Using hash tables to manage time-storage complexity in point location problem: Application to Explicit MPC ⋆

Farhad Bayat [a], Tor Arne Johansen [b], Ali Akbar Jalali [a]

[a] *Department of Electrical Engineering, Iran University of Science and Technology, Narmak, 1684613114 Tehran, Iran.*

[b] *Department of Engineering Cybernetics, Norwegian University of Science and Technology, N-7491 Trondheim, Norway.*

Abstract

The online computational burden of linear model predictive control (MPC) can be moved offline by using multi-parametric programming, so called explicit MPC. The solution to the explicit MPC problem is a piecewise affine (PWA) state feedback function defined over a polyhedral subdivision of the set of feasible states. The online evaluation of such a control law needs to determine the polyhedral region in which the current state lies. This procedure is called point location and its computational complexity is challenging and determines the minimum possible sampling time of the system. A new flexible algorithm is proposed which enables the designer to trade-off between time and storage complexities. Utilizing the concept of hash tables and the associated hash functions the proposed method solves an aggregated point location problem that overcomes prohibitive complexity growth with the number of polyhedral regions, while the storage-processing trade-off can be optimized via scaling parameters. The flexibility and power of this approach is supported by several numerical examples.

Key words: Point location, Explicit Model Predictive Control, Computational complexity, Data structures.

1 Introduction

Recently in [4], [3] and [18] it has been shown that utilizing multi-parametric programming, it is possible to solve the linear and quadratic constrained finite time optimal control (CFTOC) problems explicitly without online optimization. In [4] it has been proved that such a solution is a piecewise affine (PWA) function defined over a polyhedral subdivision of feasible states mapping the current state measurement to the optimal control value. Therefore, the online closed loop computation is simplified to a PWA function evaluation. The problem of determining the polyhedral region in which the current state lies, refers to as "point location", "region identification" or even "set membership" problem in the literature, [14] and [8]. Once the desired region is found, the associated affine optimal control law is evaluated and applied to the system, and this procedure is repeated at the next sampling time.

The point location problem requires careful attention in the explicit MPC since its complexity influences the hardware requirements in fast sampling applications.

In [19] an efficient solution was proposed where a binary search tree is constructed by dividing the polyhedral partitions using auxiliary hyper planes which can be searched in time logarithmic in the number of regions, resulting in significant computational gains. By exploiting the piecewise affinity of the convex value function for MPC with linear cost function, authors in [5] and then in [2] have shown that such a problem can be solved with no need to store the polyhedral partitions. In the case that the cost-function is linear the authors in [12] have shown that the point location problem can be posed as a weighted Voronoi diagram, which can be solved using an approximate nearest neighbor (ANN) algorithm (see [1] for details). In [15] reachability analysis has been used in order to solve a reduced point location problem instead of dealing with the entire feasible set. The application of bounding boxes is exploited in [7] combining with a particular search tree to solve the point location problem with low preprocessing computation. But the storage demand and online search time are still linear in the number of polyhedral regions in the original PWA functions. The subdivision walking idea has been presented in [20] for a special class of partitions called solution complex. The idea uses the concept of traveling from a predefined seed point in a known region, toward

⋆ This paper was not presented at any IFAC meeting.

Email addresses: fbayat@iust.ac.ir (Farhad Bayat), Tor.Arne.Johansen@itk.ntnu.no (Tor Arne Johansen), ajalali@iust.ac.ir (Ali Akbar Jalali).

the current query point by walking from one region to the next until the region of interest is found. Although the worst case complexity is still linear in the number of regions but the exact bound calculation for closed loop applications needs to generate a crossing tree which is computationally very expensive. The idea of [15] is used in [16] to simplify the reachability analysis by using some reference points. Although some improvement has been achieved in terms of computational complexity, but in the case of complicated partitions both [15] and [16] have limited applicability due to the reachability analysis required in the preprocessing. More recently, in [21] a lattice PWA representation of the explicit MPC solution obtained based on the multi-parametric programming which can save some online calculation and memory requirements. In [22] a procedure has been introduced to trade-off between the warm-start and online computational effort when a combination of explicit MPC and online iterative active set computation is used. In [6] an approximate solution for explicit MPC using set membership approximation has been introduced which is able to reduce complexity of the online implementation with the cost of sub-optimality.

