Code and Circuit Generation for Efficient Projection Computations on Embedded Platforms

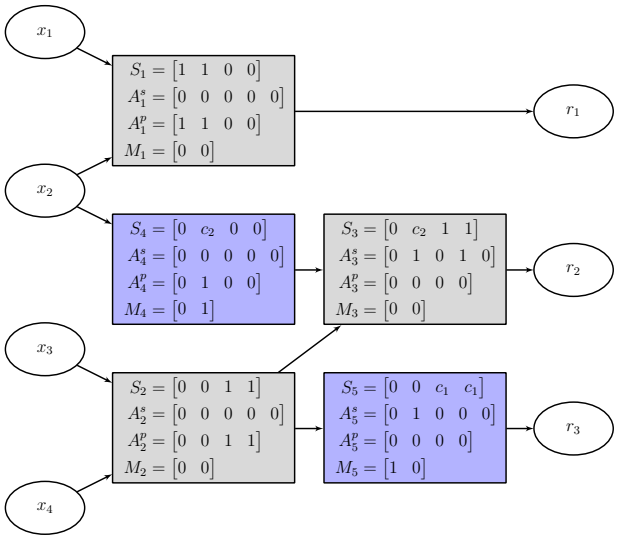
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ifa

Federal Institute of Technology (ETH) Zurich

Automatic Control Laboratory

Code and Circuit Generation for Efficient Projection Computations on Embedded Platforms



Master Thesis, April 2014

Author:
Sandro Merkli

Supervisors:
Dr. Juan Jerez
Prof. Dr. Manfred Morari

Abstract

Minimum Euclidian norm projection onto polyhedral sets is a core operation in first–order methods for convex optimization problems with convex affine inequality constraints. This projection is performed at every iteration of the solver, which makes its efficiency critical for the overall performance. For many practical problems in embedded optimization, this projection can be explicitly written as an evaluation of a piecewise affine function. State–of–the–art architectures evaluate such functions by iteratively traversing binary trees. This thesis describes a novel approach that uses graph representations of the computations to be performed to optimize operation sharing. 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 results are implemented in a software toolchain and quantitative comparisons with existing toolchains are made, demonstrating the effectiveness of the approach.

Acknowledgements

I would primarily like to thank Juan Jerez and Alexander Domahidi for their continued support and inputs throughout my work on this thesis. Also, I would like to thank Prof. Manfred Morari for the opportunity to do my masters thesis at the automatic control laboratory.
I am also thankful for the various tips and inputs I received from the people at the automatic control institute, particularly Martin Herceg, Alexander Fuchs, Damian Frick and Sébastien Mariéthoz.

Contents

1. Introduction

Euclidian projection onto convex polyhedral sets is an operation often encountered in numerical optimization, particularly in gradient methods. If the target set is given by $\{x \,|\, Gx \leq h\}$ with $x \in \mathbb{R}^n$, the projection is itself a quadratic program:

$$\begin{aligned} \mathcal{P}(x) = \arg\min_{y} \quad & \frac{1}{2} \|x - y\|_2^2 \\ \text{s.t.} \quad & Gy \leq h. \end{aligned} \tag{1.1}$$

The projection can be written as a piecewise affine (PWA) function in x [24]. It is therefore given as a polyhedral partition of $\mathbb{R}^n$ with different affine functions for every region:

$$\mathcal{P}(x) = \begin{cases} F_1 x + g_1, & A_1 x \leq b_1 \\ F_2 x + g_2, & A_2 x \leq b_2 \\ \quad \vdots & \\ F_k x + g_k, & A_k x \leq b_k \end{cases} \tag{1.2}$$

The computation of (1.2), which is referred to as the *parametric solution*, is *NP*–hard and the number of regions k usually grows exponentially with n. Nevertheless, practically relevant low–dimensional examples exist such as the projections required in multistage model predicitive control (MPC).

Recently, MPC has seen a large rise in popularity also for fast dynamical systems due to its inherent ability to treat constraints in a systematic manner. It is based on solving optimization problems that incorporate both a system model as well as the control constraints to get optimal performance. A large body of literature exists on methods improving the algorithms used for solving these optimization problems, but the computational complexity of MPC controllers for fast sampled systems is still one of their biggest challenges. For small systems, the method of choice is explicit MPC, which solves the full MPC problem offline as a multiparametric program. The online controller implementation then reduces to evaluating a piecewise affine function in x. However, the complexity of explicit controllers grows exponentially in the system size, which is why in the last decade, iterative optimization algorithms were investigated which solve the MPC problem at every controller sampling step.

Common approaches for the online solution of MPC problems include interior point and gradient methods. Interior point methods incorporate the inequality constraints into the objective function using a barrier function with an iteratively decreasing weight. Code generators exist (see for example [15]) to create code tailored to a given MPC problem. Efficient methods for multistage problems are outlined in [14]. They do not require projection onto the constraints, which is why they are not outlined in more detail here. Gradient methods solve convex optimization problems by following the negative gradient of the objective function. If the problem is constrained, this step is followed by a projection onto the feasible set. Many variations of this idea exist and a good

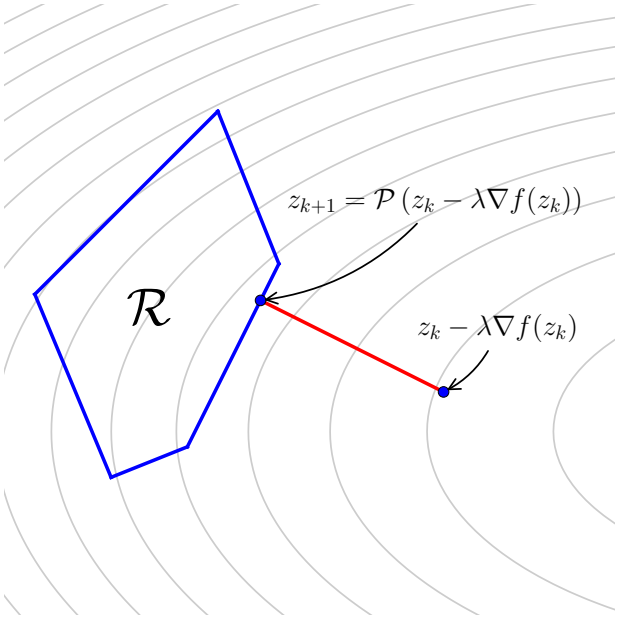


Figure 1.1.: Projection example in the context of gradient–based solvers

introduction is presented in [9], Chapter 9.3. Gradient methods yield small implementations and run fast (see [22]), though their dependence on the condition of the Hessian of the cost function poses a challenge. If the feasible set is a general polyhedral set, the projection onto it becomes a relevant issue as well. Consider for example the following optimization problem:

$$\begin{aligned} \min_{z} \quad & f(z) \\ \text{s.t.} \quad & Gz \leq h \end{aligned} \tag{1.3}$$

with $f(z)$ convex and differentiable and $z \in \mathbb{R}^n$. Denote the feasible set as $\mathcal{R} \coloneqq \left\{ z \,|\, Gz \leq h \right\}$. A classical gradient iteration for this problem is

$$z_{k+1} = \mathcal{P}\left(z_k - \lambda \nabla f(z_k)\right) \tag{1.4}$$

where λ is the step size and $\mathcal{P}$ is the projection as defined in (1.1). For an input point z, the projection operator returns y^*, the closest point to z that satisfies the constraints. An example is depicted in Fig. 1.1. Modern methods like the one used in [22] implement (1.4) with several extensions, but all of them require the projection operator $\mathcal{P}$ to be evaluated at every iteration. Since such solvers require about 5 to 20 iterations to converge, projection evaluation has to be at least an order of magnitude faster than the target controller frequency.

In multistage MPC, the constraint set is block diagonal and often repeated, which means the projection can be broken down into independent lower–dimensional projections of the stage variables. This makes the use of parametric projection solutions a tractable option. Additionally, for every iteration of the gradient solver, as many independent projections as there are stages are evaluated, which means pipelined hardware architectures would be beneficial.

While methods exist for the efficient evaluation of general PWA functions, projection computations have more structure that can be efficiently exploited by operation reuse. Intuitively,

many operations required for the location of a point among the regions in (1.2) are required again to evaluate the projection function and therefore only have to be computed once. In order to demonstrate the potential of operation reuse, consider the following example of a matrix–vector multiplication:

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 2 & 3 & 4 & 0 \\ 2 & 3 & 4 & 5 \end{bmatrix} x := Qx, \quad x \in \mathbb{R}^4 \tag{1.5}$$

Evaluated without any assumptions on Q, this would require 16 multiplications and 12 sums. However, the given product can be computed using 4 multiplications and 3 sums: The first entry of x is multiplied by 2, at which point the first entry of the result vector is already available. Next, the second entry of x is multiplied by 3 and added to the previous result, yielding the second result and so on. As demonstrated, implementations that are designed to take advantage of operation reuse could yield a performance increase over general approaches. This thesis investigates potential performance improvements through such computation reuse. It focuses on PWA functions arising from projection computations and implements the theoretical results in the form of a software toolchain for automatic digital circuit generation.

1.1. Previous work

Methods and software exist to derive the piecewise affine function for given projection functions, for example the *Multi-parametric Toolbox 3.0 (MPT)* [20], which is used as a starting point in this thesis. MPT also provides C and MATLAB code generation for PWA function evaluation. Automatic circuit generation for general PWA functions is presented in [26].

In terms of optimizing operation sharing, most approaches originate from research in digital circuit design related to constant coefficient filter implementations: The special case of multiplying a single input number by many different fixed–point coefficients is referred to as *multiple constant multiplication* (MCM) in literature (see [5], [10], [19]). The solution approaches are collectively called *common subexpression elimination*. Recently, there have been initial efforts towards exploiting intermediate result reuse across several inputs [8]. The methods mentioned exclusively deal with parallel fixed–point platforms like FPGAs and ASICs. They make use of the fact that multiplications by fixed–point constants can be written as sums of shifted versions of the input number. They therefore essentially convert the MCM problem into one of optimizing a sequence of additions. The problem of optimizing a sequence of additions of the same set of inputs for the minimum number of operations has been shown to be *NP*–hard (see [11]). The optimal result sharing problem investigated in this thesis differs from what has been investigated already in several aspects: Primarily, this work also considers floating–point numbers and fully serial platforms. Also, aside from the heuristics presented in [8], to the best knowledge of the author, there exist no MCM algorithms that deal with multiple variables.

Two state–of–the–art implementations of code generation for C and VHDL code were selected for comparisons with the toolchain implemented in this thesis. The reference implementations are toolchains that work with general piecewise affine functions, whereas this thesis focuses on PWA functions that arise from projection onto some practically relevant sets. These functions have a lot of structure, which the toolchain developed in this thesis is able to exploit. For VHDL code, the *MOBY-DIC* project [3] was selected. MOBY–DIC was developed as a toolbox that creates

circuitry for the evaluation of general PWA functions. It was used as a benchmark for the VHDL backend of the toolchain developed in this thesis. Another toolchain that generates VHDL code for PWA functions is presented in [25]. It does not perform any operation sharing, however. For C code, the code generation feature of MPT [20] was used. Both reference implementations perform point location using binary decision trees, followed by evaluation of the projection for the region in which the point was located. More details about these reference implementations is given in Sections 7.2 and 7.3, where quantitative comparisons are made between them and the software written in this thesis.

1.2. Contribution

This thesis explores a novel approach to evaluating piecewise affine functions that is particularly efficient for projection computations. Instead of iteratively computing the required hyperplane evaluations and then evaluating one projection function, this approach evaluates all hyperplanes and all projection functions concurrently, leading to increased speed. To gain efficiency despite the tremendous increase in the number of results that have to be computed, the order of operations is chosen such that many intermediate results can be reused. In addition to this idea,

- A mixed integer linear programming method is proposed to find the optimal operation sharing. Even though solving this MILP for larger practical problems is currently intractable, smaller problems can be solved. A heuristic is introduced to obtain approximate solutions for all problem instances.
- An error analysis is conducted to give worst–case bounds for the fixed–point implementations of the projection computation.
- The theoretical results are implemented in a complete software toolchain that, given a set to project on, creates ready–to–use VHDL and C implementations of an efficient projection architecture.

1.3. Outline

The first part of the thesis deals with the theoretical aspects of affine projections: the first chapter describes the structure of affine projection computations and outlines what approach will be taken in this thesis to implement it; Chapters 2 through 4 introduce the concept of operation graphs and two methods to optimize these graphs for different cost functions; Chapter 5 treats the effects of fixed–point arithmetic on hardware implementations of the projection. In the second part, the toolchain architecture and selected hardware targets are presented. The final part presents benchmark comparisons with state–of–the–art implementations and gives an outlook as to how the implemented toolchain could be extended and improved.

Part I.

Theory

2. Projection computation: Analysis and architecture

This chapter details on the required computations for projections and suggests some methods that can be applied to decrease the amount of computations. The first section outlines the conventional method of evaluating PWA functions, followed by a motivating example that demonstrates the effectiveness of operation sharing in projections. After that, a section that generalizes the insights gained in the example follows. Next, an overview of the architecture to be implemented is presented. The chapter then concludes with the introduction of graphs as a representation of matrix–vector products.

2.1. Piecewise affine function evaluation

As mentioned in the Chapter 1, the projection computation can be written as a PWA function in the form of (1.2). The problem therefore becomes one of evaluating a PWA function, which is discussed extensively in literature related to explicit MPC (see for example [16], [26]). The regions $A_i x \leq b_i \ \ (i = 1, \ldots, k)$ in (1.2) are collectively referred to as *region set*, whereas the $F_i x + g_i \ \ (i = 1, \ldots, k)$ are the corresponding *projection rules*. The usual approach taken consists of two steps:

1. Location of the input point x_0 in the region set
2. Application of the projection rule for the region in which the point is

While the second step is straightforward to implement (one evaluation of an affine function), the point location problem is challenging due to the fact that the number of regions generally grows exponentially in the number of problem dimensions.

The classical method used for point location (used for example in [26], [25]) is a decision tree traversal for point location. In this method, a decision tree is built with a hyperplane evaluation $c_i = a_i^T x - b_i$ assigned to every node i. Tree traversal starts at the root. At every node, the corresponding hyperplane evaluation is performed and a child is selected depending on the sign of c_i. Once a leaf is reached, the region in which the point is is found. Decision tree creation algorithms are still an active research topic. A recent one is outlined in [16], and MPT [20] is a software package that can generate PWA functions as well as decision trees directly. In the best case, a tree of logarithmic depth can be created for a given PWA function. Therefore, if the function has k regions, the tree has depth $\log_2 k$ and at most $\log_2 k$ hyperplane evaluations are required to find the region in which a point is. In practice, this optimal tree depth is often not reached, however. The method in [25] flattens the tree by a factor of $M \in \mathbb{N}$, by evaluating M hyperplanes per node. On the other hand, the implementation from [25] introduces an M–fold increase in hardware size.

In the next section, an example is used to demonstrate that in specific cases, alternative approaches to the classical PWA function evaluation can be very efficient.

2.2. Projection example

A set of practical relevance is the case where

$$0 \leq x_1 \leq x_2 \leq \cdots \leq x_i \leq x^* \tag{2.1}$$

has to hold. For $i = 2$, this reduces to the set shown in Fig. 2.1. This section will first outline the classical approach of computing a projection onto this set, with an analysis of the number of operations required. Then, an alternative approach is presented with the analogous analysis and compared.

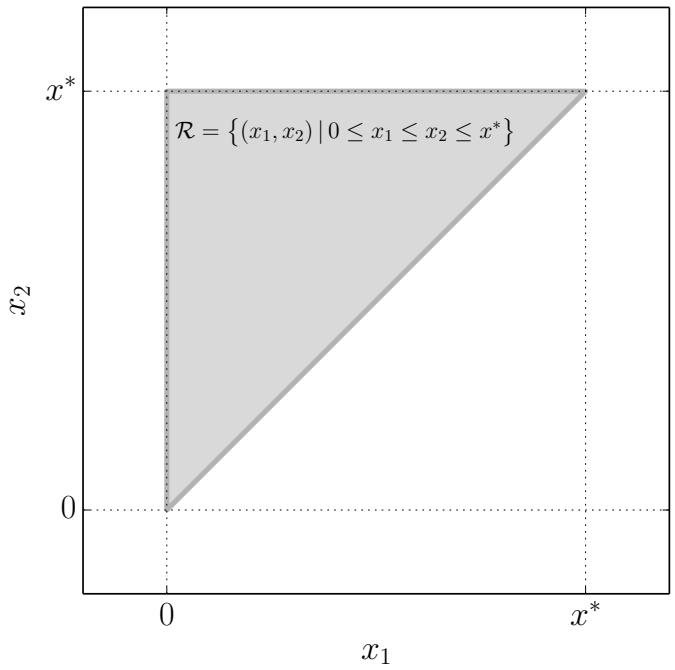


Figure 2.1.: Two–dimensional set to project on. Notice the alignment with the coordinate axis and the unity slope of the only slanted line.

