2011 50th IEEE Conference on Decision and Control and
European Control Conference (CDC-ECC)
Orlando, FL, USA, December 12-15, 2011

A Memory-Efficient Representation of Explicit MPC Solutions

Alexander Szücs, Michal Kvasnica, and Miroslav Fikar

***Abstract*— Amount of memory needed to describe explicit model predictive control (MPC) solutions is an often neglected, but a very important factor which decides whether it will be possible to implement such a control strategy on a selected control platform. We show how to exploit geometric properties of explicit MPC controllers to obtain their memory-efficient representation. The three-layer procedure first identifies similarities between polytopic regions in form of an affine transformation. If such a mapping exists, certain regions can be represented using less data. The second layer then applies data de-duplication to identify and remove repeating sequences of data. Regions are then described by integer pointers to such a unique set. Finally, Huffman encoding is applied to compress such integer pointers using prefix-free variable-length bit encoding. Reduction in memory is traded for an increase in evaluation time, which is quantified for each layer. Main advantage of the overall procedure is that it can be applied on top of most existing complexity reduction schemes available in the literature.**

I. Introduction

As shown in [2], the effort of implementing MPC in the Receding Horizon fashion (RHMPC) can be substantially reduced by pre-computing the optimal control action for all possible initial conditions as a function κ. For a large class of MPC problems, such a function can be shown to take a form of a Piecewise Affine (PWA) function, which is composed of a set of polytopic regions and the associated affine feedback expressions. The main benefit is that obtaining the optimal control input at each sampling instance reduces to a mere function evaluation, which can be performed efficiently even on simple control devices in a matter of milli- and microseconds.

On the other hand, to achieve such a simple and fast implementation, all pre-computed data have to be stored in the memory of the target control hardware. Although this aspect is often neglected in the literature, in fact it plays a prominent role when implementing explicit MPC solutions on devices with low available memory storage. Typical examples include programmable logic controllers (PLCs) and embedded microchips, which are one of the most frequently used types of industrial control platforms. Such devices usually only provide 2-8 kilobytes of memory capacity, a figure which represents a significant challenge in explicit MPC. Needless to say, unless all pre-computed data can be fit into memory, the controller cannot be implemented in practice. Therefore it is of imminent importance to keep the memory footprint $\mathcal{S}(\kappa)$ on an acceptable level.

Authors are with Slovak University of Technology in Bratislava; {alexander.szucs, michal.kvasnica, miroslav.fikar}@stuba.sk

The memory size of explicit MPC solutions is primarily determined by the number of regions R and by their complexity. The problem of reducing $\mathcal{S}(\kappa)$ is usually tackled by approximating the optimal RHMPC feedback, or the optimal value function in such a way that a less complex, albeit suboptimal, feedback function $\tilde{\kappa}$ is obtained, see e.g. [1], [11], [5], [7] and references therein.

In this paper, instead of decreasing $\mathcal{S}(\kappa)$ by reducing the number of regions, we look for a memory efficient representation of κ which requires less data. The procedure consists of three layers. The first one determines a subset of regions which can be obtained by applying affine transformation of the remaining regions. We show how to formulate the search for such a mapping by solving a mixed-integer problem, which is done off-line. If the transformation exists, the corresponding regions can then be represented using less data. The second layer can either be applied on top of the first one, or independently. Here, memory is saved by identifying positive and negative duplicities in half-space representation of several polytopic regions. The duplicate occurrences are then represented as mere integer pointers to the unique set of data. Compared to the first layer, the additional computation to be performed on-line is much smaller. Finally, in the last layer we propose to compress the integer pointers by Huffman encoding [6]. Here, variable-length bit codewords are assigned to each integer, depending on its frequency of abundance. Main benefit of the proposed strategies is that they can be applied on top of all aforementioned complexity reduction schemes. Saving in terms of memory is achieved at the price of an increase of the implementation effort performed on-line. Therefore the approach is mainly suited for situations where the implementation device poses enough computational power, but has severe memory limitations.