This paper suggests a new two-stage algorithm to efficiently solve the point location problem focussing on the explicit MPC application. The first stage identifies a subset of polyhedral regions based on an artificial interval partition of each state space axis. Subsequently the result for each axis is combined using a set intersection algorithm. This leads to a reduced subset of polyhedral regions, containing the optimal one, which is explored by direct search in the second stage. The main distinguishing feature of the approach is that the trade-off between online processing complexity and memory is effectively parameterized via scaling parameters, such that the user can optimize the implementation for the given computer architecture to make efficient use of processing and memory resources or minimize hardware costs. Furthermore, the additional offline processing is modest compared to the alternatives. The first stage sub-problems are solved efficiently supported by one-dimensional hash tables, as an alternative to e.g. hyper-rectangles organized in a binary search tree [6],[11]. This is efficient with respect to preprocessing, which is known to be the limiting factor of point location based on binary search trees [19],[11] in high dimensions which is prohibitive in terms of preprocessing time and also do not have flexibility to manage limitations in online storage. It is emphasized that the proposed approach will find the exact solution, while alternatives such as [6],[22],[17] and [11] consider sub-optimal approximations to address implementation issues. The present approach does also not make any assumptions, such as continuity [20], [21] or linear cost [2],[5],[12]. Therefore this method can be applied for a general class of partitions including discontinuous and overlapped ones. Finally, the complexity of implementation of an active set solver [22] is avoided in the second stage such that the present method can be implemented in low cost hardware using fixed-point arithmetics.

2 Explicit Solution of MPC

Consider a discrete time LTI system with constraints:

$$\begin{aligned} & x_{t+1} = Ax_t + Bu_t \\ & y_t = Cx_t \\ & x_t \in \mathbf{X}_c := \{x \in R^n : x_{min} \leq x \leq x_{max}\}, \\ & u_t \in \mathbf{U}_c := \{u \in R^n : u_{min} \leq u \leq u_{max}\}, \end{aligned} \quad (1)$$

where $A \in R^{n \times n}, B \in R^{n \times m}$ and $C \in R^{p \times n}$.
The typical MPC optimization problem with horizon N can be written in the following compact matrix form [4]:

$$\begin{aligned} & J_N^*(x_t) = x_t^{'} Y x_t + \min_{U_N} \{\frac{1}{2} U_N^{'} H U_N + x_t^{'} F x_t\} \\ & \textit{subject to}: \\ & \quad x_{t+k} \in \mathbf{X}_c, \;\; u_{t+k-1} \in \mathbf{U}_c, \quad \forall\, k = 1, ..., N \\ & \quad x_{t+k+1} = Ax_{t+k} + Bu_{t+k}, \quad \forall\, k \geq 0 \end{aligned} \quad (2)$$

Where $U_N = [u_t^T, u_{t+1}^T, ...u_{t+N-1}^T]^T \in R^s, (s = mN)$ is the optimization vector and Y, H, F, G, W and E are constant matrices of appropriate dimension [4].
In [4] and [18] it has been shown that the optimal solution U_N^* is a piecewise affine function of the current state x_t which can be calculated offline. Once the optimization problem (2) is solved offline, the explicit MPC uses this solution in a receding horizon fashion in which $u_t^* = [I_m 0...0] U_N^* = f(x_t)$, where $f : X_f \rightarrow R^m$ is a continuous, PWA function on a polyhedral partition:

$$u_t^* = F_r x_t + G_r, \quad \forall x_t \in P_r,\; r = 1, ..., N_r \quad (3)$$

where $P_r = \{x \in R^n | H_r x \leq K_r\}$ is the r-th polyhedral region. F_r, G_r, H_r and K_r are constant matrices which can be computed explicitly for each region.

3 Point location problem

Problem 1 (Point location) *Let* $\mathrm{P} := \{P_1, P_2, ..., P_{N_r}\}$ *be an arbitrary partition where* P_i, $i = 1, ..., N_r$ *are convex polyhedra. Given a query point* $x \in R^n$, *find an appropriate integer number* $i\,(x) \in \{0, 1, ..., N_r\}$ *such that* $i(x) = 0$ *if* $x \notin \mathrm{P}$ *and* $1 \leq i\,(x) \leq N_r$ *if* $x \in P_{i(x)}$.

Once the Problem 1 is solved, the optimal control law can be calculated using (3) for $r = i(x)$. The simplest algorithm to solve Problem 1 for a query point x is to check $H_r x \leq K_r$ for all regions $r = 1, ..., N_r$ one after another until the region containing the point x is found, so called exhaustive or direct search. Unambiguously the worst case computational complexity of this inefficient method is linear in the number of polyhedral regions N_r. This paper substantially aims to introduce an efficient method to solve the point location problem while enabling the designer to manage the computational and storage complexities especially when the number of regions is large.

3.1 Efficient point location: Main idea

The main idea is to solve n_x one dimensional sub-problems and then aggregate the results using a discrete set intersection method to solve the original problem. An extremely efficient method with processing time of order $O(1)$ is proposed to solve the sub-problems. This method basically puts the advantages of a hashing method (see e.g. [8], [9]) into practice and it is shown that the proposed hash functions lead to the perfect hashing (Theorem 2). In the next step the results of sub-problems are combined using a particular set intersection method (see Alg. 3) which is executed in logarithmic time and results in a reduced set of polyhedral regions containing the optimal critical region. Finally the optimal region is identified using a direct search method.