2.2.1. Classical approach: Binary tree traversal

If this example is solved parametrically, the result is 7 regions and associated computation rules, as depicted in Fig. 2.2. The point location step using binary trees (as outlined in [16]) involves at least $\lceil \log_2 7 \rceil = 3$ hyperplane evaluations to find out in which region the point lies, and only then can the projection computation be performed. This means the amount of operations performed in the best case would be three hyperplane evaluations and one projection rule computation. In N_x dimensions, a hyperplane evaluation requires N_x multiplications, $N_x - 1$ additions and a subtraction. Since subtractions and additions involve the same cost on most platforms, the operation amounts for a hyperplane evaluation will simply be referred to as N_x multiplications and N_x sums. Finally, the projection computation has the form $Fx + g$ for some $F \in \mathbb{R}^{N_x \times N_x}$ and $g \in \mathbb{R}^{N_x}$ and requires N_x^2 multiplications and N_x^2 sums. For the example in 2 dimensions, this brings the total amount of operations to:

$$\text{ops}_c = 3\,(2\,\text{mul} + 2\,\text{add}) + 4\,\text{mul} + 4\,\text{add} = 10\,\text{mul} + 10\,\text{add} \tag{2.2}$$

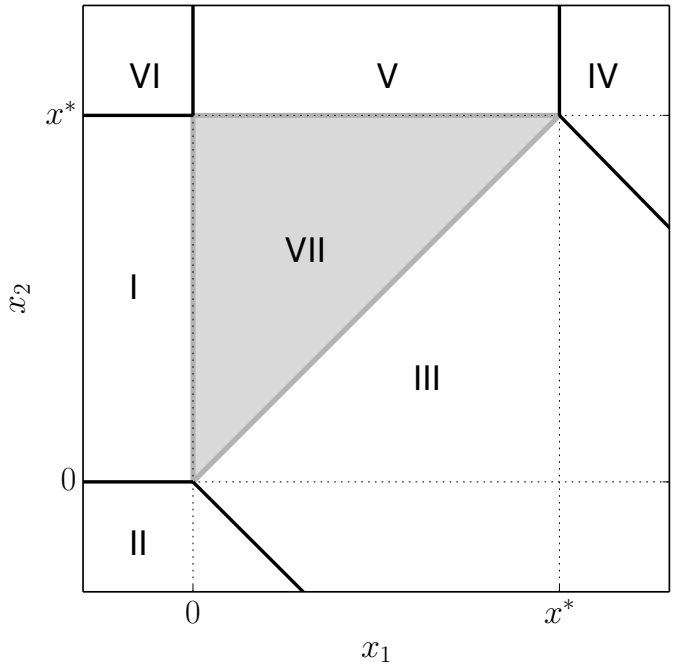


Figure 2.2.: Regions generated by the conventional approach

where "mul" denotes a multiplication and "add" an addition.

2.2.2. Direct approach: Combined computation

Upon closer investigation of the set shown in Fig. 2.1, one can notice that the only possible components of the result are elements of $\{x_1^0, x_2^0, x^*, 0, (x_1^0 + x_2^0)/2\}$, where x_1^0, x_2^0 denote the inputs to the projection. This means all the projection computations can be performed using one addition and a division by two, which is just a bit shift in fixed–point arithmetic. Looking at what hyperplane evaluations are to be performed, one can see that in addition to the aforementioned computations, $(x_1^0 - x_*)$, $(x_2^0 - x_*)$, $(x_1^* - x_2^*)$ and $(x_1^0 + x_2^0 - 2x^*)$ are required as well. Overall, the operations required sum up to

$$\text{ops}_d = 5\,\text{add} \tag{2.3}$$

since bit shifts are free in hardware. While there is still a mechanism required to select the correct results among these computed terms, the number of computations to be performed overall has decreased significantly. Note that based on the computed terms, the binary tree traversal can still be performed, however now all results are already available. Additionally, if the projection is implemented in a circuit, the binary decision tree based on the precomputed results can be implemented as a single–cycle muxing network to select the correct projection result for the corresponding region in which the point is located.

2.3. Simplifications specific to projections

2.3.1. Region merging

As already shown in the 2D example, the projection of a single element in x often has the same result for multiple regions. This fact can be exploited more systematically by, on a per element basis, collecting all regions with the same outcome into one so–called *superregion*. This reduces the number of hyperplane evaluations to be performed for the decision of one element compared to checking all regions individually, since hyperplanes that decide between regions which are part of the same superregion become unnecessary. This simplifies the muxing logic. However, since this is done for every element separately, the total number of hyperplane evaluations is not necessarily smaller than it would be if all of the original regions were checked separately.

2.3.2. Sharing operations between hyperplane and projection rule evaluations

Both hyperplane and projection rule evaluations in N_x dimensions can be written in the form

$$c^T x + d \tag{2.4}$$

for some $c \in \mathbb{R}^{N_x}$ and some $d \in \mathbb{R}$. As the previous investigations suggest, some terms are required for hyperplane evaluations as well as for projection computations. To exploit this, a matrix U is defined containing all the hyperplane evaluations and rule computations as rows:

$$U = \begin{bmatrix} c_1^T & d_1 \\ c_2^T & d_2 \\ \vdots & \vdots \end{bmatrix} \tag{2.5}$$

The resulting matrix U for a projection has N_x+1 columns and at most as many rows as there are hyperplane evaluations and projection rules. The projection then becomes a task of evaluating

$$r = U \begin{bmatrix} x \\ 1 \end{bmatrix} \tag{2.6}$$

and after that using the results associated with hyperplane evaluations to select the entries of the projected point from the results pertaining to projection rule computations. For example, consider a projection that would require hyperplane evaluations $a_i^T x_0 \leq b_i$, $i \in \{1,2,3\}$ and projection computations $f_i^T x_0 + g_i$, $i \in \{1,2\}$. The initial matrix U would then become

$$U = \begin{bmatrix} a_1^T & -b_1 \\ a_2^T & -b_2 \\ a_3^T & -b_3 \\ f_1^T & g_1 \\ f_2^T & g_2 \end{bmatrix} \tag{2.7}$$

This matrix format is convenient in several ways: Computations that are done for multiple regions can be readily identified as equal rows in U, whereas some rows do not require any computations at all. If two rows i, j are linearly dependent and at least one of them represents a hyperplane evaluation, one of them is a scaled version of the other and can be removed. The computations for hyperplane evaluations can further be simplified by scaling the hyperplanes by a positive scalar to make as many of their coefficients equal to 1 as possible. As can be easily seen, the applicability

of the simplifications listed here depends strongly on the actual hyperplane evaluations and functions to be computed. This method can also be applied to general PWA functions, however it is more effective if hyperplane evaluations and projection rules require similar results, which is the case in projection computations for many practically relevant sets.

2.4. Projection computation architecture

2.4.1. Basic concept

The investigations shown in the previous example and the selection of an FPGA as primary hardware platform suggest an architecture that favors parallelism. A high–level overview of the architecture implemented in this thesis is shown in Fig. 2.3. Essentially, the projection computation is separated into two steps:

1. The full product $r = U \begin{bmatrix} x & 1 \end{bmatrix}^T$ as defined in (2.6) is computed, reusing as many intermediate results as possible
2. Based on the signs of the entries of r, the components y_i of the projected point vector y are assigned entries of r. The function "sel" deciding which entries of r to use for which position of y is a muxing network.

The entries of r cannot be blocked into hyperplane evaluation results and projection results, since part of them are used for both hyperplane evaluations and projection rules. In a hardware implementation, the second step can be done in a single clock cycle even for the largest projection computations considered for this thesis. This is why most of the theoretical focus is put on finding efficient implementations of the computation block: Since U is constant, the order in the computation block can be chosen to reuse intermediate results, thereby reducing the amount of operations required. The method employed to increase reuse is optimizing operation graphs, which will be introduced in Section 2.5.

2.4.2. Limitations of our approach

This approach has obvious drawbacks compared to the conventional tree–based approach outlined in [26] and others. Particularly, the computation block performs all hyperplane and projection rule evaluations concurrently, whereas the existing solutions only evaluate a fraction of them by traversing a decision tree. On the other hand, existing implementations have to wait for a hyperplane computation to be completed before the control logic can choose a new one. This means that while the worst–case time a projection takes in the conventional approaches is the decision tree depth times the time required for a general hyperplane evaluation, the computation (2.6) can be performed in the time it requires to do a single hyperplane evaluation due to concurrency. This concurrency becomes possible since no decisions have to be made during the computation.

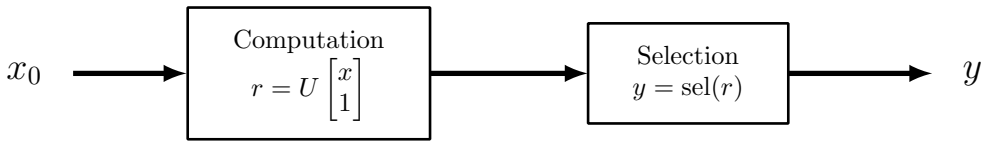


Figure 2.3.: Abstract architecture overview

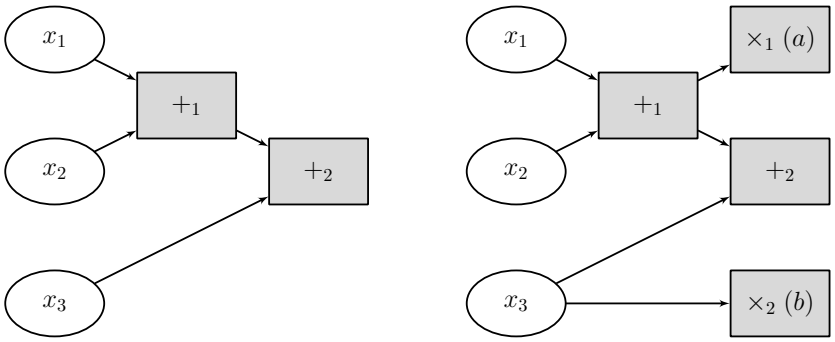


Figure 2.4.: Two example operation graphs

2.5. Operation graphs

This section introduces *operation graphs* as a concept of representing a sequence of computations on a common set of inputs, explicitly representing operation order. This order of operations is vital for operation sharing. The concept of operation graphs is widely used in fields like software compiler design, however this thesis deals with much a more limited set of operations than other research.

2.5.1. Using directed graphs to represent affine computations

In graph theory, a general directed graph is a set $\mathcal{V}$ of vertices and a set $\mathcal{E}$ of edges connecting them. If a vertex only has edges going away from it, it is called a *source*, if it only has edges going towards it, it is called a *sink*. More theory on graphs can be found in textbooks, for example in [12]. In order to use graphs to represent affine computations, the aforementioned components of a directed graph are assigned specific meanings:

- *Source vertices* are scalar input variables
- *Non–source vertices* are operations: addition, multiplication
- *Edges* represent data flow

Sinks are not assigned any special meaning here, they simply represent operations as well. Consider a simple example shown in the left part of Fig. 2.4. This graph has three sources representing the input variables x_1, x_2, x_3 and two non–source vertices representing additions. Every vertex in the graph represents a symbolic expression of the inputs: Source vertices trivially represent the corresponding input variable, and non–source vertices represent the result they compute. In the example, vertex $+_1$ represents $(x_1 + x_2)$ and vertex $+_2$ represents $(x_1 + x_2) + x_3 = (x_1 + x_2 + x_3)$. This means the complete set of results computed by the graph in the left part of Fig. 2.4 is:

$$\big\{x_1, x_2, x_3, (x_1 + x_2), (x_1 + x_2 + x_3)\big\} \tag{2.8}$$

with every result corresponding to a vertex in the graph. A slightly more complex example is shown on the right side of Fig. 2.4. This example also involves two product nodes, $\times_1$ and $\times_2$. Products in linear algebra have one constant and one variable operand, which is why product nodes only have one edge going to them. The constant with which the input is multiplied is assigned to the node and denoted in brackets. This means in addition to the expressions listed in (2.8), the example on the right also computes $(ax_1 + ax_2)$ and (bx_3).

2.5.2. Order of operations and latency

The graphs shown in Fig. 2.4 already suggest that there is an inherent order of operations. For example, $+_2$ cannot be computed before $+_1$, since the result of $+_1$ is required as an input to $+_2$. This section quantifies this suggestion. A reasonable abstraction for actual implementations of a computation is that there are *computational cycles*, in each of which a (possibly limited) number of operations can be performed in parallel. Latency in general is defined to be the delay (in computational cycles) between the input variable values becoming available and a result being computed. If no constraint is given on the amount of operations that can be performed per computational cycle, the *intrinsic* latency of a vertex V_i in the graph is defined as the maximum path length of any of the sources to V_i. This is due to the fact that an operation can only be performed once all its inputs have been calculated. Note that the intrinsic latencies can be extracted from an operation graph using recursive graph traversal.

2.5.3. Operation graph optimization: Problem definition

Assume the purpose of the graphs is to represent a matrix–vector multiplication shown in (2.6) with the dimensions defined there. Define $G_U(x)$ to mean "an operation graph describing the computation of (2.6)". The vector x will be referred to as the *input* and the result of (2.6) will be referred to as the *output* or *result*. The problem of finding an optimal operation graph for a specified cost function f (for example, the least amount of operations) can be written as follows:

$$\begin{aligned} \min_{G_U} \quad & f\left(G_U\right) \\ \text{s.t.} \quad & G_U(x) = U \begin{bmatrix} x^T & 1 \end{bmatrix}^T \end{aligned} \tag{2.9}$$

where the constraint enforces correct computation. The next two chapters present two approaches to solve this optimization problem exactly using mixed–integer programming and approximately using a heuristic. They apply slightly different implementations of the graph descriptions, however the basics outlined in this section apply for both.

3. Graph optimization using Mixed–Integer Linear Programming

3.1. Introduction

This chapter introduces a formulation of (2.9) that is compatible with common MILP solvers. The formulation contains all aspects of the described operation graphs explicitly, allowing for a wide variety of optimization goals and extensions. A mixed–integer linear programming approach has already been taken for adder–only graphs in [21], however instead of extending this approach to multiplications of multiple different inputs and floating point numbers, the approach taken here is to introduce a new formulation based on similar ideas.

3.2. Assumptions

A core assumption made at this level of abstraction is that no intermediate result is computed twice, regardless of what the optimization goal is. Whenever an intermediate result is required multiple times, all occurrences of this result can be replaced by the one computation with the lowest latency. This leads to the following:

All computed intermediate and final results are different from each other. (3.1)

In order to derive an upper bound for the operations to be performed (and with this, an upper bound on how many vertices can be in the graph), a few constants have to be introduced. Assume the matrix U from (2.6) has N_e nonzero entries and among those entries, there are N_α different numbers. Assume also that there are N_m non–zero entries different from one. This means

$$N_\alpha \leq N_m \leq N_e \leq N_x N_r. \tag{3.2}$$

For the upper bound of operations to be performed, assume the worst case, meaning no intermediate result can be used more than once. Since every row of the matrix denotes a linear combination, which requires one less addition than the number of terms it sums up, the worst case total number of additions required is

$$N_a = N_e - N_r. \tag{3.3}$$

Multiplications can possibly be shared across sums, but in the worst case, every variable with a non–zero coefficient unequal to one in every result term has to be multiplied separately, yielding a total number of operations of at most

$$N_v = N_e - N_r + N_m \tag{3.4}$$

This means that any formulation that allows for at least N_v nodes to be present in the resulting graph will be able to represent a feasible solution.

3.3. Variables

The description of an operation graph G_U is denoted as a set $\mathcal{V}$ of tuples $V_i \in \mathcal{V}$. The index $i \in 1, \ldots, N_v$ is used to refer to vertex i in the graph and this index is propagated to everything related to this vertex. If a variable has such an index, an instance of this variable exists for every node separately.

3.3.1. Describing intermediate results

Every vertex V_i represents an intermediate result represented by the variable S_i. Since all that is done in linear computations are sums and products, S_i can be defined as a vector with the same length as the input vector.

$$S_i \coloneqq \begin{bmatrix} b_{i1} & b_{i2} & \cdots & b_{iN_x} \end{bmatrix} \in \mathbb{R}^{N_x} \tag{3.5}$$

The intermediate result described by a given S_i can then be computed as $S_i x$. For example, if $x := \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$, this means $S \in \mathbb{R}^3$. If vertex i were to represent $x_1 + x_2$, this would be written as $S_i = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$, and similarly a linear combination $2.5x_2 + 8x_3$ would be represented by $S_i = \begin{bmatrix} 0 & 2.5 & 8 \end{bmatrix}$. Note that the S_i only express the result of an operation, they do not specify how this result has come to be. For later use, a matrix that stacks the intermediate results for all the vertices (denoted the intermediate result matrix) is defined here:

$$\mathcal{S} \coloneqq \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{N_v} \\ I \end{bmatrix} \tag{3.6}$$

where $I \in \mathbb{R}^{N_x \times N_x}$ is the identity matrix. The vector $\mathcal{S}x$ contains all the intermediate results computed in the graph as well as the inputs.

