2016 European Control Conference
June 29 - July 1, 2016. Aalborg, Denmark

Python Code Generation for Explicit MPC in MPT

Bálint Takács, Juraj Števek, Richard Valo and Michal Kvasnica

Abstract— **The paper shows how explicit representations of model predictive control (MPC) feedback laws can be embedded into Python applications via a new code-generation module of the Multi-Parametric Toolbox. The advantage of the explicit approach is that it provides a simple and fast computation of optimal control inputs without solving optimization problems on-line. To enable implementation of discontinuous feedback laws, the paper proposes an extended version of the sequential search algorithm which resolves possible multiplicities based on a secondary evaluation of the cost function. Two applications are considered. The first one is the Flappy Bird game where we design an MPC-based artificial player to control flapping of the bird's wings. The second application considers the design and implementation of an explicit MPC controller for a quadrocopter.**

I. Introduction

Model Predictive Control (MPC) is one of the most commonly adopted control strategies in the industrial field [14]. It is endorsed due to its natural capability to handle multidimensional systems, while incorporating state and input constraints directly into the decision making process [6]. Solution of such an optimization problem yields a sequence of optimal control inputs over a given prediction horizon. At the next sampling instant, new state measurements are taken and the optimization is repeated. There are two factors which limit the applicability of this approach. The first one is its inherited computational complexity associated with solving an optimization problem at each sampling instant. This limits (and sometimes prohibits) applicability of MPC for control of systems with a fast dynamics. The second, often overlooked, limitation is the code complexity. Specifically, MPC optimization problems are usually solved via iterative active set methods which require, among other steps, to perform a series of matrix inversions. Thus additional numerical libraries are required to implement such controllers.

Both of these limitations can be addressed by the concept of *explicit MPC* [2]. Here, parametric programming is used to pre-calculate the optimal solution to a given MPC problem in the form of a piecewise-affine (PWA) function. This function, which serves as an explicit MPC feedback law, maps state measurements onto optimal control inputs. The computation of optimal control inputs thus reduces to a mere function evaluation. The advantage over classical numerical solutions of MPC problems is thus in the reduction of the computational effort required to obtain optimal control inputs, as well as in the simplicity of the implementation algorithm, which, among other things, is division-free. Hence neither divisions, nor matrix inversions are required to obtain the optimal control input. This allows explicit MPC to be implemented in almost any programming language.

Nowadays, a large portion of the research community uses Matlab to formulate MPC-based controllers and Simulink/Real-Time Workshop to deploy them. However, recently other languages started to be more appealing for control purposes. One of them is Python. Several control-oriented packages already exist for Python. One of them is CasADi [11], a symbolic framework for numeric optimization implementing automatic differentiation in forward and reverse modes on sparse matrix-valued computational graphs. The construction of the optimization problem is defined symbolically afterward the appropriate solver is called to solve the optimization problem. It interfaces several optimization solvers such as IPOPT [3], SNOPT [8] etc.

However, to the authors' best knowledge, there is no simple way how to merge MPC control strategies into existing Python applications in an easy fashion. In this paper we describe how the generated explicit solution can be easily exported to a Python code and merged into existing applications. The generated code consists of two parts - the data and the code. The data describes the PWA explicit solution consisting of critical regions and associated function expressions. The code then evaluates the PWA feedback function for the given value of state measurements. Two versions of the evaluation code are discussed. The first one is a standard sequential search algorithm, which traverses through the pieces of the PWA function until the piece which contains the state is found. Such a simple strategy, however, requires that the explicit optimizer is a continuous PWA function. Therefore we also propose an extended version of the sequential search procedure which produces the correct output even in the case of discontinuous MPC feedback laws. In both cases we show that the Python code can be generated using just couple of lines in Matlab. The proposed code-generating module [15] for MPT [10] is demonstrated on two examples.

A. Notation and Definitions

We denote by $\mathbb{R}$, $\mathbb{R}^n$ and by $\mathbb{R}^{n\times m}$ the real numbers, n-dimensional real vectors, and $n \times m$ dimensional real matrices, respectively. Furthermore, $\mathbb{N}$ denotes the set of non-negative integers, and $\mathbb{N}_i^j$ the set of consecutive integers, i.e., $\mathbb{N}_i^j = \{i, \dots, j\}$, $i \leq j$. Given a countable set $\mathcal{I}$, $|\mathcal{I}|$ denotes its cardinality.

