Circuit Generation for Efficient Projection onto Polyhedral Sets in First-Order Methods

Conference Paper

Author(s):
Merkli, Sandro; Jerez, Juan; Domahidi, Alexander; Smith, Roy; Morari, Manfred

Publication date:
2015

Permanent link:
https://doi.org/10.3929/ethz-b-000111250

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
https://doi.org/10.1109/ECC.2015.7331066
Circuit Generation for Efficient Projection onto Polyhedral Sets in First-Order Methods

Sandro Merkli¹,², Juan Jerez¹,³, Alexander Domahidi²,³, Roy S. Smith¹, Manfred Morari¹

Abstract—First-order numerical optimization methods are a common choice for low-cost embedded MPC implementations. Their applicability is typically restricted to problems with simple constraints due to the difficulty of Euclidean projection onto more complex feasible sets. However, many practical problems have non-trivial polyhedral constraints. For such polyhedral sets, the projection can be explicitly written as an evaluation of a piecewise affine function. Existing methods evaluate such functions by iteratively traversing binary trees, which leads to small recursive circuit implementations that have a large computational latency. In this paper, we present a recursion-free approach that uses mixed-integer linear programming in the design stage to optimize result reuse. A heuristic is presented to approximately solve the design optimization problem in a practical amount of time. Automatic circuit generation is used to obtain problem-specific implementations that can significantly outperform current reference implementations with only modest increases in circuit size. The resulting projection circuits enable the application of first-order methods to problems with polyhedral constraints while retaining high performance, increasing their range of application.

I. INTRODUCTION AND PREVIOUS WORK

Model predictive control (MPC) has seen great popularity in research as well as in industry in the past decade. For linear MPC problems with convex quadratic cost functions subject to polyhedral constraints, a convex quadratic program (QP) has to be solved at every time step to determine the next control action. If the problem is of low dimensionality, the solution to this QP can be computed offline as an explicit piecewise affine function of the current state of the system [1], [2]. However, for larger problems, the explicit solution is impractical to compute and store, and the online solution of the optimization problem is more attractive.

While circuit implementations [3] of structure-exploiting interior-point methods [4] can handle more general problems, first-order methods are generally preferred in circuit implementations due to their lower iteration complexity and the possibility to implement them reliably using fixed-point arithmetic [5]. Recently, architectures for first-order methods were proposed to achieve up to 1 MHz sampling rates on embedded computing platforms [6]. However, this approach is limited to systems with simple constraints, such as (soft) lower and upper bounds on the variables. This limitation is due to the lack of efficient methods for projection onto more complex constraint sets, an operation which is performed in every iteration of first-order methods like ADMM [7] and (dual) fast gradient methods [8], [9].

One way to avoid complex projections is to convert the polyhedral constraints into equality constraints by introducing slack variables that can be trivially projected onto the non-negative orthant. However, this increases the number of variables and can have a negative impact on the convergence speed of the method. It is hence often undesirable, especially in embedded cases, where smaller problems are preferred. Another approach for dealing with polyhedral constraints is to dualize them and use gradient projection algorithms in the dual domain [10]. However, in problem instances with many constraints, the dual problem can become larger than the primal. In such cases, the direct approach presented in this paper can be advantageous.

Projection on polyhedra can be written as a multiparametric quadratic program whose solution is a piecewise affine (PWA) function of the point to be projected [11]. Current state-of-the-art methods evaluate PWA functions using an iterative decision tree traversal process, as described in Fig. 1a, followed by a simple function evaluation [11]–[13]. Implementations of these methods usually do not assume any structure in the solution. Additionally, due to the recursive nature of the tree traversal, these implementations cannot be efficiently pipelined.

The approach presented in this paper departs from the tree traversal concept by simultaneously considering all possible computations that could be required for the projection and evaluating them all in a branch-free, non-recursive manner.