3.3.2. Describing computations leading to intermediate results

In order to describe how the intermediate result of a vertex V_i is computed, a variable C_i is defined here. In graph–theoretical terms, this C_i specifies the incoming edges to vertex V_i. The possible data sources for computation inputs are primary inputs ($x_1, x_2, \ldots, x_{N_x}$), previous intermediate results (referred to as secondary inputs) or combinations of the two. Notice that the previously defined $\mathcal{S}$ conveniently captures all these input possibilities at once. Using these inputs, either sums or multiplications by constants can be computed. In order to represent all possible combinations, let C_i be a tuple

$$C_i = \left(A_i, M_i, z_i, \mu_i\right) \tag{3.7}$$

where A_i describes addition inputs and M_i describes multiplications. More specifically, A_i is a vector of binary numbers describing inputs to an addition:

$$\begin{aligned} A_i &\coloneqq \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{i(N_x+N_v)} \end{bmatrix} \in \mathbb{B}^{N_x+N_v} \\ &\coloneqq \begin{bmatrix} A_i^s & A_i^p \end{bmatrix} \end{aligned} \tag{3.8}$$

where $\mathbb{B} = \{0, 1\}$ denotes a binary number. The first N_v entries of A_i denote additions of secondary inputs, summarized as A_i^s, whereas the last N_x entries (summarized as A_i^p) denote

additions of primary inputs. Let M_i be another vector of binary numbers describing the selection of a coefficient for multiplication:

$$M_i \coloneqq \begin{bmatrix} m_{i1} & m_{i2} & \cdots & m_{iN_m} \end{bmatrix} \in \mathbb{B}^{N_m} \tag{3.9}$$

These two vectors together allow for the description of any of the possibilities outlined in the above list. In case vertex i is an addition, this addition can be written in terms of intermediate result descriptions as follows:

$$S_i = A_i \mathcal{S} \tag{3.10}$$

If, on the other hand, the vertex is a multiplication, it can be written as

$$S_i = \big(M_i Q\big) A_i \mathcal{S} \tag{3.11}$$

where $Q \coloneqq \begin{bmatrix} \alpha_1 & \cdots & \alpha_{N_m} \end{bmatrix}^T \in \mathbb{R}^{N_\alpha}$ is a vector of all the unique coefficients present in U. The selection whether the vertex is a multiplication or an addition is made by the variable $\mu_i \in \mathbb{B}$:

$$\begin{aligned} \mu_i = 1 &\iff \text{Vertex } i \text{ represents a multiplication} \\ \mu_i = 0 &\iff \text{Vertex } i \text{ represents an addition} \end{aligned} \tag{3.12}$$

For software implementation reasons, the operation graph representation is defined to have a fixed number of vertices N_v as previously mentioned. If some intermediate results are reused, part of the vertices are not used in computations. Unused vertices are marked by the flag $z_i \in \mathbb{B}$ as follows:

$$\begin{aligned} z_i = 1 &\iff \text{Vertex } i \text{ is unused} \\ z_i = 0 &\iff \text{Vertex } i \text{ is in use} \end{aligned} \tag{3.13}$$

3.3.3. Latency description

As mentioned in Section 2.5.2, there is an inherent concept of latency in the operation graph: If traversal of one edge is assigned a latency of 1, the latency of a vertex (defined as the distance of the vertex in computational cycles from the inputs) is simply the cardinality of the longest path present from any of the inputs to the vertex. However, if the number of computational units available in parallel is less than the number of operations that could be performed at that time, this inherent concept of latency becomes less relevant: Even though a vertex might only be 3 edges away from any of the inputs, there might not be any computational unit available in the third cycle to get to that vertex. To express any non–inherent latency descriptions, define $l_i \in \mathbb{N}_+$ as the latency of the vertex i. This means that the operation that is represented by vertex i is computed in the l_i–th cycle of the whole computation.

3.3.4. Graph representation example

In order to make the meaning of the variables defined in this section more clear, a graph representation example is presented here. The computation used for the example is

$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & c_2 & 1 & 1 \\ 0 & 0 & c_1 & c_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{3.14}$$

This example results in an operation graph as shown in Fig. 3.1 on page 22. Note that this is not the only graph that results in (3.14). First of all, the vertices in this graph are assigned numbers,

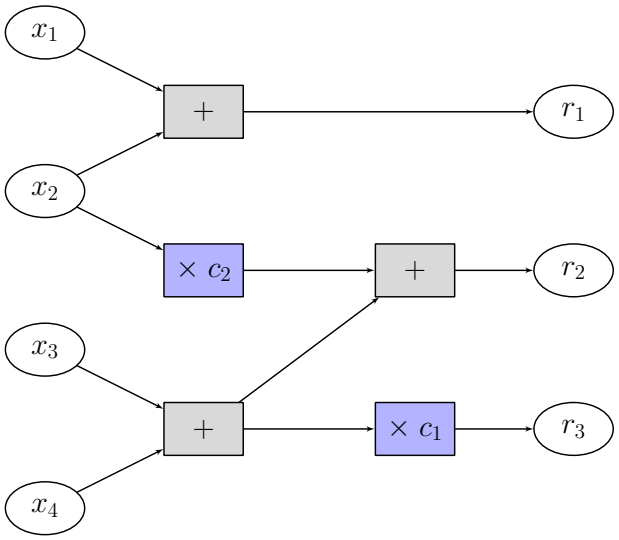


Figure 3.1.: Example operation graph representing (3.14)

as shown in Fig. 3.2 on page 23. The S_i for the example graph are then shown in Fig. 3.3 on page 23. A version of the graph annotated with all the variables is shown in Fig. 3.4 on page 24, along with some explanatory notes.

3.4. Constraints

The variables introduced in the previous section have to be constrained in several ways such that every feasible solution of the optimization problem corresponds to a feasible adder graph implementing the computation (2.6). More specifically, this means the following:

1. Every result r_k, $k \in 1, \ldots, N_r$ from the result vector $r = \begin{bmatrix} r_1, \ldots, r_{N_r} \end{bmatrix}^T$ has to be present among the intermediate results
2. Every vertex V_i is constructable, i.e. if it is a sum, the specified inputs (selected by C_i) have to actually result in the specified sum (given by S_i) (and analogous constructability has to hold for products.)

While the first kind of constraint listed above is easy to implement, the constructability requirement is more involved. The following subsections outline the implementation of these constraints using the variables defined in Section 3.3.

3.4.1. Constraints for correctness of computation

The same constructability and latency constraints hold for the end results as for the intermediate results, therefore they are simply treated the same. The presence of the results r_k can be ensured by constraining the first m intermediate results to be equal to these results:

$$S_k = u_k \qquad \forall k = 1, \ldots, N_r \tag{3.15}$$

where the u_k are the rows of U.

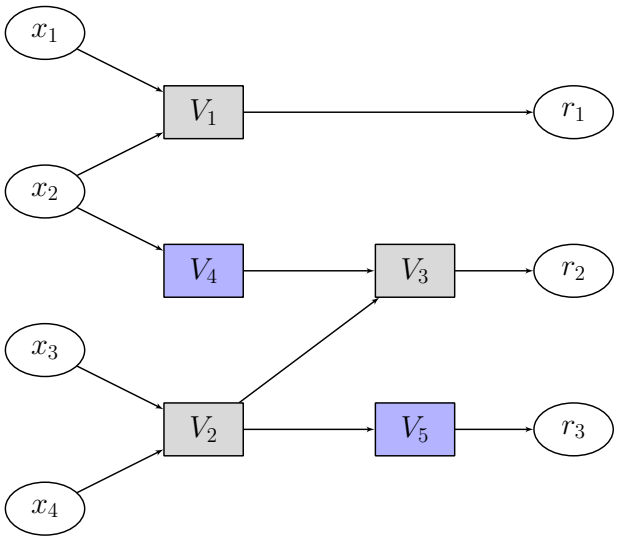


Figure 3.2.: Example operation graph: Vertices are given numbers

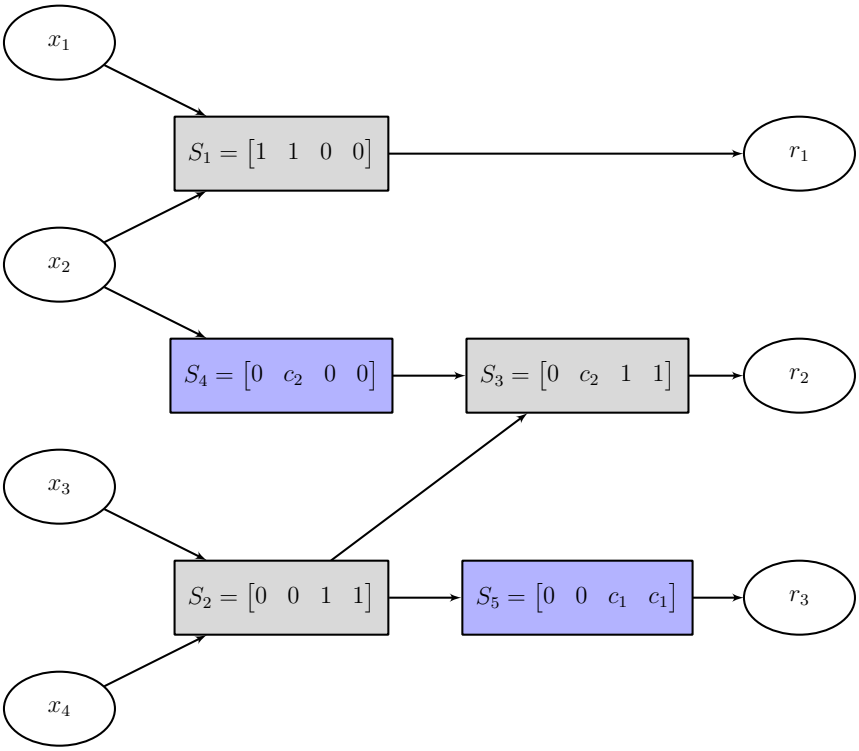


Figure 3.3.: Example operation graph: Intermediate result descriptions S_i added

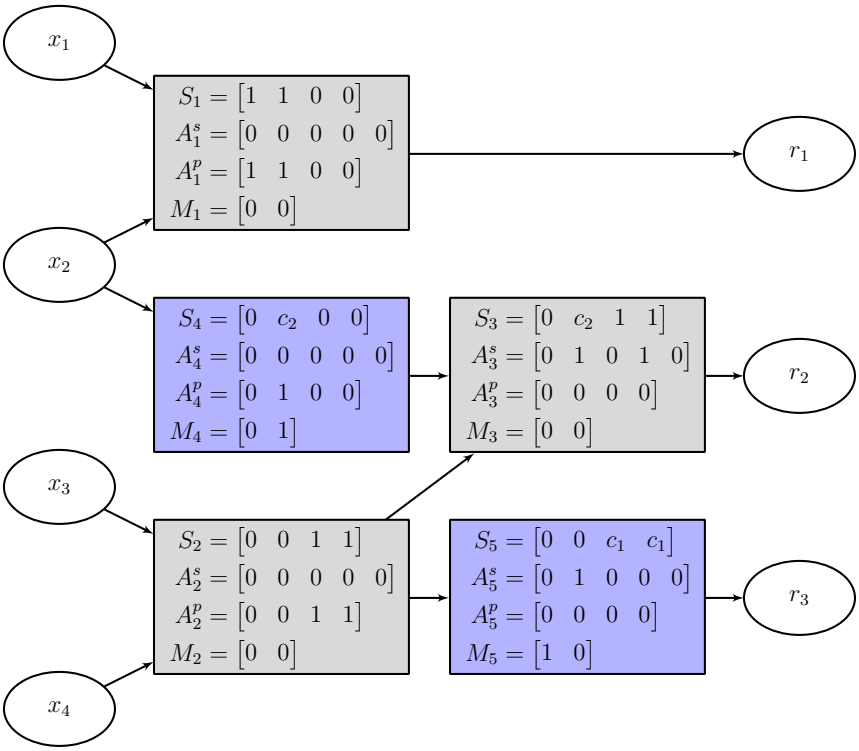


Figure 3.4.: **Example operation graph: Complete picture**
This figure shows the complete operation graph description for example (3.14). V_1 for example is a sum of x_1 and x_2. This means S_1 has ones at those positions. V_1 is constructed from two primary inputs (x_1 and x_2) at positions 1 and 2, which is why $A_1^p = [1\ 1\ 0\ 0]$. Since there are no secondary inputs summed up, A_1^s is all zeros.
Conversely, V_4 is a multiplication, which is why only one entry across A_4^p and A_4^s can be one. This single nonzero entries specifies the multiplication input (x_2 in this case). The 1 in M_4 specifies that this selected input is multiplied by c_2 to yield the $S_4 = [0\ c_2\ 0\ 0]$ which represents $c_2 x_2$.

3.4.2. Vertex constructability constraints

As noted in section 3.3.2, a vertex V_i can be constructed in several ways. If it is zero, the following holds:

$$S_i = 0 \tag{3.16}$$

If on the other hand it is a sum it can be written as

$$S_i = A_i\mathcal{S} = \sum_{j=1}^{N_v} a_{ij}S_j + A_i^p \tag{3.17}$$

Since (3.17) represents a sum, exactly 2 entries of A_i are 1 and all other entries 0. Mathematically speaking, this means: $\|A_i\|_1 = 2$. Finally, it can also be a multiplication, in which case it is written as

$$S_i = (M_iQ)A_i\mathcal{S} = \left(\sum_{k=1}^{N_\alpha} m_{ik}\alpha_k\right)\left(\sum_{j=1}^{N_v} a_{ij}S_j + A_i^p\right) \tag{3.18}$$

where only one entry of M_i is nonzero, as well as only one entry of A_i. With the help of z_i and μ_i, the three cases (3.16), (3.17) and (3.18) can be summarized into a compact form:

$$S_i = \left(1 - \mu_i + \sum_{k=1}^{N_\alpha} m_{ik}\alpha_k\right)\left(\sum_{j=1}^{N_v} a_{ij}S_j + A_i^p\right) \tag{3.19a}$$

$$\|A_i\|_1 + \|M_i\|_1 = 2(1 - z_i) \tag{3.19b}$$

$$\|M_i\|_1 = \mu_i \tag{3.19c}$$

If $\mu_i = 0$, all entries of M_i are 0 and (3.19a) becomes (3.17). If $z_i = 1$, everything becomes zero and (3.19a) becomes (3.16). Finally, in the case of a multiplication, $\mu_i = 1$ and the constraint becomes (3.18).

However, the product of two sums of variables in (3.19) is bilinear and therefore cannot be directly implemented in a MILP formulation. Products of binary and continuous variables can be implemented as shown in [7], but this approach requires an additional variable per product as well as four constraints. A more efficient way of implementing (3.19) is as follows:

$$\left|S_i - \sum_{j=0}^{N_v} a_{i,j}S_j + A_i^p\right| \leq \mathcal{M}\mu_{i,k} \tag{3.20a}$$

$$\left|\frac{S_i}{\alpha_k} - \sum_{j=0}^{N_v} a_{i,j}S_j + A_i^p\right| \leq \mathcal{M}(1 - m_{i,k}) \tag{3.20b}$$

$$\|A_i\|_1 + \|M_i\|_1 = 2(1 - z_i) \tag{3.20c}$$

$$\|M_i\|_1 = \mu_i \tag{3.20d}$$

In (3.20), Big–M reformulation is used (as already applied in [7]) with $\mathcal{M} \in \mathbb{R}$ larger than the absolute value of all elements of U. The choice of which entry of M_i is non–zero activates the corresponding constraint while turning the others off. Note that there is one instance of (3.20b) for every combination of vertex index i and unique coefficient index k. This means there are N_vN_α constraints like (3.20b), which emphasizes the importance of formulating efficiently. The

nonlinearity of (3.20) can now be formulated in the form of linear constraints: The sums in the aforementioned constraint are computed on a per–position basis, meaning (3.20a) can be written as N_x individual equations for all the posititions $p = 1, \ldots, N_x$ in this sum of vectors:

$$b_{i,p} - \sum_{j=0}^{N_v} a_{i,j} b_{j,p} + a_{i,(N_v+p)} = 0 \qquad \forall i \in 1, \ldots, N_v, \;\; \forall p \in 1, \ldots, N_x \tag{3.21}$$

Looking at (3.21), one can see that now only products of binary and continuous variables appear, which can be implemented using an auxiliary binary variable as demonstrated in [7]:

$$c = a \cdot b \iff \begin{cases} c \leq \mathcal{M} a \\ c \geq -\mathcal{M} a \\ c \leq b - (-\mathcal{M})(1-a) \\ c \geq b - \mathcal{M}(1-a) \end{cases} \tag{3.22}$$

where b and c are continuous variables and a is a binary one. For the case of only binary variables, this example simplifies to

$$c = a \cdot b = a \wedge b \iff \begin{cases} -a + c \leq 0 \\ -b + c \leq 0 \\ a + b - c \leq 1 \end{cases} \tag{3.23}$$

which is also shown in [7]. Applying this to the constraints in (3.21) results in a system of constraints linear in the variables, which is now an MILP, for which standard solution methods and software exist.