Definition 1.1 (Polyhedron): Polyhedron $\mathcal{P}$ is a convex and closed set defined as the intersection of a finite number

All authors are with the Institute of Information Engineering, Automation, and Mathematics at the Slovak University of Technology in Bratislava, Slovakia. {balint.takacs, juraj.stevek, richard.valo, michal.kvasnica}@stuba.sk.

c of closed affine half-spaces $a_i^T x \leq b_i$, $a_i \in \mathbb{R}^n$, $b_i \in \mathbb{R}$, $\forall i \in \mathbb{N}_1^c$. Polyhedra can be compactly represented by

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

with $A \in \mathbb{R}^{c \times n}$, $b \in \mathbb{R}^c$.

II. Explicit Model Predictive Control

We consider MPC problems of the form

$$J^\star(x_0) = \min \ \ell_N(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k), \tag{2a}$$

$$\text{s.t. } x_{k+1} = f(x_k, u_k), \ \forall k \in \mathbb{N}_0^{N-1} \tag{2b}$$

$$u_k \in \mathcal{U}, \ \forall k \in \mathbb{N}_0^{N-1} \tag{2c}$$

$$x_k \in \mathcal{X}, \ \forall k \in \mathbb{N}_0^{N-1} \tag{2d}$$

$$x_N \in \mathcal{T}, \tag{2e}$$

where x_k, u_k are the state and input predictions, respectively, obtained at the k-th step of the prediction horizon $N \in \mathbb{N}$. The objective function is composed of the terminal penalty function $\ell_N(\cdot)$ and the stage cost $\ell(\cdot, \cdot)$

$$\ell_N(x_N) = \|Q_N x_N\|_p, \tag{3a}$$

$$\ell(x_k, u_k) = \|Q_x x_k\|_p + \|Q_u u_k\|_p, \tag{3b}$$

where $p \in \{1, 2, \infty\}$.

The state predictions in (2b) are obtained via a state-update function $f(\cdot, \cdot)$. In this paper we consider two types of prediction models. The first case is represented by linear time-invariant state-update equations of the form

$$f(x, u) = Ax + Bu. \tag{4}$$

The second, more general type is represented by piecewise affine (PWA) functions of the form

$$f(x,u) = \begin{cases} A_1 x + B_1 u + c_1 & \text{if } \left[\begin{smallmatrix} x_k \\ u_k \end{smallmatrix}\right] \in \mathcal{P}_1 \\ \qquad\qquad \vdots & \\ A_p x + B_p u + c_p & \text{if } \left[\begin{smallmatrix} x_k \\ u_k \end{smallmatrix}\right] \in \mathcal{P}_p \end{cases} \tag{5}$$

Here, $\mathcal{P}_i$ is a polyhedron in the state-input space in which the i-th affine dynamics are locally active. Moreover, p denotes the total number of distinct local dynamics, also called *modes* in the literature.

The constraint sets $\mathcal{U}$, $\mathcal{X}$, and $\mathcal{T}$ in (2) can be either convex polyhedra, or non-convex unions therefor. A special case employed in this paper is when $\mathcal{U} = \{0, 1\}^{n_u}$, which indicates that the control input is binary.

Regardless of whether the prediction model is linear or piecewise affine, the analytic solution to (2), i.e., the map from initial conditions x_0 to optimal receding horizon control inputs $u_0^\star$, can be obtained by parametric programming [4, 1]. The properties of such solutions are summarized next.