1Automatic Control Lab, ETH Zurich, Physikstrasse 3, 8092 Zurich {merkli, smith, morari, jerez}@control.ee.ethz.ch
2Inspire-IAn, Inspire AG, Technoparkstrasse 1, 8005 Zurich, {merkli, domahidi}@inspire.ethz.ch
3embotech GmbH, Physikstrasse 3, ETL K10.1, 8092 Zurich, {jerez, domahidi}@embotech.com

© 2015 European Control Association (EUCA)
as illustrated in Fig. 1b. While the number of computations required can grow exponentially with the problem dimension, efficient reuse of common subexpressions can lead to implementations that are both faster and smaller than existing solutions for constraint sets occurring in practice. The theoretical contribution of this paper is the introduction of a formal method for minimizing the computational effort for evaluating all required results by solving, offline at design time, a mixed-integer linear program (MILP). The solution of the MILP can quickly become intractable, hence we introduce a heuristic to obtain an approximate solution. The methods outlined in this paper are inspired by similar ideas applied to the problem of multiple constant multiplication (MCM) in signal processing applications [14]–[16].

A recent circuit design tool for implementation of decision tree traversal is MOBY-DIC [12]. Circuits generated by MOBY-DIC use an architecture as shown in Fig. 1a. It was designed for the evaluation of PWA functions from explicit MPC controllers. The authors of [17] present a similar approach that evaluates $K \in \mathbb{N}$ affine expressions in parallel. This leads to a speedup by a factor of $K$ over MOBY-DIC, but increases the circuit size by the same factor.

The remainder of this paper is structured as follows: Section II presents the conventional approach. Our proposed alternative approach to PWA function evaluation and the design optimizations required to make it efficient are formally described in Section III. In Section IV, a simple heuristic is presented that yields approximate solutions for the aforementioned optimizations. Finally, Section V presents some numerical results that confirm the effectiveness of the approach and Section VI concludes the paper.

II. PRELIMINARIES

This section describes the projection problem as a multiparametric quadratic program and outlines the state-of-the-art approach for solving it via decision tree traversal.

A. Affine projection as parametric QP

The Euclidean projection of $x^0 \in \mathbb{R}^n$ onto a polyhedron $\{x \in \mathbb{R}^n | Ax \leq b\}$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ can be written as a multiparametric quadratic program (mpQP),

$$
\begin{align*}
\phi_\star &:= \arg \min_{z} \|x^0 - z\|_2^2 \\
\text{subject to} \quad Az &\leq b,
\end{align*}
$$

with parameter vector $x^0$ and result $x_\star \in \mathbb{R}^n$. The solution can be precomputed as a PWA function of the parameter $[1]$. The complexity of this so-called explicit solution grows exponentially with the problem dimension, but in this paper we are concerned with cases where the complexity of the solution is manageable. The solution comes in the form of a polyhedral partition of the parameter space,

$$
P_i = \{x \in \mathbb{R}^n | G_i x \leq g_i\}, \quad i \in \{1, \ldots, N_p\},
$$

and for each region $i$, there exists an affine function $\phi_i(x) := F_i x + f_i$ such that

$$
x^0 \in P_i \implies x_\star = \phi_i(x^0).
$$

Finding this partition and the associated functions (i.e. solving the mpQP in the parameters) can be accomplished using tools such as the Multi-Parametric Toolbox [13].

B. Classical PWA function evaluation

Once the polyhedral partition $P_i, i \in \{1, \ldots, N_p\}$ and the functions $\phi_i(x)$ are known, computing $x_\star$ given $x^0$ with the conventional approach involves two steps:

1) Point location: Finding the region $i$ such that $x^0 \in P_i$.

2) Function evaluation: Evaluating $\phi_i(x^0)$ in (3).

While the second step in this algorithm is trivial, the first step is more challenging due to the potentially high number $N_p$ of polyhedra defining the partition in (2). State-of-the-art approaches use decision tree traversal with a precomputed decision tree. Several methods exist to create such decision trees [11], [18].

C. Structure of PWA functions arising from projections

PWA functions arising from Euclidean projections have more structure than general PWA functions: The rows of the $G_i$ and $F_i$ often have common parts. For constraint sets that are partially aligned with the coordinate axes, the $F_i$ are generally sparse. These properties are exploited in the design optimizations presented in the next section.