II. Explicit Model Predictive Control

We consider the class of constrained discrete-time, stabilizable linear time-invariant systems

$$x^+ = \Gamma x + \Xi u,\ x \in \mathcal{X},\ u \in \mathcal{U}, \tag{1}$$

where $x \in \mathbb{R}^{n_x}$ is the state vector, x^+ is the successor state, $u \in \mathbb{R}^{n_u}$ is the vector of control inputs, and $\mathcal{X} \subseteq \mathbb{R}^{n_x}$, $\mathcal{U} \subseteq \mathbb{R}^{n_u}$ are given polytopic sets. For system (1) we define the constrained finite-time optimal

978-1-61284-799-3/11/$26.00 ©2011 IEEE

control problem:

$$\min_{U_N} \quad \sum_{k=0}^{N-1} \|Q_x x_{k+1}\|_p + \|Q_u u_k\|_p \tag{2a}$$

$$\text{s.t.} \quad x_{k+1} = \Gamma x_k + \Xi u_k,\ x_{k+1} \in \mathcal{X},\ u_k \in \mathcal{U} \tag{2b}$$

where x_k and u_k denote, respectively, the state and input predictions at time instance k, initialized by the measurements of the current state x_0. The prediction is carried out over a finite prediction horizon N. The explicit representation of the receding horizon MPC feedback $u^* = [I\ 0\ \cdots\ 0]U_N^*$ can be found as a PWA function of the initial condition x by solving (2) as a parametric program:

Theorem 2.1 ([2]): The RHMPC feedback u^* for problem (2) with $p \in \{1, 2, \infty\}$ is given by

$$u^* = \kappa(x) := \begin{cases} F_1 x + G_1 & \text{if } x \in \mathcal{R}_1 \\ \vdots & \\ F_R x + G_R & \text{if } x \in \mathcal{R}_R, \end{cases} \tag{3}$$

where:

- $\kappa : \mathbb{R}^{n_x} \to \mathbb{R}^{n_u}$ is a continuous PWA function;
- $\mathcal{R}_i = \{x \mid H_i x \leq K_i\}$ are polytopes with $H_i \in \mathbb{R}^{c_i \times n_x}$, $K_i \in \mathbb{R}^{c_i}$, $i = 1, \ldots, R$;
- the set of feasible initial conditions $\Omega := \{x \mid \exists u_0, \ldots, u_{N-1} \text{ s.t. (2b) holds}\}$ is a convex polytope;
- $\{\mathcal{R}_i\}_{i=1}^R$ is a partition of Ω, i.e. $\cup_i \mathcal{R}_i = \Omega$ and $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ for all $i \neq j$.

■

The advantage of such an explicit representation is obvious: obtaining the optimal control action for a given x reduces to a mere evaluation of the function κ, which is a two-stage process. In the first step, index i of the region which contains the state measurements is to be identified. This problem is referred to as the point location problem [10]. Then, in the second step, the optimal control action is computed by evaluating $u^* = F_i x + G_i$. The point location problem can be solved e.g. by traversing the regions sequentially according to Algorithm 1 (its output is $\emptyset$ if $x \notin \cup_i \mathcal{R}_i$, in which case there is no feasible u which would guarantee satisfaction of constraints in (2b)). The crucial downside of the explicit MPC approach, however, is that the number of regions tends to be large, often above the limits of typical control hardware implementation platforms. Specifically, the amount of memory needed to execute Algorithm 1 on-line at each sampling instant, expressed as the number of floating-point numbers, is

Algorithm 1 Point location

```
for i = 1, ..., R do
    if H_i x ≤ K_i then
        return i
    end if
end for
```

$$\mathcal{S}(\mathcal{R}_i) = \sum_{i=1}^{R} c_i(n_x + 1), \tag{4}$$

where R is the number of regions and c_i is the number of defining half-spaces of the i-th region. Clearly, as R increases, and as the regions become more complex (i.e. with growing c_i), the memory footprint of κ can easily exceed the provided memory capacity. Therefore, when targeting implementation devices with low memory storage, it is important to devise a more memory-efficient representation of the feedback law κ.

III. Main Results

In this section we show how to represent regions $\mathcal{R}_i$ more efficiently by exploiting their geometric properties. Each of the proposed three layers can be viewed at as a "compression" mechanism. Needless to say, additional computational effort needs then to be performed on-line to "decompress" the data. We provide quantification of such an additional effort as a function of the problem size. Decompression is performed on-the-fly on a region-by-region basis.

Only the polytopic nature of regions $\mathcal{R}_i$ is exploited by the proposed complexity reduction procedure. Continuity of κ and convexity of the feasible set Ω are not required. Therefore the approach is applicable to generic PWA function κ defined over polytopes. The scope of this work therefore extends to scenarios where tracking of a non-zero reference is achieved by a suitable augmentation of the state vector, or where linear hybrid systems are used as prediction models. For the same reason the procedure can be applied to post-process RHMPC feedback laws generated by other complexity reduction schemes, e.g. those reviewed in [7].

To quantify achievable reduction in memory, we will assume that double-precision floating point numbers consume 8 bytes, while integers can be represented by 2 bytes. Each individual mathematical operation on a float or on an integer will be denoted as one FLOP.