Theorem 2.1 ([2]): The optimal solution to (2) is a piecewise affine (PWA) function of x_0, i.e.,

$$u_0^\star(x_0) = \begin{cases} F_1 x_0 + g_1 & \text{if } x_0 \in \mathcal{R}_1 \\ \qquad\qquad \vdots & \\ F_M x_0 + g_M & \text{if } x_0 \in \mathcal{R}_M \end{cases} \tag{6}$$

where $\mathcal{R}_i = \{x \mid H_i x \leq h_i\}$ are polyhedral critical regions and M denotes the total number of such regions. Moreover, if the objective function in (2a) is composed of (3), then $J^\star(x_0)$ is also a PWA/PWQ function of the form

$$J^\star(x_0) = \begin{cases} x_0^T \gamma_1 x_0 + \alpha_1 x_0 + \beta_1 & \text{if } x_0 \in \mathcal{R}_1 \\ \qquad\qquad \vdots & \\ x_0^T \gamma_M x_0 + \alpha_M x_0 + \beta_M & \text{if } x_0 \in \mathcal{R}_M \end{cases} \tag{7}$$

■

The parameters $\mathcal{R}_i$, F_i, g_i, γ_i, α_i, β_i of (6)−(7) can be obtained using the freely available Multi-Parametric Toolbox for Matlab [10].

With the explicit form (6) in hand, obtaining the optimal control action reduces to a simple function evaluation. Such a way is typically faster compared to solving (2) using numerical algorithms. Moreover, the evaluation algorithm for (6) is very simple and allows MPC to be embedded into existing applications. The details are discussed in the following section.

III. Point location problem

To find the optimal control action $u^\star$ associated with the current state measurements x it is necessary to evaluate the PWA feedback function in (6). To do so, we first need to determine which critical region contains x. This problem is commonly known as the *point location problem*. Various algorithms exist to answer this problem. In this paper we focus on the *sequential search* algorithm, which has a very simple implementation. We consider two situations. The first one covers feedback laws in (6) which are continuous. The second, extended version, also copes with discontinuous feedbacks.

It should be noted that more efficient algorithms also exist. The binary search tree approach of [13] provides the fastest known procedure to evaluate the function in (6). However, it relies on a particular search structure, which is difficult to obtain especially when the number of critical regions is large. Other authors, such as [9, 16], have proposed that point location problem can be effectively accelerated by exploiting adjacency list that is inherently associated with the polytopic partition of the PWA function. However, a common drawback of all three referenced approaches is that they can only be applied to continuous feedback laws in (6).

The standing assumption of this section is that x, the initial condition of (2), is feasible, i.e., it is contained in at least one critical region of (6). Should this assumption be violated, there exists no solution to (2) for this specific initial condition.

A. Sequential search algorithm for continuous feedback laws

Continuity of (6) implies that for each feasible x there exists at least one critical region with $x \in \mathcal{R}_i$. If there are multiple such regions (which can happen if x is contained in the boundary shared among multiple regions), by continuity of (6) the associated control actions are all equivalent. Therefore to evaluate $u^\star(x)$ from (6) it suffices to find the first

critical region which contains the state measurements. This is done by traversing through all critical regions in a sequential order, stopping once the first region containing x is found. This behavior is summarized by Algorithm 1. The inclusion test in Step 3 can be done by verifying whether $H_i x \leq h_i$ holds. Note that only additions and multiplications are required to perform such a check. Similarly, the computation of $u^\star$ in Step 4 is also division-free.

Algorithm 1: Sequential search for continuous functions

Data: Feedback laws F_i, g_i, critical regions $\mathcal{R}_i$, number of regions M, state measurement x
Result: Optimal control input
1 **for** $i = 1, \ldots, M$ **do**
2 **if** $x \in \mathcal{R}_i$ **then**
3 return $u^\star = F_i x + g_i$
4 **end**
5 **end**

B. Sequential search algorithm for discontinuous feedback laws

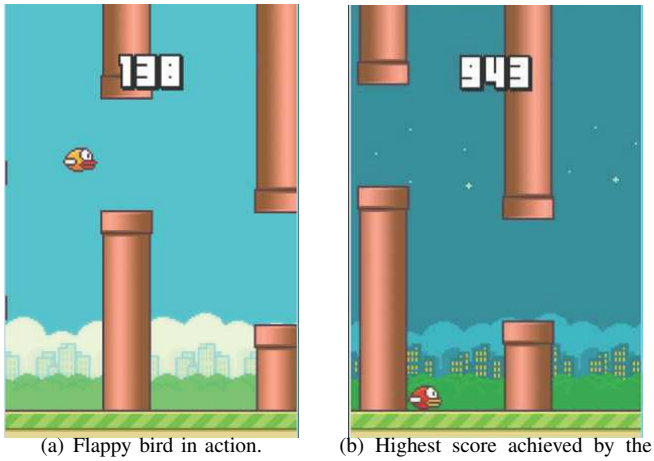
Discontinuity of (6) can occur when the critical regions overlap , or if the function values at the boundary of two adjacent critical regions do no coincide (e.g. when the control input is binary).