To illustrate the idea consider an arbitrary two dimensional polyhedral partition (Fig.1). Let I_j be the maximum interval length related to the j-th axis (e_j), which is equal to the projection of the feasible set X_f on to the j-th basis vector e_j. So $I_1 = a_{max} - a_{min}$ and $I_2 = b_{max} - b_{min}$. Then $I = \{I_1, I_2\}$ is the set of all maximum interval lengths. Now introduce an integer scaling parameter $\varepsilon_j \geq 0$ denoting the interval resolution in the j-th axis, i.e. the I_j is divided to $n_j = 2^{\varepsilon_j}$ equally spaced sub-intervals. We further define $I_j^{\overline{k}_j} = [L_j^{\overline{k}_j}, U_j^{\overline{k}_j}]$ denoting the $\overline{k}_j - th$ sub-interval in X_j. As an example, consider an $X_f = (X_1, X_2)^T$. Let $PI_1(\overline{k}_1) = \{3, 4, 8, 9, 15, 21, 22, 24\}$ and $PI_2(\overline{k}_2) = \{5, 6, 7, 8, 9, 10\}$ denoting indices of those regions which are overlapped with intervals $I_1^{\overline{k}_1}$ and $I_2^{\overline{k}_2}$ respectively. If the given query point x belongs to both subintervals, then it suffices to explore the common index set, say $PL = PI_1(\overline{k}_1) \cap PI_2(\overline{k}_2) = \{8, 9\}$, to obtain the optimal region. This simple example reveals that the original problem with 25 regions is reduced to a problem with 2 candidate regions. In what follows, two main objectives are pursued. At first an approach is established to efficiently obtain the common set PL supported by theoretical analysis of the computational and storage complexities. The second goal is to provide a systematic method which enables the designer to trade-off between the complexities of preprocessing time, storage demands and online computation time in a simple practical way.

3.2 Offline computation

For brevity, consider the j-th axis e_j for which the offline procedure for generating a data structure to support online point location is stated in Alg. 1. Then the same algorithm can be used for the other axes. Also note that steps 3 and 4 in Alg.1 together are equal to check whether $inter(proj_{e_j}^{P_r}) \cap I_j^{k_j} \neq \emptyset$ or not. Where, $proj_{e_j}^{P_r}$ denotes the orthogonal projection of P_r onto the e_j.

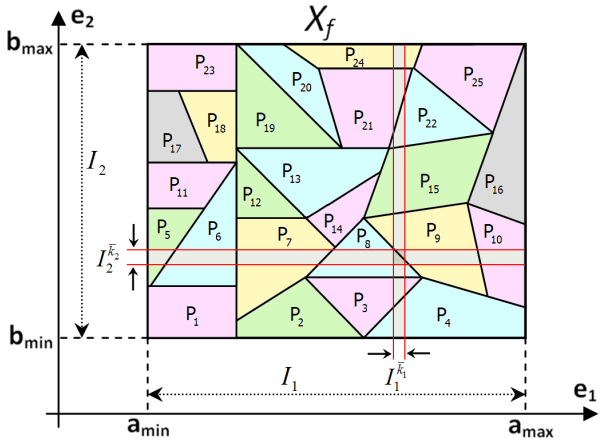

Fig. 1. Artificial 2D polyhedral partition.

Algorithm 1 : Offline computation

Step 1: Choose a suitable integer $\varepsilon_j \geq 0$.

Step 2: Divide the j-th interval I_j into the $n_j = 2^{\varepsilon_j}$ sub-intervals $I_j^{k_j} = [L_j^{k_j}, U_j^{k_j}]$, $k_j = 1, ..., n_j$.

Step 3: $\forall\, r = 1, ..., N_r$ compute $\Phi_j^r = \left[\alpha_j^r \ \beta_j^r\right]$ where $\alpha_j^r = \min_{x \in P_r} e_j^T x$, $\beta_j^r = \max_{x \in P_r} e_j^T x$.

Step 4: $\forall\, (k_j = 1, ..., n_j), (r = 1, ..., N_r)$: IF $(L_j^{k_j} < \beta_j^r)$ & $(U_j^{k_j} > \alpha_j^r)$ THEN $PI_j(k_j) \leftarrow r$.

Remark 1 *More efficiently, when the vertex representations of the polyhedral regions are available, then LPs in step 3 can be replaced by* $\alpha_j^r = \min_i V_i^r(j)$ *and* $\beta_j^r = \max_i V_i^r(j)$, *where* $V_i^r(j)$ *denotes the j-th element of i-th vertex of* P_r.