A. Complexity Reduction via Affine Transformations

First we show how to represent some regions using less data by exploiting geometric similarities of such polytopes. We remind that the memory footprint of a region $\mathcal{R}_j = \{x \mid H_j x \leq K_j\}$ with $H_j \in \mathbb{R}^{c_j \times n_x}$ and $K_j \in \mathbb{R}^{c_j}$ is $c_j(n_x + 1)$ real numbers with $c_j \geq n_x + 1$. Here, we look for affine transformations $A_{i,j}x + b_{i,j}$ such that

$$\mathcal{R}_i = \{A_{i,j}x + b_{i,j} \mid x \in \mathcal{R}_j\}. \tag{5}$$

If there exist $A_{i,j} \in \mathbb{R}^{n_x \times n_x}$ and $b_{i,j} \in \mathbb{R}^{n_x}$ which map $\mathcal{R}_j$ onto $\mathcal{R}_i$, then the memory footprint of κ is reduced as follows: for each i, j for which the mapping exists, the half-space representation of the j-th region (i.e. matrices H_j, K_j with variable number of rows c_j) can be replaced

by matrices $A_{i,j}$, $b_{i,j}$ with fixed number of rows n_x. Then, once $x \in \mathcal{R}_j$ is to be verified in Step 2 of Alg. 1, it suffices to check whether $A_{i,j}x + b_{i,j} \in \mathcal{R}_i$, i.e.

$$x \in \mathcal{R}_j \;\Leftrightarrow\; A_{i,j}x + b_{i,j} \in \mathcal{R}_i. \tag{6}$$

It follows that memory footprint of region $\mathcal{R}_j$ is reduced by $(c_j - n_x)(n_x+1)$ real numbers by only storing $A_{i,j}, b_{i,j}$ instead of H_j, K_j. Since $c_j \gg n_x + 1$ in practice, a significant reduction can be achieved.

Definition 3.1: Let the polytopic partition $\{\mathcal{R}_i\}_{i=1}^R$ be given. The index set $\mathcal{I}_G \subseteq \{1, \ldots, R\}$ is called the index set of *generating regions* of the partition if for each $j \notin \mathcal{I}_G$ there exists an $i \in \mathcal{I}_G$ and the associated affine map $A_{i,j}x + b_{i,j}$ such that (5) holds.

Fix any i–j combination with $i \neq j$. Then the parameters of the associated affine transformation in (5) can be determined by solving a mixed-integer program, as summarized by the following theorem, which is our first main result.

Theorem 3.2: Let i, j be given and let $\mathcal{V}_i = [v_{i,1}, \ldots, v_{i,n_v}]$ and $\mathcal{V}_j = [v_{j,1}, \ldots, v_{j,n_v}]$ denote, respectively, the extremal vertices of $\mathcal{R}_i$ and $\mathcal{R}_j$. Then an affine transformation which guarantees (6) exists if there exist $A_{i,j} \in \mathbb{R}^{n_x \times n_x}$, $b_{i,j} \in \mathbb{R}^{n_x}$, and a binary permutation matrix $P \in \{0, 1\}^{n_v \times n_v}$ with $\sum_{m=1}^{n_v} P_{m,k} = 1$, $\forall k$, $\sum_{m=1}^{n_v} P_{k,m} = 1$, $\forall k$ such that

$$\begin{bmatrix} A_{i,j} & b_{i,j} \end{bmatrix} \begin{bmatrix} \mathcal{V}_j \\ 1 \end{bmatrix} = \mathcal{V}_i P. \tag{7}$$

■

Problem (7) is a feasibility problem with real variables $A_{i,j}$, $b_{i,j}$ and binary variables P, which can be solved by off-the-shelf software, like GLPK [9] or CPLEX [4].

The index set of generating regions can be determined by 2, which requires solving, at most, $1/2 R(R-1)$ MIP problems (7) on Step 4. In practice, it will be less, since only regions with the same number of vertices need to be processed. The algorithm returns an auxiliary array $\mathcal{J}$ which denotes feasible i–j combinations. If $\mathcal{J}_j \neq \emptyset$ for some j, then $\mathcal{J}_j$ points to its associated generating region. If $\mathcal{J}_j = \emptyset$, then $\mathcal{R}_j$ is a generating region on its own, i.e. $\mathcal{I}_G = \{j \mid \mathcal{J}_j = \emptyset\}$.