3.4.3. Latency constraints

The latency constraints ensure that the computation of an intermediate result only depends on other results computed before it. As mentioned earlier, every vertex i has a variable l_i associated to it that specifies at what point in time (measured in computational latency cycles) it is computed. If intermediate result j, denoted by S_j, is used in the computation of S_i, its latency l_j has to be strictly less than that of S_i, which is l_i. Another way of expressing this is:

$$l_j \geq l_i \implies a_{ij} = 0 \tag{3.24}$$

This is because $a_{ij} = 1$ means that S_j is involved in the computation for S_i. The implication above can also be expressed as a constraint suitable for MILP formulations:

$$l_i - l_j + \mathcal{M}(1 - a_{i,j}) \geq 1 \tag{3.25}$$

Again, instances of this constraint are added for all combinations of j and i, $j \neq i$.

3.5. Cost functions

Altogether, the constraints listed in Section 3.4 result in the only feasible sets of V_i being representations of valid operation graphs. This means that without a cost function, the above variables and constraints represent a feasibility problem. The latency level variables l_i are deliberately not

constrained to be the intrinsic latencies for the feasibility problem, since some cost functions might lead to different latencies. The descriptions of the vertices contain all information required about the graph: number of nonzero intermediate results, which results are constructed from which others and at what latency level which result is constructed. Therefore, a multitude of different cost functions can be constructed to optimize the graph towards different goals.

3.5.1. Minimum number of operations

Since z_i directly denotes whether intermediate result V_i is zero, the cost function for the minimum number of operations is simply

$$C_{\text{min add}}(\mathcal{V}) \coloneqq \max_{\mathcal{V}} \sum_{i=1}^{N_v} z_i \tag{3.26}$$

3.5.2. Optimal latency with the minimum number of operations

If there is no constraint on the amount of operations performed at the same time, the amount of computational cycles required to compute a scalar product $a^T x + b$ with $x \in \mathbb{R}_x^N$ can be given as

$$L_{\text{opt}} = 1 + \lceil \log_2 (N_x + 1) \rceil \tag{3.27}$$

where the first cycle is required to multiply all the entries of x with the corresponding entries of a and the other $\lceil \log_2 (N_x + 1) \rceil$ cycles are required to sum up the products using a symmetric adder tree. The problem can therefore be constrained to not have any vertices with latencies larger than L_{opt}:

$$l_i \leq L_{\text{opt}} \qquad \forall i \in 1, \ldots, N_v \tag{3.28}$$

The problem is then solved with the same cost function as in Section 3.5.1. This ensures the fastest computation of U possible with the least amount of operations.

3.5.3. Minimum worst–case latency for a given parallelism

This cost function yields the minimum worst–case latency with an additional constraint of at most P operations per latency level. For the introduction of such a cost function, an upper bound for the maximum latency is required. Since there are at most N_v operations, the absolute maximum possible latency is N_v. This is a very conservative upper bound, since it is for the case of no result reuse combined with a parallelism of $P = 1$. Nevertheless, using this bound ensures every feasible solution can be represented. In order to optimize the graph for minimum worst–case latency, a new auxiliary variable $L \in \mathbb{N}$ with the constraints

$$l_i \leq L \qquad \forall i \in 1, \ldots, N_v \tag{3.29}$$

is added. The cost function then becomes

$$C_{\text{min (max lat)}}(\mathcal{V}) \coloneqq \min_{\mathcal{V}} L \tag{3.30}$$

Additional constraints have to be added to ensure the maximum parallelism is not violated. First, introduce intermediate variables $\delta_{i,k} \in \mathbb{B}$ with $k, s = 1, \ldots, N_v$ which are defined as follows:

$$\delta_{i,s} = 1 \quad \Longleftrightarrow \quad l_i = s \tag{3.31}$$

This can be expressed as:

$$\begin{aligned} l_i - \sum_{s=1}^{N_v} \delta_{i,s} s = 0 \\ \sum_{s=1}^{N_v} \delta_{i,s} = 1 \end{aligned} \tag{3.32}$$

for $i = 1, \ldots, N_v$. Now the $\delta_{i,s}$ can in turn be used to express the actual parallelism constraint for each latency level s:

$$\sum_{i=1}^{N_v} \delta_{i,s} \leq P \quad \forall s \in 1, \ldots, N_v \tag{3.33}$$

3.6. Optimization problem: Complete formulation

The complete formulation of the MILP that implements the high–level optimization problem outlined in (2.9) is presented here. The variables are as defined in Section 3.3 and the cost function $C(\mathcal{V})$ can be chosen among those presented in Section 3.5.

$$\min_{\mathcal{V}} \quad C(\mathcal{V}) \tag{3.34a}$$

$$\text{s.t.} \quad S_h = u_h \tag{3.34b}$$

$$\left| S_i - \sum_{j=1}^{N_v} a_{i,j} S_j + A_i^p \right| \leq \mathcal{M} \mu_{i,k} \tag{3.34c}$$

$$\left| \frac{S_i}{\alpha_k} - \sum_{j=1}^{N_v} a_{i,j} S_j + A_i^p \right| \leq \mathcal{M}(1 - m_{i,k}) \tag{3.34d}$$

$$a_{ii} = 0 \tag{3.34e}$$

$$\|A_i\|_1 + \|M_i\|_1 = 2(1 - z_i) \tag{3.34f}$$

$$\|M_i\|_1 = \mu_i \tag{3.34g}$$

$$l_i - l_j + \mathcal{M}(1 - a_{i,j}) \geq 1 \tag{3.34h}$$

where $h = 1, \ldots, N_r$, $k = 1, \ldots, N_\alpha$, $i = 1, \ldots, N_v$, $j = 1, \ldots, N_v$.

4. Graph optimization using a heuristic

The biggest drawback of the approach outlined in Chapter 3 is that the solution of the resulting MILP takes very long even on high–end computing hardware. A significant speedup can be achieved by applying a heuristic to find a feasible solution to use as a starting point for the numerical optimizer. In cases where the problem description outlined in Chapter 3 becomes too large for a numerical optimizer to handle, the output of the heuristic can be used directly, in the hopes that it is at least close to the optimum.

The heuristic presented here follows the general idea of a greedy algorithm. Several similar algorithms have been implemented (see for example [5], [10], [19]) for adder–only operation graphs. Due to the differences to the problem at hand, a new heuristic implementation is introduced here inspired by them.

4.1. Basic algorithm structure

This algorithm follows the idea of "at each stage, do what locally appears best". The rows of U are added to the graph one at a time in a locally optimal way. By tuning the method used to add rows to the graph, this basic idea can be customized to yield graphs that are good starting points for different cost functions. The rows are built backwards: A result is decomposed into two operands, these operands are decomposed in turn and so on, similar to the algorithm in [5]. The backward building approach makes the incorporation of parallelism constraints non–trivial. This can be tackled by splitting the problem of optimizing for a given parallelism constraint into two stages: *Graph construction* and *graph scheduling*.

4.1.1. Data structure

The data structure used for the heuristic presented in the following is a reduced version of the representation used in Chapter 3. Conceptually, the same operation graph is used, but it is not implemented as a vertex list but rather as an actual recursive graph data structure. Nodes still represent intermediate results S_i, but the construction information is not explicitly stored in them anymore but rather derived from the edge set of the graph. Since an intermediate result is described by a vector (also called *row*), the subsequent sections use the terms *node*, *vertex* and *row* interchangeably.

4.1.2. Graph construction

The general structure of the generation algorithm is shown in Algorithm 1 on page 31: An empty graph is created and the rows of the matrix to be converted to a graph are added recursively one after another. The recursive row addition is implemented as outlined in Algorithm 2 on page 31. The specific implementations of *KeyFunction* and *DecomposeFunction* are then used to tweak the graph generation toward a certain goal.

Key function design

The key function decides in which order rows are added: A scalar key is first computed for every row. A generic sort function is then used to sort the rows by key function value. This can have a big influence on the efficiency of the heuristic: If one row is added before another, no information about the operations required in the latter is known to the algorithm during the inclusion of the former. Adding the rows in order of increasing complexity lead to efficient operation sharing in many practical cases. The complexity of a row is defined as the number of operations required (in the worst case) to compute it. An upper bound for the amount of operations h required for a row can be computed as

$$h_{\max} = h_+ + h_\times \tag{4.1}$$

where h_+ is the number of additions, which is the number of non–zero entries of the row minus one. The multiplications required are denoted by $h_\times$. This number is equal to the amount of non–zero, non–unity entries of the row. If this order of rows is used, the first rows added only require few operations, which means the amount of operations that is added without considering sharing is reduced.

Decomposition function design

Given a row R, the decision function decides whether to decompose this row into two summands to be added or into an operand and a factor for multiplication. This decision should be done in a way to maximize the number of operations reused from computations previously added to the graph. Assume a row R is to be added to the graph G, which already contains all the operations of previous row additions. The best case for the addition of this row is if only one operation is required to add it to the graph. This can happen in two scenarios:

- Two rows R_1, R_2 are already part of the graph and $R = R_1 + R_2$
- R is a product of a scalar and a row which is already part of the graph

If such a case is found, the one operation is added immediately, since there can be no better way to compute R. However, if neither of the above holds, R has to be decomposed into two rows R_1 and R_2, which in turn have to be added to the graph. Several methods can be used to select R_1 (and with $R_2 = R - R_1$ also R_2) in an efficient way:

- Split R in some general way, for example into sums with equal amounts of summands
- Look for the most complex already–present part of R in G, make this R_1 and compute R_2 via $R_2 = R - R_1$
- Iterate over all possibilities of splitting R, looking at which split results in the least amount of operations added

The possibilities mentioned differ in terms of effectiveness, but also in terms of runtime. Splitting R in a general way is the fastest, but also does not consider anything present in the graph. Looking for the most complex already–present part does not take long, but might result in short–sighted decisions. Iterating over all the possibilities results in exponential runtime due to combinational explosion of the recursion, but appears to provide the best result. The actual implementation of the heuristic can be configured to select between those methods or to try several and use the best result.

```
   Data: Matrix U
   Result: Directed graph G representing the computation of matrix U times a vector
1  V(G) ← {}
2  E(G) ← {}
3  N_r ← Number of rows in U
4  s ← KeyFunction (U)
5  U_s ← sort (U,s)
6  for i ← 1 to N_r do
7  |  u_i ← i–th row of U_s
8  |  AddRowRecursive(G,u_i)
9  end
```

Algorithm 1: Greedy heuristic for graph building

```
1  Function: AddRowRecursive(G,R)
   |  Data: Graph G to add row to
   |  Data: Row R to add
2  |  if R is trivial, i.e. one of the inputs then
3  |  |  Do nothing and return
4  |  end
5  |  if R is already present in G then
6  |  |  Do nothing and return
7  |  end
8  |  (R_1, R_2) ← DecomposeFunction (R)
9  |  AddRowRecursive (R_1)
10 |  AddRowRecursive (R_2)
11 |  Add node for R
12 |  Add edges R_1 → R and R_2 → R
13 end
```

Algorithm 2: *AddRowRecursive* implementation

4.1.3. Graph scheduling

After the graph has been constructed, the problem of assigning a computation cycle to each of the nodes has to be solved. More formally, every node is assigned an integer l that specifies how many cycles after the start of the computation it is computed. The inputs to the graph are defined as having $l = 0$. A parallelism constraint is interpreted as saying "there can be at most p nodes having the same l". A simple algorithm can be used to schedule a given graph relatively effectively. It follows the idea of "add those nodes that recursively have the most successors first". A more formal definition is given in Algorithm 3 on page 32. The two auxiliary functions *FindCandidates* and *SelectBestCandidate* find a set of nodes eligible for scheduling on the given latency level and then select the one which should be scheduled first, respectively. The *FindCandidates* function iterates over all not yet scheduled nodes of the graph and finds those that satisfy the following:

- All predecessors of the node are already scheduled

- All predecessors of the node are scheduled at an earlier latency than the current one

Among those, the *SelectBestCandidate* function chooses the one with the most successors. The successors are counted recursively up to the end of the graph. While no proof is attempted here that this approach is optimal, practical tests showed good results.

```
Data: Graph G to schedule, Integer p specifying maximum parallelism
Result: Graph G is scheduled
l ← 0                                                  // Current latency level
s ← p                                                  // Slots left on this level
n ← |V(G)|                                             // Nodes left to schedule
while n > 0 do
    while s > 0 do
        C = FindCandidates (G,l)
        if |C| = 0 then
            break                                      // No candidates left, go to next latency
        end
        R ← SelectBestCandidate (G,C)
        l_R ← l                                        // Set latency of R to l
        n ← n − 1                                      // A new node was placed, one less left
        s ← s − 1                                      // A slot was filled, one less left
    end
    l ← l + 1                                          // No slots left or no candidates, go to next layer
    s ← p                                              // This means we have p slots left again
end
```

Algorithm 3: Heuristic graph scheduling algorithm

5. Fixed–point computations and error bounds

5.1. Introduction

If the projection computation is performed in fixed point arithmetic, errors accumulate in a nontrivial manner. This leads to the question: Given a specified fixed point precision, what is the worst–case deviation of the fixed point result from result computed in exact arithmetic? This chapter investigates the sources of error and introduces a theoretical upper bound on the maximum deviation (also referred to as *error*). This bound holds for all input points and can be evaluated offline after the generation of the operation graph $G_U(x)$.

As shown in Section 2.3.2, all computations required to perform projections can be summarized as one matrix–vector product of the form

$$r = U \begin{bmatrix} x \\ 1 \end{bmatrix} \tag{5.1}$$

This makes the projection of a point a three–step process:

1. Compute r
2. Determine which region contains the point x based on r
3. Select the projection results for that region among the entries of r

The fact that the vector r contains both the hyperplane evaluation results as well as the actual projection computations can lead to two kinds of error:

- **Direct computation errors ($\epsilon_{\mathbf{dir}}$):** The projection result is built from entries of r. Since these entries are inaccurate, the final result will be inaccurate. This error type encompasses all errors in the computation of r and linear error analysis can be used to derive a bound.
- **Region shift errors ($\epsilon_{\mathbf{reg}}$):** The determination of which region contains the input point is based on entries of r. Since they are inaccurate, this determination can yield a wrong result, i.e. a point can be classified as being in a neighboring region. Linear error analysis cannot be used here.

While direct computation errors occur for every input point, region shift errors only appear if a point is near a region boundary. What follows is a short section on notation, then two sections that treat these two error types in more detail and specify upper bounds on them. The chapter then concludes with a section that assembles a total worst–case error bound from the individual bounds, answering the question posed in the beginning.

The following sections rely on many basics related to fixed point arithmetic. Appendix A gives a short introduction to fixed point arithmetic as well as a treatment of what errors occur in computations involving fixed point numbers.

5.2. Notation and assumptions

In the following, variables with a hat denote the fixed point version of the same variables without hat, i.e. $\hat{U}$ is the fixed point version of U. Conversion to fixed point is a scalar operation, it is applied to matrices and vectors element–wise. Computations that involve both fixed point and real variables (for example $\hat{U}x$) are defined as exact arithmetic computations in which all the fixed point variables are cast back to real numbers with the same value. For the sake of easier notation, define

$$z := \begin{bmatrix} x \\ 1 \end{bmatrix} \tag{5.2}$$

The following assumes that $z \equiv \hat{z}$, which means there is no error in the input point. Additionally, the input points x (and with that, z) are constrained to be within a symmetric box by the fixed amount of integer bits that are used to represent them. If for example the numbers have k integer bits, they can only be in $[-\left(2^k\right), 2^k - 1]$, which is approximated here as $[-\left(2^k\right), 2^k]$ for symmetry. The set of x with all components in this range is denoted $\mathbb{X}$. The actual computation of r is performed using an operation graph as outlined in earlier chapters, denoted as

$$\hat{r} = \hat{G}_{\hat{U}}(z) \tag{5.3}$$

which means "G is evaluated in fixed point arithmetic using coefficients from $\hat{U}$". The exact arithmetic reference computation is denoted as

$$r = Uz \tag{5.4}$$

While (5.4) is only the first step in the projection computation, it is the only step which is different for exact and fixed point computations, which is why it is referred to as the *reference* even though the actual reference computation is the combination of (5.4) with the other two steps outlined in the introduction.