In what follows we propose an extended sequential search procedure for discontinuous functions in (6). The idea is to first build the index set of regions which contain the given query point x. Then, in the second stage, a single index is selected from the set of candidates. The latter stage will be referred to as resolving of *tiebreaks*. Technically, the extended sequential search procedure is captured by Algorithm 2.

Algorithm 2: Sequential search with tiebreaks

Data: Feedback laws F_i, g_i, critical regions $\mathcal{R}_i$, number of regions M, query point x
Result: Optimal control input
1 $\mathcal{I} \leftarrow \emptyset$
2 **for** $i = 1, \ldots, M$ **do**
3 **if** $x \in \mathcal{R}_i$ **then**
4 $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
5 **end**
6 **end**
7 Select $i^\star \in \mathcal{I}$
Output: $u^\star = F_{i^\star} x + g_{i^\star}$

First, Step 1 initializes the set of candidates to an empty set. Then the algorithm goes through all critical regions in a sequential order. In Step 3, $x \in \mathcal{R}_i$ is verified by checking whether all inequalities in $H_i x \leq h_i$ hold (note that the comparison is between vectors and is interpreted element-wise). If a matching region is found, its index is added to the set of candidates in Step 4. Afterwards, a single critical region is then picked in Step 7. We will detail the selection in the sequel. Finally, with the index $i^\star$ of the active critical region, the algorithm returns the optimal control action, calculated from the $i^\star$-th rule in (6).



(a) Flappy bird in action. (b) Highest score achieved by the MPC controller.

Fig. 1. Screenshots of the Flappy Bird game.

When resolving tiebreaks in Step 7, three situations can occur:

1) $|\mathcal{I}| = 0$, i.e., the set $\mathcal{I}$ is empty;
2) $|\mathcal{I}| = 1$, i.e., the set contains exactly one index;
3) $|\mathcal{I}| > 1$, i.e., there are several candidate critical regions.

The first situation indicates that problem (2) is infeasible for a given initial condition. In the second situation, $i^\star = \mathcal{I}_1$, as we have only one candidate critical region. The last situation requires identifying the critical region in which the optimal objective function is minimal. This is done as follows. For each $i \in \mathcal{I}$, compute $J_i(x)$ (7), i.e.,

$$J_i(x) = x^T \gamma_i x + \alpha_i x + \beta_i \tag{8}$$

Finally, $i^\star$ is selected as the index for which $J_i(x)$ is smallest, i.e.,

$$i^\star = \arg\min_{i \in \mathcal{I}} J_i(x). \tag{9}$$

Algorithm 2 with the tiebreak procedure described above can be easily implemented in any high-level programming language. We remark that the implementation is division-free, since only multiplications and additions are required to compute $u^\star$. In the following section we show how a Python version of Alg. 2 can be automatically generated based on the data in (6)−(7).

IV. Code Generation

In this section we introduce an extension[1] of the Multi-Parametric Toolbox (MPT) which allows the user to automatically generate the Python version of Algorithms 1 and 2. Our implementation relies on the `numpy` and `math` Python libraries, which are open-source and freely available. From

[1]The Python code generation module is available since MPT version 3.1.2.

a user point of view, the generation of the Python code is as simple as calling

```
opt.toPython('myctrl','primal','obj')
```

Here, `opt` is the variable which contains the explicit representation of the feedback law in (6) and the cost function, defined either by (7), depending on which type of objective function was employed in (2a). The variable is an instance of MPT's `PolyUnion` class which is capable of describing arbitrary functions defined over a union of polyhedra. Moreover, `'myctrl'` is the string which contains the name of the file which should be generated. Note that the `.py` suffix will be added automatically to the file name. Next, `'primal'` means that we want to evaluate the primal optimizer in (6). Finally, `'obj'` indicates that we want to resolve the tiebreaks based on the objective function per (9).