III. RESULT REUSE MAXIMIZATION USING OPERATION GRAPHS

In this paper, we consider an alternative approach for evaluating PWA functions, which is based on the following observation: All results that could possibly be required to compute the projected point $x_\star$ can be obtained by a single matrix-vector product:

$$
r := \left[ \begin{array}{c}
G \\
F \\
\end{array} \right] \left[ \begin{array}{c}
x^0 \\
1 \\
\end{array} \right] := \left[ \begin{array}{c}
d \\
f_p \\
\end{array} \right],
$$

where $G$ and $F$ are defined as follows:

$$
G := \left[ \begin{array}{ccc}
G_1 & -g_1 \\
\vdots & \vdots \\
G_N & -g_{N_p} \\
\end{array} \right], \quad F := \left[ \begin{array}{ccc}
F_1 & f_1 \\
\vdots & \vdots \\
F_{N_p} & f_{N_p} \\
\end{array} \right].
$$

The rows of $G$ will be referred to as hyperplane checks, the rows of $F$ as function evaluations. Once all entries of $r$ are computed, the index $i$ of the polyhedron containing the point $x^0$ to be projected can be found based on the signs of the entries of $d$. The result of the function evaluation is then given by the section of $p$ corresponding to $\phi_i(x^0)$. The projection has therefore been rewritten as a matrix-vector product followed by a simple result selection.

In general, evaluating the matrix-vector product (4) requires an excessive number of computations compared to the tree traversal approach when no structure is exploited. What makes the approach presented here efficient is the fact that on many practical constraint sets, the matrix-vector product (4) can be very efficiently implemented using a combination of two simplification processes:
(i) Removal of redundant rows from (4)
(ii) Efficient computations using intermediate result reuse.
Application of (i) can be done algorithmically: if multiple hyperplane checks in $U$ have the same result, all but one can be removed, and the same applies to function evaluations.

**Definition 1** Let

$$x := \begin{bmatrix} x^0 \\ 1 \end{bmatrix} \in \mathbb{R}^{n+1},$$

and let $\tilde{U}$ be a version of $U$ in (4) with all redundant rows removed. Let

$$\tilde{r} := \tilde{U}x.$$

Assume $\tilde{U} \in \mathbb{R}^{N_r \times n+1}$ and hence $\tilde{r} \in \mathbb{R}^{N_r}$.

With the redundant rows removed, (ii) can be applied. Often the rows of $\tilde{U}$ are partially equal, which means that many intermediate results could potentially be used several times. As an example, consider the following example:

**Example 1** The following matrix product is to be computed:

$$r^{ex} = U^{ex}x = \begin{bmatrix} 1 & 4 & 6 & 8 \\ 0 & 2 & 3 & 4 \end{bmatrix}x.$$  

In order to compute all entries of $r^{ex}$, one could first compute $3x_3 + 4x_4$ to get $r_3^{ex}$, then add $2x_2$ to get $r_2^{ex}$ and finally multiply by 2 and add $x_1$ to get $r_1^{ex}$. This way, 4 products and 3 sums suffice to compute $r^{ex}$. In comparison, a standard matrix product would involve 12 products and 9 sums.

What led to the reduction in the number of required operations in Example 1 was common subexpression reuse. Our goal is to create a tool to carry out this procedure automatically and optimally.

A. Computation representation using operation graphs

As was demonstrated in Example 1, the order of operations in the computation of (6) is crucial. For systematically finding an optimal order of operations, a formal description of (6) that explicitly includes this order is required. The description chosen here is an operation graph, a concept used in fields such as software compiler design. The matrix-vector product (6) can be represented as a directed graph $G = (V, E)$ in which every vertex $v_i \in V$ simultaneously represents an operation and its result. All operations are assumed to take one computational cycle. Edges represent data flow: If an edge $e_{i,j} \in E$ points from vertex $v_i$ to vertex $v_j$, this means that the result computed at vertex $v_i$ is used in the computation of the result for vertex $v_j$ and has to be computed first. The sources of the graph are defined as the elements of $x$. An example of such a graph for the computation in (7) is shown in Fig. 2. In order for the graph to be a valid representation of (6), some structural conditions have to hold, which will be formally introduced in Section III-B. These conditions generally do not define the operation graph uniquely, just like the order of operations in (6) is not uniquely defined. Rather, they define the feasible set of operation graphs, that is, all graphs leading to the same result, but with different operation orders and amounts of vertices.

B. Operation graph representation for optimization

This section introduces a formal description of operation graphs amenable to formulation as an MILP. Operation graphs will be modeled as a set of $N_v$ vertices $v_i$ together with constraints on the edges and vertices that ensure the construction of valid operation graphs expressing (6). The maximum amount of vertices $N_v$ required can be bounded, as shown in [19].