Algorithm 2

```
1: Initialize J_j = ∅, A_j = ∅, B_j = ∅, j = 2, ..., R
2: for i = 1, ..., R − 1 do
3:    for j = i + 1, ..., R do
4:       if J_j = ∅ and (7) is feasible then
5:          A_j ← A_{i,j}, B_j ← b_{i,j}, J_j ← i
6:       end if
7:    end for
8: end for
```

Given the arrays of affine transformations $\mathcal{A}$ and $\mathcal{B}$, the point-location task can be implemented by Algorithm 3. The size of its input arguments is

$$\mathcal{S}(\mathcal{R}_i) = \sum_{i \in \mathcal{I}_G} c_i(n_x + 1) + \sum_{i \notin \mathcal{I}_G} n_x(n_x + 1), \tag{8}$$

a reduction by $\sum_{i \notin \mathcal{I}_G}(c_i - n_x)(n_x + 1)$ floating point numbers compared to the standard approach, cf. (4). The memory saving is hence proportional to the number of non-generating regions. The algorithm loops through regions sequentially. If a generating region is encountered, $x \in \mathcal{R}_i$ is checked directly. Otherwise, (6) is exploited and $\mathcal{A}_j x + \mathcal{B}_j \in \mathcal{R}_i$ is checked instead. Saving in terms of memory is traded for an increase in execution time. Here, compared to Algorithm 1, one needs to evaluate the affine transformations whenever a non-generating region is encountered, which requires $\sum_{i \notin \mathcal{I}_G}(2n_x^2)$ FLOPs in the worst case.

Algorithm 3

```
1: for j = 1, ..., R do
2:    if j ∈ J then
3:       i ← J_j, x ← A_i x + B_i
4:    else
5:       i ← j
6:    end if
7:    if H_i x ≤ K_i then
8:       return j
9:    end if
10: end for
```

Example 3.3: Consider a double integrator sampled at 1 second, given by the following state-space representation:

$$x^+ = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} u, \tag{9}$$

where the states and inputs are constrained, respectively, by $|x_i| \leq 5$, $i = 1, 2$, and $|u| \leq 1$. With the choice of $p = 1$, $Q_x = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$, $Q_u = 1$, and $N = 10$ in (2), the explicit RHMPC feedback law consists of 230 regions, shown in Figure 1. Storing all regions would require 2466 floating point numbers, or 19 kilobytes. Algorithm 2 has found feasible affine transformations for 198 regions, representation of which can be simplified by only storing $A_{i,j}$ and $b_{i,j}$. Remaining 32 generating regions need to be represented using the full data, i.e. by matrices H_i, K_i. Here, the 198 affine transformations contribute by 1188 numbers, while the 32 regions require 402 floats. It follows that the total required memory is decreased from 19 to 12 kilobytes. The worst-case number of FLOPs[1] needed to perform point location via Algorithm 3 is 6143 compared to 4110 operations for Algorithm 1.

[1]It is worth noting that even slow CPUs typically found in industrial control hardware are able to perform tens of millions of FLOPs per second. With the reported computational figures the control algorithm could therefore be executed at the sampling range of hundreds of kilohertz.

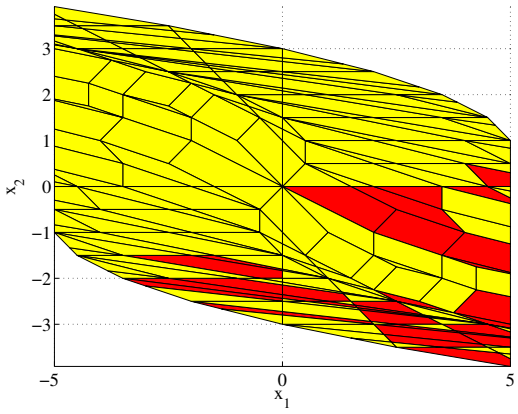

Fig. 1. Regions of the explicit MPC solution for Example 3.3. Each of the 198 yellow regions can be obtained by applying a suitable affine transformation (6) to one of the 32 generating regions, shown in red.

B. Data De-Duplication

Instead of having to store all data (i.e. all matrices H_i and K_i), one can use de-duplication to first identify unique rows of $H = \left[\begin{smallmatrix} H_1^T & \cdots & H_R^T \end{smallmatrix}\right]^T \in \mathbb{R}^{m \times n_x}$ and $K = \left[\begin{smallmatrix} K_1^T & \cdots & K_R^T \end{smallmatrix}\right]^T \in \mathbb{R}^m$ with $m = \sum_i^R c_i$. Denote the unique rows by $\mathcal{H}$ and $\mathcal{K}$. If cardinality of $\mathcal{H}$ ($\mathcal{K}$) is smaller than number of rows in H (K), then the amount of memory can be significantly decreased by storing, for each region, only the pointers to $\mathcal{H}$ and $\mathcal{K}$. The saving is twofold. First, memory size of a pointer is smaller than for a floating point number. Second, since a single pointer is assigned to each row of $\mathcal{H}$ (which is n_x dimensional), the amount of memory is decreased n_x times for each entry.