Alternatively, calling

```
opt.toPython('myctrl','primal','first-region')
```

will generate the Python version of Alg. 1, which uses the first-region tiebreak, i.e., always picking the first region which contains the given query point. Note that this setting can only be used if (6) is a continuous PWA function.

In either case, the `toPython` method generates a new file `myctrl.py`, which contains the code of the sequential algorithm as well as the data of the functions in (6)−(7). The code can then be inserted into an arbitrary Python application and executed. The execution is achieved by calling `u = myctrl(x)` in Python, where `x` is an instance of the `matrix` class (provided by the `numpy` package). The output `u` will again be a vector of control inputs, again as an instance of the `matrix` class. If no critical region contains given `x`, `u` will be returned as an array of NaN (not a number). Such an output indicates that the MPC problem (2) is infeasible for the given initial condition.

In the following section we show how the procedure can be applied to embed explicit MPC controllers into two distinct applications. The first one is a popular computer game where MPC takes the role of an artificial player. The second application discusses model predictive control of a quadrocopter.

V. Examples

A. Flappy Bird

Flappy Bird[2] is a computer game in which the player controls vertical movement of a bird with the goal to avoid obstacles. The objective of the game is to achieve the highest score, measured as the number of obstacles the bird has avoided. The bird is controlled by an ON/OFF actuator, which corresponds to flapping of the birds wings. The obstacles are represented by randomly generated pipes which form a corridor through which the bird must fly. A screenshot of the game is provided in Figure 1.

In this section we show the design of an artificial player based on model predictive control. The game runs at a fixed sampling frequency and employs a switching dynamics of the bird with two states: vertical position p and vertical velocity v (due to the coordinate system employed by the Python game, positive speeds represent descents while negative ones stand for the ascending direction). Both are normalized to pixels. The dynamics itself consists of two modes.

$$x_{k+1} = Ax_k + f, A \in \{A_1, A_2\}, f \in \{f_1, f_2\} \tag{10}$$

The first mode, with $A_1 = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ and $f_1 = \left[\begin{smallmatrix} 0 \\ g \end{smallmatrix}\right]$ corresponds to the "no flapping" mode in which the bird falls down in a free fall. Here, $x = [p\ v]^T$ is the state vector, and $g = 1$ is the normalized constant acceleration due to gravity. This mode corresponds to a zero control input, i.e., it is active when $u = 0$.

The second, "flapping", mode, with $A_1 = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right]$ and $f_1 = \left[\begin{smallmatrix} -h \\ -h \end{smallmatrix}\right]$ is active when $u = 1$. Here, $h = 9$ is a game-specific constant which says that the bird jumps 9 pixels upwards if flapping of the wings is activated. Simultaneously, the vertical speed is reset to $-h$. As noted above, negative value means that the bird moves upwards.

The overall switching dynamics of the bird is then given by

$$x_{k+1} = \begin{cases} A_1 x_k + f_1 \text{ if } u_k = 0 \\ A_2 x_k + f_2 \text{ if } u_k = 1. \end{cases} \tag{11}$$

Our objective is to devise an MPC-based artificial player which will decide the optimal value of the binary control input $u \in \{0, 1\}$ such that the bird avoids obstacles. In this paper we achieve this by letting the bird's position (represented by the first state p) to follow a user-specified reference p_{ref}, which is given as the center point of the gap between the closest set of obstacles. The corresponding MPC problem is then stated as

$$\min \sum_{k=0}^{N-1} \|p_k - p_{\text{ref}}\|_1 \tag{12a}$$

$$\text{s.t. } x_{k+1} = \begin{cases} A_1 x_k + f_1 \text{ if } u_k = 0 \\ A_2 x_k + f_2 \text{ if } u_k = 1 \end{cases} \tag{12b}$$

$$p_k = [1\ 0] x_k, \tag{12c}$$

$$u_k \in \{0, 1\}, \tag{12d}$$

where (12d) captures the binary character of control inputs. As a consequence, problem (12) is a mixed-integer optimization problem.