3.3 Online computation

Applying the steps 1 through 3 of Alg. 1 to all axes returns n_x polyhedral index sets PI_j, $j = 1, ..., n_x$. These n_x sets together with the polyhedral region descriptions (H_r, K_r) are the only data one needs to store to solve the online point location problem.

The online procedure is presented in Alg. 2. Note that the computation of the set intersection $PL = \cap_{j=1}^{n_j} PI_j(\overline{k}_j)$ in step 2 is more efficient than the sequential search through all PI_j's elements. Furthermore due to the incremental procedure in step 4 of Alg. 1, the elements in each index set $PI_j(\overline{k}_j)$ are ascending. Using this property a logarithmic-time solution to the set intersection problem is achieved in Alg. 3 based on the binary search method.

Remark 2 *The scaling parameters* (ε_j) *can be adjusted by first setting* $\varepsilon_j = 0$ *and increasing it iteratively until a satisfactory result is met. Note that increasing the* $\varepsilon_j s$ *will decrease the online time complexity and increase the storage complexity simultaneously. The tuning is com-*

pleted when both requirements are satisfied. When the online time complexity is more important than the storage, then the following thumb rule usually gives a good result:

$$\varepsilon_j = \min\left(\varepsilon_{\max},\, round\left(\log_2\left(\frac{|I_j|}{\delta P_r^j}\right)\right)\right),$$
$$\delta P_r^j = \min_{r\in\{1,\dots,N_r\}}\left(\max\left\{\delta_{thr},\, \left|proj_{e_j}^{P_r}\right|\right\}\right) \quad (4)$$

where ε_{max} is the maximum allowed value for scaling parameters given by the maximum available storage resources and δ_{thr} is the threshold value to ignore very small projected intervals. Roughly speaking, the rule (4) uses the minimum polyhedral projection length to calculate the scaling parameter in each axis while a minimum acceptable value is taken into account (δ_{thr}).

Algorithm 2 : Online computation

Step 1: At time t, for a given query point $x = [x_1, ..., x_{n_x}]^T$, and for each axis e_j determine sub-interval $I_j^{\bar{k}_j}$ which contains the j-th element of state x, i.e. $x_j \in I_j^{\bar{k}_j} = [L_j^{\bar{k}_j}\ U_j^{\bar{k}_j}]$.

Step 2: Use Alg.3 to compute set of all common indices in $PI_j(\bar{k}_j)$, $j = 1, ..., n_x$, that is $PL = \cap_{j=1}^{n_j} PI_j(\bar{k}_j)$.

Step 3: Apply direct search to the set PL. If the optimal region is found then return its index $i(x)$, else return 0.

Algorithm 3 : Set Intersection

Step 1: Initialize $PL \leftarrow \{PI_1(\bar{k}_1)\}$; $j \leftarrow 2$.

Step 2:

```
for j = 1 to n_x do
    PL ← Set_Intersect(PL, PI_j(k̄_j));
end for
```

FUNCTION ***ComSet*=Set_Intersect**(S_1, S_2)

```
ComSet ← []
for i = 1 to length(S_1) do
    first ← 1; last ← length(S_2);
    while first ≤ last do
        mid ← [(first + last) /2]
        if S_2(mid) < S_1(i) then
            first ← mid + 1;
        else if S_2(mid) > S_1(i) then
            last ← mid − 1;
        else
            ComSet ← [ComSet i];
            break;
        end if
    end while
end for
Return ComeSet;
```

3.4 Application of hash table for point location

In this paper we use the concept of hash tables and the associated hash functions in order to solve the step 1 (find $\bar{k}_j$) in processing time of order $O(1)$.
A hash table is made up of two parts [8]: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input data space (in our case, x) to the integer space that defines the indices of the array (in our case, $\bar{k}_j$). It is shown that the proposed hashing method leads to the perfect hashing. To this end, remember $PI_j(k_j)$ contains all regions intersecting with $I_j^{k_j}$. Associated with the j-th axis e_j, consider the following hash function which maps any query point $x = [x_1, ..., x_{n_x}]^T$, to an integer value:

$$S_j(x) = floor\left(\frac{x_j - L_j^1}{U_j^{n_j} - L_j^1} n_j\right) + 1,\ \ \forall j = 1, ..., n_x \quad (5)$$

Where $floor(.)$ is a function which maps a real number to the largest integer not greater than the number. The following theorem summarizes the basic properties of combining the proposed algorithm with the hash function in (5) and the associated hash table.