5.3. Direct computation errors

The maximum direct computation error of r is defined as

$$\epsilon_{\text{dir}} := \max_{z \in \mathbb{X}} \left\| Uz - \hat{G}_{\hat{U}}(z) \right\|_\infty \tag{5.5}$$

where $\mathbb{X}$ is the allowed set of inputs to the projection computation. This represents the maximum deviation of the fixed point graph evaluation from the original exact arithmetic reference. The ∞–norm is chosen to get the largest component of $|r - \hat{r}|$, since this will be useful later. Next, a derivation of an upper bound on ϵ_{dir} is presented.

5.3.1. Upper bound derivation

Since the result of the exact computation is known, determining ϵ_{dir} in (5.5) means finding out what errors are introduced in the computation of $\hat{G}_{\hat{U}}(x)$. As demonstrated in Appendix A, determining the computational error of a fixed point product that has inaccuracies in both input operands is nontrivial. The only products that appear in $\hat{G}_{\hat{U}}(x)$ are products of entries of $\hat{U}$ and intermediate results. Therefore, if $\hat{U}$ were free from errors, the graph would only have products

of one correct and one inaccurate operand. Obviously, the conversion from U to $\hat{U}$ cannot be made error free unless $U = \hat{U}$. However, (5.5) can be bounded from above as follows:

$$\begin{aligned}\epsilon_{\text{dir}} &= \max_{z\in\mathbb{X}} \left\| Uz - \left(\hat{U}z - \hat{U}z\right) - \hat{G}_{\hat{U}}(z) \right\|_\infty \\ &\leq \underbrace{\max_{z\in\mathbb{X}} \left\| Uz - \hat{U}z \right\|_\infty}_{\epsilon_{\text{discr}}} + \underbrace{\max_{z\in\mathbb{X}} \left\| \hat{U}z - \hat{G}_{\hat{U}}(z) \right\|_\infty}_{\epsilon_{\text{comp}}}\end{aligned} \tag{5.6}$$

Here, $\hat{U}z$ is defined as an exact computation with $\hat{U}$ cast to real numbers. This upper bound results in a new term ϵ_{discr} and a term ϵ_{comp} that looks like (5.5), but with a $\hat{U}$ instead of U. This means that in ϵ_{comp}, both of the terms in the subtraction use the same $\hat{U}$, which means when comparing $\hat{U}z$ to $\hat{G}_{\hat{U}}$, only errors accumulating in the variables, but no coefficient errors have to be considered. This makes the computation of ϵ_{comp} independent of z. Next, an algorithm for this computation is presented. The computation of ϵ_{discr} is shown in the section after that.

Computation of ϵ_{comp}

The basis for the determination of the worst–case direct computation error is the operation graph created by solving (2.9). The order of operations and number of multiplications in $G_{\hat{U}}$ determines what errors are introduced. Instead of deriving a conservative analytical bound, a simple algorithm can be used to recursively determine an upper bound on the worst–case error at each node of the tree, starting from the results. The algorithm is shown in Algorithm 4 and Algorithm 5 on page 36. Once this algorithm is performed, the worst-case error for each result node (in number of least significant bits (LSBs), trivially convertible to a real number) is known and can be written down as a vector:

$$\eta(\hat{U}, G) = \begin{bmatrix} \max\limits_{z\in\mathbb{X}} \left| (\hat{U}z)_1 - \left(G_{\hat{U}}(z)\right)_1 \right| \\ \max\limits_{z\in\mathbb{X}} \left| (\hat{U}z)_2 - \left(G_{\hat{U}}(z)\right)_2 \right| \\ \vdots \\ \max\limits_{z\in\mathbb{X}} \left| (\hat{U}z)_r - \left(G_{\hat{U}}(z)\right)_r \right| \end{bmatrix} := \begin{bmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_r \end{bmatrix} \tag{5.7}$$

where r is the amount of rows of U and $(Ux)_1$ is the first component of Ux and so on. The computation error is then obtained as follows:

$$\epsilon_{\text{comp}} = \|\eta\|_\infty \tag{5.8}$$

Computation of ϵ_{discr}

This error is purely due to the quantization of U. It can be bounded from above as follows:

$$\epsilon_{\text{discr}}(U, \hat{U}) = \max_{z\in\mathbb{X}} \left\| (U - \hat{U})z \right\|_\infty := \max_{z\in\mathbb{X}} \|(\Delta U)z\|_\infty \tag{5.9}$$

Note that the components of z are bounded in absolute value. Each entry of $\hat{U}$ deviates at most 1 LSB from the corresponding entry in U. This means (5.9) can be rewritten as

$$\epsilon_{\text{discr}}(U, \hat{U}) = 2^{b_i - 1} \max_{u_i \in \Delta U} |u_i|_1 \tag{5.10}$$

with b_i, b_f the numbers of integer and fractional bits used in the fixed–point representation of x and u_i is the i–th row of ΔU. The maximum error can be found by selecting the row of ΔU with the largest 1–norm and then selecting the entries of x with maximum absolute value 2^{b_i-1} and the same sign as the entries in the selected row with which they are multiplied.

Data: Directed graph G representing the computation of matrix $\hat{U}$ times a vector
Data: Worst–case error $\epsilon_{\max}$

$\epsilon_{\max} \leftarrow 0$;
for $i \leftarrow 1$ **to** r **do**
  $u_i \leftarrow i$–th row of $\hat{U}$
  $\epsilon \leftarrow$ `GetMaxError(`G,u_i`)`
  $\epsilon_{\max} \leftarrow \max(\epsilon, \epsilon_{\max})$
end

Algorithm 4: Maximum error determination algorithm

Function: `GetMaxError(`G,r`)`
  Data: Graph G to trace error in
  Data: Node r to start at
  if *r is trivial, i.e. one of the inputs* **then**
    return 0
  end
  if *r is a product* **then**
    $\alpha \leftarrow$ Factor this is a product with
    $r_i \leftarrow$ Node name of variable input
    return $1 + \alpha($ `GetMaxError(`G, r_i`)`$)$
  end
  if *r is a sum* **then**
    $r_1 \leftarrow$ Node name of variable input 1
    $r_2 \leftarrow$ Node name of variable input 2
    return `GetMaxError(`G, r_1`)` + `GetMaxError(`G, r_2`)`
  end
end

Algorithm 5: *GetMaxError* implementation

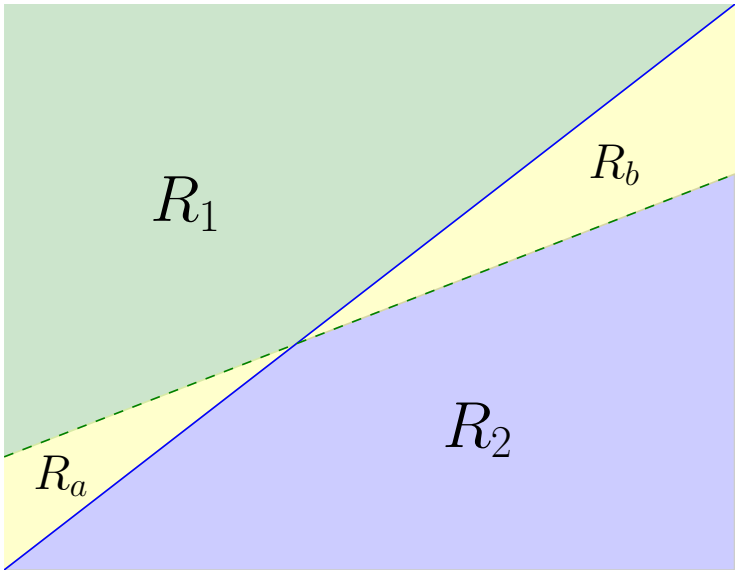


Figure 5.1.: **Region shift:** The solid blue line represents a hyperplane evaluation in exact arithmetic, and the dashed green line shows a possible inaccuracy if this evaluation is performed in fixed point arithmetic. As one can see, points in the region R_b were originally considered to be in R_2, but are classified as being in R_1 if the hyperplane evaluation is performed in fixed–point arithmetic. The opposite effect happens for points in R_a.

5.4. Errors due to region shifts

5.4.1. Source

Because the hyperplane evaluations for point location are also performed in fixed point arithmetic, points that were originally part of one region can become part of another after fixed point conversion. Consequently, the wrong projection function is evaluated for such points. The reason for this misclassification of points is that slopes and offsets of halfplanes change if their coefficients are quantized, which results in region boundaries moving. An example of this is shown in Fig. 5.1. Define R_1, R_2 as the regions in which the point is if hyperplane evaluations are performed in exact and fixed point arithmetic, respectively (this would correspond to a point in R_a in Fig. 5.1). If the projection rules associated with R_1, R_2 are F_1x+g_1 and F_2x+g_2, respectively, then the error due to a point from R_1 being classified as in R_2 is

$$\epsilon = (F_1x + g_1) - \left(\hat{F}_2x + \hat{g}_2\right) = \left(F_1 - \hat{F}_2\right)x + (g_1 - \hat{g}_2) \coloneqq Fx + g. \tag{5.11}$$

Instead of $F_1x + g_1$ in exact arithmetic, $\hat{F}_2x + \hat{g}_2$ is evaluated in fixed point arithmetic. The circumstances under which this happens can be written down in a mathematical form. Define the region description of R_1 and R_2 as $A_1x \leq b_1$ and $A_2x \leq b_2$ and the fixed point version of the second region description as $\hat{A}_2x \leq \hat{b}_2$. A point fulfills the above requirements if at the same time, $A_1x \leq b_1$ and $\hat{A}_2\hat{x} \leq \hat{b}_2$ hold. If there is no such point, then no points move across this particular region boundary. Note that the computation of $\hat{A}_2x \leq \hat{b}_2$ also has associated direct computation errors. An example of a one–dimensional region shift and the associated error is given in Fig. 5.2.

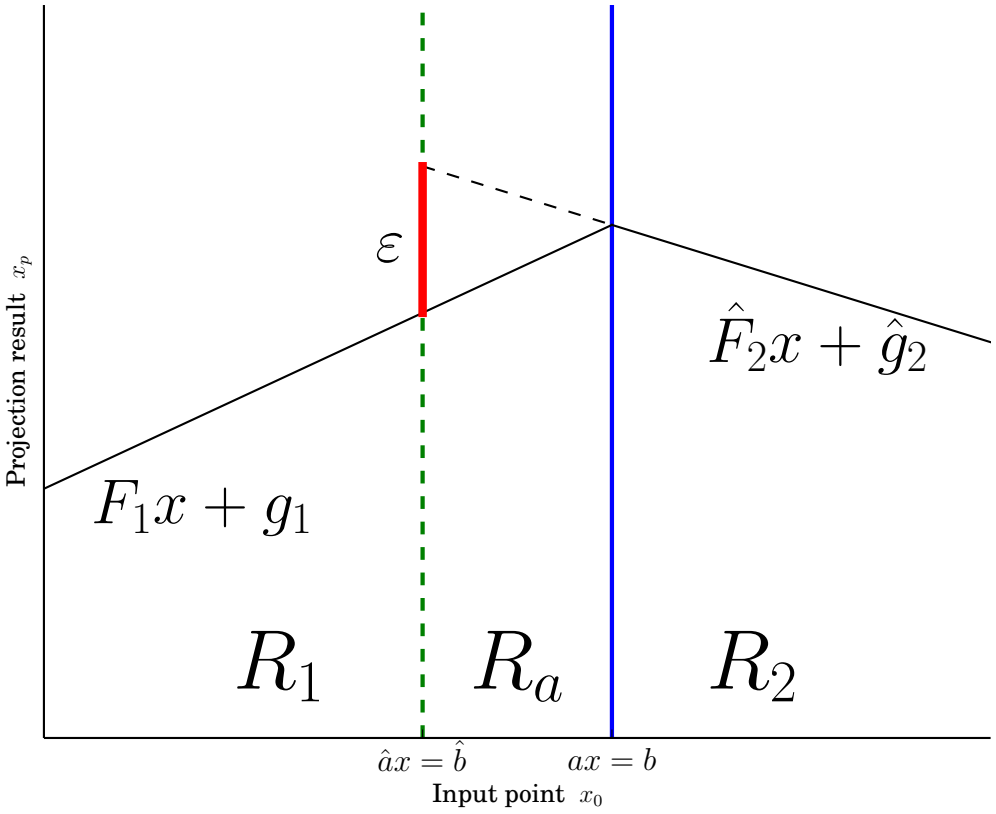


Figure 5.2.: **Region shift error in 1D:** The exact arithmetic boundary is denoted $ax = b$, whereas the fixed–point version is denoted $\hat{a}x = \hat{b}$. R_a is the region of misjudgement: points within it are part of R_1, but get classified as being in R_2 by the fixed–point arithmetic hyperplane evaluation. Two example projection functions $F_1x + g_1$ and $\hat{F}_2x + \hat{g}_2$ are shown in the figure, as well as the maximum error ε that occurs due to the misjudgement.

5.4.2. Upper bounds

The region shift errors are a combination of errors within hyperplane evaluations and direct errors as outlined in the previous section. The error of interest is the one between computing everything in exact arithmetic (reference conditions) and evaluating everything in fixed point arithmetic (actual in–circuit conditions). A relatively easy to evaluate upper bound for this error is the maximum offset resulting from using the wrong projection formula added to the maximum error introduced when computing the actual projection formula in fixed point.
Consider two regions R_1 and R_2 as defined in Section 5.4.1. In the case outlined in that Section, instead of F_1x+g_1 in exact arithmetic, $\hat{F}_2x+\hat{g}_2$ is computed in fixed point. While the worst–case difference between F_2x+g_2 in fixed point and exact arithmetic is computed using the algorithm in the previous section, the error due to using the wrong projection rule has yet to be found. The worst case can be found using an optimization problem:

$$\begin{aligned} \max_{x\in\mathbb{X}} \quad & \|Fx+g\|_2^2 \\ \text{s.t.} \quad & A_1x \leq b_1 \\ & \hat{A}_2x \leq \hat{b}_2 + \mathbf{1}\epsilon_{\text{dir}}(U,\hat{U}) \end{aligned} \tag{5.12}$$

where F, g, A_1, A_2, b_1, b_2 are as defined in Section 5.4.1 and U as defined in the introduction. The second constraint incorporates the direct computation error in computing $\hat{A}_2\hat{x}-\hat{b}_2$ by adding $\epsilon_{\text{dir}}(U,\hat{U})$ to all entries of $\hat{b}_2$, covering all possible computational errors. The true worst–case error would be the highest error value found across all region combinations. Let $i,j \in 1,\ldots,N_r$ and $i \neq j$ be indeces of two regions. The worst–case region shift error ϵ_{reg} is defined as

$$\epsilon_{\text{reg}} \coloneqq \max_{\substack{i\in1,\ldots,N_r \\ j\in1,\ldots,N_r}} \begin{pmatrix} \max\limits_{x\in\mathbb{X}} & \|(F_i-\hat{F}_j)x+(g_i-\hat{g}_j)\|_2^2 \\ \text{s.t.} & A_ix \leq b_i \\ & \hat{A}_jx \leq \hat{b}_j + \mathbf{1}\epsilon_{\text{dir}}(U,\hat{U}) \end{pmatrix} \tag{5.13}$$

Since the number of region combinations grows quadratically in the number of regions (and the number of regions grows exponentially in the number of problem dimensions), many instances of (5.12) have to be solved. Unfortunately, the maximization of a norm is not convex, and therefore direct solution is impractical. However, an upper bound can be found by decomposing $Fx+g$ into its rows as follows:

$$Fx+g := \begin{bmatrix} f_1x+g_1 \\ f_2x+g_2 \\ \vdots \\ f_nx+g_n \end{bmatrix} \tag{5.14}$$

where f_1 is the first row of F and so on. One can then write

$$\max_{x\in\mathbb{X}} \|Fx+g\|_2^2 \leq \left\| \begin{bmatrix} \max\limits_{x\in\mathbb{X}} |f_1x+g_1| \\ \max\limits_{x\in\mathbb{X}} |f_2x+g_2| \\ \vdots \\ \max\limits_{x\in\mathbb{X}} |f_nx+g_n| \end{bmatrix} \right\|_2^2 \tag{5.15}$$

where $\mathbb{X}$ is the feasible set of x. The above holds since

$$\|Fx+g\|_2^2 = \sum_{i=1}^{n} (f_ix+g_i)^2 \tag{5.16}$$

and the sum of maxima is larger than a maximum over a sum:

$$\max_{x\in\mathbb{X}} \sum_{i=1}^{n} (f_i x + g_i)^2 \leq \sum_{i=1}^{n} \left(\max_{x\in\mathbb{X}} |f_i x + g_i| \right)^2 \tag{5.17}$$

This holds since each individual term on the right side is by definition larger than its equivalent on the left:

$$(f_i x + g_i)^2 = |f_i x + g_i|^2 \leq \left(\max_{x\in\mathbb{X}} |f_i x + g_i| \right)^2 \quad \forall x \in \mathbb{X} \tag{5.18}$$

The result of (5.12) can now be bounded from above using the sum–of–squares of the results of the following set of LPs:

$$\begin{aligned} \max_{x} \quad & |f_i x + g_i| \\ \text{s.t.} \quad & A_1 x \leq b_1 \qquad\qquad (i \in 1, \ldots, n) \\ & \hat{A}_2 x \leq \hat{b}_2 + \mathbf{1}\epsilon_{\text{dir}}(U) \end{aligned} \tag{5.19}$$

where the absolute value function can be implemented by solving (5.19) once for both the objectives $f_i x + g_i$ and $-(f_i x + g_i)$ and taking the maximum of the two as the final result.