1) Vertex descriptions: Let $\mathbb{B} := \{0, 1\}$ and let $c \in \mathbb{R}^{N_v}$ be the vector of all distinct coefficients (excluding $\{0, 1\}$) appearing in $\tilde{U}$. For every vertex $v_i$, let $s_i \in \mathbb{R}^{n+1}$ describe its result (interpreted as $s_i^T x$). Define

$$S := [s_1 \ \cdots \ s_{N_v}]^T \in \mathbb{R}^{(N_v \times n+1) \times (n+1)}$$

as a matrix representing all vertex results and all entries of $x$ from (5). The latter are represented by $I$ in (8), which is the identity matrix of dimension $n + 1$. Since the represented computations are linear, every vertex is either a sum of previous results or a product of a previous result with a constant.

A product vertex $v_i^p$ computes the product of one element of $c$ (say, element with index $k$) with a previous result (for example, result $j$):

$$s_i = c_k s_j.$$

Therefore, product vertices are defined by

$$s_i = (m_i^T c) a_i^T S$$

where $a_i \in \mathbb{B}^{N_v+n+1}, \|a_i\|_1 = 1$ selects which of the previous results is used and $m_i \in \mathbb{B}^{N_v}, \|m_i\|_1 = 1$ selects which coefficient is used.

Sum vertices can be written similarly: A sum vertex $v_i^s$ computes the sum of two previous results (say, results $j, k$):

$$s_i = s_j + s_k,$$

and hence sum vertices can be written as

$$s_i = a_i^T S,$$

with $a_i \in \mathbb{B}^{N_v+n+1}, \|a_i\|_1 = 2.$
Neither the number of sum vertices, product vertices, nor the total number of vertices required are known a priori, which requires the introduction of two more binary variables per vertex: $\mu_i$, which determines if the vertex $i$ is a sum ($\mu_i = 0$) or a product ($\mu_i = 1$) and $z_i$, which determines if the vertex $i$ is used ($z_i = 0$) or not ($z_i = 1$).

Altogether, a general vertex $v_i$ can now be introduced that can become either a sum, a product or an unused vertex depending on $\mu_i$ and $z_i$ as follows:

$$s_i^T = (m_i^T c + (1 - \mu_i))a_i^T S.$$  \hspace{1cm} (11a)

$$\|a_i\|_1 + \|m_i\|_1 = 2(1 - z_i).$$ \hspace{1cm} (11b)

$$\|m_i\|_1 = \mu_i.$$ \hspace{1cm} (11c)

For $\mu_i = 0$, (11a) becomes (10), for $\mu_i = 1$ it becomes (9) and for $z_i = 1$, it follows that $s_i, a_i, m_i, \mu_i$ all become 0. Note that while (11) is bilinear in the variables $\mu_i, a_i$ and $S$, the only products of variables that appear are of binary with continuous variables, which can be reformulated into linear constraints using Big-M reformulation [20].

2) Graph-wide constraints: In addition to (11), another set of constraints has to be introduced to ensure that the graph description implements (6). First, a variable $l_i \in \mathbb{N}$ is introduced for each vertex, denoting the computational cycle when its associated operation is performed. The additional constraints are then as follows:

$$l_i - l_j + N_v(1 - a_{ij}) \geq 1$$ \hspace{1cm} (12a)

$$s_k^T = \tilde{U}_k.$$ \hspace{1cm} (12b)

\forall k \in \{1, \ldots, N_v\} \text{ and } \forall i, j \in \{1, \ldots, N_v\}, \text{ and where } \tilde{U}_k; denotes the $k$-th row of $U$. The constraints (12b) ensure that all entries of $\tilde{r}$ are actually computed by the graph. The circle avoidance constraints (12a) can be interpreted as follows: If vertex $v_j$ is used in the computation of $v_i$ (which is the case if $a_{ij} = 1$), it has to be computed first, which means $l_i > l_j \iff l_i - l_j \geq 1$ since the $l_i$ are integer. However, if the result of $v_j$ is not used in the computation of $v_i$, this constraint does not have to hold: If $a_{ij} = 0$, (12a) becomes

$$l_i - l_j \geq 1 - N_v$$

which always holds, since there are at most $N_v$ computational cycles required to compute the $N_v$ results.