Such an MPC problem can be formulated in the Multi-Parametric Toolbox as follows. First we create the "no flapping" mode:

```
1 A1=[1 1; 0 1]; B1=[0; 0];
2 f1=[0; 1]; C = [1 0];
3 P1=Polyhedron('ub', 0.5);
4 m1=LTISystem('A',A1,'B',B1,'f',f1,'C',C)
5 m1.setDomain('u', P1)
```

Here, line 2 creates the polyhedron which corresponds to $u = 0$. In fact, to improve numerical robustness, we define the region of validity as the set of inputs which are smaller than 0.5, exploiting the binary property of the control input. The final line then attaches the region of validity to the local affine dynamics. The "flapping" mode is created in a similar fashion:

[2] `http://flappybird.io`

```
A2=[1 0; 0 0]; B2=[0; 0];
f2=[-9;-9]; C = [1 0];
P2=Polyhedron('lb', 0.5);
m2=LTISystem('A',A2,'B',B2,'f',f2,'C',C)
m2.setDomain('u', P2)
```

Finally, the aggregated PWA model in (12b) is created using the `PWASystem` constructor which takes a list of the local modes as its input:

```
model = PWASystem([m1, m2])
```

The MPC problem in (12) with prediction horizon $N = 5$ is then formulated by

```
model.u.with('binary')
model.y.with('reference')
model.y.reference = 'free'
model.y.penalty = OneNormFunction(1)
N = 5
mpc = MPCController(model, N)
```

Here, in the first line the control inputs are declared as binary, which automatically adds the $u_k \in \{0, 1\}$ constraints. Subsequently, we enable the state to track a time-varying reference. Then, the objective function is specified as penalization of the 1-norm. Finally, an object which represents the MPC control is created with the provided prediction horizon. The explicit form of the feedback law in (6) is obtained by calling

```
empc = mpc.toExplicit()
```

which invokes a parametric optimization solver which results in (6) and (7). Since the MPC problem is non-convex due to the binary constraints in (12d), the feedback law $u^\star(x)$ in (6) can be discontinuous. With $N = 5$, the explicit solution consists of 94 regions in the three dimensional parametric space (one dimension is for the current position $x_1(t)$, the next is the current velocity $x_2(t)$ and the other one for the time-varying reference p_{ref}).

To obtain a Python version of the explicit MPC feedback law, we have then executed

```
opt = empc.optimizer;
opt.toPython('flappympc','primal','obj')
```

which generates the Python version of Alg. 2, which uses the tiebreak rule based on (7) as discussed in Section III-B.

The point location algorithm was subsequently embedded to the Python version of Flappy Bird, downloaded from GitHub[3]. The game uses features of the Pygame framework[4], which is a set of modules in Python specially designed for creating games. Pygame is highly portable and runs on almost every platform and operating system. Furthermore, it is open source. The game runs at a fixed sampling frequency. At each step, the main loop checks whether a specific key on the keyboard was pressed to indicate a flapping command. Then the loop calculates the new vertical position of the bird and draws a new graphical frame. Replacing the human player by the MPC algorithm is very easy. Instead of checking the key press, one calls the MPC sequential search algorithm which determines whether wings should be flapped or not.

[3]https://github.com/sourabhv/FlappyBirdClone

[4]http://www.pygame.org

TABLE I

ACHIEVED SCORE IN FLAPPY BIRD

Player	Average score	Best score
1	16	33
2	14	19
3	9	17
MPC	271	943

In our experiments, the MPC-based artificial player has achieved the best score of 943 tubes without a collision. To put this number into perspective, we have asked three colleagues to compete against MPC by controlling the bird manually. Their respective scores are reported in Table I, which also shows the average over 10 trials. As can be seen, the MPC controller outperforms human players by a large margin. Note that the MPC setup in (12) possesses no guarantees of recursive feasibility since only the nearest set of obstacles is considered via the choice of p_{ref}. It is *not* the objective of this paper to provide the best possible controller. Instead, the case study illustrates that MPC can be applied even to less traditional control setups and leaves space for further improvements.