This set of LPs is now almost trivial to solve, which means it is also tractable to solve it for all possible region combinations in the problem sizes this report is concerned with. For many combinations of regions, there will be no feasible point. This happens if for example two regions are not adjascent and the fixed-point rounding errors are not enough to make them overlap. In this case, the worst–case error due to a shift between those two regions is simply 0.

5.5. Total error

Either point location has the same result in fixed point and exact arithmetic, in which case the total error is only the error due to the fixed point evaluation of $F_i x + g_i$ for that region. In the second case, the same error occurs, but additionally, the error arising due to the wrong i being chosen has to be added.

The presented error bound can be evaluated offline for a given matrix U. An example evaluation is presented in Chapter 7.

Part II.

Practical implementation

6. Toolchain and target hardware overview

The theoretical optimizations presented in Part I were implemented as an automated toolchain. This chapter presents a conceptual high–level overview as well as an overview of the selected target platforms. More practical, academically less relevant information on library dependencies, platform specifics and miscellanea are outlined in the documentation for the toolchain software project, which is delivered with the code itself.

6.1. Framework and features

The toolchain was implemented in Python (version 3) [1], using NumPy [27] and ECOS [13] for linear algebra and linear programming and NetworkX [18] for graph manipulation. A full list of software dependencies is given in the documentation of the code. While originally designed to handle projections onto affine sets, it now supports any piecewise affine function. A few important features are listed below:

- C and VHDL code generation backends
- Fixed and floating point support for C, fixed point only for VHDL
- Automatic compilation, simulation and verification for both backends
- Semi-automatic profiling for both backends
- Evaluation of worst–case errors due to fixed–point arithmetic

While the toolchain as it is only runs on Linux, the dependencies on Linux are only due to the interface to other programs like Gurobi, Vivado, the ARM toolchain and MATLAB. The Python code itself was kept platform–independent.

6.2. Toolchain components and data flow

Figure 6.1 on page 45 gives an overview of the components present in the toolchain. The input to the toolchain is a description of a set to project on ($Ax \leq b$), however the code immediately uses MPT to convert this into a PWA function, which means general PWA functions can easily be used as input as well. The following sections give a short summary for the main components of the toolchain.

6.2.1. Frontend

The frontend launches MATLAB and uses MPT [20] and YALMIP [23] to generate a PWA function for projection onto the given set and various other results, mentioned in detail in the code documentation. A matrix of hyperplane evaluations and projection rules to be computed is generated (U) along with a dictionary that describes which hyperplane evaluation results lead to what projection result per variable (D_{mux}). Since the frontend is the only place MATLAB is used, the frontend also runs the reference code generation toolboxes (MPT and MOBY–DIC).

6.2.2. Optimization

The optimization tool takes the matrix created by the frontend as input and attempts to find an efficient way to compute products of this matrix with the input vector. This optimization is performed by either solving a MILP using the Gurobi [17] or GLPK [4] solvers or with the heuristic presented in Chapter 4. If the VHDL code generation was selected, the operation graph is optimized to have the lowest possible latency and few operations. If the C backend was selected, focus is only put on having few operations in the graph, and the resulting graph is already serialized to one operation per computational cycle. The result of this optimization is given as a directed graph data structure.

6.2.3. Backend

The selected backend takes the computation description created by the optimization tool and generates code that performs the aforementioned operations. For the VHDL backend, a pipelined architecture exactly corresponding to the operation graph is implemented. All end results that are already available somewhere within the graph are buffered with registers to make them available in the same cycle as the last result computed. The selection of results is implemented as a single–cycle muxing network. No attempts to optimize this muxing network are made here, since circuit synthesis tools already implement sophisticated methods for lower–level optimizations. The C backend implements the serialized graph as a series of C expressions, storing intermediate results in static variables. The muxing network is then implemented as a binary tree traversal which works exactly the same as the MPT–generated code.

6.2.4. Verification

While analytical verification can be used to show that the operation graph created by the optimization indeed represents the matrix computation it was generated for, all other steps of the toolchain have to be verified using numerical verification. For this purpose, the frontend already generates a set of pairs (x_0, x_p) that are used to verify the generated code numerically.

6.3. Hardware targets

6.3.1. STM32F407 ARM embedded processor

This embedded processor has an ARM Cortex–M4 architecture with a floating point unit. This architecture has a 3–stage pipeline and branch speculation. More information on the Cortex–M4 architecture can be found in [6]. The tests conducted were performed using bare–metal code, meaning no operating system and no standard library were used. This means that timing results have no variance and are accurately reproducible. As test hardware, an STMF4 Discovery board was used, as shown in Fig. 6.2 on page 46.

6.3.2. Xilinx Zynq–7020 SoC

The Xilinx Zynq platform is a family of system–on–chip (SoC) devices which feature an FPGA as well as an ARM Cortex–A9 microprocessor. For the purposes of this thesis, only the FPGA portion was used. An overview of the resources available on the FPGA portion is given in Table 6.1. As can be seen, the size of this FPGA is already quite substantial, however Xilinx also

Resource	Available amount
Registers	106400
Look–Up Tables	53200
DSP slices	220

Table 6.1.: Resources available on the Zynq–7020 FPGA portion

has devices with five times as many LUTs and registers and 10 times as many DSPs available. As a development platform for this thesis, the Zedboard was used, shown in Fig. 6.3 on page 46.

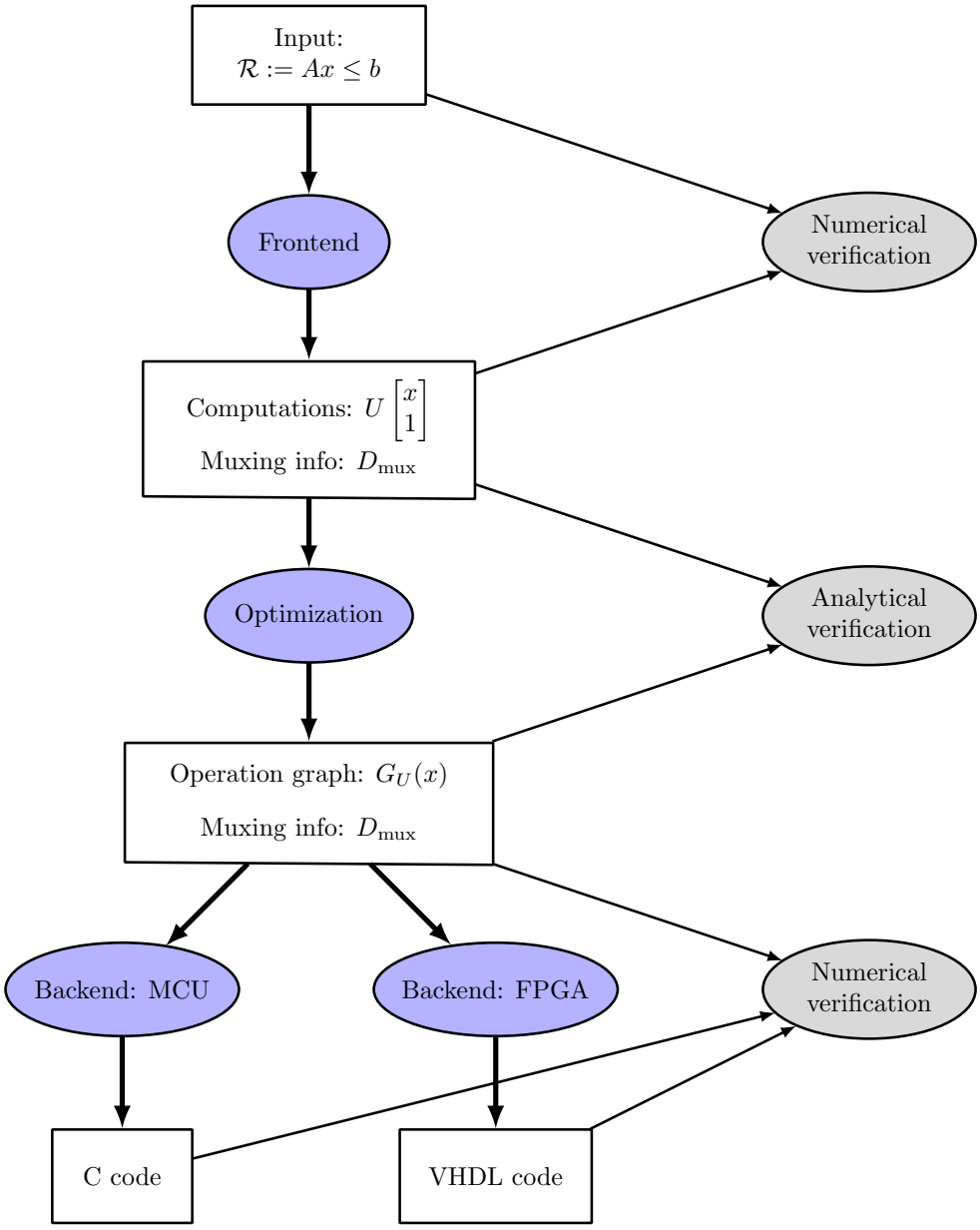


Figure 6.1.: **Toolchain overview:** This figure depicts the flow of data and the different formats used in between the tools. Tools are shown in ellipses, whereas intermediate data is in boxes. The resulting code can be directly passed on to compilers for the respective platforms.

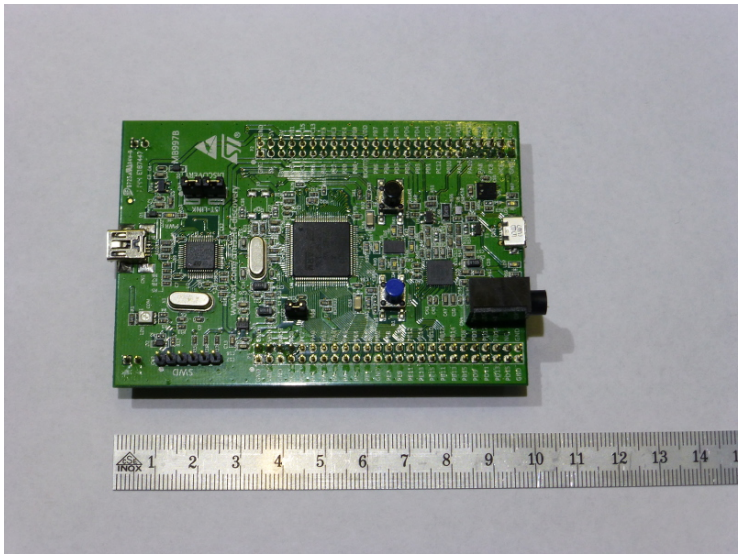


Figure 6.2.: STM32F407 Discovery board

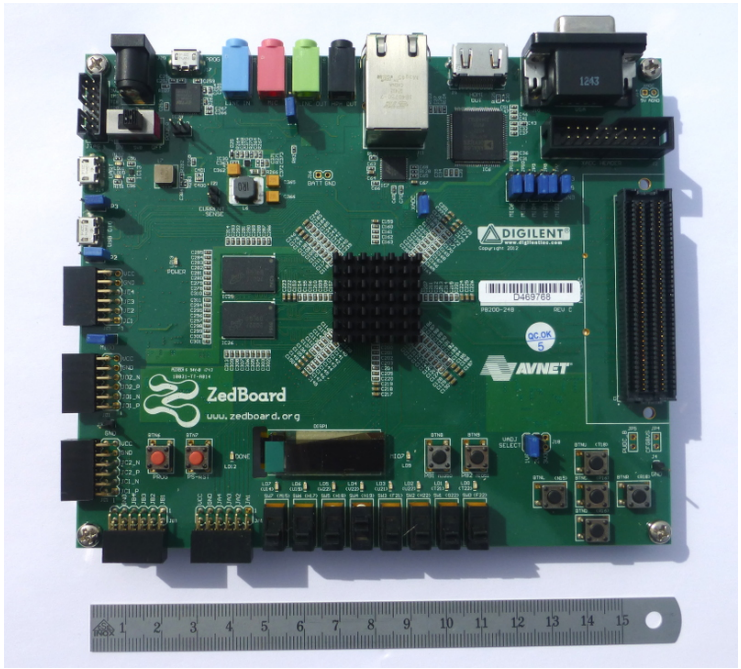


Figure 6.3.: Zedboard

Part III.

Results

7. Results

This chapter quantitatively compares the circuits and code generated by the toolchain with reference toolchains that serve similar purposes. First, a few practically interesting sets are presented. Next, comparisons for these sets are presented in separate sections for the C code output and the VHDL circuit output. The chapter concludes with sections giving practical examples for the heuristic and error bound computation.

7.1. Benchmark sets

Due to the nature of this toolchain, it is particularly useful for projections that lend themselves to operation sharing. A few suitable, practically relevant sets are presented here.

7.1.1. Ordered set

A set that often appears in practical applications like the selection of switching times in power electronics equipment is the so–called ordered set, defined in n dimensions as

$$0 \leq x_1 \leq x_2 \leq \cdots \leq x_n \leq x^*. \tag{7.1}$$

Note that the two–dimensional case of this was already used as an example in Section 2.2. A variation on this would be the non–unit slope ordered set:

$$0 \leq \alpha_1 x_1 \leq \alpha_2 x_2 \leq \cdots \leq \alpha_n x_n \leq x^* \tag{7.2}$$

with $\alpha_i > 0 \quad \forall i$. However, without any assumptions on the α_i, it is difficult to make relevant benchmarks: While a particular set of α_i might result in a high degree of operation sharing and a small circuit, another set might result in a big circuit.

7.1.2. Hexagon

The hexagon in two dimensions is a set that appears often in power electronics. It can be described as

$$\begin{bmatrix} 1 & \cot(60) \\ -1 & -\cot(60) \\ 1 & -\cot(60) \\ -1 & \cot(60) \\ 0 & 1 \\ 0 & -1 \end{bmatrix} x \leq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ \sqrt{3}/2 \\ \sqrt{3}/2 \end{bmatrix}, \qquad \cot(\alpha) = \frac{\sin(\alpha)}{\cos(\alpha)} \tag{7.3}$$

Rotated versions of this set are also practically relevant.

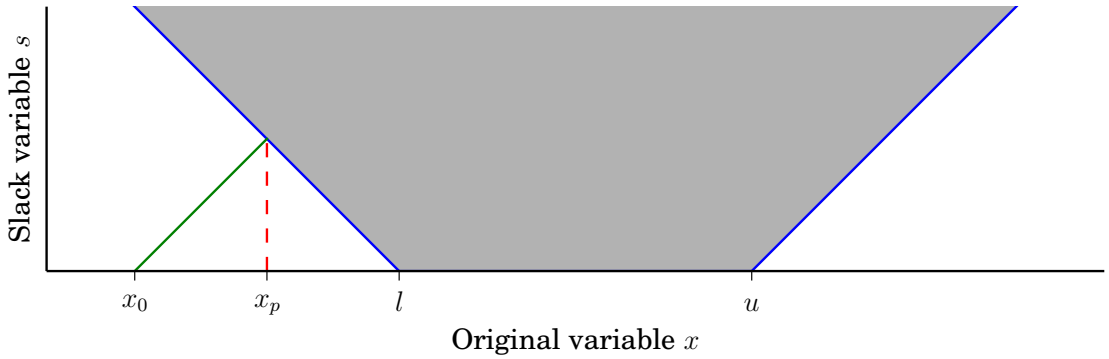


Figure 7.1.: Visualization of (7.6) for $x \in \mathbb{R}$. Instead of projecting every $x < l$ onto l and every $x > u$ onto u, constraint violations are decreased by a factor, leading to x_0 being projected onto $x_p < l$. Simple trigonometric considerations can be used to show that $|x_0 - l| \,/\, |x_p - l| = 1 \,/\, \cos^2(\arctan(\alpha))$. In this figure, α was set to 1, leading to a constraint violation reduction by a factor of 2.