As an example, consider three regions given in their respective half-space representation by

$$H_1 = \begin{bmatrix} 0 & 1 \\ 1 & -0.5 \\ 0 & -1 \\ -1 & 0.5 \end{bmatrix},\ H_2 = \begin{bmatrix} 0 & 1 \\ 1 & -0.5 \\ -1 & 0.5 \\ 0 & -1 \end{bmatrix},\ H_3 = \begin{bmatrix} -1 & 0.5 \\ 0 & -1 \\ 0 & 1 \\ 1 & -0.5 \end{bmatrix},$$
$$K_1 = \begin{bmatrix} 2.4 \\ 3.1 \\ -1.5 \\ 0 \end{bmatrix},\ K_2 = \begin{bmatrix} 2.4 \\ 5.0 \\ -3.1 \\ 0 \end{bmatrix},\ K_3 = \begin{bmatrix} 0 \\ 0 \\ 1.5 \\ 3.1 \end{bmatrix}.$$

The sets of unique rows are

$$\mathcal{H} = \{\left[\begin{smallmatrix} 0 & 1 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 & -0.5 \end{smallmatrix}\right], \left[\begin{smallmatrix} 0 & -1 \end{smallmatrix}\right], \left[\begin{smallmatrix} -1 & 0.5 \end{smallmatrix}\right]\},$$
$$\mathcal{K} = \{-3.1,\ -1.5,\ 0,\ 1.5,\ 2.4,\ 3.1,\ 5.0\}.$$

The corresponding (unsigned) index set representation of the polytopic regions is then

$$\mathcal{I}_{H_1} = \{1,\ 2,\ 3,\ 4\},\ \mathcal{I}_{K_1} = \{5,\ 6,\ 2,\ 3\},$$
$$\mathcal{I}_{H_2} = \{1,\ 2,\ 4,\ 3\},\ \mathcal{I}_{K_2} = \{5,\ 7,\ 1,\ 3\},$$
$$\mathcal{I}_{H_3} = \{4,\ 3,\ 1,\ 2\},\ \mathcal{I}_{K_3} = \{3,\ 3,\ 4,\ 6\},$$

where each element of $\mathcal{I}_H$ and $\mathcal{I}_K$ points to the corresponding entry in $\mathcal{H}$ and $\mathcal{K}$.

Cardinality of $\mathcal{H}$ and $\mathcal{K}$, and hence the required storage space, can be further reduced by eliminating entries which are negations of others, i.e.

$$\mathcal{H} = \{\left[\begin{smallmatrix} 0 & 1 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 & -0.5 \end{smallmatrix}\right]\},\ \mathcal{K} = \{-3.1,\ -1.5,\ 0,\ 2.4,\ 5.0\}.$$

Then the (signed) index set representation becomes

$$\mathcal{I}_{H_1} = \{1,\ 2,\ -1,\ -2\},\ \mathcal{I}_{K_1} = \{4,\ -1,\ 2,\ 3\},$$
$$\mathcal{I}_{H_2} = \{1,\ 2,\ -2,\ -1\},\ \mathcal{I}_{K_2} = \{4,\ 5,\ 1,\ 3\}, \quad (10)$$
$$\mathcal{I}_{H_3} = \{-2,\ -1,\ 1,\ 2\},\ \mathcal{I}_{K_3} = \{3,\ 3,\ -2,\ -1\},$$

In this simple example, memory footprint of regions $\mathcal{R}_i$ was reduced from 36 floating point numbers representing matrices H_i, K_i to 9 floats for $\mathcal{H}$, $\mathcal{K}$, and 24 integer pointers. Assuming that one float is represented by 8 bytes and an integer by 2 bytes, de-duplication reduces required memory from 288 bytes to 120 bytes.

Algorithms 1 and 3 can be easily accommodated to exploit the signed index set representation. Whenever $H_i x \leq K_i$ needs to be checked, one constructs, on-the-fly, matrices H_i and K_i by $H_i = \{\text{sign}(j)\mathcal{H}_{|j|} \mid j \in \mathcal{I}_{H_i}\}$ and $K_i = \{\text{sign}(j)\mathcal{K}_{|j|} \mid j \in \mathcal{I}_{K_i}\}$. This involves negating the corresponding rows, depending on the sign of the index. Therefore execution of Algorithms 1 and 3 requires, at most, $\sum_{i=1}^R 2c_i$ additional FLOPs.