3) Cost function: A variety of design objectives can be implemented with a suitable choice of cost function. In a sequential software implementation, one would want to minimize the total number of operations. This can be implemented using the cost function

$$C_{\text{min ops}}(z_i, i = 1, \ldots, N_v) := -\sum_{i=1}^{N_v} z_i$$

which will lead to as many vertices being set to “unused” ($z_i = 1$) as possible and therefore the minimum number of operations. If the minimum number of operations is desired with the additional constraint that the computational latency be kept below a certain level $\bar{\ell}$, the constraints

$$l_i \leq \bar{\ell} \quad \forall i \in \{1, \ldots, N_v\}$$

can be introduced. A special case of this would be to limit the implementation to take the lowest possible computational latency, which is $[1 + \log_2(n + 1)]$ cycles. Such a minimal latency implementation is desirable in parallel circuit implementations for example.

Based on the definitions of variables and constraints in the previous sections, the problem of finding a graph with the minimum number of operations implementing (6) can be formulated as follows:

$$G^* \overset{\text{arg minimize}}{=} C_{\text{min ops}}(G)$$ \hspace{1cm} (13)

subject to (11), (12)

where

$$G := \left\{(s_i, a_i, m_i, l_i, z_i, \mu_i) \mid i \in \{1, \ldots, N_v\} \right\}$$

is the graph description. The formulation in (13) is an MILP, which is NPhard to solve, even for the case where only additions are considered [21]. Nevertheless, for small problem dimensions, it is possible to compute an optimal solution using established numerical packages such as CPLEX [22] or GUROBI [23]. Since this optimization is run as a part of the design stage, long computation times might be acceptable in certain applications to find the optimal implementation.

IV. DIVIDE-AND-CONQUER HEURISTIC

For cases where solving (13) becomes prohibitively complex, a heuristic was designed to quickly find a (potentially suboptimal) graph implementing (6). This graph can then be used directly or as a warm-start for the numerical packages used to solve (13). The heuristic applies a divide-and-conquer methodology, similar to the approach taken in [14].

The heuristic, which is outlined formally in Fig. 3 and 4, starts with a graph that only contains the entries of the input vector $x$. Next, every entry of $\tilde{r}$ from (6) is added as a vertex, which ensures that all results from (6) are present. Then, each of the results is (recursively) decomposed into one operation and two operands for a sum or one operand for a product. This recursion on one result terminates once an expression is decomposed into vertices that are already in the graph. The heuristic then proceeds with the next result.

Suppose that one entry of $\tilde{r}$ were $x_1 + 2x_2 + x_4$. The heuristic would add a vertex for this expression, then decompose the expression into $(x_1 + 2x_2) + x_4$, for example, and recurse on $x_1 + 2x_2$ and $x_4$ separately.

The core operation of the heuristic is the DECOMPOSE operation in Fig. 4: In order to ensure that the recursion terminates, the decomposition operation has to yield operands that are either already present in the graph or require strictly fewer operations to implement than the original expression. A simple implementation of the decomposition function is given in the following example:

**Example 2 (DECOMPOSE implementation)** If the expression is a scalar multiple of an expression already present in the graph, make the new vertex a product and link it to the
1: function BuildGraph($\tilde{U}$)
2: $G \leftarrow \langle \emptyset, \{e_i \mid i \in \{1, \ldots, n + 1\}\}\rangle$
3: $N_r \leftarrow$ Number of rows in $\tilde{U}$
4: for $i \in 1, \ldots, N_r$ do
5: \hspace{1em} AddRowRecursive($G, \tilde{U}_i$)
6: \hspace{1em} end for
7: return $G$
8: end function

Fig. 3. Heuristic graph building algorithm. Line 2 initializes the graph with only the inputs (which are encoded as the unit vectors $e_i$) and no edges. Then the rows of the sorted matrix are added recursively one after another.

1: function AddRowRecursive($G, u$)
2: if $u$ trivial or already present in $G$ then
3: \hspace{1em} return
4: \hspace{1em} end if
5: $(u_a, u_b) \leftarrow$ Decompose($u$)
6: AddRowRecursive($G, u_a$)
7: AddRowRecursive($G, u_b$)
8: Add vertex for $u$ and edges $(u_a, u)$, $(u_b, u)$
9: end function

Fig. 4. Recursive row addition algorithm. Lines 2 to 4 represent the termination criterion for the recursion. The Decompose function divides the input row into two rows that require fewer operations, ensuring that at some point the algorithm reaches the inputs.

already present vertex. Otherwise, the expression is a sum of two or more terms. In that case, decompose this sum into two sums with fewer summands.