7.1.3. Box

Box constraints are another case of a set that requires almost no computations if it is aligned to the coordinate axes. In dimension n, box constraints can be written as

$$l \leq x \leq u \tag{7.4}$$

for $l, u, x \in \mathbb{R}^n$ and l, u constant. Rotated versions of the box constraints can also be considered:

$$l \leq Rx \leq u \tag{7.5}$$

with $R \in \mathbb{R}^{n \times n}$ being constant a rotation matrix. However, the same considerations as the ones for the non–unit slope ordered set apply: Without further specifications on the angles, no real benchmark statements can be made.

7.1.4. Soft constraint

In many applications, constraints do not have to be strictly met. This can be incorporated into the constraints by adding slack variables to them. For example, the box constraint could be expressed in a soft version as follows:

$$l - \alpha s \leq x \leq u + \alpha s \tag{7.6}$$

with $s \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}_+ \setminus 0$ a coefficient. Soft problems are still affine, but the number of variables in these problems increases by the number of slack variables added. The effect of this addition is demonstrated in Fig. 7.1 for $n = 1$.

7.1.5. Output constraint

Another fairly typical set arises when an output of a system is a linear combination of states. If for example an LTI system has two states x_1, x_2 and its output is defined by $y = c_1 x_1 + c_2 x_2$ with an allowed range of $[l, u]$, the set to project on is

$$l \leq c_1 x_1 + c_2 x_2 \leq u. \tag{7.7}$$

7.2. Embedded C code backend

7.2.1. Reference implementation

The reference used to compare the C code to was the MPT [20] function to generate C code from a binary tree. It serves as a particularly good reference since traversing binary trees is the current state–of–the–art in embedded PWA evaluation. MPT only generates C code that works with double precision floating point numbers. It is trivial to patch the generated code to use single precision floating point, but the generation code would have to be entirely rewritten to generate code that works in fixed–point arithmetic. The C code implementations generated by the novel approach simply compute all the operations in the graph in order and then traverse the same binary tree as the one used by the MPT generated code based on the computation results. Therefore, the toolchain code is expected to be faster only where the speedup due to operation sharing and lack of branch statements during the computations outweighs the drawback of computing all hyperplane evaluations and functions.

Since not all nodes of the binary decision tree are at the same depth, there are regions that take more decisions to find than others. A region that has the highest depth in a given tree will be called a worst–case region here. Points within that region are all worst–case points.

7.2.2. Comparison results

Code was generated for the sets outlined in Section 7.1 and timed for a worst–case point on the STM32F4 Discovery board. The projection was repeated 100000 times and its runtime measured using a logic output of the board and a logic analyzer. The resulting time was then divided by 100000 to get the runtime of one projection. This averaging was done to increase measurement precision and remove any potential bias by surrounding code. A summary is given in Table 7.1. All measurements were taken with *-O3* turned on in the compiler, which means it should optimize the code for both size and speed. This setting is most commonly used in practice. A plot that compares implementations of the ordered set for various dimensions is shown in Fig. 7.2 on page 53.

As can be seen in the table and the plot, the MPT reference code is mostly faster, in addition to being less sensitive to rotations of the sets. The most prominent case of the dependence on rotations is the rotated 3D box in the table: For a rotation of 45 degrees, many hyperplane evaluations only differ in some coefficient signs, whereas for a random rotation (i.e. 26°) the hyperplanes have different slopes and hence the degree of operation sharing is far lower. Conversely, for cases that hardly require any operations at all, the code generated by the toolchain is consistently faster than the MPT reference. Since in the C backend, more operations translate into longer runtimes, these measurements are therefore consistent with the expectations outlined in the previous section.

7.3. VHDL code backend

7.3.1. Reference implementation

For circuit generation, the recently released MOBY–DIC toolbox [3] was used. This toolbox implements the same binary tree traversal as the MPT reference. An affine function computa-

Set	MPT time	Toolchain time
Hexagon (2D)	1.6 μs	2.2 μs
Hexagon (2D) rotated 26°	1.6 μs	3.4 μs
Box (2D)	1.0 μs	0.8 μs
Box (3D)	1.9 μs	1.0 μs
Box (3D) rotated 26°	1.9 μs	8.3 μs
Box (3D) rotated 45°	1.9 μs	2.5 μs
Box (4D)	3.0 μs	1.3 μs
Output constraints (2D)	1.1 μs	1.0 μs
Soft output constraints (3D)	6.3 μs	2.2 μs

Table 7.1.: Time per worst–case point projection for benchmark sets (C backend)

tion unit is used iteratively to perform the hyperplane evaluations and compute the projection function. This iterative architecture makes the circuits impossible to pipeline. MOBY–DIC was designed for general PWA functions and small circuit size, which explains why it is reasonable that the software implemented in this thesis can outperform it for projections, which have more structure. Additionally, the implementations in this thesis were designed primarily for speed. A comparison with [25], another state–of–the–art toolchain, was not done because no published code for their toolchain was found. However, from the architecture designs in the paper, it is easily derivable that their circuitry would be orders of magnitude larger than both MOBY-DICs and the ones generated in this thesis, since no operation sharing is attempted at all. The generated VHDL code for both reference and toolchain was implemented on the Zynq FPGA described in Section 6.3.2 using the Xilinx Vivado software under the same clock constraint conditions. The cycle times were extracted from the designs, and proper functioning of the generated circuits for both MOBY–DIC and the software from this thesis was confirmed using behavioral simulation and numerical verification of the results. The code generated in this thesis was also implemented on the actual Zynq board and verified in hardware–in–the–loop experiments.

7.3.2. Comparison results

Comparison results for the benchmark sets are given in Table 7.2 on page 52. A plot for the ordered set in various dimensions is given in Fig. 7.3 on page 54. As can be seen in the table and the figures, the toolchain is faster by a large margin in every case. This is no surprise, since the toolchain creates circuity that computes the projection in the minimum possible amount of cycles by design. One can also see that while hardware requirements generally increase, sets which are aligned to the coordinate axes are an exception. Additionally, the circuitry generated in this thesis is pipelined, which means if several independent projections are performed, the number of projections per clock cycle approaches 1. This pipelining cannot be added to any recursive implementations based on binary tree traversals, including MOBY-DIC.

7.4. Heuristic

The MILP representation presented in Chapter 3 is limited to using multiplications by constants that are already present in the original matrix U and additions. While it could be extended to allow for subtractions as well, this would roughly double the amount of binary variables and

Set		Cycles	DSP	LUT (%)	Register (%)
Hexagon (2D)	TC	3	6	0.95	0.55
	MD	14	2	0.28	0.15
Hexagon (2D) rotated 26°	TC	3	18	2.06	1.17
	MD	14	2	0.32	0.15
Box (2D)	TC	1	0	0.05	0.10
	MD	12	2	0.30	0.15
Box (3D)	TC	1	0	0.08	0.14
	MD	18	3	0.28	0.20
Box (3D) rotated 26°	TC	3	36	4.30	2.97
	MD	18	3	0.43	0.20
Box (3D) rotated 45°	TC	3	7	2.59	1.48
	MD	18	3	0.37	0.21
Box (4D)	TC	1	0	0.10	0.19
	MD	24	4	0.76	0.27
Output constraint (2D)	TC	3	5	0.33	0.34
	MD	8	2	0.24	0.15
Soft output constraint (3D)	TC	3	9	1.65	1.18
	MD	16	3	0.45	0.23

Table 7.2.: VHDL timing and resource comparison for benchmark sets. "TC" denotes rows with toolchain implementation results, "MD" denotes rows with MOBY-DIC implementation results. The output constraint was tested with example constants $c_1 = 22, c_2 = 4.9, l = -400, u = 1000$.

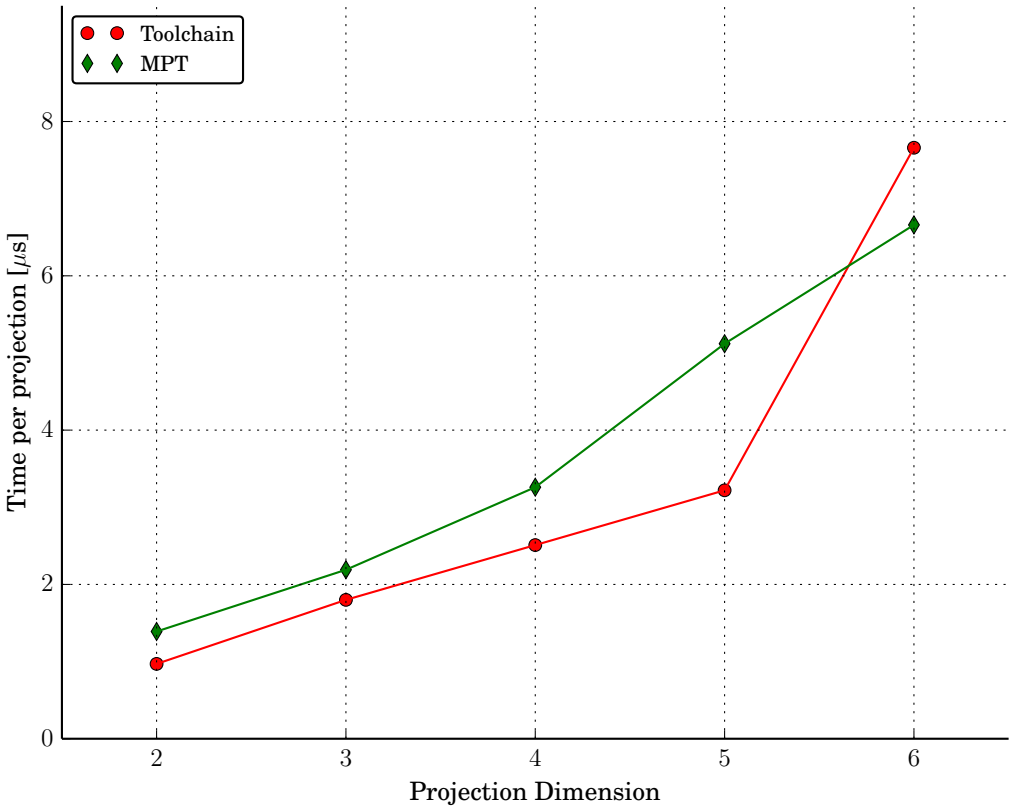


Figure 7.2.: **C generation, ordered set**
The code was compiled with *-O3* turned on in the compiler. Experiments with *-O0* to *-O1* resulted in a different plot with MPT being faster for all but the 2D and 3D cases. The large time increase for 6D in the toolchain timings are most likely a secondary effect like memory blocking issues or unlucky branch prediction.

add many more constraints. The inclusion of arbitrary factors for multiplication would make the problem into a quadratically constrained integer program, which would likely lead to even longer solution times. On the other hand, the heuristic outlined in Chapter 4 can be extended to using subtractions as well as multiplications by arbitrary factors simply by extending the row decomposition function to also consider these cases. The inclusion of these nonlinear aspects can be selectively turned on in the implemented software toolchain. As shown in Fig. 7.4 on page 55, the heuristic is indeed very effective.

7.5. Error bound

The error bound was tested with 100000 evenly spaced points on the 2D and 3D ordered sets for various fractional bit lengths. The results are plotted in Fig. 7.5 on page 56. As expected, the bound goes down monotonically if the number of fractional bits is increased.

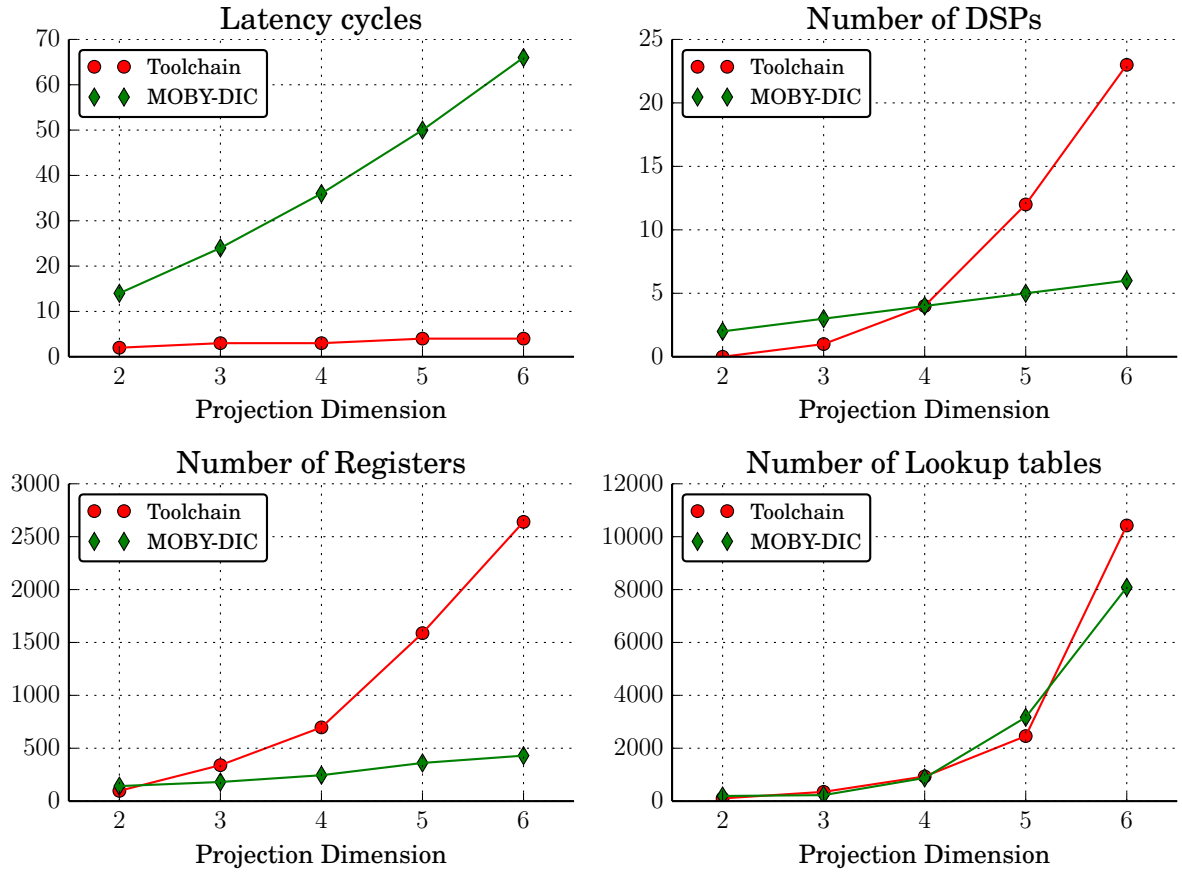


Figure 7.3.: Comparison of toolchain with MOBY-DIC for ordered set

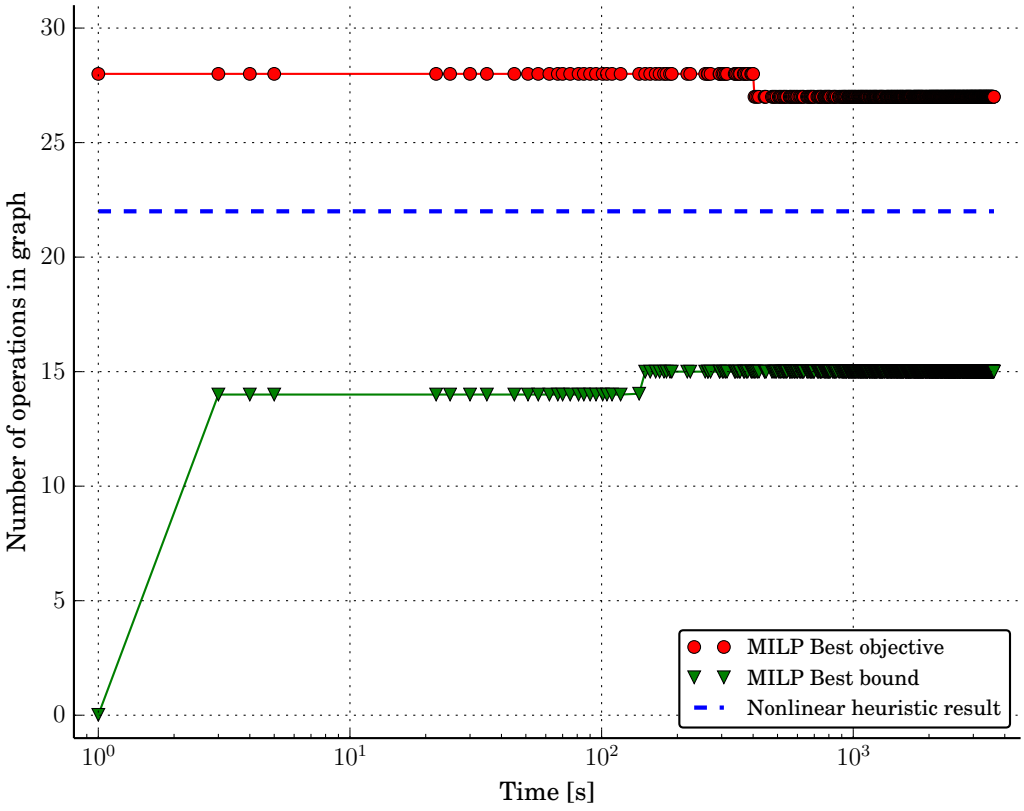


Figure 7.4.: Comparison of MILP solution process with heuristic. The heuristic takes less than a second to run, while the MILP solution process stops making progress after a few minutes. The shown plot is for the 3D ordered set, a relatively small problem. The other benchmark sets yielded qualitatively similar results.