Theorem 1 *Consider the hash function (5) and the associated hash table is the set PI_j. Then for a given query point $x = [x_1, ..., x_{n_x}]^T \in R^n$, $S_j(x) = \bar{k}_j$ and the point location problem in Problem 1 is reduced to a search through the following set of polyhedra:*

$$PL = \cap_{j=1}^{n_x} PI_j\left(S_j(x)\right),\ \ \forall\ j = 1, ..., n_x \quad (6)$$

PROOF. First we prove that $S_j(x) \in Dom(PI_j(.))$, i.e. the $S_j(x)$ is an integer-valued function for which $S_j(x) \in \{1, 2, ..., n_j\}$ where $Dom(PI_j(.))$ denotes the domain of the set-valued function PI_j. To this aim, from the definitions of $S_j(x)$ in (5) and floor function it is easy to investigate that $S_j(x)$ is an increasing piecewise constant integer-valued function. As a consequence $S_j(L_j^1 e_j) \le S_j(x) \le S_j(U_j^{n_j} e_j)$, where $e_j \in R^{n_x}$ is the j-th unit vector. This result suffices to prove that $S_j(L_j^1 e_j) \in Dom(PI_j)$ and $S_j(U_j^{n_j} e_j) \in Dom(PI_j)$. This can be simply verified by direct substitution of $L_j^1 e_j$ and $U_j^{n_j} e_j$ in (5). Now it should be demonstrated that the function $S_j(x)$ returns the index of $\bar{k}_j$-th sub-interval $I_j^{\bar{k}_j}$ that contains the j-th element of the vector x, i.e. $S_j(x) = \bar{k}_j$. To this end, note that $x_j \in I_j^{\bar{k}_j}$ leads to $L_j^{\bar{k}_j} \le x_j \le U_j^{\bar{k}_j}$, then from $(U_j^{n_j} - L_j^1) > 0$:

$$\frac{L_j^{\bar{k}_j} - L_j^1}{U_j^{n_j} - L_j^1} n_j \le \frac{x_j - L_j^1}{U_j^{n_j} - L_j^1} n_j \le \frac{U_j^{\bar{k}_j} - L_j^1}{U_j^{n_j} - L_j^1} n_j \quad (7)$$

By using the equations $L_j^{\overline{k}_j} = L_j^1 + (\overline{k}_j - 1)\delta_j$, $U_j^{\overline{k}_j} = L_j^1 + \overline{k}_j\delta_j$ and $\delta_j = (U_j^{n_j} - L_j^1)/n_j$ and substituting in (7), it can be rewritten as

$$\overline{k}_j - 1 \leq \frac{x_j - L_j^1}{U_j^{n_j} - L_j^1} n_j \leq \overline{k}_j \tag{8}$$

Then using (5) yields $S_j(x) = \overline{k}_j$. □

Remark 3 *The set intersection procedure (Step 2) and also the direct search of the regions in PL to find the optimal region (Step 3) can be executed in a parallel architecture.*

Remark 4 *Since the hash tables PI_j are calculated offline, then access to the index set of the candidate optimal polyhedral regions has complexity of order $O(1)$. For each specific problem the maximum possible number of common intersecting polyhedral regions, i.e $|PL|$, can be determined after offline calculation. Note that this constant number is strongly depended on the scaling factor ε_j, how the regions are scattered and also the dimension n_x, but less depended on N_r.*

Remark 5 *It is emphasized that the only assumption which is used in the proposed method is the convexity of the P_is. As a consequence, unlike some of the existing alternatives, the present method can be applied to more general class of partitions including discontinuous and overlapping partitions. However, for severely overlapping partitions some performance degradation can be expected.*

4 Complexity analysis

4.1 Offline processing complexity

The preprocessing in Alg. 1 consists of four steps. Step 4 is the most dominant one and determines the complexity of the offline computation. In this step, there are $n_j = 2^{\varepsilon_j}$ subintervals in each axis.
Since step 4 is performed for all subintervals, then considering Remark 1 the computational complexity of Alg.1 can be measured as the sum of the complexity of $inter(proj_{e_j}^{P_r}) \cap I_j^{k_j} \neq \emptyset$ for all axes, i.e. $\sum_{j=1}^{n_x} O\left\{inter(proj_{e_j}^{P_r}) \cap I_j^{k_j} \neq \emptyset\right\}$. For each subinterval $I_j^{k_j}$, one needs to examine the two inequalities mentioned in Remark 1 for all N_r polyhedrons. Therefore the overall complexity will be of order $O\left(N_r \sum_{j=1}^{n_x} n_j\right)$.

4.2 Storage complexity

In addition to the polyhedral regions description (H_r, K_r), $r = 1, ..., N_r$, the only data which needs to be stored for online application are hash tables (or the index sets) PI_j, $j = 1, ..., n_x$. Suppose that $|PI_j(k_j)|$ denotes the cardinality of the set $PI_j(k_j)$, i.e. the number of polyhedral region indices contained in the set $PI_j(k_j)$. Then corresponding to each axis, $M_j = \sum_{k_j=1}^{n_j} |PI_j(k_j)|$ unsigned integer numbers need to be stored. Thus the total storage complexity will be $\sum_{j=1}^{n_x} M_j$. Note that the minimum and maximum cardinality of interval sets $PI_j(k_j)$ are mainly determined by the scaling factors ε_j.