Many variations of this algorithm are possible: The rows of $\tilde{U}$ can be added in different orders, or more advanced Decompose functions can be used. Experimentally, such variations yielded only limited improvements. In the numerical experiments in Section V, rows of $\tilde{U}$ were sorted in ascending order by their number of nonzero entries before being added to the graph and Decompose was extended to reuse the largest partial sums already present in the graph. This behavior is based on the intuition of trying to use already computed results to construct others as shown in Example 1. Also, the heuristic was extended to only create graphs of the lowest possible computational latency. More elaborations on the heuristic are outlined in [19].

V. NUMERICAL RESULTS

A. Automatic circuit generation from operation graphs

The graphs created by the proposed methods can be directly translated into circuitry: Each vertex becomes a computational unit in hardware, and the edges become the data links between them. Intermediate registers are inserted between units for pipelined computations and to ensure that all results are available at the same time. A toolchain was developed to automate the design process from a PWA solution (2)–(3) to VHDL code for the projection circuit.

B. Numerical example: ordered set projection

A practical example that suits the methods from this paper well is projection onto the ordered set described by

$$X := \{x \in \mathbb{R}^k \mid 0 \leq x_1 \leq x_2 \leq \cdots \leq x_k \leq \bar{x}\},$$  \hspace{1em} (14)

which appears for example in applications where the $x_i$ represent event times [24]. The Multi-Parametric Toolbox [13] was used to create the solution map (2)–(3) for the projection onto (14) for $k = 1, \ldots, 6$. The VHDL codes generated by MOBY-DIC and our toolchain were implemented on a Xilinx FPGA. The resulting circuit sizes and clock cycle latencies for one projection are shown in Fig. 5, demonstrating a large speedup for comparable circuit sizes. While the latency of MOBY-DIC’s circuits scales linearly with the problem dimension, the proposed approach scales logarithmically. The number of circuit elements used depends on the sparsity of the matrices in the solution map (2)–(3). Experimentally, it was observed that the approach presented in this paper works best on low-dimensional constraint sets that are partially aligned with the coordinate axes. The heuristic was used throughout instead of the MILP optimization (13) to demonstrate its effectiveness. It has to be noted here that MOBY-DIC was designed for explicit MPC evaluation, which is a more general problem. The goal of the comparison is to show that approaches that take advantage of the structure in PWA functions arising from Euclidean projections can significantly outperform general methods.

C. Numerical example: MPC use case

In [6], an ADMM solver for MPC problems with input and state constraints was implemented in a circuit architecture with adjustable parallelism. In this algorithm (Algorithm 2 in [6]), projection onto the state and input constraints is required. Consider a problem as shown in [6] with 2 inputs, 4 states and a prediction horizon of 10. For this problem dimension, [6] reports that the ADMM solver with maximum parallelism requires 13 clock cycles per iteration without the projection block. Assume now that the state constraints are described in (14) for each stage separately. This is a 4-dimensional ordered set for each stage. In the presented example, evaluating the projection with a conventional method would yield a circuit that spends 97% of the computation time in the projection block. This emphasizes the need for faster projections, even at the cost of larger circuits. The number of clock cycles required for the projections and one ADMM iteration are listed in Table I for MOBY-DIC and for the approach presented in this paper. It can be seen that more than an order of magnitude speedup of the whole optimization is realized when using the proposed approach as opposed to the conventional implementation method. The reasons for the large speedup obtained with the presented approach are the increased parallelism, implemented without a significant increase in circuit size due to result reuse; and the pipelined design, which requires only one additional clock cycle for each additional independent projection. In contrast, the conventional approach implemented by MOBY-DIC requires $k$ times as many cycles for $k$ projections as it does for one.

VI. CONCLUSION

In this work, a novel approach was introduced to evaluate PWA functions. The approach considers all possibly required
computations at once and efficiently reuses intermediate results to compute them. An MILP formulation was introduced whose optimal solution yields the most efficient computation in the given framework, and a heuristic was presented to approximately solve the optimization problem much faster even for large problem dimensions. Numerical experiments were presented that showed a speedup of $14 \times$ in an ADMM solver architecture due to the novel projection method when compared to the state-of-the-art.

Acknowledgements

This work was supported by the Swiss Commission for Technology and Innovation (Grant 16946.1 PFIW-IW). It was also supported in part by ABB Schweiz AG (Contract: U1045).

References