B. Ar.Drone Control

In this section we show how to generate and apply model predictive control for the yaw angle control for the Ar.Drone2 quadrotor [5] The implementation is provided by means of the Robot Operating System (ROS) [12]. The onboard software implements basic control loops based on the measurements from various sensors, such as accelerometers, ultrasound sensor, barometer, and electronic compass. It controls the vertical speed $\dot{z}$, yaw speed $\dot{\psi}$, roll ϕ and pitch angle θ based on references it acquires via WiFi. The MPC objective is to devise these references in an optimal fashion. Specifically, the control inputs generated by MPC are:

- $u_{\dot{z}}$, the control command for velocity in vertical axis;
- $u_{\dot{\psi}}$, the control command for rotation velocity in z axis;
- u_{ϕ}, the control command for rotation over the x axis;
- u_{θ}, the control command for rotation over the y axis.

All inputs are normalized and constrained by $-1 \leq u \leq 1$.

In this paper we control the yaw angle with respect to the global frame. The simplified mathematical model of the Ar.Drone's rotational dynamic over the z axis was presented in [7]:

$$\ddot{\psi} = K_1 u_{\dot{\psi}} - K_2 \dot{\psi}, \tag{13}$$

where $\dot{\psi}$ represents the linear acceleration over the z axis. The parameters K_1, K_2 are model gains that have to be experimentally identified. The first term $K_1 u_{\dot{\psi}}$ represents accelerating force, and $K_2\dot{\psi}$ represents drag force in (13).

Applying the forward Euler discretization to (13), the discrete time model of the system takes the form

$$\underbrace{\begin{bmatrix}\dot{\psi}_{k+1}\\ \psi_{k+1}\end{bmatrix}}_{x_{k+1}} = \underbrace{\begin{bmatrix}1 - K_2 T_s & 0\\ T_s & 1\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}\dot{\psi}_k\\ \psi_k\end{bmatrix}}_{x_k} + \underbrace{\begin{bmatrix}K_1 T_s\\ 0\end{bmatrix}}_{B}\underbrace{u_{\dot{\psi}_k}}_{u_k} \tag{14}$$

where T_s is the sampling period. To achieve tracking properties, the system in (14) was augmented into the form

$$\tilde{x}_{k+1} = \underbrace{\begin{bmatrix} A & B \\ 0 & I \end{bmatrix}}_{\tilde{A}} \tilde{x}_k + \underbrace{\begin{bmatrix} B \\ I \end{bmatrix}}_{\tilde{B}} \Delta u_k, \tag{15}$$

where $\tilde{x}_k = \begin{bmatrix} \dot{\psi}_k & \psi & u_{k-1} \end{bmatrix}^T$.

Using the Multi-Parametric Toolbox we have formulated the following MPC problem, objective of which is to manipulate the rotational velocity such that the yaw angle tracks a prescribed reference. This is achieved by formulating the problem as follows:

$$\min \sum_{k=0}^{N-1} \Delta y_k^T Q_{y_k} \Delta y + \Delta u_k^T Q_u \Delta u_k, \tag{16a}$$

$$\text{s.t. } \tilde{x}_{k+1} = \tilde{A}\tilde{x}_k + \tilde{B}\Delta u_k \tag{16b}$$

$$y_k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tilde{x}_k \tag{16c}$$

$$\Delta y = y - y_{ref} \tag{16d}$$

$$\Delta u_k = u_k - u_{k-1}, \tag{16e}$$

$$-0.005 \leq \Delta u_k \leq 0.005, \tag{16f}$$

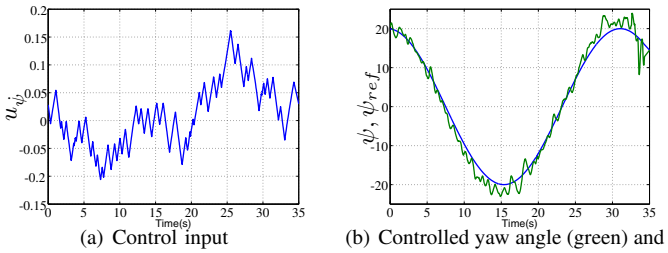
$$-1 \leq u_k \leq 1, \tag{16g}$$

$$\tilde{x}_{min} \leq \tilde{x}_k \leq \tilde{x}_{max} \tag{16h}$$

The model gains are $K_1 = 92.880$, $K_2 = 3.717$, while the state penalty matrix $Q_y = \left[\begin{smallmatrix} 500 & 0 \\ 0 & 2000 \end{smallmatrix}\right]$ and $Q_u = 1$ were chosen experimentally. The weights were adjusted to put emphasis to ψ and slightly restrict $\dot{\psi}$. The prediction horizon $N = 5$ and sampling time $T_s = 20$ms were used.