4.3 Online processing complexity

The computations in Alg. 2 is composed of two main parts. In step 1, for a given state $x = [x_1, ..., x_{n_x}]^T$ the associated subintervals $I_j^{\bar{k}_j}$ are determined in $O(1)$ using the proposed hash function. Steps 2 and 3 in Alg. 2 are critical and determine the online processing complexity. The overall computational complexity is $O(\mathrm{N}_1 + \mathrm{N}_2)$ where $O(\mathrm{N}_1)$ and $O(\mathrm{N}_2)$ denote the complexity of steps 2 and 3 respectively. The set intersection for two arbitrary index sets $PI_{j_1}(\bar{k}_{j_1})$ and $PI_{j_2}(\bar{k}_{j_2})$ can be carried out in $O(m_{j_1} \log m_{j_2})$ using Alg.3, where $m_{j_1} = |PI_{j_1}(\bar{k}_{j_1})|$ and $m_{j_2} = |PI_{j_2}(\bar{k}_{j_2})|$. The following theorem characterizes the online complexity.

Theorem 2 *Consider the set intersection in Alg. 3, then the worst case online processing complexity is of order $O(\mathrm{N}_1 + \mathrm{N}_2)$ where $O(\mathrm{N}_2) = O(\max|PL|)$ denotes the maximum possible cardinality of common set PL and $O(\mathrm{N}_1) = O\left(\sum_{j=1}^{n_x-1} \min_{i\in\{0,...,j\}} (\mathrm{M}_i) \log(\mathrm{M}_{j+1})\right)$, where $\mathrm{M}_j = \max_{k_j\in\{1,...,n_j\}} |PI_j(k_j)|$ and $\mathrm{M}_0 = \max_{j\in\{1,...,n_x\}} \mathrm{M}_j$.*

PROOF. Since the cardinality of all index sets are known from the preprocessing, $O(\mathrm{N}_2)$ is determined by the maximum possible cardinality of common set PL, i.e. $O(\max|PL|)$. Note that applying Alg. 3 to the first two index sets $PI_1(\bar{k}_1)$ and $PI_2(\bar{k}_2)$ impose a computational complexity of order $O(m_1 \log m_2)$, then for the resulting common set (PL) we have $|PL| \leq \min(m_1, m_2)$. Therefore calling the Alg. 3 for PL and the next index set $PI_3(\bar{k}_3)$ will similarly impose the complexity of order $O(|PL| \log m_3) \leq O(\min(m_1, m_2) \log m_3)$. Iterating this procedure shows that for i-th index set the complexity is $O\left(\min_{i\in\{0,...,j\}} (m_i) \log(m_{j+1})\right)$. The worst case (upper bound) can be determined by taking maximum over subintervals for each axis, i.e. $\mathrm{M}_j = \max_{k_j\in\{1,...,n_j\}} (m_j = |PI_j(k_j)|)$. Finally the total online processing complexity is the sum of all subproblems processing time. $\mathrm{M}_0 = \max_{j\in\{1,...,n_x\}} \mathrm{M}_j$ is a constant number that is used to simplify the notation. □

Remark 6 *Further simplification in the online computations can be achieved using a secondary hash table in the set intersection algorithm. This is while only a bit array H with length of N_r needs to be added to the storage complexity. Using this bit array as a hash table it is possible to solve the set intersection problem with complexity of order $O(m_1)$ instead of $O(m_1 \log m_2)$ where $m_1 \leq m_2$. Then it is sufficient to hash the second index set to the bit array H, i.e. $H(i) = 1, \forall\, i \in S_2$ and $H(i) = 0, \forall\, i \notin S_2$. Then checking if each element of first set S_1 is contained in S_2 or not can be done in $O(1)$. Sorting sets in a way that $m_1 \leq m_2 \leq ... \leq m_{n_x}$ takes time of order $O(n_x \log(n_x))$ while $m_j s$ are computed offline.*

Remark 7 *Considering Remark 6 and the proposed algorithm, an approximation to the number of arithmetic operations including summation/subtraction, multiplication/division and comparison can be found as $\#(arith.ops.) \leq 7n_x + n_x \log n_x + N_{\text{intersect}} + 2n_x N_{com} N_H$ where $7n_x$ denotes the number of arithmetic operations of hash function evaluation and $n_x \log(n_x) + N_{\text{intersect}}$ refers to the arithmetic operations of array sorting and set intersection using hash table introduced in Remark 6. Accordingly $N_{\text{intersect}}$ can be approximated by $\min_j(\max_{k_j} |PI_j^{k_j}|)$. Finally $2n_x N_{com} N_H$ denotes the arithmetic operations of direct search through the set PL where N_H is the maximum number of hyper planes describing any polyhedral region and N_{com} is the maximum possible cardinality of the set PL which can be pre-computed when the polyhedral index sets PI_j are computed offline. A quite conservative upper bound for N_{com} is $\max_j(\max_{k_j} |PI_j^{k_j}|)$.*

