $x^+ = Ax + c$, then the direct transform yields forwards reachable sets, while the inverse transform coincides with backwards reachability. MPT3 implements affine transformations via the "$*$" operator, i.e.,

```
direct = A*S+c
inverse = S*A+c
```

Note that if the dimension of A satisfies $m \leq n$, then the affine transformation eliminates variables by projecting $\mathcal{S}$ onto the space spanned by A. If $m > n$, then $A \circ \mathcal{S}$ represents lifting of the set $\mathcal{S}$. If $m = n$ (with $\mathrm{rank}(A) = n$), then the set $\mathcal{S}$ is rotated and/or scaled. This is how how reachability and invariance operations are implemented in MPT3 on the basic level.

REFERENCES

[1] V. Baldoni, N. Berline, J. A. De Loera, M. Köppe, and M. Vergne. How to integrate a polynomial over a simplex. *Mathematics of Computation*, 80(273):297, 2010.

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

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

[4] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, January 2002.

[5] P. Biswas, P. Grieder, J. Löfberg, and M. Morari. A Survey on Stability Analysis of Discrete-Time Piecewise Affine Systems. In *IFAC World Congress*, Prague, Czech Republic, 2005.

[6] F. Borrelli. *Constrained Optimal Control of Linear and Hybrid Systems*, volume 290. Springer-Verlag, 2003.

[7] Ch. Feller and T.A. Johansen. Explicit MPC of higher-order linear processes via combinatorial multi-parametric quadratic programming. In *Control Conference (ECC), 2013 European*, pages 536–541. IEEE, 2013.

[8] T. Geyer, F.D. Torrisi, and M. Morari. Optimal complexity reduction of polyhedral piecewise affine systems. *Automatica*, 44(7):1728–1740, July 2008.

[9] P. Grieder, M. Kvasnica, M. Baotic, and M. Morari. Stabilizing low complexity feedback control of constrained piecewise affine systems. *Automatica*, 41, issue 10:1683–1694, October 2005.

[10] A. Gupta, S. Bhartiya, and P. Nataraj. A novel approach to multiparametric quadratic programming. *Automatica*, 47(9):2112–2117, 2011.

[11] M. Herceg, M. Kvasnica, C. Jones, and M. Morari. Multi-parametric toolbox 3.0. In *2013 European Control Conference*, pages 502–510, 2013.

[12] M. Kvasnica. *Real-Time Model Predictive Control via Multi-Parametric Programming: Theory and Tools*. VDM Verlag, Saarbruecken, January 2009.

[13] M. Kvasnica and M. Fikar. Clipping-Based Complexity Reduction in Explicit MPC. *IEEE Trans. Automatic Control*, 57(7):1878–1883, July 2012.

[14] M. Kvasnica, J. Hledík, I. Rauová, and M. Fikar. Complexity reduction of explicit model predictive control via separation. *Automatica*, 49(6):1776–1781, 2013.

[15] M. Rubagotti, S. Trimboli, D. Bernardini, and A. Bemporad. Stability and invariance analysis of approximate explicit MPC based on PWA Lyapunov functions. In *Proc. IFAC World Congress, Milan, Italy*, pages 5712–5717, 2011.

[16] B. Takács, J. Holaza, M. Kvasnica, and S. Di Cairano. Nearly-optimal simple explicit mpc regulators with recursive feasibility guarantees. In *IEEE Conference on Decision and Control*, pages 7089–7094, 2013.