Example 3.4: We revisit Example 3.3 and remind that the full set of 230 regions can be equivalently represented by 32 generating regions and by 198 associated affine transformations (5). The generating regions are described by 134 half-spaces, which require $134(n_x + 1)$ floating point numbers ($n_x = 2$ in this example). Here, the sets of rows unique under unity scaling, i.e. $\mathcal{H}$ and $\mathcal{K}$, only contains 17 and 47 entries, respectively, which is equivalent to $17n_x + 47$ floating point numbers. The corresponding index sets $\mathcal{I}_{H_i}$ and $\mathcal{I}_{K_i}$ contribute by 2×134 integers. After de-duplication is applied to $\mathcal{A}_j$ and $\mathcal{B}_j$ as well, the total memory footprint is reduced from 12 kilobytes reported in Example 3.3 to just 7.5 kilobytes. This comes at the expense of performing additional 1644 FLOPs to reconstruct the regions on-the-fly using index set representations.

C. Compression of Index Set Representations

Given are index set representations $\mathcal{I}_H = \cup_i \mathcal{I}_{H_i}$ and $\mathcal{I}_K = \cup_i \mathcal{I}_{K_i}$, whose entries point to corresponding rows in the set of unique elements $\mathcal{H}$ and $\mathcal{K}$. In traditional implementation, each element of $\mathcal{I}_H$ and $\mathcal{I}_K$ would need to be represented as a (signed) integer, i.e. by 16 bits (provided that cardinality of $\mathcal{H}$ and $\mathcal{K}$ does not exceed 2^{16}). A more efficient representation can be obtained by using a *prefix-free variable-length encoding* where bit-wise codewords are assigned to each element of the index sets. Length of a codeword is inverse-proportional to its abundance, such that size of $\mathcal{I}_K$ and $\mathcal{I}_H$ is compressed as much as possible.

Proposition 3.5 ([3]): Given an index set $\mathcal{I}$ and an array of frequencies $\mathcal{F}$, Algorithm 4 generates an optimal coding tree $\mathcal{T}(\mathcal{I})$ as a full binary tree where the symbols to encode are at the leaves, and where each codeword is generated by a path from root to leaf, interpreting left traversal as 0 and right as 1.

As an illustration, consider the index sets in (10) and let $\mathcal{I}_K = \mathcal{I}_{K_1} \cup \mathcal{I}_{K_2} \cup \mathcal{I}_{K_3}$. Then the integers to

TABLE I

FREQUENCIES OF INTEGERS TO ENCODE.

Integer	−2	−1	1	2	3	4	5
Frequency	1	2	1	1	4	2	1

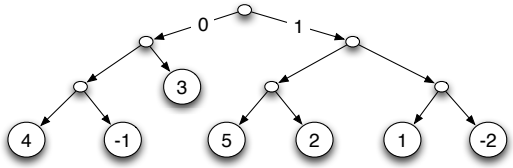

Fig. 2. Huffman tree for $\mathcal{I}_K = \{4, -1, 2, 3, 4, 5, 1, 3, 3, 3, -2, -1\}$.

encode appear with frequencies reported in Table I. The corresponding Huffman tree is shown in Figure 2. Here, the optimal codewords are $\mathcal{C}(\mathcal{I}_K) = \{[-2 : 111], [-1 : 001], [1 : 110], [2 : 101], [3 : 01], [4 : 000], [5 : 100]\}$. It is easy to verify that such an encoding is prefix-free, i.e. that no codeword is a prefix of another codeword. Moreover, the most abundant integer 3 is encoded using fewest number of bits as to minimize the total length of binary representation of $\mathcal{I}_K$. Hence, instead of storing $\mathcal{I}_K$ as an array of 12 integers (i.e., 24 bytes), it suffices to store the tree (7 integers or 14 bytes) and 12 codewords of a total size 32 bits, or 4 bytes.

Algorithm 4 Huffman encoding [3]

1: Let $\mathcal{Q}$ be a priority queue, ordered by positive frequencies $\mathcal{F} = [f_1, \dots, f_n]$
2: **for** $k = n+1, \dots, 2n-1$ **do**
3: $i \leftarrow \texttt{deletemin}(\mathcal{Q})$, $j \leftarrow \texttt{deletemin}(\mathcal{Q})$
4: Create a node k with children i, j
5: $\mathcal{F}_k \leftarrow \mathcal{F}_i + \mathcal{F}_j$
6: $\texttt{insert}(\mathcal{Q}, k)$
7: **end for**

Size of the tree is proportional to the number of unique elements of the encoded set of integers $\mathcal{I}$. Decoding of a particular sequence of bits boils down to traversing the tree until a leaf is reached, whereupon the tree returns to its root. Decompression effort is therefore proportional, in the worst case, to the length m of the longest codeword. In total, $\sum_{i=1}^{R} 2c_i m$ operations are required to reconstruct regions $\mathcal{R}_i$ on-the-fly from their respective encoded index set representations.