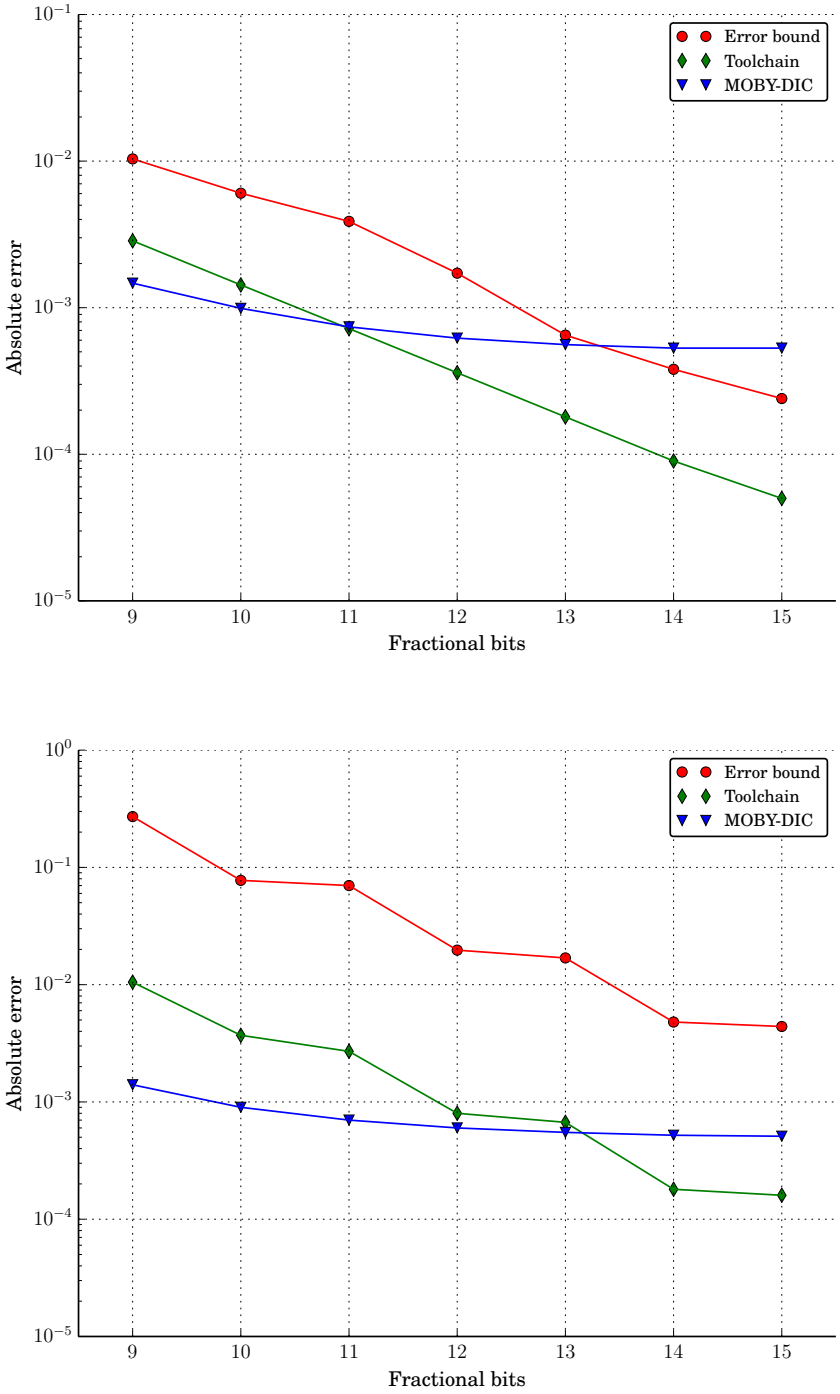


Figure 7.5.: **Error bounds for fixed–point implementations**
The upper plot shows comparisons of error bound, reference and toolchain for the 2D ordered set, the lower one the same comparison for the 3D ordered set. The errors shown for the reference and toolchain are worst–case errors found among all projected points. While this example does not make any general statement about the error bound, it shows that the bound behaves as expected.

8. Discussion and Outlook

8.1. Discussion

The benchmarks presented in Chapter 7 demonstrated that for some sets, dramatic speedups over current reference implementations can be achieved at comparatively small circuit size increases. The demonstrated speedup can be achieved for any set, since the toolchain designs a circuit that computes all results with the lowest latency possible. What varies is the size of the resulting circuit: If many intermediate results can be reused, the generated circuit can be the same size or even smaller than the reference decision tree based solutions. If however little sharing can take place, circuits quickly become impractically large. Nevertheless, compared to the existing approaches that reach similar computation speeds (i.e. [25]), circuit size is still very manageable. The presented benchmark comparisons were done with the MILP optimization disabled and only the heuristic developed in this thesis applied, since the MILP optimization would not have been tractable for the higher dimensions. This demonstrates that the implemented heuristic already yields useful results even though it is rather basic.

The pipelined nature of the generated circuits makes them a good choice for cases where many projections on independent points are required: With a growing number of independent projections, the number of cycles spent per projection approaches one. A practically important application of this feature is multistage MPC, as outlined in Chapter 1. Pipelining is not usable in any of the existing implementations.

The tests also showed that generating C code that applies operation reuse is of little benefit in embedded architectures. This was to be expected, since current compilers already apply good heuristics for intermediate result reuse. Additionally, current embedded processors have short pipelines, which means the increased amount of branching statements in the reference implementations are no big drawback compared to the mostly branchless computations performed by the code generated in this thesis.

Overall, this thesis demonstrated that for some practically relevant sets, problem–specific automatic circuit generation can in fact be very beneficial and significantly outperform current, more general reference implementations, while making better use of scarce resources. The implemented software toolchain is modular and extensively documented, making it a good starting point for further investigations related to operation reuse in projection computations.

8.2. Outlook and possible future work

A large part of the theory presented in this report is concerned with writing the problem of optimal operation sharing as a mixed integer linear program. While the underlying problem is *NP*–complete and hence intractable for everything but very small problems, there exist good heuristics for solving such problems approximately. More research into applying such methods

to the problem at hand could lead to further circuit size reductions in the generated implementations. The presented representation of operation graphs can also be used as a starting point for other related problems.

A prominent feature of the implemented toolchain is the dependence of circuit size on the alignment of the projection target sets to the coordinate axes. This suggests that it might be beneficial to investigate the application of a rotation to points before they are projected, leading to more aligned sets and leaner circuits.

To make the concept of operation sharing also applicable to C code generation, the combination of decision tree traversal with operation reuse could be investigated. This approach could potentially reduce the time per node in the tree traversal while keeping the original tree traversal mechanism in place.

Since the toolchain is not limited to projection computations, but rather designed to exploit structure in them, further research could be done to find other classes of functions for which it works well. Another approach would be to try to extend the frontend to exploit structure found in other function classes.

Bibliography

[1] Python programming language. `http://docs.python.org/3.3/`.

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[3] MOBY-DIC EU FP7 project, 2013. `http://www.mobydic-project.eu/the-project`.

[4] GLPK (GNU linear programming kit), 2014. `http://www.gnu.org/software/glpk/glpk.html`.

[5] F. Al-Hasani, M.P. Hayes, and A. Bainbridge-Smith. A common subexpression elimination tree algorithm. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 60(9):2389–2400, 2013.

[6] ARM. Cortex m4 processor specification. `http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php`.

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

[8] N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matrices. *Computers, IEEE Transactions on*, 54(10):1271–1282, 2005.

[9] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004.

[10] D.R. Bull and D.H. Horrocks. Primitive operator digital filters. *Circuits, Devices and Systems, IEE Proceedings G*, 138(3):401–412, 1991.

[11] P.R. Cappello and K. Steiglitz. Some complexity issues in digital signal processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 32(5):1037–1041, 1984.

[12] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[13] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Conference (ECC)*, pages 3071–3076, 2013.

[14] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)*, pages 668 – 674, Maui, HI, USA, December 2012.

[15] Alexander Domahidi. FORCES: Fast optimization for real-time control on embedded systems. `http://forces.ethz.ch`, October 2012.

[16] A. N. Fuchs, C.N. Jones, and M. Morari. Optimized decision trees for point location in polytopic data sets - application to explicit mpc. In *American Control Conference (ACC), 2010*, pages 5507–5512, 2010.

[17] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2014. http://www.gurobi.com.

[18] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[19] R. Hartley. Optimization of canonic signed digit multipliers for filter design. In *Circuits and Systems, 1991., IEEE International Sympoisum on*, pages 1992–1995 vol.4, 1991.

[20] M. Herceg, M. Kvasnica, C.N. Jones, and M. Morari. Multi-Parametric Toolbox 3.0. In *Proc. of the European Control Conference*, pages 502–510, Zürich, Switzerland, July 17–19 2013. http://control.ee.ethz.ch/~mpt.

[21] Yuen-Hong Alvin Ho, Chi-Un Lei, Hing-Kit Kwan, and Ngai Wong. Optimal common subexpression elimination algorithm of multiple constant multiplications with a logic depth constraint. *IEICE Transactions*, 91-A(12):3568–3575, 2008.

[22] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. Embedded predictive control on an fpga using the fast gradient method. In *Control Conference (ECC), 2013 European*, pages 3614–3620, 2013.

[23] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[24] Rainer Löwen. Branching numbers for euclidean projections onto convex polyhedra. *Geometriae Dedicata*, 72(1):99–103, 1998.

[25] M Mönnigmann and M Kastsian. Fast explicit mpc with multiway trees. In *Proc. of the 18th IFAC World Congress*, 2011.

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

[27] Travis Oliphant. Python for Scientific Computing. *Computing in Science & Engineering 9, 90*, 2007.

[28] Randy Yates. Fixed–point Arithmetic: An Introduction, 2013. http://www.digitalsignallabs.com/fp.pdf.

Part IV.

Appendix

A. Fixed–point computations and error sources

This appendix gives some background on what errors can occur in elementary fixed–point computations and how they can be bounded. The knowledge presented in this report is common knowledge amoung circuit designers and in no way introduced by the author.

A.1. Computer representation of numbers

This section gives a very short introduction to how numbers are represented in software and circuits. It also outlines the concepts of floating and fixed point arithmetic. Nothing presented in this section is new or introduced by the author, but it is essential knowledge for the error bound analysis presented in this thesis. The purpose of this section is to outline the most important facts. There are various works in literature that give a more complete treatment of the subject, for example [28].

In software and circuits alike, numbers are represented using base 2. An integer number is therefore given by a sequence of bits (also called *in binary*), each of them given a weight. By convention, the lowest weight is written on the right. The notation used in this report is to add a subscript $\mathbb{B}$ to all constants written out in binary, whereas no subscript means the number is to be read as a decimal number. A few examples:

$$\begin{aligned} 2 &= 10_{\mathbb{B}} = (1 \times 2^1) + (0 \times 2^0) \\ 28 &= 11100_{\mathbb{B}} = (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \end{aligned} \tag{A.1}$$

To extend this notation to be able to use fractional numbers, a decimal point is added after the bit sequence. More bits are then appended with weights $2^{-1}, 2^{-2}$ and so on. For example,

$$2.25 = 10.01_{\mathbb{B}} = (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \tag{A.2}$$

This principle of binary sequences with weights is used in all number representations treated in this report. What follows are two sections on specific number formats used throughout the toolchain: Floating point numbers and fixed point numbers.

A.1.1. Fixed–point numbers

In fixed–point numbers, the length of the bit sequence as well as the number of bits after the decimal point are specified and constant. For example, one might specify that numbers have 8 bits before the decimal point and 8 bits after it. The notation used for number widths in this report is shown below:

$$\mathbb{FI}(i, f) := \text{Numbers with } i \text{ integer bits and } f \text{ fractional bits} \tag{A.3}$$

By convention, numbers with a specified width are written down in full. For example,

$$\begin{aligned} 2 &\in \mathbb{FI}(8, 8) = 00000010.00000000_{\mathbb{B}} \\ 2 &\in \mathbb{FI}(4, 2) = 0010.00_{\mathbb{B}} \end{aligned} \tag{A.4}$$

and so on. This fixed amount of integer and fractional bits results in a fixed, limited range of representable numbers with a fixed resolution: For example, $\mathbb{FI}(8,8)$ has a range from 0 to $(2^8 - 2^{-8})$ with increments of $2^{-8} = 0.0039062$ for unsigned numbers, and the same increments over a -128 to 127 range for signed numbers. Signed numbers are represented using the two's complement, the details of which are irrelevant to the error bound analysis. A hat is used to denote fixed point versions of constants:

$$\hat{A} := \text{Fixed point version of } A \tag{A.5}$$

The fixed point versions of variables are by convention rounded down (toward negative infinity) and hence are at most one LSB off per component.

A.1.2. Floating point numbers

Floating point numbers represent a given number as three separate components: a sign, an exponent and a mantissa. These components have a fixed amount of bits. From these components, the actual number is obtained as follows:

$$\text{Number} = (-1)^{\text{Sign}} \times \text{Mantissa} \times 2^{\text{Exponent}} \tag{A.6}$$

There are several standards like IEEE 754 single–precision binary floating point [2] and almost all software and circuitry complies with them. The details of these standards are not relevant here, what is relevant is the characteristics of floating–point numbers: Since the decimal point can effectively be moved around by changing the exponent, a very wide range of numbers can be represented. While a number cannot be at the same time very large and very precise, floating point numbers have practically speaking almost no range limitations. However, there are various other numerical issues with floating point numbers which are not discussed here.

A.1.3. Conversion from floating to fixed point

While floating point numbers look superior in terms of range and resolution, computations using such numbers are more complex to perform. Modern processors have dedicated floating point units (FPUs) for this purpose. However, if custom circuitry is designed (and fixed point numbers are precise enough), it is often beneficial to use a fixed–point number representation for smaller, faster circuits.

Since fixed–point numbers have only finite resolution, a floating point number can not always be exactly represented by them. The usual convention of converting a number to fixed point is finding the fixed point representation closest to it. For example, if 2.1 were to be converted to $\mathbb{FI}(4,3)$, the result would be $0010.001_{\mathbb{B}} = 2.125$. If more fractional bits are allowed, the number can be represented more exactly, for example in $\mathbb{FI}(4,12)$: $0010.000110011010_{\mathbb{B}} = 2.1001$. The difference between the actual number and the number in the chosen fixed point representation is always less than the weight of the rightmost bit, because that is the difference between two representable numbers. This rightmost bit is also called the *least significant bit* or LSB.

A.2. Error sources in fixed point computations

- **Quantization errors:** Conversion from floating point to fixed point can introduce at most one LSB error per converted number

- **Accumulation errors:** If two terms with errors are added, the errors are added as well. Similarly, errors propagate through products. In the worst case, the errors have the same sign and the total error is the sum of the previous ones.
- **Truncated products:** The product of two numbers in $\mathbb{FI}(i, f)$ is in $\mathbb{FI}(2i, 2f)$ if the product is computed in full precision. However, circuits that perform many operations after each other usually concatenate numbers back to the size of the inputs. This means the uppermost i bits as well as the lowermost f bits are thrown away. This can result in at most one LSB of error every time a product is computed.

Consider the following example for the amplification of errors: Assume a product of a constant coefficient c_1 and a variable input z is to be computed. Also assume that in the fixed point computation, c_1 and z have errors ϵ_c and ϵ_z, respectively. The product then becomes:

$$\hat{c}_1(z + \varepsilon_z) = (c_1 + \epsilon_c)(z + \epsilon_z) = c_1 z + (c_1\epsilon_z + z\epsilon_c + \epsilon_c\epsilon_z) \tag{A.7}$$

This makes the worst–case value of the error introduced in this product dependent on the input itself and not just its error, which is inconvenient. However, if the constant coefficient can be considered error–free, (A.7) becomes

$$\hat{c}_1(z + \varepsilon_z) = c_1(z + \epsilon_z) = c_1 z + c_1\epsilon_z \tag{A.8}$$

in which case the error in the result is simply the input error times the constant coefficient. This means the error in a computation can be expressed independently of the inputs.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Code and Circuit Generation for Efficient Projection Computations on Embedded Platforms

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Merkli	Sandro

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 16.04.2014	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.