The explicit MPC feedback as in (6) was then generated using parametric solvers contained in MPT. The solution consisted of 281 regions. The controller was subsequently exported to Python code using the `toPython` function and embedded into the control software represented by the ROS platform.

The controller was tested in a real-time experiment with control period $\sim$ 20ms. The real control period varied from 15 to 27 milliseconds, which introduced additional disturbances into the control loop. The variation of the control period was caused mostly by the varying evaluation time of the sequential search. A Kalman filter with time-varying gain matrix was applied for estimating ψ and $\dot{\psi}$. Results of the tracking experiment are shown in Figs. 2(a) and 2(b). As can be seen, the explicit MPC controller rejects disturbances and achieves the tracking objective by bringing ψ to a harmonic reference. The small velocity oscillations were caused by air turbulence and varying control period.



(a) Control input

(b) Controlled yaw angle (green) and the reference (blue)

Fig. 2. AR Drone example.

Acknowledgments

The authors gratefully acknowledge the contribution of the Scientific Grant Agency of the Slovak Republic under the grant 1/0403/15, and the financial support of the Slovak Research and Development Agency under the project APVV 0551-11 and the internal grants of the Slovak University of Technology in Bratislava for support of young researchers and STU Grant scheme for Support of Excellent Teams of Young Researchers.

References

[1] M. Baotić. *Optimal Control of Piecewise Affine Systems - a Multi-parametric Approach*. Dr. sc. thesis, ETH, Zurich, 2005.

[2] A. Bemporad, M. Morari, V. Dua, and E. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(2):3–20, 2002.

[3] L.T. Biegler and V.M. Zavala. Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization. *Computers and Chemical Engineering*, 33(3):575–582, 2009.

[4] F. Borrelli. *Constrained Optimal Control of Linear and Hybrid Systems*, volume 50. Springer, July 2003.

[5] P.J. Bristeau, F. Callou, D. Vissière, and N. Petit. The Navigation and Control technology inside the AR . Drone micro UAV. In *Proceedings of the 18th IFAC World Congress, 2011*, volume 18, pages 1477–1484, 2011.

[6] E. Camacho. *Predictive control with constraints*, volume 39. Prentice-Hall, 2003.

[7] J. Engel, J. Sturm, and D. Cremers. Camera-based navigation of a low-cost quadrocopter. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2815–2821, 2012.

[8] P.E. Gill, W. Murray, and M.A. Saunders. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Journal on Optimization*, 12(4):979–1006, 2002.

[9] M. Herceg, S. Mariethoz, and M. Morari. Evaluation of piecewise affine control law via graph traversal. In *European Control Conference*, pages 3083–3088, 2013.

[10] M. Herceg, M.Kvasnica, C. Jones, and M. Morari. Multi-Parametric Toolbox 3.0. In *Proceedings of the European Control Conference*, pages 502–510, 2013.

[11] A. Joel. *A General-Purpose Software Framework for Dynamic Optimization*. Dr. sc. thesis, KU Leuven, 2013.

[12] B.P. Gerkey M. Quigley, K. Conley and J. Faust. ROS: an open-source Robot Operating System. In *Proceedings of ICRA Workshop on Open Source Software*, 2009.

[13] P. Tøndel, T.A. Johansen, and A. Bemporad. Evaluation of piecewise affine control via binary search tree. *Automatica*, 39(5):945–950, 2003.

[14] S.J. Qin and T.A. Badgwell. An Overview of Industrial Model Predictive Control Technology. *J. C. Kantor, C. E. Garcia and B. Carnahan (Eds.) 5th International Conference on Chemical Process Control*, 93(316):232–256, 1997.

[15] B. Takács, J. Holaza, J. Števek, and M. Kvasnica. Export of explicit model predictive control to python. In *Proceedings of the 20th International Conference on Process Control*, pages 78–83, 2015.

[16] Y. Wang, C. Jones, and J. Maciejowski. Efficient point location via subdivision walking with application to explicit MPC. In *Proc. European Control Conf.*, pages 447–453, 2007.