Remark 3.6: Traversing the tree only requires performing bit-wise operations, which are much cheaper than multiplications or additions on floating point numbers. Therefore a mere increase in FLOPs by a factor of n does not necessarily mean that evaluation speed would drop n times. In practice, it will be less.

Example 3.7: We continue with Example 3.4 where it was shown that 134 signed integers $\mathcal{I}_H$ pointing to one of the 17 unique rows of $\mathcal{H}$, and 134 signed integers $\mathcal{I}_K$ for the 47 unique elements of $\mathcal{K}$ are required for the index set representation of generating regions. The trees $\mathcal{T}(\mathcal{I}_H)$ and $\mathcal{T}(\mathcal{I}_K)$ were build by Algorithm 4 in 0.05 seconds. The trees had 26 and 63 leave nodes, respectively. Each element of $\mathcal{I}_H$, $\mathcal{I}_K$ was encoded as a prefix-free sequence of bits. For $\mathcal{I}_H$, the minimal codeword length was 3, the maximal was 6. For $\mathcal{I}_K$, the minimal and maximal code lengths were 4 and 7, respectively. It follows that the index sets $\mathcal{I}_H$ and $\mathcal{I}_K$, which originally required 2×134 integers, can be equivalently represented by the two trees (which need 89 integers) and 2×134 bit sequences, which in total attribute by 527 bits, or 66 bytes. Therefore memory footprint of the index set representations is reduced from 536 bytes to 244 bytes. Decompression of the bit codewords in a suitable modification of Algorithm 3 would require additional 3570 FLOPs, in the worst case.

IV. Efficiency Evaluation

To asses efficiency of the proposed three-layer procedure on generic data, we have analyzed randomly-generated explicit RHMPC feedback laws for dimensions $2 \leq n_x \leq 5$. For each dimension, 20 random RHMPC feedback laws were generated by the MPT Toolbox [8]. Each controller was then processed by applying, consecutively, the similarity transformation of Section III-A, then de-duplication of Section III-B, followed by data compression of Section III-C.

For various state dimensions, Figure 3 shows achieved memory reduction factors, i.e. the ratios between memory size of the original solution and the corresponding compression layer. Note that the figures show accumulated data, i.e. improvement of a particular layer upon a previous one. The unity basis corresponds to size of the original, uncompressed, RHMPC solution. As can be observed, reduction of memory size by a factor of 20 is not unusual. The average values are also summarized in Table II. As expected, the compression factors increase with growing number of states. This trend is mostly notable for the de-duplication and compression methods.

Results in Figure 4 then quantify the factor by which the number of floating point operations increases in order to "decompress" a particular layer, with Algorithm 1 being the basis. However, as noted in Remark 3.6, this factor is not directly proportional to a slowdown in evaluation speed when the Huffman encoding layer is concerned. Although the evaluation effort is substantially increased, it is always out-weighted by a more substantial reduction in terms of memory. With growing problem dimension and number of regions, complexity of Algorithm 1 naturally increases. It is due to this fact that the relative factors in Figure 4 actually tend to improve when n_x is enlarged.

V. Conclusions

We have shown how to decrease memory requirements for implementation of explicit MPC solutions by applying three compression-like approaches. First, mixed-integer

TABLE II

AVERAGE ACCUMULATED COMPRESSION FACTORS.

n_x	Sec. III-A	Sec. III-B	Sec. III-C
2	1.5	4.3	8.2
3	1.3	5.9	13.7
4	1.7	8.1	24.6
5	1.5	10.4	43.2

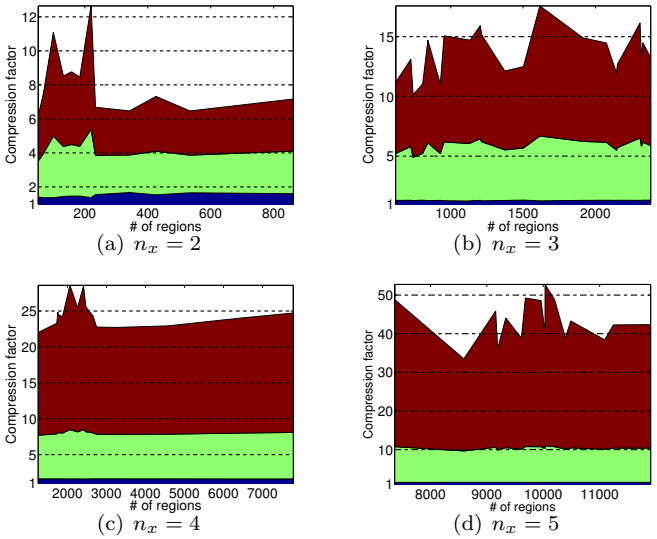

Fig. 3. Accumulated reduction in memory storage achieved by individual layers (blue is for the similarity transformation, green for de-duplication, and red for compression). Note that individual figures have different scales on the y axis.