5 Numerical examples

The proposed method has been applied to several test cases to illustrate the computational performance of the method. PWA functions have been generated with the aid of the Multi Parametric Toolbox (MPT) for MATLAB® [13] with n_x in the range $2-5$ as shown in Table 1. Then, Algs. 1-3 together with the algorithm proposed in [19] and the direct search method were applied to the test cases and the results are compared in table 1. These two alternative algorithms are considered to be representative for state-of-the-art performance in terms of online computation time or storage and preprocessing demands, respectively, and therefore provide relevant benchmarks for the proposed algorithm that aims to address the trade-off between the time and storage complexities. Fig.2 illustrates this trade-off when the scaling parameter varies from 0 to 15 for a test case. Fig.2 shows that ε_j parameterizes the solutions similar to Pareto optimality Roughly speaking, one can imagine that a good choice of scaling factor in this case is around 6-8 which makes an appropriate trade-off between online processing and storage demands when both time and storage are important. The same strategy is used to set the scaling parameters in each test case. Comparing to the [19] the results in table 1 demonstrate the effectiveness of the proposed method in the online processing. On the other side, the associated preprocessing time shows that the present method is tractable for applications with large number of polyhedral regions for which some of the existing algorithms such as [19] are not applicable. Note that the algorithms were written in MATLAB platform and executed by a 2.6 GHz processor. Furthermore, the simulations with biggest N_r in all test cases refer to the orthogonal partitions arising in the nonlinear explicit MPC (see e.g. [10]) and the results from these experiments verify that even though the partitions contain more polyhedral regions, the present method still works well. Finally, to exemplify the major advantage of the present algorithm compared to the state space cell partitioning or even pre-computation of PLs for all interval instead of using the proposed set intersection method, consider for examle the test case $n_x = 5,\ \varepsilon_j = 8$. In this case, the state space cell partitioning approach leads to $N_c = (2^8)^5 \approx 1.1 \times 10^{12}$ cells in the state space. Unambiguously this amount of complexity is not practical to deal with in offline computation.

Table 1
Performance comparison of the proposed method (Algs.1 & 2), Direct Search (DS) and algorithm proposed in [19]. $\star$ denotes that the implementation of the algorithm is not tractable due to preprocessing time for this test case.

Problem			**Offline Time**		**Arith.ops.(worst)**			**Storage**($\times 10^4$)		
$\mathbf{n_x}$	$\mathbf{N_r}$	$\varepsilon_\mathbf{j}$	**Alg.1 (sec)**	**BST (sec)**	**DS**	**Alg.2**	**Alg. [19]**	**DS**	**Alg.2**	**Alg. [19]**
2	139	5	0.3	57	2240	254	54	0.16	0.24	0.31
	853	5	5.6	2515	13656	322	72	1.2	2.3	3.7
	2258	8	8.67	5300	35984	240	82	2.7	3.4	15
	3125	8	11.7	2891	50e3	245	118	3.8	5.2	39
	15625	8	58.6	$\star$	250e3	607	$\star$	2	14	$\star$
3	1565	8	9.9	62982	57312	938	166	3.8	14.3	18
	2998	6	478	24568	106e3	1962	176	7	27	44
	16807	8	102	$\star$	605e3	2903	$\star$	40.3	79	$\star$
4	834	8	171	981	57312	1737	226	3.6	23	43
	6561	8	33	$\star$	420e3	3480	$\star$	26.2	46.3	$\star$
	13716	8	122	$\star$	878e3	5272	$\star$	55	134	$\star$
5	242	7	0.7	3998	24200	606	210	1.1	1.3	6.5
	1331	7	279	$\star$	133e3	1799	$\star$	8	24	$\star$
	14641	8	60	$\star$	146e4	11687	$\star$	87.8	77.2	$\star$
	20680	8	237	$\star$	207e4	14463	$\star$	124	373	$\star$

6 Conclusion

A new efficient and flexible approach to the point location problem arising in the recently developed explicit MPC, as well as other PWA function applications, was introduced. Some of existing methods attempt to solve