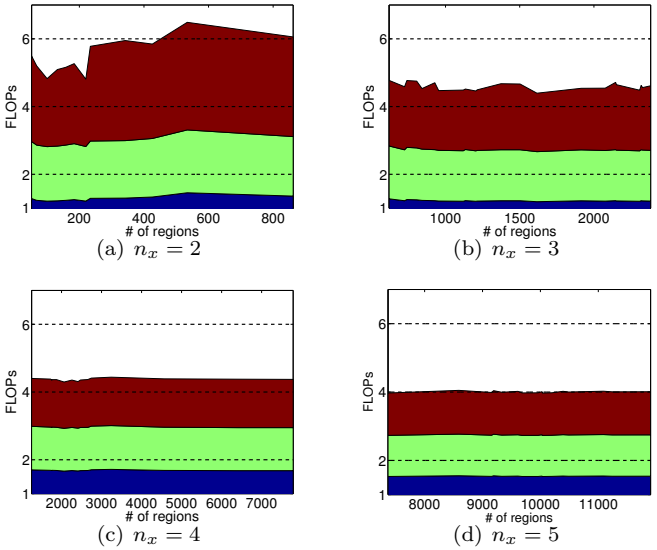

Fig. 4. Accumulated increase in on-line computation needed to implement a particular layer (blue is for the similarity transformation, green for de-duplication, and red for compression).

programming was used to derive suitable affine transformations which allow certain regions to be represented using fixed amount of data. Then, de-duplication was utilized to identify a unique subset of data and converting the regions into index set representations. Finally, the integer indices were compressed by Huffman encoding. By means of a large case study we have demonstrated that a significant memory saving can be achieved. This reduction comes at the price of having to perform additional computation on-the-fly, amount of which was quantified for each level. Efficiency of de-duplication and compression increases with growing problem dimension, which is due to the fact that regions become more complex.

ACKNOWLEDGMENTS

The authors are pleased to acknowledge the financial support of the Scientific Grant Agency of the Slovak Republic under the grant 1/0095/11. This work was supported by the Slovak Research and Development Agency under the contracts No. VV-0029-07 and No. LPP-0092-07. Supported by a grant (No. NIL-I-007-d) from Iceland, Liechtenstein and Norway through the EEA Financial Mechanism and the Norwegian Financial Mechanism. This project is also co-financed from the state budget of the Slovak Republic and from the internal grant of the Slovak University of Technology in Bratislava for support of young researchers.

REFERENCES

[1] A. Bemporad and C. Filippi. Suboptimal explicit RHC via approximate multiparametric quadratic programming. *Journal of Optimization Theory and Applications*, 117(1):9–38, April 2003.

[2] 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.

[3] S. Dasgupta, Ch. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 1 edition, September 2006.

[4] ILOG, Inc. *CPLEX User Manual*. Gentilly Cedex, France. http://www.ilog.fr/products/cplex/.

[5] C.N. Jones and M. Morari. Approximate Explicit MPC using Bilevel Optimization. In *European Control Conference*, Budapest, Hungary, August 2009.

[6] Donald E. Knuth. Dynamic huffman coding. *J. Algorithms*, 6(2):163–180, 1985.

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

[8] M. Kvasnica, P. Grieder, and M. Baotić. Multi-Parametric Toolbox (MPT), 2004. Available from http://control.ee.ethz.ch/~mpt/.

[9] A. Makhorin. *GLPK - GNU Linear Programming Kit*, 2001. http://www.gnu.org/directory/libs/glpk.html.

[10] J. Snoeyink. Point Location. In J. E. Goodman and J. O'Rouke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 558–574. CRC Press, Boca Raton, New York, 1997.

[11] A. Ulbig, S. Olaru, D. Dumur, and P. Boucher. Explicit solutions for nonlinear model predictive control: A linear mapping approach. In S. G. Tzafestas and P. J. Antsaklis, editors, *Proc. of of the European Control Conference 2007*, pages 3295–3302, 2007.