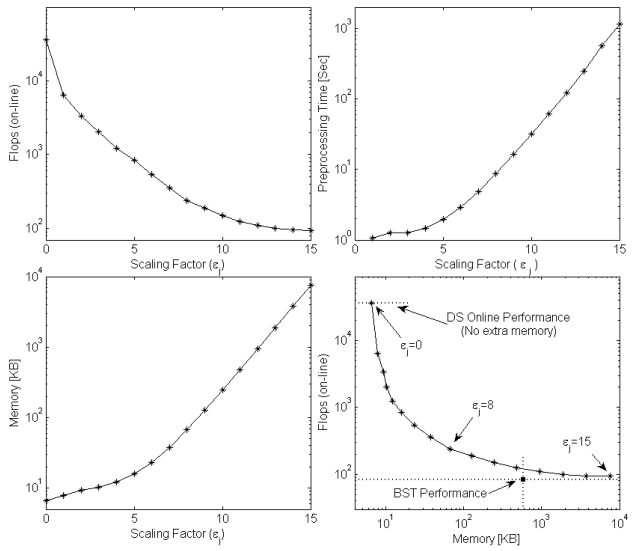

Fig. 2. Complexity trade-off when the scaling parameter varies from 0 to 15, ($n_x = 2, N_r = 2258$).

a reduced point location problem instead of the original problem (e.g. [15], and [16]). The proposed method has a similar goal in its interior. The main distinguishing property in the present method refers to the existence of some tuning parameters that enable the designer to explore the trade-off between time and storage complexities in a simple way. It was also shown that boosting this algorithm with hashing theory improves strongly the online processing complexity with modest offline computation and storage load. Also it was illustrated that the online processing complexity is mainly depended on the scaling factors ε_j, the dimension n_x and how the regions are scattered instead of the number of polyhedral region (N_r). In terms of the storage complexity one just needs to store the index sets PI_j, $j = 1, ..., n_x$ as the one-dimensional hash tables in addition to the polyhedral regions descriptions.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[2] M. Baotić, F. Borrelli, A. Bemporad, and M. Morari. Efficient on-line computation of constrained optimal control. *SIAM Journal on Cont. and Opt.*, 47(5):2470–2489, 2008.

[3] A. Bemporad, F. Borrelli, and M. Morari. Model predictive control based on linear programming - the explicit solution. *IEEE Trans. on Automatic Control*, 47(12):1974–1985, 2002.

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

[5] F. Borrelli, M. Baotic, A. Bemporad, and M. Morari. Efficient on-line computation of constrained optimal control. In *40th IEEE Conf. on Decision and Control*, volume 2, pages 1187–1192, 2001.

[6] M. Canale, L. Fagiano, and M. Milanese. Set membership approximation theory for fast implementation of model predictive control laws. *Automatica*, 45(1):45 – 54, 2009.

[7] F.J. Christophersen, M. Kvasnica, C.N. Jones, and M. Morari. Efficient evaluation of piecewise control laws defined over a large number of polyhedra. In *European Control Conference*, pages 2360–2367, 2007.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[9] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[10] T. A. Johansen. Approximate explicit receding horizon control of constrained nonlinear systems. *Automatica*, 40(2):293–300, 2004.

[11] T. A. Johansen and A. Grancharov. Approximate explicit constrained linear model predictive control via orthogonal search tree. *IEEE Trans. Automatic Control*, 48(5):810–81, 2003.

[12] C. N. Jones, P. Grieder, and S. V. Raković. A logarithmic-time solution to the point location problem for parametric linear programming. *Automatica*, 42(12):2215–2218, 2006.

[13] M. Kvasnica, P. Grieder, M. Baotić, and M. Morari. *Multi-parametric toolbox (MPT)*. Hybrid Systems: Computation and Control. Springe, 2004.

[14] J. Snoeyink. Point location. In J. E. Goodman and J. Órouke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press, Boca Raton, FL, USA, 1997.

[15] J. Spjøtvold, S. V. Raković, P. Tøndel, and T. A. Johansen. Ieee conf. on decision and cont. pages 4568–4569, 2006.

[16] D. Sui, L. Feng, and M. Hovd. Algorithms for online implementations of explicit mpc solutions. In *17th IFAC World Congres*, pages 3619–3622, 2008.

[17] P. Tøndel, T. A. Johansen, and A. Bemporad. Computation and approximation of piecewise affine control via binary search tree. *IEEE Conf. on Decision and Cont.*, 8:3144–3149, 2002.

[18] P. Tøndel, T. A. Johansen, and A. Bemporad. An algorithm for multi-parametric quadratic programming and explicit mpc solutions. *Automatica*, 39(3):489–497, 2003.

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

[20] Y. Wang, Jones, C.N., and J. Maciejowski. Efficient point location via subdivision walking with application to explicit mpc. In *European Control Conference*, pages 447–453, 2007.

[21] C. Wen, X. Ma, and B. E. Ydstie. Analytical expression of explicit mpc solution via lattice piecewise-affine function. *Automatica*, 45(4):910–917, 2009.

[22] M.N. Zeilinger, C.N. Jones, and M. Morari. Real-time suboptimal model predictive control using a combination of explicit mpc and online optimization. In *47th IEEE Conf. on Decision and Control*, pages 4718 – 4723, 2008.